Politecnico di Torino

Master Degree in Cybersecurity

Master Thesis

Bill of Materials Integration in OpenC2 for Enhanced Context Discovery

Advisors:
Daniele Bringhenti
Daniele Canavese
Matteo Repetto
Fulvio Valenza

Candidate:
Fabio Lorenzato

Academic Year 2025/2026

Abstract

The rapid evolution of the modern cybersecurity landscape, characterized by high-frequency automated threats, makes a paradigm shift towards orchestrated and interoperable defense mechanisms most important. The Open Command and Control (OpenC2) standard addresses this challenge by providing a unified language for machine-to-machine communication, decoupling high-level security intents from specific implementation technologies. However, while OpenC2 standardizes the execution of commands, effective orchestration relies heavily on a comprehensive and structured visibility of the defended environment. Currently, a standardized framework for context discovery—capable of describing the granular composition and interdependencies of network resources—remains a critical gap in the ecosystem.

This thesis addresses this limitation by proposing the integration of Bill of Materials concepts into the Otupy framework, an OpenC2 implementation in Python. Specifically, it presents the design, implementation, and evaluation of a novel OpenC2 Actuator Profile dedicated to context discovery. By leveraging the CycloneDX standard, which supports full-stack representation of software, hardware, and services, this profile enables the exchange of network resource data in a consistent and standardized format. Diffently from simple flat asset inventories, this approach models the operational environment as a structured dependency graph, allowing for accurate tracking of resource and their relationships

Contents

1 Introduction **1**
1.1 Context . . . 1
1.2 Motivations . . . 2
1.3 Thesis structure . . . 3

2 Background **4**
2.1 Open Command and Control (OpenC2) . . . 4
2.1.1 Architectural Base . . . 4
2.1.2 Actuator Profiles . . . 5
2.1.3 Commands and Responses . . . 6
2.1.4 Transfer Protocols and Message Formats . . . 7
2.2 OpenC2 Utilities for Python (otupy) . . . 9
2.2.1 Architectural Components . . . 10
2.2.2 Implementation Examples . . . 12
2.3 Bill of Materials (BOM) . . . 14
2.3.1 CycloneDX . . . 18
2.3.2 System Package Data Exchange (SPDX) . . . 25
2.3.3 Software Identification (SWID) Tags . . . 25

3 Related Work **28**

4 Thesis Objectives **30**

5 XBOM Profile Design and Implementation **32**
5.1 Introduction . . . 32
5.2 Profile Design . . . 32
5.2.1 Profile and Message Structure . . . 33
5.3 Command Structure . . . 34
5.3.1 Actions . . . 34
5.3.2 Targets . . . 35
5.3.3 Xbom Arguments . . . 35
5.3.4 Command Arguments . . . 37
5.3.5 Actuator Specifiers . . . 38
5.4 Response Structure . . . 39
5.4.1 OpenC2 Response Message . . . 39
5.4.2 Response Results . . . 40

5.4.3 XBOM Result Extension 41
5.5 Handling XBOM Generation 42
5.5.1 Context Data Model 42
5.5.2 Adapting the Data Model for BOM Generation 44
5.6 Implementation 65
5.6.1 Profile Registration 66
5.6.2 BOM Format Abstraction 66
5.6.3 Actuator Implementation 71
5.6.4 Example: OpenC2 Request and Response 74

6 Profile Validation and testing 80
6.1 Introduction 80
6.2 Unit Testing 81
6.3 Functional Testing 82
6.3.1 Scenario Description 83
6.3.2 Test Execution 85
6.3.3 Cross-Layer Linking 86
6.3.4 Discussion 87
6.4 Performance Evaluation 87
6.4.1 Evaluation Methodology 88
6.4.2 OpenStack Actuator 88
6.4.3 Kubernetes Actuator 89
6.4.4 Discussion 90

7 Conclusions 93
7.1 Extension to Additional BOM Standards 94
7.2 Extension of the Data Model 95
7.3 Final Remarks 97

Bibliography 98

List of Figures

2.1 The OpenC2 Producer-Consumer architecture [1]. 5
2.2 The internal architecture of the `otupy` library. [2] 10

3.1 Context Discovery Actuator Profile data model, adapted from [3]. 28

5.1 Data Model used for the XBOM Actuator Profile 77
5.2 Serialization and deserialization flow for `CyclonedxXbom`. 78
5.3 Discovery and BOM generation flow in `XBOMActuator`. 79

6.1 Simplified diagram of the functional test scenario, showing the network topology. 84
6.2 Visualization of the simplified topology discovered in the functional test scenario 92

Chapter 1

Introduction

In this chapter, the context, as well as the thesis motivations, will be explained. At first, the critical challenges of the modern cybersecurity domain will be described, with a particular focus on the imperative need for effective and interoperable defense mechanisms. In this context, the Open Command and Control (OpenC2) is introduced as a promising solution to address these challenges, providing a consistent and standardized communication framework for cyber defense operations. Then, the motivations behind this thesis will be outlined, emphasizing the importance of developing a standardized way to describe the network resources that are part of the broader context in which OpenC2 operates. Finally, the structure of the thesis will be presented, providing an overview of the subsequent chapters and their content.

1.1 Context

The modern cybersecurity landscape is characterized by a rapid evolution, in both the sophistication and volume, of digital threats. Organizations are continuously confronted with high-frequency attacks, ranging from targeted phishing campaigns to automated ransomware attacks. Consequently, traditional defense paradigms, which rely heavily on human intervention and reactive measures, are proving increasingly inefficient. While human analysis remains relevant for specific cases, it creates a heavy latency gap that may fail to keep up with the velocity of modern automated adversaries. This makes manual responses insufficient for delivering the consistency required to mitigate threats in a timely manner.

Furthermore, digital infrastructures are becoming more complex and interconnected than ever before, with an ever increasing reliance on cloud services, Internet of Things (IoT) devices, and distributed computing environments. This complexity introduces a concrete necessity for orchestrating and automating defense mechanisms across a wide variety of platforms and technologies. Through these mechanisms, security responses can be scheduled, executed, and verified with a consistency that isolated, and occasionally manual, efforts cannot achieve.

In this context, remote control over the security operations is becoming increasingly crucial. The ability to manage and coordinate security measures in an automated way, reducing the need for human intervention, is most essential. The Command and Control (C2) paradigm, which allows for the remote management and centralized management of security operations, has emerged as a promising solution to address these challenges, being a key aspect for managing a wide variety of security tools and technologies, such as firewalls, intrusion detection systems, and threat intelligence platforms. By empowering the infrastructure to execute instantaneous containment measures, such as the revocation of network access or the isolation of compromised devices, centralized C2 architectures are able to drastically reduce the time between detection and response, thus mitigating the impact of cyber attacks.

Nonetheless, the lack of standardization in C2 communication protocols and formats has hindered the widespread adoption and interoperability of C2 solutions. Security platforms and tools often use proprietary protocols and formats, which are usually incompatible with each other, which makes orchestration and automation efforts difficult to implement. In this scenario, the Open Command and Control (OpenC2), proposed by the OASIS organization, is introduced as a promising solution to address these challenges, providing a consistent and standardized communication framework for cyber defense operations. OpenC2 specifies a common language for expressing commands, responses, and status information, virtually enabling interoperability among different security tools. OpenC2 thus allows for a faster, more efficient, and more consistent response to cyber threats, being a key enabler for the orchestration and automation of security in a ever-evolving threat landscape.

1.2 Motivations

While OpenC2 provides a standardized framework for communication, visibility of network resources still remains a feature that has yet to be fully addressed. Without it, effective countermeasures to threats can be difficult to implement, as security operators may lack the necessary information about the network environment to make informed decisions. But to enable effective and efficient context discovery, there's a need for a standardized way to describe the network resources that are part of the broader context in which OpenC2 operates.

The thesis addresses this gap by proposing a standardized way to describe network resources, by designing, implementing and evaluating a ad-hoc OpenC2 Actuator Profile, to represent the network resources in a consistent and interoperable way, extending well recognized standards such as Bill of Materials (BOM) to represent the network resources. To achieve this goal, the following objectives have been identified:

- A common language for describing cyber resources at various levels of granularity

- Enhanced capabilities for resource discovery and tracking during active incidents
- Better alignment between BOM concepts and the requirements of modern cybersecurity frameworks like OpenC2

1.3 Thesis structure

This thesis is structured as follows:

- in Chapter 2, the necessary background information on OpenC2 and related concepts will be provided, including an overview of the OpenC2 language, a brief history of the Bill of Materials (BOM) concept, as well as the current state of the art.
- in Chapter 3, a brief review of the related work in the field of cybersecurity and OpenC2 will be presented.
- in Chapter 4, the requirements for the proposed OpenC2 Actuator Profile will be defined, based on the analysis of the current state of the art and the identified gaps in the field.
- in Chapter 5, the design of the proposed OpenC2 Actuator Profile will be described, including the data model, the syntax, and the semantics of the profile, as well as the rationale behind the design choices made. Additionally, the implementation of the profile will be discussed, including the tools and technologies used.
- in Chapter 6, the testing and validation of the proposed profile will be described, including the unit testing and functional testing activities performed, as well as the results of these activities.
- in Chapter 7, the conclusions of the thesis will be presented, including a summary of the main findings, the contributions of the work, and the future directions for research in this area.

Chapter 2

Background

This chapter provides the necessary background to comprehend the principal concepts and components addressed in this thesis. The discussion includes an overview of the OpenC2 standard, its implementation within the `otupy` framework, and pertinent details. Additionally, the section examines Bill of Materials (BOM) standards, with particular emphasis on CycloneDX, illustrating its application as an XBOM to represent network structure and node interactions.

2.1 Open Command and Control (OpenC2)

Open Command and Control (OpenC2) is a suite of specifications designed to facilitate the direction and management of cyber defense components (such as Firewalls) and systems in real time. Its main goal is to enable interoperability across different technologies to allow for rapid and coordinated responses to cyber threats. This is achieved through a standardized language for machine-to-machine communication [4], which allows security tools from various vendors to understand and execute commands efficiently.

At its heart, this approach transforms fragmented security stacks into an interoperable mesh, which is achieved though the decoupling of high-level intents from low-level execution, mapping defensive logic onto a standardized set of operations, referred to as **Actions**, and their corresponding network targets, known as **Targets**.

2.1.1 Architectural Base

The OpenC2 architecture consists of a modular design, made up of several key components which follow a Producer-Consumer model, also shown in Figure 2.1. As such, the standard defines two main roles:

- **Producer**: This component is responsible for generating OpenC2 commands based on the organization's security policies, threat intelligence, or other inputs. It formulates the high-level intents and translates them into actionable commands that can be understood by the consumer. In

short, it's the component responsible to determine *what* actions need to be taken in response to specific events or conditions and *which* targets should be affected by those actions.

- **Consumer**: This component receives OpenC2 commands from the producer and executes them on the relevant security tools or systems. It acts as an intermediary that translates the standardized commands into specific actions that can be performed by the underlying technology, such as firewalls, intrusion detection systems, or endpoint protection platforms. In short, it's the component responsible for determining *how* the specified actions should be executed on the identified targets, ensuring that the intended security measures are effectively implemented independently of the underlying technology.

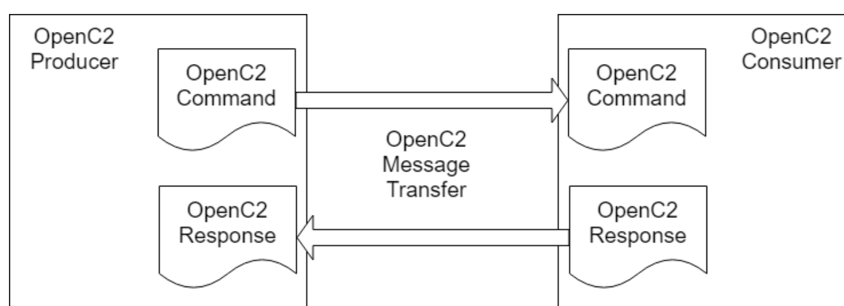

Figure 2.1: The OpenC2 Producer-Consumer architecture [1].

2.1.2 Actuator Profiles

To handle the diversity of security technologies and ensure consistent execution of commands across different environments, actuator profiles in OpenC2 serve as comprehensive specifications that define how particular classes of actuators, such as firewalls, intrusion detection systems, or endpoint protection platforms, should interpret and implement OpenC2 commands.

These profiles act as a detailed guide, mapping the abstract, standardized actions and targets defined by OpenC2 to the concrete capabilities, interfaces, and operational semantics of specific technologies. While the consumer is responsible for receiving commands from the producer and orchestrating their execution, it is the actuator profile that instructs the consumer on how to translate these commands into actionable instructions for the underlying security tools. Each actuator profile delineates the set of supported actions (such as `deny`, `allow`, `query`, or `update`, in the SPLF profile), the types of targets that can be acted upon (such as IP addresses, domains, or files), and the expected behaviors and response formats, ensuring that commands are interpreted and executed in a consistent and predictable manner regardless of device or vendor.

Furthermore, actuator profiles specify any constraints, prerequisites, or optional features relevant to the actuator, which enables the consumer to vali-

date commands, handle exceptions, and provide meaningful feedback to the producer when a requested action is unsupported or cannot be performed.

By using profiles, the standard ensures that the flexibility required to accommodate a wide range of security technologies does not come at the expense of a standardized command structure.

The only profile currently defined by the OASIS Technical Commitee and published as part of the OpenC2 standard is the SPLF (Simple Profile for Firewalls) [5], identified by the `splf` namespace identifier (NSID). This profile focuses on firewall management, providing a standardized set of actions and targets for controlling network traffic, such as allowing or denying specific connections based on IP addresses, ports, and protocols.

2.1.3 Commands and Responses

Communication in OpenC2 is structured around a standardized set of two message types: **Commands** and **Responses**. A Command, generated by the consumer, is a structured message that encapsulates a specific action to be performed on a defined target. The standard defines a set of fields, two of which are mandatory with a few optional ones [4]:

- **Action**: The operation to be performed, such as `deny`, `allow`, `query`, or `update`. This is usually a verb.
- **Target**: The object of the action, which could be an IP address, domain, file, process, or other entity relevant to the security context.
- **Arguments**: Optional parameters that refine the action, such as time constraints, reasons, or additional configuration details. Some examples include the `duration` and `response_requested` fields.
- **Profile**: An optional field that specifies the actuator profile to be used for executing the command.
- **Command ID**: An optional unique identifier for the command, which can be used for tracking and correlation purposes.

The structure of OpenC2 commands mirrors conventional language patterns, which enables clear and straightforward assignment of roles: the actuator serves as the subject, the action as the verb, the target as the object, and arguments provide additional context or modifiers. Every command must specify a single action and a single target type. The actions are strictly defined in the OpenC2 language specification, and it's not possible to create custom ones [4]. This is necessary to ensure that the commands are universally understood and can be executed by any compliant consumer.

After a Command has been correctly received and processed by the consumer, a Response is generated to provide back the outcome of the command execution. A Response typically includes:

```json
{
  "action": "deny",
  "target": {
    "ipv4_connection": {
      "protocol": "tcp",
      "dst_port": 22,
      "dst_addr": "192.168.1.92",
      "src_addr": "192.167.1.67"
    }
  },
  "actuator": {
    "splf": {}
  }
}
```

Listing 1: Example of a simple OpenC2 command in JSON format.

- **Status**: Indicates the success or failure of the command execution. This is provided as a status code.
- **Status_text**: An optional field that provides human readable details about the status field.
- **Result**: An optional dictionary that contains any relevant data or information resulting from the command execution, such as query results or error details.

```json
{
  "status": 200,
  "status_text": "Command executed successfully",
}
```

Listing 2: Example of a simple OpenC2 response in JSON format.

Take a look at listings 1 and 2. The two examples illustrate a simple OpenC2 command in which a producer instructs a consumer to deny an SSH connection from a specific source IP address to a destination IP address on port 22. The corresponding response indicates that the command was executed successfully, providing a status code of 200 and a human-readable message confirming the action taken.

2.1.4 Transfer Protocols and Message Formats

The OpenC2 standard is designed to be transport-agnostic, which allows commands and responses to be transmitted over a variety of different communication protocols. This ensures a certain degree flexibility is maintained, allowing

to adapt to different network environments and requirements. However, to promote consistent interoperability across different implementations, the OASIS technical committee has defined a set of specific transfer protocols that are recommended for use with OpenC2. These include:

- **HTTP/HTTPS** [6]: A widely used protocol for transmitting OpenC2 messages, leveraging RESTful APIs for reliable communication between producers and consumers. This protocol provides a series of desirable features such as security, scalability, and ease of integration with existing web-based systems.
- **MQTT** [7]: A lightweight messaging protocol designed for "constrained" environments, which makes it suitable for IoT devices and other resource-limited systems that need to communicate OpenC2 commands and responses efficiently.

In terms of message formats, OpenC2 supports multiple serialization formats regarding the message payload, to ensure that the information can be easily parsed and understood. The default solution for OpenC2 messages is JavaScript Object Notation (JSON) [8], which is a widely adopted format for data interchange due to its simplicity and readability.

Additionally, OpenC2 also supports Concise Binary Object Representation (CBOR), a binary serialization format specifically designed to be used in constrained environments by providing a more compact alternative to JSON.

Listing 3 provides an example of how an OpenC2 command can be encapsulated within an HTTP POST request following the HTTPS Transfer Specification. First, a connection is established between the producer and the consumer using the TCP protocol. Then, the producer sends an HTTP(S) POST request to the consumer's endpoint, which is typically defined as a well-known URI (Uniform Resource Identifier), which in this case is `/.well-known/openc2`. The request includes the OpenC2 command in the body, formatted as JSON, and contains necessary headers such as `Content-type`, `Date`, and a unique `X-Request-ID` for tracking purposes. The consumer processes the command and generates an appropriate response based on the execution outcome.

```
POST /.well-known/openc2 HTTP/1.1
Content-type: application/openc2+json;version=1.0
Date: Mon, 09 Feb 2026 12:00:00 GMT
X-Request-ID: d1ac0489-ed51-4345-9175-f3078f30afe5

{
  "headers": {
    "request_id": "d1ac0489-ed51-4345-9175-f3078f30afe5",
    "created": 1739102400000,
    "from": "producer.example.com",
    "to": ["actuator.example.com"]
  },
  "body": {
    "openc2": {
      "request": {
        "action": "deny",
        "target": {
          "ipv4_connection": {
            "protocol": "tcp",
            "dst_port": 22,
            "dst_addr": "192.168.1.92",
            "src_addr": "192.167.1.67"
          }
        },
        "args": {
          "actuator": {
            "splf": {}
          }
        }
      }
    }
  }
}
```

Listing 3: Example of an OpenC2 deny command encapsulated in an HTTP POST request

2.2 OpenC2 Utilities for Python (otupy)

OpenC2 Utilities for Python, commonly referred to as `otupy` [2] and previously known as `openc2lib`, serves as a fundamental component for implementing the OpenC2 standard in a Python-based environment, being supported by the open-source community.

The primary objective of `otupy` is to provide a cohesive framework that encapsulates the specifications defined in the OpenC2 Language Specification

and the various Transfer Specifications briefly discussed in section 2.1. By doing so, it allows developers to instantiate Producer and Consumer entities without needing to manually parse raw JSON strings, manage HTTP headers, or implement the low-level handshake details of message queuing protocols. The library is designed with a modular architecture that strictly separates the application layer logic, where high-level commands are generated and processed, from the transport layer mechanics, supporting both the HTTP and MQTT protocols out of the box.

2.2.1 Architectural Components

The internal architecture of `otupy` is built around the interaction between several key components: the Message object, the Transcoder, the Transport module, and the high-level Producer-Consumer classes.

The core architecture is shown in Figure 2.2. This diagram serves not only as a structural map but as a guide to the data lifecycle within an OpenC2 transaction.

At the highest level, the diagram distinguishes between the `otupy` framework itself and the external application logic. On the left, the **Security Controller** represents the user's orchestration logic, the decision making entity that decides when or not to issue a command. On the right, the **Security Servers** represent the target environment where defense actions are enforced. The framework sits as the intermediary, translating intent into actual commands that can be enforced. The dotted lines indicates the workflow of each component, whereas the solid lines represent the external API and thus, the data flow between the application and the framework.

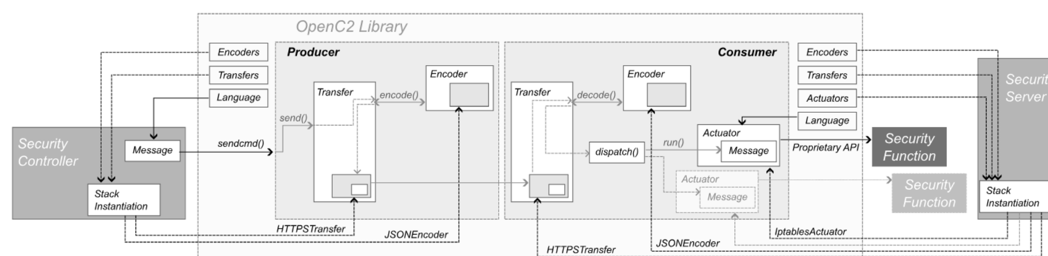

Figure 2.2: The internal architecture of the `otupy` library. [2]

The architectural design of `otupy` is centered around a rigorous implementation of the objects described in the OpenC2 Language Specification. The framework provides a comprehensive API that mirrors the standard's definitions for Messages, Commands, Responses, and all associated target and data types.

Internally, these objects are built upon widely-used, robust Python libraries; for instance, the `IPv4Addr` object is a wrapper around the standard `ipaddress.IPv4Address` class found in Python's networking stack. This allows to effectively decouples the OpenC2 API from the underlying implementation details, facilitating future updates to the framework without breaking

external software dependencies. Furthermore, it flattens the learning curve for developers already familiar with the standard, as the terminology used in the code matches the specification exactly—with minor exceptions for syntax compatibility, such as converting dashes to underscores and trailing reserved keywords.

A minor deviation from this strict mapping is observed in the `Message` object. This design choice addresses a specific characteristic of the OpenC2 ecosystem: while the standard rigorously defines the *content* of a command (via the Language Specification), it intentionally leaves the *container*, the message syntax itself, open to interpretation by the specific Transfer Specifications (such as HTTPS or MQTT).

To bridge this gap without complicating the developer's experience, `otupy` introduces a unified internal structure for the message object. Rather than forcing the application logic to handle the complexities of different protocols, this object serves as a standardized "envelope" within the library. It holds not only the core command or response payload but also the essential metadata—such as serialization formats, encoding types, and tracking identifiers—in a format that is agnostic to the transport layer.

Core Components: Producer and Consumer

The operational core of the framework consists of the **Producer** and **Consumer** objects, which implement the respective OpenC2 roles. Unlike monolithic implementations, these objects rely on a modular protocol stack composed of **Encoder** and **Transfer** interfaces.

When a Producer or Consumer is instantiated, it is injected with concrete implementations of these interfaces (e.g., a `JSONEncoding` object for serialization and an `HTTPTransfer` object for communication). In the case of the Consumer, the object is additionally provisioned with a registry of available **Actuators** and their configurations. The logic for message routing is handled by a `dispatch()` function, which selects the appropriate actuator to execute a received command based on the message content.

The Two-Step Serialization Process

A distinct feature of the `otupy` architecture is its serialization pipeline. The conversion from Python objects to wire-format data occurs in two distinct steps:

1. **Meta-serialization**: An internal process generates an intermediary, nested key–value representation of the object. This format is agnostic to the OpenC2 language and the final output format.

2. **Encoding**: The generic `Encoder` interface translates this intermediary representation into the specific serialization format required, such as JSON, YAML, or CBOR.

Deserialization functions symmetrically, with Encoders parsing the raw data into the intermediary format, which the internal meta-serialization process then uses to reconstruct the `otupy` objects.

Transfer and Actuation Interfaces

The **Transfer** interface encapsulates both the transport protocols and associated security services. In `otupy`, separating these elements was deemed impractical for many use cases, so they are embedded together. Each Transfer implementation is responsible for managing its own message structure and adapting the payload to its communication paradigm, handling details such as HTTP headers or MQTT topics.

Finally, **Actuators** serve as the bridge between the standardized OpenC2 messages and the proprietary interfaces of specific security functions. An Actuator is the concrete implementation of a Profile for a specific cyber-appliance. The interface is streamlined to a single method, which executes a Command and, if applicable, returns a Result. For example, an implementation of the Simple Profile for Firewalls (SPLF) might be realized as an `IptablesActuator`, translating abstract `deny` commands into specific `iptables` rules, while another might map the same commands to `pfsense` APIs.

This modular design allows for an easier interation and extension of the framework: new protocols or proprietary security tools can be integrated by simply implementing these interfaces, which hopefully reduces the barrier to entry for developers and encourages a wider adoption of the framework by the community.

2.2.2 Implementation Examples

To bring the architectural concepts of modularity and dependency injection to life, an example of instantiating both a Consumer and a Producer is provided in the following listings. These examples should help understanding how the framework can work with different protocols and encodings, and how the actuator dispatch logic is implemented in practice.

Listing 4 illustrates the initialization of a Consumer entity. The code explicitly defines the protocol stack by instantiating concrete classes for the encoding (`YAMLEncoder`) and transport (`HTTPTransfer`) layers. These instances are passed into the Consumer's constructor, effectively "injecting" the desired behavior.

Furthermore, the example demonstrates the actuator dispatch logic. An `actuators` dictionary is defined to map specific profiles to concrete implementations. In this case, the standard Simple Profile for Firewalls (SPLF), identified by its namespace ID (`nsid`), is mapped to an `IptablesActuator`. This registration process tells the internal dispatcher that any incoming command targeting the SPLF profile should be routed to this specific Python object for execution.

```python
import otupy as oc2
from otupy.encoders.yaml_encoder import YAMLEncoder
from otupy.transfers.http_transfer import HTTPTransfer
import otupy.profiles.slpf as slpf
from otupy.actuators.iptables_actuator import IptablesActuator

actuators = {
    (slpf.nsid, 'iptables'): IptablesActuator()
}

consumer = oc2.Consumer(
    "yaml-consumer.example.net",
    actuators,
    YAMLEncoder(),
    HTTPTransfer("0.0.0.0", 9090)
)

consumer.run()
```

Listing 4: Example of instantiating a Consumer with HTTP transfer, YAML encoding, and a registered Actuator.

Listing 5 shows the corresponding Producer configuration. Similar to the consumer, the Producer is initialized with matching dependencies, specifically the `YAMLEncoder`, to ensure the messages it generates can be understood by the consumer. The `HTTPTransfer` in this context is configured with the target URL of the consumer.

```python
import otupy as oc2
from otupy.encoders.yaml_encoder import YAMLEncoder
from otupy.transfers.http_transfer import HTTPTransfer

# Connect to the YAML/HTTP Consumer
producer = oc2.Producer(
    "yaml-producer.example.net",
    YAMLEncoder(),
    HTTPTransfer("127.0.0.1", 9090)
)
```

Listing 5: Example of instantiating a Producer with HTTP transfer and YAML encoding.

Finally, it is important to clarify that while the Security Controller is a key architectural concept, it is not implemented as a standalone class in this example. Instead, the logic of a generic controller is constructed by combining the `Consumer` and `Producer` primitives. In this context, the instantiated `Producer` serves as the controller's operational interface, acting as the gateway

to the high-level API. Listing 6 illustrates the generation and transmission of a fully typed OpenC2 command.

This approach enforces type safety and ensures adherence to the standard's schema constraints regarding action-target compatibility and argument types before the message is ever serialized. In the example, a `deny` action is constructed targeting a specific IPv4 connection. The command is further enriched with arguments requesting a complete response and explicitly specifying the parameters for the `slpf` (Simple Profile for Firewalls) actuator profile.

```
import otupy as oc2
import otupy.profiles.slpf as slpf
from otupy.encoders.yaml_encoder import YAMLEncoder
from otupy.transfers.http_transfer import HTTPTransfer

target = oc2.Targets(ipv4_connection={'src_addr': '198.51.100.55'})

pf = slpf.slpf({
      'hostname':'firewall',
      'named_group':'firewalls',
      'asset_id':'iptables'
     })
arg = slpf.ExtArgs({'response_requested': oc2.ResponseType.complete})

cmd = oc2.Command(
    action=oc2.Actions.deny,
    target=target,
    args=arg,
    actuator=firewall_actuator
)

response = producer.send_command(cmd)
```

Listing 6: Example of generating an OpenC2 command using the Producer's API.

2.3 Bill of Materials (BOM)

A Bill of Materials (BOM) is a comprehensive list of parts, items, assemblies, and other materials required to create a product. In the cyber domain, Software Bill of Materials (SBOM) has become a standard for tracking software components and dependencies. However, the scope of BOMs is expanding to include other domains such as hardware and services.

The concept of a Bill of Materials (BOM) is not a novel invention of the digital age; rather, its origins are deeply rooted in the industrial revolution and the manufacturing sectors. Historically, a BOM served as a definitive inventory, detailing the raw materials, sub-assemblies, and specific quantities

required to manufacture a physical product [9]. This centralized record was critical for supply chain management, ensuring that production lines were not halted due to material shortages and that costs could be accurately estimated. As engineering disciplines matured throughout the 20th century, the BOM evolved from a static list into a dynamic engineering document utilized for lifecycle management, maintenance, and regulatory compliance. It became the authoritative source of truth for the composition of complex machinery, allowing engineers to trace defects back to specific batches of components and facilitating the standardization of parts across different product lines.

In the digital era, the transition of the BOM concept to software development was made necessary by the exponential increase in the complexity of modern applications. If you just think about it, many contemporary software projects rely on a vast ecosystem of open-source libraries, third-party APIs, cloud services, and so on. The shift towards modular architectures, microservices, and the heavy reliance on open-source libraries created a landscape where a significant portion of a software's codebase is not authored by the primary developers, but rather inherited from third-party dependencies. This is not necessarily a problem in itself, as it allows for rapid development and innovation, but it often leads to a phenomenon, often referred to as "dependency hell", introduces systemic opacity [10]. Without a comprehensive inventory, organizations are effectively deploying "black boxes", being unaware of the underlying components that make up their critical infrastructure. This lack of visibility is bond to become a significant liability, as vulnerabilities in deep dependencies could remain undetected for extended periods, buried under an ever-increasing layer of abstraction.

The strategic necessity for Software Bills of Materials (SBOMs) has become increasingly clear with a series of high-profile supply chain attacks that exposed the fragility of the digital ecosystem. Incidents such as the SolarWinds compromise and the Log4j vulnerability demonstrated that the security of a system is defined not just by its perimeter defenses, but by the integrity of its constituent parts [11]. In the wake of these events, the SBOM emerged not merely as an inventory tool, but as a critical instrument for general security and operational resilience. This stance was solidified by regulatory frameworks, most notably the US Executive Order 14028 on Improving the national cybersecurity levels [12], which mandated the provision of SBOMs for software sold to the federal government, which highlighted the importance of transparency in the software supply chain. This regulatory pressure marked a paradigm shift, transforming the BOM from a best practice into a compliance necessity, following the broader trend of "security as code" where transparency and automation are a necessity.

However, the confinement of the BOM concept solely to the software layer is strictly insufficient for modern, cyber-physical architectures. As systems became increasingly hybrid, spanning on-premises hardware, cloud-native microservices, and embedded firmware, the industry began to align around the

concept of the "XBOM" (eXtended Bill of Materials). This concept extends beyond the traditional static inventory of software packages; it encompasses Hardware (HBOM), Software-as-a-Service (SaaS-BOM), and even Data (DBOM). Crucially, the significance of the XBOM lies not only in the mere enumeration of these distinct elements, but in the rigorous mapping of their interdependencies and interactions. A static list of assets is similar, at it's core, to a set of unassembled mechanical parts; it offers no insight into function or criticality, as such, it could be insufficient for effective risk assessment and response planning in a security context. By contrast, a mature XBOM models the system as a directed graph, explicitly defining which software runs on which hardware, which service consumes data from another, and which API endpoints are exposed to the public internet, among other relationships.

This structural evolution serves as the fundamental prerequisite for establishing a truly structured representation of the environment in which a security platform operates. This structured representation possesses intrinsic value, serving as an authoritative "source of truth" that documents the operational landscape with an, often, high degree of granularity. Moreover, this standardization is the most important for interoperability; it transforms the environment into a machine-readable dataset that can be easily queried, analyzed, and utilized by other state-of-the-art analysis and orchestration tools. It also comes with the added benefit of eanabling integration with external, well recognized tools and standards, such as vulnerability databases, asset management systems, and risk assessment frameworks, which can leverage the structured data to provide more accurate and context-aware insights.

In the domain of active defense, such as that orchestrated by OpenC2, the availability of such a rigorous, structured context is most important. Automated response mechanisms demand a high-fidelity, interoperable model of the "defended landscape" to operate with safety and precision. When a threat is detected, the standardized nature of the BOM allows disparate security tools to instantaneously query the environment to gather information about the potential impact. For instance, if a specific node is compromised, the graph-based structure of the XBOM enables an orchestrator to traverse the dependency tree, identifying every service or business function that is provided by or relies on that component. This capability elevates context discovery from a simple cataloging exercise into a dynamic, real-time process that informs decision-making and response actions.

Following this subsection, we will provide a brief overview of the different types of BOMs that are currently being standardized.

Software Bill of Materials (SBOM)

The SBOM remains the cornerstone of digital transparency and the most mature of the BOM domains. It provides a formal, nested record of the components used to build software, detailing the complex web of proprietary source code, open-source libraries, and transitive dependencies that make up a modern application. Its primary utility extends beyond simple inventory; it is

the fundamental enabler for software supply chain security. By mapping the "ingredients" of an application, organizations can instantly correlate newly disclosed Common Vulnerabilities and Exposures (CVEs) to specific artifacts within their ecosystem, transforming vulnerability management from a chaotic search into a deterministic lookup process.

Software-as-a-Service Bill of Materials (SaaS-BOM)

As infrastructure increasingly shifts towards cloud-native and serverless paradigms, the traditional boundaries of software ownership have blurred. The SaaS-BOM extends the inventory concept to include the remote services, API endpoints, and microservices that an application consumes but does not host or control. This domain addresses the opacity of the "black box" dependencies inherent in modern web applications, allowing architects to assess risks related to data sovereignty, service availability, and third-party trust. It essentially maps the "external" attack surface, documenting not just code, but the flow of data across organizational boundaries.

Cryptography Bill of Materials (CBOM)

The CBOM is a specialized inventory designed to catalog cryptographic assets, such as algorithms, libraries, certificates, and key lengths. In an era where cryptographic standards are evolving rapidly, particularly with the impending necessity of migrating to post-quantum cryptography (PQC), the CBOM provides critical visibility. It allows organizations to assess their "crypto-agility" by identifying weak, deprecated, or non-compliant primitives (such as SHA-1 or RSA-1024) buried deep within compiled binaries. This inventory is a prerequisite for planning the systematic upgrade of cryptographic defenses before current standards become obsolete or compromised.

Hardware Bill of Materials (HBOM)

The HBOM applies the principles of traceability to the physical domain, which is particularly relevant for IoT devices, embedded systems, and critical infrastructure. It details the components of a physical device, from the printed circuit board (PCB) and microcontrollers to the specific firmware versions residing on them. By linking the physical layer to the digital stack, the HBOM helps in detecting counterfeit parts, managing hardware end-of-life cycles, and analyzing the security implications of the hardware-software interface, ensuring that the physical substrate of the system is as secure as the code running atop it.

Machine Learning Bill of Materials (ML-BOM)

Artificial Intelligence systems introduce unique challenges that standard software inventories cannot address. The ML-BOM documents the composition of AI models, including the neural architecture, the specific datasets used for

training and validation, and the lineage of the model's weights. This transparency is essential for AI governance and safety, ensuring that models are robust against adversarial attacks (such as data poisoning) and free from inherent biases introduced during the training phase. It provides the information required to audit decision-making algorithms in any kind of environment in which they may be employed.

Operational Bill of Materials (OBOM)

While an SBOM describes the static composition of software as it was built, the OBOM captures its runtime environment and configuration. It details the operating system versions, environment variables, compiler flags, loaded modules, and active settings present during deployment. This context is critical for risk assessment because the presence of a vulnerable library does not always equate to exploitability; the OBOM helps distinguish theoretical risk from actual exposure by confirming whether the vulnerable code path is actually accessible or active in the current production environment.

Bill of Vulnerabilities (BOV)

The BOV serves as a companion artifact to the standard BOM, focusing on the security status of the components rather than their mere presence. Often implemented through mechanisms like the Vulnerability Exploitability eXchange (VEX), a BOV allows software vendors to communicate the disposition of vulnerabilities—asserting whether a component is "affected," "fixed," or "not affected." This significantly reduces the operational burden on security teams by suppressing false positives, allowing them to ignore vulnerabilities that, while present in the code, are rendered inert by the application's architecture or configuration.

2.3.1 CycloneDX

In response to the escalating complexity of supply chain risk management, the Open Web Application Security Project (OWASP) architected CycloneDX [13] as a comprehensive, security-centric standardization format. Unlike earlier specifications that were primarily derived from license compliance workflows, CycloneDX was conceived to support the high-velocity requirements of modern DevSecOps and automated vulnerability analysis. It stands as a "full-stack" implementation of the XBOM concepts previously discussed, providing a unified schema capable of describing the composition of software, hardware, and services within a single, coherent document.

The standard distinguishes itself through a design philosophy centered on lightweight extensibility and machine readability. By supporting multiple serialization formats—including JSON, XML, and Protocol Buffers, it ensures easy integration into both human-centric auditing workflows and automated

pipelines. This flexibility allows the standard to serve as a universal data interchange format, capable of modeling the intricate relationships between an SaaS-BOM's remote endpoints, an HBOM's physical components, the runtime configurations and environment captured by an OBOM, and the cryptographic primitives defined in a CBOM. Furthermore, the specification has evolved to include advanced capabilities for distinguishing between the mere presence of a vulnerability and its actual exploitability. By natively integrating VEX data, CycloneDX allows vendors to assert the status of a component—justifying why a specific CVE is not applicable due to compiler flags or runtime configurations—thereby acting as a dynamic intelligence layer rather than a static inventory.

This capability has driven widespread industry adoption [14] , establishing the standard as a critical enabler for regulatory compliance, such as meeting the stringent transparency requirements of the US Executive Order 14028.

Components and Services

To operationalize the concept of an XBOM, the CycloneDX specification divides the inventory of a system into two primary distinct categories: **Components** and **Services**. This architectural distinction is fundamental, enabling the standard to accurately model modern, distributed systems where functionality is derived not only from static assets, assembled during the build process, but also from dynamic interactions between remote elements. This separation allows security tools to apply distinct risk assessment methodologies appropriate for internal artifacts versus external dependencies.

Components The `components` array constitutes one of the foundational inventory of the BOM, designed to describe, mostly, the tangible and logical parts that comprise a system. Unlike legacy inventory formats, usually limited to software libraries, the CycloneDX component taxonomy covers a wide range of asset types. The standard defines a rigorous classification system via the `type` attribute, which distinguishes between diverse asset classes such as `library`, `application`, `framework`, `container`, `operating-system`, `device`, `firmware`, and `file`.

Services As architectural paradigms shift towards cloud-native, microservices, and serverless designs, the security posture of an application is increasingly defined by the external APIs and services it consumes, among other things. The `services` array addresses this by describing the external capabilities that the system relies upon but does not necessarily own, host, or distribute. This section is particularly relevant for the generation of SaaS-BOMs and for mapping the external attack surface.

A service definition in CycloneDX encapsulates operational metadata, which are significantly different from static component attributes. Key fields include the service `provider`, the specific `endpoints` (URIs) accessed, and the

`authenticated` status required for interaction. Crucially, the standard allows for the modeling of **Data Flow**. Service definitions can include classifications of data that cross the trust boundary, specifying the direction of flow (inbound, outbound, or bidirectional). This capability is instrumental for data flow mapping and privacy impact assessments, necessary for compliance with data sovereignty regulations (e.g., GDPR). This allows security architects to visualize where sensitive data is leaving the controlled environment.

Structural Composition and Reference Mechanisms

The true power of CycloneDX lies not only in its ability to list components, but in its capacity to model the structural integrity and dependency graph of a system. To achieve this, the standard employs a sophisticated set of linking mechanisms that transform a flat list of assets into a relational topology. In the following paragraphs, the more relevant of these mechanisms will be described.

BOM Reference (bom-ref) Every component, service, or vulnerability within a CycloneDX document can be assigned a unique identifier known as a `bom-ref`. This attribute serves as an internal anchor, and occasionally external, allowing other parts of the document to point specifically to that element. This is crucial for distinguishing between multiple instances of the same library used in different contexts or versions within the same project, as well as referencing a component multiple times without redundancy.

The example of a CycloneDX SBOM is shown in Listing 7. It illustrates how two components, "acme-library" and "my-app," are defined with unique `bom-ref` identifiers. The `dependencies` section then explicitly states that "my-app" depends on "acme-library" by referencing their respective `bom-ref` values. This structure allows for a clear mapping of relationships, enabling consumers of the BOM to understand not just what components are present, but how they interact and depend on each other.

Dependency Graph Utilizing these references, the standard defines a `dependencies` node, which functions as an adjacency list. This structure explicitly maps the direct relationships between components, identifying which library imports another, among other things. By chaining these relationships, a consumer of the BOM can reconstruct the full dependency tree (or graph), enabling the "blast radius" analysis required to trace a vulnerability from a deep transitive dependency up to the main application.

Assembly and Composition To accurately model the hierarchical nature of modern systems, CycloneDX distinguishes between the "Subject" of the BOM and its constituent "Inventory." In standard configurations, the `metadata.component` field serves as the root of the assembly—the primary entity (e.g., the final application, container, or device) being described—while the `components` list defines the sub-assemblies and parts contained within it.

```
...
{
  "components": [
    {
      "type": "library",
      "name": "acme-library",
      "version": "1.0.0",
      "bom-ref": "pkg:maven/com.acme/acme-library@1.0.0"
    },
    {
      "type": "application",
      "name": "my-app",
      "version": "2.0.0",
      "bom-ref": "pkg:maven/com.mycompany/my-app@2.0.0"
    }
  ],
  "dependencies": [
    {
      "ref": "pkg:maven/com.mycompany/my-app@2.0.0",
      "dependsOn": [
        "pkg:maven/com.acme/acme-library@1.0.0"
      ]
    }
  ]
}
...
```

Listing 7: Example SBOM JSON with dependencies

```
{
  ...
  "metadata": {
    "component": {
      "type": "application",
      "name": "Final Product"
    }
  },
  "components": [
    {
      "type": "library",
      "name": "Internal Component"
    }
  ],
  ...
}
```

Listing 8: Example of a CycloneDX BOM with a defined root component

However, the standard also accommodates scenarios where the `metadata.component` field is intentionally omitted. In this configuration, the document shifts from describing a single, hierarchical assembly (a Product BOM) to representing an unrooted collection of artifacts (an Inventory BOM). This structural mode is particularly relevant for the environment discovery use cases discussed in this chapter, where a scanner may identify a disparate set of assets—running services, installed packages, or network devices—that inhabit the same operational space but do not form a singular buildable unit. Here, the BOM acts as a flat catalog for context data, allowing security tools to ingest the "state of the world" without forcing artificial parent-child relationships onto independent entities.

Regardless of the root structure, a list of parts does not inherently guarantee an accurate representation of the whole. To address this, the standard introduces the `compositions` structure, which formally defines the completeness of the assembly. Through the `aggregate` property, an author can explicitly assert whether a given set of components represents a `complete` inventory, an `incomplete` subset, or merely `incomplete_first_party_only` artifacts. This distinction is mathematically significant for risk scoring; if a BOM is declared `complete`, the absence of a known vulnerability implies safety. Conversely, if declared `incomplete`, the absence of a vulnerability implies only a lack of visibility, necessitating different risk mitigation strategies.

```json
{
  "compositions": [
    {
      "aggregate": "complete",
      "assemblies": [
        "pkg:maven/com.example/application@1.0.0"
      ]
    },
    {
      "aggregate": "incomplete_first_party_only",
      "assemblies": [
        "pkg:maven/com.partner/library@2.5.0"
      ]
    }
  ]
}
```

Listing 9: JSON example of composition aggregate assertions

External References To support "System of Systems" architectures, the standard utilizes `externalReferences`. This mechanism allows a BOM to link to external resources, such as security advisories, source code repositories, or even other BOMs. For example, an HBOM describing a physical router can use an external reference to point to the SBOM of the firmware running on

it. This creates a federated graph where distinct BOMs, potentially managed by different teams or vendors, can be logically connected to form a complete picture of the operational environment. This also improves readability and maintainability, as it allows for the separation of concerns; a single BOM can focus on describing a specific component or layer of the system, while still providing the necessary links to the broader context through external references. This is also essential since modern CycloneDX XBOMs can become very large very quickly.

```
...
"components": [{
  "type": "library",
  "name": "cyclonedx-specification",
  "externalReferences": [
    {
      "type": "bom",
      "url": "https://cyclonedx.org/bom.json",
      "comment": "The Bill of Materials (BOM) for this component"
    }
  ]
}]
...
```

Listing 10: JSON representation of CycloneDX External References

Listing 10 demonstrates how external references are structured within the CycloneDX JSON format, allowing components to link to version control systems, websites, or other relevant resources.

Extensibility

The design of the CycloneDX specification recognizes that while standardization is crucial, it should not come at the expense of practical flexibility. Real-world environments often have unique requirements that a global schema cannot fully anticipate. To bridge this gap, the standard provides built-in extensibility mechanisms that allow organizations to include additional, specialized data without compromising the document's structure or its ability to be read by standard security tools.

The primary mechanism for this flexibility is the `properties` taxonomy. Properties allow authors to attach arbitrary key-value pairs to almost any entity within the BOM, including the metadata, components, services, and vulnerabilities. This name-value pair structure is instrumental for organizations that need to correlate standardized BOM data with internal, domain-specific systems. For example, a financial institution might use properties to map a software component to an internal cost center or a specific regulatory compliance mandate that is not natively represented in the CycloneDX schema.

Because these properties are structured, they can be easily queried by custom orchestration tools while being safely ignored by standard-compliant tools that do not recognize them.

Furthermore, CycloneDX supports the use of custom namespaces, allowing for even more complex, structured data to be embedded directly into the document. This capability is vital for specialized industries—such as aerospace or healthcare, which may require the inclusion of telemetry data, safety certifications, or environmental impact metrics within the Bill of Materials. By providing a standardized way to "carry" this extra information, CycloneDX facilitates a unified data exchange where the "source of truth" for a system's composition also serves as the vehicle for its broader operational and organizational context. A list of the currently registered namespaces can be found on the organization's repository[1]

```
{
  "bomFormat": "CycloneDX",
  "specVersion": "1.7",
  "components": [
    {
      "type": "library",
      "name": "secure-auth-provider",
      "version": "4.2.0",
      "bom-ref": "pkg:npm/secure-auth@4.2.0",
      "properties": [
        {
          "name": "internal:asset-id",
          "value": "FIN-99023-X"
        },
        {
          "name": "security:risk-score",
          "value": "low"
        },
        {
          "name": "governance:data-classification",
          "value": "highly-confidential"
        }
      ]
    }
  ]
}
```

Listing 11: Example of CycloneDX component utilizing the properties extension mechanism.

As illustrated in Listing 11, the `properties` field enables the enrichment of a library component with organizational metadata, such as an internal asset

[1] https://github.com/CycloneDX/cyclonedx-property-taxonomy

identifier and a custom data classification. This approach ensures that while the core component identity remains interoperable and machine-readable for any global security tool, the document simultaneously provides the specific context required for internal risk management and governance workflows.

2.3.2 System Package Data Exchange (SPDX)

Software Package Data Exchange (SPDX) [15] is a set of open standards designed to facilitate the exchange of component information, licenses, copyrights, and security references within the software supply chain. Being maintained by the Linux Foundation, its main goal is to reduce the friction and redundancy associated with compliance processes by providing a common language for metadata exchange. This is achieved through a rigorous specification that has been ratified as ISO/IEC 5962:2021, which defines the syntax and semantics for describing software packages, their dependencies, and associated legal and security information.

At the heart of the SPDX ecosystem is its capability to provide legally significant data with high fidelity. A central pillar of this capability is the SPDX License List, a curated and versioned database encompassing over 600 distinct software licenses and exceptions. This list serves as the industry's authoritative reference, eliminating the ambiguity of ad-hoc naming conventions and enabling automated tools to deterministically map obligations to components. Unlike formats restricted to package-level metadata, SPDX offers granular introspection capabilities, allowing for the description of assets down to the specific file and code snippet level. This allows for the identification of partial code reuse, where fragments of open-source logic are embedded within proprietary files, ensuring that intellectual property risks are managed with the same precision as security vulnerabilities.

Furthermore, the standard supports a diverse set of serialization formats, ranging from the human-readable Tag-Value format to machine-parsable JSON and RDF (Resource Description Framework). With the transition to SPDX 3.0, the standard has adopted a modular architecture, introducing profiles for security, AI, and data, thereby expanding its scope from a compliance instrument to a semi-comprehensive XBOM solution: it still presents a relationship mode, being able to make explicit definitions of semantic connections between elements, such as `contains`, `dependsOn`, `descendantOf`, but it still lacks many of the expressive XBOM capabilities which makes other standards more suitable for wider ranges of use cases.

2.3.3 Software Identification (SWID) Tags

While SBOM standards like CycloneDX and SPDX focus heavily on the granular decomposition of software into its constituent dependencies, the Software Identification (SWID) tag serves a complementary but distinct purpose in the

lifecycle of digital assets. Standardized under ISO/IEC 19770-2 [16], SWID tags provide a standardized, machine-readable format for identifying and describing software products.

Historically rooted in Software Asset Management (SAM), the primary utility of SWID tags is to facilitate license compliance and inventory management by providing authoritative "evidence" of software installation. A SWID tag is essentially an XML file that is installed alongside the software executable, containing a wide set of metadata, such as the software name, version, creator (licensor), and a unique identifier (TagID). Unlike the complex dependency graphs of an XBOM, a SWID tag acts as a definitive label—similar to a barcode on a physical product—that asserts the identity of the deployed artifact.

In the context of cybersecurity and the Bill of Materials ecosystem, SWID tags have evolved into a critical mechanism for endpoint visibility. NIST Internal Report 8060 [17] highlights the role of SWID tags in supporting continuous monitoring and vulnerability assessment. For an automated context discovery system, the presence of a SWID tag significantly lowers the computational overhead required to identify assets. Instead of resorting to heuristic scanning or binary analysis to determine what software is running on an endpoint, an agent can simply query the filesystem for valid SWID tags, instantly retrieving high-fidelity identity data.

```xml
<?xml version="1.0" encoding="utf-8"?>
<SoftwareIdentity
  xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  name="Acme Enterprise Firewall"
  tagId="com.acme.firewall-1.0.0"
  version="1.0.0"
  versionScheme="multipartnumeric">
  <Entity
    name="Acme Corp"
    regid="acme.com"
    role="tagCreator softwareCreator" />
  <Link
    rel="supplemental"
    href="https://acme.com/sbom/firewall-1.0.0.json"
    type="application/cyclonedx+json"
    ownership="shared"
    use="required" />
  <Meta
    product="Enterprise Firewall"
    colloquialVersion="1.0"
    edition="Enterprise" />
</SoftwareIdentity>
```

Listing 12: Example of a SWID tag linking to an external SBOM

However, the scope in which SWID tags are used is a little different from the ones described in section 2.3.1 and 2.3.2. It is not intended to be a comprehensive inventory format, but rather a high-level identifier that can be used to link to more detailed BOMs. Take a look at the example in Listing 12, it shows a SWID tag for a hypothetical "Acme Enterprise Firewall" product. The tag includes a link to an external CycloneDX SBOM, which would contain the detailed inventory of components, dependencies, and vulnerabilities associated with that specific software. This illustrates how SWID tags can serve as a bridge between the high-level identity of a software product and the granular details provided by a BOM, but it does not replace the need for a comprehensive inventory format by any means.

Name	**Supported Data Formats**	**Developed by**	**BOM Scope**
CycloneDX	JSON, XML, Protobuf	OWASP Foundation	SBOM, SaaS-BOM, HBOM, AI-BOM, CBOM, OBOM, MBOM, BOV, VDR, VEX, CDXA
SPDX	JSON-LD, Turtle, N-Triples, RDF/XML	Linux Foundation	SBOM, AI-BOM, HBOM, License-BOM
SWID	XML	NIST	None

Table 2.1: Comparison of the considered BOM Standards and more

Chapter 3

Related Work

The design and implementation of efficient data models for context-aware computing have been the subject of extensive research. This thesis does not exist in a vacuum; rather, it builds significantly upon established architectural patterns defined in previous academic works. Specifically, the laying ground for part of the solution proposed in this document is built on top of the Context Discover Actuator Profile (CTXD) data model proposed by Tanzarella et al, in 2024 [3].

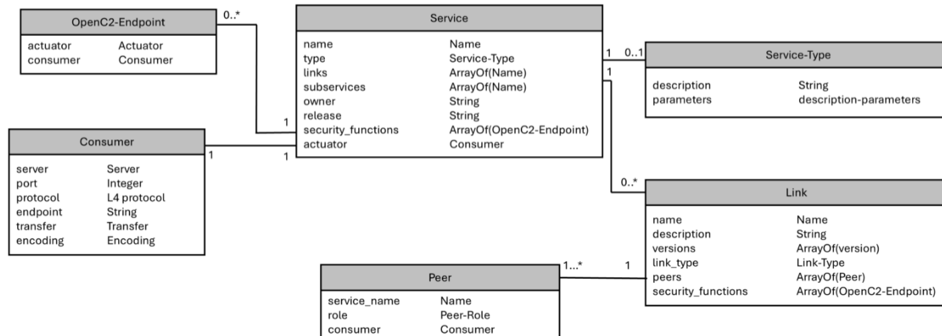

Figure 3.1: Context Discovery Actuator Profile data model, adapted from [3].

As illustrated in Figure 3.1, the core of the Context Discovery Actuator Profile data model revolves around the `Service` entity, which acts as the central node representing a discoverable context-aware application or resource. This entity hold a variety of attributes that help to define the characteristics of the same, such as it's name, it's owner and the release version. Furthermore, a "recursive" relationship can be estrablished via the `subservices` attribute, wheres a service type can be defined with the `ServiceType` entity, which includes a short description and the description of the parameters required to interact with the service.

Communication logistics are handled by the Consumer class. This entity encapsulates the necessary networking parameters to reach a service or endpoint, including the server address, port, Layer 4 protocol, the specific string endpoint, as well as transfer and encoding types. Every Service possesses ex-

actly one actuator of type Consumer, which serves as its primary entry point.

To facilitate context-aware network topologies, the model introduces the Link and Peer entities. A Service may maintain multiple connections represented by the Link class, which defines the link_type, supported versions, and its own set of security_functions. Each Link must connect to at least one Peer. The Peer class represents the remote entity in the connection, identifying it by service_name, assigning a specific role (Peer-Role), and specifying its consumer for direct communication.

This data model provides a structured framework for representing and managing context-aware services and their interactions, but has since been modified to make up to some of the major limitations of the original design.

Chapter 4

Thesis Objectives

The OpenC2 standard establishes a rigorous language for the direction of cyber defense components, decoupling the intent of a command from its low-level execution. However, this decoupling introduces a significant operational blind spot: the standard presupposes that the Producer possesses a priori knowledge of the environment it defends. In dynamic, cloud-native, or heterogeneous networks, the topology of resources, their dependencies, and their current state are fluid, and without this visibility, effective countermeasures can be difficult to implement, as security operators often lack the necessary contextual information to make informed decisions regarding the "blast radius" of a threat or the collateral impact of a defensive action.

The primary objective of this thesis is to bridge this gap by designing and implementing a new OpenC2 Actuator Profile dedicated to Context Discovery which leverages the concept of Bill of Materials to provide a structured and standardized method for representing and exchanging detailed inventory information about network nodes, including their hardware composition, software inventory, and service dependencies. This profile aims to enable an OpenC2 Producer not merely to command an endpoint to take action, but to interrogate it regarding its structural and operational context. By leveraging the extensible nature of the CycloneDX standard—specifically its capabilities as an XBOM, this thesis proposes a method to represent the complex environment of modern computing infrastructures in a standardized, machine-readable format that can be easily parsed and utilized by downstream tools for enhanced security and management capabilities.

To achieve this overarching goal, the work focuses on the formal design of the Context-Aware Actuator Profile. This involves defining a specification that adheres strictly to the OASIS OpenC2 Architecture. The profile maps standard OpenC2 Actions, such as `query`, to the generation and transmission of CycloneDX documents. Consequently, the Actuator provides a unified view of the endpoint without requiring complex negotiation, delivering a strictly typed and machine-readable snapshot of the system's internal and external relationships ready for immediate analysis by the Producer.

In parallel with the theoretical design, a significant objective of this thesis

is the concrete implementation of the proposed profile within the `otupy` framework. As the reference implementation for OpenC2 in Python, `otupy` provides a modular architecture suitable for extension. This work entails developing a new `BOMActuator` class capable of interfacing with underlying system tools to generate real-time BOMs. A single BOM provides the context for a single actuator, which can than be used for network or host visibility. By parsing the `service` and `externalReference` definitions within a received CycloneDX BOM, the Producer logic is designed to identify adjacent nodes and dependencies.

Furthermore, another objective of this thesis is to provide a representation of the current data model, derived from the original CTXD one, as a XBOM. This involves defining a mapping between the entities and relationships of the data model and the components of a CycloneDX BOM. This mapping is crucial to ensure that the generated BOMs accurately reflect the complex relationships and dependencies present in the environment in which the Actuator is deployed.

To validate the efficacy of the proposed solution, this thesis adopts a comprehensive approach that combines three key strategies: unit tests to verify the correctness of the implementation, a concrete use case in a realistic multi-layered networked environment to demonstrate practical applicability, and performance evaluation to quantify the operational overhead introduced by the solution. The validation process involves setting up a test bed with multiple interconnected nodes, each running the appropriate Actuator implementation. The Producer executes a series of queries to discover the network topology and the relationships between nodes, while performance measurements are collected to assess latency and resource usage. This approach ensures that the solution is not only correct and robust, but also efficient and suitable for real-world deployment.

Chapter 5

XBOM Profile Design and Implementation

5.1 Introduction

The main objective of this chapter is to present the design and implementation of the Extended Bill of Material (XBOM) Profile within the OpenC2 framework. This profile is intended to enhance the capabilities of OpenC2 by providing a structured method for representing and exchanging detailed inventory information about network nodes, including their hardware composition, software inventory, and service dependencies. The chapter is structured as follows: first, we will discuss the design of the XBOM profile, detailing its specifications and how it adheres to the OASIS OpenC2 Architecture and the CycloneDX specification. Next, the implementation of the profile within the `otupy` framework will be described, including the development of the necessary classes and methods to generate and handle the bill of materials, generated thanks to the `cyclonedx-python-lib` [18] library.

5.2 Profile Design

As discussed in Section 2.1, the OpenC2 standard is designed to be extensible through Actuator Profiles. While the standard currently defines profiles for firewalls (SPLF) and other defensive components like Intrusion Detection Systems (IDS), it still lacks support for many other types of actuators. It's still worth noting that currently, a rough draft for a Software Bill of Materials actuator profile is in the works [19], but none of the branch available in the GitHub repository of the OpenC2 standard contains a complete implementation of it, and have received contributions at all in over three years. Thus, it can be safely considered that the BOM Actuator Profile is still in its infancy, and that the work of this thesis can still be considered relevant since both the scope and the design of the profile are not defined yet, and the implementation has yet to start. Furthermore, the scope of the profile remains limited to just software inventory, while the design proposed in this thesis extends beyond just

that to include many more aspects of a network node and it's environment. For all those reasons, the design of a new profile should not be considered as a redundant effort, but rather as a complementary one, that can , eventually, be useful to the community as a reference for the design of the official BOM Actuator Profile.

To design a novel Actuator Profile, a systematic approach is required to ensure compatibility with the core OpenC2 language while addressing the specific needs of the extended capabilities. Once the specific goals of the profile – which in this case is the generation and handling of an Extended Bills of Material – have been established, the subsequent step involves identifying the specific data structures required to support these goals. If the required data structures are not present in the core OpenC2 language specifications, they must be rigorously defined within the context of the new Actuator Profile. The definition of a custom data type necessitates the selection of a suitable base type, such as an `Array`, `Map`, or `Choice`, accompanied by a semantic description of the data and any necessary conformance clauses.

Following the establishment of data types, the Actuator Profile specification allows for the introduction of novel Targets and Command Arguments. These elements are scoped specifically to the profile and are valid only when the Actuator supports the designated profile. While the Response messages may also utilize these custom data types to convey rich information back to the Producer, the OpenC2 specification strictly prohibits Actuator Profiles from defining new Actions; they must utilize the pre-defined set of Actions (e.g., `query`, `locate`, `get`) provided by the core language.

At last, a set of conformance clauses must be articulated to guarantee the deterministic behavior of the Actuator. These clauses dictate the permissible combinations of Actions and Targets, as well as the expected behavior of the Consumer when processing specific Command Arguments.

The following subsections detail the architecture of the XBOM Actuator Profile, including the command structure, as well as the response format.

5.2.1 Profile and Message Structure

The XBOM Actuator Profile is defined through a formal schema that extends the core OpenC2 language. According to Section 3.2 of the *OpenC2 Architecture Specification* [1], this profile specifies the unique Action-Target pairs, Arguments, and Response structures necessary for BOM exchange.

First of all, the language specifications defines the Command as the description of the action that has to be performed by the Consumer. The Command is composed of many fields, but the ones that are relevant for the design of the XBOM profile are:

- `action`: the type of action that the Producer wants the Consumer to perform.
- `target`: the specific entity or resource that the action is directed towards.

- `arguments`: additional parameters that may be required to execute the command.
- `actuator specifiers`: these are used to indicate that the command is intended for a specific Actuator Profile.

Another important aspect of the profile design is the structure of the Response messages. The Response is the message sent by the Consumer back to the Producer after processing a Command. For the XBOM profile, the Response is designed to include a field that contains the generated BOM, which is structured according to the CycloneDX specification. Furthermore, the standard OpenC2 Response structure is defined to include just a few fields:

- `status`: indicates the success or failure of the command execution.
- `status_text`: provides a human-readable description of the status.
- `results`: contains any data or information resulting from the command execution.

5.3 Command Structure

The command structure of the XBOM Actuator Profile is designed to leverage the existing OpenC2 Actions while introducing specific Targets and Arguments relevant to the generation and exchange of BOMs. In this section, the specific command structure for the XBOM profile will be detailed, including the definition of the Targets and Arguments that are unique to this profile.

5.3.1 Actions

As per the OpenC2 specifications, the XBOM Actuator Profile does not introduce new Actions but rather utilizes the existing set of Actions defined in the core language, in which it's also described as a mandatory field. For the purpose of this profile, the primary Action utilized is `query`. Since an OpenC2 Producer does not have any prior knowledge of the internal structure of the Consumer, as well of the security features of the system it is commanding, the `query` action is used to request information from the Consumer, such as the `features` of the system, or the `capabilities` of the Actuator. The `query` action, in this context, is also used to request the generation of a BOM, which is then sent back to the Producer in the Response message. As such, the only Action used in the XBOM profile is `query`, since the main purpose of the profile is to enable the Producer to query the Consumer for information about its environment. The Actions relevant to the XBOM Actuator Profile are summarized in table 5.1.

ID	Name	Description
3	query	Initiate a request for information from the Consumer, such as features, capabilities, or to request the generation of a BOM.

Table 5.1: Actions used in the XBOM Actuator Profile

5.3.2 Targets

In OpenC2's architecture, the **Target** is a mandatory component of every Command message. It explicitly identifies the entity, resource, or object to which an action applies. While standard Targets are defined by the core specification, OpenC2 allows the creation of custom Targets to accommodate specialized use cases, such as those required by the XBOM Actuator Profile.

A strict structural constraint governs Command definitions: each Command must include exactly one Target. The formal schema enforces this via the Target's definition as a Choice type, requiring the Producer to select one of the available Target structures. This ensures commands are syntactically and semantically valid.

The XBOM Actuator Profile introduces a new specific Target in addition to the `features` one, which is defined in the base specification, yet relevant to the scope of this profile: the `xbom` Target. They are designed to facilitate the querying of an Actuator's capabilities and the generation of a Bill of Materials, respectively. The detailed definitions of these Targets are provided in Table 5.2. The additional data types – which are not defined in the core OpenC2 specification – are also defined in the profile, such as `Xbom-ctx`, but will be explained in the following sections.

5.3.3 Xbom Arguments

To facilitate a highly granular, structured, and precise request, the `xbom` target utilizes a specific data structure known as the `xbom-ctx` map, which defines the specific arguments and constraints used to accurately identify, filter, or format the requested Bill of Materials, ensuring that the Consumer can interpret and execute the Producer's exact inventory requirements without ambiguity.

ID	Name	Type	Count	Description
1	`format`	`Xbom-format`	0..1	Optional specifier used to explicitly indicate the desired format of the XBOM data. If omitted, the Actuator defaults to generating the BOM in the CycloneDX format

Table 5.3: `Xbom-ctx` Map Elements defining the XBOM Context

The `format` field within this context inherently defaults to the CycloneDX standard, reflecting the main, and only, implementation focus of this profile.

ID	Name	Type	Count	Description
9	`features`	`Features`	1	Represents a comprehensive set of items intrinsically supported by the Actuator. This includes valid Action/Target operational pairs, supported profile versions, and specific configuration options. This Target is predominantly utilized in conjunction with the `query` Action to dynamically determine, enumerate, and map an Actuator's underlying operational capabilities before issuing complex commands.
10	`xbom`	`Xbom-ctx`	1	A highly specialized target specifically engineered for this profile. It is designed to request a robust, detailed description of the service and component environment, providing the necessary operational context required for the generation, filtering, or retrieval of a targeted Bill of Materials.

Table 5.2: Target Definitions for the XBOM Actuator Profile

However, this field is explicitly designed with future extensibility in mind, leaving the structural space open for the seamless integration of other prominent BOM formats, such as an improved version of SPDX, should the need arise. The `Xbom-format` data type is defined as an enumerated type, which allows for the precise specification of the desired BOM format in a standardized manner, ensuring that the Consumer can accurately interpret the Producer's request and generate the BOM in the correct format. A list of the currently defined values for this enumerated type is provided in Table 5.4.

ID	Name	Description
1	`cyclonedx`	Represents the CycloneDX standard, a lightweight, highly structured BOM specification explicitly designed for application security contexts, vulnerability identification, and comprehensive supply chain component analysis.

Table 5.4: `Xbom-format` Enumerated Values

5.3.4 Command Arguments

Beyond the specific Target context, the XBOM Actuator Profile heavily leverages Command Arguments to provide additional precision, control, and operational parameters to the Command execution. These arguments are encapsulated within a dedicated `Args` map and are absolutely instrumental in dictating how the Consumer should process the request, manage its resources, and structure its subsequent Response message.

ID	Name	Type	Count	Description
4	`response_requested`	`Response-Type`	0..1	Explicitly dictates the type and verbosity of the Response required from the Consumer for the given Action. Acceptable values include `none`, `ack`, `status`, and `complete`.
1024	`cached`	`Boolean`	0..1	A performance-tuning boolean flag. If True, the Actuator MAY return a previously generated, stored XBOM. If False, the Actuator MUST execute a fresh generation of the SBOM to accurately reflect the current system state.

Table 5.5: Command Arguments (`Args` Map) for the XBOM Profile

The `response_requested` argument is of particular importance as it allows the Producer to specify the desired level of feedback from the Consumer. This can range from no response at all (`none`), to an acknowledgment of command receipt (`ack`), to periodic status updates during command execution (`status`), or a final response upon command completion (`complete`). The specific values for this argument are defined in the `Response-Type` enumerated type, which

is detailed in Table 5.6.

ID	Name	Description
0	none	No response
1	ack	Respond when Command received
2	status	Respond with progress toward Command completion
3	complete	Respond when all aspects of Command completed

Table 5.6: `Response-Type` Enumerated Values [4]

At last, the `cached` argument provides a mechanism for performance optimization. By allowing the Producer to indicate whether a previously generated XBOM can be returned, it enables the Consumer to potentially reduce processing time and resource utilization, while still providing accurate and relevant information when necessary. This flag is particularly useful in testing scenarios, where the most up-to-date information may not be critical, and a cached response can significantly enhance responsiveness.

5.3.5 Actuator Specifiers

The Actuator Specifiers defined within the XBOM Profile provide the necessary addressing and identification constraints to ensure that a Command is directed to and processed by the correct entity. While the Target context defines what is being acted upon, the Actuator Specifiers define which specific instance or administrative grouping of a service should execute the Action. These specifiers are critical for maintaining operational integrity in environments where multiple actuators may be listening on the same control plane.

ID	Name	Type	Count	Description
1	`domain`	`String`	0..1	Specifies the distinct network, administrative, or operational domain that falls strictly under the responsibility and jurisdictional control of the target actuator.
2	`asset_id`	`String`	0..1	Provides a unique, unambiguous alphanumeric identifier for the specific instance of the actuator, ensuring the command is parsed and processed solely by the exact intended system.

Table 5.7: Actuator Specifiers for the XBOM Profile

The `domain` specifier allows the Producer to target a specific operational or administrative domain, which is particularly useful in large, segmented networks where multiple actuators may be present. By specifying the domain, the Producer can ensure that the command is directed to the correct segment of

the network, reducing the risk of unintended consequences and ensuring that the command is processed by the appropriate system that has the necessary context and authority to execute the requested action.

The `asset_id` specifier provides a critical layer of precision in environments where multiple actuators may be present. By including a unique identifier for the target actuator, the Producer can ensure that the command is processed by the correct instance, avoiding any potential confusion or misrouting that could arise in a multi-actuator environment. This is particularly important for the XBOM profile, as the generation of a Bill of Materials can be a resource-intensive operation that should only be executed by the intended system to ensure accurate and relevant information is returned to the Producer.

5.4 Response Structure

The Response structure of the XBOM Actuator Profile is designed to provide comprehensive feedback to the Producer regarding the execution of the Command, as well as to deliver the requested BOM data in a structured and standardized format. The Response message includes several key fields that convey the status of the command execution, any relevant information or data resulting from the execution, and the generated BOM itself. The following subsections detail the specific fields and their intended use within the context of the XBOM profile.

5.4.1 OpenC2 Response Message

As established, the Response is the message sent by the Consumer back to the Producer after processing a Command. This structure is fundamental for conveying the outcome of the requested Action and delivering the generated inventory data. As outlined in the core specification, the OpenC2 Response is implemented as a Map that consists of three primary fields, which are detailed in Table 5.8.

ID	Name	Type	Count	Description
1	`status`	`Status-Code`	1	An integer status code indicating the outcome of the Command execution.
2	`status_text`	`String`	0..1	A free-form, human-readable description of the Response status.
3	`results`	`Results`	0..1	Map of key:value pairs that contain additional results based on the invoking Command.

Table 5.8: OpenC2 Response Structure

The `status` field is a mandatory component that provides a machine-readable indication of the success, failure, or ongoing processing state of the Command execution. A list of the specific status codes relevant to the XBOM profile is provided in Table 5.9, those codes are quite standard for HTTP-like response status codes, and are defined in the OpenC2 Language Specification [4], allowing for a consistent interpretation of the Command execution outcome across different implementations and use cases.

The `status_text` field offers a human-readable description of the status, which can be useful for debugging and operational awareness. The `results` field is an optional map that can contain additional data or information resulting from the Command execution, such as the generated BOM in the case of the XBOM profile. This field allows the Consumer to return rich, structured information back to the Producer, enabling more informed decision-making and further actions based on the Command's outcome.

ID	Description
102	**Processing** - An interim Response used to inform the Producer that the Consumer has accepted the Command but has not yet completed it.
200	**OK** - The Command has succeeded.
400	**Bad Request** - The Consumer cannot process the Command due to something that is perceived to be a Producer error (e.g., malformed Command syntax).
401	**Unauthorized** - The Command Message lacks valid authentication credentials for the target resource or authorization has been refused for the submitted credentials.
403	**Forbidden** - The Consumer understood the Command but refuses to authorize it.
404	**Not Found** - The Consumer has not found anything matching the Command.
500	**Internal Error** - The Consumer encountered an unexpected condition that prevented it from performing the Command.
501	**Not Implemented** - The Consumer does not support the functionality required to perform the Command.
503	**Service Unavailable** - The Consumer is currently unable to perform the Command due to a temporary overloading or maintenance of the Consumer.

Table 5.9: `Status-Code` Enumerated Values [4]

5.4.2 Response Results

The `results` field within the Response message is a critical component for conveying additional information back to the Producer following the execution of a Command. In the context of the XBOM Actuator Profile, this field is particularly important for returning the generated Bill of Materials data, as

well as any relevant metadata about the Actuator's capabilities and supported features. The standard properties that can be included in the `results` map are defined in the OpenC2 Language Specification and are detailed in Table 5.10. These properties provide a structured

ID	Name	Type	Count	Description
1	`versions`	`Version`	0..n	List of OpenC2 language versions supported by this Actuator.
2	`profiles`	`ArrayOf(Nsid)`	0..1	List of profiles supported by this Actuator.
3	`pairs`	`Action-Targets`	0..1	List of targets applicable to each supported Action.
4	`rate_limit`	`int{0..n}`	0..1	Maximum number of requests per minute supported by design or policy.

Table 5.10: Standard `Results` Properties for Actuator Responses [4]

5.4.3 XBOM Result Extension

Finally, the XBOM Actuator Profile extends the standard `results` map to include a specific property for returning the generated BOM data. This property is defined as `xbom` and is designed to contain the structured BOM information generated by the Consumer in response to a Command targeting the `xbom` Target. The `xbom` property is a mandatory field when the Command execution is successful and a BOM is generated, ensuring that the Producer will always receive a result when requesting a BOM, even if no relevant data is found during the discovery process. The structure of the `xbom` property is defined as an `Xbom` data type, which won't be detailed in this section since it will be defined in the implementation chapter, but it is designed to encapsulate all the necessary information about the generated BOM, including the components, services, and dependencies discovered during the context discovery process. The specific definition of the `xbom` property within the `results` map is provided in Table 5.11.

ID	Name	Type	Count	Description
1	`xbom`	`Xbom`	1	The Xbom object, which includes all the services, components, and dependencies found during the context discovery process.

Table 5.11: Extended `Results` Property for the XBOM Profile

5.5 Handling XBOM Generation

With the structure of the Command and Response messages being defined, the next critical aspect of the XBOM Actuator Profile is the handling of the actual generation of the Bill of Materials. This process involves the collection of the various components, services, and dependencies within the Consumer's environment, as well as the organization and structuring of this information in a way that can be effectively returned to the Producer. The data model used for representing this inventory information is a key component of this process, as it provides the necessary framework for organizing the collected data and ensuring that it can be easily interpreted and utilized by the Producer. After the data model is defined, the Actuator must implement the logic for collecting the relevant information from the Consumer's environment, populating the data model with this information, and then converting it to the desired bill of material format before returning it in the Response message.

This section will detail all the in-scope steps necessary for the generation of the BOM.

5.5.1 Context Data Model

The data model of the XBOM Actuator Profile is designed to provide a comprehensive and structured representation of the inventory information collected from the Consumer's environment. This data model, courtesy of professor Matteo Repetto, is critical for ensuring that the information returned to the Producer is organized a structured way that can be easily interpreted and utilized for further decision-making and actions. This data model is also the foundation for generating the BOM in the desired format, since it encapsulates all the necessary information that will be included in the BOM.

Figure 5.1 illustrates the foundational data model of the XBOM Actuator Profile at it's latest iteration, which is designed to encapsulate the comprehensive inventory information collected from the Consumer's environment. This data model is made up of a few key entities that are not standard in the OpenC2 specification, and revolves around three foundational concepts: `Service`, `Link`, and `Peer`.

First and foremost, a `Service` represents any digital resource or functionality that can be uniquely identified and interconnected with other resources within the Consumer's environment. This broad definition include software applications, code repositories, underlying infrastructure, network components, physical or virtual devices, and essentially any asset that could potentially be subject to operational scrutiny or cyber-attacks. The structural description of a `Service` includes a set of attributes that provide critical metadata, including its logical `domain` and `namespace`, the designated `owner`, the specific `release` version.

To support an higher level of granularity and precision in the representation of services includes an `SId` entity, which operates as a comprehensive structural identifier. Where a standard `Name` is defined as a strictly mutually exclusive

choice between a `URI`, a `reverse-dns` hostname, a `UUID`, or a `local` string, the `SId` acts as a alternative. It mandates a primary `type` and `name`, while offering optional, highly descriptive fields for `subtype`, logical `domain`, `namespace`, and specific `version`. Furthermore, the hierarchical composition of a service is now defined via the `subservices` attribute as a list of `SId` objects, allowing for significantly richer metadata tracking of nested micro-components compared to rudimentary string names.

It is important to note that the most granular and relevant descriptive information is inherently embedded within the `ServiceType` attribute, as these details are highly specific to the functional nature of the service itself. This choice-based enumeration allows for a more precise and standardized categorization of services, which is essential for the accurate generation of the BOM and for ensuring that the Producer can effectively interpret the inventory data. A fundamental operational assumption of this model is that there is only one authoritative actuator responsible for providing the description of a given `Service` object, avoiding the risk of having multiple versions to merge.

The `ServiceType` enumeration groups the entities that can appear in a managed environment into distinct categories. Each type corresponds to a different kind of resource — infrastructure, runtime, connectivity, software, or network function — which together cover the full range of components typically found in modern networked systems.

A `Host` represents any infrastructure-level entity, physical or virtual, that provides computational resources. This includes bare-metal servers, Virtual Machines (VMs), Kubernetes Pods, and IoT devices, all further discriminated by the `HostType` sub-enumeration. The key distinction is that a `Host` is not a software artifact; its role is to provide the substrate on which an `ExecutionEnvironment` runs.

An `ExecutionEnvironment` is a software runtime context that provides the primitives needed to execute applications — a process space, a filesystem, a network slice, and the associated libraries and packages. The type covers a wide range of realizations: full Operating Systems, containers, Python virtual environments, chroot jails, and Linux network namespaces, among others. These are captured by the `ExecutionEnvironmentType` sub-enumeration (`os`, `container`, `netns`).

A `Network` represents any physical or virtual communication infrastructure, which ranges from Ethernet, IP, and VLANs to overlay protocols like VXLAN and VPN tunnels, and wireless technologies such as Wi-Fi, ZigBee, LoRaWAN, and 5G. IP is intentionally included as a generic network type for environments where no specific link layer is present or relevant. The concrete technology is selected via the `NetworkType` sub-enumeration.

An `Application` is simply a software program, which may be deployed and running inside an `ExecutionEnvironment`. The type is intentionally kept broad, with more specific details — such as version, owner, and subtype — carried as attributes of the object itself.

Finally, a `NetworkFunction` models any component whose primary role is

to process network traffic, whether for forwarding, address translation, or security enforcement. This covers both traditional hardware appliances and their virtualized equivalents (VNFs): routers, NAT gateways, bridges, and firewalls, all discriminated by the `NetworkFunctionType` sub-enumeration. Keeping this as a separate type makes the semantic difference explicit: a `NetworkFunction` mediates the flow of data, while an `Application` produces or consumes it.

To delineate the topology of the environment, a `Link` defines a concrete relationship between two or more services. The `Link` object structurally describes the connection through a primary `name`, a mandatory `sid` for extended structural identification, and a human-readable `description`. The relationship fundamentally exists between the currently described entity and an array of `peers`, which may be managed by the same actuator or an entirely different one. Similar to the service entity, the majority of the descriptive semantic elements characterizing the relationship are dictated by the `LinkType` attribute, while the `role` attribute strictly defines the perspective of the link itself. Furthermore, multiple peers can be provided within a single `Link` (e.g., multiple VMs hosted on the same hypervisor), though this does not preclude implementations from alternatively returning distinct `Link` objects for each peer relationship.

Each participant in this relationship is instantiated as a `Peer`. The `Peer` object associates the connection with a specific target using its mandatory `sid`, alongside a strictly defined `role` from the `PeerRole` enumeration. The concept of a peer dynamically extends beyond standard application components to include security functions managed by other OpenC2 profiles.

In scenarios where a peer is managed by a disparate actuator, the corresponding `Peer` object may include an optional `Consumer` object. This `Consumer` class is highly relevant for locating and interacting with the actuator responsible for external services or security functions. It provides the essential information required to reach the consumer endpoint, detailing both communication parameters, such as the `host` IP address, `port`, specific API `endpoint`, data `encoding`, and `transfer` protocol, as well as the specific OpenC2 `profile` and `actuator` specifiers. If a consumer lacks the knowledge of where to find a service description, or if no appropriate actuator is available (rendering it a hidden node in the chain), this `Consumer` object is simply omitted. It remains the explicit responsibility of the Producer orchestrating the discovery process to sequentially query these subsequent consumers to construct the complete service chain, as the model inherently assumes no centralized caching mechanism.

5.5.2 Adapting the Data Model for BOM Generation

The data model described in the previous section provides a comprehensive framework for representing the inventory information collected from the Consumer's environment. However, it's not really possible to directly convert this data model into a standard BOM format, such as CycloneDX or SPDX, with-

out some form of adaptation or transformation. This is for a few reasons, but the main one is that the data model was not really designed with the specific structure and requirements of a BOM format in mind. For instance, the data model defines it's own internal structure for representing services, links, and peers, which may not directly align with the way components and dependencies are represented in a standard BOM format. Additionally, the data model includes a lot of metadata and descriptive information that may not be relevant or necessary for the BOM itself. Those details are, however, critical for the internal processing and organization of the context discovery process, and as such, those concerns were prioritized in the design of the data model. Additionally, keeping the data model as a separate, internal representation whilst still addressing each specific requirement of the same data model would allow to reconstruct the same discovered context from it's bill of materials representation.

Profile Field	**CycloneDX Representation**
`application`	components.type.application
`cloud`	services
`computer`	components.type.platform
`consumer`	*.properties
`container`	components.type.container
`iot`	components.type.device
`link`	*.properties
`network`	services
`operating System`	components.type.operating-system
`peer`	*.properties
`pod`	components.type.platform
`server`	components.type.platform
`service`	services
`vm`	components.type.platform
`web_service`	services
`api`	services
`host`	components.type.*
`execution_environment`	components.type.platform
`network_function`	services

Table 5.12: Mapping of Profile Fields to CycloneDX

Table 5.12 provides a summary of the mapping between the fields defined in the XBOM data model and their corresponding representations in the CycloneDX BOM format. This mapping is not necessarily one-to-one, as some fields in the data model may need to be combined or transformed in order to fit into the structure of the BOM format. For instance, the various types of services and components defined in the data model may need to be categorized and represented as specific component types in CycloneDX, while the links

and peers may need to be represented as dependencies or properties within the BOM. The specific mapping is what will be discussed in the following subsection, as well as motivations for the design choices made in the adaptation process.

It's also important to notice that this thesis follows the release 1.7 of the CycloneDX specification, which is the latest release at the time of writing, and as such, the mapping is designed to be compatible with this specific version of the specification. Future releases of CycloneDX may introduce new fields or change the structure of the BOM format, which could require adjustments to the mapping in order to maintain compatibility.

Custom Taxomomy for Otupy-specific Fields

In the process of mapping the fields from the XBOM data model to the CycloneDX BOM format, it became apparent that there are certain fields in the data model that do not have a direct equivalent in the standard CycloneDX specification. These fields, which are specific to the XBOM data model and are critical for accurately representing the inventory information, needed to be accommodated in a way that would not interfere with the standard fields defined by CycloneDX. To address this issue, a custom taxonomy was created for these specific fields, using a dedicated namespace (otupy) to ensure that they can be included in the BOM without causing any conflicts with the standard fields. This approach allows for the necessary flexibility to include all the relevant information from the XBOM data model while still adhering to the structure and requirements of the CycloneDX BOM format. The use of a custom taxonomy also provides a clear distinction between the standard fields defined by CycloneDX and the additional metadata provided by the XBOM data model.

To better understand the role that those custom fields play in the mapping process, each custom field is designed to capture specific information that is not covered by the standard CycloneDX fields, but is essential for conveying accurate informations. As such, each of those custom fields' keys will present the `otupy` namespace prefix, followed by each specific field being prefixed with the relevant entity type. Furthermore, each component and service will store the specific internal type of the entity in the XBOM data model in a custom field named `otupy:type`, which will allow producers to easily identify the nature of the component or service being represented, even if it does not fit neatly into one of the standard CycloneDX component types.

For instance, an application's internal ID would be represented as `otupy:application:id`, while a peer's role would be represented as `otupy:peer:role`. This structured naming convention allows for a clear and organized representation of the custom fields, making it easier for producers to understand the nature of the information being conveyed and how it relates to the overall inventory data. Additionally, this approach ensures that the custom fields can be easily identified and processed by any tools or systems that are designed to work with the CycloneDX BOM format, while still providing the necessary

flexibility to include all the relevant information from the XBOM data model.

This taxonomy was also designed with keeping possible future registration of the custom fields in mind, while still ensuring that the mapping is flexible enough to accommodate potential changes or additions to the data model or the BOM format in the future. A custom taxonomy can be registered by submitting a pull request to a dedicated repository [1].

Application

In the context of the XBOM data model, an `application` is defined as a *software* program or suite of programs that provide specific functionality to users or other systems. In the CycloneDX BOM format, applications are typically represented as components with a specific type designation. The mapping of an `application` from the XBOM data model to CycloneDX would involve categorizing it as a component and assigning it the type "application". This allows for a clear and standardized representation of applications within the BOM, making it easier for producers to understand the nature of the component and its role within the overall system.

Attribute	Mapped to
description	components:*:description
name	components:*:name
version	components:*:version
id	components:*:properties:otupy:application:id
owner	components:*:properties:otupy:application:owner
app_type	components:*:properties:otupy:application:type

Table 5.13: Mapping of Profile Fields to CycloneDX

Table 5.13 summarizes the specific mapping of the attributes defined for an instance of an `application` in the XBOM data model to their corresponding representations in the CycloneDX BOM format. The `description`, `name`, and `version` attributes are mapped to the standard component fields in CycloneDX, while the `id`, `owner`, and `type` attributes are mapped to custom properties within the component, using a specific namespace (otupy) to avoid conflicts with standard fields.

[1] `https://cyclonedx.github.io/cyclonedx-property-taxonomy/`

```json
 {
   "bomFormat": "CycloneDX",
   "specVersion": "1.7",
   "serialNumber": "urn:uuid:3e671687-395b-41f5-a30f-a58921a69b79",
   "version": 1,
   "metadata": {
       "timestamp": "2020-08-03T08:53:09.834Z"
   },
   "components": [
       {
           "type": "application",
           "name": "iptables",
           "version": "1.8.10",
           "description": "generic description",
           "properties": [
               {
                   "name": "otupy:type",
                   "value": "application"
               },
               {
                   "name": "otupy:application:id",
                   "value": "1"
               },
               {
                   "name": "otupy:application:owner",
                   "value": "NetFilter"
               },
               {
                   "name": "otupy:application:type",
                   "value": "Packet Filtering"
               }
           ]
       }
   ]
}
```

Listing 13: Example of an Application Component in CycloneDX with Custom Taxonomy

Listing 13 provides an example of how an instance of an `application` from the XBOM data model can be represented in the CycloneDX BOM format, including the use of the custom taxonomy. The example shows a component of type "application" with a name, version, and description, as well as custom properties that include the application ID, owner, and type, all using the "otupy" namespace to avoid conflicts with standard CycloneDX fields. This example also includes the necessary metadata necessary for the CycloneDX

format, such as the BOM format version, specification version, serial number.

Execution Environment

In the context of the XBOM data model, an `execution environment` is defined as a set of software resources that allow for the execution of an application. This can include various subsystems such as a process ID space, a filesystem, a network slice, and other resources that are necessary for running an application. The execution environment can take various forms, such as a full operating system, a container, a Python virtual environment, a chroot, or other types of isolated environments. With it's definition being one of a runtime environment, it's actually fits perfectly within the concept of a "platform" component in the CycloneDX specifications.

This class it's to be inteded as a bundle to hold together common attributes and metadata that are relevant for both the `Container` and `Operating System` classes, and as such, it's worth noting that description of this class will only be used, in practice, if the underlying type representation is not provided by the consumer. As such, it will be used as a sort of fallback for the edge cases in which the consumer is not able to provide a more specific description of the execution environment (via it's subtype parameter), but still wants to provide some information about the runtime environment of the application.

Attribute	Mapped to
libs	components:*:components:*
pkgs	components:*:components:*
apps	components:*:components:*
name	components:*:name
id	components:*:properties:otupy:execenv:id
description	components:*:description

Table 5.14: Attribute mapping for ExecutionEnvironment in CycloneDX

Table 5.14 summarizes the mapping of the attributes defined for an instance of an `execution environment` in the XBOM data model to their corresponding representations in the CycloneDX BOM format. The `libs`, `pkgs`, and `apps` attributes, which represent the various libraries, packages, and applications that are part of the execution environment, are mapped to the components field in CycloneDX, meaning that they will be represented as subcomponents of the execution environment component, or the corresponding type one.

Host

In the context of the XBOM data model, a `host` is defined as an environment that provides resources to an execution environment. As previously mentioned, the host component has replaced the previous concept of a server to provide a more generic representation of anything an execution environment can be

hosted on. A host can be any physical or virtualised environment that provides resources such as disks, network cards, CPUs, memory, GPUs, and other hardware peripherals. This can include a wide range of different types of hosts, such as servers, IoT devices, virtual machines, Kubernetes pods, and other types of environments that are designed to host an execution environment. The combination of hosts and execution environments creates a recursive hierarchy of dependencies, where execution environments are contained within hosts, and hosts may themselves be implemented within execution environments.

With the definition of a host being so broad and generic, it's not really possible to directly map this specific component to a specific type in the CycloneDX BOM format, since it could potentially encompass a wide variety of different types of components, depending on the specific implementation and the context in which it is used. As such, the responsibility of categorizing the host component into a specific type in the CycloneDX format is left to the specific `type` attribute of the host component. The for possible values that is currently defined for this attribute are "server", "vm", "iot", and "pod". This means that a host component with the type "server" would be represented as a component with the type "platform" in CycloneDX, while a host component with the type "iot" would also be represented as a component with the type "device" in CycloneDX, and so on for the other types. This approach allows for a flexible representation of the host component in the CycloneDX BOM format, which can then be enriched with the specific attributes and metadata from the host component in the XBOM data model. You can see the specific mapping of the attributes of the host component to the CycloneDX format in table 5.15.

Attribute	**Mapped to**
id	components:*:properties:otupy:host:id
vendor	components:*:properties:otupy:host:vendor
model	components:*:properties:otupy:host:model
release	components:*:properties:otupy:host:release
serial	components:*:properties:otupy:host:serial
firmware	components:*:properties:otupy:host:firmware
version	components:*:properties:otupy:host:version
type	components:*:properties:otupy:host:type
name	components::name
description	components::description

Table 5.15: Attribute mapping for Host in CycloneDX

Container

In the context of the XBOM data model, a `container` is defined as a software image that is executed within a sandboxed environment, such as a Linux namespace or similar technology. A container provides an isolated execution

environment that includes its own subsystems, such as network interfaces, file systems, and other resources. While the specific subsystems and resources that are part of the container model are not explicitly defined within the data model, they are implicitly included as part of the container service.

Attribute	**Mapped to**
description	component:*:description
id	component:*:properties:otupy:container:id
name	component:*:name
namespace	component:*:otupy:container:namespace
status	component:*:otupy:container:status
image	component:*:otupy:container:image

Table 5.16: Attribute mapping for Container in CycloneDX

When mapping a container from the XBOM data model to the CycloneDX BOM format, it would be represented as a component with the type "container". The specific attributes of the container, such as its description, name, namespace, status, and image, would be mapped to the corresponding fields in the CycloneDX format, with custom properties using the "otupy" namespace to include additional metadata specific to the container.

Operating System

In the context of the XBOM data model, an `operating system` is defined as a common execution environment that can run a wide variety of software applications. An operating system typically includes a full set of libraries and applications, as well as various subsystems such as file systems, network interfaces, and other resources that are necessary for the execution of software applications. When mapping an operating system from the XBOM data model to the CycloneDX BOM format, it would be represented as a component with the type "operating-system" since it's already provided as a standard type in the CycloneDX specification.

Attribute	**Mapped To**
name	components:*:name
version	components:*:version
family	components:*:properties:otupy:os:family
arch	components:*:properties:otupy:os:arch

Table 5.17: Attribute mapping for Operating System in CycloneDX

The attributes of the operating system, are handled in the same fashion as the application and container components, with the attributes that are available in the standard CycloneDX format (name and version) being mapped to the corresponding standard fields, while the additional metadata specific to

the operating system (family and type) is included as custom properties using the "otupy" namespace as required by the CycloneDX specification to handle non standard fields.

IoT Device

In the context of the XBOM data model, an `IoT device` is defined as a, usually, physical device that is connected to a network to carry out specific functions or tasks. CycloneDX does have a specific component type for devices, which is "device", so an IoT device from the XBOM data model would be represented as a component with the type "device" in the CycloneDX BOM format since it falls under the standard's definition of a device, that being "A hardware device such as a processor or chip-set" [20]. The specific attributes of the IoT device, as you can see in table **??**, would be mapped to the corresponding fields in the CycloneDX format, with the custom properties included as extension properties using the "otupy" namespace to include additional metadata specific to the IoT device.

Attribute	**Mapped to**
description	components:*:description
name	components:*:name
type	components:*:properties:otupy:iot:type
vendor	components:*:properties:otupy:iot:vendor
model	components:*:properties:otupy:iot:model
serial	components:*:properties:otupy:iot:serial
firmware	components:*:properties:otupy:iot:firmware
version	components:*:version

Table 5.18: Attribute mapping for IoT Device in CycloneDX

Virtual Machine

In the context of the XBOM data model, a `virtual machine` is defined as a virtualisation environment that emulates a full computer hardware. A virtual machine provides virtualised hardware resources such as network interfaces, virtual CPUs, virtual RAM, and storage. Since a virtual machine shares many components with any other network host, it can be represented in the CycloneDX BOM format as a component with the type "platform", which is a standard component type in CycloneDX. The specific attributes of the virtual machine would be mapped to the corresponding fields in the CycloneDX format, with custom properties included as extension properties using the "otupy" namespace to include additional metadata specific to the virtual machine.

Attribute	Mapped to
hypervisor	components:*:properties:otupy:vm:hypervisor
hypervisor-type	components:*:properties:otupy:vm:hypervisor_type
image	components:*:properties:otupy:vm:image
id	components:*:properties:otupy:vm:id
vendor	components:*:properties:otupy:vm:vendor
model	components:*:properties:otupy:vm:model

Table 5.19: Attribute mapping for Virtual Machine in CycloneDX

Server

In the context of the XBOM data model, a `server` is defined as a physical computing hardware that provides real hardware resources such as network interfaces, CPUs, RAM, and storage. A server can be represented in the CycloneDX BOM format as a component with the type "platform", which is a standard component type in CycloneDX. Currently, this class is just a legacy class from a previous representation of the data model, as it was intended to be a bundle of a serial number, a model, a vendor and a firmware. However this concept is only reported here for completeness, as this is to be considered a legacy type that is not really relevant anymore.

Cloud

In the context of the XBOM data model, a `cloud` follows the general definition of a cloud service, which is a service that is provided over the internet and is typically hosted on a remote server or a cluster of servers. A cloud service can include a wide range of different types of services, such as infrastructure as a service, platform as a service, software as a service and so on. This definition makes mapping a cloud service from the XBOM data model to the CycloneDX BOM format a bit more complex, since there is no specific component type in CycloneDX that directly corresponds to a cloud service. Furthermore, a cloud infrastructure, unlike the other definitions that were given in the previous sections, can be also interpreted as a service that is being consumed by the overall system, rather than a component that is part of the system itself. However, in most use cases tested during the design and implementation of the profile, some instances of IaaS services should have been described differently: take for instance a OpenStack deployment, which is a common IaaS solution, the various components of the OpenStack deployment, such as the block storage service (Cinder), the compute service (Nova), the networking service (Neutron), and so on, would be described as separate components in the CycloneDX BOM format, but are ultimately part of the same cloud infrastructure while still being a foundational part of the cloud service itself. However, this description of the cloud infrastructure as a collection of components is not always true: in the case of a Kubernetes cloud infrastructure, the service is typically consumed as an unified api rather than a collection of separate components, and as such, it

would be more appropriate to represent the Kubernetes cloud infrastructure as a single service in the CycloneDX BOM format, rather than a assembly of separate components.

As such, the mapping of a cloud service from the XBOM data model to the CycloneDX BOM format would involve representing the cloud service as both a service that is provided and consumed by the overall system, and as the composition of the underlying infrastructure components and services that make up the cloud service if appropriate.

Listing 14 provides an example of how a cloud service from the XBOM data model can be represented in the CycloneDX BOM format. Unlike a standard dependency relationship, the cloud service can optionally be represented as an assembly of the underlying infrastructure services that make up the cloud service itself. In the example, the cloud service is represented as a service with the internal type "cloud", which are expressed through the use of custom properties using the "otupy" namespace, and it includes a provider field that indicates the specific cloud service being used (in this case, OpenStack). The cloud service also includes a list of services that represent the underlying infrastructure services that make up the cloud service, such as the Cinder block storage service, the Glance image service. Those service, being described as a constituent and fundamental part of the cloud service itself, are not represented as dependencies of the cloud service, but rather as subservices that are provided by the cloud service itself, while also being part of the same.

Attribute	**Mapped to**
name	services:*:name
type	services:*:properties:otupy:cloud:type
id	services:*:properties:otupy:cloud:id

Table 5.20: Attribute mapping for Cloud in CycloneDX

Pod

In the context of the XBOM data model, a `pod` is defined as a logical unit in Kubernetes that is used to run one or more containers. A pod provides a shared execution environment for the containers that are hosted within it, with the containers sharing the same network interface but having their own process ID and filesystem namespaces. The pod is more of a management unit than a true isolation environment, as the containers within the same pod can communicate with each other over the shared network interface. Additionally, it is able to allocate real resources for it's containers — a shared network and filesystem mounts — which is exactly what makes it a `Host` in the data model.

In CycloneDX it is mapped to a component with type "platform". In CycloneDX, a platform is a runtime environment — hardware, software, or both — on which other components run. A pod fits that description: it is the namespace-based environment inside which containers execute. This also keeps

```json
{
  "bomFormat": "CycloneDX",
  "specVersion": "1.7",
  "serialNumber": "urn:uuid:3e671687-395b-41f5-a30f-a58921a69b79",
  "version": 1,
  "metadata": {
    "timestamp": "2020-08-03T08:53:09.834Z"
  },
  "services": [
    {
      "bom-ref": "cloud:none/none/none/openstack@none",
      "name": "openstack@regionone.cloud.tnt-lab.local",
      "properties": [
        {
          "name": "otupy:cloud:type",
          "value": "iaas"
        },
        {
          "name": "otupy:service:owner",
          "value": "tnt-lab"
        },
        {
          "name": "otupy:type",
          "value": "cloud"
        }
      ],
      "provider": {
        "name": "openstack"
      },
      "services": [
        {
          "bom-ref": "api:none/none/none/cinderv3@none",
          "name": "cinderv3@regionone.cloud.tnt-lab.local",
          "endpoints": [
            "http://controller.cloud.tnt-lab.local:8776/v3/%7bproject_id%7d"
          ],
          "properties": [
            {
              "name": "otupy:type",
              "value": "api"
            }
          ]
        },
        {
          "bom-ref": "api:none/none/none/glance@none",
          "name": "glance@regionone.cloud.tnt-lab.local",
          "endpoints": [
            "http://controller.cloud.tnt-lab.local:9292"
          ],
        },
        {
          "bom-ref": "api:none/none/none/keystone@none",
          "name": "keystone@regionone.cloud.tnt-lab.local",
          "endpoints": [
            "http://controller:5000/v3/"
          ],
        }
      ]
    }
  ]
}
```

Listing 14: Example of a Cloud Service in CycloneDX with Custom Taxonomy

it consistent with other `Host` subtypes such as VMs and servers, which map to the same type for the same reason. The `namespace` field (the Kubernetes namespace the pod belongs to) has no standard CycloneDX equivalent, so it is stored as the custom property `otupy:pod:namespace`.

Attribute	**Mapped to**
name	components:*:name
namespace	components:*:properties:otupy:pod:namespace
id	components:*:properties:otupy:pod:id
vendor	components:*:properties:otupy:pod:vendor
model	components:*:properties:otupy:pod:model

Table 5.21: Attribute mapping for Pod in CycloneDX

Web Service

In the context of the XBOM data model, a `web service` is defined as a service that is accessed over the web, typically using standard web protocols via it's own APIs and endpoints. A web service can include a wide range of different types of services, such as RESTful APIs, SOAP services, GraphQL APIs, and so on. When mapping a web service from the XBOM data model to the CycloneDX BOM format, it would be represented as a service with the type "service", which is a standard service type in CycloneDX. The specific attributes of the web service would be mapped to the corresponding fields in the CycloneDX format, with custom properties included as extension properties using the "otupy" namespace to include additional metadata specific to the web service, which are shown in table 5.22.

Attribute	**Mapped to**
name	services:*:name
description	services:*:description
endpoints	services:*:endpoints
port	services:*:properties:otupy:webservice:port
owner	services:*:properties:otupy:webservice:owner

Table 5.22: Attribute mapping for Property in CycloneDX

This is another example of a legacy type that is not really relevant anymore, since the concept of a web service is already encompassed by the more general concept of an API, which is defined in the next section. However, this is still included here for completeness, as it was part of the original data model design and implementation, and it can still be used in some specific use cases where a more specific definition of a web service is needed.

API

In the context of the XBOM data model, an `API` is defined as an abstraction of any application programming interface that can be invoked over the network. This can include a wide range of different types of APIs, such as older-style remote procedure calls (RPCs), newer web service APIs such as RESTful APIs, SOAP services, GraphQL APIs, and other custom interfaces and APIs that may be used in different architectures and protocols. Given the broad scope and heterogeneous terminology used to describe different types of APIs, the definition of an API in the XBOM data model is intentionally broad and may be extended and refined in the future to better capture the different alternatives and variations of APIs that are used in different contexts. When mapping an API from the XBOM data model to the CycloneDX BOM format, it would be represented as a service with the type "service", which is a standard service type in CycloneDX. This is mostly due to the fact that APIs are typically accessed over the network and provide specific functionality to users or other systems, which can consume the API as a service to access the functionalities provided by the same. The specific attributes of the API would be mapped to the corresponding fields in the CycloneDX format, as usual, with custom properties included as extension properties using the "otupy" namespace to include additional metadata specific to the API.

Attribute	**Mapped to**
name	services:*:name
description	services:*:description
endpoints	services:*:endpoints
provider	services:*:properties:otupy:api:provider
id	services:*:properties:otupy:api:id
type	services:*:properties:otupy:api:type

Table 5.23: Attribute mapping for Property in CycloneDX

Table 5.23 summarizes the mapping of the attributes defined for an instance of an `API` in the XBOM data model to their corresponding representations in the CycloneDX BOM format. The `name`, `description`, and `endpoints` attributes are mapped to the standard service fields in CycloneDX, while the `provider` attribute, which represents the specific provider of the API, is mapped to a custom property using the "otupy" namespace to include additional metadata specific to the API. It's also worth noting that the specific field `endpoints` of the API component, which represents the various endpoints through which the API can be accessed, presents it's own mapping as a custom property extension with the `otupy:api:endpoints` prefix.

Table 5.24 summarizes the mapping of the attributes defined for an instance of an `endpoint` in the XBOM data model. As you can see, all the attributes are mapped as a custom property extension. Furthermore, and id template is used in the property name to allow for multiple endpoints to be represented

Attribute	Mapped to
description	services:*:properties:otupy:*:endpoint:{id}:description
endpoint_type	services:*:properties:otupy:*:endpoint:{id}:type
transport	services:*:properties:otupy:*:endpoint:{id}:transport
transfer	services:*:properties:otupy:*:endpoint:{id}:transfer
encoding	services:*:properties:otupy:*:endpoint:{id}:encoding
uri	services:*:properties:otupy:*:endpoint:{id}:uri
provider	services:*:properties:otupy:*:endpoint:{id}:provider

Table 5.24: Attribute mapping for Endpoint in CycloneDX

for the same API while still being able to distinguish between the different endpoints attributes.

Network

In the context of the XBOM data model, a `Network` is defined as an abstraction of any logical or physical network segment that can be described as part of the infrastructure inventory. The concept is intentionally broad and is designed to accommodate a wide range of different network technologies and architectures, from classical layer-2 Ethernet segments and IP subnets to virtual overlay networks such as VLANs and VXLANs, point-to-point virtual links, tunnels, and mobile cellular networks. The `Network` class inherits from `XBOMObject`, which provides the common `name`, `id`, and `description` fields shared by all top-level XBOM objects. The specific technology is captured through a single `type` field, which holds an instance of `NetworkType`, a discriminated union that selects the appropriate concrete subtype at runtime based on a short string key. This design allows the same `Network` wrapper to carry richly typed, technology-specific data without requiring separate top-level classes for every network technology.

When mapping a `Network` instance to the CycloneDX BOM format, it is represented as a `Service` component. This choice reflects the fact that, from the perspective of a running system, network infrastructure is typically consumed as a service by the hosts and applications that depend on it. The top-level attributes of `Network` are mapped as described in Table 5.25.

Attribute	Mapped to
name	services:*:name
description	services:*:description
id	services:*:properties:otupy:{type}:id
type	services:*:properties:otupy:type

Table 5.25: Attribute mapping for `Network` in CycloneDX

The `name` and `description` attributes are mapped to the standard CycloneDX service fields, while `id` and `type` are stored as custom extension prop-

erties using the `otupy` namespace. The value of `otupy:network:type` is the discriminator key that identifies which concrete network subtype is present (e.g., `ip`, `eth`, `vlan`, `5G`).

NetworkType and its subtypes `NetworkType` is the discriminated union used to carry the technology-specific details of a `Network` instance. The currently implemented subtypes cover the most common network technologies: `IPNetwork` for generic IP subnets, `EthernetNetwork` for layer-2 segments, `VLANNetwork` for 802.1Q virtual LANs, `VEthNetwork` for Linux veth pairs, `TunnelNetwork` for generic tunnel interfaces, and `MobileNetwork` for cellular deployments. `VXLANNetwork` is partially implemented. A number of additional keys — including `802.11`, `802.15`, `zigbee`, `vpn`, `lorawan`, and `wan` — are reserved as stubs for technologies that are planned but not yet modelled in detail.

The following paragraphs describe each implemented subtype together with its CycloneDX mapping.

IPNetwork `IPNetwork` is the most generic network subtype and is intended to be used when the only information available about the underlying infrastructure is a set of IP address prefixes, without any knowledge of the concrete layer-2 or overlay technology in use. Its sole attribute is `nets`, an ordered list of `IPNetAddress` values. `IPNetAddress` is itself a discriminated union that transparently wraps either an `IPv4Net` or an `IPv6Net` CIDR prefix, with automatic detection of the address family from the string representation; in both cases it serializes to a single `address` property. Table 5.26 shows the mapping.

Attribute	Mapped to
otupy:type	ip_network
nets[{i}].address	services:*:properties:otupy:ip_network:{i}:address

Table 5.26: Attribute mapping for `IPNetwork` in CycloneDX

EthernetNetwork `EthernetNetwork` extends the generic IP network concept with Ethernet-specific addressing. Table 5.27 shows the full mapping.

Attribute	Mapped to
otupy:type	ethernet_network
nets[{i}].address	services:*:properties:otupy:ethernet:{i}:address

Table 5.27: Attribute mapping for `EthernetNetwork` in CycloneDX

VLANNetwork `VLANNetwork` models a Virtual Local Area Network segment as defined by IEEE 802.1Q and related standards. Beyond the network

address list it carries three additional identifying fields: a human-readable `name`, a numeric `vlan_id` that uniquely identifies the segment within a switching domain, and a `type` string recording the VLAN encapsulation protocol (e.g., `802.1Q`, `QinQ`). Table 5.28 details the mapping.

Attribute	Mapped to
otupy:type	vlan_network
name	services:*:properties:otupy:vlan:name
vlan_id	services:*:properties:otupy:vlan:id
type	services:*:properties:otupy:vlan:type
nets[{i}].address	services:*:properties:otupy:vlan:{i}:address

Table 5.28: Attribute mapping for `VLANNetwork` in CycloneDX

VXLANNetwork `VXLANNetwork` models a Virtual eXtensible LAN overlay network as defined by RFC 7348. VXLAN encapsulates layer-2 Ethernet frames inside UDP datagrams, enabling logical layer-2 segments that span layer-3 boundaries in data-centre and cloud environments. The subtype encodes two technology-specific identifiers: the `vni` (VXLAN Network Identifier), a 24-bit value that distinguishes up to approximately 16 million independent overlay segments, and the `port` field recording the UDP destination port used for encapsulation (IANA-assigned default: 4789). At the time of writing, `VXLANNetwork` does not yet implement an `as_cyclonedx()` method; its CycloneDX serialization is therefore a planned extension and the expected property mapping, following the conventions established by the other subtypes, is shown in Table 5.29.

Attribute	Mapped to
otupy:type	vxlan_network
vni	services:*:properties:otupy:vxlan:vni
port	services:*:properties:otupy:vxlan:port
nets[{i}].address	services:*:properties:otupy:vxlan:{i}:address

Table 5.29: Planned attribute mapping for `VXLANNetwork` in CycloneDX

VEthNetwork `VEthNetwork` abstracts a Linux *veth* (virtual Ethernet) pair as a point-to-point pseudo-network. A veth pair consists of two virtual network interfaces that act as the two ends of a virtual Ethernet cable; traffic sent into one end emerges from the other, and the construct is commonly used to connect network namespaces in container and virtual-machine environments. The interface names are stored in the `peers` tuple, with one indexed property emitted per peer, and `nets` holds the IP prefixes assigned to the link. Table 5.30 details the mapping.

Attribute	Mapped to
otupy:type	veth_network
peers[{i}]	services:*:properties:otupy:veth:peer:{i}
nets[{i}].address	services:*:properties:otupy:veth:{i}:address

Table 5.30: Attribute mapping for `VEthNetwork` in CycloneDX

TunnelNetwork `TunnelNetwork` abstracts a generic tunnelled or VPN-style point-to-point link. The distinguishing attribute is `server`, which holds the address or hostname of the tunnel endpoint (e.g., a VPN concentrator or an OpenVPN server). In the CycloneDX representation the `server` value is used as the service name when available, providing a meaningful human-readable label for the component, and is also recorded as a dedicated property. Table 5.31 summarises the mapping.

Attribute	Mapped to
otupy:type	tunnel_network
server	services:*:name and services:*:properties:otupy:tunnel:server
nets[{i}].address	services:*:properties:otupy:tunnel:{i}:address

Table 5.31: Attribute mapping for `TunnelNetwork` in CycloneDX

MobileNetwork `MobileNetwork` models a mobile cellular network, primarily targeting 4G/LTE and 5G NR deployments. It is the richest of all implemented subtypes in terms of domain-specific metadata. The `mcc` (Mobile Country Code) and `mnc` (Mobile Network Code) fields together form the PLMN (Public Land Mobile Network) identifier that globally and uniquely identifies the network operator. The `region` integer encodes the geographic or administrative region to which the network instance belongs, while `sst` is a further partitioning field reserved for future use. Address information is provided through a list of addresses. Table 5.32 details the full mapping.

Attribute	Mapped to
otupy:type	mobile_network
name	services:*:name
mcc	services:*:properties:otupy:mobile:mcc
mnc	services:*:properties:otupy:mobile:mnc
region	services:*:properties:otupy:mobile:region
sst	services:*:properties:otupy:mobile:sst
nets[{i}]	services:*:properties:otupy:mobile:nets{i}

Table 5.32: Attribute mapping for `MobileNetwork` in CycloneDX

NetworkFunction

In the context of the XBOM data model, a `NetworkFunction` is defined as an abstraction of any network-layer processing element that handles, forwards, or filters network packets. This encompasses both traditional, dedicated hardware appliances — such as physical routers, switches, firewalls, and NAT devices — and their software-defined or virtualised counterparts, including Network Virtual Functions (NVFs) running inside containers, Linux namespaces, or general-purpose virtual machines. The same data model therefore applies uniformly across legacy and cloud-native network infrastructure.

The data model for this class includes the typical `name`,`id`, and `description`, found in most class of the XBOM data model . In addition to those, it introduces a `version` field to record the firmware or software release of the function, and a `type` field that holds an instance of `NetworkFunctionType`, a discriminated union that selects the appropriate concrete subtype at runtime. When mapping a `NetworkFunction` instance to the CycloneDX BOM format it is represented as a `Service` component, consistently with the `Network` mapping convention. The top-level attribute mapping is summarised in Table 5.33.

Attribute	Mapped to
name	services:*:name
description	services:*:description
id	services:*:properties:otupy:netfunc:id
version	services:*:properties:otupy:network_function:version
type	services:*:properties:otupy:netfunc:type

Table 5.33: Attribute mapping for `NetworkFunction` in CycloneDX

NetworkFunctionType and its subtypes `NetworkFunctionType` is the discriminated union used to carry the function-specific details of a `NetworkFunction` instance. The currently implemented subtypes cover the most common network processing functions: `Router` for IP routing, `NAT` for network address translation, `Bridge` for Ethernet bridging, and `Firewall` for packet filtering. The `vpn` type is reserved as a stub for future extension.

The following paragraphs describe each subtype in detail, together with its CycloneDX property mapping.

Router `Router` models an IP routing function. At the time of writing its internal structure is still being defined: it currently exposes a single `routes` field, intended as a placeholder for a future structured routing table whose individual entry format has not yet been finalised. The CycloneDX mapping is shown in Table 5.34.

NAT `NAT` models a Network Address Translation function. Its `rules` field is an list, intended as a placeholder for a future structured rule format whose

Attribute	Mapped to
otupy:type	router
routes	services:*:properties:otupy:router:routes

Table 5.34: Attribute mapping for `Router` in CycloneDX

individual entry structure has not yet been finalised. Each rule string in the list is serialised as a separately indexed property. Table 5.35 details the mapping.

Attribute	Mapped to
otupy:type	nat
rules[{i}]	services:*:properties:otupy:nat:rule:{i}

Table 5.35: Attribute mapping for `NAT` in CycloneDX

Bridge `Bridge` models an Ethernet bridge or software switch. Its `table` attribute is a placeholder for the MAC forwarding table, and `ifaces` lists the physical or virtual ports attached to the bridge as `NetworkInterface` entries. Each entry captures the interface name, MAC address, an optional identifier, a description, and the IP addresses assigned to the interface. Table 5.36 summarises the mapping.

Attribute	Mapped to
otupy:type	network_bridge
table	services:*:name and services:*:properties:otupy:bridge:table
ifaces (count)	services:*:properties:otupy:bridge:iface_count
ifaces[{i}].description	services:*:properties:otupy:bridge:iface:{i}: description
ifaces[{i}].id	services:*:properties:otupy:bridge:iface:{i}:id
ifaces[{i}].iface	services:*:properties:otupy:bridge:iface:{i}:iface
ifaces[{i}].mac	services:*:properties:otupy:bridge:iface:{i}:mac
ifaces[{i}].ips[{j}].ip	services:*:properties:otupy:bridge:iface:{i}:ip:{j}: ip
ifaces[{i}].ips[{j}].prefix	services:*:properties:otupy:bridge:iface:{i}:ip:{j}: prefix
ifaces[{i}].ips[{j}].gw	services:*:properties:otupy:bridge:iface:{i}:ip:{j}: gw

Table 5.36: Attribute mapping for `Bridge` in CycloneDX

Firewall `Firewall` models an IP packet-filtering function. At the time of writing it is still a placeholder: it exposes a single `routes` attribute intended

to eventually hold a structured firewall rule set, but whose entry format has not yet been finalised. The CycloneDX mapping is shown in Table 5.37.

Attribute	**Mapped to**
otupy:type	fw
routes	services:*:properties:otupy:fw:routes

Table 5.37: Attribute mapping for `Firewall` in CycloneDX

Link

In the context of the XBOM data model, a `Link` is defined as a directed relationship between two or more services. `Link` models the *semantic* dependency or interaction that exists between two context services: service calls which API, which service hosts another, which network carries which traffic flow, and similar relationships. The general structure of a `Link` instance has already been provided in Section 5.5.1.

Given this structure, a link can be interpreted as a logical association between a *source* service (the subject) and one or more *target* services (the peers), where the source and target(s) may be the same or different services, and the relationship may be unidirectional or bidirectional. As such it's unsuitable to be represented with the generic dependency mechanism provided by the `dependencies` field of a CycloneDX component, which is extensively used for modelling the sub service relationship between XBOM services. Instead, `Link` is mapped as a custom property extension on the service components representing the subject and target services.

Table 5.38 summarises the attribute mapping.

Attribute	**Mapped to**
(link_id)	services:*:properties:otupy:link:id
description	services:*:properties:otupy:link::{link_id}::desc
role	services:*:properties:otupy:link::{link_id}::role
link_type	services:*:properties:otupy:link::{link_id}::type
peers[{i}]	services:*:properties:otupy:link::{link_id}::peer:* (see Table 5.39)

Table 5.38: Attribute mapping for `Link` in CycloneDX

Peer A `Peer` identifies one of the object-side participants of a `Link`. It carries a `sid` (`SId`) to resolve which service is being referenced, an optional `role` (`PeerRole`) declaring the peer's role in the relationship, and an optional `consumer` (`Consumer`) that provides the informations about the actuator which will actually consume the produced XBOM. When serialised, the effective service identifier used in the property path is the `sid` value. Table 5.39 details the mapping.

Attribute	**Mapped to**
sid / service_name	otupy:link::{link_id}::peer:sid
role	otupy:link::{link_id}::peer::{sid}::role
consumer.host	otupy:link::{link_id}::peer:{sid}:consumer:host
consumer.port	otupy:link::{link_id}::peer:{sid}:consumer:port
consumer.protocol	otupy:link::{link_id}::peer:{sid}:consumer: protocol
consumer.endpoint	otupy:link::{link_id}::peer:{sid}:consumer: endpoint
consumer.transfer	otupy:link::{link_id}::peer:{sid}:consumer: transfer
consumer.encoding	otupy:link::{link_id}::peer:{sid}:consumer: encoding
consumer.profile	otupy:link::{link_id}::peer:{sid}:consumer:profile
consumer.actuator[*key*]	otupy:link::{link_id}::peer:{sid}:consumer: actuator:{key}

Table 5.39: Attribute mapping for `Peer` and `Consumer` in CycloneDX

Consumer The `Consumer` record embedded inside a `Peer` provides the concrete connection parameters needed to reach a referenced service over the network. It is the representation of the informations needed to reach the consumer: `host` holds the hostname or IP address of the target service, `port` the TCP/UDP port, `protocol` the layer-4 protocol (`L4Protocol`), and `endpoint` the URL path (e.g., `/.well-known/openc2`). The `transfer` and `encoding` fields record the OpenC2 transfer binding and message encoding in use, while `profile` identifies the OpenC2 profile implemented by the consumer. Finally, `actuator` is an optional dictionary of profile-specific actuator specifiers. All consumer attributes are serialised as extension properties nested under the peer prefix, as shown in Table 5.39. It is worth noting that, as a consequence of this nesting, the full depth of a link property key can become quite long, since it combines the link identifier, the peer service identifier, and the consumer field name into a single property path.

5.6 Implementation

With the profile definition and CycloneDX mapping established, the next step is implementing the profile in code. This involves creating a class to hold xbom data, modifying the automated discovery process to return a CycloneDX-compliant BOM, and enabling each data model class to provide its own BOM representation according to the previous mapping. The implementation uses Python and the `cyclonedx-python-lib` library to construct and serialize BOM objects to a json format.

5.6.1 Profile Registration

The Otupy framework uses a registration mechanism to connect profiles, targets, arguments, and results into a coherent OpenC2 namespace. Each profile is identified by a namespace identifier, as previously mentioned, and the extension convention mandated by the OpenC2 Language Specification requires that custom profiles carry the prefix x-. Therefore, the xbom profile is registered under the identifier x-xbom through the framework's @extension decorator, which makes the profile discoverable at runtime without any hard coded lookup tables.

In addition to the profile itself, the framework requires that all profile-specific types be explicitly registered: the Args class extends the base OpenC2 argument type with the only custom profile-specific Boolean field (cached), which controls whether to use a cached result or re-run the discovery before responding. The Results class extends the base OpenC2 results type by adding a xbom field of the abstract Xbom type, which is the top-level container that carries the assembled bill of materials in a response. The target type, XbomCtx, is registered under the name xbom within the profile's namespace and accepts an optional format field (of type XbomFormat) to allow the consumer to request a specific BOM format.

Together, these registrations allow the framework to correctly route incoming OpenC2 commands, validate them against the profile definition, encode and decode the payloads, and dispatch them to the appropriate actuator — all without any manual wiring by the developer writing the actuator.

5.6.2 BOM Format Abstraction

One of the design goals of the implementation is to isolate format concerns from discovery concerns: actuators should be written in terms of the context data model, and the translation to a specific BOM standard should be encapsulated elsewhere. To achieve this, the BOM container is introduced as a two-level hierarchy — an abstract base class that defines a uniform interface, and one or more concrete subclasses that implement that interface for a specific serialization format.

The Abstract Interface

The abstract base class Xbom inherits jointly from the framework's Record type and Python's ABC metaclass. Inheriting from Record is necessary because Xbom instances are transferred over the network as part of OpenC2 responses: the framework must be able to serialize and deserialize them using the same encoder infrastructure used for all other profile types, which is a necessary feature since the Xbom class is referenced in the Results type describe in section 5.4.1: the Results class has an xbom field of type Xbom, so the framework must be able to handle Xbom objects as part of the normal otupy encoding and decoding process. Inheriting from ABC enforces that no instance of Xbom can be

created directly, and that every concrete subclass provides implementations for the abstract methods.

The class declares two fields: `format`, of type `XbomFormat`, which records which BOM standard the instance represents; and `bom`, typed as `Any`, which holds the format-specific BOM object. The `Any` type is intentional: the abstract class must not import format-specific libraries, both to avoid forcing unnecessary dependencies on consumers of the interface and to keep the module's import graph clean. The concrete subclass is the only place where library-specific types appear.

The abstract interface mandates four methods. `add(item)` inserts a single item into the BOM, accepting any Python object; the concrete implementation is responsible for dispatching on the type of the item. `find_ref_by_name(name)` returns the stable `bom_ref` identifier assigned to a named entry, or `None` if no such entry exists. `add_dependency (parent_ref, child_ref)` records a structural dependency between two BOM entries identified by their `bom_ref` values. `add_dependency_with_external_ref(depends_on_xbom, from_ref)` handles the cross-BOM case where the depended-upon entry lives in a different BOM instance, combining an external reference (using the CycloneDX bom-link URI format) with a dependency record.

The module-level dictionary `_XBOM_FORMAT_REGISTRY` maps `XbomFormat` values to their corresponding concrete class. It is populated at import time by each concrete subclass, and therefore grows automatically whenever a new format implementation is added to the project, without requiring any changes to existing code. The registry is the mechanism through which format-aware polymorphic deserialization is implemented: the overridden `fromdict()` class method, shown in Listing 15, inspects the `format` field of the incoming dictionary and delegates to the registered concrete class.

The guard `if clstype is Xbom` ensures that this custom logic only runs when the framework attempts to deserialize directly to the abstract type. When a concrete subclass calls `fromdict()`, the condition is false and the method falls through to the standard `Record` deserialization path, avoiding infinite recursion. The import of `CyclonedxXbom` is placed inside the method body rather than at the module level to avoid a circular import: the concrete class imports the abstract one, so the reverse import must be deferred.

The only currently supported format is CycloneDX, identified by the `XbomFormat.cyclonedx` enumeration value. This enumerated type is defined as a profile-level type and is used consistently wherever a BOM format specifier is expected — in the `XbomCtx` target, in the `Xbom` class itself, and in the actuator layer through the format registry.

The CycloneDX Concrete Implementation

The concrete class `CyclonedxXbom` implements the abstract interface for the CycloneDX format. Its `bom` field is typed as the `Bom` class from `cyclonedx-python-lib`, and all format-specific library calls are confined to this class.

```
@classmethod
def fromdict(clstype, dic, encoder):
    if clstype is Xbom:
        if isinstance(dic, dict) and 'format' in dic:
            format_value = dic.get('format')
            if isinstance(format_value, str):
                format_enum = XbomFormat[format_value]
            elif isinstance(format_value, int):
                format_enum = XbomFormat(format_value)
            elif isinstance(format_value, XbomFormat):
                format_enum = format_value
            else:
                format_enum = XbomFormat.cyclonedx  # Default

            concrete_class = _XBOM_FORMAT_REGISTRY.get(format_enum)
            if concrete_class is not None:
                return concrete_class.fromdict(dic, encoder)

        # Default to CyclonedxXbom if format not specified or not found
        from otupy.profiles.xbom.data.xbom import CyclonedxXbom
        return CyclonedxXbom.fromdict(dic, encoder)

    return Record.fromdict(dic, encoder)
```

Listing 15: Polymorphic deserialization in `Xbom.fromdict()`.

The `add()` method is implemented as a type dispatch chain. If the incoming item is a CycloneDX `Component` or `Service`, it is added directly to the corresponding set in the underlying `Bom` object. If it is a list, each element is processed recursively. Otherwise, if the object exposes an `as_cyclonedx()` method, that method is called and the result is passed back into `add()` recursively. This last branch is what allows any xbom profile class to be added to the BOM by a simple call, without the caller needing to know how it maps to CycloneDX types. This feature comes with the benefit of keeping the actuator code clean and focused on discovery logic, while the BOM container handles all the format-specific mapping concerns.

The `add_dependency()` method encodes a structural dependency relationship between two BOM entries, identified by their `bom_ref` values. Before creating the dependency, the method verifies that both the parent and the child actually exist in the BOM, raising an exception if either is missing. This guards against programming errors in actuators that might attempt to record a dependency involving a service that was never successfully added. This mechanism was necessary to adapt to the existing service hierarchy of the XBOM data model: each service holds a list of its sub-service identifiers, and `add_dependency()` makes it possible to record that hierarchy as a set of explicit dependencies in the BOM, which in this context have been mapped as a

```
def add(self, item):
    if isinstance(item, Component):
        self.bom.components.add(item)
        return
    if isinstance(item, Service):
        self.bom.services.add(item)
        return
    if isinstance(item, list):
        for p in item:
            if isinstance(p, Property):
                self.add(p)
        return
    if hasattr(item, "as_cyclonedx"):
        self.add(item.as_cyclonedx())
        return
    raise TypeError(
        f"Cannot add item of type {type(item)} to XBOM."
    )
```

Listing 16: Type dispatch in `CyclonedxXbom.add()`.

generic *depends-on* relationship.

Cross-BOM dependencies, where a service in one BOM depends on a service discovered by a different actuator running on a different host, are handled by the `add_dependency_with_external_ref()` method. This method generates a CycloneDX *bom-link* URI for the referenced entry, following the format `urn:cdx:{serial-number}/{version}#{bom-ref}`, and records it both as an external reference on the dependent item and as a dependency relationship. Bom-links are the official CycloneDX mechanism for referencing entries across BOM boundaries, and their use here allows a merged view — assembled later by the `_merge_discovery` utility — to represent inter-host dependencies without duplicating the referenced entries.

The serialization and deserialization flow for `CyclonedxXbom` is illustrated in Figure 5.2. Every type that travels over the network as part of an OpenC2 message must implement `todict(e)` and `fromdict(cls, dic, e)`, where `e` is an `Encoder` instance that drives the recursive conversion of the whole message tree. The contract of `todict()` is to return a plain Python `dict` — never a string — because the encoder accumulates the entire message as a nested Python object first, and only at the very end serializes the whole thing to the wire format (JSON, CBOR, etc.) in a single pass. Returning a string from inside `todict()` would break this contract: the encoder would treat it as an opaque scalar and would not be able to apply wire-format encoding to its contents.

The CycloneDX library's own serializer, `JsonV1Dot7`, produces a JSON string as its native output. `serialize()` therefore calls the library and then

immediately parses the resulting string back into a Python `dict` via `json.loads()`. This round-trip is intentional: it helps addressing the mismatch between what the CycloneDX library produces (a JSON string, unfortunately) and what the otupy encoder expects (a Python `dict`). The resulting `dict` is what `todict()` places under the `bom` key alongside the `format` identifier, so that when the encoder finally serializes the full OpenC2 response, the CycloneDX content is encoded together with the rest of the message in one consistent pass.

Deserialization reverses this exactly. The otupy encoder calls `fromdict(cls,dic,e)` with a `dict` that already contains the parsed JSON. `fromdict()` reconstructs the `XbomFormat` value, instantiates the concrete class, then passes the `bom` sub-dictionary to `deserialize()`. `deserialize()` converts the `dict` back to a JSON string with `json.dumps()` so that it can be passed to `JsonStrictValidator`, which validates the document against the CycloneDX 1.7 schema before the library parses it into a live `Bom` object via `Bom.from_json()`.

```
def todict(self, e):
    return {
      'format': e.todict(self.format) if self.format else None,
      'bom': self.serialize() if self.bom else None
    }

def serialize(self) -> dict:
    serializer = JsonV1Dot7(self.bom)
    return json.loads(serializer.output_as_string())

def deserialize(self, data: dict | str) -> None:
    validator = JsonStrictValidator(
        schema_version=_cyclonedx_schema_version
    )
    data = data if isinstance(data, str) else json.dumps(data)
    if validator.validate_str(data):
        raise ValueError("Invalid CycloneDX JSON data")
    self.bom = Bom.from_json(json.loads(data))
```

Listing 17: Serialization and deserialization in `CyclonedxXbom`.

The `todict()` and `fromdict()` methods integrate `CyclonedxXbom` into the otupy serialization infrastructure, as shown in Listing 17. The presence of the `format` key inside the outer dict is what enables the polymorphic `fromdict()` logic in the abstract `Xbom` class to reconstruct the correct concrete type on the receiving end, without the caller needing to know which BOM format was used.

5.6.3 Actuator Implementation

The xbom profile's actuator layer is organized as a class hierarchy rooted at `XBOMActuator`. The base class provides all the logic that is common across environments: receiving and validating OpenC2 commands, managing the format registry, orchestrating BOM assembly, and returning properly structured responses. Concrete subclasses are responsible only for populating two lists — `self.services` and `self.links` — with instances of the xbom profile data types. Everything else is handled by the base class.

Base Class Structure

The constructor accepts a standardized set of keyword arguments: `auth` carries credentials for external APIs; `config` provides additional endpoint or behavioural parameters; `peers` holds a pre-configured list of `Consumer` objects representing known services that may appear as link endpoints but that the local actuator cannot discover independently; `owner` identifies the owner of the resources being discovered; and `specifiers` describes the actuator's own identity, used to determine whether an incoming command is addressed to this instance.

A format registry at the module level maps each `XbomFormat` value to the corresponding `Xbom` concrete class. The `create_bom()` factory method consults this registry to instantiate the correct BOM type at runtime:

```python
def create_bom(self) -> Xbom:
    bom_class = _BOM_REGISTRY.get(self.sbom_format)
    if bom_class is None:
        raise NotImplementedError(
            f"SBOM format {self.sbom_format} is not supported"
        )
    return bom_class()
```

Listing 18: `create_bom()` factory method in `XBOMActuator`.

Concrete actuators are registered with the framework by decorating their class with `@actuator_implementation`, passing a string identifier (e.g. `"xbom-host"`, `"xbom-docker"`, `"xbom-kubernetes"`). This identifier appears in the configuration file to associate a service endpoint with the actuator responsible for querying it, keeping configuration fully decoupled from code.

Command Handling

When an OpenC2 command arrives it is received by the `run()` method, which first validates it against the profile's allowed action-target pairs and argument set, then optionally checks whether the actuator specifier in the command matches this instance's own `asset_id`, and finally dispatches to the appropriate handler. The dispatch logic is straightforward:

```python
def run(self, cmd):
    if not xbom.validate_command(cmd):
        return Response(status=StatusCode.NOTIMPLEMENTED,
                        status_text='Invalid Action/Target pair')
    if not xbom.validate_args(cmd):
        return Response(status=StatusCode.NOTIMPLEMENTED,
                        status_text='Option not supported')
    try:
        if not self.__is_addressed_to_actuator(cmd.actuator.getObj()):
            return Response(status=StatusCode.NOTFOUND,
                            status_text='Requested Actuator not available')
    except AttributeError:
        pass  # No actuator specifier: execute unconditionally

    match cmd.action:
        case Actions.query:
            response = self.query(cmd)
        case _:
            response = self.__notimplemented(cmd)
    return response
```

Listing 19: Command validation and dispatch in `XBOMActuator.run()`.

The only currently supported action is `query`, which is routed to two internal handlers depending on the target type. If the target is `Features`, `_query_feature()` returns the profile identifier, supported OpenC2 version, and allowed action-target pairs, following the requirements of the OpenC2 Language Specification. If the target is `XbomCtx`, `_query_sbom()` triggers the discovery and BOM assembly pipeline described below.

Discovery and BOM Generation

The general flow of the discovery and BOM generation process is shown in Figure 5.3.

The `_query_sbom()` handler first inspects the `XbomCtx` target's optional `format` field and, if present, stores it in `self.sbom_format` so that the correct BOM class is instantiated later. It then checks the `cached` argument: if `False` or absent, it calls `_update()`, which resets all internal state and runs a fresh discovery cycle; if `True`, the previously assembled BOM is returned directly, avoiding a potentially expensive re-discovery.

`_update()` is a thin orchestrator that delegates the environment-specific work to the concrete subclass via `discover_context()`, then calls `_build_bom()` to assemble the result:

`_build_bom()` is where the flat lists produced by discovery are translated into a structured CycloneDX BOM. It proceeds in three sequential phases. In the first phase a fresh `Xbom` instance is created via `create_bom()` and each

```
def _update(self):
    self.bom = None
    self.services = ArrayOf(Service)()
    self.links = ArrayOf(Link)()
    self.discover_context()
    self._build_bom()
```

Listing 20: The `_update()` method in `XBOMActuator`.

service in `self.services` is passed to `self.bom.add()`, which internally calls the service's `as_cyclonedx()` method and inserts the result into the underlying CycloneDX object. Services whose `type` field is `None` are skipped with a warning, since a type is required to determine the correct CycloneDX mapping.

In the second phase the subservice hierarchy is encoded as CycloneDX dependency relationships. Each service that carries a non-empty `subservices` list generates one `add_dependency()` call per entry, using the parent service's `SId` string as the parent reference and the subservice `SId` string as the child reference. This maps the compositional structure of the xbom data model onto the CycloneDX dependency graph as a set of *depends-on* relationships.

In the third phase links are attached to their owning services via `_add_link_to_bom()`. The method locates the corresponding service in the BOM by matching the link's `SId` against the `SId` of each service in `self.services`. The `SId` is designed to be globally unique in the actuator context and environment-agnostic, making it the natural stable identifier for this lookup. In cases where no match is found, a warning is logged and the link is silently dropped from the BOM. This is a pragmatic choice to avoid failing the entire BOM generation process due to a mismatch between the two lists. Furthermore, CycloneDX properties are scoped to individual entries rather than to the BOM as a whole, making this approach keeps relationship metadata co-located with the service that owns it.

```
def _add_link_to_bom(self, link: Link) -> None:
    for service in self.services:
        if (service.sid is not None and
                str(service.sid) == str(link.sid)):
            self.bom.add_link(item_ref=str(service.sid), link=link)
            return
    logger.warning(
        "Could not find service/component '%s' to add link", link.name
    )
```

Listing 21: Name-then-SId fallback matching in `_add_link_to_bom()`.

Because CycloneDX properties are scoped to individual entries rather than to the BOM as a whole, this approach keeps relationship metadata co-located with the service that owns it.

Once `_build_bom()` returns, `_query_sbom()` wraps the assembled `Xbom` instance in an `xbom.Results` object and embeds it in an OpenC2 `Response` with status code `OK`. The framework's encoder then serializes the result: the BOM is fully embedded in the response body under the `xbom` field, alongside the `format` identifier that allows the receiving side to reconstruct the correct `Xbom` subclass during deserialization.

5.6.4 Example: OpenC2 Request and Response

The following example illustrates a complete OpenC2 exchange with a `xbom-file` actuator. The file actuator is a static mockup that reads `Service` and `Link` definitions directly from its configuration, making it useful for representing components that do not expose a query-able API. In this scenario the configuration describes a cloud Kubernetes cluster containing three pods.

Request

The consumer sends a `query xbom` command. The target is `XbomCtx` with the format set to `cyclonedx`. The `cached` argument is explicitly set to `false`, so the actuator resets its internal state and rebuilds the BOM before responding. The actuator specifier narrows the command to the specific `xbom-file` instance by its `asset_id`.

```
{
  "action": "query",

  // The xbom target requests a BOM in CycloneDX format
  "target": {
    "x-xbom:xbom": {
      "format": "cyclonedx"
    }
  },

  // cached: false forces a fresh discovery run
  "args": {
    "cached": false
  },

  // The actuator specifier addresses this specific file actuator instance
  "actuator": {
    "x-xbom": {
      "asset_id": "xbom-file-cloud-example"
    }
  }
}
```

Listing 22: OpenC2 `query xbom` request to the file actuator.

Response

The actuator calls `discover_context()`, which copies the pre-loaded services and links into `self.services` and `self.links`. The base class then calls `_build_bom()`, which adds each service to the BOM. The assembled `CyclonedxXbom` is returned in the `xbom` field of the `Results` object, serialized as a dictionary with the `format` identifier and the full CycloneDX BOM document, and embedded in the standard OpenC2 response envelope.

```
{
  "status": 200,
  "status_text": "OK",

  "results": {
    "xbom": {

      // Format identifier, used by the receiver to pick the correct
      // Xbom subclass during deserialization
      "format": "cyclonedx",

      // The full CycloneDX BOM document, serialized at schema version
      ↪  1.7
      "bom": {
        "bomFormat": "CycloneDX",
        "specVersion": "1.7",
        "serialNumber":
        ↪  "urn:uuid:4f3a1c2e-9b7d-4e8a-bc01-2d5f6e7a8b9c",
        "version": 1,

        // The cloud cluster is a CycloneDX Service: cloud providers
        ↪  are
        // abstract service entities in the xbom data model and do not
        // map to a physical component
        "services": [
          {
            "bom-ref": "cloud:gcp/None/None/gke-cluster",
            "name": "gke-cluster",
            "description": "Google Kubernetes Engine cluster",
            "provider": { "name": "gke-cluster" },
            "properties": [
              // otupy:type identifies the xbom data model class
              { "name": "otupy:type",       "value": "cloud" },
              // otupy:cloud:type carries the cloud provider subtype
              { "name": "otupy:cloud:type", "value": "gcp"   }
            ]
          }
        ],

        // No components and no dependencies for a single-service BOM
```

```
        "components":    [],
        "dependencies":  []
      }
    }
  }
}
```


Listing 23: OpenC2 response carrying a single cloud service in CycloneDX format.

The cloud cluster is represented as a CycloneDX `Service`, reflecting the fact that cloud providers are modelled as abstract service entities in the xbom data model. Custom properties prefixed with `otupy:` carry the type information and identifiers that belong to the xbom data model but have no direct equivalent in the CycloneDX schema. The `bom-ref` is derived from the service's `SId`, making it both human-readable and stable across discovery runs for the same resource.

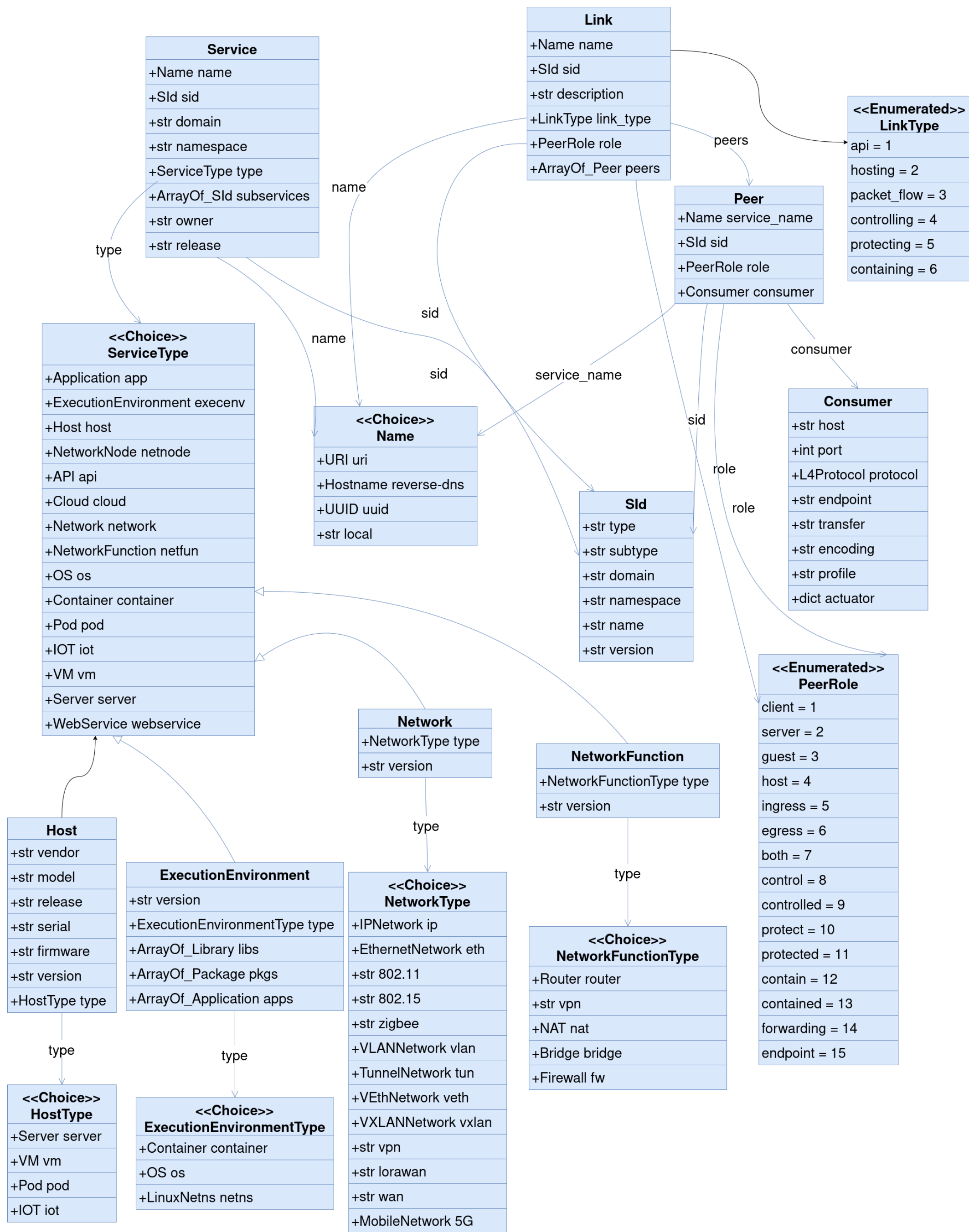

Figure 5.1: Data Model used for the XBOM Actuator Profile

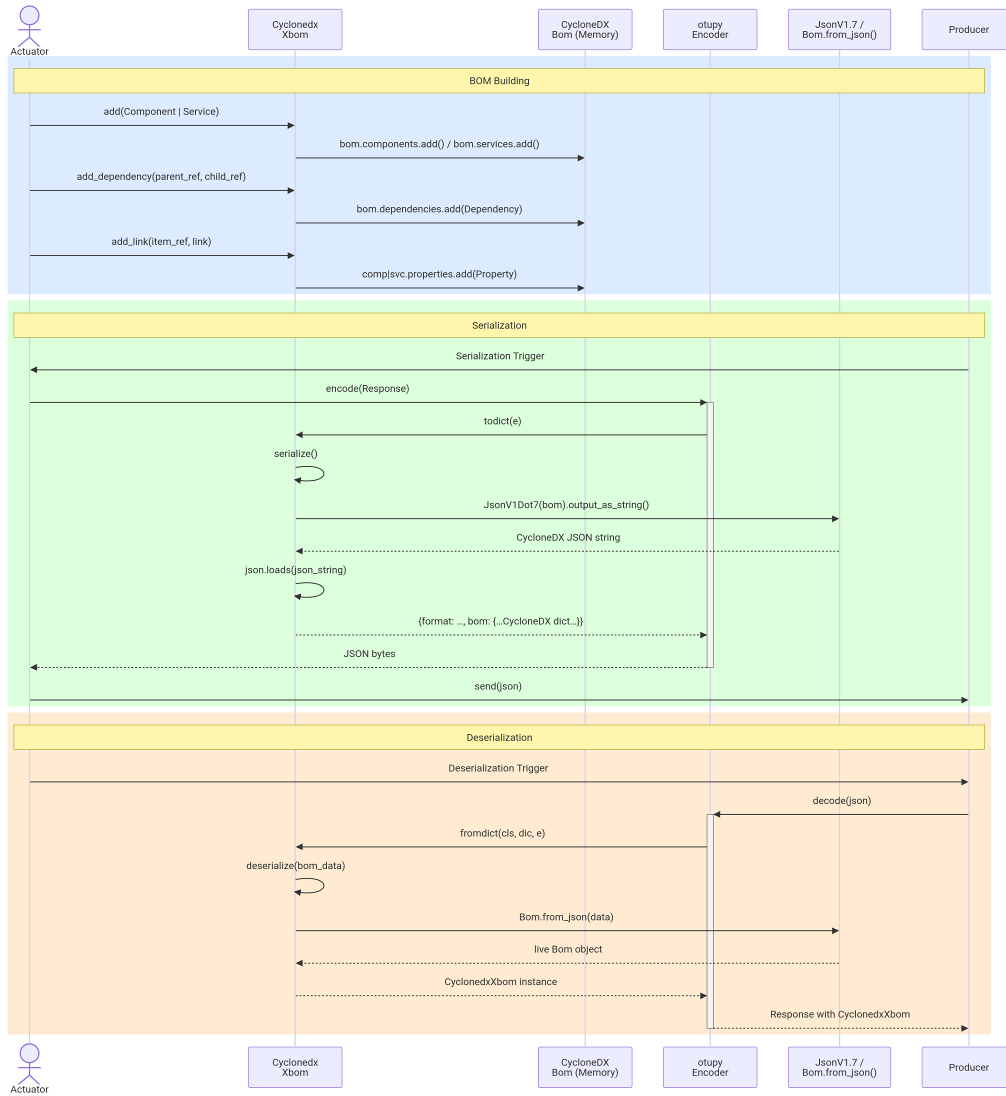

Figure 5.2: Serialization and deserialization flow for `CyclonedxXbom`.

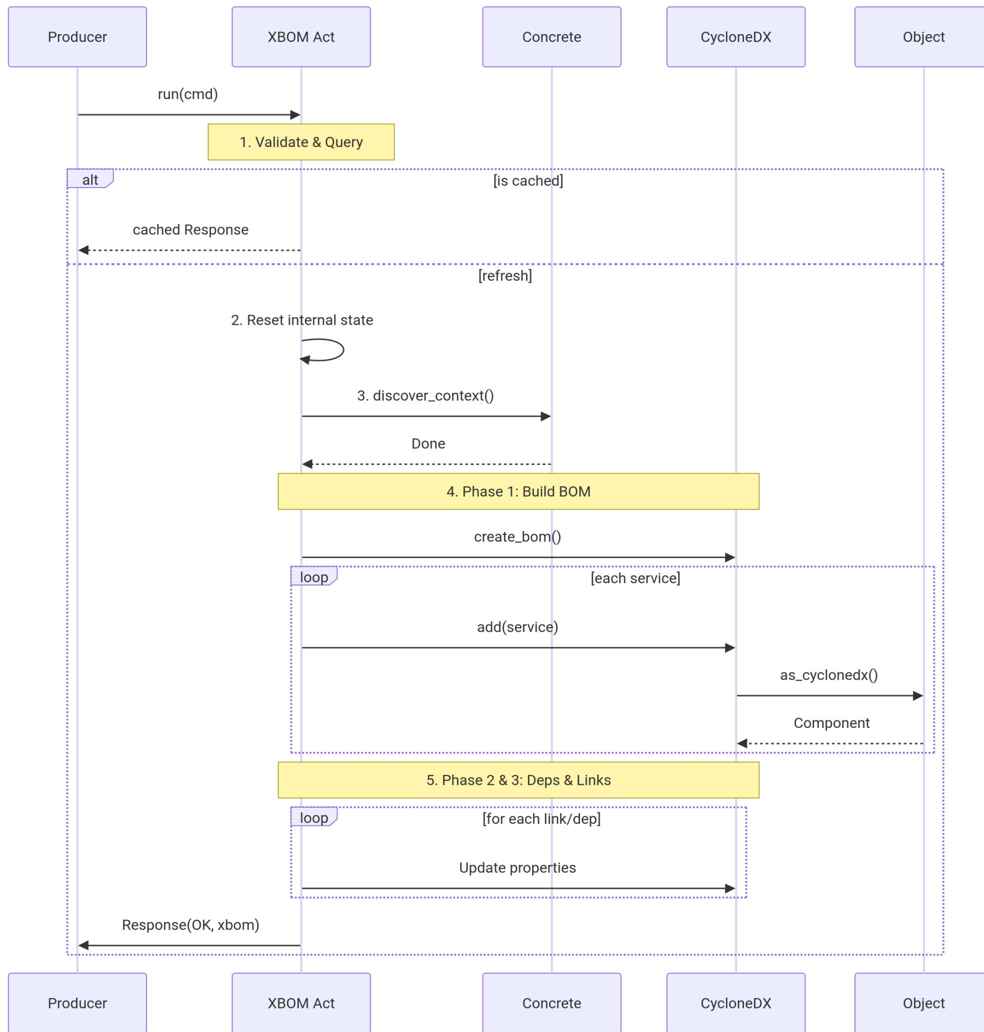

Figure 5.3: Discovery and BOM generation flow in `XBOMActuator`.

Chapter 6

Profile Validation and testing

6.1 Introduction

To ensure the correctness, reliability, and conformance of our system to the OpenC2 specification and profile requirements, it is necessary to perform some testing and validation activities. Since the XBOM profile is designed to be used in a variety of contexts and applications, a variety of testing and validation methods may be necessary to ensure that the profile is correctly implemented and can be used effectively in different scenarios. In this chapter, we will discuss the systematic approach to testing and validating the XBOM profile, which are carried out in three distinct approaches:

- **unit testing**: this approach focuses on the individual components of the profile, such as the data model, the syntax, and the semantics. Unit testing involves creating test cases that cover different aspects of the profile and verifying that the profile behaves as expected in each case. This can include testing for compliance with the OpenC2 specification, as well as testing for specific use cases and scenarios.
- **functional testing**: this approach focuses on the overall functionality of the profile and how it interacts with other components of the system. To this end, a real-world use case is implemented, and the profile is tested in a realistic environment to ensure that it can be used effectively in practice.
- **performance evaluation**: this approach focuses on the performance of the profile and the overhead introduced by the additional structure and validation of the CycloneDX format. By measuring the time taken for different stages of the command processing pipeline, we can quantify the impact of the profile on the overall performance of the system and identify any potential bottlenecks or areas for optimization.

These testing and validation activities are essential to ensure that the XBOM profile is robust, reliable, and can be used effectively in a variety of contexts and applications. By systematically testing and validating the profile, we can

identify and address any issues or limitations, and ensure that the profile meets the needs of its intended users and applications. In the following sections, we will discuss the specific testing and validation activities that we have performed for the XBOM profile, and the results of these activities.

6.2 Unit Testing

Unit testing is a software testing method that focuses on testing individual components or units of a software system in isolation from the rest of the system. The goal of unit testing is to ensure that each component of the system is working correctly and as expected, and to identify and fix any issues or bugs in the code. Unit testing typically involves creating test cases that cover different aspects of the component being tested, such as its functionality, performance, and error handling. These test cases are designed to verify that the component behaves as expected in different scenarios and under different conditions. Unit testing can be performed manually or using automated testing tools, and it is an essential part of the software development process to ensure the quality and reliability of the software system.

The test suite for this profile implementation is organized into three modules that collectively cover the profile from the data model layer up to the full command-processing pipeline, for a total of 120 test cases, all of which pass successfully.
The framework adopted is `pytest`, chosen for its clean, assertion-based syntax and its straightforward support for organizing related cases into classes. One practical challenge that arises when testing an actuator of this kind is that the context discovery phase - the part of the code that collects information about running processes, network interfaces, installed packages, and so on — necessarily interacts with the host environment. Executing those routines in a unit test would make each test dependent on the specific machine it runs on, producing fragile and non-deterministic results. To avoid this, a `MockXBOMActuator` class is introduced that subclasses the real `XBOMActuator` and replaces the discovery methods with no-ops. A `set_test_bom` helper allows each test to inject a pre-constructed `CyclonedxXbom` instance, so the actuator's command-handling logic can be exercised in full without ever touching the underlying system.
The first group of tests, `test_xbom_creation`, focuses on the translation between the context data model and the CycloneDX representation. The profile defines eleven service types — including `Application`, `Host`, `VM`, `Container`, `OS`, `Server`, `IOT`, `Cloud`, `Network` and `Pod` — each of which must be correctly mapped onto either a CycloneDX `Component` or a CycloneDX `Service` object. The tests verify both the explicit conversion path, where `as_cyclonedx()` is called directly, and the implicit path, where the native XBOM object is passed to `CyclonedxXbom.add`, confirming that both routes produce the same artifact with all properties intact. Additionally, some the code was tested with delib-

erately malformed inputs, such as null values, and the tests confirm that the implementation raises explicit errors in these cases instead of silently producing invalid BOMs.
The second group of tests, `test_xbom_discovery`, targets the structural conformance of the profile itself. This includes checking that the profile's namespace identifier and human-readable name are correctly defined, that the `XbomCtx` target and the profile `Args` map can be instantiated and deserialized from plain dictionaries, and that the `Specifiers` map correctly handles the `domain` and `asset_id` fields used for actuator routing. A dedicated set of cases exercises the `validate_command` and `validate_args` functions, confirming that only `query` is accepted as a valid action, that invalid action-target combinations are rejected, and that the profile-specific `cached` argument passes validation. Further cases cover the utility methods of `CyclonedxXbom` — such as BOM merging, dependency tracking, BOM-Link generation, and serialization to a Python dictionary — as well as a set of deliberate fault-injection tests that verify the implementation raises explicit errors when given `None` values, wrong types, or unsupported enum variants.
The third group, `test_xbom_actuator` places the `XBOMActuator` at the center and tests its behavior as a complete command processor. The tests verify that the actuator can be initialized with any combination of optional parameters, that the `create_bom` factory returns the correct type, and that `query/features` commands for `versions`, `profiles`, and `pairs` all return `StatusCode.OK` with a populated results map, while a request for the unsupported `rate_limit` feature correctly returns `StatusCode.NOTIMPLEMENTED`. For `query/xbom` commands, the tests confirm that a Response with a `bom` field in `results` is returned when a BOM is available, that the `cached` flag allows a subsequent query to reuse a previously generated BOM without triggering a new discovery cycle, and that commands carrying non-matching actuator specifiers or unsupported actions are rejected with an appropriate error status.
All 120 test cases pass on the current implementation running on Python 3.13.12 with `pytest` 9.0.2. The results are summarized in Table 6.1.

Test Module	**Cases**	**Status**
test_xbom_creation	25	Passed
test_xbom_discovery	65	Passed
test_xbom_actuator	30	Passed
Total	**120**	**Passed**

Table 6.1: Summary of unit test cases and their status.

6.3 Functional Testing

While the unit tests described in the previous section verify each component of the XBOM profile in isolation, they do not demonstrate whether the system

as a whole can fulfil its intended purpose: letting an OpenC2 Producer incrementally discover the structure of a complex, multi-layered infrastructure by querying different actuators, each of which contributes a partial view from its own vantage point. To validate this capability, a functional test was designed around a realistic scenario that exercises five distinct actuator types and spans five infrastructure layers, from a simple IoT entry point down to the physical cloud fabric.

6.3.1 Scenario Description

The scenario represents a simplified edge-to-cloud architecture: a mobile IoT device communicates with a remote HTTP-based web service through a 5G mobile network whose core components are deployed as cloud-native workloads on a Kubernetes cluster, which in turn runs inside virtual machines managed by an OpenStack cloud. In this setting, the full picture of the infrastructure is not available from any single observation point; each layer can only be described by an actuator that has direct access to its local view. The goal of the functional test is to show that the XBOM profile can be used to progressively reconstruct the complete topology by querying these actuators one layer at a time, and that the resulting set of BOMs can be stitched together into a coherent graph by means of the linking, and eventually cross-Bom linking mechanisms. To allow for a better understanding of the scenario, a simplified diagram of the topology is provided in Figure 6.2. , while the full topology is visualized in Figure 6.2, where each layer is highlighted in a different color and the cross-layer links are shown as dashed lines. Each of the clusters of nodes, representing a BOM produced by a different actuator, is annotated with the name of the corresponding actuator instance, they will be described in more detail in the following sections.
The five layers of the scenario are as follows.

Layer 1 — Entry point (file actuator). The discovery process begins with the least amount of knowledge: an external observer, in this case an IoT sensor, knows only that a mobile terminal exists and that it contacts a remote web server over a 5G network. This information is captured by two instances of the **file actuator** (`xbom-file`), a special actuator type that does not query any live system API but instead reads its service and link definitions from a static YAML configuration. This is due to the fact that tipically this kind of high level information is held by those who operate the system, being a operational description. As such, it can be exposed through this specific actuator type. To this end, two actuator instances are configured with two separate YAML files, each describing a different part of the topology. The first instance (`xbom-file-mobileuser`) produces a single IoT `Host` component — the mobile terminal — together with two links: one pointing to the web server API and one pointing to the 5G network. The second instance (`xbom-file-webserver`) describes the web server itself as an `API` service. These two BOMs are inten-

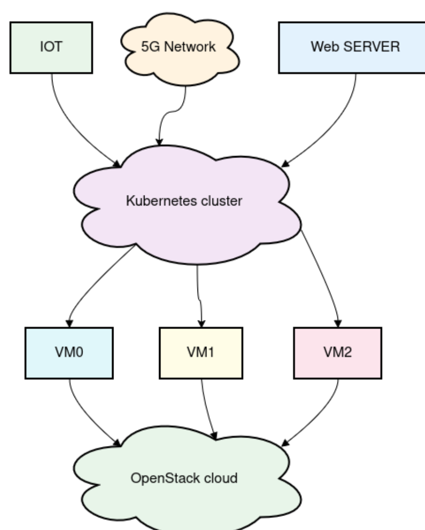

Figure 6.1: Simplified diagram of the functional test scenario, showing the network topology.

tionally minimal; their role is to provide the starting nodes from which the deeper layers will be expanded.

Layer 2 — 5G mobile network (Open5GS actuator). The 5G network referenced in Layer 1 is modelled by a purpose-built actuator (`xbom-open5gs`) that reads the Open5GS deployment manifests and configuration files to reconstruct the logical structure of the mobile core. The actuator discovers two services: the 5G core network itself, described as a `MobileNetwork` with its MCC, MNC, and network name, and the backing data network, described as an `IPNetwork`. In addition, it extracts the internal network functions (AMF, gNodeB, UPF) and the links between them to the Kubernetes applications that implement them. Because the 5G core is deployed in Kubernetes, the actuator also creates a cross-layer link from the 5G network service to the Kubernetes cloud, so that the next layer can be discovered.

Layer 3 — Kubernetes cloud (Kubernetes actuator). The Kubernetes actuator (`xbom-kubernetes`) queries the cluster API to discover the full PaaS-level view. It produces the richest BOM in the scenario, containing fifteen components and thirteen services: the Kubernetes cloud itself, the worker nodes, the pods running the 5G network functions and the web server, the containers inside each pod, the StatefulSet and ReplicaSet controllers that manage the pods, the `ClusterIP` and `NodePort` Services that implement NAT rules, the pod network, and any secondary Multus networks. Sixteen dependency relationships capture the controller → pod → container hierarchy. Links connect pods to their host nodes and to the networks they attach to, as well as to the web server API from Layer 1. At this point, the topology already covers the full application stack, but the underlying virtual infrastructure remains

opaque.

Layer 4 — Host operating systems (host actuators). One instance of the **host actuator** (`xbom-host`) runs inside each of the three Kubernetes worker VMs (`kube0`, `kube1`, `kube2`). Each instance uses the `pyroute2` library and the Linux `ip` tooling to discover the local operating system, its network namespaces, all virtual Ethernet (`veth`) pairs connecting pod namespaces to the host bridge, the VXLAN overlay tunnels that carry inter-node traffic, and the software routers that forward packets between them. The three resulting BOMs expose the low-level networking plumbing that is invisible at the Kubernetes level: which pods share a `veth` bridge on a given node, how the overlay mesh is formed, and which IP subnets are routed through each interface. Each host also creates a cross-layer link to its corresponding OpenStack virtual machine, opening the door to the final layer. Additionally, it's worth noting that it's possible to represent the hypervisor as a node of the context data model, as thus, as component in the BOM. However, in this this scenario, the hypervisor is not represented due to the fact that that it was not available from the host OS, being a nested virtualization setup.

Layer 5 — OpenStack infrastructure (OpenStack actuator). The OpenStack actuator (`xbom-openstack`) connects to the OpenStack API to discover the IaaS-level structure: the VMs, the hypervisor hosting them, the Neutron virtual networks (VLAN-segmented and flat), the virtual routers, and the Nova and Neutron management services. The resulting BOM contains five components, ten services, and eight dependency relationships. Links capture the hosting relationships between VMs and the hypervisor, the packet-flow relationships between VMs and networks, and the controlling relationships between Nova/Neutron and the resources they manage.

6.3.2 Test Execution

The functional test was executed by deploying a set of OpenC2 Consumers — one per actuator — on the target infrastructure and running the orchestraton script, which is configured through a YAML file listing the endpoints to query. For each configured service, the orchestrator creates an OpenC2 `Producer`, constructs a `query` command targeting the `XbomCtx` target, and sends it via the HTTP transfer to the corresponding Consumer. The Consumer's actuator performs its local context discovery, generates a CycloneDX BOM, and returns it in the OpenC2 Response. The orchestrator collects all returned BOMs and publishes them to a JSON file.

In total, the discovery process queried eight actuator instances across the five layers and collected eight BOMs. A summary of the discovered inventory is provided in Table 6.2.

Layer	Actuator	Comp.	Svc.	Dep.	Links
1	`xbom-file-mobileuser`	1	0	0	2
1	`xbom-file-webserver`	0	1	0	0
2	`xbom-open5gs`	0	2	0	7
3	`xbom-kubernetes`	15	13	16	26
4	`xbom-host-kube0`	2	6	7	10
4	`xbom-host-kube1`	1	5	5	9
4	`xbom-host-kube2`	1	5	5	9
5	`xbom-openstack`	5	10	8	14
	Total	**25**	**42**	**41**	**77**

Table 6.2: Inventory discovered during the functional test, broken down by actuator instance.

6.3.3 Cross-Layer Linking

A central goal of the functional test is to verify that the independently generated BOMs can be interconnected into a single, navigable graph. This is achieved through the linking mechanism introduced in the profile design: each service or component that refers to an entity managed by a different actuator carries a `Peer` object whose `sid` field matches the `bom-ref` of the target item in the other BOM. When the target actuator's Consumer endpoint is known, it is also recorded in the `Peer`'s optional `Consumer` object. It is important to emphasize that successfully resolving these cross-layer references requires actuators operating in different domains to use the same criteria when deriving the `SiD` for a shared entity. This requirement is fundamentally a matter of standardization and semantic compatibility rather than a technical limitation of the profile: a consistent naming conventions is fundamenta so that a reference generated by a functional application cleanly resolves to the identifier defined by the underlying hosting infrastructure.
In the test scenario, the cross-layer links form a vertical spine that connects every layer to the one below it:

- Layer 1 → Layer 2: the mobile terminal links to the 5G network (`net:5G/Open5Gs`), pointing the Producer toward the Open5GS actuator.
- Layer 2 → Layer 3: the 5G network and data network services link to the Kubernetes applications that implement the network functions (`amf`, `upf`, `gnb`), connecting to the Kubernetes actuator.
- Layer 3 → Layer 4: the Kubernetes cloud links to the host nodes (`kube0`, `kube1`, `kube2`), connecting to the host actuators running inside each VM.
- Layer 4 → Layer 5: each host OS links to its corresponding OpenStack VM (`host:vm/Default/astrid/kube0`, etc.), connecting to the OpenStack actuator.

6.3.4 Discussion

The functional test confirms that the XBOM profile, combined with the actuator framework provided by `otupy`, can successfully model a realistic multi-layered infrastructure and produce a coherent inventory from multiple independently operating actuators. Several observations are worth highlighting.
First, the scenario demonstrates that the profile's design choice of encoding extended metadata as CycloneDX `Property` objects under the `otupy` namespace is expressive enough to carry the link and peer information needed for cross-layer stitching, without requiring any modification to the CycloneDX schema itself.
Second, the file actuator proves useful as a mechanism for bootstrapping the discovery process: it provides the initial entry points and the coarse-grained topology that guides the orchestrator toward the more specialized actuators. This pattern — starting with a manually curated, high-level description and progressively replacing it with automatically discovered detail — mirrors how an operator would realistically approach the inventory of an unfamiliar infrastructure.
Finally, the diversity of collection methods across actuators — from static YAML files to the Kubernetes API, the Linux networking stack, and the OpenStack SDK — shows that the profile is agnostic to the data-collection technology and can accommodate very different types of environments under the same command-and-control abstraction while still producing a consistent output format. This kind of flexibility is crucial for the profile's applicability in real-world scenarios.

6.4 Performance Evaluation

In addition to validating correctness and functional behaviour, it is important to understand the operational cost of adopting the XBOM profile in place of the original Context Discovery (CTXD) profile. In the CTXD design, actuators returned ad-hoc Python data structures that were serialized to JSON in a relatively direct way. With XBOM, the same contextual information is first mapped into the CycloneDX domain model and then serialized as a standards-compliant Bill of Materials. On the Producer side, the response payload is parsed back into CycloneDX objects and into the profile's own data structures. This additional structure and validation inevitably introduces extra processing and larger payloads.
The purpose of this section is therefore not to optimise every last millisecond, but to quantify how much overhead is attributable to:

- constructing CycloneDX model objects from the internal context representation (conversion);
- serializing these objects into JSON according to the CycloneDX schema (encoding);

- deserializing and validating the CycloneDX JSON on the Producer side (decoding).

To keep the measurements realistic, the evaluation reuses the existing actuators and infrastructure rather than synthetic benchmarks.

6.4.1 Evaluation Methodology

The measurements were collected by instrumenting the OpenC2 processing pipeline with a custom `TimingCollector`. This utility is injected via Python monkey-patching and records timestamps at the boundaries of nine logical stages, starting from the moment a Producer encodes the request and ending when the response has been fully decoded and post-processed. The approach is non-intrusive: no actuator logic was modified beyond the addition of timing hooks, so the numbers reflect normal execution behaviour.
Two actuators were selected as representative targets:

- the OpenStack actuator, which interacts with a wide range of remote APIs and produces a relatively simple BOM
- the Kubernetes actuator, which discovers a much richer topology but against comparatively stable cluster APIs.

For each actuator and for each profile (CTXD and XBOM), the same OpenC2 `query` command was executed sequentially 100 times ($N = 100$) in an otherwise idle environment. The tables in this section report, for each stage, the mean, minimum, and maximum duration in milliseconds. The *Total Round Trip* aggregates the whole end-to-end path from request encoding to final response processing.

6.4.2 OpenStack Actuator

Table 6.3 summarizes the timing results for the `xbom-openstack` and `ctxd-openstack` actuators.

Stage	**XBOM-OpenStack (ms)**			**CTXD-OpenStack (ms)**			**Δ Mean**
	Mean	Min	Max	Mean	Min	Max	
Req Encoding	0.082	0.069	0.155	0.075	0.062	0.130	+0.007
HTTP Send Prep	0.028	0.019	0.040	0.018	0.014	0.043	+0.010
Req Net + Decode	1.279	1.039	1.600	1.364	1.107	1.574	-0.085
Resource Discovery	4890.114	4441.983	13051.693	5221.356	4696.174	8457.468	-331.241
CycloneDX Conv	6.012	3.717	57.496	0.000	0.000	0.000	+6.012
Response Building	0.016	0.013	0.028	0.000	0.000	0.000	+0.016
Res Network	3.811	3.373	6.477	3.311	2.904	5.961	+0.500
Otupy Res Decoding	28.913	26.741	79.187	5.858	4.978	10.979	+23.055
Final Processing	0.037	0.028	0.069	0.001	0.000	0.002	+0.037
Total Round Trip	**4930.292**	4479.326	13089.500	**5232.040**	4706.083	8467.763	-301.748

Table 6.3: Performance Comparison: XBOM-OpenStack vs CTXD-OpenStack ($N = 100$)

For OpenStack, the dominant contribution to the total round-trip time is the *Resource Discovery* stage, which encapsulates all calls to the Nova and Neutron APIs and the subsequent local processing of their responses. This stage alone accounts for approximately 4.9–5.2 seconds on average, and its variability (up to $\sim$13 s in the worst case) is primarily driven by the behaviour of the remote cloud control plane. The small difference in mean discovery time between XBOM and CTXD (about 331 ms) is not due to the profile itself: both implementations use the same underlying logic to carry out the discovery process.
The impact of XBOM becomes visible in the profile-specific stages. The *CycloneDX Conv* stage corresponds to the creation of CycloneDX model objects from otupy's internal representation. Each OpenStack VM, network, router, and service must be mapped into the appropriate CycloneDX `Component` or `Service`, and extended metadata (hypervisor, tenant, network segment, and so on) is attached as namespaced `Property` entries. On average, this structured conversion costs about 6 ms for the OpenStack BOM.
On the Producer side, the *Otupy Res Decoding* stage reveals the largest relative difference between the two profiles. In the CTXD implementation, the response is a relatively shallow JSON object that can be decoded into native Python types with minimal interpretation. In the XBOM case, the CycloneDX JSON is first parsed by the CycloneDX Python library into strongly typed model instances and then rewrapped into otupy's own abstractions. This two-step process, combined with schema-level validation and the handling of nested property bags, results in an average decoding time of about 29 ms, compared to roughly 6 ms for CTXD.
Overall, however, the *Total Round Trip* for XBOM remains very close to CTXD for the OpenStack actuator and is in fact slightly faster in the measured runs. The additional CycloneDX work is effectively hidden behind the noisy latency of the cloud infrastructure.

6.4.3 Kubernetes Actuator

The Kubernetes actuator provides a complementary perspective. Here the APIs are significantly more responsive and stable, but the discovered topology is richer: multiple nodes, controllers, pods, containers, and network objects must be represented, together with their internal relationships. Table 6.4 reports the corresponding timings.

Stage	XBOM-K8s (ms)			CTXD-K8s (ms)			Δ Mean
	Mean	Min	Max	Mean	Min	Max	
Req Encoding	0.140	0.078	0.207	0.082	0.066	0.143	+0.058
HTTP Send Prep	0.057	0.021	0.189	0.019	0.016	0.042	+0.038
Req Net + Decode	1.354	1.075	1.654	1.384	1.099	1.646	-0.030
Resource Discovery	2400.615	2316.895	2646.023	2414.179	2313.760	2521.228	-13.564
CycloneDX Conv	54.889	45.256	108.055	0.000	0.000	0.000	+54.889
Response Building	0.017	0.015	0.024	0.000	0.000	0.000	+0.017
Res Network	24.335	23.012	26.161	19.088	17.438	89.996	+5.247
Otupy Res Decoding	221.141	203.955	279.802	41.259	33.980	110.331	+179.882
Final Processing	0.042	0.036	0.073	0.001	0.001	0.002	+0.041
Total Round Trip	**2702.590**	2601.034	2997.882	**2476.140**	2369.702	2622.098	+226.450

Table 6.4: Performance Comparison: XBOM-K8s vs CTXD-K8s ($N = 100$)

In this setting, the *Resource Discovery* stage for XBOM and CTXD is almost identical (around 2.4 seconds), confirming that both implementations perform the same API calls and local aggregation. The differences appear once more in the CycloneDX-related stages.
Because the Kubernetes actuator must express a dense inventory is more expensive than in the OpenStack case. Each pod, controller, container, Service, and network is mapped into an appropriate CycloneDX element, with cross-layer and intra-cluster relationships encoded through references and `Property` values. This construction step takes on average about 55 ms.
The size and richness of the resulting BOM also affect the network and decoding phases. The XBOM response payload is significantly larger than the CTXD equivalent, which explains the small but consistent increase in *Res Network* time. More importantly, the *Otupy Res Decoding* stage has to parse a deep JSON document into CycloneDX model objects, traverse nested collections of components and services, and reconstruct the profile-specific view. This costs about 221 ms on average, compared to approximately 41 ms for the simpler CTXD response.
In aggregate, the XBOM profile adds roughly 226 ms to the mean end-to-end latency for a Kubernetes query. This overhead is mainly the cost of structured CycloneDX serialization and deserialization, not of the discovery logic itself.

6.4.4 Discussion

Across both actuators, the measurements show a consistent pattern:

- stages that are common to both profiles and dominated by external factors (network latency, OpenStack and Kubernetes API responsiveness) have very similar timings;
- stages that specifically involve CycloneDX model construction, JSON serialization, parsing, and validation are where XBOM incurs additional timing cost.

From an absolute perspective, these costs are modest. Even in the Kubernetes scenario, which is the most demanding in terms of BOM size, the extra

work associated with the CycloneDX library and the profile's type mapping represents a fraction of a second on top of multi-second discovery and control-plane latency. In typical defence use cases, where commands are not issued at human-interactive frequencies and where remote APIs often dominate the execution time, this difference is unlikely to be operationally significant.

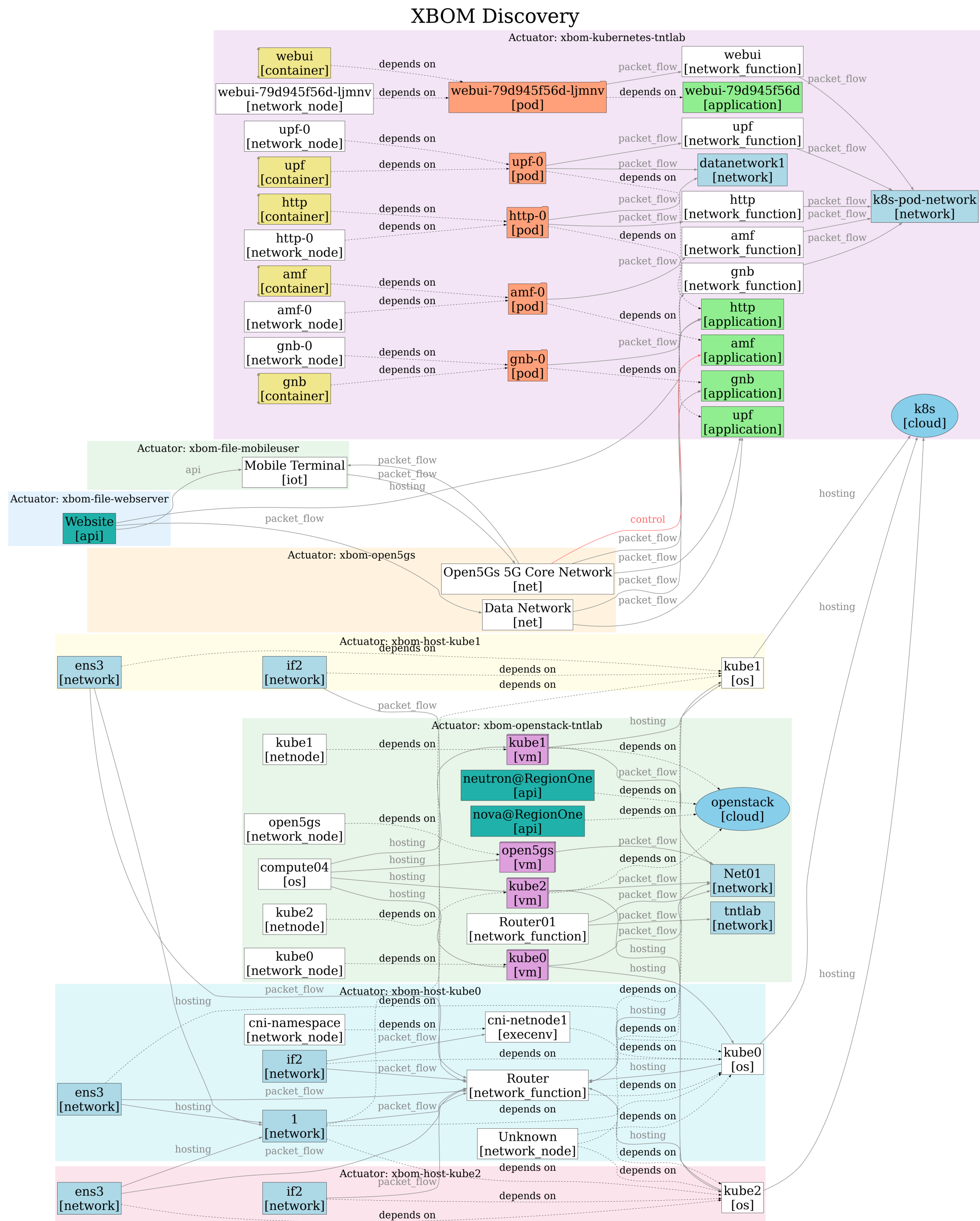

Figure 6.2: Visualization of the simplified topology discovered in the functional test scenario

Chapter 7

Conclusions

This thesis presented the design, implementation, and validation of the Extended Bill of Materials (XBOM) Actuator Profile for the OpenC2 framework. The profile addresses a concrete gap in the current OpenC2 ecosystem: the absence of a standardized mechanism for discovering, representing, and exchanging detailed inventory information about the context in which security functions are deployed and operated. By combining the extensibility of the OpenC2 language with the structured inventory format provided by CycloneDX, the XBOM profile enables OpenC2 Producers to progressively reconstruct the topology of complex, multi-layered environments through targeted queries to specialized actuators, each contributing a partial view from its own domain of expertise.

The resulting XBOM profile provides a powerful new capability for the OpenC2 framework, enabling structured invertory representation capabilities with no significant trade-offs in expressiveness or flexibility. It provides the foundations for extending the context model to additional resource types, for supporting multiple BOM standards, and for enabling a wide range of use cases in security auditing, compliance verification, attack surface analysis, and incident response orchestration. Furthermore, the CycloneDX-based implementation demonstrates that it is possible to leverage an existing BOM standard to represent the rich semantic relationships and cross-references required by automated context discovery use cases in multi-domain environments, without requiring any modification to the underlying schema or any loss of precision in the data representation.

The results confirm that the XBOM profile fulfils its intended purpose. The unit tests verify structural conformance, correct CycloneDX mapping for all service types, and proper command validation and dispatch. The functional test demonstrates that independently generated BOMs can be interconnected through the linking and cross-BOM referencing mechanisms, enabling incremental topology reconstruction from edge to cloud. The custom property taxonomy proved expressive enough to carry the full semantic richness of the XBOM data model without requiring any modification to the CycloneDX schema itself.

Despite these results, the current work has limitations that open several directions for future development. The following sections discuss the two most significant avenues for extension: support for additional BOM standards and expansion of the data model to cover a broader range of service types.

7.1 Extension to Additional BOM Standards

The current implementation uses CycloneDX 1.7 as its sole serialization format, a choice motivated by its mature tooling ecosystem, its native support for both components and services, and its extensible property mechanism. However, the profile was deliberately designed with format independence as a guiding principle: the abstract `Xbom` base class, the format registry, and the polymorphic deserialization logic in `fromdict()` all exist precisely to allow additional BOM formats to be introduced without modifying existing code.

The most natural candidate for a second format implementation is SPDX (Software Package Data Exchange). SPDX is widely adopted in the software supply chain domain and is recognized as an ISO/IEC standard, making it a compelling choice for environments where regulatory compliance or organizational policy mandates its use.

However, extending the profile to support additional BOM standards is not merely a matter of implementing a new serialization backend. Several core mechanisms of the XBOM profile rely on features that are specific to CycloneDX and do not have direct equivalents in other standards, making a careful adaptation essential for each new format.

The first and most critical mechanism is the representation of **links and peers**. In the CycloneDX implementation, links are encoded as custom `Property` objects attached to the service or component that owns the relationship. This approach leverages CycloneDX's permissive property model, which allows arbitrary key-value pairs to be associated with any entry and which is the foundation of the `otupy` custom taxonomy. SPDX, by contrast, models relationships through a dedicated `Relationship` type that connects two SPDX elements by a relationship type drawn from a fixed enumeration (e.g., `DEPENDS_ON`, `CONTAINED_BY`, `BUILD_TOOL_OF`). While this enumeration covers many common dependency types, it does not natively support the richer semantic roles defined by the XBOM data model's `LinkType` and `PeerRole` enumerations, such as `host`, `protected`, or `endpoint`. An SPDX implementation would therefore need to either map XBOM link types onto the closest SPDX relationship types, accepting some loss of precision, or use SPDX's annotation mechanism to carry the additional semantics as structured text. Both approaches involve trade-offs that would need to be evaluated against the specific use cases of the target deployment.

The second mechanism requiring adaptation is the **subservice hierarchy**. In the CycloneDX implementation, the compositional structure of services — where a cloud service contains API subservices, or a Kubernetes controller manages a set of pods — is encoded using CycloneDX's native `dependencies`

array, which records directed *depends-on* relationships between `bom-ref` identifiers. SPDX provides an analogous capability through its `CONTAINS` and `DESCRIBED_BY` relationship types, but the mapping is not one-to-one: CycloneDX dependencies are expressed as a flat list of parent-to-children pairs at the BOM level, while SPDX relationships are expressed as triples (`source`, `type`, `target`) that are scoped to individual elements. A faithful translation would need to emit one SPDX relationship per subservice entry, taking care to preserve the directionality and the distinction between compositional containment and functional dependency.

The third mechanism is the **cross-BOM referencing** system. The CycloneDX implementation uses the *bom-link* URI format (`urn:cdx:{serial-number}/{version}#{bom-ref}`) to reference entries in other BOM documents, a mechanism that is natively supported by CycloneDX's external reference model. SPDX 3.0 provides a similar capability through its updated `ExternalDocumentRef` construct and its native Element graph model, which allows connecting elements across different documents using persistent logical identifiers. While functionally equivalent, the syntax and the resolution semantics differ, and any SPDX implementation would need to generate and consume these references in a way that is consistent with the SPDX specification while preserving the ability to stitch multiple BOMs into a single navigable graph.

Beyond SPDX, additional formats could also be considered in the future, depending on the evolution of the BOM ecosystem. For instance, the emerging OWASP Dependency-Track format, or industry-specific inventory schemas used in telecommunications or industrial control systems, could be accommodated by following the same pattern: registering a new `XbomFormat` value, implementing a concrete `Xbom` subclass, and adapting the link, subservice, and cross-reference mechanisms to the target format's native constructs. Furthermore, it is important to notice that the SPDX format is more focused on software components and their dependencies, unlike CycloneDX which has a more general component and service model. This means that the mapping from the XBOM data model to SPDX may require additional abstraction layers or custom annotations to capture the full range of infrastructure resources and relationships that the profile is designed to represent.

7.2 Extension of the Data Model

The context data model, used in the XBOM profile to represent the inventory of resources and their relationships, in its current form, covers a broad range of infrastructure components — from IoT devices and physical servers to containers, virtual machines, Kubernetes pods, cloud services, and various network technologies. However, the model was designed with extensibility as a first-class concern, and several categories of resources that are commonly encountered in production environments are not yet represented. Extending the data model to include these resource types would increase the profile's

applicability and the completeness of the inventories it can produce.
One significant area for expansion is **storage services**. Modern infrastructures rely heavily on distributed storage systems, including block storage volumes (e.g., OpenStack Cinder volumes, AWS EBS), object storage services (e.g., S3-compatible stores, Swift), shared file systems (e.g., NFS, CephFS, GlusterFS), and persistent volume claims in Kubernetes. A `StorageService` class could capture attributes such as the storage backend type, the capacity, the access mode (read-write-once, read-write-many, read-only-many), the encryption status, and the replication policy. Since storage resources are typically consumed by execution environments and managed by cloud or orchestration platforms, they would naturally integrate into the existing host-to-execution-environment hierarchy through appropriate link types.
A second area is **database services**. Relational databases (PostgreSQL, MySQL, MariaDB), document stores (MongoDB, CouchDB), key-value stores (Redis, etcd), and message brokers (RabbitMQ, Kafka) are foundational components of virtually every modern application stack, yet the current data model does not include a dedicated representation for them. A `DatabaseService` class could capture attributes such as the engine type and version, the connection endpoint, the authentication mechanism, the database name, the replication topology (primary-replica, multi-master), and the backup policy. Similarly, a `MessageBroker` class could model queue-based and topic-based messaging systems with attributes for the protocol (AMQP, MQTT, Kafka), the cluster membership, and the configured topics or queues. These service types would map naturally onto CycloneDX services and would integrate with the existing link mechanism to express which applications depend on which data stores.
A third area concerns **identity and access management (IAM) resources**. In cloud-native environments, service accounts, roles, policies, certificates, and authentication providers (e.g., LDAP directories, OIDC providers, SAML identity providers) play a critical role in defining the security posture of the infrastructure. An `IAMService` class could capture attributes such as the provider type, the authentication protocol, the list of managed service accounts or roles, and the associated policies. Representing these resources in the XBOM would enable the profile to contribute to security auditing and compliance verification workflows, complementing the topology-oriented information currently provided. This concearn is already being addressed in the context of the `otupy` framework by Repetto et al. [21].
A fourth and final area worth mentioning is the refinement of **network function subtypes**. The current `NetworkFunctionType` registry includes `Router`, `NAT`, and `Bridge`, with `VPN` reserved as placeholders. Fully implementing these placeholders and adding new subtypes — such as `LoadBalancer`, `Proxy` (both forward and reverse), `DNSServer`, `DHCPServer`, and `IntrusionDetectionSystem` — would significantly enrich the profile's ability to describe the security-relevant network infrastructure. Each of these subtypes would carry technology-specific attributes (e.g., virtual server defini-

tions and backend pools for a load balancer, zone files and forwarder configuration for a DNS server, rule sets and alert policies for an IDS) and would map onto CycloneDX services following the same conventions already established for routers and bridges.
In all cases, the extension pattern is consistent: a new class is defined as a subtype of `Service` or as a new discriminated union variant within an existing type registry, the `as_cyclonedx()` method is implemented to produce the appropriate CycloneDX component or service with custom properties under the `otupy` namespace, and the corresponding actuator is updated to discover and instantiate the new type during its context collection phase. This regularity is a direct consequence of the data model's design, which separates the concerns of identity (`SId`), topology (`Link`/`Peer`), and type-specific detail (`ServiceType` variants) into orthogonal dimensions that can be extended independently.

7.3 Final Remarks

The XBOM Actuator Profile demonstrates that the OpenC2 framework can be extended beyond its original focus on command-and-control of security functions to encompass the structured discovery and representation of the infrastructure context in which those functions operate. By grounding the profile in an established BOM standard and by designing the data model for extensibility from the outset, the work provides a foundation that can grow with the evolving needs of the cybersecurity community. The open questions identified in this chapter, support for additional serialization formats and coverage of additional resource types, represent natural next steps that build directly on the architecture and conventions established in this thesis.

Bibliography

[1] "Open command and control (c2) architecture specification." https://docs.oasis-open.org/openc2/oc2arch/v1.0/oc2arch-v1.0.pdf, 2024.

[2] M. Repetto, "Otupy: A flexible, portable, and extensible framework for remote control of security functions," *Computers & Security*, vol. 158, p. 104597, 2025.

[3] S. Tanzarella, "Developing the context discovery actuator profile for openc2 language," Master's thesis, Politecnico di Torino, 2024.

[4] "Open command and control (c2) language specification." https://docs.oasis-open.org/openc2/oc2ls/v2.0/oc2ls-v2.0.pdf, 2024.

[5] "Open command and control (c2) stateless packet filtering language specification version 1.0." https://docs.oasis-open.org/openc2/oc2splf/v1.0/oc2splf-v1.0.html, 2021. OASIS Standard, 2021.

[6] "Openc2 implementation considerations for https transfer specification version 1.1." https://docs.oasis-open.org/openc2/open-impl-https/v1.1/open-impl-https-v1.1.html, 2021. OASIS Committee Specification 01.

[7] "Openc2 mqtt transfer specification version 1.0." https://docs.oasis-open.org/openc2/transf-mqtt/v1.0/transf-mqtt-v1.0.html, 2021. OASIS Standard, 2021.

[8] "Openc2 json abstract data notation (jadn) version 1.0." https://docs.oasis-open.org/openc2/jadn/v1.0/jadn-v1.0.html, 2023. OASIS Standard, 2023.

[9] F. R. Jacobs, W. L. Berry, and D. C. Whybark, *Manufacturing Planning and Control for Supply Chain Management*. McGraw-Hill Education, 6th ed., 2011.

[10] R. G. Kula, D. M. German, R. J. Oentaryo, and D. Lo, "The hidden costs of code reuse: A study of the impact of dependency hell on software maintenance," *Empirical Software Engineering*, vol. 23, pp. 167–196, 2018.

[11] Cyber Safety Review Board, "Review of the december 2021 log4j event," tech. rep., Department of Homeland Security, July 2022.

[12] The White House, "Executive order 14028: Improving the nation's cybersecurity." https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/, 2021.

[13] "Cyclonedx." https://cyclonedx.org/.

[14] S. Nocera, S. Romano, M. D. Penta, R. Francese, and G. Scanniello, "Software bill of materials adoption: A mining study from github," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 39–49, 2023.

[15] "Software package data exchange (spdx)." https://spdx.dev/.

[16] "Information technology — software asset management — part 2: Software identification tag," standard, International Organization for Standardization, Geneva, CH, oct 2015.

[17] D. Waltermire, K. Schaffer, K. Scarfone, and M. Souppaya, "Guidelines for the creation of interoperable software identification (swid) tags," Tech. Rep. NIST IR 8060, National Institute of Standards and Technology, apr 2016.

[18] "Cyclonedx python library." https://github.com/CycloneDX/cyclonedx-python-lib. Version 11.6.0.

[19] "Open command and control (c2) bill of materials (bom) actuator profile draft." https://github.com/oasis-tcs/openc2-ap-sbom.

[20] "Ecma-424: Cyclonedx bill of materials specification," Tech. Rep. ECMA-424, ECMA International, December 2025.

[21] N. Poidomani, D. Canavese, D. Bringhenti, F. Valenza, and M. Repetto, "Homogeneous control of security functions via cross-domain delegation," June 2026. To appear.