



Degree Project in Technology

Second cycle, 30 credits

# The Embedding and Retrieval of Software Supply Chain Information in Java Applications

DANIEL WILLIAMS



# **The Embedding and Retrieval of Software Supply Chain Information in Java Applications**

DANIEL WILLIAMS

Date: 9th August 2024

Supervisor: Aman Sharma

Examiner: Martin Monperrus

School of Electrical Engineering and Computer Science

Swedish title: Inbäddning och extraktion av mjukvaruleverantörskedjan i  
Java-applikationer



## Abstract

As attacks on the software supply chain become increasingly prevalent, many steps have been taken to provide further insight into the components that make up the software. A common way to do this is through a software bill of materials – a document providing information about each component in the supply chain as well as their relationships. However, these are often generated by statically inspecting a project and contain no information about which code is actually executed at runtime. Knowing what components are used and when can be useful in many ways, such as for dependency trimming or dynamic vulnerability querying.

In this thesis, we present and evaluate a collection of tools, Classport, which can be used to produce runtime-aware dependency representations for Java programs built using the Maven build system. This is achieved in two main steps. First, we use a Maven plugin to automatically embed supply chain information inside the class files required by the project, including those contained in its dependencies. Second, we use a Java agent to monitor the loaded classes at runtime and extract the supply chain information from them.

We evaluate Classport by testing it on eight open source projects, comparing its dependency representations to those provided by Maven itself. We also investigate side effects of the embedding, such as the size increase and whether it leads to changes in the program's behaviour. The results show that we are able to achieve our goal of accurately reflecting the runtime state of the program in terms of its loaded dependencies. Additionally, the results show that while our method of adding metadata does not seem to affect the correctness of a program, it does noticeably increase its size. Making the impact of Classport less noticeable to the average user could increase the possibility of mainstream adoption, and is thus an appropriate candidate for future work.

## Keywords

Java, Metadata, Software supply chain, Maven



## Sammanfattning

Den så kallade mjukvaruleverantörskedjan, bestående av alla komponenter som används i ett programs utveckling, har blivit ett allt vanligare mål för cyberattacker. I takt med detta har ett flertal lösningar tagits fram för att öka förståelsen för dessa komponenter – exempelvis den kod som ingår i mjukvaran. En materialförteckning för mjukvara (eng: software bill of materials) används ofta för detta ändamål, och innehåller bland annat information om ett programs komponenter samt deras relation till varandra. De allra flesta sådana förteckningar genereras dock medan koden byggs eller efter att den packats ihop till en fil, och har ingen insikt i vilka delar av koden som faktiskt exekveras. Kunskap om detta kan vara användbart på många sätt, till exempel för att hitta tredjepartskomponenter som inte används eller för att undersöka koden för kända sårbarheter under tiden den körs.

I detta examensarbete presenterar och utvärderar vi en samling verktyg, Classport, som kan användas för att producera representationer av mjukvaruleverantörskedjan som tar hänsyn till vilken kod som faktiskt exekveras. Verktygen är utvecklade för att användas på Java-projekt som använder byggsystemet Maven, och består av två huvudsakliga steg. I det första används Maven för att identifiera alla klassfiler som används i projektet, och information om deras ursprung bäddas in i dem. I det andra steget används en Java-agent för att fånga upp alla klasser som laddas in för exekvering, varpå den tidigare inbäddade information extraheras igen.

Vi utvärderar Classport genom att testa verktygen på åtta projekt med öppen källkod, och jämför informationen det ger oss med den information som tillhandahålls av Maven självt. Vi undersöker även hur vår inbäddade data påverkar applikationen. Resultaten visar på att vi med hög precision kan återge både vilken kod som laddats in under körtiden och dess ursprung. Den inbäddade informationen verkar inte heller ha någon påverkan på programmets funktion. Däremot leder den till att programfilerna blir märkbart större, vilket riskerar att minska verktygens användningsområde. Hur dessa storleksökningar kan undvikas utan att minska mängden extraherbar information vore således ett lämpligt område för framtida forskning.

## Nyckelord

Java, Metadata, Mjukvaruleverantörskedjor, Maven





## Acknowledgments

I would like to extend my sincerest thanks to everyone who has made this thesis possible and supported me along the way.

To my supervisor, Aman Sharma, for his relentless encouragement and time spent helping me improve my work.

To my examiner, Prof. Martin Monperrus, for his invaluable ideas, feedback, and contagious passion.

To my fiancée, Madeleine, for always believing in me and never letting me forget it.

To my friends and family, for their continuous love and unwavering support.

Stockholm, August 2024

Daniel Williams



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Overview . . . . .	2
1.2	Research Questions . . . . .	3
1.3	Goal and Purpose . . . . .	4
1.4	Contributions . . . . .	4
1.5	Structure of the thesis . . . . .	5
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	The software supply chain . . . . .	7
2.2	SBOMs . . . . .	8
2.2.1	Current challenges . . . . .	8
2.3	Java . . . . .	10
2.3.1	The <code>class</code> File Format . . . . .	10
2.3.2	Class loading . . . . .	12
2.3.3	JAR files . . . . .	12
2.3.4	Java agents . . . . .	13
2.4	Maven . . . . .	13
2.5	Related work . . . . .	14
2.5.1	Fine-grained dependency querying . . . . .	15
2.5.2	Provenance identification . . . . .	16
2.5.3	Compiler-embedded dependency information . . . . .	17
<b>3</b>	<b>Design and Implementation</b>	<b>19</b>
3.1	System Overview . . . . .	19
3.2	Embedding the Supply Chain . . . . .	21
3.2.1	The Embedder . . . . .	21
3.2.2	Embedding as annotations . . . . .	22
3.3	Extracting the Supply Chain . . . . .	24
3.3.1	The Static Analyser . . . . .	24

3.3.2	The Dynamic Analyser . . . . .	24
3.3.3	Reconstructing the tree . . . . .	26
3.4	Additional capabilities . . . . .	28
<b>4</b>	<b>Experimental methodology</b>	<b>29</b>
4.1	RQ1 – To what extent can supply chain information be embedded in Java artefacts? . . . . .	30
4.2	RQ2 – To what extent can embedded supply chain information be retrieved at runtime? . . . . .	31
4.3	RQ3 – How applicable is Classport on real-world programs? .	33
<b>5</b>	<b>Experimental results</b>	<b>35</b>
5.1	RQ1 – To what extent can supply chain information be embedded in Java artefacts? . . . . .	35
5.2	RQ2 – To what extent can embedded supply chain information be retrieved at runtime? . . . . .	38
5.3	RQ3 – How applicable is Classport on real-world programs? .	39
5.4	Reliability Analysis . . . . .	40
<b>6</b>	<b>Discussion</b>	<b>43</b>
6.1	Limitations . . . . .	43
6.2	Sustainability and Ethics . . . . .	44
6.3	Validity Analysis . . . . .	44
6.4	Future work . . . . .	44
6.4.1	Addressing limitations . . . . .	45
6.4.2	Software Bill of Materialss (SBOMs) and Vulnerabil- ity querying . . . . .	45
6.4.3	Improving applicability . . . . .	46
<b>7</b>	<b>Conclusions</b>	<b>47</b>
	<b>References</b>	<b>49</b>
<b>A</b>	<b>Supporting material</b>	<b>57</b>

# List of Figures

1.1	Simplified overview of a build system capable of producing a compile-time SBOM . . . . .	2
2.1	An excerpt of a CycloneDX SBOM [26] . . . . .	9
3.1	An overview of Classport's design . . . . .	20
3.2	An overview of the tree generation algorithm . . . . .	26



# List of Tables

4.1	The projects chosen for evaluation and their number of dependencies (counted using the Maven dependency plugin) .	30
4.2	The projects and their corresponding workloads . . . . .	32
5.1	Static dependency list results for the tested projects . . . . .	36
5.2	Dynamic dependency list results for the tested projects . . . .	39
5.3	Tests passed and application size before and after embedding the Classport metadata . . . . .	40





# List of Listings

3.1	The Classport annotation . . . . .	23
3.2	Abbreviated <code>diff</code> output after embedding an annotation in the main class of the Graphhopper project's web module . . . . .	25



## List of acronyms and abbreviations

ABOM	Automatic Bill of Materials
API	Application Programming Interface
JAR	Java Archive
JVM	Java Virtual Machine
JVMS	Java Virtual Machine Specification
OmniBOR	Universal Bill of Receipts
POM	Project Object Model
SBOM	Software Bill of Materials
SSH	Secure Shell



# Chapter 1

## Introduction

Supply chain security has become more and more relevant in recent years, with calls for improvements by both the United States [1] and the European Union [2]. A compromised supply chain does not affect only one entity and has the potential to cause a great deal of harm, as can be seen in the SolarWinds attack [3] which left thousands of companies vulnerable, as well as the Log4Shell vulnerability [4] which is considered to be one of the most dangerous vulnerabilities ever [5]. A report published by the European Union Agency for Cybersecurity in 2021 [6] analysing the threat landscape for supply chain attacks found that in a majority of cases, attacks on the supply chain target their code rather than data, configurations or other assets. Thus, knowing what code is part of an application's supply chain is perhaps the most important step when identifying threats to it.

To keep track of what third party code is being used in a project, it is common to make use of a Software Bill of Materials (SBOM). An SBOM is a formal record containing information about the components of an application and their relationships within the supply chain [7]. Each component is represented by an entry containing metadata such as its name, checksum, version and other unique identifiers. This allows anyone who wishes to build and/or run that project for themselves to see what components are being used, run them through vulnerability databases, check their licenses and much more. The process of generating an SBOM is not always straight-forward, however, as it tends to vary based on either the language or build system used. Even when focusing on a specific language or build system, perfect accuracy can be hard to achieve. Recent research has shown that out of six major SBOM producers for the Java programming language, none were able to correctly enumerate all of a project's dependencies. [8]

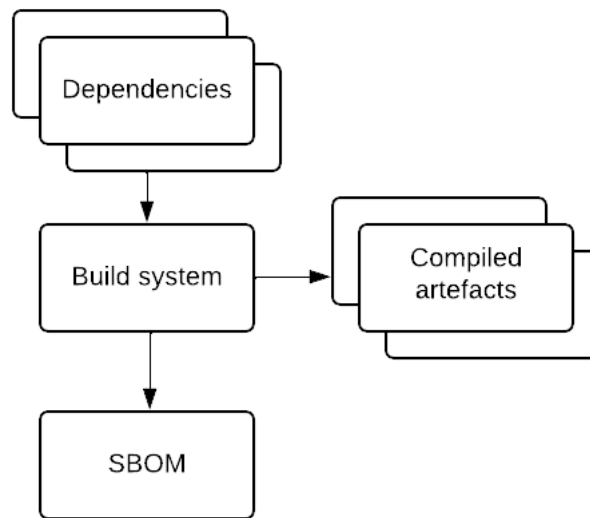


Figure 1.1: Simplified overview of a build system capable of producing a compile-time SBOM

But even if a fully reliable SBOM could be ensured, most well-known, open-source SBOM producers do not operate at runtime [9] and would thus not necessarily reflect the actual behaviour of the program. For example, some code may be included as a dependency but never actually used, be it due to code paths never being taken or simply forgetting to remove a now-obsolete dependency declaration. The inverse may also be true; in the Java programming language, for example, it is possible for code to be loaded or cached despite not being explicitly requested [10]. While current runtime SBOM producers for Java are able to detect this [11], they are also limited in both granularity [12] and properly identifying dependency metadata [8]. This thesis explores build-time embedding of said metadata as a solution to both of these problems, and evaluates its potential for real-world applications.

## 1.1 Overview

In this thesis, we present **Classport** – a collection of tools that work together to enable accurate runtime inspection of the supply chain in Java programs. Unlike many other SBOM producers, Classport begins by embedding supply chain metadata in each class file that is to be part of the application in question.

By monitoring which of these are then loaded by the running application, Classport is able to accurately connect them to their corresponding upstream dependencies and build a runtime representation of the application's code-related supply chain.

## 1.2 Research Questions

To evaluate Classport and its capabilities, it is tested on eight different open-source projects. Metadata is embedded into the project artefacts as well as its dependencies, and these are then packaged together into one self-standing, executable archive. Both static and dynamic supply chain representations can then be generated by Classport using only these archives themselves, without the need for a build system or other metadata. The generated representations are subsequently compared against those provided by the build system itself. Finally, the applications' tests are run to ensure stability, and the size of the Classport-built archive is compared to that without the Classport metadata. The results of these comparisons form the primary basis of the evaluation.

This process is also reflected in the research questions. These aim to investigate the problem of runtime supply chain representations from three main angles: embedding the metadata, accessing this metadata at runtime, and the feasibility of using similar technologies in real-world applications.

**RQ1** *To what extent can supply chain information be embedded in Java artefacts?* This research question aims to investigate how supply chain information can be automatically embedded in class files or Java Archives (JARs). This must be done in a way supported by the language, as to avoid crashes or other unexpected behaviour while using the embedded version of the program.

By using bytecode manipulation to place this information in a class-level annotation, we are able to achieve an ergonomic, reliable and extensible way of storing arbitrary data using only standard features of the language.

**RQ2** *To what extent can embedded supply chain information be retrieved at runtime?* To represent the runtime state of the program, the information embedded as part of RQ1 must also be accessible during program execution.

Using standard Java Application Programming Interfaces (APIs), we are able to develop a program that can be run alongside the main

application, intercepting close to all class files being loaded into it. Moreover, we are able to successfully parse their origins and build a complete, runtime-aware dependency tree.

**RQ3** *How applicable is Classport on real-world programs?* Technologies may face troubles being widely adopted unless the friction of doing so is sufficiently low.

We evaluate the methods used in this thesis not only by their success, but also by their effects on usability. We see that while embedded programs appear to work just as well as non-embedded ones, the embedding process does result in potentially significant size increases.

## 1.3 Goal and Purpose

The main goal of this project is to provide a way of inspecting the currently (i.e. at runtime) used subset of an application's supply chain. Such information could then be used for dependency trimming, runtime vulnerability querying, or other ways of hardening the security of the application.

Meanwhile, the purpose of the thesis itself is to provide insight into what can be done by utilising the flexibility of Java class files, and inspire more creative ways of working with the software supply chain. Thus, while the primary audience for this thesis is the academic community, it may also be of interest to anyone interested in the software supply chain or Java bytecode manipulation.

Advances in this area have the potential to directly benefit developers, who are able to gain increased insight into what code is being run by their applications. If this then helps the developers create better software, it would also benefit the end users of said software.

## 1.4 Contributions

The main contributions of this thesis are threefold.

1. We present a novel technique for runtime introspection of the supply chain.
2. We implement a suite of tools which serve as a proof of concept.
3. We analyse the success of our approach and provide suitable guidelines for future work.



## 1.5 Structure of the thesis

This thesis is divided into 7 chapters, of which this is the first. Chapter 2 presents relevant background information about the software supply chain, Java and Maven. It also discusses and analyses related work. Chapter 3 provides a view into the Classport tool suite, how it works and the reason behind certain design choices. Chapter 4 introduces the experiments used to evaluate Classport and its methods. Chapter 5 describes the results of these experiments and potential difficulties in obtaining them. Chapter 6 discusses the limitations of the tool as well as suggesting future work. Finally, Chapter 7 summarises and concludes the thesis.



# Chapter 2

## Background

### 2.1 The software supply chain

The software supply chain for a project is comprised of the code, tools and other services used to build, run, and distribute it. Knowledge of a project's software supply chain thus can be hugely beneficial in many ways, from ensuring license compliance and evaluating the reliability of dependencies to being able to scan a code base for indirect, or supply-chain, vulnerabilities.

In modern code bases, the software supply chain is seldom comprised of only in-house code, and there are often substantial dependencies on libraries or tools developed by third-parties [8]. In the ninth edition of the Open Source Security and Risk Analysis Report [13], Synopsys analysed 1067 commercial codebases across 17 fields during 2023 and found that 96% of these contained open source code, with an average of 526 open source dependencies per project. This reliance on third-party code is often very convenient, since taking on external dependencies can make the development process a lot quicker, but it is not always that said dependencies continue to be properly monitored for updates and vulnerabilities. According to Synopsys, in 91% of the codebases analysed, at least one of the open source dependencies was 10 or more versions behind the latest. This suggests that even after patches are available for disclosed vulnerabilities, they might not be adopted in their 'parent' codebases until a much later date. An example of this can be seen with the Log4Shell vulnerability, where Zhang *et al.* [14] analysed the dependency trees of 9220 GitHub repositories and found that while 35% of these updated their Log4J version within three months of the disclosure, it took 308 days until half of the repositories were no longer vulnerable.

The software supply chain has thus become an increasingly attractive

attack vector [15], with its complexity inspiring more and more sophisticated attacks [16]. And while some exploit existing vulnerabilities [17], others create their own. As recently as March 2024, a backdoor was found to have been introduced in the `xz` utility in order to bypass the authentication process of certain Secure Shell (SSH) versions [18]. The fact that this attack rendered a tool as ubiquitous as SSH vulnerable despite it not being directly compromised, highlights the importance of supply chain security in modern software.

## 2.2 SBOMs

An SBOM is a human and machine readable document containing information about a project and its supply chain; it is essentially the software version of a list of ingredients [19]. The three formats currently deemed appropriate [7] are Software Package Data eXchange (SPDX) [20], CycloneDX [21], and Software Identification (SWID) tags [22]. Each SBOM format is also required to contain a minimum set of data fields for each component in the supply chain [23]. First is the name of the supplier – or manufacturer – of the component. Second is its name, version, and other identifiers such as Package Uniform Resource Locators [24] or Universal Bill of Receipts (OmniBOR) [25] IDs. Last are dependency relationships between components, the author of the SBOM, and its timestamp. An abbreviated excerpt of a CycloneDX SBOM can be seen in Figure 2.1.

### 2.2.1 Current challenges

Generating accurate SBOMs has proven to be a complex process. While an SBOM could be manually maintained, doing so is error-prone and requires substantial effort due to the large number of dependencies involved in most modern codebases [8, 13]. Instead, SBOMs are often generated by allowing a third-party tool (known as an SBOM producer) to scan the project in question.

Due to the differences between build systems and environments, an SBOM producer may only target a specific environment and expect to be invoked in a certain stage of the development cycle – each of which comes with their own pros and cons [27]. A study of six state-of-the-art SBOM producers for Java projects [8], for example, revealed that none of the producers were consistently able to generate fully accurate dependency information. Furthermore, only one of the producers evaluated, `jbom` [11], is capable of producing runtime SBOMs. Even then, the algorithm employed by `jbom` to detect runtime

```

{ "bomFormat": "CycloneDX",
  "specVersion": "1.2",
  "metadata": {
    "timestamp": "2020-08-03T01:28:52.765Z",
    "tools": [
      { "vendor": "CycloneDX",
        "name": "Node.js module",
        "version": "2.0.0"
      }
    ],
    "component": {
      "type": "library",
      "bom-ref": "pkg:npm/lhc-vdm-editor@0.0.1",
      "name": "lhc-vdm-editor",
      "version": "0.0.1",
      "licenses": [
        { "license": { "id": "Apache-2.0" } }
      ],
      "purl": "pkg:npm/lhc-vdm-editor@0.0.1",
      "externalReferences": [
        { "type": "website",
          "url": "https://github.com/CERN/lhc-vdm-editor#readme"
        },
        [...]
      ],
      [...]
    },
    [...]
  },
  "components": [
    { "type": "library",
      "bom-ref": "pkg:npm/puppeteer@1.19.0",
      "name": "puppeteer",
      "version": "1.19.0",
      "hashes": [ { "alg": "SHA-512",
                    "content": "d92e84e...4dfd65f" } ],
      "licenses": [
        { "license": { "id": "Apache-2.0" } }
      ],
      "purl": "pkg:npm/puppeteer@1.19.0",
      "externalReferences": [
        { "type": "website",
          "url": "https://github.com/GoogleChrome/puppeteer#readme"
        },
        [...]
      ],
      [...]
    },
    [...]
  ],
  [...]
}

```

Figure 2.1: An excerpt of a CycloneDX SBOM [26]

dependencies does not differentiate between a dependency being actively loaded and it simply being packaged together with one that is [12]. This limitation becomes especially noticeable in applications for which many dependencies are packaged together. For example, in an application with all dependencies bundled, jbm would consider all dependencies used regardless of which ones are actually loaded into the running process. All in all, these shortcomings illustrate the importance of continued SBOM research, especially within the field of runtime SBOMs.

## 2.3 Java

Java is a high-level, object oriented programming language that has been widely used since its first appearance in 1995. It differed from other popular languages at the time by compiling to platform-independent bytecode instead of the usual, platform-dependent, machine code – something that significantly simplified program distribution as Java programs were able to run on any platform with access to a Java Virtual Machine (JVM).

### 2.3.1 The `class` File Format

The Java compiler compiles the Java source code into bytecode, which is subsequently stored in files known as class files. Each class in the source code is written to its own class file, and the `java` tool can run these class files by loading them into a JVM. For a class file to be successfully executed, it must follow a specific format. This is known as the class file format and is defined in Chapter 4 of the Java Virtual Machine Specification (JVMS) [28].

All class files are required to begin with the bytes `0xCAFEBADE`; these are known as the class file's 'magic bytes'. After the magic bytes, a class file contains a minor and a major version. Since the format may vary between versions, these help the JVM determine how the class file should be processed.

The next section of the class file concerns the 'constant pool'. The constant pool contains all constants used in the class, such as strings, integers and names of classes or fields. This allows instructions to refer to any constants used by their index in the constant pool, rather than hard-coding them multiple times.

After the constant pool are the access flags. While this section consists of only two bytes, they are integral to the Java model as they make up the bitmask describing a class' properties and access permissions. For example, a bit may declare whether the class is `public` (accessible outside its package), `final`

(unable to be subclassed), or whether it is an enum or interface type rather than a ‘regular’ class.

After the access flags, the class file contains two entries known as `this_class` and `super_class`. These each contain an index into the constant pool, where the items at those indices represent the current class and its superclass respectively. For the `Object` class, which is the root of the class hierarchy and thus has no superclass, the `super_class` entry will contain a zero.

A class file also contains information about the class’ implemented interfaces, fields and members. Constants related to these, such as the class names of interfaces and return types, are stored in the constant pool and resolved at runtime.

The last part of each class file contains information about a class’ attributes. Attributes are things such as the name of the source file from which a class was compiled, its permitted subclasses, or its signature. The JVM requires a JVM to support a number of specific, predefined attributes such as these, but also allows for the definition of custom ones. While custom attributes could be very useful, a feature known as annotations was introduced in Java 5 [29]. This provided an easier, standardised way of adding metadata to code and is commonly used to increase the amount of information available to the compiler and other tools. For example, one might annotate a parameter with `@NotNull` to indicate that the value passed in should not be `null`, or one could annotate a method with `@Deprecated` to inform users that said method should no longer be used. Annotations can also be applied to classes, providing context for the entire class rather than just parts of it.

While annotations offer great benefits [30], they also have some practical limitations. Under the hood, an annotation type is an interface with zero or more declared methods. Each of these methods defines an element in the resulting annotation, where the method’s return type is the type of the element’s value [31]. Values can be either primitive types, Strings, class types, enums, other annotation types or an array of any of the above. Note that this means that nested arrays and other complex types are not permitted.

Moreover, to control the scope of the annotation, a `Retention meta-annotation` may be applied to any annotation type. This gives compilers and JVMs information about how long the annotation should persist. The possible values are `SOURCE`, where the annotation is only present in the source code and discarded by the compiler; `CLASS`, where the annotation is retained in the class file but may be discarded by the JVM at runtime; and `RUNTIME`, where the annotation is available even at runtime.

### 2.3.2 Class loading

Class loading is an integral part of any Java program; this is the process in which the bytecode for a class is located and converted into a Java object. It can be divided into three main steps: loading, linking and initialisation.

Loading is the process in which the binary representation of a class or interface is located, and a representation of said class or interface is created within the JVM [32]. This is handled by ‘class loaders’ in Java, and while the Java runtime provides three built-in class loaders [33] that support the majority of use cases, users are also able to define their own by creating a subclass of the `ClassLoader` class.

Linking is the process in which a loaded class is verified and prepared for being inserted into the runtime state of the JVM [34]. Verification ensures that the class file is structurally correct, and preparation ensures that any static fields are created and initialised to their default values. The linking process also involves symbol resolution, where symbolic references in the constant pool are resolved. Unlike verification and preparation though, symbol resolution may be performed lazily.\* For the sake of consistency, any resolution errors encountered must thus also be thrown lazily, regardless of when the actual resolution is performed.

Initialization is the final step of the process. In this step, any final static fields are initialised with their constant values, and a special initialisation method called `<clinit>` is executed [35]. Despite having only two responsibilities, the initialisation stage requires careful synchronisation as multiple threads may try to initialise a class at the same time.

### 2.3.3 JAR files

To distribute Java projects, class files and other resources are often packaged into a single JAR file [36]. The JAR file format is built on top of the ZIP format, meaning that it natively supports both archiving and compression. The JAR format also supports features such as executable archives and cryptographic signing through the use of Manifest files – a special file stored in a JAR, containing information about the packaged files [37].

---

\*The concept of laziness in programming often means to avoid doing the work until it’s required, e.g. not resolving the reference until its value will actually be used.



### 2.3.4 Java agents

A Java agent is a specific type of JAR file meant to run alongside other programs. Whenever a program is started with the `-javaagent` command-line flag, the specified agent can be used to monitor and change the program's classes before they are loaded into the JVM. This makes agents well suited for tasks such as enhanced logging or benchmarking, as code can be injected at runtime to help perform these actions without having to include it in the source.

An agent is implemented using the Instrumentation API, which provides the `Instrumentation` interface. This interface contains methods used to modify objects and properties integral to the class loading process, and classes implementing it are only intended to be instantiated as arguments to an agent's entry point: the `premain` or `agentmain` methods. Agents can then intercept the class loading process by using this object to register `ClassFileTransformers` – classes whose `transform` methods get called for every loaded class before it is defined by the JVM. Each of these calls contains information about the class to be loaded, such as its name, loader, and the contents of its class file as an array of bytes. The agent can either choose to transform this class, in which case it should return a new array consisting of the changed class file bytes, or to not transform it, in which case it should return `null`.

## 2.4 Maven

Apache Maven is a project build and management tool, used primarily for Java-based projects. It aims to facilitate all parts of the build process, from downloading dependencies and compiling the project to packaging and deploying it. This is done declaratively, using only a Project Object Model (POM) file and a set of plugins, resulting in a uniform but powerful build process. For example, to include a dependency in a project, an entry would be added to the POM containing three main attributes: the group ID, which represents the group or organisation that developed the dependency; the artefact ID, which can be thought of as the name of the dependency; and the desired version or version range. Maven also allows developers to create their own plugins to perform custom tasks such as lockfile generation [38] or class file manipulation [39]. The latter is greatly helpful for this project, as Maven also contains supply chain information for each dependency.

The core concept of Maven is that of a build lifecycle, where each lifecycle

handles different parts of the project management. There are three built-in lifecycles: `default`, which builds, packages and deploys a project; `clean`, which cleans up the project; and `site`, which builds the web site for the project. A lifecycle is in turn made up of a list of stages, or build phases. The `default` lifecycle which is arguably used the most, contains a number of ordered phases such as `compile`, `test`, and `deploy`. When invoking Maven from the command line, it expects one or more of these phases to be specified as a target. It will then run all phases up to and including the one specified, according to their predefined order [40]. But despite these phases being part of the build process, they do not do anything by themselves. In order for a phase to execute, there needs to be at least one plugin goal bound to it. A plugin goal is a task defined by a plugin, and binding it to a phase means that it is automatically executed every time said phase is run. There is a set of bindings automatically defined for each way a project can be packaged using Maven [41], and any additional bindings are declared in the project's POM file. A goal can also be invoked directly from the command line, in which case it does not need to be bound to any phase at all.

Moreover, a plugin is able to require certain things of Maven before it begins execution. For example, a plugin that needs access to dependency information can specify `requiresDependencyCollection`. This will list information about group IDs, artefact IDs and versions by downloading the required POM files. If a plugin needs access to the actual class files and not just the POM, it can specify `requiresDependencyResolution` instead which will download the artefacts as well.

## 2.5 Related work

This section documents previously developed tools and solutions relevant to this thesis. It discusses three fields centred around the software supply chain. First is fine-grained dependency querying, which involves querying a project for whether it was built using a specific source file. Next is provenance identification, which is about identifying the origins of software components. Finally, this section highlights work related to compiler-embedded dependency information, where the compiler is used to automatically embed supply chain information in the resulting artefact.

### 2.5.1 Fine-grained dependency querying

Boucher and Anderson [42] propose a technique called Automatic Bill of Materials (ABOM). The main idea of ABOM is being able to query an executable for a dependency in order to facilitate vulnerability detection. To do this, ABOM utilises the compiler to gather dependency information, computes the SHA-3 hash of each input file, and stores the hashes in a compressed Bloom filter [43]. This data structure is then embedded into a dedicated section of the output file. Naturally, there will also be times where an upstream dependency itself contains an ABOM. In these cases, once the main ABOM is produced, it will be merged with any upstream ABOMs before being embedded into the binary.

As (compressed) Bloom filters are probabilistic data structures, there is a chance that a non-vulnerable binary will be incorrectly reported as vulnerable. As such, the specific implementation used in ABOM was carefully tuned to achieve a low false-positive rate while still being small enough in size to avoid memory issues on any modern hardware.\* Furthermore, while querying a Bloom filter may produce a false positive, a false negative will never happen. Thus, a binary will always correctly report the cases in which it *does* contain a vulnerable dependency. In cases where a binary is marked as being vulnerable, the list of hashes comprising the Bloom filter can be recreated and used as a witness to either validate or invalidate the result. However, doing so requires access to the input files – something which is not always guaranteed.

Seshadri *et al.* [25] propose OmniBOR – a framework for building dependency graphs from source code files. The main idea of OmniBOR is that each output artefact has a corresponding Input Manifest (IM) containing the hashes of each input artefact directly used to build it. If one of these input artefacts, A, were to have an IM itself, the hash of A's IM is also stored next to the hash of A. These IMs are stored on disk with filenames based on their hashes, constituting a Merkle tree. When the final artefact has been built, the hash of its IM is embedded into it allowing tools to locate the root of the IM tree.

Since OmniBOR packages most of its dependency graph in auxiliary files, it does not suffer from the same space constraints as ABOM. Thus, the authors propose including further metadata about the build process in a file whose name is derived from the output artefact's hash. While they do not explicitly list what the contents of this file should be, their suggestions include names

---

\*Whether the false positive rate used is 'rare enough' is of course subjective, something that was acknowledged by the authors in the original paper.

and paths of dependencies, as well as build commands. Having access to these could be useful, especially since the output file often looks different depending on which options (such as the level of optimisation) it was compiled with.

## 2.5.2 Provenance identification

Davies *et al.* [44] propose a framework they call software Bertillonage, which can help locate a Java Archive file in a list of candidates (such as the Maven central repository) by computing their signatures. This signature is defined in the original article as being the set of signatures for each class present within the archive. A class' signature, in turn, is defined as a tuple of its type signature and the set of type signatures for each of its methods. To find the best matches for an archive  $A$ , its signature is compared against the signature of each candidate  $B$  by computing a 'similarity index' between them:

$$\text{sim}(A, B) = \frac{|\vartheta(A) \cap \vartheta(B)|}{|\vartheta(A) \cup \vartheta(B)|}$$

Where  $\vartheta(X)$  represents the signature of an archive  $X$ . The candidates with the highest similarity index are considered to be the best matches. The authors tested their approach on a proprietary Java application and found that the binary archives of 48 out of its 84 open source dependencies were correctly and uniquely identified. In 19 other cases the tool found multiple, equally likely, matches of which one was correct. Their tool was also able to perform binary-to-source matching, but this performed worse than the binary-to-binary matching – most likely since their dataset contained more than six times as many binary archives as source archives.

The concept of machine learning has also made its way into provenance detection. Jang, Murodova and Koo present ToolPhet [45], a system that can be used to trace a binary executable back to the compiler toolchain used to generate it. ToolPhet uses two fine-tuned models: one that looks for internal features specific to a certain toolchain, and one that detects similarities in the semantics of generated code. Compared to a well-known, signature-based system [46], ToolPhet was able to correctly predict the toolchain in almost twice as many cases. ToolPhet also performed on par with, or better than, other state-of-the-art machine learning approaches.

### 2.5.3 Compiler-embedded dependency information

Information about software dependencies is traditionally stored in the source repository and not necessarily distributed alongside the compiled artefacts. Despite this, there are some cases in which the compiler automatically embeds such supply chain information as part of the build process.

In the Go programming language, the standard compiler unconditionally embeds a data structure in the resulting binary containing metadata about the project [47]. This includes the names of every source code file used as input for the compilation, as well as names of all the functions and what packages they originate from. While this information is primarily useful to the Go runtime, it is also interesting from a supply chain standpoint. Having such information embedded means that the list of package dependencies for an artefact can easily be extracted by a third party without requiring any auxiliary files.



## Chapter 3

# Design and Implementation

This chapter presents an overview of the resulting tool suite, Classport\*, and aims to explain the reasons behind its design choices. Classport is made up of several components with two main goals: embedding supply chain information in Java artefacts, and being able to extract this information at runtime. The components thus work together to provide a way of generating supply chain representations for an application, using only the class files within it. Classport targets applications meant to be packaged into executable ‘Uber-JARs’ (henceforth just JARs or ‘executable JARs’ depending on context), and once these are built, dependency representations can be generated from them both statically (i.e. without running the JAR) and dynamically. Dynamically generated representations only include dependencies loaded by the JVM during execution, and can thus be used to inspect and learn about the runtime supply chain.

A diagram of the process can be seen in Figure 3.1.

### 3.1 System Overview

The first part of Classport is its Maven plugin. The plugin expects to be run in the root directory of a Maven project, where it will begin by analysing the project’s POM file and identify all its dependencies. Both direct and transitive dependencies are included, and before the plugin proceeds, it will ensure that any such dependencies are downloaded onto the local machine. Classport will then run each dependency JAR through an embedding algorithm. This algorithm begins by creating copies of all class files in the dependency, and

---

\*Classport is openly developed at <https://github.com/chains-project/classport>.

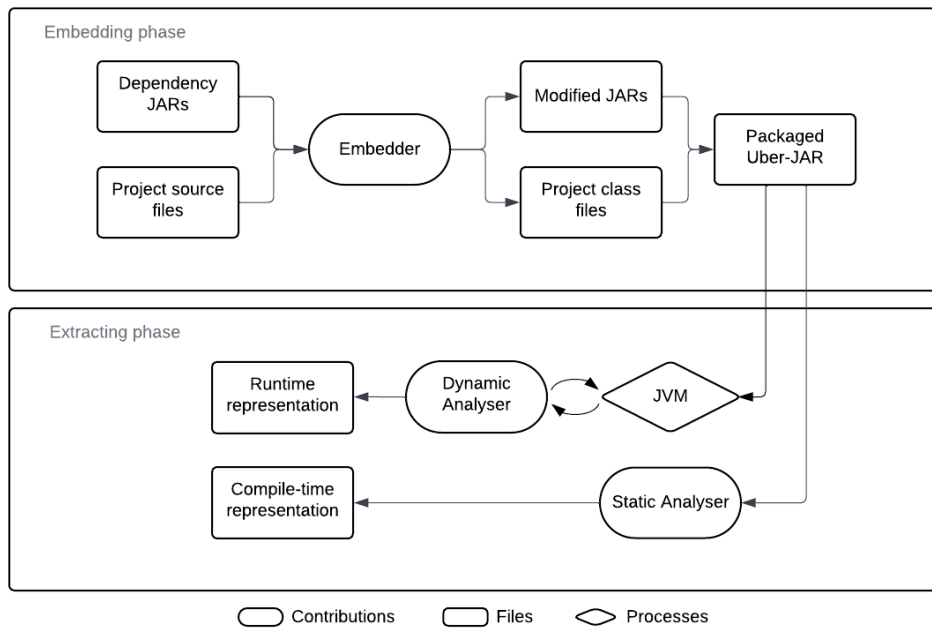


Figure 3.1: An overview of Classport's design

embeds supply chain information within them. Next, it re-packages these modified class files into a new JAR file with the same structure as the original. Finally, the project is packaged with Maven using these new JAR files in place of the original dependencies.

To generate a runtime dependency tree from the resulting JAR, it should be run with the Classport Dynamic Analyser. The Dynamic Analyser will start by registering a shutdown hook with the JVM. This allows it to run a specified block of code during the JVM's shutdown process, when no further dependency classes should be loaded. Next, the Dynamic Analyser will hook into the class loading process. This allows it to intercept each loaded class and attempt to extract the supply chain data from it. Any classes for which this succeeds will be logged together with their provenance information. Finally, during shutdown, the Dynamic Analyser will produce two files: one containing a list of all loaded dependencies, and one containing a tree.

The Classport Static Analyser can also be used to generate these dependency representations without having to run the application. This tool expects a JAR file to be provided as a command-line argument, and will go through all class files within the specified JAR to extract their supply chain data. Once the entire JAR file has been scanned, the tool can output either



a list or a tree of the dependencies. Since this tool has no notion of what is ‘used’ or not, it will always output a complete dependency list or tree for the JAR in question.

## 3.2 Embedding the Supply Chain

The first part of the Classport process concerns the embedding of supply chain information. This embedding process must be automatic, as modifying class files one-by-one would be far too cumbersome and not scalable. To implement this, the Maven build system was chosen due to its excellent plugin support. The Maven API allows plugins to interact deeply with the build process itself, e.g. by obtaining a project’s dependencies – both direct and transitive – and working with them as native Java objects. While this means that only projects using Maven are supported by Classport, the underlying theory is build system agnostic and future work could expand on this to target other build systems or languages as well.

### 3.2.1 The Embedder

Maven resolves external dependencies by locating their artefacts in an external repository and downloading them to a local repository on disk. Since Classport can only operate on dependencies that exist locally, the Maven plugin API is used to ensure that all dependencies are downloaded to the local repository before execution. Moreover, the plugin also ensures that the project is compiled before it proceeds. This allows embedding information in the project’s own class files too, instead of just those contained in its dependencies.

To obtain the maximum amount of information for the dependencies, the Embedder once again uses the Maven API to model each dependency as a project of its own. This enables it to extract information such as transitive dependencies, which would not otherwise be obtainable. Once it has the required information, Classport starts processing the files within the dependency JARs. There are four different file types that need to be considered. First, if a file is a class file (as determined by the magic bytes discussed in Section 2.3.1), Classport copies it into a new JAR, embedding any supply chain information in the process. Second, the file may be the JAR’s manifest (see Section 2.3.3). Since the class files within the JAR are changed as part of the embedding process, any signatures in the manifest related to

class files will no longer match.\* Thus, all signature-related entries are simply removed from the manifest, and the remainder of the file is copied over to the new JAR. The third type of file is a signature file, which is automatically generated when signing a JAR. These have no purpose outside of being a part of the signature, and copying these can thus be skipped completely. All other files fall into the fourth and final category of requiring no special treatment, and are therefore simply copied to the new JAR without modification.

For the re-packaged JAR files to be of use, Maven has to be able to locate them during dependency resolution. Classport helps out with this by placing them in a new directory whose structure adheres to the Maven repository layout, meaning that it can serve as a drop-in replacement for Maven's own local repository. By overriding the local repository using the `-Dmaven.repo.local` command line flag, the project can thus easily be packaged using the new JAR files in place of the originals.

### 3.2.2 Embedding as annotations

When embedding information inside class files, there are a few desirable properties. First, the metadata should be added in a supported way, meaning that the class file should be well-formed and not cause any problems even if the project is being run without Classport. For example, prepending or appending the dependency information to the class file bytes directly is not acceptable, since this would result in a malformed class file. Methods of questionable or unknown validity are also ignored, such as adding entries directly to the constant pool. While this does not appear to be explicitly prohibited, it is not explicitly encouraged either and was therefore not deemed reliable enough as a solution. Next, the presence of the metadata should have minimal effect on performance. Accessing it should be efficient, and when it is *not* being accessed, the only overhead required should be that of parsing the extra bytes. Finally, the format should be flexible and easy to work with. While this may not impact the end user at all, it does impact the developer experience and thus also its adoption potential.

As noted in Section 2.3.1, the attributes section of a class file accepts arbitrary developer-defined entries, making it a good candidate for the embedded information. However, the manual for the popular bytecode manipulation tool ASM [49]<sup>†</sup> discourages the use of 'raw' attributes unless

---

\*When a JAR is signed, an entry is added to the manifest for each file within the JAR. This entry contains the file's name and the hash of its contents [48].

<sup>†</sup>ASM is the bytecode manipulation library used by Classport. For future solutions, the

necessary, since working with annotations is a lot more ergonomic [51]. Annotations can also be configured to remain accessible at runtime using reflection. This means that any overhead of extracting the annotation from a class can be deferred until a time where this is deemed acceptable. With the aforementioned benefits in mind, a custom annotation type was created to hold the necessary metadata. The definition of this annotation can be seen in Listing 3.1, and is inspired by the minimum SBOM requirements discussed in Section 2.2.

---

**Listing 3.1** The Classport annotation

---

```
,
@Retention(RetentionPolicy.RUNTIME)
public @interface ClassportInfo {
    String sourceProjectId();

    boolean isDirectDependency();

    String id();

    String artefact();

    String group();

    String version();

    String[] childIds();
}
```

---

Embedding this annotation adds a few entries to the class file's constant pool corresponding to the data contained in its fields, as well as a single entry in the `RuntimeVisibleAnnotations` attribute. This is shown in brief in Listing 3.2. While other sections of the class file may change solely as a result of running them through ASM (and independently of Classport-related operations), it is trusted that no such changes affect the class files' validity in any meaningful way.

Finally, as annotations do not support nested lists nor complex objects such as maps, only direct dependencies for an artefact are stored in the Classport annotation. This is not a problem in itself, as transitive relationships can be naturally discovered by recursively inspecting direct dependencies. While entire dependency trees could be serialised and embedded as strings or bytes,

---

native `java.lang.classfile` package [50] previewed in Java 22 may also be a viable alternative to avoid external dependencies.

this would increase the size of the annotations substantially and was thus not deemed a viable solution.

### 3.3 Extracting the Supply Chain

Once the application has been re-packaged with the new JAR files, Classport can build dependency representations both statically and dynamically using only the application JAR as input. This is done in two steps. The first involves parsing each class file to obtain the embedded dependency information. The second step formats this information and presents it to the user.

#### 3.3.1 The Static Analyser

To statically analyse a JAR file, Classport makes use of the JAR API provided by the Java standard library to walk through its files. The first four bytes of each file is checked to see whether it is a class file or not (see Section 2.3.1). For each class file, the Static Analyser attempts to parse the annotation using the ASM library and stores each parsed annotation in a map using its `id` as the key. Any class files with missing annotations are immediately reported as warnings.

#### 3.3.2 The Dynamic Analyser

To represent the runtime state of a program as accurately as possible\*, Classport uses a Java agent to intercept all loaded classes, regardless of their loading class loader. This is made possible by registering a `ClassFileTransformer` using Java's Instrumentation API. This transformer can then read the class file bytes as mentioned in Section 2.3.4, parse the annotation (if it exists), and allow the JVM to continue the loading process without performing any transformations on the class.

While this method is generally highly accurate, being able to see *when* each dependency is loaded in relation to the others, there are three main limitations to it. First, the agent will not see any classes loaded before the transformer has been registered. Second, a class will only be interceptable if it is not required by any registered transformer [52]. This means that classes used to process the intercepted class files, such as parts of the ASM library, will not be passed through the agent. Finally, parsing each class on load incurs a

---

\*Even higher accuracy is possible, but this would require advanced interfacing with the JVM. Agents require only the standard library and can be written in pure Java.

---

**Listing 3.2** Abbreviated diff output after embedding an annotation in the main class of the **Graphhopper** project's web module
 

---

```

--- old 2024-06-06 18:27:46.296508314 +0200
+++ new 2024-06-06 18:29:59.133175820 +0200
@@ -1,6 +1,6 @@
[omitted: checksums and paths]
@@ -8,7 +8,7 @@
    flags: (0x0031) ACC_PUBLIC, ACC_FINAL, ACC_SUPER
    this_class: #7
    super_class: #2
-   interfaces: 0, fields: 0, methods: 5, attributes: 3
+   interfaces: 0, fields: 0, methods: 5, attributes: 4
    Constant pool:
        #1 = Methodref          #2.#3
        #2 = Class               #4
@@ -154,6 +154,24 @@
    #142 = Class                #143
    #143 = Utf8                 javax/servlet/FilterRegistration
    #144 = Utf8                 Dynamic
+   #145 = Utf8                 Lio/github/.../ClassportInfo;
+   #146 = Utf8                 group
[omitted: other added constant pool entries]
+   #161 = Utf8                 sourceProjectId
+   #162 = Utf8                 RuntimeVisibleAnnotations
    {
        public com.graphhopper.application.GraphHopperApplication();
        descriptor: ()V
@@ -314,7 +332,18 @@
        <no name>                synthetic
        <no name>                synthetic
    }
    InnerClasses:
        public static #144= #103 of #142;
    Signature: #138
+SourceFile: "GraphHopperApplication.java"
+RuntimeVisibleAnnotations:
+ 0: #145(#146=s#147,#148=s#149,#150=s#151,#152=s#153,#154=Z#155,...)
+   io.github.chains_project.classport.commons.ClassportInfo(
+     group="com.graphhopper"
+     version="9.1"
+     id="com.graphhopper:graphhopper-web:jar:9.1"
+     artefact="graphhopper-web"
+     isDirectDependency=false
+     childIds=["io.dropwizard:dropwizard-core:jar:2.1.11", ...]
+     sourceProjectId="com.graphhopper:graphhopper-web:jar:9.1"
+   )

```

---

certain amount of overhead which may be undesirable. The first limitation of classes being loaded before the agent is not considered a problem for Classport. This is due to the agent being invoked at the very beginning of the program, meaning that any classes loaded at this point will have been system classes and not from the program's dependencies. To mitigate the other two limitations (at the cost of not being able to know *when* a class was loaded), the agent could make use of the JVM's reflection capabilities instead, as mentioned in Section 3.2.2. This does not require any third party dependencies at all and, thanks to the `Instrumentation.getAllLoadedClasses()` method, allows the annotations to be read at any point in time.

### 3.3.3 Reconstructing the tree

Once the data collection is finished, the list of annotations obtained during extraction must be 'un-flattened' to accurately reflect the dependency tree. This is done in a depth-first manner, by first finding the root of the tree and then recursively looking up the dependencies contained in the `childIds` array. Figure 3.2 illustrates this with a simple example of three dependencies.

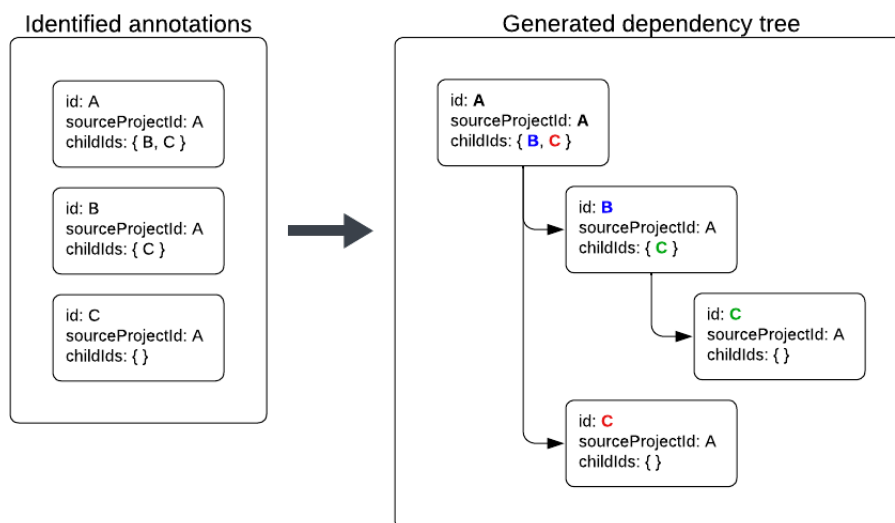


Figure 3.2: An overview of the tree generation algorithm

Note that dependency C is required by both A and B, meaning that it is added to the tree twice.

While reconstructing the tree, there were 3 main obstacles to overcome. First, the algorithm must know where to start. Since annotations contain no parent information\*, this is inferred using the `sourceProjectId` and `isDirectDependency` fields. By finding which dependency has an ID equivalent to the `sourceProjectId`, or if there is none, which dependencies are direct dependencies of the root project, Classport knows where to start the recursion.

Second, some dependencies may appear in cycles. For example, from the `org.apache.commons` group, the `commons-text` artefact depends on `commons-lang3` as a compile-time dependency. `commons-lang3` also depends on `commons-text`, but expects it to be provided at runtime. This forms a cycle which needs to be broken during the tree generation. Classport does this by taking note of which dependencies have already been visited in the current ‘path’, and only continues the recursion if the current dependency has not already been seen.

The third obstacle has to do with Maven’s dependency resolution algorithm. When resolving dependencies, Maven may encounter the same artefact declared several times in the tree. These declarations may also request different versions of this artefact. In these cases, Maven will choose the version requested by the dependency closest to the root artefact [53], and all other occurrences of that same artefact will be changed to use the same version. Thus, it is possible for version  $v_1$  to appear as a `childId` of some dependency in the tree, but only version  $v_2 \neq v_1$  is actually available due to being declared closer to the project root. In these cases, Classport must adapt to correctly use version  $v_2$  for the child dependency to match its annotation. This is done by first checking for the existence of the exact version specified by the `childId`. If this does not exist, Classport falls back to checking for existence of only the artefact, ignoring the version. If a match is found, this version is used instead.

If no match is found for an artefact, even when accounting for version mismatches, this is because Classport has not encountered any annotations for it while parsing class files. Usually, this is either because the artefact has not been included in the JAR (static analysis) or has not been used (dynamic analysis), and it will thus also be omitted from the dependency tree.

---

\*This was a deliberate choice due to artefacts being able to have multiple parents – the embedding process handles dependencies one by one, and accounting for previously embedded parent IDs would complicate the embedding for little reason.

### 3.4 Additional capabilities

In addition to the Embedder, Static Analyser and Dynamic Analyser, the Classport suite also contains a tool for modifying executable JARs. This modification ensures that at least one class is loaded from each dependency contained in the JAR, and was developed to aid in the evaluation of RQ2.

In total, two passes are made over the specified JAR file. In the first pass, the name of the JAR's main class (the class whose `main` method becomes the entry point) is retrieved, alongside the supply chain information, name and access modifiers for each class. Access modifiers are retrieved to ensure that only public classes are considered candidates for loading, as these are guaranteed to be reachable from the main class. Once the JAR has been fully traversed, one class from each dependency is selected. In the second pass, the tool makes an almost identical copy of the JAR file. However, when it encounters the main class, it will generate a function in which each of the selected classes is loaded. This function is then placed in the main class, and a bytecode instruction that calls it is placed at the beginning of the class' `main` method. Finally, this altered main class is used in the new JAR.

Due to how Maven dependency scopes work, it is possible (albeit rare in practice) that any one of the aforementioned class files cannot actually be loaded. This may happen if the loaded class depends on another class which is expected to be provided at runtime, but is not. To mitigate this, the tool allows for creating a deny-list of classes. This prevents them from being seen as candidates for loading in the first place, and can be used to force the tool into choosing another class to load.



## Chapter 4

# Experimental methodology

In order to evaluate how well Classport accomplishes the goals of this thesis and answers the research questions, it was tested on several real-world projects found on GitHub. There were 3 main requirements placed on these projects. First, they should be primarily written in Java and use Maven as their build system. While Classport should work with any JVM-based language, Java was chosen as it is the most popular of these. Second, building the project needs to produce an executable JAR, as this is what Classport was designed for. Third, the projects must be open-source as Classport needs to embed the metadata in the class files before the project is packaged.

For the purposes of this evaluation, a project's popularity was deemed a good indicator of its real-world usefulness. Thus, projects with more GitHub stars were preferred. A summary of the selected projects can be seen in Table 4.1, and while most of the projects were evaluated at their latest tagged releases, there were a few cases in which this was not preferable. For Checkstyle [54], the experiments were all conducted when checked out to commit 15e0c7e82, as this was the latest commit at the time of evaluation where all tests passed. For GraphHopper [55], some modifications had to be performed to be able to build the executable JAR for the web module alone. These modifications consisted of explicitly pinning the project to version 9.1 instead of 9.0-SNAPSHOT\*, so that dependencies could be resolved properly from Maven Central. For torrent [56], version 1.5 was used instead of the newer 2.0 due to test failures in the latter. However, two of the dependency versions required by torrent 1.5 were found to have invalid POMs. While this does not cause any build problems (as the dependencies' POMs are not required for building a project), it does prevent Classport

---

\*A patch file containing this modification can be found attached to Appendix A.

Table 4.1: The projects chosen for evaluation and their number of dependencies (counted using the Maven dependency plugin)

Project (module)	Commit SHA/Tag	Direct dependencies	Total dependencies
Checkstyle	15e0c7e82	24	71
Graphhopper (web)	9.1	10	201
PDFBox (app)	3.0.2	5	12
ttorrent (client)	ttorrent-1.5	3	7
jacop	4.10.0	6	12
jd-cli (jd-cli)	jd-cli-1.2.1	5	8
certificate-ripper	2.3.0	9	28
mcs	v0.7.0	11	76

from resolving dependency relationships during its embedding stage. As this resolution is required to obtain accurate Classport data, the first dependency, `slf4j-log4j12`, was manually updated from version `1.6.4` to `1.6.5`. The second problematic dependency, `net.sf.jargs`, has only one version published on Maven Central and could not be updated. Thus, the invalid section was manually removed\*. Finally, the `ttorrent` POM also had to be changed to target Java 8 instead of the now unsupported Java 6.

Additionally, both `ttorrent` and `jacop` had to be built with a Java version older than Java 17 – the version required by Classport. To resolve this, the Java version was set to Java 8 for building and testing these projects – which is done without Classport’s involvement – and to Java 17 when Classport was involved. All the other projects were compatible with Classport without modification.

## 4.1 RQ1 – To what extent can supply chain information be embedded in Java artefacts?

This research question aims to answer how fine-grained the embedding process can be made. The aim is to provide class-level accuracy, such that every class file contains information about which Maven dependency it belongs to.

---

\*The error in this case was that the `distributionManagement.status` value was set. This is not allowed as it is automatically set by Maven [57], and was thus safe to remove.

Ensuring that the annotation is embedded in each one – regardless of whether it is a direct or transitive dependency – is crucial in guaranteeing an accurate dependency tree since not every execution path of a program will require all class files. To answer the research question, the output from the Static Analyser is compared to the corresponding output from the Maven dependency plugin. The output from the Static Analyser represents all the information Classport has managed to embed, and the output from Maven represents all information available.

The evaluation is performed in three steps. In the first step, the Embedder is invoked to embed the supply chain information in the dependency class files, as shown in the embedding phase of Figure 3.1. The project is then packaged into an executable JAR using the embedded dependencies. Once packaged, the Static Analyser is invoked on the executable JAR to print both a dependency list and a dependency tree.

Second, the ground truth is generated using the Maven dependency plugin. More specifically, the commands `mvn dependency:list` and `mvn dependency:tree -Dverbose` are used to generate the dependency list and tree respectively.

Finally, the dependency list and tree generated by Classport are compared to those generated by the Maven dependency plugin. For the lists, this involves counting the total dependencies seen by each one, as well as investigating how the Classport list differs from that of Maven. If any false positives or false negatives are encountered, their reasons are explained. Comparing the trees also involves the analysis of false positives and false negatives, but in a more qualitative way compared to the list comparisons. This is partially due to the output being more complex, but also to allow for reasoning about the quality of the tree regeneration algorithm mentioned in Section 3.3.3, as it works differently than Maven's.

## 4.2 RQ2 – To what extent can embedded supply chain information be retrieved at runtime?

The overall goal of this research question is being able to report what dependencies within a running program are actually in use. This is evaluated by constructing dependency lists at runtime and comparing these to the static dependency lists achieved in RQ1. However, since certain dependencies may only be loaded for some specific control flow, the output from the Dynamic

Table 4.2: The projects and their corresponding workloads

Project	Workload
Checkstyle	Linting the Classport project
Graphhopper	Plotting a route with intermediate stops
PDFBox	Export the text from a PDF of this thesis
ttorrent	Downloading a small torrent
jacop	Solving an example input provided by the project
jd-cli	Decompiling a Classport class file
certificate-ripper	Printing certificates for <a href="#">the CHAINS website</a>
mcs	Searching for 'search'

Analysers is expected to vary based on which workloads are used. To account for this, the evaluation consists of three main steps.

For the first step, the application is packaged in the same way as for RQ1. The Dynamic Analyser is then run alongside the application during execution of a specific workload, to show how the runtime supply chain may differ from the static one. These workloads are meant to reflect a typical use case for each application, and are specified in Table 4.2. The torrent file used in the ttorrent evaluation can be found attached to Appendix A.

In the second step, Classport is used to generate a modified version of the application JAR using the method described in Section 3.4. This ensures that the set of runtime dependencies is no longer dependent on the control flow. The Dynamic Analyser is then run alongside this modified application, executing the same task as in step 1.

Finally, the lists and trees for both runs are compared against those output by the static analyser in the evaluation of RQ1, as this is the most amount of information obtainable by Classport. The list and tree produced by the unmodified application are thus expected to be smaller than those obtained through static analysis, due to only a subset of dependencies being used. In the case of the modified application, however, both the tree and the list are expected to contain the exact same items as their static counterparts, albeit perhaps in a different order.

### 4.3 RQ3 – How applicable is Classport on real-world programs?

There are two main aspects to Classport's applicability. First, it is important that embedding the metadata does not change the target program's behaviour. To evaluate this, the target projects' unit tests are run both with and without the embedded annotations to ensure that there are no changes in the number of passing tests. For multi-module projects, annotations are embedded for all modules and not just the ones tested in RQs 1 and 2, to ensure that as many tests as possible can be run. Second, the additional space required by the embedded annotations is measured to ensure that the spatial overhead is acceptable. This is only done for the modules investigated in the previous RQs, however, since these contain the executable JARs that constitute the actual applications. Boucher and Anderson's [42] ABOM tool achieves size increases of around 2% at most, which is considered a suitable target for Classport.

Evaluating Classport through the projects' test suites consists of three steps. First, its unit tests are run using the command `mvn test -fn` to ensure that all tests run, regardless of failures. Second, the Embedder is used to embed the annotations in the compiled class files, and the same tests are run again. Finally, for each project, the number of passing unit tests before and after the addition of Classport metadata are compared to each other.

To evaluate the size increase of the JARs packaged with Classport metadata, the applications are packaged twice: once with the Classport metadata and once without. The sizes of the resulting executable JARs are then measured using the `stat` tool.



## Chapter 5

# Experimental results

### 5.1 RQ1 – To what extent can supply chain information be embedded in Java artefacts?

To evaluate this research question, Classport constructs both a dependency list and dependency tree by statically scanning the JAR file and parsing the annotations within it. This section starts by presenting the results related to the dependency list, and finishes up by presenting the results related to the tree.

**Dependency list.** Table 5.1 shows the comparison between the dependency lists from Maven and Classport for each project. First, it lists the number of dependencies discovered by the Maven dependency plugin, and next is the number of dependencies discovered by Classport. It is worth noting that all dependencies discovered by Classport are true positives (dependencies identified by both Classport and Maven), and as such, no false positives (dependencies identified by Classport not reported to exist by Maven) were encountered.

However, it can be seen that for most projects, Classport only managed to identify a subset of the dependencies identified by Maven. To reason about these results, it is important to understand *why* they occurred. All such occurrences, regardless of project, had one thing in common: the missing dependency contributed no class files to the executable JAR. As such, there was no way for Classport to neither embed nor extract any dependency information related to it. During these evaluations, four main reasons for this

Table 5.1: Static dependency list results for the tested projects

Project	Dependencies (Maven)	Dependencies (Classport)
Checkstyle	71	33
Graphhopper	201	125
PDFBox	12	12
ttorrent	7	7
jacop	12	0
jd-cli	8	6
certificate-ripper	28	4
mcs	76	4

have been observed. The first, and by far the most common, is that some dependencies are simply not meant to be included in the executable JAR. For many projects this involves dependencies used only for testing, such as JUnit and Dropwizard’s testing module. Dependencies expected to be provided by the runtime may also not be included, such as Checkstyle’s dependency on Apache Ant (which has the scope ‘provided’). In the case of jacop, none of its 12 dependencies are actually required at runtime and are thus not packaged within the JAR. Most are only used for compiling and building the project, and two (`slf4j-api` and `slf4j-log4j12`) are not used at all.

The second reason is that dependencies may contain only metadata files. The `com.google.guava:listenablefuture` artefact of version `9999.0-empty-to-avoid-conflict-with-guava` is used by Guava to signal that it provides the `ListenableFuture` class, and contains no actual class files itself. For Classport, this means that the `ListenableFuture` class is annotated as part of Guava since that is where it is actually located.

The third reason is that dependencies may contain code, but not in the form of class files. For example, Graphhopper depends on a lot of ‘web JARs’, such as `org.webjars.npm:react`, which contain mostly JavaScript and JSON files.

Finally, cases were also encountered in which one dependency ‘shadowed’ another, usually in conjunction with an artefact having its name changed. For example, Graphhopper includes both `javax.inject:javax.inject` and its newer counterpart, `jakarta.inject:jakarta.inject-api`, as indirect dependencies. But while the name of the dependency itself was changed, the names of its class files and their package were not. Thus, when



the Maven Shade plugin [39] is used to package the project, it only includes one version of each class file [58].

In summary, Classport was able to correctly identify all dependencies that contributed class files to the final JAR. Furthermore, no warnings were emitted by the Static Analyser during this process, indicating that it was able to successfully parse the annotations for all class files contained in the JARs. Since jacop uses Scala for a few of its classes, this also hints at Classport being capable of embedding annotations in, and parsing them from, other JVM-based languages. Since the focus of this thesis is Java, however, further evaluation of other such languages is left to future work.

**Dependency tree.** After the dependency tree had been generated by Classport as described in Section 3.3.3, it was compared to the dependency tree generated by Maven. Three main types of differences were detected between the two, which are each described and discussed below.

The first difference concerns the verbosity of the output, and arises due to the Maven dependency plugin omitting certain transitive dependencies in cases of duplicates or conflicts – even in verbose mode. Classport does not do this, and as such, Classport’s dependency trees will often be larger than Maven’s even when modelling the exact same underlying dependencies.

Second, Classport’s tree will not contain entries for the dependencies it cannot detect, much like the dependency list discussed above. Examples of this include the `listenablefuture` or `web-JAR` dependencies which don’t contain any class files at all, but also dependencies which are not present due to being shadowed by another, such as `jakarta.inject-api` and `javax.inject`.

Finally, there were some entries missing from Maven’s tree which seemingly did not fall into any of the categories above. For example, Maven’s tree for Graphhopper’s `web` module lists 3 dependencies for version `2.6.1` of `hk2-locator`, while both Classport and the `hk2-locator` POM lists 6. The dependencies omitted by Maven have nothing obvious in common and the reason for their omission is thus unclear. However, in all such cases, the information provided by Classport was manually verified to be accurate according to the POM, which is still seen as an acceptable result.

## 5.2 RQ2 – To what extent can embedded supply chain information be retrieved at runtime?

Table 5.2 presents the results of evaluating RQ2. It shows the dependencies detected by the Static Analyser for all projects, as well as the loaded dependencies detected by the Dynamic Analyser at runtime. First, the executable JAR is run together with the Dynamic Analyser using the workload defined in Section 4.2. The number of dependencies detected is reported under ‘Regular’. Next, the JAR is modified to ensure that all dependencies are loaded, and the Dynamic Analyser is run alongside the application once more. Finally, these new numbers are reported under ‘Post-modification’.

The first thing that should be acknowledged is that the dependencies used by a program can vary greatly depending on how it is executed, highlighting the importance of runtime-aware supply chain information. Second, the table shows that for all modified JARs, Classport was able to successfully detect the same number of dependencies at runtime as it could statically. The lists were also compared to verify that they actually contained the same dependencies, and that they were not just the same size. Moreover, the trees generated by Classport from post-modification runtime data were in all cases identical to the ones generated statically. This indicates that Classport is able to successfully extract its embedded metadata for every such class file loaded, and thus accurately represent the runtime supply chain.

In addition to these results, this evaluation also highlighted the possibility of JARs containing un-loadable classes. While no issues arose from running the regular Graphhopper JAR (with Classport annotations included), the first attempt at running the modified one resulted in a crash. This was due to one of the classes in the JAR implementing an interface from another dependency, which was not part of the JAR. When this particular class then happened to be selected for loading, the JVM was unable to resolve the transitive dependency on the interface, causing a `NoClassDefFoundError` to be thrown. Overall, this incident prompted some changes in the modification tool to allow for a deny-list on which classes were selected for loading, but has no affect on the overall validity of the method.

Table 5.2: Dynamic dependency list results for the tested projects

Project	Dependencies in JAR (static inspection)	Dependencies loaded	
		Regular	Post-modification
Checkstyle	33	7	33
Graphhopper	125	97	125
PDFBox	12	8	12
torrent	7	6	7
jacop	0	0	0
jd-cli	6	6	6
certificate-ripper	4	4	4
mcs	4	4	4

### 5.3 RQ3 – How applicable is Classport on real-world programs?

This section presents the findings and results regarding Classport’s potential for real-world use. It begins by discussing how Classport affected the projects’ test cases, and concludes with a summary of Classport’s spatial requirements.

**Testing.** Before being able to obtain accurate test results, two things needed to be considered. First, many projects recommend cleaning up artefacts from previous compilations before running the tests, to avoid errors related to stale files [59, 60, 61, 62]. While this is generally regarded as good practice, it poses some difficulties when testing with Classport since the act of recompiling everything before running the tests would remove the metadata. Thus, the commands used to test the projects are modified to run the embedding process in between cleaning the working directories and running the tests. Second, Checkstyle will lint itself as part of its integration tests. This means that Classport’s embedded dependencies must be moved out of the project directory before running the tests. If this is not done, Checkstyle will attempt to lint the JAR files too, and its tests will fail.

With these factors taken into consideration, comparing the unit test output shows no difference in test results between the Classport version and the original for any program. The numbers can be seen in Table 5.3. For example, a total of 5165 tests were run during the evaluation of Checkstyle\*, of which

---

\*The test suite for this commit consists of 5594 tests in total, but 429 were automatically

Table 5.3: Tests passed and application size before and after embedding the Classport metadata

Project	Tests passed		Size (MB)		
	Before	After	Before	After	Increase
Checkstyle	5165	5165	19.59	23.20	18.45%
Graphhopper	3043	3043	42.03	47.55	13.14%
PDFBox	2029	2029	12.95	14.73	13.75%
ttorrent	4	4	1.14	1.34	17.54%
jacop	210	210	2.14	2.81	31.31%
jd-cli	3	3	1.64	1.94	18.29%
certificate-ripper	25	25	0.69	0.79	14.49%
mcs	108	108	1.18	1.30	10.17%

all passed. While these results are by no means exhaustive, they do indicate that the embedding of Classport metadata in a project's class files does not significantly change the program's behaviour.

**Size.** Table 5.3 also shows the difference in size once the Classport data has been embedded. The size is listed in megabytes and was obtained using the Unix `stat` command. While additions of this size (just above 5 MB for an application with 201 dependencies) are unlikely to cause any significant problems on modern machines, it is significantly higher than the target 2% mentioned in Section 4.3. Aiming to reduce the size increase while keeping the class-level granularity is thus a good objective for future studies in this area.

## 5.4 Reliability Analysis

In general, the addition of the Classport annotation should not cause any unexpected behaviour, as indicated by the results in Table 5.3. This being said, there are still a few edge cases that should be acknowledged. First, running the program with the Classport annotation present in the classpath will result in an extra element returned by methods such as `Class.getAnnotations()` and `Class.getDeclaredAnnotations()`. However, when running the program *without* the annotation in the class path, the aforementioned

---

skipped.

methods will silently ignore the annotation since it isn't recognised [63]. Thus, the returned array will be the same as if the methods were called on the non-modified class. Second, adding the Classport annotation could cause the constant pool to overflow since each string stored in the annotation adds one entry to it. This being said, the maximum number of entries in the constant pool is 65535 [64] and so this is very unlikely\*. Third, if another annotation with the same name is already present, the Embedder will not overwrite it and Classport may thus fail during parsing. While an unrelated annotation being named `ClassportInfo` is not expected, the possibility should still be acknowledged. Not overwriting previous annotations also means that should the content of Classport annotations be updated, the class files would need to be recompiled and the Embedder run anew to ensure that the latest version is used.

---

\*For comparison, in terms of size, the largest class in the Graphhopper JAR packaged during these experiments (without Classport annotations) is one of Kotlin's internal array classes, `kotlin.collections.ArraysKt___ArraysKt`. This class contains a total of 1650 methods and 3598 constant pool entries, and while the number of methods is not the only factor in constant pool size, it illustrates the point well.



# Chapter 6

## Discussion

This chapter contains observations and reflections made over the course of writing this thesis. This includes current limitations, reflections on validity, as well as ideas for future work.

### 6.1 Limitations

There are three main limitations to Classport. First, as seen in Section 5.1 and Section 5.2, it is unable to represent dependencies that do not contain any class files. Such artefacts may instead consist of audio files, images or other data necessary for the application to function properly, and so Classport cannot be entirely relied on if the goal is to filter out unused artefacts. To resolve this, the Embedder would need a way of storing supply chain information inside the JAR for dependencies without class files. Moreover, the analysers would need to be extended to consider this information.

Second, Classport will in many cases not be able to incorporate class files loaded from sources other than the application JAR into its dependency representations. This is an inherent limitation of the Embedder, since it cannot process dependencies it does not know about. As of the current version of Classport, if an application relies on loading classes dynamically over network connections or similar at runtime, these classes would need to be annotated separately to show up in the dependency representations.

Finally, Classport can easily be ‘tricked’ by simply changing or replacing the annotation in one or more class files, since there is no trust mechanism built in. This could be partially mitigated by signing the Classport JARs after packaging them, although doing so would still not prevent modifications whereupon the signature is also removed from the JAR.

## 6.2 Sustainability and Ethics

While all research that goes towards securing the supply chain (or any other piece of the software puzzle) helps build more sustainable digital infrastructure, this thesis is not directly related to sustainability. Classport in its current state is not meant to be adopted outside of research, and thus any increases in size or computing power resulting from its use are not deemed high enough to be significant.

Moreover, the topic of this thesis does not naturally pose any ethical dilemmas. Of course, the act of embedding information into executable artefacts, and later reading it by monitoring the runtime state of said artefacts, could be used with malicious intent. However, when using the reference implementation provided in this thesis, none of this information ever leaves the host's file system or is used for anything but generating dependency representations. If in the future, Classport were to be extended to collect statistics or do anything that in a similar sense deviates from its core 'embed, read, output to user' functionality, this would have to be explicitly advertised.

## 6.3 Validity Analysis

The largest threat to the validity of this research is the number of projects used in the evaluation. While Classport has been studied on both long running applications (e.g. Graphhopper) as well as short lived tools (e.g. mcs) to ensure variety, it would ideally have been tested on many more projects as well. This could help determine whether there are any other edge cases that have not been handled, both during embedding and extraction of the metadata. Furthermore, being able to use Classport-modified JARs in place of their non-Classport variants for an extended period of time could help determine whether there are any unforeseen consequences of the embedding. There could be crashes due to assumptions made by the application that no longer hold when Classport is used, dramatic slowdowns for certain use cases, or other issues that affect the general usability of the application.

## 6.4 Future work

This section describes the areas in which this research could be improved upon by future studies. It focuses on three main areas: addressing the limitations of the tool, possible extensions, and improving its overall applicability.



### 6.4.1 Addressing limitations

As mentioned in Section 6.1, Classport is currently unable to identify dependencies that do not contribute any class files to the final executable JAR. Many of these dependencies, such as `listenablefuture` and the web JARs mentioned in Section 5.1, do however contain a POM which is packaged into the JAR. Classport could thus be extended to identify these POMs and use the information contained in them to augment the dependency representations. Ideally, artefacts that provide data but do not contain a POM would also be handled. However, this information would have to be recorded separately within the JAR rather than being embedded, since altering the data itself may render it invalid. Note that despite these additions allowing the Static Analyser to produce more accurate representations once extended, this information would not reach the current Dynamic Analyser since it is not tied to a class file. Thus, using this information at runtime would require not just extending, but reworking the Dynamic Analyser significantly.

The second limitation concerns Classport not being able to resolve the origins of class files unknown to the Embedder at compile time, for example those loaded dynamically through a network connection. Identifying such classes even when they do not contain annotations themselves would require extending Classport with another method of provenance identification. This could be a method similar to the one proposed by Davies *et al.* [44], discussed in Section 2.5.2, or another method entirely.

Finally, Classport could also be extended to support distinguishing a genuine annotation from a spoofed one. While spoofed annotations may not be likely for the current scope of Classport, they are nonetheless possible and could cause subtle misrepresentations of the supply chain. Furthermore, being able to verify annotations would allow further extensions to Classport, such as taking action when encountering class files of unknown or unwanted provenance.

### 6.4.2 SBOMs and Vulnerability querying

Since the Embedder embeds information about the group, artefact, version and inter-dependency relationships for every class file included in the executable JAR, it contains enough metadata to produce a proper SBOM [23]. As such, this is a natural extension of Classport.

The embedded information could also be used to query dependencies against a vulnerability database. Such an ability could be useful in both static and dynamic scenarios. For example, when generating a dependency list or

tree from a JAR on the file system, the Static Analyser could highlight any vulnerable dependencies within the tree and/or generate a report containing the vulnerable dependencies and from where they originate. While the OWASP Maven plugin offers similar functionalities [65], Classport could be used even without Maven (provided that the JAR contains the necessary annotations). Moreover, the Dynamic Analyser could use this information to log any classes loaded that are, or come from a dependency which is marked as vulnerable. It could also be configured to halt the JVM if the loading of any such classes is attempted, or perform other actions of the users' choice.

### 6.4.3 Improving applicability

While using Classport should require no more effort than running the Embedder and modifying the project's build command, there may still be cases in which it is not desirable due to spatial constraints. Thus, the size increases highlighted in Section 5.3 could be reduced to make Classport's impact less noticeable to the end user. Currently, the amount of space required by Classport is dependent on two main things: the overall number of class files (since one annotation is embedded in each) and the amount of information stored in each annotation. For example, an artefact declaring a lot of dependencies but consisting of mostly small class files would result in a larger size increase, due to a lot of information being duplicated amongst files that are otherwise very small. One idea that was briefly considered during the design process was storing all artefact metadata in a database placed somewhere inside the JAR. The database would be indexed by a short key, unique for each artefact, and the annotation type would be changed to store only this key. This would store the metadata for each artefact only once and instead only duplicate small keys, dramatically reducing the size increase for most applications at the cost of slightly slower extraction (due to the required lookups). Another, similar approach could be choosing one class file within a dependency to hold the metadata, and have all other class files from the same artefact 'point' to the selected one. This would avoid having to add a separate database file to the JAR.

Moreover, the results of the evaluation indicate that Classport works with other JVM-based languages as well. While this is a reasonable assumption to make, as Classport works with the class files and not the source language, further evaluations into this could be beneficial to ensure that no unexpected problems occur. Being able to use Classport with languages such as Scala, Kotlin and Clojure as well as Java, would increase its applicability even more.

# Chapter 7

## Conclusions

Knowing what code is being run and where it came from can be a valuable asset when attempting to understand an application. While traditional SBOMs can provide large amounts of information about the application's supply chain, they are usually generated at build time and do not necessarily reflect which dependencies are actually used. This thesis investigates the possibility of bridging this gap, making it possible to generate runtime-aware dependency representations.

By embedding annotations into Java class files, we are able to accurately identify metadata about all dependencies being loaded by the JVM at runtime. Moreover, these annotations can be used as an 'embedded SBOM' to accurately enumerate all dependencies that contribute code to the final application as well as their relationships. The act of embedding the annotations does not appear to affect the correctness of the application in any way, but does result in a significant size increase compared to related work, marking this a good target for future optimisations.

Overall, the work done in this thesis should serve as a good basis for future research focused on runtime and/or embedded supply chain representations. There is a lot more to explore in this area, and hopefully the research presented here can inspire new methods and tooling that help secure the software of tomorrow.

---



# References

- [1] Federal Register, “Improving the nation’s cybersecurity,” <https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity>, accessed: 2024-03-22. [Page 1.]
- [2] Council of European Union. The Council agrees to strengthen the security of ICT supply chains. The European Union. [Online]. Available: <https://www.consilium.europa.eu/en/press/press-releases/2022/10/17/the-council-agrees-to-strengthen-the-security-of-ict-supply-chains/> [Page 1.]
- [3] D. Temple-Raston. A ‘worst nightmare’ cyberattack: The untold story of the solarwinds hack. National Public Radio. [Online]. Available: <https://www.npr.org/2021/04/16/985439655/a-worst-nightmare-cyber-attack-the-untold-story-of-the-solarwinds-hack> [Page 1.]
- [4] CVE. CVE-2021-44228. The MITRE Corporation. [Online]. Available: <https://www.cve.org/CVERecord?id=CVE-2021-44228> [Page 1.]
- [5] IBM. What is Log4Shell? [Online]. Available: <https://www.ibm.com/topics/log4shell> [Page 1.]
- [6] E. U. A. for Cybersecurity. (2021, July) Enisa Threat Landscape for Supply Chain Attacks. European Union Agency for Cybersecurity. [Online]. Available: <https://www.enisa.europa.eu/publications/threat-landscape-for-supply-chain-attacks> [Page 1.]
- [7] NTIA, “SBOM FAQ,” National Telecommunications and Information Administration, Tech. Rep. [Online]. Available: [https://www.ntia.gov/sites/default/files/publications/sbom\\_faq\\_-\\_20201116\\_0.pdf](https://www.ntia.gov/sites/default/files/publications/sbom_faq_-_20201116_0.pdf) [Pages 1 and 8.]
- [8] M. Balliu, B. Baudry, S. Bobadilla, M. Ekstedt, M. Monperrus, J. Ron, A. Sharma, G. Skoglund, C. Soto-Valero, and M. Wittlinger,

- “Challenges of Producing Software Bill of Materials for Java,” *IEEE Security and Privacy*, vol. 21, no. 6, p. 12–23, Nov. 2023. doi: 10.1109/msec.2023.3302956. [Online]. Available: <http://dx.doi.org/10.1109/MSEC.2023.3302956> [Pages 1, 2, 7, and 8.]
- [9] CycloneDX Team. Tool center. [Online]. Available: <https://cyclonedx.org/tool-center/> [Page 2.]
- [10] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java® Language Specification*, Oracle. [Online]. Available: <https://docs.oracle.com/javase/specs/jls/se21/html/jls-12.html#jls-12.2.1> [Page 2.]
- [11] jbm contributors. jbm. [Online]. Available: <https://github.com/eclipse/jbm> [Pages 2 and 8.]
- [12] ——. jbm – Libraries.java. [Online]. Available: <https://github.com/eclipse/jbm/blob/99dff173ddd2aa50c0023f4e099dba88cb2244e2/src/main/java/com/contrastsecurity/Libraries.java#L237> [Pages 2 and 10.]
- [13] Synopsys Software Integrity Group, “Open Source Security and Risk Analysis Report,” Tech. Rep., 2024. [Online]. Available: <http://web.archive.org/web/20240423032730/https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2024.pdf> [Pages 7 and 8.]
- [14] L. Zhang, C. Liu, S. Chen, Z. Xu, L. Fan, L. Zhao, Y. Zhang, and Y. Liu, “Mitigating persistence of open-source vulnerabilities in maven ecosystem,” *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 191–203, 2023. [Online]. Available: <https://api.semanticscholar.org/CorpusID:260682834> [Page 7.]
- [15] Sonatype, “State of the Software Supply Chain,” Tech. Rep., 2023. [Online]. Available: <https://www.sonatype.com/hubfs/9th-Annual-SSS-C-Report.pdf> [Page 8.]
- [16] M. Ohm, H. Plate, A. Sykosch, and M. Meier, “Backstabber’s knife collection: A review of open source software supply chain attacks,” 2020. [Page 8.]
- [17] Check Point Research Team. Protect Yourself Against The Apache Log4j Vulnerability. [Online]. Available: <https://blog.checkpoint.com/security>

ty/protecting-against-cve-2021-44228-apache-log4j2-versions-2-14-1/  
[Page 8.]

- [18] S. James, “FAQ on the xz-utils backdoor (CVE-2024-3094),” <https://gist.github.com/thesamesam/223949d5a074ebc3dce9ee78baad9e27>. [Page 8.]
- [19] Framing Working Group, “Framing Software Component Transparency: Establishing a Common Software Bill of Material (SBOM),” National Telecommunications and Information Administration, Tech. Rep. [Online]. Available: [https://www.ntia.gov/files/ntia/publications/framingsbom\\_20191112.pdf](https://www.ntia.gov/files/ntia/publications/framingsbom_20191112.pdf) [Page 8.]
- [20] The Linux Foundation. System Package Data Exchange (SPDX®). [Online]. Available: <https://spdx.dev/> [Page 8.]
- [21] OWASP Foundation. Specification overview. [Online]. Available: <https://cyclonedx.org/specification/overview/> [Page 8.]
- [22] U.S. National Institute of Standards and Technology. Software Identification (SWID) Tagging. [Online]. Available: <https://csrc.nist.gov/projects/Software-Identification-SWID> [Page 8.]
- [23] Department of Commerce, “The Minimum Elements for an SBOM,” National Telecommunications and Information Administration, Tech. Rep. [Online]. Available: [https://www.ntia.gov/sites/default/files/publications/sbom\\_minimum\\_elements\\_report\\_0.pdf](https://www.ntia.gov/sites/default/files/publications/sbom_minimum_elements_report_0.pdf) [Pages 8 and 45.]
- [24] package-url and PURL contributors. purl-spec. [Online]. Available: <https://github.com/package-url/purl-spec> [Page 8.]
- [25] B. Seshadri, Y. Han, C. Olson, D. Pollak, and V. Tomaević, “Omnibor: A system for automatic, verifiable artifact resolution across software supply chains,” *ArXiv*, vol. abs/2402.08980, 2024. [Online]. Available: <https://api.semanticscholar.org/CorpusID:267657786> [Pages 8 and 15.]
- [26] CycloneDX Team. BOM Examples, cern-lhc-vdm-editor-e564943. [Online]. Available: <https://github.com/CycloneDX/bom-examples/blob/0979663521c4623792dc432d09f88bcb85862a62/SBOM/cern-lhc-vdm-editor-e564943/bom.json> [Pages ix and 9.]
- [27] M. Mirakhorli, D. Garcia, S. Dillon, K. Laporte, M. Morrison, H. Lu, V. Koscinski, and C. Enoch, “A Landscape Study of

- Open Source and Proprietary Tools for Software Bill of Materials (SBOM),” *ArXiv*, vol. abs/2402.11151, 2024. [Online]. Available: <https://api.semanticscholar.org/CorpusID:267750339> [Page 8.]
- [28] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, *The Java® Virtual Machine Specification*, Oracle. [Online]. Available: <https://docs.oracle.com/javase/specs/jvms/se19/html/jvms-4.html> [Page 10.]
- [29] Oracle. (2004) New Features and Enhancements, J2SE 5.0. [Online]. Available: <https://docs.oracle.com/javase/1.5.0/docs/relnotes/features.html> [Page 11.]
- [30] Z. Yu, C. Bai, L. Seinturier, and M. Martin, “Characterizing the usage, evolution and impact of java annotations in practice,” *IEEE Transactions on Software Engineering*, vol. 47, pp. 969–986, 2018. [Online]. Available: <https://api.semanticscholar.org/CorpusID:102351817> [Page 11.]
- [31] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java® Language Specification*, Oracle. [Online]. Available: <https://docs.oracle.com/javase/specs/jls/se21/html/jls-9.html#jls-9.6.1> [Page 11.]
- [32] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, *The Java® Virtual Machine Specification*, Oracle. [Online]. Available: <https://docs.oracle.com/javase/specs/jvms/se19/html/jvms-5.html#jvms-5.3> [Page 12.]
- [33] *Class ClassLoader*, Oracle, 2023. [Online]. Available: <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/ClassLoader.html#builtinLoaders> [Page 12.]
- [34] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, *The Java® Virtual Machine Specification*, Oracle. [Online]. Available: <https://docs.oracle.com/javase/specs/jvms/se19/html/jvms-5.html#jvms-5.4> [Page 12.]
- [35] —, *The Java® Virtual Machine Specification*, Oracle. [Online]. Available: <https://docs.oracle.com/javase/specs/jvms/se19/html/jvms-5.html#jvms-5.5> [Page 12.]



- [36] Oracle. Lesson: Packaging Programs in JAR Files. [Online]. Available: <https://docs.oracle.com/javase/tutorial/deployment/jar/index.html> [Page 12.]
- [37] ——. Working with Manifest Files: The Basics. [Online]. Available: <https://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html> [Page 12.]
- [38] Maven Lockfile contributors. Maven Lockfile. [Online]. Available: <https://github.com/chains-project/maven-lockfile> [Page 13.]
- [39] Apache. Apache Maven Shade Plugin. [Online]. Available: <https://maven.apache.org/plugins/maven-shade-plugin/> [Pages 13 and 37.]
- [40] ——. Introduction to the build lifecycle. [Online]. Available: <https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html> [Page 14.]
- [41] ——. Plugin bindings for default lifecycle reference. [Online]. Available: <https://maven.apache.org/ref/3.9.6/maven-core/default-bindings.html> [Page 14.]
- [42] N. Boucher and R. Anderson, “Automatic bill of materials,” 2023. [Pages 15 and 33.]
- [43] M. Mitzenmacher, “Compressed bloom filters,” Aug. 2001. [Online]. Available: <http://dx.doi.org/10.1145/383962.384004> [Page 15.]
- [44] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle, “Software bertillonage: finding the provenance of an entity,” in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR ’11. New York, NY, USA: Association for Computing Machinery, 2011. doi: 10.1145/1985441.1985468. ISBN 9781450305747 p. 183–192. [Online]. Available: <https://doi.org/10.1145/1985441.1985468> [Pages 16 and 45.]
- [45] H. Jang, N. Murodova, and H. Koo, “ToolPhet: Inference of Compiler Provenance From Stripped Binaries With Emerging Compilation Toolchains,” *IEEE Access*, vol. 12, pp. 12 667–12 682, 2024. doi: 10.1109/ACCESS.2024.3355098 Conference Name: IEEE Access. [Online]. Available: <https://ieeexplore.ieee.org/document/10401926> [Page 16.]

- [46] horsicq and Detect It Easy contributors. Detect It Easy. [Online]. Available: <https://github.com/horsicq/Detect-It-Easy> [Page 16.]
- [47] M. Grenfeldt, “Decompiling go : Using metadata to improve decompilation readability,” Master’s thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2023. [Page 17.]
- [48] Oracle. Understanding Signing and Verification. [Online]. Available: <https://docs.oracle.com/javase/tutorial/deployment/jar/intro.html> [Page 22.]
- [49] ASM contributors. ASM. [Online]. Available: <https://asm.ow2.io/> [Page 22.]
- [50] Oracle. (2024) Package java.lang.classfile. [Online]. Available: <https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/classfile/package-summary.html> [Page 23.]
- [51] *ASM 4.0, A Java bytecode engineering library*, ASM, 2011. [Online]. Available: <https://asm.ow2.io/asm4-guide.pdf> [Page 23.]
- [52] Oracle. addTransformer. [Online]. Available: [https://docs.oracle.com/en/java/javase/21/docs/api/java.instrument/java/lang/instrument/Instrumentation.html#addTransformer\(java.lang.instrument.ClassFileTransformer,boolean\)](https://docs.oracle.com/en/java/javase/21/docs/api/java.instrument/java/lang/instrument/Instrumentation.html#addTransformer(java.lang.instrument.ClassFileTransformer,boolean)) [Page 24.]
- [53] Apache. Introduction to the dependency mechanism. [Online]. Available: <https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html> [Page 27.]
- [54] Checkstyle contributors. Checkstyle. [Online]. Available: <https://checkstyle.org> [Page 29.]
- [55] Graphhopper contributors. Graphhopper routing engine. [Online]. Available: <https://github.com/graphhopper/graphhopper> [Page 29.]
- [56] Maxime Petazzoni and ttorrent contributors. ttorrent. [Online]. Available: <https://github.com/mpetazzoni/ttorrent/> [Page 29.]
- [57] Apache. POM Reference: status. [Online]. Available: <https://maven.apache.org/pom.html#status> [Page 30.]




- [58] Maven Shade Pugin contributors. DefaultShader.java. [Online]. Available: <https://github.com/apache/maven-shade-plugin/blob/49f6e13de5748700ef2185795f89f66be61554ef/src/main/java/org/apache/maven/plugins/shade/DefaultShader.java#L581> [Page 37.]
- [59] Checkstyle contributors. Quality matters. [Online]. Available: [https://checkstyle.org/contributing.html#Quality\\_matters](https://checkstyle.org/contributing.html#Quality_matters) [Page 39.]
- [60] Graphhopper contributors. Contributing.md. [Online]. Available: <https://github.com/graphhopper/graphhopper/blob/master/CONTRIBUTING.md> [Page 39.]
- [61] Apache. PDFBox. [Online]. Available: <https://github.com/apache/pdfbox> [Page 39.]
- [62] Hakan Altundağ and Certificate Ripper contributors. Certificate Ripper. [Online]. Available: <https://github.com/Hakky54/certificate-ripper/> [Page 39.]
- [63] Issue JDK-6322301: unknown annotations are not ignored in Class.getAnnotations. [Online]. Available: <https://bugs.openjdk.org/browse/JDK-6322301> [Page 41.]
- [64] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, *The Java® Virtual Machine Specification*, Oracle. [Online]. Available: <https://docs.oracle.com/javase/specs/jvms/se19/html/jvms-4.html#jvms-4.11> [Page 41.]
- [65] OWASP. OWASP Dependency-Check. [Online]. Available: <https://owasp.org/www-project-dependency-check/> [Page 46.]



# Appendix A

## Supporting material

This appendix contains files that could be useful to readers of this thesis:

- The list of BibTeX references used. 
- The patch for Graphhopper mentioned in Chapter 4, used to change the version from 9.0-SNAPSHOT to 9.1. 
- The torrent file used for the ttorrent evaluation, mentioned in Table 4.2. Downloads a small text file. 









# €€€€ For DIVA €€€€

```
{
  "Author1": { "Last name": "Williams",
    "First name": "Daniel",
    "Local User Id": "u19u2xsf",
    "E-mail": "dwilli@kth.se",
    "organisation": { "L1": "School of Electrical Engineering and Computer Science",
    }
  },
  "Cycle": "2",
  "Course code": "DA237X",
  "Credits": "30.0",
  "Degree1": { "Educational program": "",
    "programcode": "TCYSM",
    "Degree": "Masters degree",
    "subjectArea": "Technology"
  },
  "Title": {
    "Main title": "The Embedding and Retrieval of Software Supply Chain Information in Java Applications",
    "Language": "eng",
    "Alternative title": {
      "Main title": "Inbäddning och extraktion av mjukvaruleverantörskejdan i Java-applikationer",
      "Language": "swe"
    }
  },
  "Supervisor1": { "Last name": "Sharma",
    "First name": "Aman",
    "Local User Id": "u1w0zffw",
    "E-mail": "amansha@kth.se",
    "organisation": { "L1": "School of Electrical Engineering and Computer Science",
      "L2": "Computer Science"
    }
  },
  "Examiner1": { "Last name": "Monperrus",
    "First name": "Martin",
    "Local User Id": "u13jhcyf",
    "E-mail": "monperrus@kth.se",
    "organisation": { "L1": "School of Electrical Engineering and Computer Science",
      "L2": "Computer Science"
    }
  },
  "National Subject Categories": "10201, 10206",
  "Other information": { "Year": "2024", "Number of pages": "1,57",
    "Copyrightleft": "copyright",
    "Series": { "Title of series": "TRITA – EECS-EX", "No. in series": "2024:0000" },
    "Opponents": { "Name": "Johan Ekroth",
      "Presentation": { "Date": "2024-06-20 10:00"
        "Language": "eng",
        "Room": "Germund Dahlquist",
        "Address": "Osquars backe 2",
        "City": "Stockholm"
      },
      "Number of lang instances": "2",
      "Abstract[eng ]": "€€€€"
    }
  }
}
```

As attacks on the software supply chain become increasingly prevalent, many steps have been taken to provide further insight into the components that make up the software. A common way to do this is through a software bill of materials -- a document providing information about each component in the supply chain as well as their relationships. However, these are often generated by statically inspecting a project and contain no information about which code is actually executed at runtime. Knowing what components are used and when can be useful in many ways, such as for dependency trimming or dynamic vulnerability querying.

In this thesis, we present and evaluate a collection of tools, Classport, which can be used to produce runtime-aware dependency representations for Java programs built using the Maven build system. This is achieved in two main steps. First, we use a Maven plugin to automatically embed supply chain information inside the class files required by the project, including those contained in its dependencies. Second, we use a Java agent to monitor the loaded classes at runtime and extract the supply chain information from them.

We evaluate Classport by testing it on eight open source projects, comparing its dependency representations to those provided by Maven itself. We also investigate side effects of the embedding, such as the size increase and whether it leads to changes in the program's behaviour. The results show that we are able to achieve our goal of accurately reflecting the runtime state of the program in terms of its loaded dependencies. Additionally, the results show that while our method of adding metadata does not seem to affect the correctness of a program, it does noticeably increase its size. Making the impact of Classport less noticeable to the average user could increase the

possibility of mainstream adoption, and is thus an appropriate candidate for future work.

€€€€,

"Keywords[eng ]": €€€€

Java, Metadata, Software supply chain, Maven €€€€,

"Abstract[swe ]": €€€€

Den så kallade mjukvaruleverantörskedjan, bestående av alla komponenter som används i ett programs utveckling, har blivit ett allt vanligare mål för cyberattacker. I takt med detta har ett flertal lösningar tagits fram för att öka förståelsen för dessa komponenter -- exempelvis den kod som ingår i mjukvaran. En materialförteckning för mjukvara (eng: software bill of materials) används ofta för detta ändamål, och innehåller bland annat information om ett programs komponenter samt deras relation till varandra. De allra flesta sådana förteckningar genereras dock medan koden byggs eller efter att den packats ihop till en fil, och har ingen insikt i vilka delar av koden som faktiskt exekveras. Kunskap om detta kan vara användbart på många sätt, till exempel för att hitta tredjepartskomponenter som inte används eller för att undersöka koden för kända sårbarheter under tiden den körs.

I detta examensarbete presenterar och utvärderar vi en samling verktyg, Classport, som kan användas för att producera representationer av mjukvaruleverantörskedjan som tar hänsyn till vilken kod som faktiskt exekveras. Verktygen är utvecklade för att användas på Java-projekt som använder byggsystemet Maven, och består av två huvudsakliga steg. I det första används Maven för att identifiera alla klassfiler som används i projektet, och information om deras ursprung bäddas in i dem. I det andra steget används en Java-agent för att fånga upp alla klasser som laddas in för exekvering, varpå den tidigare inbäddade information extraheras igen.

Vi utvärderar Classport genom att testa verktygen på åtta projekt med öppen källkod, och jämför informationen det ger oss med den information som tillhandahålls av Maven självt. Vi undersöker även hur vår inbäddade data påverkar applikationen. Resultaten visar på att vi med hög precision kan återge både vilken kod som laddats in under körtiden och dess ursprung. Den inbäddade informationen verkar inte heller ha någon påverkan på programmets funktion. Däremot leder den till att programfilerna blir märkbart större, vilket riskerar att minska verktygens användningsområde. Hur dessa storleksökningar kan undvikas utan att minska mängden extraherbar information vore således ett lämpligt område för framtida forskning.

€€€€,

"Keywords[swe ]": €€€€

Java, Metadata, Mjukvaruleverantörskedjor, Maven €€€€,

}

# acronyms.tex

```
%%% Local Variables:
%%% mode: latex
%%% TeX-master: t
%%% End:
% The following command is used with glossaries-extra
\setabbreviationstyle[acronym]{long-short}

\newacronym{KTH}{KTH}{KTH Royal Institute of Technology}
\newacronym{SBOM}{SBOM}{Software Bill of Materials}
\newacronym{JVM}{JVM}{Java Virtual Machine}
\newacronym{JAR}{JAR}{Java Archive}
\newacronym{ABOM}{ABOM}{Automatic Bill of Materials}
\newacronym{OmniBOR}{OmniBOR}{Universal Bill of Receipts}
\newacronym{JIT}{JIT}{just-in-time}
\newacronym{AOT}{AOT}{ahead-of-time}
\newacronym{JDK}{JDK}{Java Development Kit}
\newacronym{JLS}{JLS}{Java Language Specification}
\newacronym{JVMS}{JVMS}{Java Virtual Machine Specification}
\newacronym{NTIA}{NTIA}{National Telecommunications and Information Administration}
\newacronym{POM}{POM}{Project Object Model}
\newacronym{CLI}{CLI}{command-line interface}
\newacronym{API}{API}{Application Programming Interface}
\newacronym{URL}{URL}{Uniform Resource Locator}
\newacronym{SSH}{SSH}{Secure Shell}
```