AndroLyze: A Distributed Framework for Efficient Android App Analysis

Lars Baumgärtner, Pablo Graubner, Nils Schmidt, Bernd Freisleben
Department of Mathematics & Computer Science, University of Marburg
Hans-Meerwein-Strasse 6, D-35032 Marburg, Germany
Email: {baumgaertner, graubner, schmidt89, freisleb} @informatik.uni-marburg.de

Abstract—In recent years, the number of mobile applications has grown significantly. Not surprisingly, various security and privacy concerns associated with mobile applications have emerged. Several researchers are addressing these problems by analyzing the security properties of mobile application code. Most of the security checks rely on custom scripts and are quite heterogeneous with respect to dependencies, deployment and reporting. In this paper, we present AndroLyze, a distributed framework with unified logging and reporting functionality to perform security checks on large numbers of applications in an efficient manner. AndroLyze provides optimized scheduling algorithms for distributing static code analysis tasks across several machines. Moreover, AndroLyze can handle several versions of a single mobile application to generate a security track record over many versions. To demonstrate the benefits of AndroLyze, we have analyzed the Top Free 500 Android applications of all categories in Google Play collected over three years. The resulting data set consists of almost 40,000 mobile applications and requires about 227 GB of storage space.

Keywords—mobile applications; Android; static analysis

I. INTRODUCTION

With the rise of mobile applications ("mobile apps") and the availability of app market places, the security and privacy risks associated with mobile apps increase accordingly. Not only mobile malware is growing, but also the risk for private and corporate data posed by badly written software. As a consequence, more and more security checks are developed [1] [2] [3] [4] [5] to analyze mobile apps. The focus is not solely on identifying malicious software, but also on spotting potential privacy breaches [6] and corporate data leaks [7] or indicating bad programming issues such as energy bugs [8]. In the last years, researchers did not only look at cherry picked applications from various market places, but also mass-analyzed apps ranging from a few hundred [1] over several thousand [2] up to over a million mobile apps [5].

Performing mobile app analysis in an automated manner is interesting for the research community as well as for the corporate world. However, setting up an environment for different analyses is a tedious task, and conducting mass security scans of apps is quite time-consuming.

In this paper, we present AndroLyze, a distributed framework to analyze large numbers of apps in an efficient manner. AndroLyze combines several key features in a novel way that distinguish it from other work on mobile app analysis:

• AndroLyze offers a utility library for standardized writing of static analysis scripts
• AndroLyze provides unified logging and reporting functionality backed by a database
• AndroLyze can handle large sets of mobile apps, including different versions of an app
• AndroLyze achieves efficiency through parallelization and distribution among CPU cores, CPUs and servers
• AndroLyze relies on optimized job scheduling to obtain faster analysis times

By having a standardized way of writing and deploying scripts as well as a properly defined output format, it is much easier to incorporate security checks into the development process or use the knowledge of various script sources to improve corporate security by enriching black-/white-lists of mobile device management solutions. To demonstrate the benefits of AndroLyze, we have analyzed the Top Free 500 Android apps of all categories in Google Play collected over three years. The whole data set consists of almost 40,000 apps requiring about 227 GB of storage space.

This paper is organized as follows. Section II discusses related work. The framework designed for managing analysis scripts, mobile apps and job scheduling is presented in Section III. Section IV describes implementation details. AndroLyze is evaluated in Section V. Section VI concludes the paper and outlines areas for future work.

II. RELATED WORK

Several approaches in the literature rely on Android app analysis, ranging from the detection of privacy leaks to misuse of cryptography and certificates as well as identification of malicious software. However, very few publications address the issue of mass-analysis of Android apps. Most approaches are "hand-made" and are only suitable for a particular use case, and they do not provide version and result management.

A common basis for manual and automatic app audits has been developed by Desnos et al. [9]. This framework written in Python is called Androguard and offers features such as disassembly, decompilation, control flow graphs and similarity search. The high number of features and the possibility to easily write scripts using them, combined with constant improvements over the years, added to Androguard’s popularity.

Viennot et al. [5] have conducted a study of Google Play and developed PlayDrone, a crawler of the Google Play store including a distributed analysis framework. Downloaded Application Package Files (APKs) are decompiled with dex2jar and JD-Core and stored in Git repositories located on worker nodes as well as on ElasticSearch. The latter offers full text search and an analysis engine for an app’s source code. AndroLyze differs from PlayDrone in the fact that it offers a native decompiler and uses MongoDB, a NoSQL database for result evaluation and storage. Furthermore, it achieves efficiency through parallelization and optimized job scheduling, and can handle different versions of an app.

DOI 10.1109/MS.2015.20
Fahl et al. [2] have conducted a mass audit of APKs using a script based on Androguard named MuliDroid to identify security issues in the use of transport layer encryption in apps. No special infrastructure used in the analysis is presented, and the system is specifically tailored to perform SSL/TLS analysis. Cryptography related analysis has been performed by Egele et al. [10] in their empirical study. Their static analysis tool called CryptoLint is also based on Androguard, and the number of APKs analyzed is comparable to the work by Fahl et al. [2]. A similar approach has been taken by Felt et al. [1] for their static analysis tool Stowaway to analyze app permissions. Using Dedexer to disassemble the APKs, the output is analyzed and the results are manually stored. Further research on permission based problems with Android applications has been done by Bartel et al. [11] by using static analysis to build call graphs, but focusing on the Android framework itself and not on the apps. Fend et al. [12] use static analysis of applications to detect malware. By performing inter-component control flow analysis and static taint analysis, APKs from Google Play are analyzed. Static analysis is based on SOOT\(^1\) for the task at hand. Kim et al. [6] have developed static tests to detect privacy leaks in Android applications. The main component is a modified decompiler and an intermediate language for their analyzer. Another approach to static analysis has been presented by Payet et al. [13] by extending the Julia static analyzer to work with the specific requirements and features of Android applications. Only tens of apps have been tested, and with a runtime of 7 minutes per app, speed can be an issue for mass audits. Static analysis is also part of the work done by Schmidt et al. [14]. For their collaborative malware detection function, calls made in the Android environment are extracted for automatic classification and comparison with malware. All of these papers focus on their specific tests and have their own implementation to handle large amounts of APKs. In contrast, AndroLyze provides a standardized way offering a single platform for all kinds of security checks regardless of the target and also with analysis speed in mind to execute them as fast as possible.

Grace et al. [4] have developed RiskRanker to detect apps exhibiting dangerous behavior. Apart from offering this particular type of analysis, the authors focus on execution speed to potentially handle large numbers of APKs. Nevertheless, RiskRanker does not provide a general purpose platform for other tests or a mixture of different analysis scripts.

SAAF developed by Hoffmann et al. [15] is a general framework for static Android app analysis. It can be used for manual and automated checks. While it aids developers in writing new security checks and provides fast execution times for single scripts, it does not provide support for effectively handling results and large numbers of APKs and scripts.

An approach combining different solutions for Android malware detection has been presented by Maggi et al. [3]. Similar to Google’s VirusTotal\(^2\), AndroTotal is a platform based on virtual Android devices running different malware detection software and giving users submitting samples feedback based on the results. The focus is solely on an infrastructure for dynamic analysis including user-interaction automation. Integration of different ready-to-run native anti-malware apps in the Android emulators is provided, but there is no support for third-party developers, and there is no scheduler optimized for fast results while conducting mass audits in place.

Andrubis developed by Weichselbaum et al. [16] is a platform combining dynamic and static analysis. Static analysis is used to aid and fine-tune dynamic analysis and providing additional information about an app, such as permissions, activities, broadcast and receivers. While Andrubis has been designed for broad tests written for dynamic analysis on larger sets of APKs, the requirements for this hybrid approach are quite different from our approach, and it lacks the unification and rapid writing of static analysis tasks AndroLyze provides.

III. ANDROLYZE

This section presents the design and analysis approach taken by AndroLyze, as illustrated in Figure 1. The bottom part of Figure 1 shows that AndroLyze has a direct link to Google Play to download apps. Moreover, it shows an analyst with an import database carrying information about the APKs to be analyzed. Importing apps into a local database enables certain filter and sorting capabilities, but AndroLyze can also be used without using a local database. AndroLyze already comes with a built-in set of scripts, such as extracting permissions from the manifest, disassembling or decompiling the APK. Additional analysis criteria require users to write custom scripts based on the functionality of Androguard [9].

The top part of Figure 1 illustrates the analysis approach. There are different modes: Depending on the number of applications, a user can start an analysis either locally (local mode) or on a cluster of nodes (distributed mode) to further improve performance. Both modes operate in a fully parallel manner, leveraging all available cores of the used processor(s).

The distributed mode shown in Figure 1 works as follows: Initially, the scripts are deployed via SSH on the available nodes. Tasks are sent to a distributed job queue and processed by a pool of worker nodes. The APKs are either stored in the jobs or predistributed and available in the APK cache. After a node has finished a job, it stores the outcomes in the result database. Finally, an analyst can view and evaluate his or her findings either using AndroLyze directly, sending custom queries to the database, or by syncing all results and performing a local analysis or review.

AndroLyze is designed to analyze large numbers of APKs in a short amount of time. More nodes can be added on the fly to further improve performance and reduce the analysis time.

In the following, we explain the design of AndroLyze in more detail, describing the script framework, the storage system and the parallel analysis engine. Finally, we guide the reader through a typical workflow using AndroLyze.

A. Script Framework

The heart of AndroLyze is the script framework. User defined analysis features can be implemented with Python 2 scripts leveraging the power of Androguard [9].

1) Script Requirements: Scripts in AndroLyze can be written for different purposes, such as collecting information from the Android manifest, accessing information gathered by analyzing the disassembly, checking where and which permissions are used in the code, creating the control flow graph or decompling the application. These tasks differ in their demands they pose on Androguard. AndroLyze takes advantage of this fact to improve performance by requiring script authors to signal their requirements to implement certain analyses. Therefore, AndroLyze can scale down the requirements of all scripts to

\(^1\)https://sable.github.io/soot/
\(^2\)https://www.virustotal.com

74
the lowest common denominator, resulting in the best analysis performance. The built-in scripts are written in a modular way, and functionality can be chained together on demand.

2) Logging: Logging the results of an analysis in a structured fashion is important for any evaluation study. The AndroLyze script framework comes with a set of functions to log common data types. Additionally, results can be summarized to groups of particular interest. Examining the manifest, for instance, might lead to a structure such as \( \text{manifest} \rightarrow \{ \text{permission}_1, \ldots, \text{permission}_n \} \).

Very large and/or binary data is supported by the logging component. It is directly connected to the storage architecture, thus all results are kept in the result database and optionally in the file system of the analyst.

AndroLyze does not enforce a static schema for results, but requires a script author to register a basic structure such that a result can be rendered to hold initial default values. This offers the possibility to visualize results in the absence of any logging and makes the comparison of app versions easier.

B. Storage & Analysis

AndroLyze provides an import database that carries meta-information about the APKs, a result database for storing the results and an APK cache for the .apk files. One of the main design goals of the import database is to manage different versions of an application, either from different sources (e.g. Amazon Market or Google Play) or different builds. This can be used for historic analysis, e.g., to see how long different bugs were present or to determine application trends on a broader scale, such as the adoption of technologies like HTML5-based frameworks or the use of encryption over the course of several month or years. To collect these different versions, AndroLyze allows analysts to update the database by downloading newer versions, if present, directly from Google Play. Thus, evaluations can not only be performed on the most recent APKs, but also on earlier versions several years back.

The analysis can be performed in two different modes, called local parallel and distributed parallel. The local parallel mode offers the possibility to use AndroLyze without the need to install a distributed system. The distributed parallel mode enables the system to further improve performance by adding more nodes on demand.

Both implement dynamic scheduling to utilize all available nodes, hence all cores/processors as long as possible. However, since the analysis time between jobs may differ strongly, long running jobs are scheduled first. Since most scripts analyze some piece of code, the size of the \( \text{classes.dex} \) file is used as a simple metric for a job’s runtime. We call this approach Code Size Scheduling (CSS). With both dynamic scheduling and CSS, parallel execution can be performed as long as possible.

The actual analysis process is as follows: A worker fetches a job from the task queue, performs an analysis and stores the outcomes in the result database. A job represents the analysis of a single APK with a given set of scripts. An APK is either part of the job or fetched from the APK cache if the applications have been imported beforehand. In addition to the analysis results, the system stores the identifiers of the analysis results in the job queue such that the analyst can view his or her findings directly after the analysis has finished.

C. Workflow

To illustrate our approach in action, we now guide the reader through a typical workflow while using AndroLyze.

Figure 2 summarizes the architecture and common steps required for an analysis. First, the analyst needs to obtain the APKs (s)he wants to inspect. The link to Google Play\(^3\) offers the possibility to download single applications via the package name or to download the newest or most famous APKs from either a particular category or all categories (Step 1). Importing the apps into a local database (Step 2) is not required, but enables an analyst to use our advanced scheduling and filtering functionality. The next step is to create a script (Step 3) suitable for AndroLyze, leveraging the power of Androguard in order to implement the needed analysis functionality. The more nodes we have, the faster the entire analysis is. Therefore, it is recommended to use the distributed parallel mode for larger sets of applications. It assumes that all nodes have been started and that the scripts have been deployed (Step 4). The deployment process can be handled through AndroLyze and requires an analyst to have SSH access on the nodes. After Steps 1-4, the actual analysis process can be triggered (Step 5). Initially, the APK meta-information is loaded, either from the .apk files supplied by the analyst or from the import database. For an efficient analysis (Step 6), the minimum requirements that satisfy all scripts are determined and sent together with information about the scripts and the applications to the distributed job queue. The worker nodes in the pool fetch one job after the other, perform the analysis

\(^3\)The listing of the most famous or newest applications in a certain category is limited by \( n = 500 \) \([5]\)
and store the outcomes in the result database. The results can be synchronized, concurrently to the analysis process, to the local hard disk drive of the initiator of an analysis. Finally, the findings can be further processed or evaluated.

IV. IMPLEMENTATION

In this section, we describe implementation details of AndroLyze, including the script framework, the storage and the analysis components.

A. Script Framework

There are two types of scripts, implemented as subclasses of either AndroScript or ChainedScript. The first provides the analysis functionality for scripts, the latter enables the chaining of scripts to bundle their functionality. Depending on the script requirements the analyst needs, the following analysis objects are provided:

- DalvikVMFormat: Creates a disassembly and provides access to classes, fields and methods.
- VMAnalysis: Analyzes the disassembly to check, for example, where which permissions are used or if reflection or dynamic code loading is used.
- GVMAnalysis: Creates the control flow graph.
- XREF: Detects cross-references between methods.
- DREF: Detects cross-references between data fields.

Through the definition of requirements, AndroLyze can query all scripts and determine the minimum script requirements to provide the necessary analysis functionality. Without defining any requirements, it is only possible to access the raw .apk file or its contents, such as the manifest file.

Every script has access to the logging framework, allowing unified logging in a structured fashion. AndroLyze supports logging of all JSON serializable data types, such as booleans, integers, strings, lists and dictionaries. Internally, a dictionary keeps all reported results, extended by static information about the analyzed APK as well as the script. Moreover, AndroLyze offers a custom storage interface for text files, graphs or other binary data. Statistics about the analysis time help to improve the script performance.

B. Storage

The storage implementation is divided into two parts: the import and the result database. Both are described below.

1) Import Database: Even if the analyst has many APKs, she might be interested in a subset or a single application only. To provide filters, AndroLyze needs information from the application’s manifest. The import mechanism extracts data such as the package name and version number from the manifest. Moreover, it stores the size of the classes.dex file as a metric for the script runtime used for our improved job scheduling. Instead of relying on Androguard to perform the import, we developed a faster solution that extracts only the manifest, thus preventing the extraction of the whole .zip file as Androguard does it. The import process is fully parallelized and all operations are executed in-memory after the APK has been loaded. A SHA 256 message digest\(^4\) over the whole APK serves as a unique key in the database and allows us to manage different versions of a single app. Therefore, AndroLyze can provide a security track record over many releases. We allow labeling a data set in order to provide a better distinction between different APK sets. Finally, the meta-information is stored in a local SQLite database so that different analysts can manage their databases independently of others.

2) Result Storage: In contrast to the local import database that is basically a flat file, we use MongoDB\(^5\), a schema-free NoSQL database for the distributed result storage. Therefore, all users of AndroLyze share the same result database. Nevertheless, different views on the database can be used to distinguish between data sets.

AndroLyze adds meta-information for an APK, such as package name, version name, build date and message digest, to the result. Additionally, information such as script name, hash, version number and the analysis date are added as script meta-information. This meta-information about the APK and script add fields to the result layout, allowing to formulate queries for the database without any knowledge of the actual results. We define a result as the outcome of the analysis of a single APK using a single script. New results with the same script name and APK hash replace old ones\(^6\). In addition to the result storage in the database, AndroLyze can store the results formatted as JSON strings on the local hard disk drive.

a) GridFS: To support the storage of files larger than the MongoDB limit of 16 MB, AndroLyze uses MongoDB’s GridFS that splits BSON documents into binary chunks. Two collections, one for the chunks, the other for the meta-information, are used to reassemble the whole file. The method is used for storing large results and APKs\(^7\). AndroLyze does not decide on the fly to use GridFS for large results due to the fact that storing results in a binary format allows analysts

---

\(^4\)In the following, message digest or hash always refer to the usage of the SHA 256 algorithm

\(^5\)https://www.mongodb.org

\(^6\)The keys for the results are generated by hashing the script name and the APK message digest

\(^7\)We integrated the GooglePlayCrawler into AndroLyze to access the Play Store and download the APKs
b) Key Escaping: Reserved characters such as "," and ";" need special treatment if they are used as keys in the result. This is important when using the package name in a query. AndroLyze escapes them with "\"" and "\;" respectively. Key escaping is only necessary if queries are sent directly to MongoDB, since AndroLyze automatically handles this while creating queries.

C. Analysis

Although the local parallel and distributed parallel mode share the same conceptual design, they differ in their implementation details. Both modes use processes instead of threads, because, due to the Global Interpreter Lock, only one Python thread can execute code at once in the interpreter. On each node, we spawn as many processes as the CPU has cores (including multithreading). Providing the Androguard analysis objects, such as the disassembly, implicates overhead that can be reduced by the definition of script requirements. This enables AndroLyze to use the minimum script requirements needed to perform an analysis. Due to the overhead involved, an APK is opened only once and all scripts are run afterwards.

1) Local Parallel Mode: The local parallel mode uses a synchronized queue to distribute the work among the processes. The queue memorizes the path to the APK and the meta-information. After a successful analysis, the results are stored in MongoDB.

2) Distributed Parallel Mode: The Celery framework is used to implement a message-oriented middleware in combination with RabbitMQ as a distributed task queue. Celery acts as a message broker and provides worker instances to execute the actual analysis jobs. Each process maintains a single Celery task instance that is kept alive as long as Celery is started on the node. This allows us to keep a persistent database connection for the result storage. To avoid idle times, a process reserves one task per process that is preloaded. Therefore, if the APK has been integrated into the message, it is already available when the process starts the analysis. The actual analysis process is as follows: Tasks are serialized with the Python Pickle module and sent to RabbitMQ. Celery workers retrieve a job and perform the analysis on the given APK with all scripts. Afterwards, the results are stored in MongoDB. The identifiers of each result are kept in RabbitMQ as a reserved queue only for the job results. The initiator of an analysis has a callback handler registered for this queue so that (s)he can synchronize the results concurrently to his or her local hard disk drive. All steps in the analysis process are fault tolerant. Thus, if a node fails, the job can be executed by any other node. The connection to RabbitMQ and MongoDB can be fully encrypted, using X.509 certificates.

V. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of AndroLyze. First, we demonstrate speed improvements by import parallelization and on-demand script requirements. Afterwards, we investigate the local parallel and distributed parallel modes, and the APK distribution strategies. Then, we illustrate the overall performance of AndroLyze by evaluating the analysis time of almost 40,000 applications (i.e., the Top Free 500 APKs from all categories collected in three years: 2012, 2014 and 2015). Finally, the results of a study on the use of cryptographic code in this set of APKs is presented.

A. Test Environment

In total, we used 7 computers in the experiments. Each has an Intel Core i7-4771 processor, 32 GB RAM, a 240 GB SSD, and two hard disk drives with 3 TB storage. They are connected via Gigabit Ethernet, and all except one computer host a KVM virtualized machine (VM) that has only 16 GB RAM and a 128 GB HDD (no RAID), except for the job queue which has 512 GB of additional space.

Table 1 shows the APK sets used in the experiments: the Top Free 4, 100 and 500 applications available from all categories in Google Play. Furthermore, we collected snapshots of the Top Free 500 applications of the years 2012, 2014, and 2015, and combined them in the set Top Free 500 Archive to demonstrate the versioning system and security track capabilities of AndroLyze.

The used script sets shown in Table II start with scripts that extract information from the manifest only. The next set adds partial SSL analysis capabilities from MalloDroid [2]. Bytecode has additional scripts listing the classes and fields using the disassembly as well as creating a method call graph. In addition to Bytecode, Source code performs decompilation with the help of Androguard’s DAD decompiler.

B. Import

The first experiment is performed on a single computer and impacts the performance improvements achieved by parallelizing the import and using our import code compared to the standard Androguard import.

Figure 3 shows three curves: The orange (middle) curve is the import using Androguard to open the APK and to extract the needed manifest information. The blue (bottom) curve points out the speed improvements over Androguard since only the manifest needs to be extracted from the zip file and all other processing of the APK can be omitted at this stage. The green (top) curve shows the influence of concurrently copying the APKs to the local hard disk drive.

<table>
<thead>
<tr>
<th>Table I. APK test sets</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set</strong></td>
</tr>
<tr>
<td>Top Free 4</td>
</tr>
<tr>
<td>Top Free 100</td>
</tr>
<tr>
<td>Top Free 500</td>
</tr>
<tr>
<td>Top Free 500 Archive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table II. Script test sets</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set</strong></td>
</tr>
<tr>
<td>Manifest</td>
</tr>
<tr>
<td>+SSL</td>
</tr>
<tr>
<td>+Bytecode</td>
</tr>
<tr>
<td>+Source code</td>
</tr>
</tbody>
</table>

9http://www.celeryproject.org
10The shown numbers do not take duplicate apps (same hash) into account. Duplicates can occur because an app may be present in more than one category.
For each run, we deleted the import database as well as the import directory for the APKs. The database is created on the SSD while the APKs are copied from and to the local hard disk drive. With only one process, the improved import mechanism is 2.03 times faster than Androguard. Even though the performance benefits decrease with more processes, there is still an improvement of 19.24% with 19 processes, which is the best value for the Androguard import. Moreover, copying the APKs clearly reduces the import speed because the hard disk drive is involved, but even here we have a performance increase of 98.87% with 18 processes compared to 1 process.

C. Scripts

The next experiment measures the impact of the analysis object provisioning time on the script runtime. For this purpose, we created a script for each available analysis object, signaling only the particular script requirement we want to examine. We left the method body of the procedure responsible for the actual analysis empty. Nevertheless, meta-information about the script and APK is still logged and stored in the file system as well as in MongoDB, because this is automatically done by Androlyze. The network is not involved in this test scenario, since we used only a single computer with a local MongoDB instance.

The results of this experiment shown in Figure 4 indicate that providing the analysis objects only on demand improves performance. Providing access to the manifest for all 102 applications of the APK set Top Free 4 is very fast. As soon as the first requirement is needed, the runtime gets approximately 100 seconds longer (DalvikVMFormat, VMAnalysis and GV-MAnalysis) because the disassembly etc. have to be created. Creating cross-references between methods (XREF) and fields (DREF) are the most time-consuming requirements.

D. Local Parallel Mode

To evaluate the performance of the local parallel mode, we used the APK set Top Free 4 and analyzed it with the script set Source code. The results in Figure 5 show that a speed-up of 3.19 is achieved using 4 processes instead of only 1 process, each of them assigned to a single CPU core of the used computer that is equipped with 4 CPU cores.

Figure 6 shows the distribution of analysis tasks among the CPU cores using two different scheduling strategies. The first scheduling strategy is based on the ascending order of the package names, while the second (CSS) uses the code size and schedules jobs with a large code sizes first. Each job is represented by a different color and its block size in the chart corresponds to its runtime. The figure shows that the analysis of a particular job took very long, which is caused by the decompilation script. With CSS, this long running job is scheduled earlier, resulting in a reduced analysis time (233.15 seconds). With improved scheduling, the speed-up of 3.19 shown in Figure 5 is improved to 3.59.

To validate the performance benefits of CSS, we carried out an experiment that uses the same script set as the experiment before, but does not use the decompilation script, thus eliminating the long running process. The results in Figure 7 show again that CSS exhibits a better performance. The speed-up is improved from 3.69 to 3.8.

E. Distributed Parallel Mode

To evaluate the distributed parallel mode, we compare the performance properties of both modes including CSS. Using 4 computers (3 of them with a KVM VM each), the distributed environment has been set up as follows: The result database is located on the computer without a VM, whereas the distributed message queue runs in a VM. The analyst initiates the analysis from the computer without a VM and integrates the APKs located on the hard disk drive into the messages sent to RabbitMQ. We take into account the serialization of the APK data as well as the network.

Figure 8 shows the analysis of the data set Top Free 4 (left) and the analysis with the set Top Free 100 (right). Both use the
The results for Top Free 4 indicate that AndroLyze is optimized for a large number of jobs. The speed-up between the local parallel and distributed parallel mode is only 2.17 and 2.41 with CSS. This result can be explained by the reservation of tasks. Each process reserves one task, hence in the worst case, a worker may be free for work, but all other workers have reserved the remaining tasks. Another problem is the granularity of the tasks, because in our scenario the last 31 jobs are executed by less than 32 processes (the total number of processes). The results for Top Free 100 show that a speed-up of 3.26 and 3.31 with CSS could be obtained. In both cases (Top Free 4 and Top Free 100), CSS improves the performance.

F. APK Distribution

The next experiment investigates the data throughput of the proposed APK distribution strategies. Figure 9 compares both APK distribution strategies (Send APK vs. Send APK-ID) in the distributed parallel mode with the local sequential and local parallel mode. The used test set Top Free 500 consists of 12,689 APKs with a total size of 91.8 GB to measure the throughput of serializing the APKs and the distribution through MongoDB. The script set Manifest for unzipping the APK and extracting the information, is used in this experiment. The result synchronization from MongoDB to the initiator of the analysis has been disabled in this experiment because it puts more load on the database and may influence its APK distribution. The results shown in Figure 9 indicate that the local parallel mode is faster than the distributed parallel mode, no matter which APK distribution strategy is used. Integrating the APK into the message that is sent to RabbitMQ allows preloading the message, because every process reserves one task. Prior importing of the APKs into MongoDB and only sending the identifier of the application forces the worker to fetch the APK from MongoDB before it can start the analysis. Moreover, the APK has to be built together from all of its chunks\textsuperscript{11}. The distribution via MongoDB is 17 minutes and 26 seconds slower than the distribution via RabbitMQ.

Nevertheless, the distributed parallel mode is valuable, since MongoDB’s sharding facilities can be used to distribute data records among several instances. Having enough MongoDB nodes can circumvent the bottleneck that can occur while serializing and sending apps from the analysis initiator over the network. Furthermore, the caching behavior of MongoDB allows MongoDB to be used as an in-memory database. In the experiment, we stopped MongoDB, removed all unused pages and started the service again to prevent this effect.

G. Overall Performance

We now present an overall performance evaluation of AndroLyze. Figure 10 shows the results of executing a single script on the data set Top Free 500 Archive on 7 computers, each running 8 Celery processes.

The first script (left) does not need any analysis objects. Thus, job execution is much faster than loading the APK data from the hard disk drive and transferring it over the network. Obviously, increased script requirements increase the runtime. ClassListing (i.e., list all classes) and ClassDetails (i.e., list all classes and additionally show fields and methods) do not need any cross-references because they only access the disassembled APKs. Creating cross-references, as needed by CodePermissions, WebView (i.e., list all methods using the WebView class), SSL, CryptoStats (see Section V-H) and MethodCallGraph, adds more than one hour to the analysis time. Most of the time is needed to decompile all apps with Androguard’s DAD decompiler, which consumes 9 hours and 24 minutes. Since there is an overhead for opening the APKs and providing the analysis objects, an analysis with all scripts where the objects are only provisioned once and shared, is 19 hours and 35 minutes faster compared to running all scripts alone.

\textsuperscript{11}AndroLyze sets the chunk size as large as possible, preventing more chunks than necessary.
H. AndroLyze®Work: Use of Cryptographic Code in Apps

To demonstrate the benefits of AndroLyze in a concrete analysis scenario, we have conducted a study on the use of cryptographic code in APKs in the data set Top Free 500 Archive. Our script checks for explicit calls to the block cipher modes ECB (Electronic Codebook) and CBC (Cipher Block Chaining) as well as for the most preferred ciphers, which in total took 3.29 hours to complete on 7 computers. The results are shown in Figure 11. They are in line with the results by Egele et al. [10], since many apps use cryptographic functions (65.65 %) somewhere in their code. Since we have snapshots of the Top Free 500 apps of all categories from different years, we have also analyzed which apps changed their cryptographic behavior, given they were still in Google’s Top Free 500 apps. Throughout the years, AES (Advanced Encryption Standard) has always been the most popular cipher, and our study shows that most developers still favor AES. Overall, many developers requested CBC mode (87.51 %) independent of the used cipher, and only a fraction asked explicitly for ECB mode (still 20.35 %). Some apps use both modes in different areas of code, either for backward compatibility, as a fallback, for file format compatibility or for no apparent reason. For apps relying on ECB mode, our results show that 134 apps removed the ECB mode in their latest version. It is evident that checks like these and the ones performed by Egele et al. [10] should not only be performed once, but should be incorporated in a unit test-like fashion into build or release cycles. Having a standardized infrastructure, script, APK and result management as offered by AndroLyze helps providing audit trails for app lifecycles and various checks developed in-house, for the research community or for commercial entities.

VI. CONCLUSION

In this paper, we have presented a distributed framework called AndroLyze to support researchers in the process of conducting mass-audits of Android apps. We have shown that intelligent scheduling and APK management leads to faster results with potentially more useful information by incorporating past versions of apps. AndroLyze has been evaluated using the Top 500 Free apps from all categories of Google Play collected over three years, consisting of almost 40,000 apps requiring about 227 GB of storage space.

There are several areas of future work. Currently, the analysis scripts are based on Androguard, but AndroLyze can be used to easily plug-in other script backends for custom checks. Furthermore, the scheduling algorithms and script management facilities could also be used to optimize and streamline app runtime analysis. Finally, potential applications for AndroLyze include open repositories for static checks to bring the knowledge from the research community to app developers and into the development process of companies.

ACKNOWLEDGMENTS

This work has been funded by the LOEWE initiative (Hessen, Germany) within the NICER project.

REFERENCES


