EMERALD

Deliverable D2.1

Graph Ontology for Evidence Storage

Editor(s):	Verena Geist, Stefan Schöberl
Responsible Partner:	Software Competence Center Hagenberg GmbH
Status-Version:	Final – v1.0
Date:	31.07.2024
Type:	Report
Distribution level (SEN, PU):	PU

Project Number:	101120688
Project Title:	EMERALD

Title of Deliverable:	D2.1 Graph Ontology for Evidence Storage
Due Date of Delivery to the EC	31.07.2024

Workpackage responsible for the Deliverable:	WP2 - Methodology for Knowledge Extraction
Editor(s):	Verena Geist, Stefan Schöberl (SCCH)
Contributor(s):	Christian Banse, Immanuel Kunz, Angelika Schneider, Florian Wendland (FHG) Franz Deimling (FABA)
Reviewer(s):	Nico Haas (FHG) Cristina Martínez, Juncal Alonso (TECNALIA)
Approved by:	All Partners
Recommended/mandatory readers:	WP2, WP3

Abstract:	EMERALD aims to integrate evidence collected at different levels of the cloud service into a single graph-based structure, the *Certification Graph* (CertGraph). This document describes the development of a uniform schema for storing and linking these heterogenous data. The report mainly involves T2.1, but also inputs of T2.2, T2.3, T2.4, T2.5, and T3.1 are considered.
Keyword List:	Evidence collection, knowledge graph schema, ontology extensions, knowledge integration, combined evidence analysis.
Licensing information:	This work is licensed under Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) http://creativecommons.org/licenses/by-sa/3.0/
Disclaimer	Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. The European Union cannot be held responsible for them.

Document Description

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
v0.1	30.04.2024	First draft version, key information, and TOC.	Verena Geist, Stefan Schöberl (SCCH), Christian Banse, Immanuel Kunz (FHG), Franz Deimling (FABA)
v0.2	21.05.2024	Basics on ontologies and knowledge graphs, architecture, and requirements.	Verena Geist, Stefan Schöberl (SCCH)
v0.3	23.05.2024	Executive summary, introduction, and architecture.	Verena Geist, Stefan Schöberl (SCCH)
v0.4	27.05.2024	Reconciliation of TOC and initial contents by internal reviewer.	Nico Haas (FHG)
v0.5	11.06.2024	Ontology extensions and collaboration.	Verena Geist, Stefan Schöberl (SCCH)
v0.6	26.06.2024	Consolidation of input from task leaders, and illustrative example.	Angelika Schneider, Florian Wendland (FHG), Franz Deimling (FABA), Verena Geist, Stefan Schöberl (SCCH)
v0.7	11.07.2024	QA review by internal reviewer in accordance with the QA process.	Nico Haas (FHG)
v0.8	15.07.2024	Address comments and suggestions from the QA review.	Stefan Schöberl (SCCH)
v1.0	31.07.2024	Submitted to the European Commission.	Cristina Martínez (TECNALIA)

Table of contents

List of tables

List of figures

Terms and abbreviations

AI	Artificial Intelligence
AMOE	Assessment and Management of Organizational Evidence
API	Application Programming Interfaces
AST	Abstract Syntax Tree
BSI	Bundesamt für Sicherheit in der Informationstechnik
CertGraph	Certification Graph
CPG	Cloud Property Graph
CSA or EU CSA	EU Cybersecurity Act
CSP	Cloud Service Provider
CRY	Cryptography and Key Management
DB	Database
DoA	Description of Action
EC	European Commission
GA	Grant Agreement to the project
HTTP	Hypertext Transfer Protocol
IRI	Internationalized Resource Identifier
KPI	Key Performance Indicator
KR	Key Result
ML	Machine Learning
NLP	Natural Language Processing
OWL	Web Ontology Language
PoC	Proof-of-Concept
Protobuf	Protocol Buffers
RDF	Resource Description Framework
RDFS	RDF Schema
SW	Software
SWRL	Semantic Web Rule Language
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

Executive Summary

This deliverable describes the design and development of the *CertGraph Ontology*, a central ontology for storing evidence in a graph-based format. It addresses the key result CERTGRAPH (KR2) of the EMERALD project by outlining a concept for a uniform graph-based model to consolidate all necessary evidence information extracted from a cloud service and to enable the retrieval of combined evidence. In this way, it serves as a common structure that is filled by all evidence extraction tools of WP2.

EMERALD follows a knowledge graph-based approach to provide this unified view of the cloud service under certification at different layers of the service. The schema for storing and linking heterogenous evidence information is developed in WP2, and the model is then implemented in WP3 as a knowledge graph that can be leveraged by assessment tools to measure certification-relevant security metrics.

This document starts by sketching the prerequisites for the transition from the MEDINA ontology to the EMERALD knowledge graph. It introduces a list of requirements for developing the ontology, such as using a formal language and providing a clear conceptualization for the cloud service certification domain. The main part describes the ontology extensions to support the holistic approach to evidence collection, including all levels of the cloud service, ranging from the infrastructure layer (e.g., virtual resources), to the business layer (e.g., policies and procedures), to the implementation and data layer, (e.g., source code and increasingly used artificial intelligence (AI) models). Afterwards, it provides an illustrative example of modelling and combining evidence information for TLS encryption from different sources (e.g., runtime information, policy documents, and source code) as proof-of-concept (PoC). Finally, the document concludes with a short summary and discussion of future work.

The graph-based approach described in this deliverable allows to aggregate individual aspects and fragments of information to a higher-level viewpoint of combined evidence, not previously detectable by a single tool. At the same time, the approach maintains traceability back to different information sources and extraction processes. The uniform schema of evidence information will then be analysed using intelligent algorithms and leveraged to acquire new insights or knowledge in future WP2 deliverables, namely D2.10 “Certification Graph–v1” (M15) and D2.11 “Certification Graph–v1” (M27).

1 Introduction

For automated compliance tools to work, suitable evidence needs to be extracted and linked from different layers of a cloud service. This includes, on the one hand, (i) the virtual infrastructure, such as virtual machines, containers, or storage (based on the *Cloud Property Graph* ontology from the MEDINA project). In addition, also the following sources should be taken into account in EMERALD: (ii) the source code of services, often written in different programming languages, such as Java, Go, or Python); (iii) relevant parts of legal and policy documents, such as requirement or architecture documents; (iv) applied machine learning (ML) models with respect to various criteria, such as robustness, fairness, and explainability; and (v) runtime information, such as configuration or log files. Extraction tools, which will be developed as part of the future deliverables D2.2 - D2.9, will extract and provide evidence from the different layers and sources described above.

The *CertGraph Ontology* with its respective extensions, described in this document, is a central tool to bridge the different layers and sources of extracted information.

1.1 About this Deliverable

This document aims to describe the ontology for modelling evidence information in the cloud service certification domain, i.e., the schema of the EMERALD knowledge graph, consisting of entities, relations, and properties. In addition, a further task is to define guidelines for designing the ontology extensions and domain-specific schema constraints for the underlying data. Thereby, defining additional data properties for enriching data with provenance information (meta data from sources and extraction processes) is essential for providing traceability down to different sources for certification.

The ontology represents the basis for integrating and instantiating the knowledge graph as a repository of target values in the *Evidence Store* (a microservice of *Clouditor*) in Task 3.1. It is also the foundation for analysing the semantic information and context of the heterogeneous evidence information in Task 2.6 to build a higher-level viewpoint of combined evidence, which facilitates querying of certification evidence and provides the basis for the evaluation and assessment of metrics in Task 3.4.

1.2 Document Structure

The document is structured as follows.

In Section 2, we discuss how to extend the MEDINA ontology to the EMERALD knowledge graph. Therefore, we start by presenting the main differences between an ontology and a knowledge graph, then give a short recap of the *Cloud Property Graph* ontology, sketch the planned extensions, and describe how we intend to embed the new ontology in the EMERALD architecture.

Section 3 provides the requirements for designing the ontology.

Section 4 details the ontology extensions for the different cloud service layers, i.e., for extracted evidence from source code, from policy documents, from ML models, and from cloud runtime environments. We further discuss refinements of data properties for combining evidence and supporting traceability, as well as of security features to assess new security metrics.

In Section 5, a seamless example for modelling and combining extracted evidence information from different sources is provided.

Section 6 ends up with the conclusions, including a summary of the main contributions, open challenges, and future work.

The deliverable also includes two appendices:

- *APPENDIX A: Collaborative Ontology Development using Protégé*, which includes remarks for collaborative development of ontology extensions using the Protégé tool.
- *APPENDIX B: Owl2proto – Converting Ontology Files to Protobuf, which* presents Owl2proto, a new tool to convert ontology files to Protocol Buffers (Protobuf) structures.

2 From the MEDINA Ontology to the EMERALD Knowledge Graph

This section explains how to extend the ontology developed in the MEDINA[1] project (for structuring and defining rules and categories of knowledge in the cloud certification domain) to a knowledge graph (for creating a network of that knowledge, detailing how specific facts extracted from cloud services are interrelated).

We start by elaborating the key differences of the semantic technologies, then give a short recap of the *Cloud Property Graph* Ontology from MEDINA, and shortly present the planned extensions in EMERALD. Finally, we discuss how the new ontology can be embedded in the EMERALD architecture.

2.1 Differences between an Ontology and a Knowledge Graph

Ontologies and knowledge graphs are both common components in the field of semantic technologies, knowledge management, and artificial intelligence, but they serve different purposes and are structured differently.

Table 1 highlights the main differences between an ontology and a knowledge graph regarding their definition, main purpose, structure, and use cases.

Table 1. Ontology vs. knowledge graph

	Ontology	**Knowledge Graph**
Definition	Formal representation of a set of concepts and their relationships within a domain.	Graphical representation of real-world entities and their interrelations, typically stored in a graph database.
Main Purpose	Enable knowledge sharing and reuse through structured domain knowledge; reasoning about the entities within the domain.	Integrating information from diverse data sources and querying of facts; effective handling of complex, interconnected information.
Structure	Highly structured, including classes (concepts), instances, attributes (properties), and relationships.	Schema can be described by ontologies, but knowledge graphs are more focused on the instance level.
Use Cases	Semantic web, schemas for knowledge graphs, data integration, NLP, reasoning.	Search enhancements, business intelligence, AI tasks like question answering, semantic search.

We can summarize the **key differences** as follows:

- **Abstraction level**: Ontologies define and categorize the types of concepts and relationships that can exist in a domain (a higher level of abstraction), while knowledge graphs focus on specific instances of those concepts and relationships.
- **Representation**: Ontologies are usually created using formal languages that support complex expressions and logical inferences, like the Web Ontology Language (OWL), including constraints, class hierarchies, and more. Knowledge graphs are primarily represented in graph databases, emphasizing the connections and relationships between entities.
- **Purpose and usage**: Ontologies provide a framework for knowledge representation and reasoning, offering a shared vocabulary for a domain. Knowledge graphs, on the other

[1] https://medina-project.eu/

hand, are focused on connecting real-world entities to form a navigable and query-able graph of facts and to enhance information retrieval.

These differences revealed the benefits of knowledge graphs over ontologies, leading to the decision to make the transition.

2.2 Recap: *Cloud Property Graph* Ontology

The *Cloud Property Graph* [1] ontology from MEDINA proposes a vendor-independent ontology of cloud resources and related security features. The main purpose of this ontology was to harmonize evidence gathering and assessment. Security controls defined in different certifications or catalogues can be assigned to ontological concepts and those ontological types can be further used in metric definitions. Therefore, the ontology defines a vocabulary for mapping between the properties that shall be measured and the respective gathering of adequate evidence.

The ontology consists of three essential taxonomies (i.e., for cloud resources, functionality, and security features) and defines relationships between them (e.g., *offers* to describe which cloud resource generally offers which security feature). The nodes along the whole hierarchy can have (and inherit) relationships to other taxonomies (see Figure 1).

***Cloud Resource* taxonomy:**

- Classifies cloud resources across all major cloud providers and architectures, like Microsoft Azure, Amazon Web Services, Google Cloud Platform, and OpenStack.
- Is ordered by cloud service categories according to functional purposes (e.g., *Compute*, *Networking*, etc.).
- Goal: Representing a generic cloud system (superset of several cloud systems).

***Security Feature* taxonomy:**

- Classifies security properties that can be used in a cloud service.

Functionality* taxonomy*:

- Includes additional utility entities to cloud resources and security features, such as data flows in hypertext transfer protocol (HTTP), reading from and writing to a database (DB), etc.

Note: The relationships in the *Cloud Property Graph* ontology should be refined in EMERALD. That is, e.g., *offers* could mean both, *must have* or *can offer*, thus it is not clear which properties are mandatory and which are optional. Furthermore, *hasMultiple* or *offersMultiple* include mapping information for later code generation, which should be avoided.

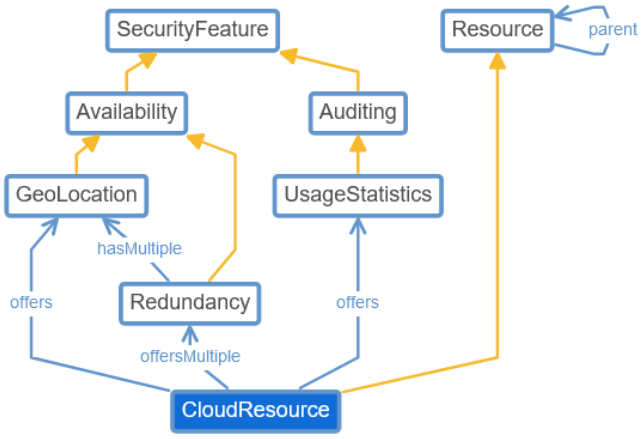

Figure 1. Excerpt of the Cloud Property Graph Ontology showing different relationships – between entities in blue and inheritance in yellow

The design of the ontology is based on the *Cloud Property Graph* (CPG) applications, application programming interfaces (APIs) of cloud providers, and deployment scripts. *Codyze*[2] uses classes implemented in Kotlin, and Java is used to map between classes used in *Codyze* and elements of the *Cloud Property Graph* ontology. For example, *FunctionDeclaration* is such a class and is mapped to *HTTPEndpoint* (see Figure 2). A similar approach will be used for mapping Java classes of the abstract syntax tree (AST) from *eknows*[3] to security controls that are addressed by metrics.

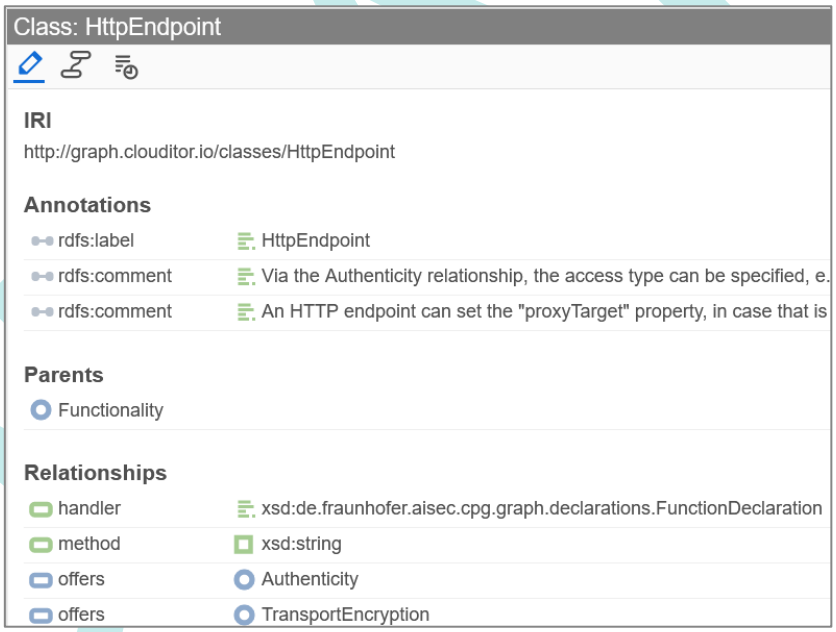

Figure 2. Mapping the CPG to the Code Property Graph Ontology

[2] https://www.codyze.io/
[3] https://www.scch.at/software-science/projekte/detail/eKnows

2.3 Overview of Planned Extensions

We take up the key idea of the *Cloud Property Graph* Ontology from MEDINA, which has the major advantage that metrics (or rules) can be defined for abstract resource types and/or security features, while extractor tools can agnostically gather evidence for these abstract concepts as well.

However, this ontology will only be one part of the EMERALD knowledge graph. We will extent the existing work by adding:

- A ***Source Code*** **taxonomy** to categorize and organize code elements based on their characteristics, functionalities, and security aspects.
- An ***Organizational*** **taxonomy** to categorize and organize textual information from policy documents.
- An ***Artificial Intelligence (AI)*** **taxonomy** to categorize and organize information extracted from ML models based on certain criteria.
- Additional *properties* to extend evidence gathering of cloud resource configurations and to enhance evidence with application-specific runtime information, e.g., from log files.
- Additional *security feature properties* based on new metrics and pilot requirements.

2.4 Embedding the new Ontology in the EMERALD Architecture

This section describes how the *CertGraph Ontology* interacts with (selected) EMERALD components on a conceptual level. Figure 3 shows the current status of the EMERALD component diagram. In EMERALD, a component is any part of the EMERALD ecosystem that has a specific functionality, i.e., it can be considered as a separate entity with respect to other components.

Accordingly, the following terms have been defined:

> The ***CertGraph*** **Ontology** is a (formal) model but not considered as a component itself within the overall EMERALD architecture. It defines the structure and data model (i.e., the schema) of the knowledge graph using abstract types of the cloud certification domain.

> The ***Evidence Store*** implements the model as a component, see deliverable D3.1 [2]. This knowledge graph is based on the schema represented by the *CertGraph Ontology* and primarily focuses on instances (i.e., concrete evidence extracted by the WP2 extraction tools). The *Evidence Store* can either be deployed as a standalone component.

> The *CertGraph Ontology* and the *Evidence Store* form the **Certification Graph** (KR2 CERTGRAPH). It serves as a common graph-based structure that is filled by all evidence extraction tools of WP2.

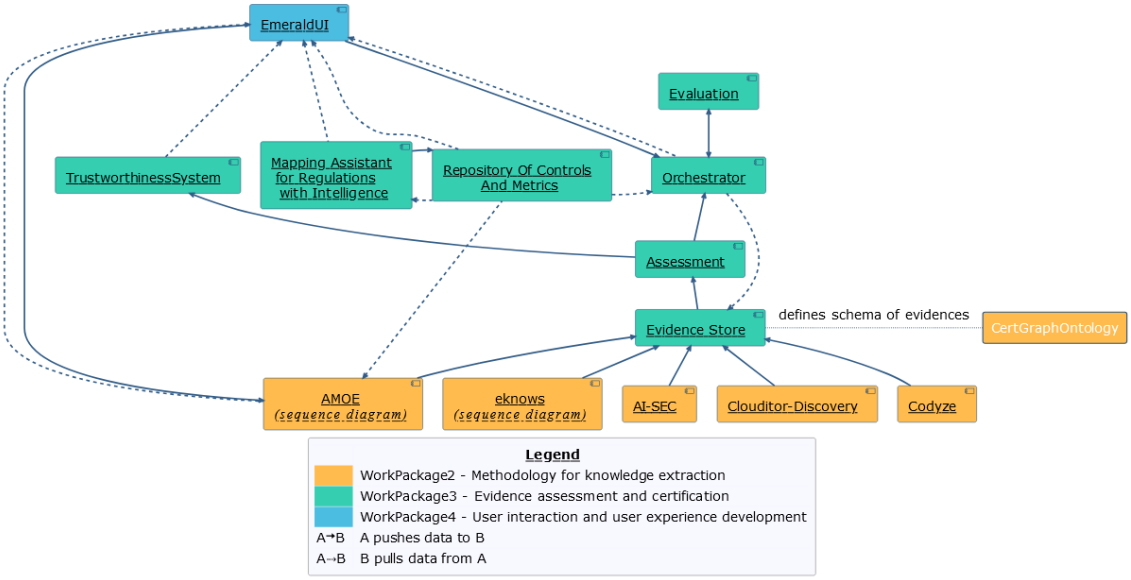

Figure 3. Excerpt of the EMERALD component diagram [3]

In MEDINA, some extraction tools implemented their own assessment. This causes more maintenance effort if requirements change over time. Thus, centralizing assessment is one of the goals in EMERALD. This is done by delivering exclusively (or as far as possible) raw evidence to the *Evidence Store*. All WP2 extraction components (i.e., AMOE, *Codyze*, *eknows*, *Clouditor-Discovery*, and AI-SEC), which extract knowledge from the various layers of a cloud service (i.e., policy documents, source code, cloud interfaces, ML models, etc.), provide (part of) evidence (e.g., for transport encryption), which is then mapped to the EMERALD evidence format using the terms described in the *CertGraph Ontology* (see Figure 4). This evidence information is stored in the *Evidence Store* following the defined schema and is used to assess the metrics defined in the *Repository of Controls And Metrics*.

Please note that metrics are not part of the ontology (as this was the case in MEDINA).

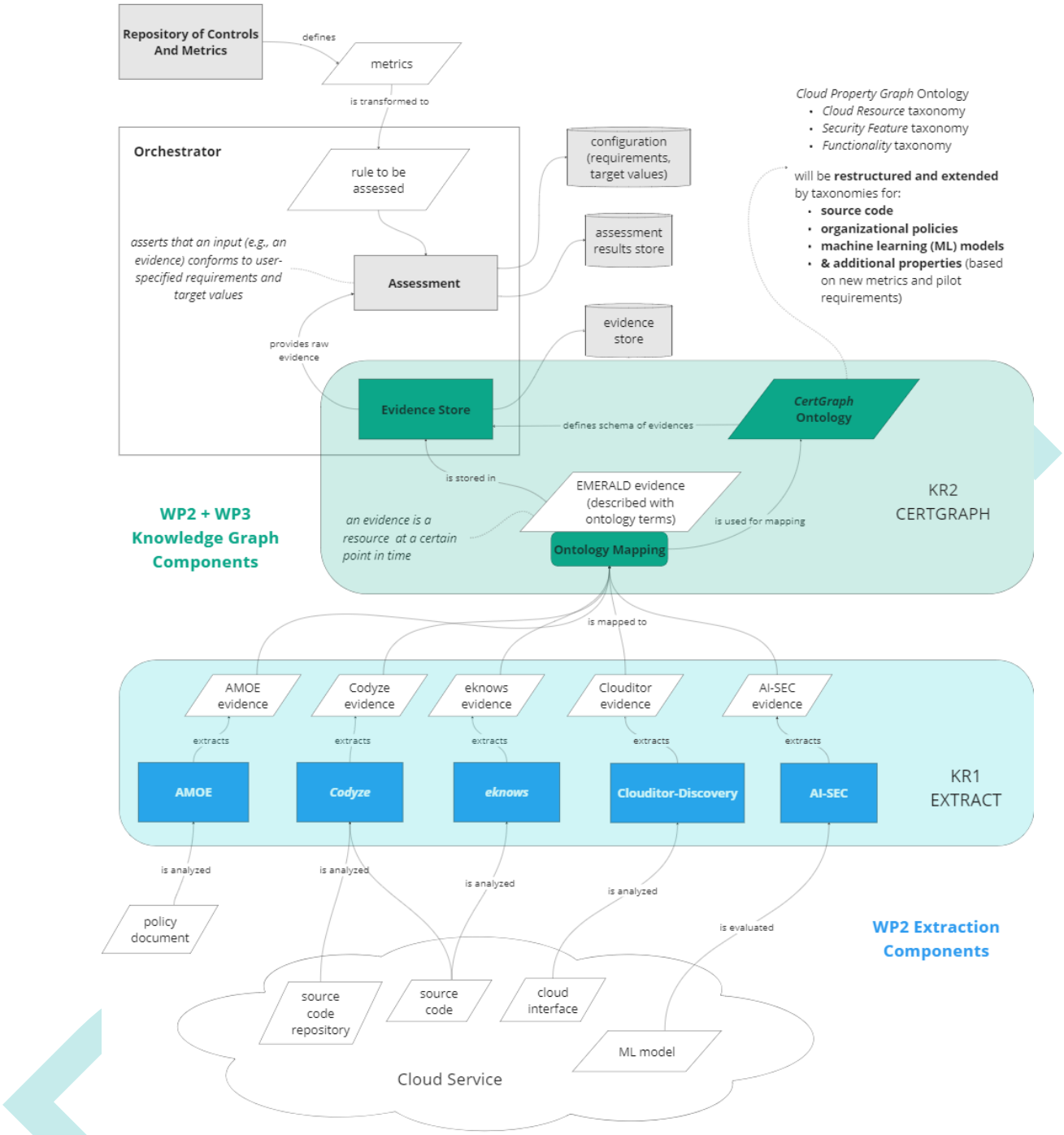

Figure 4. Overview of how the CertGraph Ontology logically interacts with (selected) EMERALD components

3 Requirements for Designing the Ontology

As ontologies are formal representations of knowledge with a rich set of concepts within a domain and the relationships between those concepts, they are used to reason about the objects within that domain and to describe how they are related. The following requirements are essential to enable sophisticated knowledge management, retrieval, and reasoning capabilities:

- **Formal language**. The ontology should be defined using a formal language that allows for the expression of concepts, relationships, instances, and axioms. Examples of ontology languages include the Web Ontology Language (OWL)[4], Resource Description Framework (RDF)[5], and RDF Schema (RDFS)[6].
- **Clear conceptualization**. The ontology should provide a clear and comprehensive conceptualization of the domain it represents, including the definition of classes (or concepts), properties (attributes or relationships), and instances (individual examples of classes). Concepts should be consistently defined (e.g., *Database* was divided into *Storage* and *Service*, whereas *Backup* was not) and relationship should be properly refined (e.g., *offer* should be refined into *specifies* (for policy documents) and *implements* (for source code)). Information, which is needed for code generation from the ontology (e.g., the *Cloud Property Graph* ontology used the *has* and *hasMultiple* properties to model *to-one* or *to-many* relationships and code generators could generate appropriate code to represent them) should not be included in the ontology. Instead, the domain should be the focus and the ontology should reflect it in a meaningful way.
- **Hierarchical structure of concepts**. The ontology should support the creation of a hierarchical structure of concepts, allowing for subclass relationships and the organization of concepts into a taxonomy.
- **Reasoning and consistency checking**. The ontology should be compatible with inference engines and allow for the definition of logical rules that enable automated reasoning about the concepts and their relationships. In addition, tools and methods should be available for checking the consistency and validity of the ontology, ensuring that there are no logical contradictions within the defined concepts and relationships.
- **Interoperability and extensibility.** The ontology should be developed in a way that ensures interoperability with other ontologies, facilitating data exchange and integration across different layers of a cloud service. The ontology should be accessible to both humans and machines, with clear naming conventions and identifiers, allowing parts of the ontology to be reused in different namespaces and contexts. It should also be extensible regarding novel security schemas and standards, in case they require additional evidence, which has to be modelled as an extension.
- **Documentation and annotation**. Comprehensive documentation and annotation of the ontology should be available, including descriptions of the purpose, scope, and structure of the ontology, as well as the meaning of all concepts and relationships.
- **Versioning**. There should be a clear strategy for handling the releases of the ontology (e.g., annually, quarterly, or on demand) and how changes and new versions are announced.

[4] https://www.w3.org/OWL/
[5] https://www.w3.org/RDF/
[6] https://www.w3.org/TR/rdf12-schema/

4 Core Ontology and Extensions

In this section, we discuss how to integrate evidence extracted from the multiple cloud service layers (i.e., infrastructure, platform, and software), including policy documents and runtime information, into a single graph-based structure (KR2-CERTGRAPH). Furthermore, evidence information for the security evaluation of AI models (KR5-AIPOC) will be included. Based on the general idea of (harmonized) security metrics, we allow different evidence collection tools to gather different layers of evidence for the same metric, enhancing reuse of evidence collected, and providing answers to assess the metrics.

Therefore, we plan different extensions of the overall ontology (see Figure 5), which together represent the unified source of types in the cloud service certification domain. Content is typically represented by a set of triples (subject, predicate, object), where the predicate describes the relation between the subject and object entity [4]. Knowledge graphs [5] are a well-established method for managing complex multi-relational relationships based on the provided schema. This way, fine-grained entities from different heterogenous sources (structured data, semi-structured data, free text) can be extracted and linked step by step [6]. Of importance are the ideas of the extended DIKW hierarchy (data, information, knowledge, and wisdom) [7], where each concept is related to the previous concept, forming a chain of increasing interconnectedness and evaluated human understanding [8].

The growing availability of large amounts of evidence data, which evolves over time and is continuously extracted for cloud service certification, requires to annotate the graph with temporal information, such as timestamps [9]. Explicitly capturing temporal dependencies in addition to structural properties increases the traceability of facts back to extraction tools and transparency in processes and procedures required to run cloud services.

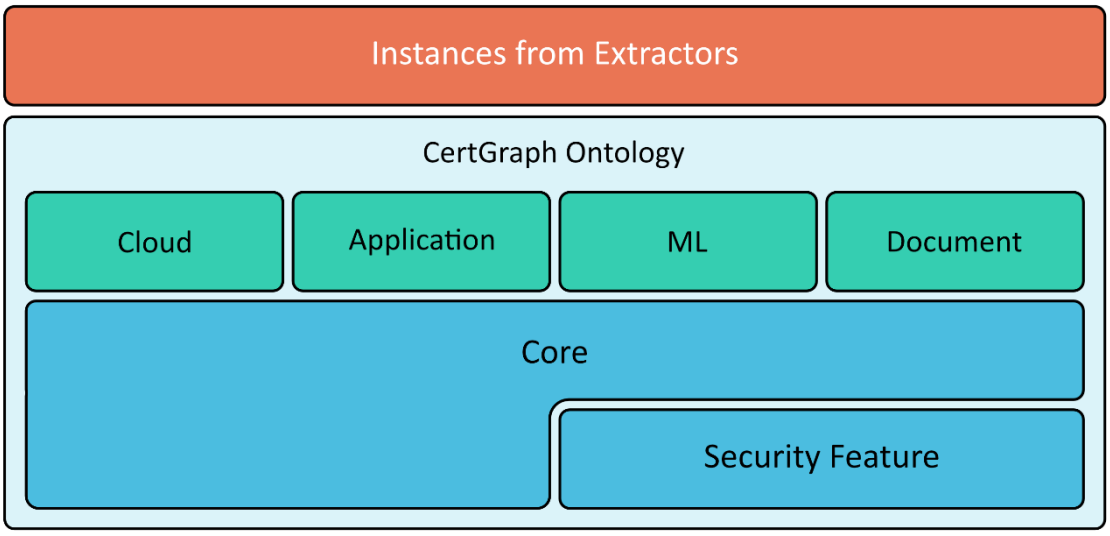

Figure 5. Modular design of the CertGraph Ontology with the extensions in green

As shown in Figure 5, the *CertGraph Ontology* consists of multiple sub-ontologies and extensions, which cover individual aspects. The *Core* ontology, together with the *Security Feature* ontology, builds the foundation of the ontology and contains base classes and properties. Specifically, *Security Feature* models different security related concepts. The extensions are built on top of this foundation and each extension models the evidence gathered by a different type of extractor (see Table 2). The collected evidence from the extractors is represented as instances within a separate part that, in turn, is built upon the ontology and implemented in the *Evidence Store*.

Table 2. Ontology extensions and their dedicated extractors

Extension	Extractor
Cloud	*Clouditor-Discovery*
Application	*eknows* and *Codyze*
ML	*AI-SEC*
Document	*AMOE*

Each sub-ontology and extension has its own namespaces (see Table 3). This allows for interoperability and a flexible extension of the ontology beyond the aspects considered within EMERALD.

Table 3. Sub-ontologies and their namespaces

Sub-ontology	Namespace
Core	https://ontology.emerald-he.eu/core
Security Feature	https://ontology.emerald-he.eu/core/securityfeature
Cloud	https://ontology.emerald-he.eu/resources/cloud
Application	https://ontology.emerald-he.eu/resources/application
ML	https://ontology.emerald-he.eu/resources/ml
Document	https://ontology.emerald-he.eu/resources/documents
Evidence from <tool>	https://ontology.emerald-he.eu/evidence/<tool>

At the time of writing, each sub-ontology is not modelled in detail. In the following sections we outline the main concepts which will be included in each sub-ontology. Section 5 shows an example of the planned content and includes a diagram, which *zooms in* and just shows the relevant parts of the example. The planned collaboration for creating the *CertGraph Ontology* and its sub-ontologies and extensions is described in *APPENDIX A: Collaborative Ontology Development using Protégé*.

A newly developed tool, called *Owl2proto*[7] allows to convert an ontology to Protobuf structures. With this tool, we are able to automatically generate proto files from the ontology files and use them directly in different supported programming languages. Previously, we had to create and update all ontology objects for each programming language manually. The Owl2proto tool is described in *APPENDIX B: Owl2proto – Converting Ontology Files to Protobuf*.

4.1 Core with Security Feature

4.1.1 *Core* – A Base Ontology

Core is an ontology that constitutes the core of the overall *CertGraph Ontology*, where different extensions can be imported (depending on the actual certification use case).

- Root node: *Resource* (an abstract concept being the anchor of all imported extensions)
- Content:
  - Contains the (mandatory) *Security Feature* Ontology.
  - Specifies refinements for combining evidence and supporting traceability, such as required properties for the extraction source (i.e., which extractor performed the evidence extraction, in which version of the tool, etc.), timestamps, etc.

[7] https://github.com/oxisto/owl2proto

4.1.2 *Security Feature* – Containing Data Properties for Security Metrics

Security Feature is an ontology where all extracted evidence information from the different layers of a cloud service will be linked to specific data properties of security features to assess security metrics / requirements from the pilots.

- Root node: *SecurityFeature*
- Content:
  - Based on the existing taxonomy from the *MEDINA Ontology*, additional security features and properties will be added based on new metrics and pilot requirements.
  - Additional security features and properties for existing extractors of MEDINA will be added (e.g., regarding AMOE and *Codyze*).
  - The *Functionality* taxonomy of the *MEDINA Ontology*, which comprises a collection of general concepts, will also be included.

4.2 Ontology Extensions

This section describes the key information of the ontology extensions for different cloud service layers. Note that a mixed approach will be followed when creating the extensions: We will start "top-down" by modelling the hierarchical structure of concepts and relationships in a rather generic way – independent of security metrics and requirements. These taxonomies will then be refined and linked to data properties of security features "bottom-up", i.e., depending on what we need, what we want to measure, and what we actually get from the given artifacts in the pilots. The more concrete the links, the more added value the ontology will provide. The starting point for concrete metrics will most likely be official security schemes, such as BSI AIC4[8] or C5[9].

4.2.1 *Application* – A Taxonomy for Source Code

Application is a source code taxonomy to categorize and organize code elements based on their characteristics, functionalities, and security aspects in software systems (with regard to evidence extracted in Task 2.2).

- Root node: *Application*
- Extractor(s)[10]: *Codyze* / *eknows*
- Content:
  - Sub-concepts of an application are described in more detail, represented as a superset of several languages, and linked to (additional) security features. It may include:
    - Source code file with line information,
    - security-related APIs,
    - business rules,
    - security guidelines,
    - project configuration and repository (meta)data.
  - *Framework* taxonomy from the *MEDINA Ontology*.
  - Mapping to *eknows* classes (AST) will be defined, analogous to mapping to *Codyze* classes (CPG).

[8] https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/CloudComputing/AIC4/AI-Cloud-Service-Compliance-Criteria-Catalogue_AIC4.html

[9] https://www.bsi.bund.de/dok/7685384

[10] Please note that the two extractors should complement each other. For example, one security control can be better covered by Codyze, another by eknows. There are no plans to combine the two tools.

- Extracted evidence for security requirements / features may include cryptography, secure storage, dependency management, transport encryption, authentication, authorization, logging, input validation and best practices for (secure) coding.

4.2.2 *Document* – A Taxonomy for Policy Documents

Document is an organizational taxonomy to categorize and organize textual information from policy documents (with regard to evidence extracted in Task 2.3).

- Root node: *Document*
- Extractor: *AMOE*
- Content:
  - Differentiation between different kinds of policy documents as high-level nodes (e.g., architecture, requirements, etc.) – the better classified, the more precisely linkable to security features described in policy documents.
  - Might also contain information about author, page number, responsible person, link to the document, etc.
  - Extracted evidence for security requirements/features may include encryption (transport, browser, password, API, etc.), certificate (key length, validity period, etc.), authentication (login, password, etc.), security incident, malware protection, data access, and backup.

4.2.3 *ML* – A Taxonomy for AI/ML Models

ML is a taxonomy to categorize and organize information extracted from AI/ML models based on certain criteria (with regard to evidence extracted in Task 2.4).

- Root node: *ML*
- Extractor: *AI-SEC*
- Content:
  - Differentiation between different kinds of ML models (e.g., for images, text, etc.) and different kinds of tasks (e.g., classification, prediction etc.) as high-level nodes.
  - Different kind of information denoting relevant criteria (e.g., fairness, robustness, privacy-preserving, etc.) which will be linked to specific security features.
  - Types for extracted evidence for security features are not defined yet (types may contain strings, vectors, etc.).

4.2.4 *Cloud* – A Taxonomy for Cloud Resources including Runtime Information

Cloud is a taxonomy (based on the contents of MEDINA) with additional properties to extend evidence gathering of cloud resource configurations and to enhance evidence with application-specific runtime information, e.g., from log files (with regard to evidence extracted in Task 2.5).

- Root node: *Cloud Resource*
- Extractor: *Clouditor-Discovery*

- Content:
  - Based on the existing taxonomy from the MEDINA Ontology. The taxonomy is divided into different resource categories (e.g., *Compute*, *Storage*, *Networking*) which contain the corresponding cloud resources (e.g., *Virtual Machine*, *Block Storage*, *Network Interface*). For example, the resource category *Compute* contains the underlying *Cloud* resources *Container*, *Function* and *Virtual Machine*.
  - Additional and refined links to the security features.
  - Extracted evidence for security requirements/features may include *encryption /transport*, *encryption in use*, *at rest encryption*, etc.), *logging* (*enabled, retention period*, etc.), *authentication* (*password*, *multi factor*, *token based*, etc.), *access restriction* (*restricted ports*, *firewall*, etc.), *backup* (*transport encryption*, *location*, *retention period*, etc.), *redundancy*.

5 Illustrative Example – Modelling and Combining Evidence Information for "TLS Version"

This section presents an illustrative example for modelling and combining extracted evidence information from different sources. At the time of writing this document, the final choice of the used security schema(s) and security controls/metrics in EMERALD has not yet been made. We, therefore, start an initial proof-of-concept with a meaningful security property from a software perspective taken from BSI C5, e.g., encryption of data for transmission (BSI C5: CRY-02).

The key idea is to represent security-related parts of the source code of a cloud service in a graph structure and provide additional context through the discovery of the cloud resources the service is running on and related policy documents, e.g., regarding the used TLS version. Bridging the world of static code analysis and extraction of a cloud service's runtime information allows to combine evidence at a higher level of knowledge and also enables a comparison of what is described in policy documents.

5.1 Overview of Used Concepts

The focus of this example is on illustrating the big picture and interconnectivity between sub-ontologies (see Figure 6) and not on details within a certain ontology. Furthermore, OWL will be used as formal language to describe the ontologies. In the diagram, classes are visualized as rectangles and instances as hexagons. Open-headed arrows with a filled line (⇾) represent "subclass of" relations, which connect subclasses to their parent class, and open headed arrows with a dashed line (⇢) represent "instance of" relations, which connect instances to their class. Simple arrows (→) represent data and object properties. These arrows are used between classes to define the schema, as well as between instances in their materialized form.

As described in Section 4, the two ontologies *Core* and *Security Feature* form the basis for the *CertGraph Ontology*. The *Core* ontology defines the metamodel for EMERALD evidence and uses the concepts defined in the *Security Feature* ontology.

The *Security Feature* ontology contains a variety of security features and data properties, which are based on the same-named taxonomy from MEDINA. To keep things simple, only a single feature (*TransportEncryption* class) is showcased in this example and the hierarchy has also been simplified to two levels. The base class of this hierarchy is *SecurityFeature*. Also, for simplicity reasons, just one data property *version* is defined to store the TLS version.

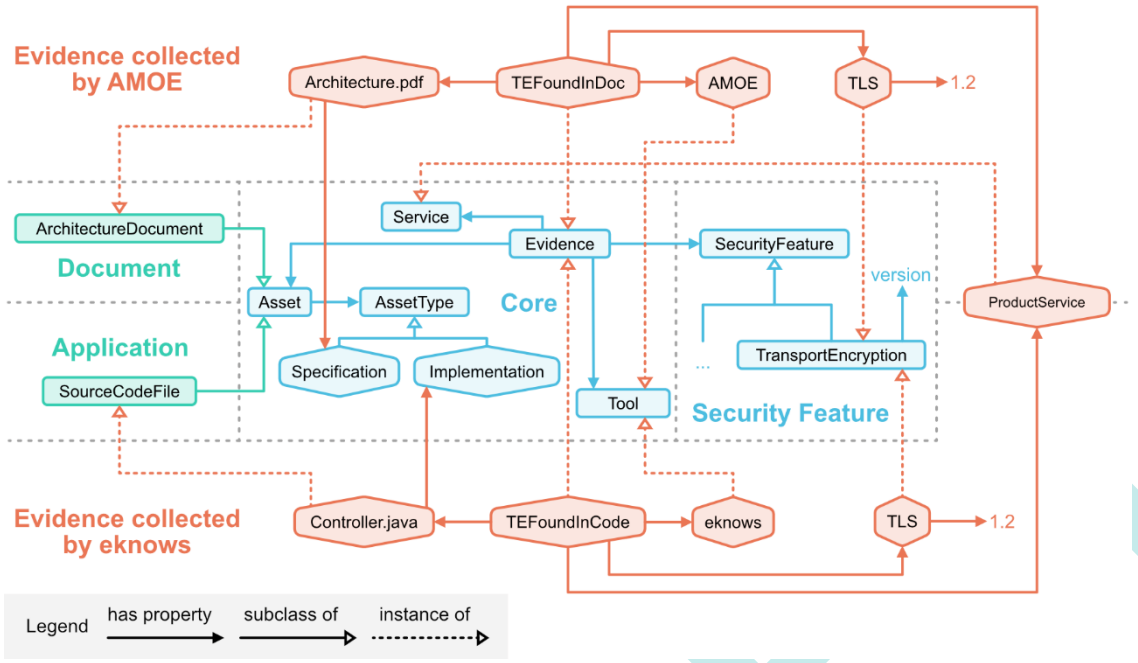

Figure 6. Classes and instances for the TLS example

The *Core* ontology contains classes, which define the model of evidence and their connections to other related information fragments. At the time of writing, our proof-of-concept consists of the following classes:

- ***Evidence***, which is the central class and instances of it represent detected security evidence. Each evidence has connections to *Security Feature, Service, Asset,* and *Tool*.
- ***Asset***, which represents the source of a piece of evidence and should store relevant metadata for the location, as it best fits the asset. Each *Asset* has a connection to an *AssetType*.
- ***AssetType***, which classifies the role of assets within the system. *AssetType* is modelled as an enumeration type in ontology terms. For this, a class is needed, and an instance is created for each possible variant. Currently, we distinguish between these two possible variants:
  - The first variant, ***Specification***, is used for evidence found in assets, which describe, how the system *should* behave. The main application for this variant is in human-readable documents which are not automatically processed for compilation, for example, architecture descriptions or policy documents.
  - The second variant, ***Implementation***, is used for evidence found in assets, which describe, how the system *actually* behaves. This variant is mainly used for evidence found in machine-processed assets, for example, source code, configuration files, or runtime information.
- ***Service***, which ties the evidence to a certain service.
- ***Tool***, which represents the extractor component that has collected the evidence.

In particular, the connection to *Service* should enable the fusion of evidence from multiple sources. This requires a unique identifier for each service, which will be used as URI for the service instance.

Extensions are built on top of the *Core* and *Security Feature* ontologies. In this example, we used the *Document* and *Application* extensions and limit the scope to just one class per extension. As

previously described, the classes in the extensions should model their respective domains. The following two classes are used in the example:

- ***ArchitectureDocument***, which represents a human-readable textual document for software architecture, and
- ***SourceCodeFile***, which represents a source code file which is compiled for a given service and is stored in a repository.

5.2 Adding Instances for Extracted Evidence

Gathered evidence from the system are modelled as OWL instances. In the example in Figure 6, evidence extracted by *eknows* will be used. The found evidence is represented as the instance *TEFoundInCode* in the diagram and has connections to other instances. Please note that "TransportEncryption" is abbreviated as "TE" in this example for better readability in the diagram.

TEFoundInCode connects to the following instances:

- *Controller.java*, an instance of the *SourceCodeFile* class,
- *TLS*, an instance of *TransportEncryption* with the version property set to "1.2",
- *eknows*, an instance of *Tool*, to represent the extraction component,
- *Implementation*, an instance provided by *Core*, to indicate, that the evidence has been found as actual behaviour, and
- *ProductService*, an instance of *Service*, to represent the service, to which the evidence belongs to.

Note: Evidence from other extraction components must link to the same service instance. In OWL, two instances are considered as *the same* if they are identified by the same URI. This enables knowledge fusion later on for the assessment, and therefore one must ensure that the same URI is used to identify a given service across all extraction components. This challenge was tackled in MEDINA by creating an ID for each cloud service, and the same strategy will be applied here.

5.3 Challenges and future work

Based on this example, the ontology will continuously be extended in the course of the project. Furthermore, some design decisions are not final and are still discussed. This includes, but is not limited to, connections between classes in general.

Another open discussion is the structure of *Evidence* and *SecurityFeature*. Currently they are modelled as two separate classes and it is being evaluated whether it would be more sensible and simpler to merge these two classes into one.

At the time of writing, the implications of each decision cannot yet be estimated entirely, and the structure of the ontology will continue to evolve. The results will be reported in the upcoming deliverables D2.10 "Certification Graph–v1" (M15) and D2.11 "Certification Graph–v1" (M27).

6 Conclusion

The aim of this deliverable is to document how to establish a unified view of the cloud service under certification by extracting and enriching knowledge on different layers of the service and providing a suitable schema for storing this evidence. Therefore, we describe a seamless approach consisting of data acquisition, knowledge extraction, and knowledge fusion to build the *CertGraph Ontology* and its sub-ontologies and extensions to consolidate all necessary information of the service as the basis for implementing the knowledge graph in WP3. An illustrative example on how to model and combine evidence extracted by different EMERALD extractors is provided as an initial proof-of-concept (PoC).

The main **contributions** of the ontology for evidence storage include the provision of:

1 A concept for *generic models* to map security aspects.
Firstly, generic models (i.e., sub-ontologies and extensions) provide support for evidence extraction from different sources (e.g., infrastructure, source code, and documents) of the service and a schema for storing the heterogeneous evidence information. Following a knowledge graph-based approach, these models allow to view partial evidence from different perspectives.

2 A clean basis *multi-evidence fusion*.
Secondly, linking of heterogeneous evidence allows to aggregate individual aspects and fragments of information to a higher-level of combined evidence, while providing support for traceability to information sources and extraction processes. This way, the graph serves as a common structure filled by all evidence extraction tools that can be leveraged by the assessment tools in WP3 to measure security metrics.

3 Enhanced *quality of measurement* and *possibility of comparison*.
Thirdly, assessing (partial) evidence from different sources allows a qualitative statement about the accuracy of measured results for auditors and, furthermore, enables the comparison between specification (e.g., in policy documents) and implementation (e.g., in source code) of security features.

4 Representation of *evidence about AI model security*.
Lastly, by integrating also evidence extracted by novel methods for the security assessment of AI models, EMERALD will also be able to certify cloud-based AI systems and transfer the innovation results to upcoming AI certification schemes.

There are also some open **challenges** that we will address in **future work:**

- *Security controls and metrics* to be addressed are not yet defined.
It is not clear yet what evidence should be included in the ontology extensions in detail. This heavily depends on the security controls and metrics to be addressed in the pilots, which are not known by the time of writing. To mitigate this challenge, we use a mixed approach (“top-down” followed by “bottom-up”) for developing the ontology extensions and mapping them to required security feature properties later. In addition, a workshop is planned together with responsible technical persons of the pilot partners to discuss functional requirements of the technical extraction components.

- Details on *fusion of partial evidence* and implementation options need to be clarified. Further details on ensuring unique identifiers, temporal constraints, and semantic refinements need to be investigated. In Task 2.6, we will research on creating a graph abstraction layer to facilitate querying of evidence and apply graph-based analysis to generate new insights or knowledge. We will further investigate reasoning techniques

and explore to which extent they can be reused in the knowledge graph implementation.

7 References

[1] C. Banse, I. Kunz, A. Schneider and K. Weiss, "Cloud Property Graph: Connecting Cloud Security Assessments with Static Code Analysis," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, Chicago, 2021.

[2] EMERALD Consortium, "D3.1 Evidence assessment and Certification - Concepts - v1," 2024.

[3] EMERALD Consortium, "D1.1 Data modelling and interaction mechanisms-v1," 2024.

[4] X. Zhao, A. Li, R. Jiang and Y. Song, "Multi-source knowledge fusion: a survey," vol. World Wide Web, no. 23(4), pp. 2567-2592, 2020.

[5] H. Paulheim, "Knowledge graph refinement: A survey of approaches and evaluation methods," vol. Semantic web, no. 8(3), pp. 489-508, 2017.

[6] J. Pujara, H. Miao, L. Getoor and W. Cohen, "Knowledge graph identification," in *International Semantic Web Conference, pp. 542-557*, 2013.

[7] M. Frické, "The knowledge pyramid: the DIKW hierarchy," *Knowledge Organization,* vol. 46(1), pp. 33-46, 2019.

[8] C. Zins, "Conceptual approaches for defining data, information, and knowledge," *Journal of the American society for information science and technology,* vol. 58(4), pp. 479-493, 2007.

[9] R. Trivedi, H. Dai, Y. Wang and L. Song, "Know-evolve: Deep temporal reasoning for dynamic knowledge graphs," in *International Conference on Machine Learning, pp. 3462-3471*, 2017.

APPENDIX A: Collaborative Ontology Development using Protégé

In this appendix, we describe the future collaboration between the partners FHG and SCCH to collaboratively develop the *CertGraph Ontology* and its sub-ontologies and extensions. We, therefore, outline the governance and technical aspects of the planned collaboration.

A.1 Governance

To ensure an effective collaboration, the following governance aspects have been defined:

- Repository: A new project called "Ontology" is created in the TECNALIA GitLab under package "Private / Evidence Management / Ontology"[11]
- License: Apache 2 license is required for all sub-ontologies and extensions.
- Output format: The OWL ontology format would be preferable because then only minor changes are required to the previous workflow at FHG using OWL export and *Owl2proto* tool for *Clouditor*.

A.2 Technical Aspects

A.2.1 Restructuring and Extending the Ontology

Restructuring the MEDINA Ontology will be done as outlined in Section 4. In doing so, we will not start with the existing exported format but build the sub-ontologies from scratch together.

For all sub-ontologies and new extensions, customized URIs will be used to avoid interoperability issues when working with multiple namespaces and combing evidence information.

A.2.2 Used Tools: Protégé and Git

In contrast to the MEDINA project, where *WebProtégé*[12] was used for joint development, we have now decided to use the *Protégé*[13] and *Git*[14] tools for EMERALD. This decision is based on the following reasons:

- *WebProtégé* is not able to support linking of concepts using different namespaces, i.e., through different sub-ontologies.
- *WebProtégé* is not able to apply reasoning, which is very useful for fusion of multiple evidence parts.
- The *Comment* feature of *WebProtégé* can be replaced by pull requests in Git.

Protégé (see Figure 7) is a desktop application developed by Stanford university that enables the modelling of ontologies using OWL concepts. Furthermore, it supports a variety of data formats, including RDF/XML, OWL/XML, Turtle and Manchester OWL. Ontologies can also be imported into other ontologies, which supports the splitting of the *CertGraph Ontology* into multiple files for better structuring. *Protégé* also provides a reasoning component. This component can derive new information based on rules (which can be declared as characteristics of properties or can be written in SWRL, for example). Furthermore, the reasoning component can detect inconsistencies in the ontology. Beyond this, *Protégé* offers extensibility via a plugin mechanism.

[11] https://git.code.tecnalia.com/emerald/private/evidence-management/ontology [internal use only - authentication required]
[12] https://webprotege.stanford.edu/
[13] https://protege.stanford.edu/
[14] https://www.git-scm.com/

The collaboration workflow will look as follows: Editing of the ontology will be done in *Protégé* and will be saved as OWL/XML. Changes will be checked into the *Git* repository. The discussion and review of these changes will occur via pull requests on *GitLab*. Finally, the changes will be merged into the main branch.

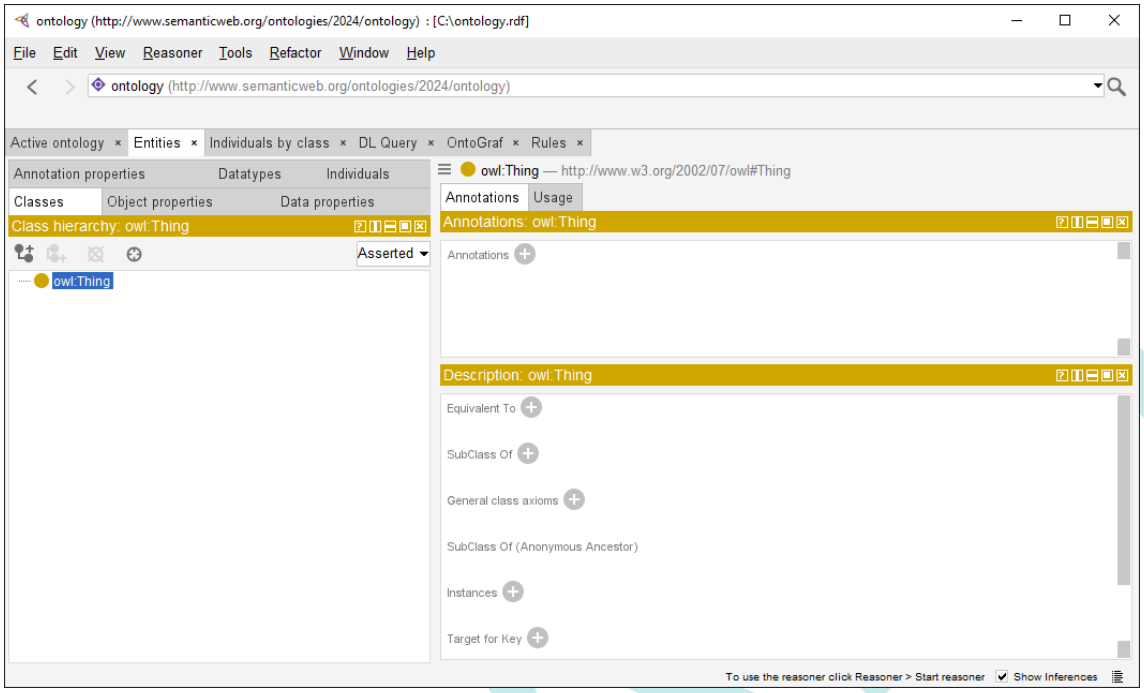

Figure 7. Screenshot of Protégé

APPENDIX B: Owl2proto – Converting Ontology Files to Protobuf

The tool *Owl2proto*[15] converts the modelled ontology to an appropriate *protobuf* schema, which can be directly used in different programming languages.

B.1 Motivation

The *CertGraph Ontology* is developed in *Protégé*[16]. Programmers that want to use this ontology definitions as objects in their programming language have to create these objects manually and update them after the ontology changes. This process involves a lot of work and is prone to errors.

Owl2proto minimizes the manual effort by auto-generating the *protobuf* schema from the ontology definition. The generated *protobuf* files can then be used by the different components and their different specific programming languages.

B.2 Approach

The tool *Owl2proto* auto-generates the *protobuf* schema out of the modelled ontology. Therefore, the ontology must be exported in the OWL format. Currently, this is the only file format that can be processed.

The output format for the protocol buffers is the *.proto* file format. The protocol buffer file data is structured as *Messages* which contain name-value pairs called *fields* which need to be unique per package. Each field contains an assigned field number which must also be unique within a message.

Since ontologies use inheritance and *protobuf* does not support this, we must remove the inheritance from the *protobuf* messages by flattening the hierarchy. An example can be seen in the figures below. Figure 8 shows the important classes and hierarchy for the example resource *VirtualMachine*. The path to the *VirtualMachine* is *Resource* → *CloudResource* → *Compute* → *VirtualMachine*. All classes can have their own properties. Two examples are shown in Figure 9 for the class *CloudResource* and in Figure 10 for the class *VirtualMachine*.

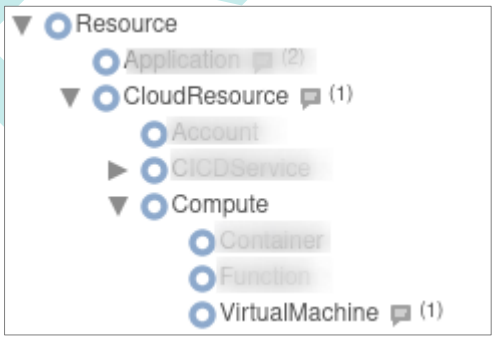

Figure 8. Overview of the ontology hierarchy of the resource VirtualMachine

[15] https://github.com/oxisto/owl2proto
[16] https://protege.stanford.edu/

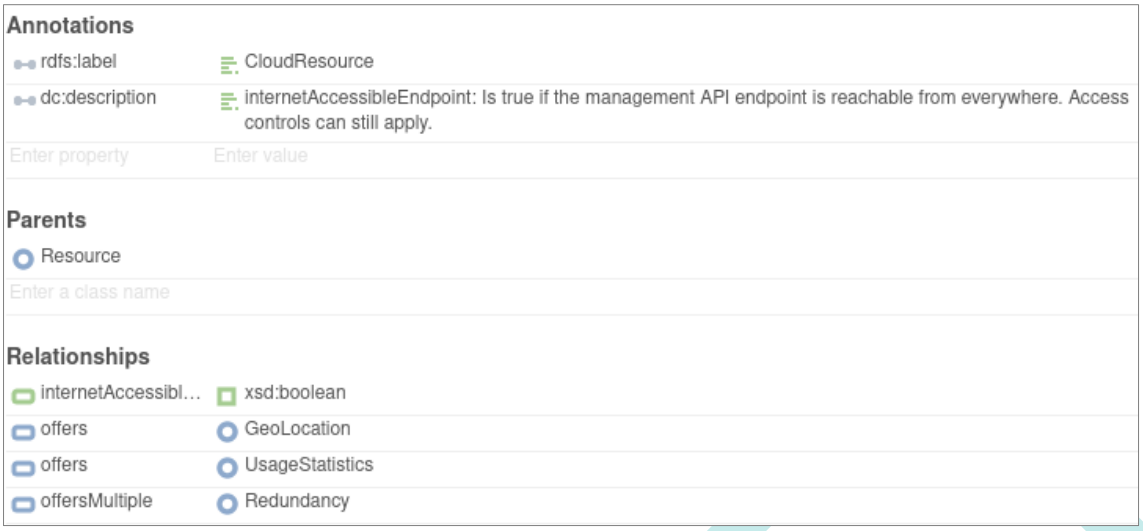

Figure 9. Example for the properties of the resource CloudResource

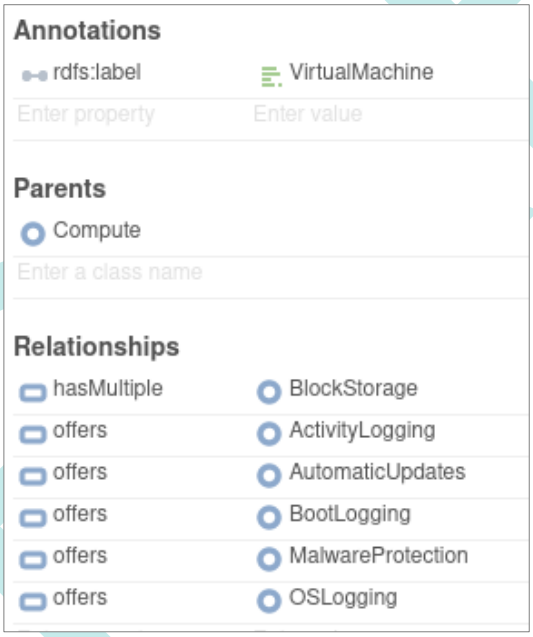

Figure 10. Example for the properties of the resource Virtual Machine

As already mentioned, *protobuf* does not support inheritance, and the automatically generated output must flatten the hierarchy. Figure 11 shows an example. All properties of the path from *Resource* to *VirtualMachine* are added to the *VirtualMachine* protobuf message without any hierarchy information.

```
// VirtualMachine is an entity class in our ontology. It can be instantiated and
contains all of its properties as well of its implemented interfaces.      You, 4
message VirtualMachine {
  option (resource_type_names) = "VirtualMachine";
  option (resource_type_names) = "Compute";
  option (resource_type_names) = "CloudResource";
  option (resource_type_names) = "Resource";

  google.protobuf.Timestamp creation_time = 1;
  string id = 2 [(buf.validate.field).required = true];
  bool internet_accessible_endpoint = 3;
  map<string, string> labels = 4;
  string name = 5 [(buf.validate.field).required = true];
  // The raw field contains the raw information that is used to fill in the
  fields of the ontology.
  string raw = 6;
  ActivityLogging activity_logging = 7;
  AutomaticUpdates automatic_updates = 8;
  repeated string block_storage_ids = 9;
  BootLogging boot_logging = 10;
  EncryptionInUse encryption_in_use = 11;
  GeoLocation geo_location = 12;
  MalwareProtection malware_protection = 13;
  repeated string network_interface_ids = 14;
  OSLogging os_logging = 15;
  repeated Redundancy redundancies = 16;
  optional string parent_id = 17;
  ResourceLogging resource_logging = 18;
  UsageStatistics usage_statistics = 19;
}
```

Figure 11. Example for the auto-generated protobuf message for the VirtualMachine resource

Another specialty is that we use the *oneOf* keyword in *protobuf* messages. This is used for all nodes (classes) that are not leaf-nodes (the lowest classes). This allows to obtain the individual *type* information of the intermediate nodes. Figure 12 shows an example of the *protobuf* message.

```
// Compute is an abstract class in our ontology, it cannot be instantiated but
acts as an "interface".
message Compute {
  oneof type {
    Container container = 1;
    Function function = 2;
    VirtualMachine virtual_machine = 3;
  }
}
```

Figure 12. Example for the auto-generated protobuf message for the intermediate node/resource Compute

B.3 Future Work

We plan to implement the following improvements:

- **Field numbers.** To keep compatibility with previous *protobuf* version, field number must not change. Currently, the tool is not able to deal with that if new properties are added.
- **Several files.** The tool can currently only read in one owl file to generate the corresponding *protobuf* file. Since the *CertGraph Ontology* consists of several import files, it would be desirable if the tool could directly process several files as input.

- **Load Ontology files.** Ontology files are usually available at a specific URL. It would be desirable if only the URL of the root file had to be specified and this and all other imported files would be automatically retrieved and used for the file generation.