CALAPPA: A Toolchain for Mining Android Applications

Vitalii Avdiienko[♣], Konstantin Kuznetsov[♣], Paolo Calciati[♠], Juan Carlos Caiza Román[♠], Alessandra Gorla[♠], Andreas Zeller[♣]

[♣]Saarland University
Germany
lastname@cs.uni-saarland.de

[♠]IMDEA Software Institute
Spain
firstname.lastname@imdea.org

ABSTRACT

Software engineering researchers and practitioners working on the Android ecosystem frequently have to do the same tasks over and over: retrieve data from the Google Play store to analyze it, decompile the Dalvik bytecode to understand the behavior of the app, and analyze applications metadata and user reviews. In this paper we present CALAPPA, a highly reusable and customizable toolchain that allows researchers to easily run common analysis tasks on large Android application datasets. CALAPPA includes components to retrieve the data from different Android stores, and comes with a predefined, but extensible, set of modules that can analyze apps metadata and code.

CCS Concepts

•**Software and its engineering** → *Automated static analysis; Software libraries and repositories;* •**Information systems** → Document topic models; •**Computing methodologies** → Anomaly detection;

Keywords

Android app mining; Android Analysis; App market analysis

1. INTRODUCTION

The Android ecosystem, which is accessible by means of the many app stores such as the Google Play, boosted the creativity and the research possibilities of many software engineering researchers and practitioners. Whether a new software engineering technique needs software artifacts to be evaluated [5, 7, 4], or rather there is the need of large amounts of data to run statistically significant software engineering studies [9, 10], the Android ecosystem is a precious source of data and information.

However, despite the numerous attempts to allow researchers to replicate previous research studies thanks to publicly available datasets [1], and despite the numerous research techniques and corresponding open source tools that are available [11], there is a lack of common frameworks to minimize researchers' efforts in terms of development to run their studies. In essence, the problem is that researchers have to re-implement the same tasks over and over: They have to write scripts to crawl Google Play (or similar stores) to download data, retrieve the metadata information of a specific app to have further information on it (e.g. which category the app belongs to, who developed it, its natural language description on the store etc.), analyze the manifest to extract the list of permissions that an app requires, and so on, just to mention few of the common tasks they have to do for many studies.

In this paper we present CALAPPA, a set of ready-to-use modules that allow to quickly run common analysis tasks on the Android ecosystem. Examples of such tasks are crawling an Android store, extract app descriptions, and run simple analyses on the Dalvik bytecode. CALAPPA's modules can be combined in a pipeline as users want, and the whole tool-chain can easily be extended with new modules.

This tool-chain model comes handy for researchers for several reasons: it allows them to reuse code and thus speed up development time; it allows researchers to easily share the results of their studies such that other researchers can reproduce them; it is highly customizable and can be easily extended to run different analyses.

CALAPPA includes modules to retrieve data from different sources (e.g. the Google Play store), modules to parse and manipulate apps metadata such as the natural language description, and modules to analyze the app code (e.g. to extract the sensitive information flows). Thus, for instance, running a new study as the one presented in [7, 8] would require to simply use the CALAPPA module to download several thousands apps from Google Play, process descriptions thanks to modules that remove stop-words, do stemming, and apply topic modeling on them. Finally it would require to extract the sensitive API calls for each app, and use these as features for anomaly detection. If one would want to replace sensitive API calls with information flows as features for the anomaly detection (i.e, similar to what [4] presents), researchers would simply need to use the CALAPPA module to extract the sensitive information flows instead.

The remainder of the paper is structured as follows: Section 2 provides an overview of CALAPPA as a tool-chain and presents the key underlying technologies that we used to implement it. Section 3 presents the components to download the data from Google Play and similar app stores. Section 4 describes the components to run several analyses on the metadata information (e.g. extract and process description). Section 5 introduces the key modules to analyze and extract information from the apk code (e.g. information flows, sensitive API calls). Section 6 describes some of the components that CALAPPA includes for post-processing the data (e.g. to run anomaly detection and export data).

2. TASKS AND TARGETS IN A PIPELINE

Analyzing a large dataset of Android apps may require several days. The risk with such long-running batch processes is that some tasks may fail at some point (e.g. because of a transitive problem or because of a bug in the code to analyze a particular app), and invalidate days of work. Ideally users want to save the intermediate results, want to see the execution progress, and want to resume the execution from intermediate results.

In order to provide these —and other— features, we built CALAPPA on top of Luigi, a Python framework to implement and run long-batch pipelines. Beside the previously mentioned features, Luigi allows to easily parallelize operations, in order to use the available computation power at its best. Luigi is based on top of two main concepts: Tasks and Targets. *Targets*, in essence, correspond to files on the disk, or some kind of similar checkpoints. *Tasks* specify what work should be done, on what inputs, and what Target should be produced. Each Task should implement three methods:

```
//returns the dependencies
def requires(self):
//returns the target this tasks produces
def output(self):
//includes the implementation of the task
def run(self):
```

We implement a separate Task for each of the main functionalities of CALAPPA. We then use Google protobuf to produce Targets. This is a very efficient library to define and serialize structured data, and it makes it easy to allow the communication among different modules in the CALAPPA tool-chain.

CALAPPA is thus simply a collection of Luigi tasks that communicate thanks to protobuf streams. CALAPPA allows to create different pipelines that involve different modules that typically cover the following phases: retrieving data from one of the available sources (Section 3), metadata analysis (Section 4), application code analysis (Section5), and further analyses on the extracted data (Section 6).

3. DATA RETRIEVAL

CALAPPA includes different modules to retrieve Android related data, and they can be used depending on what information users desire. In essence, CALAPPA allows to retrieve an arbitrary amount of Android data that include:

- App metadata (including natural language description, list of requested permissions, developer information and app category)
- The list of user reviews.
- Latest app binary code.
- App source code.
- Specific app binary release.

A user can decide to download an arbitrary number of the most recent Android binaries thanks to the Google Play crawler (Section 3.1), a specific (or a range of) binary release from the apk mirror repository (Section 3.2), or a large number of open source apps from the F-Droid repository (Section 3.3). We now describe each module in further details.

3.1 Google Play Crawler

The Google Play crawler is the most valuable module to retrieve Android related data, since it crawls Google Play, which is the most popular Android store. This module has been built on top of the unofficial Google Play python API [1].

This module allows users to select apps based on specific criteria including:

- Category filtering. We allow users to select a specific Google Play category or to keep all of them into account.
- Package name filtering. The crawler can consider only apps whose package name matches a specific string.
- Rate filtering. The crawler allows to download apps with a star rating above a specific value.

Given the previously mentioned set of filtering criteria, the user needs to specify how many apps to download and which information to retrieve. Regarding what to download, a user can choose any of the following data (by default all of them would be downloaded):

- The apk binary
- The app metadata, which include information on the developer, the category of the app, the natural language description, the average rating given by users, and the list of permissions.
- The most recent 500 reviews done by users.

The Google Play crawler uses configurable proxies, and thus can retrieve the data from different locations and stores (US, Germany, Spain, etc.). This feature is important if a user wants, for instance, to retrieve app descriptions mainly in a specific language.

3.2 APK Mirror Crawler

The Google Play store has a major limitation for researchers: It allows to retrieve only the most recent release of the application binary. Thus, when researchers need multiple releases of the same application, they either have to let the Google Play crawler run continuously for several weeks, in order to retrieve new releases of the same app as soon as developers release them, or may resort to the *APK Mirror crawler* module that CALAPPA offers. This module crawls www.apkmirror.com, a website storing several releases of the same Android apps uploaded directly by users. For each apk file, this website stores the MD5 checksum and the upload date.

This module allows to easily retrieve multiple versions of the same app. Given an app name, the module allows to specify multiple filters to obtain either a list of download URLs of all apks matching the app name, or to download the apk files directly.

The crawling process first searches for apks matching the provided name, and then parses the search result pages, which contain links to the corresponding apk download page, and it adds them to a working list. In the second step the crawler visits each download page in the working list and extracts the download URL together with relevant information displayed on the page, including the MD5 checksum of the apk file. Download pages contain a download URL and various information about the apk, such as size and upload date: the crawler compares this data with the search parameters specified by the user.

In addition to the apk package name, which is mandatory, users can specify the following parameters when searching for APKs:

- APK MD5 checksum
- APK architecture
- APK version
- Minimum required Android API
- Release date interval
- Exclude/include alpha and beta versions

[1]http://github.com/egirault/googleplay-api

If the apk info matches the search parameters, the crawler downloads the apk and saves the download URL for future usage purposes. The primary purpose of this module is to download several binary releases of the same app at the same time without waiting for weeks or even months for them to appear on Google Play. An alternative, but not less interesting, possible usage of this module is to share datasets by simply sharing APK download URLs instead of APK packages directly.

3.3 F-Droid Crawler

App source code may be required for some research studies. While source code is not available for most apps published on the Google Play and alternative stores, there exists F-Droid, a large repository for free and open-source Android apps. CALAPPA thus offers a module to retrieve data from this repository too.

It allows users to specify the number of apps to be downloaded, using category and package name as filtering criteria. Apps available on this repository come with links to a repository and to an issue tracker, and thus allow many studies that require this type of information.

4. METADATA PROCESSING

A large set of CALAPPA modules provide functionalities to parse metadata information on the set of Android apps under analysis. The most relevant functionalities allow to easily extract information from the Android manifest and to filter and analyze natural language descriptions. We now describe some of the most relevant features and corresponding modules.

4.1 Android Manifest Parsing

CALAPPA offers a module to parse and extract relevant information from the manifest file of each apk. Out of the manifest file, this module extracts information on the developer, and on the dependencies the app requires (e.g. the target Android version, if available). Regarding the app itself, it extracts the following information:

- app fullname
- package name
- app version
- app category, as declared by the developer
- app raw description, as provided by the developer
- list of permissions, as requested by the developer

Some of these elements, such as the natural language description or the list of permissions, can be further analyzed by other modules that we describe in the following sections.

4.2 Natural Language Description Analysis

CALAPPA implements several features to clean, filter and analyze the app natural language descriptions that have been extracted from the Android manifest file by means of the module we described in Section 4.1. We implemented simple tasks as separate modules in order to facilitate reuse. Users can simply use multiple natural language processing modules by concatenating them in the CALAPPA pipeline. Each module related to app descriptions requires the following parameters: the path to the corpus of apps metadata, extracted by the metadata analysis module, and this has to mandatorily include the app name and the app description. Moreover, each module needs to be aware of the working language to use the appropriate dictionaries. English is the default language for all these modules, given the predominance of the language.

CALAPPA implements modules for the most common tasks, including:

- **Stop words removal**: It removes common words such as conjunctions and articles from each app description. The module uses the default stop words dictionary provided with the nltk Python package, but can take an alternative dictionary (with custom/domain specific terms) as an optional parameter.

- **Stemming**: It keeps the root of each word in each description to limit the whole corpus of terms. Thus, for instance, words such as "player", "playing" and "plays" would all be transformed to the common root "play". This step is useful as a pre-analysis of topic modeling, for instance. This module is also based on the nltk package.

- **HTML tags removal**: Most descriptions contain HTML tags to improve readability for users. Such tags, though, should most of the times be removed before any textual analysis. This module simply removes such tags from the description thanks to the HTML parser module.

- **Term frequency in corpus**: There are many scenarios for which term frequencies may be of interest. For instance, users may want to identify the most common terms in all app descriptions (after stop words removal) in order to remove them. For instance, in the Android corpus terms such as "Android" and "app" would appear in pretty much each descriptions. Removing them would would thus reduce the description size and would not loose specific description semantics. Users may also want to identify terms with the lowest frequencies, as these may also not be relevant. For instance, they may reveal the presence of typos.

- **Different language text removal**: Descriptions contain language in different languages. Most often, they are largely in English, and they contain shorter descriptions in other languages to catch the attention of regional audience. Many techniques assume to work with a single language (e.g. English), and would thus need to remove non-English text before processing. This module uses Google's Compact Language Detector to first break descriptions into paragraphs, and for each paragraph it identifies its likely language, and will finally remove anything that does not belong to the language of interest. In future we plan to include features to automatically translate these paragraphs rather than removing them.

5. PROGRAM ANALYSIS

Arguably the most interesting artifact related to an Android application is its code. Many techniques analyze the code in the apk file for multiple reasons, including to optimize it (e.g. to make it energy efficient), to test it, or to understand its behavior (e.g. to identify malicious behavior). CALAPPA currently offers three modules to analyze the Dalvik bytecode of each Android app under analysis:

- **Sensitive API calls**: CALAPPA transforms the Dalvik bytecode into the SMALI intermediate representation thanks to apktool [2], and then parses it to extract the frequency of sensitive API calls that occur in each app. We consider sensitive API calls those that require at least one permission. CALAPPA uses [3] to have an accurate mapping between APIs and permissions.

[2]http://ibotpeaches.github.io/Apktool

- **Essential permissions**: Studies report that many applications are over-privileged, meaning that some of the permissions that an app asks for are not used [6]. CALAPPA offers a module to extract the actual *list of required permissions* of an app (i.e. permissions that are used at least once by a sensitive method call).

- **Information flows**: CALAPPA offers a module that statically analyzes the Dalvik code and extracts the sensitive information flows (e.g. the device IMEI is sent to an unknown server). In order to compute information flows, CALAPPA resorts to using a customized version of FlowDroid [2]. More details on how our implementation differs from the original FlowDroid can be found in [4].

6. POST-PROCESSING ANALYSES

Thanks to the modules described in Section 4 and 5, it is possible to easily extract information from the app metadata and from the program code. Many analyses can be done on these data. CALAPPA provides modules that implement analyses that have been used in previous research works, and are likely to be reused in the future for related works:

- **Clustering**: CALAPPA has a module to automatically group samples in clusters. Samples can be anything (e.g. applications, reviews, API calls) and can have an arbitrary number of features. This module uses K-means, a well known clustering algorithm and can automatically identify the best number of clusters.

- **Topic modeling**: CALAPPA provides a module to identify "topics", i.e. sets of terms that frequently occur together, out of a corpus of documents. While most of the works on Android used this technique on descriptions [7, 8], this module can work on any other feature as well. CALAPPA uses the Mallet library to provide this functionality [3].

- **Anomaly detection**: Users may want to identify anomalous samples within clusters. This is useful, for instance, to identify malware or grayware if features that describe the behavior of an application are used as features. CALAPPA uses the ORCA package [4] to identify anomalies, and requires users to specify a threshold parameter above which anomalies are reported as such.

- **Virus Total**: Users who are analyzing application behavior looking for malware may need to compare their analysis against a ground truth. For this purpose, CALAPPA provides a module that reports whether an Android app is malicious or not, based on what information VirusTotal [5] provides about it. CALAPPA uses the public VirusTotal API to provide this feature.

7. CONCLUSION

In this paper we presented CALAPPA, a tool-chain to easily retrieve Android datasets and analyze them. CALAPPA comprises several ready-to-use modules that already implement the most common tasks on Android data. Running long batch processes with CALAPPA is easy, since users can easily deal with failures and can pause and resume executions in a cost efficient way. CALAPPA also allows to parallelize analyses to achieve results in less time. CALAPPA can be easily configured and extended with new modules.

To learn more about CALAPPA and our work check:

https://www.st.cs.uni-saarland.de/appmining/

[3]http://mallet.cs.umass.edu/

[4]http://www.stephenbay.net/orca

[5]http://www.virustotal.com

8. ACKNOWLEDGMENTS

This work was supported by the European Research Council, project "SPECMATE", the European Union FP7-PEOPLE-COFUND project AMAROUT II (grant n. 291803), by the Spanish Ministry of Economy project DEDETIS, by the Madrid Regional Government project *N-Greens Software* (grant n. S2013/ICE-2731), and by the EIT Digital project SMAPPER.

9. REFERENCES

[1] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of MSR*, pages 468–471, 2016.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of PLDI*, 2014.

[3] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: Analyzing the android permission specification. In *Proceedings of CCS*, pages 217–228, 2012.

[4] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *Proceedings of ICSE*, pages 426–436, 2015.

[5] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? (E). In *Proceedings of ASE*, pages 429–440, 2015.

[6] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of CCS*, pages 627–638, 2011.

[7] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proceedings of ICSE*, pages 1025–1035, 2014.

[8] K. Kuznetsov, A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Mining android apps for anomalies. In *The Art and Science of Analyzing Software Data*, pages 257–281. Morgan Kaufmann, 4 2015.

[9] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman. A survey of app store analysis for software engineering. Technical report, University College London, 2016.

[10] A. von Rhein, T. Berger, N. S. Johansson, M. M. Hardø, and S. Apel. Lifting inter-app data-flow analysis to large app sets. Technical Report MIP-1504, Department of Informatics and Mathematics, University of Passau, September 2015. Technical Report MIP-1504, Department of Informatics and Mathematics, University of Passau.

[11] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of CCS*, pages 1329–1341, 2014.