D2.2 The FASTEN Analyzer

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
v0.1	2020/12/08	Started	Mehdi Keshani, Amir M. Mir, Sebastian Proksch

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	
Deliverable	**D2.2 The FASTEN Analyzer**
Due Date	December, 2020
Submission Date	December 22, 2020
Deliverable Responsible	TU Delft
Author(s)	Mehdi Keshani Amir M. Mir Sebastian Proksch

[0]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

1 Introduction

The deliverable D2.2 (Fasten Analyzer) is a deliverable of code and data. In this document, we will provide an overview over the implemented analyzers, describe their task, and link to their implementation in the repository. Please find a high-level introduction to the analyzer architecture in deliverable D2.5 (Knowledge Base) and more information about the validation in deliverable D6.4 (Validation Matrix).

In the FASTEN project, two kinds of analyzers exist. *Preprocessing Analyzers* run in the data processing pipeline and are responsible for pre-computing data that is stored in the meta-data database. *On-Demand Analyzers* are triggered for a specific use case and perform analyses on the data that is available in the database.

2 Preprocessing Analyzers

Within the project, many analyzers are used to pre-compute data or executed as part of requests of REST API. In the following, we will briefly introduce these analyzers and point to their implementation.

2.1 Maven Crawler

`https://github.com/fasten-project/mvn-crawler`

This analyzer crawls Maven repositories and gathers Maven coordinates. It produces Maven coordinates to a Kafka topic to trigger downstream analyzers.

2.2 Pom-analyzer

`https://github.com/fasten-project/fasten/tree/develop/analyzer/pom-analyzer`

The Pom-analyzer parses the central meta-data file of a Maven project, the `pom.xml`, to extracts information about dependencies and meta-data, like the repository URL of a project. The information is published in a Kafka topic to trigger downstream analyzers, but also stored in the metadata database.

2.3 Dependency Resolver

`https://github.com/fasten-project/fasten/tree/develop/core/src/main/java/eu/fasten/core/maven`

Builds a Maven ecosystem-wide dependency graph that can be used to resolve all (transitive) dependencies and dependents of any Maven artifact.

2.4 Javacg-opal

`https://github.com/fasten-project/fasten/tree/develop/analyzer/javacg-opal`

The JavaCG-Opal analyzer uses the static analysis toolkit OPALv3 to analyze Maven artifacts and to build partial call graphs in the ExtendedRevisionCallGraph[1] representation. The partial call graphs are stored on the shared file system, downstream analyzers are notified through a message on a Kafka topic.

The call graph generator can also be locally as a standalone applications. For example, to analyze local libraries that are not available on a public repository.

2.5 Metadata-plugin

`https://github.com/fasten-project/fasten/tree/develop/analyzer/metadata-plugin`

The Metadata-analyzer consumes a Kafka topic to be notified about the successful generation of new call graphs. The call graphs are then read from the shared file-system and added into the metadata database. After successful completion, downstream analyzers are being triggered through a Kafka topic.

2.6 Graph-plugin

`https://github.com/fasten-project/fasten/tree/develop/analyzer/graph-plugin`

The graph-plugin consumes the output create by the metadata analyzer and transforms the callgraph into simple GID Graphs[2] and stores them in a RocksDB graph database. This makes it easy and efficient to traverse the call graphs (RocksDB).

2.7 Repo-cloner-plugin

`https://github.com/fasten-project/fasten/tree/develop/analyzer/repo-cloner-plugin`

The RepoCloner consumes a Kafka topic to be notified about the existence of a new POM that has repository information. The RepoCloner clones the linked repository into the shared file system and notifies downstream analyzers about a successful checkout by publishing a message in a Kafka topic.

2.8 Vulnerability-producer

`https://github.com/fasten-project/vulnerability-producer`

The FASTEN Vulnerability Producer gathers information from different sources, enriches the data with patch details, and then publishes it to a Kafka topic. It is designed to be used as a standalone tool.

2.9 Vulnerability-consumer

`https://github.com/fasten-project/fasten/tree/develop/analyzer/vulnerability-consumer`

The FASTEN Vulnerability Consumer is notified about new vulnerabilities through a Kafka topic and inserts said Vulnerability Information into the Metadata Database.

[1] `https://github.com/fasten-project/fasten/wiki/Revision-Call-Graph-format`
[2] `https://github.com/fasten-project/fasten/wiki/GID-Graph-format`

2.10 License and Compliance Producer

https://github.com/fasten-project/fasten/wiki/QMSTR-plugin,-License-and-Compliance

The License and Compliance Producer is triggered by the synchronized messages from the output of the repository cloner and call graph generator. A custom QMSTR reporter interrogates the internal graph database to fetch license information and stores the result in a Kafka topic to trigger downstream analyzers.

2.11 RAPID Quality Analyzer (Java)

https://github.com/fasten-project/fasten/tree/develop/analyzer/quality-analyzer

The RAPID Quality Analyzer (Java) is a FASTEN plugin to store/update code quality metadata from Kafka topic. There is a single Kafka message per single callable produced by Rapid Plugin using Lizard. Once a metadata message is consumed, the metadata is stored in the respective database depending on the forge (mvn, Debian, PyPI). It can be used both as a standalone tool and as a part of the FASTEN server. This analyzer is triggered by the synchronized output messages of the repository cloner and call graph generator.

2.12 RAPID Quality Analyzer (Python)

https://github.com/fasten-project/quality-analyzer

The RAPID Quality Analyzer (Python) is a FASTEN plugin that generates code complexity data for the product by analyzing its source code. The plugin consumes messages in the Kafka topics, generates code complexity data using Lizard, and produces Kafka topics with complexity data at the callable level. This analyzer is also triggered by the synchronized output messages of the source code downloader of Python and its call graph generator.

2.13 License and Compliance Consumer

Implementation in progress. Consumes Licensing data from License producer analyzer output topic and populate the data base with the meta-data that is extracted.

3 On-Demand Analyzers

FASTEN partners implement several analyzers that are built on top of the FASTEN Knowledge Base. These analyzers enable a specific use case, e.g., calculate the risk for a specific Maven artifact, and are triggered on demand. Some of the use cases might even be run in the local environment and only include the FASTEN infrastructure to request the precomputed parts.

The different use case implementations are still in the conceptualization phase and still need to be integrated into the FATEN infrastructure. To illustrate how the use

cases can be added, we will describe a several scenarios in this section that detail the concrete approach.

3.1 Quality and Risk (RAPID)

Quality analysis in FASTEN aims to calculate different quality measurements for the source code of applications and their dependencies. To achieve this, pre-computed and on-demand analysis should be used together. The pre-computation part of quality analysis is done by quality-analyzer FASTEN plug-in. Quality-analyzer[3] is a plug-in that goes through code entities, calculates quality metrics for them, and stores them as meta-data records in the database. These pre-computed meta-data records can be used later to be directly reported to users through REST API, or to be used in the on-demand analysis. Some on-demand analyses may require aggregation between multiple pre-computed metrics. For example, a FASTEN build integration tool's user wants to know how up to date is the whole dependency set that he is using. To achieve this the following steps need to be taken:

1. FASTEN build integration tool requests REST API to resolve the dependencies,
2. Based on the returned dependency set build tool requests to receive call graphs of the dependencies,
3. Build tool locally stitches the dependencies,
4. Build tool requests meta-data of the callables from REST API,
5. Build tool runs a local analysis that extracts freshness metrics of the dependencies from the meta-data and aggregates them based on the propagation of freshness on the stitched call graph,
6. Build tool shows the results to the user.

3.2 Impact Analysis

Change impact analysis goal is to investigate the impact of changing or removing a callable on the dependent packages. Such analysis can not be pre-computed, because released packages of the ecosystem are always changing and new releases are coming out constantly. Currently, we have implementations in the FASTEN code base that can calculate different kinds of centrality measures on top of the call graphs stored in the Graph Database. Extensive details about all centrality measures that we calculate in FASTEN can be found in deliverable *"5.2 Computation on Query-Independent Centralities on the Call Graphs"*. The calculation of these measures will be triggered by REST API and will be served to users based on the parameters they select. Let's assume that a user wants to calculate the number of dependent methods that will

[3] https://github.com/fasten-project/fasten/tree/develop/analyzer/quality-analyzer

break due to removing method *m1* using FASTEN build integration tool, the following steps should be taken:

1. Build tool requests the REST API to calculate the number of reachable methods from *m1*,
2. REST API creates the task of reachability analysis for *m1*, puts it into the task queue and returns the task id,
3. Once the task is given to the responsible worker it resolves the dependent packages, retrieves their call graphs, stitches them to their dependencies, and performs a reachability analysis for *m1*,
4. Build tool listens to another REST endpoint based on the given task id,
5. REST API shows the status of the task,
6. Once the status of the task is done, the build tool shows the results to the user.

3.3 Vulnerability

This analysis goal is to find out paths that use vulnerable methods. In other words, having vulnerability information in method-level enables this analyzer to precisely tell users of a vulnerable package whether they inherit the vulnerability in method-level or not. Vulnerability analysis can be pre-computed and will be combined with a local analysis part. First, an analyzer annotates callables of the ecosystem with security vulnerability information, and later another lightweight local analyzer performs a reachability analysis to identify all vulnerable methods in the graph and find actual violations for a concrete package. The pre-computation part is done by the FASTEN Vulnerability plugin[4] that collects vulnerability information from different sources of information and annotates[5] the data in the package and callable level in the form of meta-data in the database. Once this meta-data is available reachability analysis will be done locally. For instance, if a user of the FASTEN build integration tool wants to know whether there exists any vulnerability that is used in his project, the following steps should be done:

1. Build tool requests resolution of dependencies from REST API,
2. Based on the returned dependency set build tool requests to receive call graphs of the dependencies,
3. Build tool locally stitches the dependencies,
4. Build tool requests meta-data of the callables from REST API,

[4] https://github.com/fasten-project/vulnerability-producer
[5] https://github.com/fasten-project/fasten/tree/develop/analyzer/vulnerability-consumer

5. Build tool runs a local analysis to find all vulnerable methods in dependencies from the meta-data.
6. Build tool performs a local co-reachability analysis to find path in the application that reach a vulnerable method,
7. Show results to the user.

3.4 Licensing

License and Compliance analysis goal is to investigate if there is any licensing violation among all files used in applications and their dependencies. This analysis also combines pre-computations and on-demand analysis. First, an analyzer[6] goes through source files of the ecosystem and annotates the files with their licensing information. This data is stored as file-level meta-data in the database. Having this information available in the database enables a local analysis to investigate license compatibilities. This on-demand analysis, gathers licensing information of all files in a provided dependency set and investigates if any licensing violation is happening. As an example, if a user of the build integration tool wants to investigate whether there is any license violation in his dependencies, the following steps should be taken:

1. Build tool requests resolution of dependencies to REST API,
2. REST API returns a list of dependencies,
3. Based on the returned dependency set build tool requests to receive call graphs of the dependencies,
4. Build tool locally stitches the dependencies,
5. Build tool requests meta-data of the callables from REST API,
6. Build tool extracts all licensing information in dependencies from meta-data and then performs a license compatibility analysis on top of them,
7. Once the results are ready, the build tool shows them to the user.

4 Conclusion

This deliverable is a complementary report to the code and data deliverable D2.2 ("Fasten Analyzer") and provides additional information for reviewers to make it easier to understand the different parts. We have elaborated how analyzers fit into the whole FASTEN system architecture that has been presented in D2.5 ("FASTEN knowledge

[6] https://github.com/fasten-project/fasten/wiki/QMSTR-plugin,-License-and-Compliance

base"), and how we differentiate between pre-computed analysis and on-demand analysis. Furthermore, we have elaborated several example scenarios to illustrate how the different use cases might approach their analyses using the FASTEN infrastructure. For further information that goes beyond the deliverables, please refer to the deliverables D2.4 (REST API), D2.5 ("FASTEN knowledge base"), or D2.9 (Server Plug-Ins).