D2.6 FASTEN Maven plug-in

Disclaimer
The FASTEN project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 825328

Version history

Ver.	Date	Comments/Changes	Author
0.1	2021-06-18	Initial draft of the report	T. Mortagne (XWiki SAS)
0.2	2021-06-30	Improvements based on SIG's review	T. Mortagne (XWiki SAS)
1.0	2021-07-02	Improvements based on Endocode's review	T. Mortagne (XWiki SAS)

Project Acronym	FASTEN	
Project Title	Fine-Grained Analysis of Software Ecosystems as Networks	
Grant Agreement N^{o}	825328	
Instrument	Innovation Action (IA)	
Thematic Priority	ICT-16-2018 Software Technologies	
Project Start Date	2019-01-01	
Project Duration	36 months	
Work Package	**WP2 Knowledge Base**	
Task	**T5 Integrating FASTEN with Maven**	
Deliverable	**D2.6**	
Due Date	June 30, 2020	
Submission Date	July 2, 2021	
Dissemination Level [1]	PU	
Deliverable Responsible	XWiki SAS	
Author(s)	Thomas Mortagne	XWiki SAS
Reviewer(s)	Vincent Massol	XWiki SAS
	Chushu Gao	SIG
	Michele Scarlato	Endocode AG

[1] PU=Public, CO=Confidential, only for members of the consortium (including the Commission Services), CL=Classified, as referred to in Commission Decision 2001/844/EC

Table of Contents

Abstract

This report gives a status of the FASTEN Maven plugin at month 30 of the FASTEN project. It starts with a description of the generic use cases that are planned to be covered by the end of the project at M36. It also describes the requirements and the architecture which illustrate the choices made in terms of load, distributed between the FASTEN server and the client. Finally, it summarizes the current state of the plugin and its KPIs.

1 Project information

The project is open-source and publicly managed in the repository of the FASTEN project.[2] That's where the code can be found and where the plugin will be released. It also provides the infrastructure to report bugs and improvements requests[3] and the documentation is planned to be public on the project's wiki on GitHub.

2 Use cases

In the context of the FASTEN project, the plugin is designed with the following generic use cases in mind:

2.1 Report risks associated to a Maven module at build time locally

While developing a project and before publishing the modifications it's very useful to be able to validate some dependency you would like to add or upgrade on various criteria (security, quality, license, etc.). Everyone applies some criteria when choosing a dependency by hand, but some might be harder to find and even harder when it comes to the transitive dependencies of this dependency.

This plugin goal being to validate the entire dependency tree for a configured list of objective criteria, it becomes much faster and less error-prone to validate the dependencies as part of the standard build of the module or with a specific Maven command line.

[2] https://github.com/fasten-project/fasten-maven-plugin
[3] https://github.com/fasten-project/fasten-maven-plugin/issues

Like any Maven plugin, it is triggered as part of the `<build>` section of the pom.xml file:

```
<project>
  ...

  <build>
    <plugins>
      <plugin>
        <groupId>eu.fasten</groupId>
        <artifactId>fasten-maven-plugin</artifactId>
        <version>1.0</version>
        <configuration>
          ...
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>check</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

which will execute it as part of the standard build with `mvn install` for example.

But it can also be isolated in a Maven profile:

```
<project>
  ...

  <profiles>
    <profile>
      <id>fasten</id>
      <build>
        <plugins>
          <plugin>
            <groupId>eu.fasten</groupId>
            <artifactId>fasten-maven-plugin</artifactId>
            <version>1.0</version>
            <configuration>
              ...
            </configuration>
            <executions>
              <execution>
                <goals>
                  <goal>check</goal>
                </goals>
              </execution>
            </executions>
          </plugin>
        </plugins>
      </build>
    </profile>
  </profiles>
</project>
```

in which case, running it require to either enable the profile with `-Pfasten` as in `mvn install -Pfasten` or alone using `mvn fasten:check`.

A more detailed example project is available at Appendix B 7.

2.2 Continuous Integration

It's always a good thing to validate as much as possible what you work on before committing, but it's not always easy to notice side effects on other modules you did not directly modify. To catch this a continuous integration platform (Jenkins, GitHub actions, GitLab CI, etc.) is often used. Every time a modification is pushed to the reference repository, it automatically triggers a build of the project and its depends.

In this context, the plugin will generally be configured to fail the build if any blocker risk is identified. The result will be reported in a dashboard and through notifications to know as soon as possible that there is a problem with the project.

2.3 Contribution pre validation

In systems like GitHub or GitLab it's possible to contribute to a project by create a pull/merge request. This means that you create a copy (a "fork") of the project on your side, modify it, and then propose to the original project's author to integrate your modifications. From there, the author can review the modifications and request changes before integrating ("merging") it.

To help the review process, various tools are generally available. A very common validation help is to automatically build the branch of the project which contains the proposed modification and report the result in the request UI.

3 Architecture

The architecture of the plugin is illustrated in figure 1.

The clear blue elements are other FASTEN components used by the Maven plugin, for which the components are in a darker blue.

1. Maven provides as input to the plugin the project (`P`) and the artifacts representing each of the project direct and transitive dependencies (`A`, `B`, `C`, `D`).

2. For each of those artifacts (green boxes in the figure) the plugin obtains a call graphs either remotely using the FASTEN Central REST API or locally using the `javacg-opal` library. We don't try to search the project on the server because it's not released, so it's necessarily unknown. The server returns an HTTP 202 code for `A`, this means that the package does not exist yet on the FASTEN server but thanks to the coordinates and Maven repository sent with the request it started an asynchronous job in charge of downloading and analyzing this artifact for later. The plugin fallback on locally the local analysis of `A` for now.

3. It then provides all those call graphs locally to the `fasten-core` stitching tool library to produce a stitched call graph in which all the project and dependencies graphs are merged into a single one where all the inaccessible branches (calls which are never hit when fallowing the calls from the main project) are stripped.

4. Finally, it will produce enrich the stitched call graph with metadata provided by the REST API (from a list of calls). One of the metadata illustrated in the figure is a known vulnerability and its associated CVE identifier (`CVE X`).

5. It's then possible to navigate this graph locally to conduct various analysis to produce a report about the risks that could be found.

3.1 FASTEN Knowledge Base server

The FASTEN Maven plugin relies extensively on the FASTEN REST API to access metadata that can be found on a FASTEN server. By default, the plugin is trying to

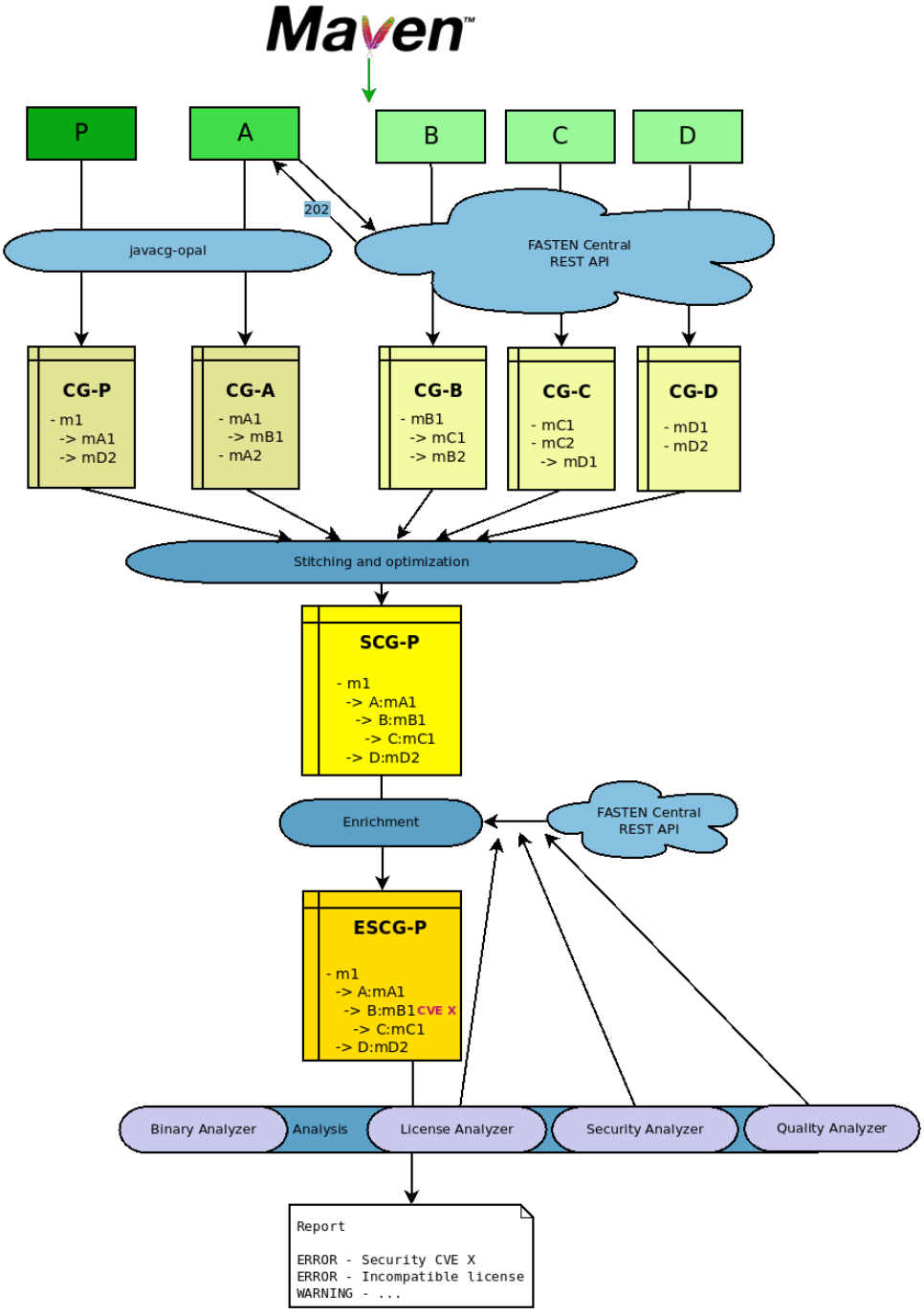

Figure 1: FASTEN Maven plugin architecture

reach the FASTEN Central server which is exposed on https://api.fasten-project.eu/api, but it's also possible to host ones own FASTEN server in which case the plugin can be configured to use this custom server using the Maven property `fastenApiUrl` as in:

```
<configuration>
  <fastenApiUrl>http://localhost/api</fastenApiUrl>
</configuration>
```

3.2 Dependency resolution

Thanks to Maven, the plugin does not need to do anything to resolve the dependency tree of the module, this is directly provided to it as input.

However, what is provided is only the list of Maven identifiers, version and file of each dependency and transitive dependency. This is enough for most tasks, but sometimes the plugin also needs to access some of the information provided as part of each module descriptor (the pom.xml) file, like the licenses. For this, it is using the Project Builder provided by the Maven API to resolve a complete description of each dependency present in the list provided in input.

3.3 Call graphs

At the root of the FASTEN project, and the Maven plugin is no exception, is the building of graphs containing the calls ("methods" in the case of the Java environment) which can be found in a given product.

For Java the tool in charge of doing this is `javacg_opal` and it is used by the FASTEN Maven plugin in two ways:

- indirectly by asking the FASTEN server for a previous built call graph. This is not a technical requirement but is used to speed up analysis by using the FASTEN server as a remote cache of call graph and not redo the same work over and over again
- locally by directly using `javacg_opal` as a Java library to build call graph that could not be found on the server

For debug purposes, the plugin also serialize the obtained call graphs in the `target/call-graphs/` folder locally. Whether or not the call graph should be serialized and in which location can be controlled respectively using the plugin properties `serialize` and `outputDirectory` as in

```
<configuration>
  <serialize>true</serialize>
  <outputDirectory>/some/other/location</outputDirectory>
</configuration>
```

3.4 Stitching and optimization

Once we produced or found a call graph for each module involved in the dependency tree resolved by Maven we need to merge them.

This is done in several steps.

Each call graph contains “internal” and “external” calls. Internal calls are targeting callables located in the graph itself, while external calls target callables which are coming from third party dependencies. For each call graph the plugin will start by trying to find external calls in all the others modules graphs, if a callable is found it become a resolved call while others remain external calls.

Once each call graph is resolved, a big one is created to merged all the individuals call graphs and representing the whole dependency tree as if everything was in the same package.

The plugin can then produce an “optimized” version of the graph by navigating the big one produced in the previous step, starting for the main module calls to retain only the calls which are directly or indirectly used by the built project.

3.5 Enrichment

We now have an optimized call graph but it only contains callables which is useful but is very limited in term of analysis. The main reason why we need to the FASTEN server if to benefit from analysis running on it and leading to the producing of various metadata (security, quality, etc.) associated to the elements of the call graph (callables, packages, etc.) which would be too expensive to run locally as part of the Maven build.

The plugin limit the metadata requested from the server in two ways:

- it only requests it for callables located in the optimized call graph produced in the previous step
- it asked analyzer executed in the following step what kind of metadata they will need to find in the graph

To even more limit the load on the server at a given time, the plugin executes several requests instead of a single one, each one requesting a configurable batch of metadata. The number callables for which to retrieve the metadata in a single request is configurable using the plugin property `metadataBatch` as in

```
<configuration>
  <serialize>true</serialize>
  <metadataBatch>50</metadataBatch>
</configuration>
```

3.6 Analysis

At the end of the processing chain is the analysis of the gathered metadata. This is conducted by an extensible system of analyzer components which are provided by the plugin itself for some but can also be provided by anyone when declaring the dependencies of the plugin in the pom.xml file of the built module.

```
<project>
  ...
      <plugin>
        <groupId>eu.fasten</groupId>
        <artifactId>fasten-plugin</artifactId>
        <version>1.0</version>
        <dependencies>
          <dependency>
            <groupid>org.myproject</groupid>
            <artifactid>my-custom-analyzers</artifactid>
          </dependency>
        </dependencies>
        ...
      </plugin>
  ...
</project>
```

3.6.1 Binary analyzer

This is the only standard analyzer which does not depend on the FASTEN server, since it can find all the information it needs locally.

The core idea behind this analyzer is to find and report “broken calls” in the call graph, this means expected callables but which cannot be found in the module itself or any of the resolved dependencies. While this is something you can see easily by building the module for methods directly used, it is a lot harder in the case of transitives calls. Even in a case where your module has a 100% code coverage using unit tests, developers tend to mock the implementation behind method coming from dependencies for performances reason and to avoid side effect when focusing on the tested code. This means that you can only really notice this kind of problem in integration tests, which can be slow and only catch the use cases you tests, or after the release at runtime.

This situation often comes for having various dependencies which themselves depends on the same product but with conflicting versions. A simple example could be:

- project `P` depends on `A` and `B`
- `A` depends on `C` in version `1.1`
- `B` depends on `C` in version `1.0`

In this case, at the end of the resolution process, the selected version of `C` will be `1.1`. While this is very often not a problem because library generally try to maintain retro compatibility, you still have cases where `1.1` version introduce a breaking change: the method `C.m1()` was removed in version `1.1` compare to version `1.0`.

In this example, the plugin will report that the expected method `C.m1()` cannot be found anywhere in the call graph, similarly to the error you would get at runtime.

3.6.2 Security analyzer

Many tools exist to report security vulnerability in the dependencies but most of them have two drawbacks:

- they only work at package level so they cannot know if you are actually using a method/class affected by a security vulnerability. This can produce a lot of false positives.
- they only look at direct dependencies generally to reduce the number of false positives so they can actually miss vulnerabilities located in transitive dependencies

Thanks to the optimized stitched call graph built during the first phase of the plugin, this analyzer can gather security related metadata for all the calls which are actually used directly or indirectly by the built module and report related risk.

More details about the security metadata that are analyzed as part of the FASTEN project can be found in Deliverable 3.4[1].

3.6.3 Quality analyzer

While it is very common to use tools like Checkstyle to report code quality risks on the build module, it is very rare to be able to exploit this type of information for the dependencies of the module.

This analyzer takes as configuration input a set of thresholds for quality metrics that dependencies have to respect in order to be assumed safe enough to be used. The developer of the module have the possibility to configure precisely for each type of quality metric what is accepted, and the plugin will report any violation in any of the methods found in the call graph.

More details about the quality metric metadata that are analyzed as part of the FASTEN project can be found in Deliverable D3.3[2].

3.6.4 License analyzer

This is the only standard analyzer which does not yet have a working version.

While analyzing the module and its dependencies, the plugin can gather licenses information from two sources:

- the FASTEN REST API will give license information at package and file level
- if the dependency is unknown on the FASTEN server, then the license used will be the one indicated in the module or dependency pom.xml

Once the output license (the license declared for the built module) and the list of inbound licenses (the licenses found in the dependencies) are found they are given as input to the (currently missing) license validation tool which then report incompatibilities. A first version of this tool is planned to be exposed on M30 as a REST API in the FASTEN server, and later we implemented as a Java library that could be locally used by the Maven plugin without depending on the FASTEN server.

3.6.5 Exclusions

There is various reason why someone would want to explicitly ignore reported risks:

- the plugin is simply wrong and report a risk which is not there either because of a bug or because of the limitations inherent to static analysis
- the risk is there but is judged minor enough compared to the work required to fix it
- the risk is there in the build context but the runtime context in which this will be used in the end is actually different enough to not be affected

To work around this, the plugin offer an exclusion configuration followed by all analyzers:

- it's possible to pass a list of regular expressions which are applied to the signature of each method/class involved in a risk before reporting it
- it's possible to pass a list of regular expressions which are applied on the identifier of each dependency package involved in a risk before reporting it

4 Requirements and dependencies

This section gives an overview of technical constraints and the other parts of FASTEN tooling the Maven plugin is depending on and why.

4.1 Call graph

Java call graphs are produced by the FASTEN tool `javacg-opal`. It's in charge of analyzing the Java bytecode to find classes and methods.

The current LTS branch of Java is 11.x (and the next one, Java 17, is planned for September 2021) making it the recommended target for a new Java project. However,

Java 8 is still by far the most used minimum version on projects available on Maven Central, especially among libraries (many of which support even older version of Java).

This makes Java 8 and 11 two important bytecode versions to support. Unfortunately, the support of Java 11 is still very experimental in the version of OPAL currently being used. A lot of work to improve this support have been done for the very last version of OPAL (4) and the FASTEN project is planning to upgrade this dependency.

4.2 Resolution

The plugin uses a tool provided by the `fasten-core` module to produce a resolved version of each call graph as described in the architecture.

While the `fasten-core` module do provide merging tools they don't yet fully cover the need of the Maven plugin especially regarding keeping external calls and the reduction of unique calls from the main module, so this part is currently implemented on the plugin side. This might change in the future.

4.3 Enriched call graph and analysis

While the FASTEN Central server produces and stores call graphs too, its main goal is to associate each call with various metadata (identifier security vulnerabilities, license, quality metrics, etc.). The combination of a call graph and those metadata is called an "enriched" call graph in the context of the FASTEN project.

In order for the Maven plugin to benefit from those metadata it needs the FASTEN REST API to expose an entry point which allow getting all the metadata associated with the manipulated calls.

This API is described in the Deliverable D2.4[3] and made available publicly as the FASTEN Central server on https://api.fasten-project.eu/api/mvn/.

5 Current status and KPIs

The plugin currently implemented everything described in the architecture above except for License validation reports.

Indicator	***How to measure it***	***Target Values***	***Baseline***	***Current***
Support Java 8 projects	Build Maven projects with Java 8 compatibility	> 99% successful builds with accurate serialized call graphs	0%	90%
Support Java 11 projects	Build Maven projects with Java 11 compatibility	> 99% successful builds with accurate serialized call graphs	0%	30%
Report security vulnerabilities	Build Maven projects with known vulnerabilities in the dependencies	> 99% of known security vulnerabilities are reported for a set of	0%	90%

Indicator	How to measure it	Target Values	Baseline	Current
		well known dependency trees		
Report call conflicts	Build Maven projects with known call conflicts in the dependencies	> 99% of known call conflicts are reported for a set of well know dependency trees	0%	90%
Report license conflicts	Build Maven projects with known license conflicts in the dependencies	> 99% of known license conflicts are reported for a set of well known dependency trees	0%	0%

While most of the tested Java 8 projects builds were successful, it was only tested with very test focused examples so far. The 10% of failures were caused by minor reported bugs in `fasten-core` or `javacg-opal` which should be fixed shortly.

As already mentioned, the of OPAL currently used by FASTEN only have a very limited support for Java 11 bytecode which greatly limit the type of projects this plugin can analyze. It has been possible to run the plugin with simple projects in Java 11 but many fails or provide only partial results because of unsupported elements found in the bytecode.

The plugin is in waiting state for three requirements described in the previous section to fulfilling the other KPIs:

- the fasten tools and especially `fasten-core` and `javacg-opal` have not been released yet, which block the release of the Maven plugin and greatly limit the possibility for external people to try the plugin and report problems on real life use cases
- no license related metadata is injected in the FASTEN metadata database yet, so the licenses are limited to what Maven find declared in the pom.xml of each dependency
- no tool is currently available to validate the compatibility of those licenses, so there is not yet any way to report risks in this area

6 Appendix A: Documentation

6.1 `check` goal:

Here is the current configuration of the Maven plugin. It might (and most probably will) change during the development of this project to add more and refine the way to configure it based on users feedbacks.

The plugin configuration is located in the `<configuration>` element of the plugin definition in the `pom.xml` file of the Maven module.

- `failOnRisk`: true to fail the build if any risk is identifier in the dependency tree call graph
- `risks`: the list of configured risk to validate
  - `risk`: the configuration of a specific risk validation
    - `type`: the type of the risk to configure (e.g. `fasten.quality`, `fasten.license`, `fasten.security`, etc.)
    - `ignoredCallables`: a list of regular expression used to match classes/methods signatures involved in risks to ignore
    - `ignoredDependencies`: a list of regular expression used to match dependencies identifiers involved in risks to ignore
    - other options depending on the risk type
- `outputDirectory`: the folder where to store serialized call graphs
- `genAlgorithm`: the algorithm to use to generate call graphs, currently supports `CHA` (the default), `RTA`, `AllocationSiteBasedPointsTo` and `TypeBasedPointsTo`
- `fastenApiUrl`: the base URL of the FASTEN REST API to use, default is https://api.fasten-project.eu/api
- `metadataBatch`: the number of callables metadata to request to the REST API at the same time
- `metadataDownload`: control for which dependency to download metadata
  - `auto`: the default, only released (non SNAPSHOT) dependencies for which the call graph could be found on the server
  - `releases`: for all released dependency (non SNAPSHOT)
  - `all`: for all dependencies
- `serialize`: control if generated call graph should be serialized

The support for the following risk types is currently implemented:

6.1.1 "fasten.security" risk

Search for known security vulnerabilities in the call graph.

6.1.2 "fasten.binary" risk

Searching for Java conflict in the call graph (like external calls which cannot be found resolved).

6.1.3 "fasten.quality" risk

Comparing quality metrics found in the call graph with configured thresholds.

- `complexity`: the complexity above which the analyzed callable is a risk
- `length`: the length above which the analyzed callable is a risk
- `nloc`: the number of lines of code above which the analyzed callable is a risk
- `parameter_count`: the number of parameters above which the analyzed callable is a risk
- `token_count`: the number of tokens above which the analyzed callable is a risk

The support for the following risk type is currently planned:

6.1.4 "fasten.license" risk

Searching for licenses incompatibilities in the call graph.

7 Appendix B: A detailed example use case

Take the project `my-groupid:my-product` which hasn't been released yet (it is in version 1.0-SNAPSHOT for now). This module also has several dependencies (`a-product-depending-on-` and `infinispan-core`) which themselves have various other dependencies (among which `mario`).

Before releasing this project the author would like to make as sure as possible that none of its dependencies comes with a known risk or incompatibility. The FASTEN Central server contain a lot of information about those dependencies but since `my-product` is not released or deployed anywhere it is not part of the products already analyzed by FASTEN Central. Searching for known risk in each dependency by hand would be very tedious and error prone because the top level project can have a very big impact on the resolving of the transitive dependencies.

The FASTEN Maven plugin goal is to fill the gap locally to build a stitched and enriched call graph of the Maven module and all its dependencies in order to extract accurate information about this whole dependency tree. The goal is also to avoid false positives caused by risk coming from unused calls or even entire transitive dependencies which is something very common in the Maven ecosystem.

This project build descriptor (a `pom.xml` file) could look like:

```
<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4
  <modelVersion>4.0.0</modelVersion>

  <groupId>my-groupid</groupId>
  <artifactId>my-product</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>My product</name>

  <dependencies>
    <dependency>
      <groupId>some-groupid</groupId>
      <artifactId>a-product-depending-on-mario</artifactId>
      <version>42.0</version>
    </dependency>
    <dependency>
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-core</artifactId>
      <version>9.4.14.Final</version>
    </dependency>
  </dependencies>

  <licenses>
    <license>
      <name>LGPLv2.1</name>
      <url>http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html</url>
      <distribution>repo</distribution>
    </license>
  </licenses>

  <build>
    <plugins>
      <plugin>
        <groupId>eu.fasten</groupId>
        <artifactId>fasten-maven-plugin</artifactId>
        <version>1.0</version>
        <configuration>
          <!-- Fail if a problem is found in one of the dependency on
               FASTEN Central -->
          <failOnRisk>true</failOnRisk>
```

```
          <!-- The list of problems types to check -->
          <risks>
            <risk>
              <type>fasten.license</type>
            </risk>
            <risk>
              <type>fasten.security</type>
            </risk>
            <risk>
              <type>fasten.binary</type>
              <ignoredCallables>
                <!-- Class known to be provided in the target environment so we ignore
                <ignoredCallables>org.foo.LegacyBar.*</ignoredCallables>
              </ignoredCallables>
            </risk>
          </risks>
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>check</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Building this project might produce the following result:

```
...
[INFO] --- fasten-plugin:1.0:check (default) @ my-product ---
[ERROR] Errors found:
[ERROR] * the dependency some-groupid:some-gpl-dependency:42.0 license (GPLv3) is
 not compatible with the my-module license (LGPLv2.1)
[ERROR] * the dependency org.infinispan:infinispan-core:9.4.14.Final contains a
 known security vulnerability
[ERROR]    https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-10158 fixed in
 10.0.0
[ERROR] * the method call 'public boolean org.foo.Bar#isRed()' used from the call
 `public com.plumb.Mario#isNotBlue()` located in dependency com.plumb:mario:12.6
 cannot be found in the dependencies tree
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 22.971s
[INFO] Finished at: Tue Aug 11 12:07:12 EET 2021
[INFO] Final Memory: 6M/11M
[INFO] ------------------------------------------------------------------------
```

According to the analysis conducted on the stitched and enriched call graph three blockers risks were found:

- the module is using a component which is incompatible with its own license
- the module is using a component which has a known security vulnerability, fortunately a new version exist where this vulnerability is fixed
- a call located in one of the dependencies is going to fail at runtime because it is using a call which is not available (this is generally cause by conflict between several transitive dependencies expected in conflicting versions)

Also according the plugin configuration the build was failed to indicate that it is mandatory to fix those errors if we want to release this project.

References

[1] A. M. Mir, E. Lanzini, and C. Gao, “Report on application of the propagation model to security vulnerabilities,” FASTEN project, 2021.

[2] M. Zivkovic, C. Gao, L. Bergmans, and M. Bruntink, “D3.3 report on application of the propagation model to quality metrics,” FASTEN project, 2021.

[3] M. Keshani, A. M. Mir, and S. Proksch, “Documentation for REST API endpoints,” FASTEN project, 2020.