Defining a framework for automated Software-BOM generation from package metadata in a CI/CD environment

BACHELOR THESIS

Alexander Gschrei

Submitted on 28 September 2022

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

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

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 28 September 2022

License

Erlangen, 28 September 2022

Abstract

With modern applications growing in complexity an increasing reliance on third party components can be observed across the industry. Such reuse of Free/Libre Open Source Software (FLOSS) or Commercial off-the-shelf (COTS) artifacts can help reduce time to market and enable faster release cycles. At the same time, managing a large toolbox of dependencies presents developers with new challenges, both in terms of license compliance and vulnerability monitoring. Package managers like `npm` can help with these tasks by providing at-a-glance information about the dependencies present in a project. However, these tools are often tailored to specific technologies or use-cases and require domain knowledge to be used effectively. As an alternative, this thesis proposes and subsequently implements a new, plugin-based toolchain that provides an abstraction layer on top of existing tools and can be deployed as part of a Continuous Integration (CI) pipeline to generate a Software Bill of Materials (SBOM) from package metadata. To allow for future extensibility, the framework offers a Remote Procedure Call (RPC) interface that allows plugins to exchange data across process and technology boundaries.

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface

CI Continuous Integration

CD Continuous Delivery

CLI Command Line Interface

DBMS Database Management System

ECC Export Control Compliance

FLOSS Free/Libre Open Source Software

COTS Commercial off-the-shelf

HTTP Hypertext Transfer Protocol

IDL Interface Definition Language

IPC Inter-Process Communication

JOSE Javascript Object Signing and Encryption

JSON Javascript Object Notation

JWS JSON Web Signature

NTIA National Telecommunications and Information Administration

OCI Open Container Initiative

OSS Open Source Software

OWASP Open Web Application Security Project

PKI Public Key Infrastructure

PURL Package Uniform Resource Locator

RPC Remote Procedure Call

SBOM Software Bill of Materials

SCP Siemens Clearing Platform

SDLC Software Development Life Cycle

SPDX Software Package Data Exchange

TCP Transmission Control Protocol

TOML Tom's Obvious, Minimal Language

TUI Terminal User Interface

UDS Unix Domain Socket

UPX the Ultimate Packer for eXecutables

VCS Version Control System

XML Extensible Markup Language

YAML Yet Another Markup Language

1 Introduction

Software development is becoming increasingly reliant on code reuse from existing components. Building on top of such artifacts has the inherent advantage that a development team does not need to build the entire functionality of its product from scratch, but can instead leverage existing libraries, frameworks, database systems or, for example in the case of Linux, an entire operating system kernel.

Reuse in this manner capitalizes on effort that has already been expended by third parties and allows developers to focus on implementing the core business logic of their product.

At the same time, introducing external dependencies into a software product, whether they are open source, commercial or in-house, inadvertently adds complexity.

Essential tasks across the Software Development Life Cycle (SDLC) such as adding components to a project, keeping these components up-to-date and referencing them in a central point of record so they become trackable are all challenges that need to be overcome for software products today.

Within *Siemens* we see a trend that products, particularly web applications, often rely on hundreds or even thousands of third-party components. For this volume of dependencies handling the tasks outlined above is not feasible. One way of tackling these pain points are *package managers* like `npm` or `pip` that provide dependency management capabilities for software dependencies, such as installing, updating and listing project dependencies and their relevant metainformation, in most cases also including basic copyright and license information as declared on the package.

Data provided by such package managers can be leveraged in the context of the Siemens Clearing Platform (SCP), a Siemens-internal service that helps projects ensure compliance with the licenses of third-party software components used within their product, a process that this thesis refers to as *Software Clearing*.

While SCP also provides support with Software Clearing for commercial software components, the framework developed in this thesis focuses on open source

dependencies. That is not to say that the given plugin architecture couldn't also be leveraged for the purpose of analyzing commercial dependencies, but the reference tools developed currently do not support this.

The main goal for this thesis is to evaluate the existing tooling within SCP and to improve upon it by providing a framework that unifies the core functionalities required for the generation of Software Bills of Materials (SBOMs).

1.1 Benefits of SBOMs

As an introduction to the topic the thesis will first investigate some of the key benefits that an SBOM can provide for a software product and the stakeholders interacting with it.

In it simplest form, an SBOM is a record of the artifacts included in the component the SBOM describes. The concept of producing such a Software-BOM was adopted from traditional engineering disciplines, where it is commonplace and often required for regulatory reasons to provide a Bill of Materials, that is, a comprehensive record of the the parts and items included in the product.

Producing such a document can provide valuable insight to development teams by making information readily available that would normally have to be retrieved from technology-specific build files. Knowing the dependencies a software artifact has is an inevitable requirement for many regulatory and compliance-related tasks.

One such task as already outlined above is *Software Clearing*, but Export Control Compliance (ECC), where detection of dependencies affected by an embargo or other trade regulations is essential, and vulnerability monitoring to identify known security vulnerabilities are other examples. Additionally, if regressions are introduced by library conflicts after dependency changes in a product, a comprehensive record of how the product dependencies have changed over time can help identify the origin of such conflciting changes.

In the scenarios above, the main consumer of the SBOM is often the producer itself, however publishing or shipping such a document alongside a product also offers benefits to customers and users. On the one hand, security experts can incorporate this data and the derived vulnerabilities in their threat model and, if such documents are centrally stored for all products in use in an organization, it becomes possible to quickly determine which applications are affected by known vulnerabilities. The detection of *log4shell* ('CVE-2021-44228', 2021), a critical bug in Apache Log4J, a popular logging library used in many Java applications, serves as a prime example of a situation in which the ability to quickly identify vulnerable applications is essential for organizations. (Arora et al., 2022, p. 123)

One essential requirement in this context is for the consumers and producers of these documents to agree on a common format. Many companies, including Siemens, have been building records that qualify as an SBOM in the sense that they provide a list of used components, for a long time. However, often these records were only available in a proprietary format which made it difficult to share them with third parties.

As a result, in 2011, the Linux Foundation published the first version of Software Package Data Exchange (SPDX), a data format that facilitates the exchange of package information (Linux Foundation, 2011). Since then, various addtional formats such as `CycloneDx` and `SWID` have been introduced. Section 4.2.5 will investigate the `CycloneDx` format in more detail.

2 Problem Identification

As mentioned in the introduction SCP already provides a toolchain for automated dependency detection to its customers. This thesis will first give an overview of these existing tools and subsequently identify the key problem areas in open source and in-house tools alike.

2.1 Existing solution

A toolchain to automatically detect and analyze third-party package dependencies has been in use within SCP since 2019. This toolchain consists of standalone applications that are designed to run in sequence within customer CI pipelines. These applications can broadly be classified into three main types:

- Scanners, i.e. applications that can understand package metadata and construct an SBOM from it
- SBOM Tools that take an existing SBOM, e.g. the output of a scanner and operate on it
- An SBOM ingester that takes an SBOM and submits its contents to a central system that all software dependencies are tracked in

These existing tools are either bundled as plugins for specific package managers or build tools or they are distributed as container images compliant with the Open Container Initiative (OCI) specification. Each application also provides a basic template for its integration into a `GitLab` CI pipeline. Depending on their target environment, the applications are variably implemented in `Java`, `Python`, `C#`, `Dart`, `TypeScript` or `Bash`.

2.2 Drawbacks of the existing solution

While this existing toolchain is currently deployed in hundreds of internal projects it has a number of drawbacks that are discussed in the following sections.

2.2.1 Code duplication

Many of these tools, while they might consume different inputs, share core functionalities such as resolving license names to SPDX IDs or computing file checksums. In their current form, these core functionalities are not bundled in a common library, nor is it easy to do so given the polyglot nature of the existing toolchain. This leads to code duplication and, particulary in the context of writing output data, any changes to the format either need to be coordinated across all tools or consumers of these outputs need to support various iterations of the format at any given moment. This is especially painful for tooling maintainers if a blindspot or bug in the marshalling of an output file is identified and subsequently needs to be patched in all producers.

The existing tools outlined above rely on shared libraries wherever possible, for example, the company-internal `standard-bom` format provides library implementations for both Java and .NET. However, for integration with existing third-party tools implemented in other technologies, it is currently necessary for developers to provide their own implementation that complies with this format. Keeping these implementations in sync is not trivial, despite efforts to automate compliance validation through a published *jsonschema*.

2.2.2 Inconsistent User Experience

Since the existing tools are loosely coupled and therefore inherently modular, it is possible for users to pick and choose the tools that need to run within their pipelines. At the same time, this also means that each tool needs to be set up and configured individually. To do so, users need to consult the respective documentation for each tool and identify the prerequisites that need to be met for its integration. This process can be error-prone and time-consuming.

Moreover, knowledge about the configuration of one tool may not translate to other tools. For example, while one general guideline for applications that are part of the toolchain is to favor configuration through environment variables, there is no enforced naming scheme for the variables an application expects.

2.2.3 Lack of signing capabilities

The existing toolchain does not provide users the option to cryptographically sign their SBOMs which limits the ability to verify their source and makes it impossible to identify modifications made to the documents in transit. This can lead to scenarios that result in altered SBOMs that may contain defective data entering the central record. Such modifications can be difficult to detect, if at all, and always require manual investigation. Without the ability to verify that a document was produced by a specific user, the current tooling is forced to default

to a trust-all model, that is, any SBOMs that enter the toolchain need to be considered as trustworthy. That is not to say that sanity checks on the data contaiend in the document do not take place, but these checks cannot negate the risks introduced by such an approach.

2.3 General Challenges with SBOM generation

Beyond the drawbacks specific to the existing solution within SCP there are a number of general challenges that arise in the context of automated SBOM generation. This section demonstrates these challenges and exemplifies them on existing open source tools.

2.3.1 Unwarranted trust in Package Metadata

Many tools operate on the premise that package metadata is reliably accurate. There are many real-world examples that prove this assumption generally doesn't hold. For one thing, many packages only allow maintainers to declare a *main* license which means that this data point does not necessarily reflect all licenses present within a package. Some package managers rectify this by allowing *SPDX expressions* to be declared as licenses. This approach is slightly better, but also misses critical information as, for example, it is impossible to reconstruct which files within a package are covered by what license based on the license expression alone. As an example, if a package declares the license expression `LGPL-3.0-or-later AND GPL-3.0-or-later` the only thing that can be inferred from this declaration is that both licenses apply to *some* portions of the package.

Furthermore, while large open source communities equipped with their own governing bodies and review processes may be dilligent about the accuracy of the metadata they publish alongside their packages, the same can not be said for individual maintainers. This problem is aggravated by the fact that many popular packages are authored by a single individual. For example, a metaanalysis of all packages hosted on the `pypi` Index revealed that only a fraction of publicly available python packages were authored by organizations or multiple individuals (Bommarito & Bommarito, 2019).

2.3.2 Unclear definition of SBOM Formats

Another major problem affecting SBOMs is the fact that many of the established formats such as `CycloneDX` and `SPDX` expose many fields that may be populated by a producer, but the standards themselves only require a minimal set of this information to be present for a document to be considered valid. The `CycloneDx` 1.4 specification for example only requires a `type` and a `name` for a component

entry to be compliant with its specification (OWASP Foundation, 2022). However, these two data points alone are hardly useful in any context an SBOM might be consumed in.

For the purposes of Software Clearing in particular, since analysis of a component's source code is often imperative, data points that allow a component to be resolved to the source code it was produced from, such as integrity checksums, are essential. Not all existing open source SBOM generators available today manage to reliably provide this information. Special care needs to be taken when consuming such documents as the fuzzy nature of the data contained within them may jeopardize any efforts of documenting a project's supply chain. Chapter 7 will elaborate on some of the blindspots inherent in existing FLOSS tools.

3 Objective Definition

Based on the identified problems given in the previous chapter, this chapter will focus on defining clear and verifiable objectives for the design and implementation stages of the thesis.

3.1 Framework Definition

Since the goal is to provide a solution that solves problems specific to the domains of Software Clearing and Software Dependency Analysis, it is an obvious choice to implement the toolchain around a framework that encapsulates the domain-specific concepts. To better understand the requirements of such an implementation, this section introduces the core elements of a framework as defined in Riehle (2000).

At its basic level a framework serves as a scaffolding that introduces constraints on how the different elements within the domain model may interact with each other. In object-oriented langauges, such constraints are often defined through interface definitions and abstract classes. As the core of `hyperion`, the tool described in this thesis, is implemented in Go abstract classes are not available and the design solely relies on interfaces.

Users of the framework may then introduce custom behavior at predefined injection points in the framework. Doing so often involves providing a custom implementation for one of the interfaces exposed by the framework. Additionally, template methods may be used to invoke custom logic at certain points in the application lifecycle. This common design pattern described in Gamma et al. (1994) allows framework designers to define the high-level operations an implementation needs to provide, delegating the application-specific implementation to the user.

3.2 Extensibility through plugins

As outlined in the previous section, one of the core properties exhibited by a framework is the ability to extend it. In order to satisfy this requirement, the resulting tool needs to allow its users to override, change and enhance its key functionalities by providing their own implementations. To that end, for the core functionalities of dependency detection, result aggregation and output formatting the framework needs to support the integration of plugins that either build upon or replace the baseline implementation provided by the framework. The plugin subsystem needs to flexibly support implementations written in various different technologies and needs to provide dynamic plugin detection capabilities at runtime based on provided user configuration.

3.3 Project-level configuration

Because of its execution context within a CI environment, the tool needs to allow users to provide configuration on a per-project level. To satisfy this, `hyperion` needs to support loading configuration options from a file that a user can check into their Version Control System (VCS) such that it is available in the CI pipeline's execution context. The exposed configuration options need to cover both the core application including any logging options and the registered plugins, that is, the core application needs to parse the plugin configuration nested in its own configuration and delegate it to the respective plugin. The framework and its documentation also need to provide users with comprehensive documentation about the available configuration options.

3.4 SBOM Contents

The tool and all of its reference plugins need to provide SBOMs that contain the minimum elements put forth by the National Telecommunications and Information Administration (NTIA) as outlined in the table below (Telecommunications & Administration, 2021).

As the tool is primarily expected to detect and report packages provided through package managers we rely exclusively on a Package Uniform Resource Locator (PURL) as as a unique identifier for components within the SBOM and the application's execution context (Ombredanne, 2017). Beyond these minimal requirements one essential data field each plugin needs to provide is a reference to the source code a package was derived from. For the purposes of `Software Clearing` downstream analysis of the source code through an SBOM consumer to detect copyright information and additional licenses within a source set is a mandatory requirement for SCP. In cases where providing a direct reference to source code

Table 3.1: The minimum elements of an SBOM as defined by the NTIA

Data Field	Description
Supplier Name	The name of an entity that creates, defines, and identifies components.
Component Name	Designation assigned to a unit of software defined by the original supplier.
Version of the Component	Identifier used by the supplier to specify a change in software from a previously identified version.
Other Unique Identifiers	Other identifiers that are used to identify a component, or serve as a look-up key for relevant databases.
Dependency Relationship	Characterizing the relationship that an upstream component X is included in software Y.
Author of SBOM Data	The name of the entity that creates the SBOM data for this component.
Timestamp	Record of the date and time of the SBOM data assembly.

or a deep link it can be obtained from is not feasible, tools should alternatively provide fingerprinting information that allows a package entry to be resolved to its source code. One concrete example of such a fingerprint are the hash digests of binary artifacts. These digests, together with the package coordinates, that is, namespace, name and version of a package, make it possible to query public registries for artifacts matching these checksums and subsequently resolve them to the source artifacts they were created from. Should the automation fail to provide such references for open source components, the source code for the affected package would then need to be procured manually, creating a bottleneck in the clearing process.

Even in cases where a package entry in the SBOM provides references to its source code outright, checksums that allow to verify the integrity of this source code are beneficial. For the scope of `hyperion` a cryptographically secure digest, preferably a `sha256` digest is expected to be present. This algorithm and, more broadly, all algorithms of the SHA-2 family, are approved by NIST for the purposes of generating digital file signatures (Dang, 2009). Additionally, for the purpose of allowing straightforward matching against legacy datasets that only provide `md5` and `sha1` checksums, these algorithms also need to be supported, but should only be used alongside a NIST-approved algorithm that additional verifications can be conducted with.

3.5 Command-Line Interface

Because of its execution context within a CI/Continuous Delivery (CD) pipeline the application needs to provide a Command Line Interface (CLI) that can be invoked by a job script. Furthermore, this CLI needs to provide an entrypoint for guided setup, leading the user through the various steps and prompting them for information about the project that is being scanned and the plugins that need to be configured. The CLI application, once invoked, needs to stream logs to `STDOUT` so they can be displayed within a web UI that is commonly provided by CI/CD platforms. Additionally, it must be possible to specify a file as a logging target such that users may persist the job logs as part of their build artifacts.

3.6 Portability

The application needs to be portable, i.e. a user must be able to run the tool on different operating systems and hardware architectures. On the one hand, both `Unix` and `Windows` are in use as CI agents by our customers, additionally `Windows` is the main operating system in use across `Siemens` for developer workstations so while the application may run on a `Linux` machine once deployed in the CI pipeline, the initial setup is likely to happen in a `Windows` environment. Additionally, with the availability of cheaper `Graviton` instances within `AWS` we see a growing number of customers running their pipelines on `ARM64` hardware so this target also needs to be supported alongside the ubiquitous `AMD64` instruction set.

3.7 Reference implementations

Alongside the core application a reference implementation for each plugin type needs to be provided. These reference implementations serve as guidelines for developers looking to develop their own plugins and their code needs to be documented and provided alongside the core application. Additionally, these reference plugins need to be useful, production-ready tools that enhance the core application. Towards that end, the following reference plugins are to be developed:

- A scanner that enables detection of npm packages from package-lock.json and yarn.lock files, developed in `Go`
- A scanner for pypi packages developed in `Python`, showcasing the capability to execute plugins developed in different technologies
- A scanner that is capable of resolving go module references from `go.mod` files checked into a repository.

- A source code downloader developed in `Go` that takes intermediary results and attempts to obtain their source code
- A `CycloneDX` output plugin that consumes the internal SBOM representation and marshalls it into a `CycloneDX 1.4` compliant Javascript Object Notation (JSON)

3.8 Output Signing

The core application needs to provide signing capabilities that allow output plugins to take their output and sign it with a user-provided cryptographic key. The application needs to take special care not to leak the key.

4 Solution Design

This chapter provides an overview of `hyperion`, the CLI tool that was implemented in the context of this bachelor thesis. It covers the subcomponents developed as part of the tool and their respective functionalities. Additionally, an outline of the technologies that enabled its development is provided.

4.1 Subcomponents

In order to fulfill the requirement of flexibility, *hyperion* is composed of multiple subcomponents. This section showcases these components and explains their purpose within the larger application. The following sections are split into an introduction to the main application and the plugins that integrate with it.

4.1.1 Core application

The main application provides common functionalities such as loading and storing configuration, loading and orchestrating plugins and aggregating outputs. To provide these features it exposes its own subcomponents.

Terminal User Interface

`hyperion` is a tool targeted towards users that are expected to be familiar with command-line environments. Consequently setup and configuration of the tool happen through a Terminal User Interface (TUI) that allows users to provide any information necessary to run `hyperion` as part of their CI pipelines. This TUI can be invoked through the `autopilot` command and guides the user through the setup process. After successful completion of this setup, the resulting configuration is written to disk and can be stored alongside the project that is to be analyzed in the VCS.

Filesystem Walker

Scanning the filesystem for package manager metadata that can be analyzed is a common concern for all scanner plugins, therefore the core application centrally provides this functionality to reduce the need for code duplication. Scanners, once loaded, can provide a set of patterns that are of interest to them. The `Filesystem Walker` component will then recursively traverse the directory structure provided in the application configuration, checking each directory and file encountered against these patterns. Additionally, to prevent the component from scanning irrelevant parts of the filesystem, a user can provide an exclusion list of paths to skip. If a path provided on this list is a directory, then all subpaths contained in this directory will also be skipped. The default implementation can be replaced with custom logic that implements the respective interface.

Plugin Loader

This component realizes the detection and loading of plugins. The reference implementation allows users to define a set of search paths that will be considered during plugin detection on top of the default paths. Its interface also exposes methods that allow users to limit the plugins that should be loaded to a known set or to override this behavior and simply load any plugins. The loader's default implementation can be replaced with a custom loader that satisfies its interface.

4.1.2 Plugins

While the core application provides shared functionality, more specialized functionality is implemented in plugins. *hyperion* supports three different types of plugins:

- `Scanners`
- `Transformers`
- `Writers`

Each of these plugin types exposes a public interface that allows developers to provide custom implementations.

`Scanners` are tools that detect the presence of packages or package metadata on the filesystem and construct a result entry for each detected package. The core application is responsible for passing the user-provided configuration options to the plugin when loading or invoking it. The core application executes scanners concurrently and subsequently aggregates the results into an intermediary SBOM expressed in `hyperion`'s internal format. It is the plugin developer's choice whether additional resources, such as a Web Application Programming

Interface (API) are consulted during a Scan to enrich the detections with additional data. Apart from the resulting SBOM entries Scanners are expected to report any errors that occurred during a scan to the host application.

`Transformers` receive the intermediary results outlined above as inputs and return a modified SBOM as output. Transformers may have additional side effects beyond modifications to the BOM. For example, a Transformer could resolve any download URLs provided by package entries, download the associated source code bundle and store it alongside the SBOM. Other examples might include the removal of duplicate entries and the enrichment of existing entries with additional data. By default transformer plugins are executed in sequence in the order they are declared in the configuration.

`Writers` receive the final scan result in `hyperion`'s internal format and write outputs. For example a Writer may convert the internal representation into a well-known SBOM format like `CycloneDX` and store it on the local filesystem or it may choose to instead send the result to a web service for further processing. `Writers` are executed concurrently.

4.2 Technology Stack

This section references the core technologies used to build `hyperion`.

4.2.1 The Go Programming Language

The goal for this project was to provide a tool with a responsive CLI that is performant enough to run as a regular CI job without prolonging pipeline runtimes significantly. The languages evaluated for the implementation of the core application were `Java`, `Python` and `Go` with `Go` being chosen over the other contenders.

`Go` is an open source programming language originally developed by `Google` for in-house use (Robert Griesemer, 2009). It is a statically typed and compiled language that also offers garbage collection. Because `Go` binaries contain native machine code they start up quickly, making the language and its runtime ideal for CLI applications. At the same time, the `Go` toolchain distributed as part of the language allows for straightforward cross-compilation of binaries to other target architectures and operating systems, therefore the compilation to native machine instructions does not impede portability of the application. Due to its classification as a compiled language, `Go` has also been shown to be more performant than an interpreted langauge like `Python` for many use-cases, the *Computer Language Benchmark Game* may serve as empirical evidence of this (Gouy, n.d.). One additional advantage `Go` has over the other options given above is that the language runtime is compiled into the binary and it is therefor not

necessary for a user to have any additional runtime installed in order to execute a `Go` application. More specifically, if CGo support is disabled the go compiler goes as far as producing a purely static binary that does not require any dynamic libraries to be present for a Go application to run. For linux binaries this can be verified by compiling the application and subsequently invoking `ldd` to analyze the binary for any dynamic references. Listing 10 shows an example of this. It is worth nothing that disabling CGo is generally not a risky operation as a build with CGo would simply fail if a Go application requires CGo.

```
$ go build -o hyperion ./core/main.go # default build without additional options
$ ldd hyperion # invoke ldd to show any shared object dependencies
> linux-vdso.so.1 (0x00007fff715a3000)
  libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f531a024000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5319dfc000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f531a045000)
$ CGO_ENABLED=0 go build -o hyperion ./core/main.go # compile again with CGo
    explicitly disabled
$ ldd hyperion # run again on purely static binary
> not a dynamic executable
```

Listing 4.1: Statically compiled Go binaries

4.2.2 gRPC

`gRPC` is an RPC framework that enables efficient communication between *remote* services. A server may expose methods that can be called by clients as if they were local resources. The qualifer *remote* may describe the scenario of client and server running on different machines with calls being brokered over a network, but `gRPC` can also be used for Inter-Process Communication (IPC). This scenario, where client and server both run on the same machine but in different processes is how `hyperion` uses `gRPC`.

The framework allows the use of `Protocol Buffers`, another open source standard, as its Interface Definition Language (IDL) and message format. While other message formats are supported, `hyperion` exclusively uses `Protocol Buffers` to define its interfaces exposed through the framework.

4.2.3 JSON Web Signature

JSON Web Signature (JWS) is a proposed standard (M. Jones, 2015) for the representation of signed content within JSON data structures. A JWS consists of a header, which encodes information about the cryptographic algorithms and input parameters that were used to produce the signature, a payload, which can be an arbitrary sequence of bytes, and a signature which is the result of applying the cryptographic operations defined in the header to both the header and the payload. If an asymmetric key pair is used and a JWS was signed with a private key, then given the corresponding public key, a consumer can verify that the JWS

was not modified in transit and that it was produced by somebody with access to the private key.

4.2.4 syft

`syft` is an open source CLI application maintained by `Anchore Inc.` and distributed under the Apache-2.0 license. It provides the ability to scan container images and filesystems and produce an SBOM from its findings. Its detection capabilities are realized through `catalogers` and as of release `v0.55.0` 18 package types are supported. `syft` exclusively relies on filesystem data for its detections and does not reach out to any external sources to enrich the content of its detections. This approach allows it to produce results very quickly. A typical `syft` scan of a container image only takes a few seconds. However, this decision also comes at the cost of producing incomplete results. In such cases, it is the responsibility of the SBOM consumer to take the result produced by `syft` and enhance it with additional information as needed.

4.2.5 CycloneDX

`CycloneDX` is an open-source SBOM format published under the Apache-2.0 license. The project is backed by the `CycloneDX Core working group` and originated in the `OWASP` community. Release 1.4 of the `CycloneDX` specification supports both Extensible Markup Language (XML) and JSON as output formats and provides schemas for both that SBOM producers can validate their outputs against.

5 Implementation

The following chapter demonstrates how the design decisions outlined in the previous chapter are reflected in the implementation of `hyperion`. Additionally, any third-party libraries and data formats used in the application are referenced where applicable.

5.1 Repository Structure

All source artifacts produced as part of this thesis are stored in a single repository hosted within the company's `GitLab` instance. The layout of its directory presents itself as shown in listing 53. Any code related to the main application is contained in the `core` directory, including the CLI commands to invoke the application and the TUI implementation. The contents of the `pkg` folder will be discussed in section 5.2. The remaining portion of the repository, placed in the `plugins` directory contains both the reference implementations for each plugin type and the shared library portion, including the interface definitions for each plugin type. Section 5.4 will revisit these plugins in more detail.

```
.
|-- core
|   |-- cmd
|   `-- pkg
|       |-- concurrency
|       |-- hyperion
|       |   |-- config
|       |   `-- errors
|       |-- loader
|       |-- models
|       |   |-- artifacts
|       |   |-- hashes
|       |   |-- licenses
|       |   `-- patterns
|       |-- repository
|       |-- signer
|       `-- tui
`-- plugins
    |-- common
    |   |-- proto
    |   |-- scanner
    |   |   `-- proto
    |   |-- transformer
```

```
|   |   `-- proto
|   |-- utils
|   `-- writer
|       `-- proto
|-- scanners
|   |-- go_scanner
|   |   |-- models
|   |   |-- parser
|   |   `-- utils
|   |-- npm_scanner
|   |   |-- models
|   |   |-- testdata
|   |   `-- utils
|   |-- pypi_scanner
|   |   |-- dist
|   |   |-- pypi_scanner
|   |   |   |-- installer
|   |   |   `-- parsers
|   |   |-- testdata
|   |   `-- tests
|   |       `-- parsers
|   `-- syft_scanner
|       |-- converter
|       |-- runner
|       `-- testdata
|-- transformers
|   `-- source_downloader
`-- writers
    `-- cyclonedx
```

Listing 5.1: Tree view of directory structure as produced by 'tree -n –charset=unicode -d'

5.2 Hyperion

The core of `hyperion` provides three main artifacts

- The domain models, i.e. the data representation of dependencies and their metadata within the application
- Utilities that operate on the domain data and realize functionalities commonly required in the context of SBOM generation
- The scaffolding for an application that can detect dependencies, transform their representations and write them to an output format

The following subsections will look at each of these artifacts in detail.

5.2.1 Domain Models

The `models` subdirectory contains struct definitions that represent key elements of an SBOM alongside operations that can be applied to that data. Additionally, it also contains the definitions for any metadata exchanged between the host and its plugins.

Appendix A.1 contains detailed information about all models defined as part of the application.

SBOM-related Models

The keystone in all of these models is the `Package` type which provides a generic representation of an artifact consumed from a package manager. It is conceptually similar to the component type defined by `CycloneDx` but differs in a number of key ways.

For one thing the dependencies of a package are directly defined on the package in hyperion's representation, allowing us to quickly resolve the source graph without the indirection of consulting an additional lookup table. Secondly, as the ability to reference source code is of vital importance to the core use-case of Software Clearing the package struct embeds information about its source artifacts directly on the struct in a special-purpose field. Go's struct embedding feature is used to embed a `PackageCoordinates` subtype in a package. This allows the struct to be passed around as if it were an instance of the embedded type and is conceptually similar to subclassing in other languages. These package coordinates constitute a package namespace, name and version and are sufficient to uniquely identify a package within an SBOM. For convenience, a package coordinates struct also includes a PURL that can be used as a key for many mapping purposes. The `packageurl-go` [1] library and its `PackageURL` type are used for convenience.

Beyond these fields a package may also hold references to the URL its VCS is reachable from, as well as a project homepage and authorship and maintainer information. Any fields referencing organizations or people are of type `LegalEntity` and encapsulate name, email and homepage which are the data fields commonly available on package metadata such as an `npm package.json`.

Information about a package's licenses is encapsulated in the `LicenseContainer` type which is similar to the `LicenseChoice` type defined in the `CycloneDx` specification in that it may either hold an SPDX license expression or a concrete license that may optionally contain a URL reference and the embedded license text.

Beyond these fields a package also provides other metadata such as a copyright field that may contain any string or metadata about the plugin that detected it and where it was detected.

The `SoureArtifact` type is a simple container that either links to a download URL for source or a BOM-relative path. Additionally, checksums that facilitate integrity verification and fingerprinting may be provided.

[1] https://github.com/package-url/packageurl-go

An aggregation of all detected packages is simply referred to as a BOM within the application and, in addition, a BOM contains a metadata section that, among other data points, includes a comprehensive list of any tools or plugins that participated in the creation of the BOM.

Internal Models

Apart from the models describing core SBOM elements outlined above, the application also provides models that are only relevant within the context of the application itself and are used to exchange data between the host and plugin applications.

For example, if a plugin notifies the host about file patterns it is interested in, it does so by providing an implementation of the `SearchPattern` interface.

At its core, the interace, shown in listing 5.2 provides the `Matches()` method which expects a string as input and returns a boolean value. This approach allows the framework to remain agnostic about the underlying implementation.

```
//SearchPattern allows to define rules that paths can be matched against
type SearchPattern interface {
  //Matches returns true if path matches the underlying rule, false otherwise
  Matches(path string) bool
  //Tag returns the tag associated with a pattern, if any
  //These tags can be used by downstream tools to classify patterns/findings
  Tag() string
  //ToDto is a helper method that allows marshalling a Pattern
  // into a DTO that can be sent over an RPC connection
  ToDto() SearchPatternDto
  //ToProto is a helper method that allows marshalling a Pattern
  //into a Protobuf DTO that can be sent over a GRPC connection
  ToProto() proto.SearchPattern
  //Pattern returns a string representation of the search pattern
  Pattern() string
  fmt.Stringer
}
```

Listing 5.2: The SearchPattern interface

Two implementations of this interface are also provided as part of the framework, the `SimpleSearchPattern` that trivially matches whether the string defined by the pattern is contained in a path and the `RegexSearchPattern` which allows for more sophisticated matches based on regular expressions.

5.2.2 Processing Utilities

As already outlined, the implementation also provides common utilities that aid in processing package metadata and related entities.

JWS Signer

One such tool is the `JWS Signer` that allows producing JWS signatures for SBOM output, this subcomponent satisfies the requirement defined in section 3.8. While defined as generically as possible, its main purpose is to provide signing capabilities for output plugins. A plugin can simply invoke the signer with a byte stream payload and the signer will provide a signed JWS token that complies with M. Jones (2015)

The signer supports keys in three different PEM formats: PKCS8 and PKCS1 for RSA keys and EC1 for signing keys based on elliptic curve cryptography. These three formats cover the requirements within Siemens. On top of the standard JWS signing procedure as defined in RFC7515 which has the side effect that the signed data is stored as a Base64 payload on the token itself which forfeits readabilty by humans, the signer also supports detached signing, where the payload is not marshalled to Base64 and added to the token, but rather placed alongside the token. Many library implementations of the JWS standard support this scheme, including `go-jose` which is the implementation the signer relies on.

Concurrency Helpers

One downside of the lightweight design principles for the Go language is that unlike in other languages such as Java, its standard library does not offer higher order concurrency primitives that allow users to conduct operations such as `map`, `filter` or `foreach` in parallel. Such facilities either have to be implemented using the lightweight concurrency facilities available in Go, namely *goroutines*, which allow concurrent execution of code and *channels*, which provide message-passing capabilities between these concurrently executed routines, or imported in the form of existing libraries.

For the case of `hyperion` the decison was made to provide a custom implementation rather than adding a dependency to a library that offers far more advanced primitives than are needed for the purposes of the application. These primitives implement the operations mentioned above using Go Generics which were added to the language in early 2022 (The Go Maintainers, 2022a), as such they are highly reusable. The primitives are particularly helpful in scenarios where slow filesystem or network calls can be overlapped to hide latencies and maximize throughput.

Plugin Loader

`PluginLoader` is a core interface in the application that handles detection and instantiation of plugins. The reference implementation included in the application will crawl a predefined set of directories for files matching predefined naming patterns for plugins. For each of these plugins, the loader will then instantiate an

interface implementation that can be called directly from the core application. If additional features are required, such as signature verification on the binaries to ensure that they are provided by a trusted entity or downloading plugins from remote locations, users may implement their custom loader satisfying the interface and register that with the main application.

5.2.3 The Core Application

The application core is mainly responsible for orchestrating all the primitives defined until this point. Once it has been initialized from configuration, it will progress through the following steps in sequence:

1. Invoke the registered `PluginLoader` instance to retrieve any plugins required for the scan. In addition to these loaded plugins, it is possible for users to explicitly register additional plugin implementations on the application context
2. Query each registered scanner plugin about its declared `SearchPattern` instances and store them on the application context in a map that allows it to correlate the pattern with the plugin it came from
3. Invoke the registered `FsWalker` instance, passing the root path for the filesystem scan and all registered searches
4. Gather the `FsWalker` file matches and partition them based on the scanner plugin they need to be delegated to.
5. Invoke all registered scanner plugins concurrently, passing the relevant identified matches to the plugins.
6. Parse the scan results retrieved from each scanner plugin, handle any scan errors and aggregate the results in a central `Repository` instance that also takes care of removing duplicate findings
7. Construct an intermediate `Bom` instance that is then passed to each registered transformer plugin. The transformer plugins are executed in sequence in the order in which they are defined on the configuration files.
8. Submit the `Bom` as it exists after the transformer call chain has concluded to all registered writers in parallel.
9. Wait for all Writers to complete, process any errors and tear down the application context by invoking the `Cleanup` method that also allows users to register cleanup hooks for their manually registered plugins

The `Run` and `RunAndCleanup` methods encapsulate all of these steps in a single entrypoint that users can call. Additionally, for more specialized needs, the ap-

plication exports additional methods that allow callers to manually orchestrate the execution steps.

5.3 Command Line Interface

The CLI exposes two main commands, `autopilot` which provides the setup and configuration functionality and `scan` which triggers a project scan using the main application. In the implementation of these commands `hyperion` relies on `cobra`, a common `Go` library that aids in the development of CLI tools and provides convenient routines to declare subcommands, define command-line flags and validate input.

5.3.1 autopilot Command

The `autopilot` command is implemented as a TUI that guides the user through the setup process.5.1 shows the user journey for the command. A user is expected to provide key project information that will be referenced in the SBOM such as the main component it is valid for and an optional author reference.

While the user is prompted for this information, the application concurrently scans the known plugin locations for available plugins. By default `hyperion` attempts to detect plugins in the current working directory and the `.hyperion/plugins` subpath of the user's home directory, but these default locations may be overriden through the `--plugin-location` option.

After all plugins are loaded, the application obtains the search patterns from each of the detected scanner plugins and scans the working directory, or alternatively the path provided as an input argument to the command, for files matching these patterns. Based on these findings a user will then be presented with a dialog that allows them to select the plugins they want to enable for the scan. Allowing the user to make an informed decision, detected matches are listed here for each plugin. If the number of matches is too large, the output is truncated and limited to a few exemplary matches. To provide flexibility when running on terminals of different sizes, the TUI supports pagination to enable scrolling in cases where either many plugins are loaded or searches produce many matches.

After proceeding to the next view of the TUI the user will be able to select additional plugins detected on the system. This also includes any transformer and writer plugins. This view can also include plugins that weren't detected on the filesystem, but are instead defined in the *Plugin Catalog* file. This listing of known plugins and their respective default configurations allows references to third-party plugins to be enabled as they become available. The format of a catalog entry is also defined in such a way to facilitate the implementation

of an auto-installer that downloads a plugin from a canonical URL provided in its definition. After a user has selected all desired plugins, the last view of the `autopilot` TUI offers the opportunity to review the `.hyperion.yaml` file that was generated based on the previous user choices and the default configurations for each plugin. The textview component rendered as part of this view allows users to interactively apply configuration options to the config file directly on the terminal.

The interactive portions of the command are implemented using `bubbletea` [2], a Go framework that allows developers to build TUI applications based on the Model View Update architecture popularized by the `Elm` programming language (Maintainers, 2021). `hyperion` defines several message types that are processed in the Update Loop to transition through the setup process, including events that are triggered asynchronously, such as plugin loading, that send data from outside the TUI's context.

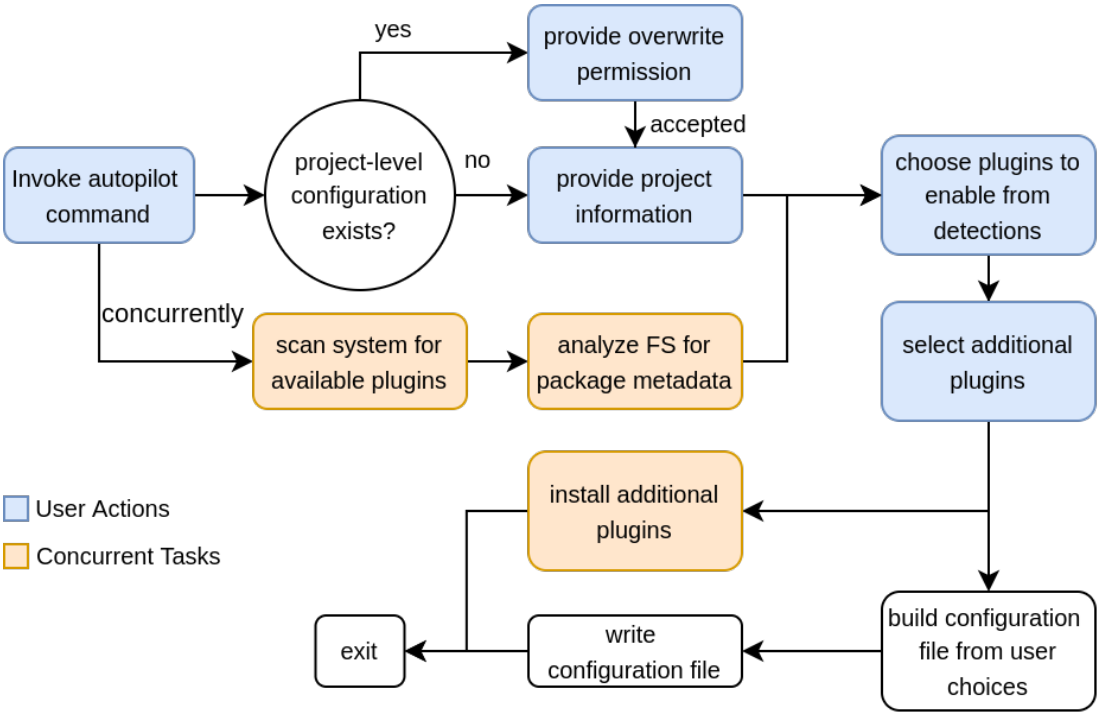


Figure 5.1: User Journey for the autopilot commmand

[2]Please see https://github.com/charmbracelet/bubbletea

5.3.2 scan Command

The other available command on the `CLI` application is the `scan` command that loads the application configuration from file, defaulting to the aforementioned `.hyperion.yaml` file in the working directory the command is invoked in, but accepting overrides through the `-c` option to enable support for referencing centrally stored configuration. The configuration is used to bootstrap the main application and setup its logger. After these setup tasks are complete, this command will simply hand over control to the main application context.

5.4 Plugin Architecture

This section describes the plugin architecture as it is implemented in `hyperion`. While the solution design section mainly focused on the capabilities each plugin type provides, this section outlines how the plugins are implemented from a technical perspective. Table 5.1 showcases the three supported plugin types and explains how they integrate with the main application.

As the most straightforward implementation, the framework supports native plugins that need to be implemented in Go and are essentially consumed as a regular Go library by the host application. The main benefit of this approach is that it suffers from the smallest amount of communication process as there is no requirement for IPC and all data exchanged between the host and a native plugin lives in the same process. At the same time, this tight integration has the downside that the plugin, due to the fact that it is running in the same process, is not *airgapped* meaning that it may be able to access data on the application context that is not intended for its consumption. Additionally, if a bug in the plugin implementation such as an attempt to dereference a null pointer leads to an access violation, the operating system will kill the entire process, including the host.

As a result, native plugins, while requiring the least amount of communication overhead at runtime, require additional care in their development and testing to ensure that they don't threaten the stability of the entire application.

The remaining two plugin types both rely on `go-plugin`, a plugin system developed by `HashiCorp` for use in its own Infrastructure-as-Code tools and subsequently published under the `Mozilla Public License, version 2.0` a weak copyleft license that requires developers to, in turn, publish any changes made to the MPL-licensed code, retaining the MPL license for it. For the purposes of this application, this is however not an issue as it is primarily intended for internal use and qualifies as a "Larger Work" as defined in section 1.7 of the license. No modifications to `go-plugin` source code were necessary in the development of this project.

The second type of implementation are plugins that run as standalone processes spawned and managed by the host process through the functionalities provided by the `os/exec` module that is part of the Go standard library. In essence, the host launches the plugin application and subsequently brokers a connection with it through `net/rpc` an implementation of an RPC protocol on top of Transmission Control Protocol (TCP) that is also provided as part of the Go standard library. While this plugin type does not suffer from the same risks outlined for native plugins, as plugin and host are separated by a process boundary and exclusively communicate through IPC in the form of a well-defined interface, plugins of this type, in practice, still have the limitation that they need to be implemented in Go. It is theoretically possible to write an implementation of the protocol defined by `net/rpc` in other languages. However, doing so would require to additionally implement the `gob` format, a binary data representation used in conjunction with `net/rpc`. Moreover, apart from the significant effort required for such an implementation, given the existence of the third plugin type there is simply no need to go that route.

Plugins implemented in `gRPC` constitute the third supported implementation type. Their inner workings are mostly identical to those of `net/rpc` plugins, but the major difference is, as their name implies, that they rely on the `gRPC` framework for host-plugin communication. While it adds additional overhead to RPC calls compared to the raw TCP connections used by `net/rpc`, this overhead is entirely negligible for the purposes of the application described in this thesis. Interested readers may refer to Duberstein (2020) and Werner (2021) for benchmarks that investigate the impact `gRPC` has on call latencies for local IPC. The main benefit the framework provides is that plugins built with it don't have the same stringent limitations in terms of the underlying technologies and runtimes they may be built on. `gRPC` is an Open Source framework with a cleanly defined protocol.

Apart from the official Google-maintained implementations that exist for the likes of Java and Python, there are also community-maintained implementations for many other runtimes and languages, such as Node.js and Rust. This provides parties interested in contributing plugins to `hyperion` the flexibility to choose the right tool for their needs. The project repository provides the `.proto` files that define the services a plugin needs to implement, including definitions of any data that may be sent to and from the service implementations. These files can either be compiled to language-specific service implementations using `protoc` (The Protocol Buffer Maintainers, 2022) or, depending on the technology used, they may also be parsed dynamically at runtime. One additional advantage this plugin type has is that, should the application require integration with a remote service in the future, the scaffolding to support this is already implemnted simply by virtue of relying on gRPC.

Table 5.1: Overview of available plugin types

Plugin Type	Integration	Overhead
native	run in same process as host, communicate directly through in-process primitives	none - memory is shared with main application
net/rpc	separate process, host application and plugin communicate via RPC over TCP or UDS	data needs to be marshalled in order to be sent over the wire
gRPC	separate process, host application and plugin communicate via gRPC over HTTP/2	data needs to be marshalled and gRPC introduces additional overhead

5.4.1 Interface Definitions

Now that the ground work of introducing the supported plugin types has been laid, this section will elaborate on the application-specific usage of the technologies outline above.

From the perspective of the host application, all plugin types need to implement two common interfaces, `Configurer` and `Informer` that allow it to get and set the plugin configuration, and retrieve information about a plugin, respectively.

```
type Configurer interface {
  SetConfig(pluginConfig map[string]interface{}) error
  GetConfig() map[string]interface{}
}
```

Listing 5.3: The Configurer interface

Listing 5.3 shows the definition of the `Configurer` interface. As can be seen on the signatures of the two methods it provides, the data it receives and returns is deliberately chosen to be as generic as possible. The reason for this is the fact that the host does not need to understand the plugin-specific configuration. To the host, all plugin configuration is simply interpreted as a potentially nested key/value map that is delegated to the plugin. All the host needs to be able to do is to load the plugin configuration from a configuration file and, in the contetx of the autopilot command that generates these config files, write it to one.

```
type PluginInfo struct {
  Name                 string
  Version              string
  SupportedPackageTypes []string
  Build                BuildInfo
}

type BuildInfo struct {
  BuildTimestamp string
  CommitHash     string
  CommitAuthor   string
}

type Informer interface {
  GetPluginInfo() PluginInfo
}
```

Listing 5.4: The Informer interface and its response format

The `Informer` interface on the other hand, is essentially a plugin's calling card, allowing a plugin to self-declare its name and version, supported package types and, optionally, a `BuildInfo` container that holds information about the commit that produced the plugin and the timestamp of its build, if any.

The three plugin types all embed these interfaces as can be seen below.

```
type Scanner interface {
  GetSearchPatterns() []patterns.SearchPattern
  Scan(scanLocations []patterns.SearchResult) (*models.ScanResults, error)
  common.Informer
  common.Configurer
}
```

Listing 5.5: The Scanner interface

Beyond these embedded interfaces, the `Scanner` interface additionally declares a method that allows the host to query its search patterns and a `Scan` method that receives all matches to plugin-specific search patterns and returns any package references the scanner implementation was able to detect.

The `protobuf` service definition for a scanner looks, as one might expect, quite similar, the main difference being that the `protobuf` format does not allow the embedding of services within other services to replicate the embedded interfaces present in listing 5.5.

```
service ScannerService {
  //allows passing of configuration options to a plugin
  rpc SetConfig(common.PluginConfig) returns (google.protobuf.Empty);
  rpc GetConfig(google.protobuf.Empty) returns (common.PluginConfig);
  rpc GetPluginInfo(google.protobuf.Empty) returns (common.PluginInfo);
  //retrieves a list of search patterns from a plugin
  rpc GetSearchPatterns(google.protobuf.Empty) returns (SearchPatternResponse);
  //triggers a scan
  rpc Scan(ScanRequest) returns (ScanResultsResponse);
}
```

Listing 5.6: The gRPC ScannerService definition

The `Transformer` interface presents similarly, introducing only the additional `TransformBom` method that takes a BOM representation and returns a modified version of it or an error if the operation failed.

```
type Transformer interface {
  TransformBom(bom *artifacts.Bom) (*artifacts.Bom, error)
  common.Informer
  common.Configurer
}
```

Listing 5.7: The Transformer interface

As the `protobuf` service definition is similar enough to not require any additional explanation, its listing 1 can be found in the appendix A

Lastly, a `Writer` implementation needs to provide one additional method simply called `WriteBom` that receives a BOM as input and returns an error on failure. Its corresponding gRPC service definition, again, may be found in listing 2

```
type BomWriter interface {
  WriteBom(bom *artifacts.Bom) error
  common.Informer
  common.Configurer
}
```

Listing 5.8: The BomWriter interface

The simplicity of these interfaces has the advantage that plugins are as decoupled from the host as possible. They only serve one specific purpose and that purpose is encapsulated in the interface they sit behind. This simplifies any test fixtures that need to be set up significantly. In its current implementation all gRPC services are implemented through unary rpc calls, that is, the caller sends an rpc request and then blocks until a timeout, a response or an error is received. In practice the host application simply calls these RPC stubs in separate *goroutines* that block until the call completes while the main *goroutine* is free to do other work, if any.

Since the types consumed by the gRPC services are distinct from the types defined in the domain models declared on the main application, the `plugins/common` folder also provides conversion helpers that take the internal representation and map it into the types generated by `protoc` and vice-versa. These types and the code that wraps the gRPC client and server implementations in the interface expected by the main application only need to be defined once per language that needs to be supported.

5.5 Reference Implementations

As part of this thesis, and in order for the host application to be useful, reference implementations for each of the plugin types are provided. These plugins and

their functionalities are outlined in the following section.

5.5.1 npm-scanner

The first implementation to be discussed in this context is the npm-scanner, a scanner plugin implemented in Go that supports consumption as both a net/rpc and gRPC plugin and can detect packages provided by the node package manager. The plugin can parse the following metadata formats to retrieve information about packages:

- package-lock.json files, the format used by npm to store the resolved dependency tree on disk (NPM Maintainers, 2022a)
- package.json files, the format used by npm to express information about an individual package, including its potentially unresolved dependencies (NPM Maintainers, 2022b)
- yarn.lock files, the lock format used by yarn v1, another popular package manager for npm packages

The parsing of these different formats is realized by concrete implemenations of the `NpmMetaResolver` interface shown below.

```
type NpmMetaResolver interface {
  ResolveFromPath(path string) (map[string]models.LockPackage, error)
  Resolve(reader io.Reader) (map[string]models.LockPackage, error)
}
```

Listing 5.9: The NpmMetaResolver interface

For any of the three implementations `ResolveFromPath` is simply a convenience function that calls the `Resolve` method internally with a Reader produced from the file pointed to by the path parameter.

The two lock file variants already contain basic information about each package, at a minimum a package name and version identifier and the dependencies declared by a package. In the case of the yarn.lock file entries are guaranteed to also include a URL the package was obtained from, or in npm terms, *resolved* to. For package-lock.json files whether such a *resolved* link is present depends on the version of the file and a number of other parameters.

In both cases, if they are present, these entries also include subresoure integrity digests [3] that not only allow to verify that the resource hasn't been modified, but that serve as a fingerprint that can be useful in resolving packages from public registries in cases where the resolved URL given on the lock file references a private registry mirror as is often the case in corporate settings.

[3] https://w3c.github.io/webappsec-subresource-integrity/

```
"node_modules/@babel/helper-validator-option": {
  "version": "7.18.6",
  "resolved": "https://registry.npmjs.org/@babel/[...]/helper-validator-option
    -7.18.6.tgz",
  "integrity": "sha512-XO7gESt5ouv/[...]01PCwmR0SJHnkW6i8OwW/EVWRShfi4j2x+KQw
    ==",
  "dev": true,
  "engines": {
    "node": ">=6.9.0"
  }
},
```

Listing 5.10: Example of a single package-lock.json entry (truncated)

```
"@babel/code-frame@^7.0.0", "@babel/code-frame@^7.14.5":
  version "7.14.5"
  resolved "https://registry.yarnpkg.com/@babel/[...]-7.14.5.tgz#23
    b08d740e83f49c5e59945fbf1b43e80bbf4edb"
  integrity sha512-9pzDqyc6OLDaqe+z[...]+vOtCS5ndmJicPJhKAwYRI6UfFw==
  dependencies:
    "@babel/highlight" "^7.14.5"
```

Listing 5.11: Example of a single yarn.lock entry (truncated)

One additional, noteworthy, difference between the two formats is that the yarn.lock file is not a JSON file and the top level key for each entry are actually the dependencies declared by other packages in the same lock file it satisfies. This allows fast resolution of the dependency tree, even for tools that do not implement the resolution strategies used by npm.

On top of its ability to parse these files, the scanner can also query the upstream npm registry at `registry.npmjs.org` for additional package metadata to enrich the BOM. The functionality that handles this is implemented in the `RegistryClient` and complies with the format documented on the npm registry GitHub repository (The NPM Maintainers, 2022).

As lock files can contain well over a thousand packages in some scenarios, it makes sense to send the requests to the registry concurrently rather than in sequence. To handle this and, at the same time, prevent the requests to the registry from running into the undocumented rate-limit defined by its maintainers , the client wraps the default Go HTTP client with a limiter that, in its default configuration, offers a burst quota of 500 requests and refills the bucket at a rate of 100 tokens per second. This limit is low enough to remain in the guaranteed safe-zone while, at the same time, being lax enough that lock files with in excess of 1000 requests can be processed within reasonable time.

Another measure the scanner takes before submitting any API requests to the registry involves deduplication of the aggregated findings from all metafiles.

The default behavior for the scanner is to include all dependencies regardless of their scope, but its configuration options allow users to exclude development

dependencies and to declare additional exclusion rules that prune packages by namespace, full name or a pattern.

5.5.2 pypi-scanner

The pypi-scanner plugin is currently the only plugin that is not implemented in Go. On the one hand it serves as a specific example of how to implement plugins in other technologies based on the gRPC service definitions, on the other hand, relying on the python runtime to implement this plugin also makes sense from the perspective of the resulting detection quality.

The plugin can detect references defined in two common metadata formats:

- `requirements.txt` entries
- `poetry.lock` entries

`poetry.lock` files are reguar Tom's Obvious, Minimal Language (TOML) files, making it possible to easily consume them through a parser for this format, as is the approach taken by the pypi-scanner. `requirements.txt` files on the other hand follow a specialized format defined in Python Software Foundation (2022a) with incarnations of its entries ranging from simple version constraints to complex replace directives that may also point to locations on a filesystem that provide the given dependency.

However, to facilitate the analysis of both formats, the pypi-scanner takes a best-effort approach of parsing the high level information from both files. For `poetry.lock` files this always includes, at a minimum, the name and version of a package, but often also additional data.

To fill in any gaps these detections have the scanner subequently uses `virtualenv`, a tool that allows the installation of isolated python environments, to install the dependencies for each metafile to a temporary directory. After the installation `importlib.metadata`, a package that was introduced to the python standard library in python 3.8 Python Software Foundation, 2022b, is used to discover and parse the metadata for each installed package.

To keep the scan duration low, the implementation relies on the `multiprocessing` package to schedule the installation and parsing steps for each detected metafile to a separate process running the python interpreter.

Any additional data is queried from the official pypi registry API at https://pypi.org that is defined in Python Software Foundation, 2022c. To keep the response latency low, asyncio, which is already present on the environment as a dependency of gRPC, is used to submit any API requests concurrently.

5.5.3 go-scanner

The go-scanner serves primarily as a basic example of a native plugin. It can parse metadata about Go modules, the Go equivalent of a package, using the official modfile library The Go Maintainers, 2022b.

`go.mod` files, unlike the lock files previously mentioned for npm, are not required to include all transitive dependencies. As a result only parsing the top-level `go.mod` file for a project would only yield incomplete data The Go Maintainers, n.d. To resolve this problem, an analyzer needs to recursively resolve any modules listed in a `go.mod` file, retrieve them and proceed resolving the dependencies declared in its `go.mod` file until it either arrives at a package that is not a module or a module that has no dependenices. The go module proxy [4] facilitates this process by providing a central URL that all publicly available packages, even if they are hosted elsewhere, can be retrieved from.

Resolving through this proxy also provides the additional benefit that, in contexts where firewall rules are necessary to access public URLs, only this singular URL needs to be whitelisted to guarantee access to all public go modules.

5.5.4 syft-scanner

The syft-scanner is a shallow wrapper around `syft` that can be used to instrument `syft` scans as part of a `hyperion` scan. Several different approaches of integrating the tool into `hyperion` were investigated, from consuming it as a library directly that can then be treated as a native plugin, over bypassing the plugin architecture and calling it through `os/exec`, to wrapping it in a simple plugin that uses the parsing logic provided by `syft` itself to consume its metadata.

Including it directly into the host application proved problematic, as `syft` relies on several dependencies that require CGo, an integration layer between C and Go to compile, significantly increasing the complexity of multiplatform builds. Since the ability to build binaries for several platforms was one of the requirements defined in section 3.6 this possibility was discarded.

Similarly, a native plugin that only consumes the `syft.Decode` method to read a `syft` BOM from a file, also suffers from the problem that, while only requiring a subset of these dependencies, all dependencies declared in the module are inclued in the final binary, significantly bloating binary size.

To that end, the decision was made to implement the `syft` integration as a gRPC plugin that may optionally be included. The plugin, once its scan endpoint is invoked, executes `syft`, parses its resulting JSON BOM and converts the data from this representation to the `hyperion` models which are subsequently sent

[4] See https://proxy.golang.org/

back to the host. The evaluation chapter discusses future improvements on this current solution.

5.5.5 source-downloader-transformer

The source-downloader-transformer is an example for a native transformer plugin that receives a BOM as input and attempts to resolve its entries to source code that is subsequently downloaded to a location on disk. To achieve this, it first checks whether a component already has a `SourceArtifact` entry populated with a download URL. In this trivial case it downloads the file from this URL and, if checksums are present on the enry, validates the source artifact against them.

The other case supported by this plugin is that of attempting to resolve a homepage or VCS URL to a source artifact. For example, if a GitHub link is provided, it will attempt to query the GitHub API for a download bundle.

5.5.6 cyclonedx-writer

The cyclonedx-writer plugin is a native writer plugin that consumes the internal BOM representation and writes it to a JSON document compliant with the CycloneDx 1.4 specification. The plugin may optionally use the JWS Signer to sign the resulting document with a private key provided by the user.

The CycloneDx format provides the possibility to define dependencies between components within an SBOM through a top-level *dependencies* key that contains the *bom-ref* of the dependent component and a list of *bom-ref* values that denote its dependencies. For the sake of simplicity, the implementation of the cyclonedx-writer simply uses the PURL, that is guaranteed to be unique in the domain model, as a *bom-ref*.

`SourceArtifact` entries from the `hyperion` domain are mapped to *externalReferences* of type *distribution* in the CycloneDx format. Since these entries only allow, and actually require, URLs to be provided, in cases where no download URL for a source artifact is present and only a relative path is set on the package entry, a relative reference as defined in Kerwin, 2017 is used.

This concludes the overview on reference implementations provided as part of the toolchain. The following chapter will demonstrate the functionality of the implementation.

6 Demonstration

As a foundation for a subsequent evaluation that validates the implementation against the objectives defined in chapter 3, this chapter first showcases how `hyperion` may be used in a real-world context.

6.1 Setup

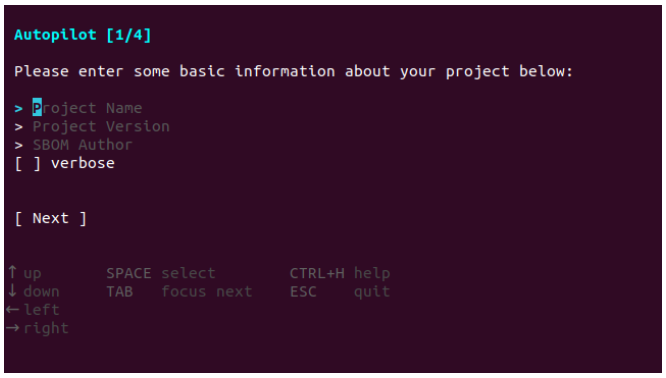
The setup process for the tool, at its most basic level, requires the following steps.

1. Downloading a distribution archive from the project release page on code.siemens.com
2. Extracting the archive to a location of the user's choosing, the recommendation is to store the application and its plugins in `$HOME/.hyperion/`
3. Ensuring that the location of the `hyperion` binary is part of the `$PATH` variable so it can be invoked by its name with the full path (optional, but recommended)
4. Invoking the `autopilot` command to generate a configuration file as a starting point for further modification

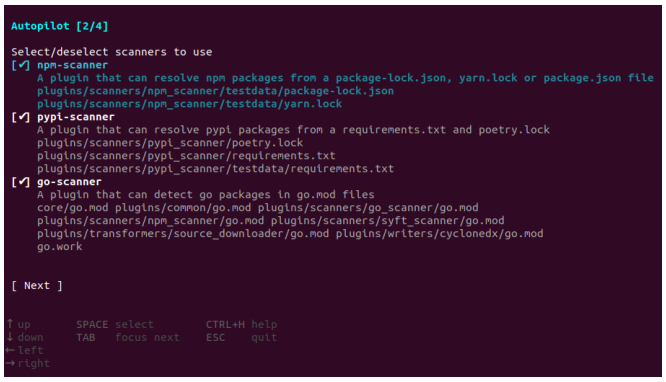
Once the `autopilot` is invoked, the TUI, as outlined in section 5.3.1 will lead the user through the configuration process.

Figure 6.1a shows the view that allows a user to provide baseline information about the BOM to be generated. Additionally, the `verbose` may be set globally here. The expanded help command at the bottom of the TUI is also visible and explains how a user may navigate through the application.

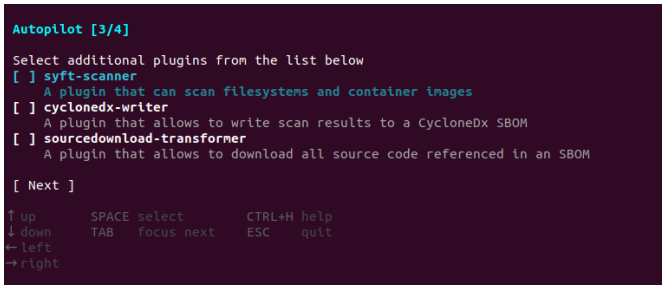
Once a user has proceeded through this input form and accepted the inputs, thy are taken to the next view that showcases the findings produced by all available scanner plugins. Figure 6.1b shows this view for the case where `hyperion` runs inside its own repository root. The findings for each scanner are clearly visible, and all scanners that had detections will be pre-selected. A user may then deselect any scanners that are not relevant to their use-case.



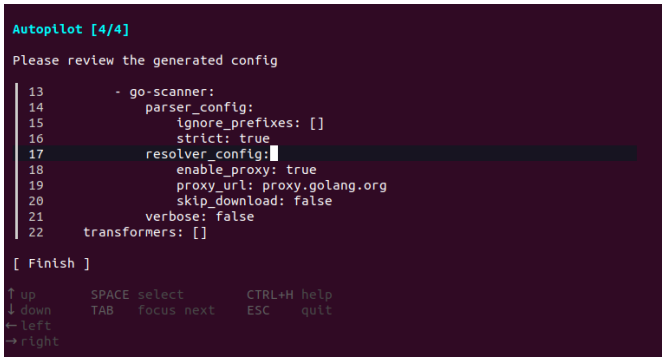
(a) Project Information View



(b) Plugin Findings View



(c) Additional Plugin View



(d) Configuration Review View

After the configuration for scanners with findings is complete, the next view allows a user to select additional plugins available for hyperion. Any plugins already selected in the previous view will not be listed. However, such scanners that are available but did not have explicit findings are listed. [1]

Lastly, the final step of the autopilot command that allows users to review the generated configuration and edit it inline in a `textarea` component can be seen in figure refautopilot-config. Once a user accepts the reviewed and potentially modified documentation, the configuration file is written to disk. By default, the file will be available in `.\.hyperion.yaml` but a user may override this output path using the `-c` flag

[1] This can be useful for the syft-scanner as that does not define any searches and instead expects users to point it to relevant OCI images or files

6.2 Usage

Once this configuration file is available, a user may then trigger the `scan` command to run the tool within their project. This can happen either manually through the CLI, or a user may choose to execute the tools directly as part of a CI pipeline.

To support this use-case, the repository also provides a docker image that bundles the core application and all reference plugins.

This image is provided as part of the container registry within *code.siemens.com* and accessible to any internal users for inclusion in their projects.

The project pipeline is configured using `goreleaser` a build tool that supports integration with `GitLab` and, given a token, may publish the built artifacts to a project release that users may subsequently retrieve it from.

The `gorelaser` build for hyperion is configured in such a way that it first builds all plugins, for the Go plugins binaries are produced for `AMD64` and `ARM64` architectures, targetting Windows and Linux. Once these plugins are built, they are compressed using the Ultimate Packer for eXecutables (UPX) and a detached GPG signature is produced using a private signing key stored as a *secret* on the CI pipeline.

6.3 Sample Project

Now that we have established how `hyperion` can be configured and consumed within a CI pipeline, it makes sense to have a look at a concrete example of a scan.

The sample project consists of three different demo applications:

- A simple python hello world app built on top of `FastAPI`
- A showcase application that demonstrates how Go may be used within a Kubernetes deployment
- A hello world scaffolding built with `vite`, relying on the `npm` package manager

These primitive examples were chosen because, while the project code itself may be very simple, each of these sample applications still declares a reasonably large number of dependencies. Listing 6.1 shows the configuration that `hyperion` was invoked with.

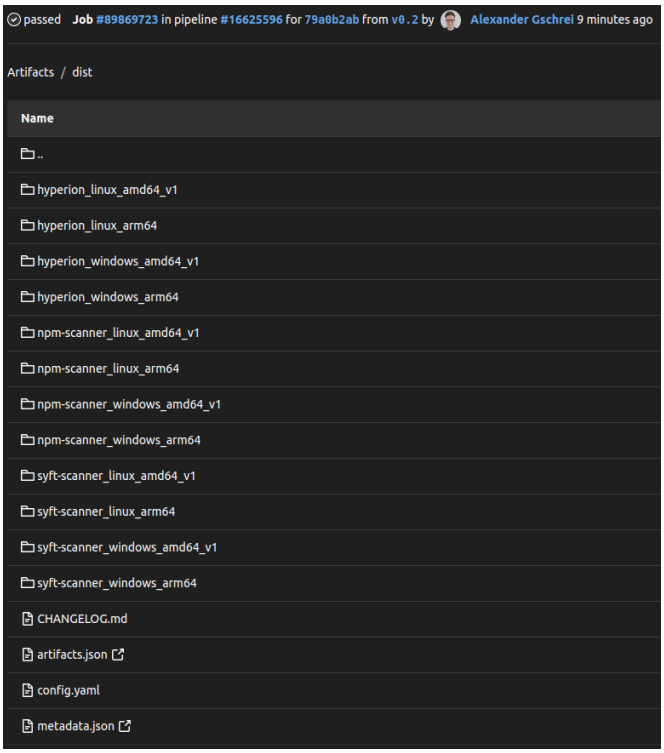


Figure 6.1: Artifacts produced as part of the CI build

```yaml
project_name: Hyperion SBOM Demonstrator
project_version: "0.2"
debug: false
silent: false
plugins:
    scanners:
        - npm-scanner:
            exclude_namespaces: []
            exclude_packages:
              - vite-sandbox
            exclude_patterns: []
            filter_dev: true
            verbose: false
        - pypi-scanner:
            include_dev: true
            include_optional: false
            verbose: false
        - go-scanner:
            parser_config:
                ignore_prefixes: []
                strict: true
            resolver_config:
                enable_proxy: true
                proxy_url: proxy.golang.org
                skip_download: false
            verbose: false
    writers:
        - cyclonedx-writer:
            debug: false
            output_path: output.cyclonedx.json
```

```
            overwrite: true
            pretty: false
            sign_detached: true
            sign_jws: true
            signing_key: ""
            signing_key_path: "signing_key.pem"
plugin_locations:
    - .
    - /root/.hyperion/plugins
ignore_dirs:
    - .git
sbom_author:
    name: Alexander Gschrei
    email: "alexander.gschrei@siemens.com"
    homepage: ""
sbom_supplier:
  name: Siemens AG
  email: softwareclearing@siemens.com
sbom_license:
  spdx_id: MIT
scan_path: ""
load_all: false
fail_fast: false
logging:
    log_path: "hyperion.log"
```

Listing 6.1: hyperion configuration for the example project

A few things are noteworthy about this configuration. First of all, in addition to the logs that are written to `STDOUT` logs are also redirected to a file for later analysis. Additionally, the writer is configured to produce a compact JSON, a JWS signing key loaded from the filesystem is used and hyperion is configured to create a detached signature for the resulting SBOM. Lastly, the `npm-scanner`, `pypi-scanner` and `go-scanner` plugins are all enabled and are be used during the scan.

The resulting CycloneDx SBOM metadata header can be seen in listing 6.2 in a prettifed form as the compact representation produced by the scan does not lend itself to human consumption.

Each of the plugins that participated in the creation of the SBOM is listed in the `tools` key. Additionally, authorship information as given on the configuration file above is included and the `properties` key contains the entire plugin configuration as a base64-encoded string.

```
{
    "bomFormat": "CycloneDX",
    "specVersion": "1.4",
    "serialNumber": "urn:uuid:03779a74-e206-4747-9030-86ff42adb4bc",
    "version": 1,
    "metadata": {
        "timestamp": "2022-09-26T23:59:15z",
        "tools": [
            {
                "vendor": "Siemens AG <scp.it@siemens.com>
    [https://siemens.com]",
                "name": "go-scanner",
                "version": "0.1"
            },
            {
                "vendor": "Siemens AG <scp.it@siemens.com>
    [https://siemens.com]",
                "name": "npm-scanner",
                "version": "0.1"
            },
            {
                "vendor": "Siemens AG <scp.it@siemens.com>
    [https://siemens.com]",
                "name": "pypi-scanner",
                "version": "0.2"
            },
            {
                "vendor": "Siemens AG <scp.it@siemens.com>
    [https://siemens.com]",
                "name": "cyclonedx-writer",
                "version": "0.1"
            }
        ],
        "authors": [
            {
                "name": "Alexander Gschrei",
                "email": "alexander.gschrei@siemens.com"
            }
        ],
        "supplier": {
            "name": "Siemens AG"
        },
        "licenses": [
            {
                "license": {
                    "id": "MIT"
                }
            }
        ],
        "properties": [
            {
                "name": "siemens:hyperion:plugin-config",
                "value":
    "WwogIHsKICAgICJjeWNsb2[...]lLAogICAgInZlcmJvc2UiOiB0cnVlCiAgfQpd"
            }
        ]
    },
```


Listing 6.2: The resulting CycloneDx Metadata

This provenance information, especially as a signature that allows to verify its source is provided alongside it, can be extremely helpful for downstream consumers of the SBOM in determining the context in which the document was created.

After this metadata section, the detected components are listed. An example for one of the npm components detected by `hyperion` can be seen in listing 6.3


```
  {
    "bom-ref": "pkg:npm/%40babel/code-frame@7.18.6",
    "type": "library",
    "group": "@babel",
    "name": "code-frame",
    "version": "7.18.6",
    "scope": "required",
    "hashes": [
        {
            "alg": "SHA-512",
            "content": "4c30a694ae5e3af19[...]3146280cdc8b5daa37fbbd1"
        }
    ],
    "purl": "pkg:npm/%40babel/code-frame@7.18.6",
    "externalReferences": [
        {
            "url":
    "https://registry.npmjs.org/@babel/code-frame/-/code-frame-7.18.6.tgz",
            "comment": "Remote URL",
            "hashes": [
                {
                    "alg": "SHA-512",
                    "content": "4c30a694ae5e3af1[...]fe787edc740aa37fbbd1"
                }
            ],
            "type": "distribution"
        }
    ]
},
```


Listing 6.3: A component entry for an npm package (checksum truncated)

The component coordinates and checksum digests are present, additionally the URL the package source was resolved from is given as an external refrence of type `distribution`. For this particular package, `hyperion` was not able to detect its license.

Similarly, listing 6.4 shows an entry for a `pypi` package, one of 17 detected in total, that has most of its metadata filled in, including a direct link to the pypi index `sdist` download and authorship information.

```
{
        "bom-ref": "pkg:pypi/fastapi@0.85.0",
        "type": "library",
        "author": "Sebastian Ramirez",
        "name": "fastapi",
        "version": "0.85.0",
        "description": "FastAPI framework, [...] ready for production",
        "scope": "required",
        "hashes": null,
        "licenses": null,
        "purl": "pkg:pypi/fastapi@0.85.0",
        "externalReferences": [
            {
                "url":
    "https://files.pythonhosted.[...]43f87338a273/fastapi-0.85.0.tar.gz",
                "comment": "Remote URL",
                "hashes": [
                  {
                      "alg": "MD5",
                      "content": "0ab4758246c22450dfae4046442879f
                  },
                  {
                      "alg": "SHA-256",
                      "content": "bb219cfafd[...]6f5a9f9f1416878"
                  }
              ],
              "type": "distribution"
          }
      ]
}
```


Listing 6.4: A component entry for a pypi package (checksum and URL truncated)

Lastly, an example for a Go module detected by `hyperion` is given in listing ??


```
{
  "bom-ref": "pkg:golang/github.com/jackc/pgtype@v1.10.0",
  "type": "library",
  "group": "github.com/jackc",
  "name": "pgtype",
  "version": "v1.10.0",
  "scope": "required",
  "hashes": [
      {
          "alg": "SHA-256",
          "content":
    "20b9c15ab44c49719ccd8be69180fa3ec63254550d2d39c85091c70cb9aa937f"
      }
  ],
  "licenses": null,
  "purl": "pkg:golang/github.com/jackc/pgtype@v1.10.0",
  "externalReferences": [
      {
        url: "https://proxy.golang.org/github.com/jackc/pgtype/@v/v1.10.0.zip",
        "comment": "Remote URL",
        "hashes": [
            {
              "alg": "SHA-256",
              "content":
    "20b9c15ab44c49719ccd8be69180fa3ec63254550d2d39c85091c70cb9aa937f"
            }
        ]
```

```
        }
    ]
}
```

Listing 6.5: A component entry for a Go package

One thing worth to note in this listing is that the `go-scanner` will always set the canoncial path resolved through the golang proxy as the download URL to maximize compatibility. Additionally a `sha256` digest is present that was resolved from a `go.sum` file.

These example components show that `hyperion`, while its plugins require additional work to maximize the information they are able to extract from the package metadata, is capable of building an SBOM populated with enough information about a component that an interested party may use these data points and subsequently enhance them with, for example, license data obtained from scanning the referenced source archives.

7 Evaluation

This chapter evaluates the design and implementation of `hyperion` both in terms of the objectives defined in section 3 and the accuracy of the SBOMs produced by the reference plugins.

7.1 Extensibility

The extensibility of the toolchain to support additional use-cases as defined in section 3.2 was one of the key objectives for this thesis. The plugin architecture designed in 4.1.2 provides users flexibility to pick and choose the plugin implementation type that best fits their needs. The interfaces plugins need to implement are clearly defined, be it in Go directly or through the gRPC services the host application consumes. Using the existing reference implementations and, in the case of a gRPC plugin, the official gRPC documentation, development of a plugin is possible without requiring any particular knowledge about the core application. Developers can focus on implementing the interfaces expected of a valid plugin and have the peace of mind that as long as this interface is implemented properly, the plugin can be loaded by the host application.

As such the implementation succeeds in fulfilling this requirement. However, in retrospect, it may have been a better decision to simply ignore the `net/rpc` plugin type altogether. The library, and particularly the underlying `gob` encoding scheme are not straightforward to use and the Go core team has made the decision to freeze the `net/rpc` plugin due to known bugs in the implementation that are difficult to fix [1]. As a result, developers implementing plugins for `hyperion` should always prefer gRPC over `net/rpc`

[1] See https://github.com/golang/go/issues/16844 issue on GitHub for details

7.2 CLI and TUI implementations

The tool provides both of these features, as demonstrated in sections 6.1 and 6.2 allowing developers comfortable with a command line environment to quickly configure and run the tool. The frameworks `hyperion` relies on to provide this functionality are both popular Go projects with an active developer community. Additionally, `hyperion` currently only uses a small subset of the features offered by `bubbletea` and adding new features to the TUI as the application evolves is straightforward.

7.3 Validity and accuracy of produced SBOMs

The results produced by the `pypi-scanner` offer by far the highest level of detail. The scanner is successfully able to detect all transitive dependencies leveraging `pypi` and the official `pypi` registry API which allows it to not only resolve the dependency graph, but also to reference available *sdist* packages. These source distributions, unlike their precompiled counterparts, offer valuable insight into any additional copyrights and license references that may be available in the source files. The reliance on `pip` and `virtualenv` however also has the side effect that, measured in time spent per detected component, this scanner is by far the slowest of the developed reference implementations. For the example given in section 6.3 the pypi-scanner took roughly 5 seconds to obtain results for 17 packages while, in comparison, the `npm-scanner` processed more than 118 packages in 12 seconds.

Furthermore, the `npm-scanner` offers results of an acceptable quality by combining the metadata directly extractable from the respective lock files with additional information queried from the upstream `npm` registry. However, there are some cases in which the npm-scanner is unable to correctly resolve the dependencies between packages that were parsed from yarn.lock files. This is due to an oversight in the current implementation that does not support complex version fields in the `yarn.lock` format shown in listing 5.11. A `yarn.lock` entry may actually contain complex version expressions for these top level keys that define wich requirements are satisfied by a package. The current parsing logic does not take this into account and future releases will need to address this.

The `go-scanner`, by referencing the Go package proxy, can reliably resolve any publicly available modules. Additionally, it is capable of parsing `go.sum` files that contain checksum digests for each dependency within a module to verify that a module retrieved through the is indeed resolved correctly. The current version, however, still lacks a mechanism to automatically classify the licenses bundled alongside Go modules.

Nevertheless, all three of these implementations can produce SBOMs that fulfill the requirements given in table 3.1.

The `syft-scanner` produces less comprehensive results however. This is due to a number of difficulties in consuming the detections parsed from `syft` results. First of all, `syft` maintainers chose the approach of embedding specific metadata types for each package technology into the output format, which necessitates an implemenation consuming this data to consider each of these types separately even in cases where the metadata keys themselves are actually identical. A future improvement of the `syft-scanner` may rectify this problem by dropping the dependency on the `syft.Decode` call to unmarshall the data and instead implement its own target struct the JSON results can be marshalled into following the schema definitions published as part of the syft repository (Syft Maintainers, 2022). Resolution from `syft` detections to actionable download URLs has also proven difficult and requires future improvements.

The blindspots identified in this section will serve as actionable tasks for the future development of `hyperion` and lead us to th efinalconclusion for this thesis.

8 Conclusions

The design and implementation of `hyperion` can be considered a success. While the toolchain still exhibits a number of known issues and blindspots that need to be rectified to improve the accuracy and reliability of its detections, the tool in its current form is able to provide what can best be described as baseline findings for each of the scanner reference implementations.

The tool solves most of the problems identified in section 2. On the one hand, it provides the desired BOM signing capabilities, on the other hand, its `autopilot` command helps to unify the configuration for all plugins and reduces the setup complexity when compared to the existing tools.

Another major benefit of the plugin architecture implemented in `hyperion` is that it allows to decouple proprietary tools from the rest of the application such that the core and many of its plugins that do not contain intellectual property the company may want to protect, may eventually be published as Open-Source Software.

This is a natural next step for the SCP following the registration of an official Siemens namespace in the `CycloneDx` property taxonomy (CycloneDx Maintainers, 2022).

The FLOSS space provides a plethora of tools that enable SBOM generation already, but the main novelty of `hyperion`, its truly multiplatform plugin architecture can be helpful in scenarios where users need to integrate complex systems from different technology stacks with one another.

Appendices

A Code Listings

```
service TransformerService {
  rpc SetConfig(common.PluginConfig) returns (google.protobuf.Empty);
  rpc GetConfig(google.protobuf.Empty) returns (common.PluginConfig);
  rpc GetPluginInfo(google.protobuf.Empty) returns (common.PluginInfo);
  rpc TransformBom(common.Bom) returns (common.Bom);
}
```

Listing 1: The TransformerService Protocol Buffer definition

```
service WriterService {
  rpc SetConfig(common.PluginConfig) returns (google.protobuf.Empty);
  rpc GetConfig(google.protobuf.Empty) returns (common.PluginConfig);
  rpc GetPluginInfo(google.protobuf.Empty) returns (common.PluginInfo);
  rpc WriteBom(common.Bom) returns (google.protobuf.Empty);
}
```

Listing 2: The WriterService Protocol Buffer definition

A.1 Go Domain Models

This section contains a comprehensive overview of the core models defined as part of the main application. The source code comments have been retained to provide additional information about the intent of each field where applicable.

Please note that the search pattern model is omitted here as it was already described in-depth in the main section of the thesis.

```
// PackageCoordinates are used to identify an artifact both in its local context
// (specific to an artifact manager) and globally through a PackageURL
type PackageCoordinates struct {
  //Namespace is optional as not all package types have the concept of a namespace
  Namespace string `json:"namespace,omitempty"`
  //Name is the package name in package space
  Name      string `json:"name"`
  Version string `json:"version"`
  //Purl effectively contains redundant information to the fields above, but we choose this
     approach as it allows
  //consumers of this struct to choose which data point they want to use
  Purl purl.PackageURL `json:"purl,omitempty"`
}

//Package represents an artifact retrieved from a package manager and/or its metadata
type Package struct {
  PackageCoordinates //embedding
  //Type can be 'application' or 'library' - use 'library' as default
  Type string `json:"type"`
  //Hashes contains checksums that allow fingerprinting of packages
  Hashes Hashes `json:"hashes"`
  //Sources holds all source artifacts associated with a package,
  //i.e. the source the package was built from
  Sources      []SourceArtifact             `json:"sources,omitempty"`
  Description string                       `json:"description,omitempty"`
  Licenses     []licenses.LicenseContainer `json:"licenses,omitempty"`
  //VcsUrl is a direct link to the official VCS that houses the source code for the package
  VcsUrl string `json:"vcs_url,omitempty"`
  //Homepage is a direct link to the project URL
  Homepage string `json:"homepage,omitempty"`
  //Copyrights as declared on package metadata
  Copyright string `json:"copyright,omitempty"`
  //Scope can either be 'required','optional','dev' or 'compile'
  Scope Scope `json:"scope"`
  //Dependencies is a list of purl-references to other packages this package depends on
  Dependencies []purl.PackageURL `json:"dependencies"`
  //Author as declared by the package metadata
  Author *LegalEntity `json:"author,omitempty"`
```

```
  //Maintainers is a pass-through field for package-level maintainer info
  Maintainers []LegalEntity `json:"maintainers,omitempty"`
  //FoundBy contains a list of plugins that detected the package
  FoundBy []string `json:"found_by,omitempty"`
  //FoundIn contains a list of locations where a package reference was found
  FoundIn   []string      `json:"found_in,omitempty"`
  Supplier *LegalEntity `json:"supplier,omitempty"`
}
```

Listing 3: The Package type and its PackageCoordinates embedding

The `Bom` and `BomMetadata` types are shown below:

```
//Bom is the main container type for any package findings and execution metadata
type Bom struct {
  FormatVersion string        `json:"bom_format"`
  Metadata      *BomMetadata `json:"metadata,omitempty"`
  Packages      []Package     `json:"packages,omitempty"`
}

//BomMetaddata is the metadata header of the BOM
type BomMetadata struct {
  //Name of the product the BOM is generated for
  ProductName string `json:"product_name"`
  //Version of the product the BOM is generated for
  ProductVersion string `json:"product_version"`
  //Plugins that were used to generate the BOM
  Plugins map[string]PluginInfo `json:"plugins"`
  //Author of the BOM
  Author *LegalEntity `json:"author"`
  //Supplier of the BOM
  Supplier *LegalEntity `json:"supplier"`
  //License of the product the bom is associated with
  License *licenses.LicenseContainer `json:"license"`
}
```

Listing 4: The Package type and its PackageCoordinates embedding

```
  //License identifies a single license
type License struct {
  //SpdxId references the ID as given on https://spdx.org/licenses/ if applicable
  SpdxId string `json:"spdx_id,omitempty" yaml:"spdx_id"`
  //Name gives the canonical name of the license
  Name string `json:"name,omitempty" yaml:"name"`
  //LicenseText is a reference to a text finding
  LicenseText *LicenseText `json:"license_text,omitempty"`
  //Url contains a direct link to the license information - a simple GET call should return
    the data
  Url string `json:"url,omitempty" yaml:"url"`
}

type LicenseText struct {
  //Location denotes a package relative path where the license was found
  Location string `json:"location"`
  //Text contains the full license text - may contain copyright holder specific information
  Text string `json:"text"`
}

type LicenseExpression string

type LicenseContainer struct {
  License    License           `json:"license,omitempty"`
  Expression LicenseExpression `json:"expression,omitempty"`
}
```

Listing 5: The models declared in the licenses package

```
type SourceArtifact struct {
  RelativePath string
  DownloadUrl  *url.URL
  Hashes       Hashes
}
```

Listing 6: The SourceArtifact struct

The types related to hash digest can be seen here:

```
//HashAlgorithm is a type alias that can represent common hashing algorithms.
//The `hashes` package offers constants that can be used with this type
type HashAlgorithm uint8

const (
  Unknown HashAlgorithm = iota
  Md5
  Sha1
  Sha256
  Sha512
  Sha3_256
  Sha3_512
)

// HashDigest encapsulates a hex-encoded digest and information about the algorithm it was
    generated with
type HashDigest struct {
  Algorithm HashAlgorithm `json:"algorithm"`
  Digest    string        `json:"digest"`
}
```

Listing 7: The HashDigest models

Lastly, the models that encapsulate scanner results are shown below. Please note that the ScanError struct implements the builtin `error` interface and facilitates the construction of error chains

```
type ScanStatistics struct {
  ArtifactsFound uint64
  ScanDuration   time.Duration
}

type ScanResults struct {
  Statistics ScanStatistics       `json:"statistics,omitempty"`
  Packages   []artifacts.Package  `json:"packages,omitempty"`
  Errors     []error               `json:"errors,omitempty"`
}

type ScanError struct {
  cause []error
}
```

Listing 8: The HashDigest models

References

Arora, A., Wright, V., & Garman, C. (2022). Strengthening the security of operational technology: Understanding contemporary bill of materials. *JCIP The Journal of Critical Infrastructure Policy*, *3*(1), 111.

Bommarito, E., & Bommarito, M. (2019). An empirical analysis of the python package index (pypi). https://doi.org/10.48550/ARXIV.1907.11073

CVE-2021-44228. (2021). Retrieved August 22, 2022, from https://nvd.nist.gov/vuln/detail/CVE-2021-44228

CycloneDx Maintainers. (2022). Cyclonedx property taxonomy.

Dang, Q. H. (2009). Sp 800-107. recommendation for applications using approved hash algorithms.

Duberstein, T. (2020). Cockroachdb/rpc-bench.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1994). *Design patterns: Elements of reusable object-oriented software* (1st ed.). Addison-Wesley Professional.

Gouy, I. (n.d.). The computer language benchmarks game - go vs python3. Retrieved August 15, 2022, from https://salsa.debian.org/benchmarksgame-team/benchmarksgame

Kerwin, M. (2017). *The "file" uri scheme* (RFC No. 8089). RFC Editor. RFC Editor. https://datatracker.ietf.org/doc/html/rfc8089%5C#appendix-E.2.1

Linux Foundation. (2011). *SPDX Specification version 1.0*. Retrieved August 30, 2022, from https://spdx.dev/wp-content/uploads/sites/41/2017/12/spdx-1.0.pdf

M. Jones, N. S., J. Bradley. (2015). *Json web signature (jws)* (RFC No. 7515). RFC Editor. RFC Editor. https://www.rfc-editor.org/rfc/rfc7515

Maintainers, E. (2021). The elm architecture. Retrieved September 16, 2022, from https://guide.elm-lang.org/architecture/

NPM Maintainers. (2022a). Package-lock.json - a manifestation of the manifest. Retrieved September 23, 2022, from https://docs.npmjs.com/cli/v8/configuring-npm/package-lock-json

NPM Maintainers. (2022b). Package.json - specifics of npm's package.json handling. Retrieved September 23, 2022, from https://docs.npmjs.com/cli/v8/configuring-npm/package-json

Ombredanne, P. (2017). Package-url specification.

OWASP Foundation. (2022). Cyclonedx specification version 1.4. Retrieved August 30, 2022, from https://cyclonedx.org/docs/1.4/json/%5C#components_items_type

Python Software Foundation. (2022a). *Pypi requirements file format*. Retrieved September 1, 2022, from https://pip.pypa.io/en/stable/reference/requirements-file-format/

Python Software Foundation. (2022b). *Python standard library documentation*. Retrieved September 2, 2022, from https://docs.python.org/3/library/importlib.metadata.html

Python Software Foundation. (2022c). *Python warehouse json api*. Retrieved September 2, 2022, from https://warehouse.pypa.io/api-reference/json.html

Riehle, D. (2000). *Framework design: A role modeling approach* (Doctoral dissertation). ETH Zurich.

Robert Griesemer, K. T., Rob Pike. (2009). Hey! ho! let's go!

Syft Maintainers. (2022). Syft github repository. Retrieved September 23, 2022, from https://github.com/anchore/syft/tree/main/schema

Telecommunications, N. N., & Administration, I. (2021). *The minimum elements for a software bill of materials (sbom)* (tech. rep.). NTIA National Telecommunications and Information Administration. "1401 Constitution Ave., NW Washington, DC 20230".

The Go Maintainers. (n.d.). *Go modules reference*. Retrieved September 3, 2022, from https://go.dev/ref/mod%5C#go-mod-file

The Go Maintainers. (2022a). Go 1.18 release notes. Retrieved August 21, 2022, from https://tip.golang.org/doc/go1.18

The Go Maintainers. (2022b). *Pkg.go.dev documentation*. Retrieved September 3, 2022, from https://pkg.go.dev/golang.org/x/mod/modfile

The NPM Maintainers. (2022). Npm registry api. Retrieved August 22, 2022, from https://github.com/npm/registry/blob/master/docs/REGISTRY-API.md

The Protocol Buffer Maintainers. (2022). Protocol buffer language specification v3. Retrieved July 29, 2022, from https://developers.google.com/protocol-buffers/docs/proto3

Werner, F. (2021). *Using grpc for (local) inter-process communication*. Retrieved September 15, 2022, from https://www.mpi-hd.mpg.de/personalhomes/fwerner/research/2021/09/grpc-for-ipc/