KOWALSKI: Collecting API Clients in Easy Mode

Manuel Leuenberger, Haidar Osman, Mohammad Ghafari, and Oscar Nierstrasz
Software Composition Group, University of Bern
Bern, Switzerland
{*leuenberger, osman, ghafari, oscar*}*@inf.unibe.ch*

***Abstract*—Understanding API usage is important for upstream and downstream developers. However, compiling a dataset of API clients is often a tedious task, especially since one needs many clients to draw a representative picture of the API usage.**

In this paper, we present KOWALSKI, a tool that takes the name of an API, then finds and downloads client binaries by exploiting the Maven dependency management system. As a case study, we collect clients of Apache Lucene, the de facto standard for full-text search, analyze the binaries, and create a typed call graph that allows developers to identify hotspots in the API.

A video demonstrating how KOWALSKI is used for this experiment can be found at https://youtu.be/zdx28GnoSRQ.

***Index Terms*—API client collection; API usage analysis; repository mining; dependency management systems;**

I. Introduction

Understanding API usage is important for upstream and downstream developers. For upstream developers it is important to know how their APIs are used, so that they can estimate the impact of changes. Downstream developers require a self-explanatory API in the best case or at least documentation [1]. With the lack of documentation, usage examples of an API serve as a good entry point to learn and explore the API [2], but finding the clients of a specific API to extract usage patterns is a non-trivial task. Many studies therefore analyze a few hand-selected projects to mine the API usage [3][4]. Others collect usage patterns by analyzing a large corpus of projects and select those patterns with the highest support [5][6]. Few studies mine unit test cases to synthesize API usage examples when other sources of client code are rare [7]. Nevertheless, high diversity exists in API usage [8] since an API can provide many callable methods, while different clients make use of different subsets of them. Therefore, to find different possible usages of an API, one should find enough client code to cover as many usage scenarios as possible.

In this paper we present KOWALSKI, a tool to collect clients of specific Java APIs. KOWALSKI exploits the widespread use of Maven as a dependency management system. KOWALSKI takes the name of an API as an input, crawls Maven repositories, and outputs JAR (Java ARchive) artifacts of the API clients, including their dependencies. While most existing large-scale miners operate on sources with limited type-awareness only [9], KOWALSKI enables type-aware analysis of API clients as it collects them in bytecode format. The classes referenced in the client bytecode can be resolved, so that typed call graphs can be constructed. The call graphs could be used to extract protocols, *i.e.*, methods that need to be called in a certain order [10][11][12]. Moreover, KOWALSKI facilitates tracking of the evolution of clients and APIs, as collected artifacts are tagged with their version numbers. The source code of KOWALSKI is available on GitHub.[1]

We use KOWALSKI to collect clients of Apache Lucene, the de facto standard for full-text search, available in the Maven Central repository.[2] Within one hour KOWALSKI collects 7 755 client artifacts of Lucene, for which we extract call graphs in six hours. From the call graphs we determine in how many clients a Lucene method is used and how often a method is used in those clients. API developers can use this information to distinguish between API hotspots, which affect many clients if changed, and cold spots, which can be changed with little impact. We find hotspots in the high-level API methods to create queries and documents, and to read from and write to the full-text index. Cold spots are API methods that deal with file format of the index. The dataset, KOWALSKI binaries, and setup scripts for this experiment can be fetched from Figshare.[3]

We also use KOWALSKI to collect clients of Apache Lucene to infer the nullness of API methods [13], *i.e.*, whether a method may return a null value or not.

The rest of the paper is structured as follows: In section II we describe how we collect clients of specific APIs with KOWALSKI, and we discuss the tool's architecture and implementation in section III. The hotspot identification in Apache Lucene, for which we use KOWALSKI to collect the client binaries, is presented in section IV. In section V we discuss limitations of our approach that could affect its applicability. A comparison with other API usage tools and datasets can be found in section VI. We conclude this paper and discuss future directions in section VII.

II. API Client Collection

We want to find clients of a specific API, so that our downstream analysis finds many API calls. We also require the called methods to be precisely identifiable. For that we need type information about the called methods, namely method signatures and declaring types. As APIs evolve over time, methods may be added, removed, or change the contract. Different versions of an API may co-exist. For example, the method `org.apache.lucene.search.Weight.scorer()` never returns null in an early version of Lucene, but does so in later versions.[4] Hence a method invocation must be traceable

[1] https://github.com/maenu/kowalski
[2] http://search.maven.org/
[3] https://figshare.com/projects/KOWALSKI_ICSME_Tools_2017/22756
[4] https://issues.apache.org/jira/browse/JCR-3481

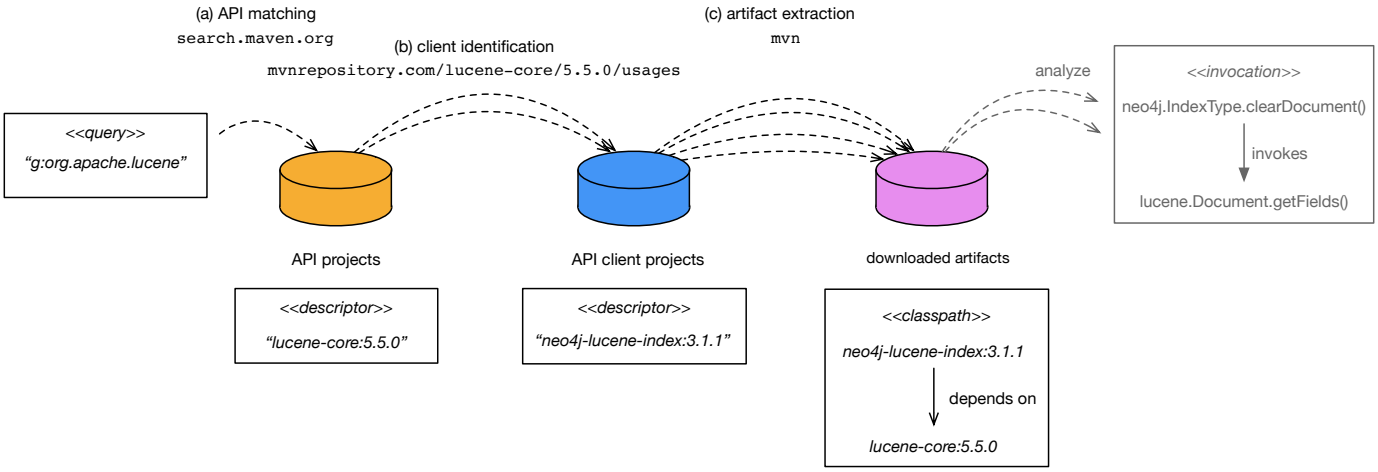

Fig. 2. Dataflow in KOWALSKI from the initial query to the class path of dowloaded binaries to analyze.

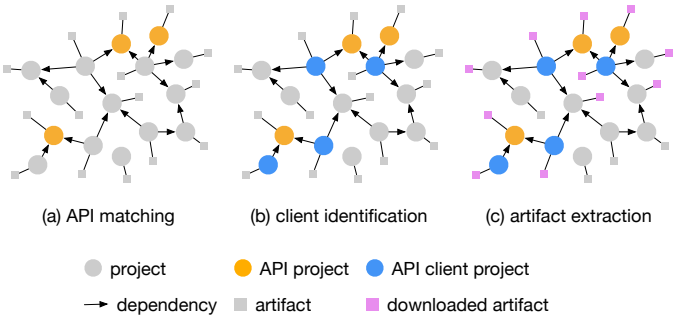

Fig. 1. Dependency subgraph extraction steps with project nodes marked as API, API client and artifacts.

to the method signature, the declaring class, the declaring API, and the API's version. This information constructs a *universally unique method identifier*. We need a way to find the API clients and the universally unique identifier of the called API methods.

Collecting clients of a specific API means that we need to extract a subgraph of the dependency graph spanned by all projects, as shown in Figure 1. First, we need to match the APIs for which we want to collect clients (a). Second, we need to identify clients of the matched APIs (b). Third, we need to extract the API client artifacts we want to analyze (c). These artifacts can be sources, binaries, or documentation.

Many Java projects use Maven as a dependency management system, which we can exploit for our purpose. Maven projects declare their dependencies in a meta-data file. For example, one version of Neo4j declares the artifact descriptor `org.neo4j:neo4j-lucene-index`, version `3.1.1`, and a dependency to Lucene version `5.5.0`. Neo4j developers use Maven to automatically collect the required Lucene dependency from a package repository. Just as Lucene, Neo4j itself is published to this package repository. Therefore, both the API and its client are stored in the same repository and linked through the declared dependency.

III. IMPLEMENTATION

We implement the aforementioned dependency subgraph extraction in KOWALSKI. KOWALSKI is designed for high concurrency and collects artifacts rapidly. In Figure 2 we show how the extraction is implemented using the three *tasks* (a, b, c) as introduced in Figure 1. Each task is run in a *job* that pipes the input and output of the tasks between *streams* of cached intermediate results.

A. Tasks

The three tasks are decoupled from each other, so that multiple instances of the same task run in parallel. The output of one task is the input of another task, which enables piping different tasks in sequence.

The *API matching* task (a) finds projects that match a query for *Maven Central Search*.[5] For example, to find all Apache Lucene versions, the task takes a query in the form `g:org.apache.lucene` as input. The task then collects all matching artifacts and outputs their descriptors, *e.g.*,`org.apache.lucene:lucene-core:5.5.0` for Lucene 5.5.0.

The *client identification* task (b) accepts an artifact descriptor and collects their clients. Clients of an artifact are found by scraping the *mvnrepository* website.[6] The task outputs artifact descriptors again, *e.g.*,`org.neo4j:neo4j-lucene-index:3.1.1` for Neo4j 3.1.1, as it is a client of Lucene.

The *artifact extraction* task (c) takes an artifact descriptor and collects the corresponding JAR binaries, including dependencies. This task can be configured using a traditional Maven *setting.xml* to declare the repositories the JARs should be fetched from. It is also possible to configure the dependency scopes that should be used to resolve the necessary dependencies. For instance, a dependency to a unit testing framework

[5]See http://search.maven.org/#api REST API

[6]For example https://mvnrepository.com/artifact/org.apache.lucene/lucene-core/5.5.0/usages

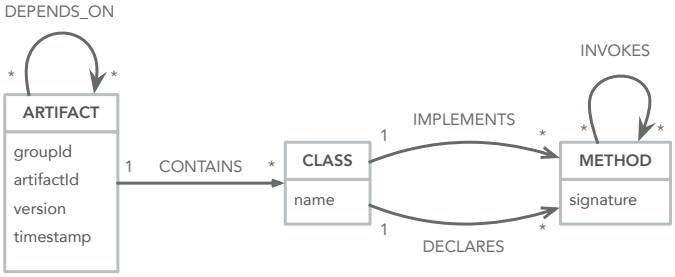

Fig. 3. Neo4j database schema for the analysis call graph.

is declared in the *test* scope. If we were to analyze tests, we could configure the artifact extraction task to include the test dependencies.

B. Jobs

The tasks are executed in jobs, denoted by the dashed arrows in Figure 2. Jobs are responsible for providing input for the wrapped tasks and deciding what to do with the task's output. The output stream of one job is the input stream of another job. The chaining of jobs through streams acts as the composition mechanism to build the pipeline required to collect the dataset. A job reads the task's input from an input stream, delegates the input to the task to process, and writes the task's output to an output stream. Since tasks are independent, and the access to input and output streams is synchronized, jobs can be executed concurrently. Streams are named, so that a job can be configured by the type of the task it executes together with the names of the input and output streams.

C. Collector

The collector runs multiple workers that are responsible for carrying out the jobs. A worker is merely a thread running a job. All workers are executed in parallel by the collector. The collector can be configured by specifying how many workers for each job are instantiated. At startup, all jobs are created based on this configuration and executed with the number of workers desired.

IV. Experiment

For our experiment we extract call graphs of Apache Lucene clients and identify hotspots in the API. We deploy Kowalski to a multi-core Ubuntu server to collect the clients. The collection and analysis runs on a 64 bit Ubuntu machine with 32 cores at 1.4 GHz and 128 GB of RAM. We use Apache Artemis as a JMS server to persist the streams of intermediate results and we deploy Neo4j as the database to store the extracted call graphs. The artifact extraction task from Kowalski writes downloaded JAR and POM artifacts to the local Maven repository, so that they can be read by the downstream analysis. We run 16 worker threads to collect the dependencies of multiple API clients in parallel. The collected clients are analyzed in 4 concurrent processes.

The analysis applied to the API clients processes all methods that are defined in classes of the client artifact. All invocations of Lucene methods are extracted. The invoked methods are tracked to the defining classes and Lucene version, so that for each analyzed artifact, a typed call graph is stored in the database. The schema of the database is shown in Figure 3. As each artifact is stored exactly once in the database, different clients calling the same API method in the same API artifact are reflected by multiple incoming invocations of the same method. Therefore, it is not just that we have a call graph for each analyzed client, but we have an aggregated call graph over the whole dataset.

TABLE I
Client major versions per Lucene major version, invoked Lucene methods, and their invocations.

Lucene version	clients	methods	invocations
1	1	23	37
2	48	999	8 819
3	87	1 141	11 619
4	109	1 446	12 830
5	45	1 794	24 773
6	35	1 299	6 720
7	3	4	7

It takes one hour to collect the 7 755 identified clients and six hours to analyze them. The analysis of a single client takes eleven seconds on average. As the analysis starts as soon as the binary artifacts of the first client are extracted, the whole process terminates within six hours. 1 685 binaries are part of Lucene itself, as it is a multi-module project. In 3 009 of non-Lucene binaries we find invocations to Lucene. In the remaining 3 061 binaries we cannot find Lucene usage, as Lucene is a transitive dependency and not directly used by the analyzed methods. The binaries belong to 186 different projects identified by the unique artifact descriptor. We group Lucene and client releases by major version to get an overview of the usage, as shown in Table I. For Lucene versions 1 and 7 we observe very small usage, so we ignore them in the experiment hereafter.

Figure 4 shows for each major Lucene version how widely a method is used among clients and how often it is invoked when used. The product of these two usage metrics results in the number of total invocations of a method, denoted by the color of a method point. This evaluation can serve as an estimation of the impact when Lucene changes its API methods. From the plots we can read how many call sites need to be refactored when a Lucene method changes its signature and how many clients are affected. We find hotspots in the high-level API methods to create queries and documents, and to read from and write to the full-text index. Cold spots are API methods that deal with file format of the index. We observe that the general usage changes over time. While some methods are used in 40 out of 48 clients of Lucene 2, the most widespread methods are used in only 17 out of 35 clients for Lucene 6. In newer Lucene versions, methods are generally invoked more often. Two projects, Elasticsearch and Solr, which have grown together with Lucene, are responsible for this phenomenon.

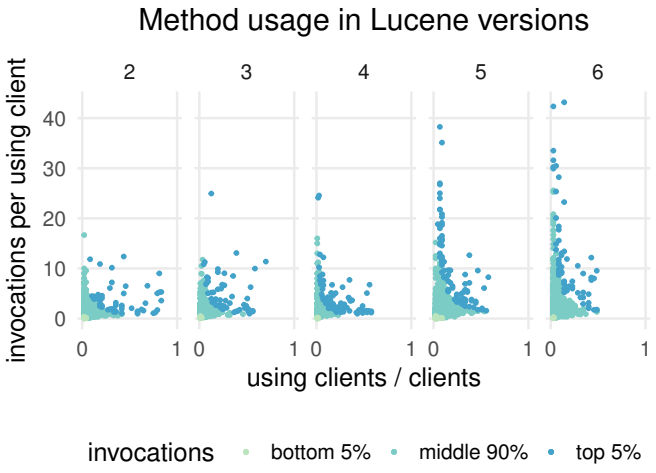

Fig. 4. Method usage in Lucene versions 2 to 6. We show in how many clients a method is used, and how often it is called in those clients.

V. Limitations and Threats to Validity

For our experiment KOWALSKI collects only Lucene and other OSS projects that depend on Lucene and are published on Maven Central. First, this excludes all closed source projects. If given access to a company repository, KOWALSKI can also be used to collect a dataset of clients of a company-internal library. The static analysis can be reapplied as well. Second, all open source projects that are not published on Maven Central are excluded. There are other popular Maven repositories that may contain other Lucene clients, for example jcenter[7] and clojars.[8] However, Maven Central is a large repository that serves 1 935 045 versions of 185 693 artifacts.[9] Package repositories are primarily used to distribute reusable libraries, therefore our dataset has a strong bias towards libraries as clients. Libraries may use Lucene differently than projects further down the dependency hierarchy. Third, we lack a measure to estimate how many Lucene clients are only distributed as sources, for example on GitHub. As our analysis is tailored to run on binaries, it would require a build of these projects. While building arbitrary projects from source is non-trivial [6], we conjecture the widespread use of Travis CI among active projects might facilitate this issue. The identification of clients of a dependency on GitHub requires a searchable index similar to *mvnrepository* for Maven, but for GitHub projects. Parsing POMs alone will not detect clients that rely on a transitive dependency [14]. Identifying transitive dependencies requires resolving all dependencies of a projects, which is an time-intensive task. Fourth, we only analyze the Lucene ecosystem, and the results may not generalize to other ecosystems.

[7]https://bintray.com/bintray/jcenter
[8]https://clojars.org/
[9]https://search.maven.org/#stats, date of access May 3, 2017

VI. Related Work

There are several datasets over large parts of GitHub. Google's BigQuery GitHub dataset[10] can be queried for contents of source files. It even runs static analysis remotely,[11] but type information is not provided and must be reconstructed. Dyer *et al.* provide ASTs that include partial type information from sources in GitHub projects in the Boa dataset [9]. The binaries collected by KOWALSKI provide more type information as we can track method invocations to the invoked method and library version without ambiguity. By choosing Maven repositories as our datasource we are restricted to a smaller set of OSS projects than GitHub, but we gain type precision.

Lämmel *et al.* check out 6 286 SourceForge projects and manage to build 1 476 of them with Ant to analyze them for API usage [6]. They manually search for missing dependencies to fix build errors in 15% of the built projects. Instead of building projects from source, we collect binary Maven artifacts with resolved dependencies. We use the SOOT analysis framework that creates phantom references for unresolvable classes.

Sawant *et al.* build a typed dataset for five APIs and their usages in 20 263 GitHub projects using Maven [15]. They use partial compilation to work around unresolvable classes. By compiling from source, projects can be inspected for any revision of a source file in a version control system. Our dataset includes only built binary artifacts, yet they are versioned as well, therefore we can track the evolution of a project as well, although on a coarser level of releases.

VII. Conclusions

We present KOWALSKI, a tool to collect API clients for API usage analysis. Our experiment shows that KOWALSKI is performant and produces datasets that can be analyzed to find hotspots.

The KOWALSKI pipeline can be used to collect datasets about any API that is hosted in the supported Maven repositories, from large ecosystems around Apache, Eclipse, or Mozilla artifacts to more focused sets of clients of a single product such as Hibernate, JUnit, or Guava. KOWALSKI can also be used on a company internal repository. The collected clients of closed source frameworks and libraries can be analyzed to identify hotspots.

To conquer the bias in the collected datasets towards libraries, one future direction is to utilize Travis CI for extending KOWALSKI with a task to build projects from GitHub.

Acknowledgments

The authors gratefully acknowledge the funding of the Swiss National Science Foundations for the project "Agile Software Analysis" (SNF project No. 200020_162352, Jan 1, 2016 - Dec. 30, 2018).[12]

[10]https://cloud.google.com/bigquery/
[11]https://medium.com/google-cloud/static%2Djavascript%2Dcode%2Danalysis%2Dwithin%2Dbigquery%2Ded0e3011732c
[12]http://p3.snf.ch/Project-162352

REFERENCES

[1] N. Haenni, M. Lungu, N. Schwarz, and O. Nierstrasz, "A quantitative analysis of developer information needs in software ecosystems," in *Proceedings of the 2nd Workshop on Ecosystem Architectures (WEA'14)*, 2014, pp. 1–6. [Online]. Available: http://scg.unibe.ch/archive/papers/Haen14a-QuantitativeEcosystemInformationNeeds.pdf

[2] R. P. L. Buse and W. Weimer, "Synthesizing API usage examples," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 782–792. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337316

[3] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: From usage scenarios to specifications," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 25–34. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287630

[4] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy API usage patterns in Android apps: an empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 2–11.

[5] C. D. Roover, R. Laemmel, and E. Pek, "Multi-dimensional exploration of API usage," in *2013 21st International Conference on Program Comprehension (ICPC)*, May 2013, pp. 152–161.

[6] R. Lämmel, E. Pek, and J. Starek, "Large-scale, AST-based API-usage analysis of open-source Java projects," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11. New York, NY, USA: ACM, 2011, pp. 1317–1324. [Online]. Available: http://doi.acm.org/10.1145/1982185.1982471

[7] M. Ghafari, K. Rubinov, and M. M. Pourhashem K., "Mining unit test cases to synthesize API usage examples," *Journal of Software: Evolution and Process*, pp. e1841–n/a, 2017, e1841 smr.1841. [Online]. Available: http://dx.doi.org/10.1002/smr.1841

[8] D. Mendez, B. Baudry, and M. Monperrus, "Empirical evidence of large-scale diversity in API usage of object-oriented software," in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*. IEEE, 2013, pp. 43–52.

[9] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 422–431. [Online]. Available: http://design.cs.iastate.edu/papers/ICSE-13/icse13.pdf

[10] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 383–392. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595767

[11] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking API protocol conformance with mined multi-object specifications," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 925–935. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337332

[12] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Path-sensitive inference of function precedence protocols," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 240–250.

[13] M. Leuenberger, H. Osman, M. Ghafari, and O. Nierstrasz, "Harvesting the wisdom of the crowd to infer method nullness in Java," in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, [to appear].

[14] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, pp. 1–34, 2017.

[15] A. A. Sawant and A. Bacchelli, "A dataset for API usage," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 506–509.