DroidSearch: A Tool for Scaling Android App Triage to Real-World App Stores

Siegfried Rasthofer[1], Steven Arzt[1], Max Kolhagen[1], Brian Pfretzschner[1],
Stephan Huber[2], Eric Bodden[1,2], Philipp Richter[3]
Center for Advanced Security Research Darmstadt (CASED)
[1] Technische Universität Darmstadt, Germany
[2] Fraunhofer SIT, Darmstadt, Germany
[3] Universität Kassel, Germany
{firstname.lastname}@cased.de

***Abstract*—The Android platform now features more than a million apps from thousands of developers. This abundance is convenient, as it caters to almost every need. But users and researchers also worry about the security and trustworthiness of these apps. While precise program-analysis tools are helpful in this context, unfortunately they do not scale to the large number of apps present in current app stores.**

In this work we thus present DROIDSEARCH, a search engine that aids a multi-staged analysis in which fast pre-filtering techniques allow security experts to quickly retrieve candidate applications that should be subjected to further automated and/or manual analysis. DROIDSEARCH is supported by DROIDBASE, a middleware and back-end database which associates apps with metadata and the results of lightweight analyses on bytecode and configuration files that DROIDBASE automatically manages and executes.

Experiments on more than 235,000 applications from six different application stores including Google Play reveal many interesting findings. For instance, DROIDSEARCH identifies 40 known malware applications in Google Play and detects over 35,000 applications that use both `http` and `https` connections for accessing the same resources, effectively rendering the `https` protection ineffective. It also reveals 11,995 applications providing access to potentially sensitive data through unprotected content providers.

***Keywords*—Android, App Stores, Database, Pre-Filtering, Scalability, Static Analysis**

I. INTRODUCTION

According to a recent market study, more than 500 million smartphones will be sold in 2014 [**creditsuisse**]. Android leads the market with a share of more than 80% [**androidMarketShare**]. Much of this popularity is due to the availability of applications for almost every need. Smartphones are at the center of lively app markets in which not only companies, but also individual developers can offer their products. In total, the app market is expected to reach a volume of about 35 billion dollars in 2015 [**marketSurvey**]. Current markets such as the Google Play Store contain more than one million applications and grow at fast speed. However, such rising app ecosystems raise a lot of questions. Security analysts wonder which applications contain known security vulnerabilities such as allowing data theft through unprotected content providers or which applications exhibit known malware patterns such as data theft, SMS fraud, or SMS message interception. Social scientists are interested in how (potentially malicious) behavior influences the app ratings and comments in the store. End users wonder which apps are safe to install and which ones may misuse their private data. Finally, app-market providers wonder how they can best prevent such apps from entering their market that are of low quality or are outright malicious.

Because of the sheer number of apps available in current eco systems, it is impossible to answer those questions through manual analysis. Even automated analyses have their limits: Highly precise analyses take several minutes per app [**flowdroid**], which is already too long to scale to full markets. A practical process for large-scale app evaluation thus has to be multi-staged: Lightweight pre-filters retrieve only those apps that can possibly exhibit the behavior in question. Applications that e.g., dont even have the permissions required to read certain types of data cannot possible leak them. Similarly, apps that e.g., don't have the permission to send SMS messages cannot conduct SMS fraud. Such apps can be ruled out quickly with a pre-filter that conducts a very shallow, but fast analysis. The pre-filter only leaves those applications as candidates for a more in-depth analysis for which it cannot directly decide that the behavior in question is impossible. These apps are then passed on to more precise (and probably costly) analysis tools or to a human export for manual inspection. Note that the pre-filter is an over-approximation: It will only rule out impossibilities and always pass on an application when in doubt.

To provide a means for such pre-filtering, this paper proposes DROIDSEARCH, a sophisticated app analysis and search engine for Android applications. DROIDSEARCH combines DROIDBASE, a database associating application metadata and analysis results, with a web-based and scriptable search frontend that allows analysts to pre-filter and assess apps with ease using expressive queries. The database is automatically maintained by crawlers and analyzers that do not require any human interaction. DROIDSEARCH thereby acts as a semantic pre-filter to the underlying application store without being store- or analysis-specific. All specific filtering knowledge is provided in the user's queries which makes the database applicable to even pre-filter for attacks yet to be discovered.

The instance of DROIDBASE used for this paper comprises more than 235,000 applications from six different application

stores, including over 210,000 applications from Google Play and almost 27,000 collected malware applications. With it, we identified 40 malware applications in the Google Play Store and over 35,000 applications that use both `http` and `https` connections for accessing the same resources, effectively rendering the `https` protection ineffective. We furthermore detected 11,995 applications that contain open content providers, allowing unrestricted access to potentially sensitive data. To summarize, this paper presents the following original contributions:

- the design and implementation of DROIDBASE, a middleware for apps, app metadata and analysis results,
- the design and implementation of DROIDSEARCH, an expressive frontend over DROIDBASE,
- and a number of interesting findings in a large sample set of apps from various stores, including reports on malicious applications, vulnerabilities in benign applications, and general market statistics.

The remainder of this paper is structured as follows: Section II explains the architecture of DROIDBASE and DROIDSEARCH. We then demonstrate how the tool can help in large-scale security analyses on Android app markets in Section III. Section IV discusses legal aspects. In Section V we present related work, and in Section VI we conclude.

II. ARCHITECTURE

Figure 1 shows the overall architecture of the system. An arbitrary number of crawlers constantly crawls the various app stores such as Google Play, AppChina, or ApkHiapk to find new or updated Android applications as explained in Section II-A. The analyzer which we explain in Section II-B then takes these applications and creates the analysis data that will be stored along with the application. DROIDBASE (see Section II-C) is the storage and coordination middleware at the center of the system. It not only provides a semantic interface to the underlying relational database, but also manages application analysis states (already analyzed or not, analyzer version, etc.) and overall statistics. DROIDSEARCH (see Section II-D) finally allows researchers to access the data stored in DROIDBASE and execute semantic queries either through a web application or through custom scripts.

A. Crawler

The crawlers constantly scan the various application stores for new or updated applications. There is one specific crawler implementation for each store. For the purpose of this paper, we used DROIDBASE to crawl apps from Google Play, FDroid, FreewareLovers, SlideMe, ApkHiapk, and AppChina. One can easily extend the tool with more crawlers for other stores or sophisticated crawling tools such as PlayDrone [**playdrone**].

Note that we download only free apps. Obtaining paid apps would require special contracts with app store providers and/or app developers to provide for a cost-efficient and legal solution. Downloading the apps is a requirement for analyzing them since the bytecode is otherwise not available.

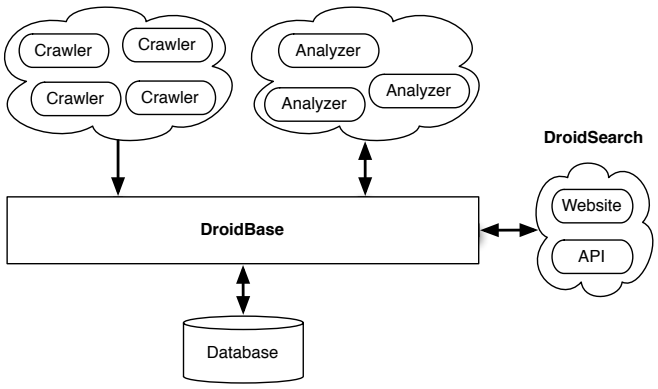

Fig. 1. Overall Architecture Overview

B. Analyzer

The analyzer produces semantic data for the Android applications obtained by the various crawlers. Since the goal of DROIDSEARCH and DROIDBASE is to act as a fast pre-filter, the analyzer does not need to conduct highly precise analyses. The data should rather be efficiently extractable, and should allow the triage of interesting apps which will then be passed to more sophisticated, longer-running analyses, for instance the FlowDroid [**flowdroid**] static data-flow tracker or the EPICC [**epicc**] inter-component communication analyzer, or to a human expert.

In our current setup, the analyzer extracts the following main types of information for DROIDBASE:

- App Store specific data (App description, number of downloads, etc.)
- User ratings and reviews
- All information in the `AndroidManifest.xml` file
- The certificate used to sign the application
- Libraries used in the application (both for functional purposes and for advertisement)
- The list of files contained in the application package
- URIs with which the application interacts
- All Android API calls made in the application
- Sensitive sources and sinks of private data called in the application
- Uses of reflective calls or native methods

The security-related information (e.g., API calls, native methods, URIs, etc.) provided by the app's bytecode are directly extracted from the `classes.dex` file. To retain effciency, DROIDBASE does not decompile the bytecode into an intermediate representation such as smali/baksmali [**baksmali**], but instead directly processes the binary *dex* file using a custom parser implementation.

C. DroidBase

DROIDBASE provides a middleware based on IBM DB2 for storing and accessing the analysis results for the Android

applications retrieved from the various stores. We chose DB2 as it supports XML searching capabilities using XPath (used for searching Android manifest files) as well as a full text search index (used for application descriptions and user comments) out of the box. Furthermore, DB2 can be extended with user-defined functions which simplifies the implementation.

In contrast to other approaches for large-scale app analysis that are based on ElasticSearch, our database cannot easily be distributed across multiple servers. We favored complex search features over distributivity since all metadata collected for our more than 235,00 apps fitted in a database of 34 GB that could be handled by a machine with three virtualized cores and 8 GB of memory. Note that the APK files need not be retained after the respective applications have been analyzed: Only the extracted metadata is required for running the DROIDSEARCH queries.

DROIDBASE also manages the analysis status of all applications. This status is either zero if the app has not been analyzed yet, or the version number of the analyzer that has processed the app. While the database is central and cannot be distributed in our approach, the status flag allows running multiple distributed analyzers on different machines that each process a certain fraction of the overall crawled app set. Even more, the version flag also allows incremental updates to the analyzers. If an app has a non-zero status code, only the analyses that have been added or changed between the respective analyzer versions need to be re-executed on the app.

D. DroidSearch

DROIDSEARCH is the search engine that runs on top of DROIDBASE. It allows security analysts to retrieve Android applications that fulfill certain properties such as declaring a specific set of permissions, calling a specific API method, or using reflective method calls. The search engine is provided both through a web application and an API that serves queries issued by custom scripts. The latter allows for extendability and is intended to encourage integration into other research projects.

With the query builder shown in Figure 2, users can create so-called *matchers*. A matcher is a set of field-operator-value triples which are implicitly connected by a logical *and*. One matcher can for instance require the permission list to contain `android.permission.ACCESS_FINE_LOCATION` and the app name to contain the word "weather". The operators used in matchers include equality, arithmetic comparison, substring matching, and XPath matching for the Android manifest file.

Matchers can freely be combined using the *and* and *or* logical operators to form more complex queries such as those used to produce the findings on which we report in the following sections. DROIDSEARCH converts these queries to SQL `select` statements which are passed to DROIDBASE and executed on the underlying DB2 relational database. Figure 3 shows a query combined from three matchers.

Executing the SQL qureries generated by DROIDSEARCH is usually very fast and takes less than a minute. This makes DROIDSEARCH an ideal pre-filter. With most analyses, processing even a handful of spurious applications that cannot exhibit a certain behavior in question takes longer than having these applications filtered out by DROIDSEARCH before starting the actual analysis.

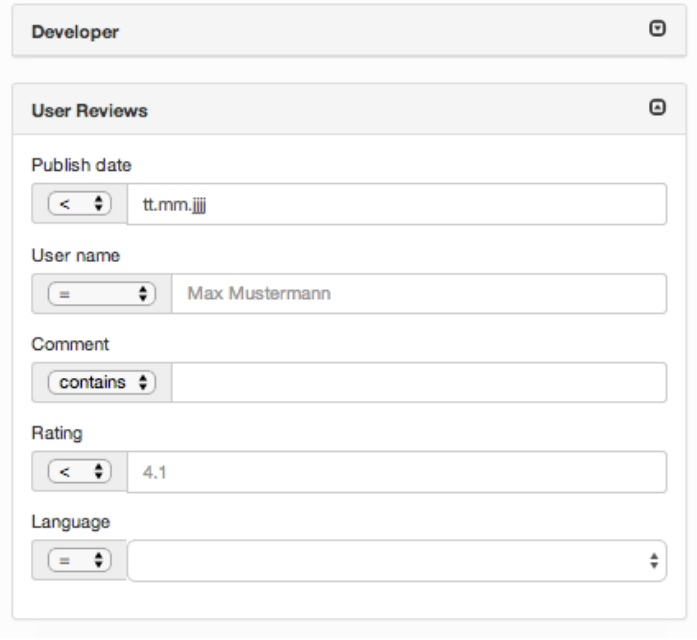

Fig. 2. DROIDSEARCH Filter Excerpt

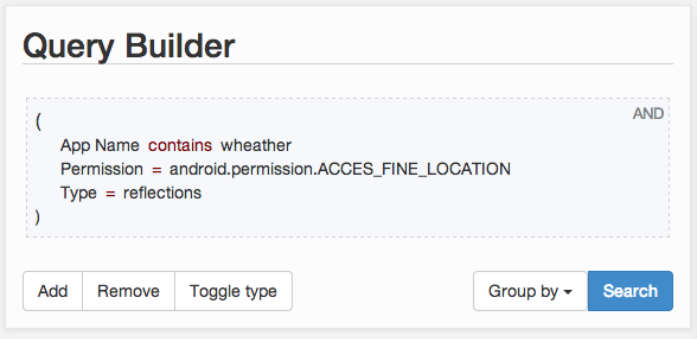

Fig. 3. DROIDSEARCH Query Builder

III. THE USAGE OF DROIDSEARCH

The following section describes how DROIDSEARCH can be used to pre-filter Android application markets for potentially vulnerable apps (Section III-A), and candidates of malicious behavior (Section III-B). We also give security-related statistics on the Google Play Store (Section III-C). For all our findings, we describe which DROIDSEARCH queries were used to either directly identify these problems or pre-filter the applications for further analysis.

A. Application Vulnerabilities

In this section, we show how DROIDSEARCH can be used to identify applications that are potentially vulnerable to certain types of attacks.

1) SSL Certificate Validation: When an application communicates with a remote server via SSL, the identity of this server needs to be verified. The Android operating system provides a default verification implementation for its `WebView` component, but developers can overwrite certificate validation and error handling. The latter is often used during development to allow the use of arbitrary self-signed certificates until a final certificate has been obtained. In some cases, developers forget to re-enable SSL validation before the app is deployed to the market which allows for man-in-the-middle attacks [**eve**].

DROIDSEARCH is used to pre-filter applications that have the permission to access the Internet (XPath query on the manifest) and which overwrite the `onReceivedSslError` method of the `WebViewclient` class. 59,269 (28.3%) of the 209,507 applications we crawled from the Google Play Store are potentially vulnerable. Appicaptor [**appicaptor**], a commercial Android security testing framework, confirmed a `Webview`-based SSL vulnerability in 11,802 apps, which is 19.9% of our prefiltered set and 5.6% of the overall Play Store sample.

Faulty implementations of the TrustManager interface are also prominent in Android apps. Using DROIDSEARCH, we identified 49,339 apps with custom implementations. Appicaptor confirmed 32,600 apps with vulnerable TrustManagers, which is 55% of the prefiltered apps and 15% of all apps we crawled from the Play Store. Note that these `TrustManager`s might only be enabled in debug configurations in some cases. Determing whether this flaw can actually be exploited in these applications is beyond the scope of this paper.

In both cases, the pre-filtering using DROIDSEARCH significantly reduced the number of applications that had to be processed by Appicaptor.

2) Mixed http / https Access: Many applications access sensitive data stored in the cloud or on external servers using the `https` protocol for security. However, if the developer is not careful and the webserver does not refuse to accept unsecured `http` connections as well, mixed access patterns might occur: Data is sometimes accessed via `https`, but sometimes also using plain `http` where it is transmitted in plain and easily accessible.

Using DROIDSEARCH's URI functionality, we queried the database for apps that contain URL patterns matching both `http://<url>` and `https://<url>` with `<url>` being the same string in both cases. This filter matched 35,143 applications in the Google Play Store which is 16.6% of all applications that we crawled. We found 3,041 different apps that use Twitter's implementation of the OAuth authentication protocol both through `http` and `https`. A further, more precise analysis is not even necessary in this case as all such mixed access patterns can possibly be exploited.

3) Static Passwords: When connecting to some FTP servers or to web servers, credentials must be supplied. These credentials are often included in the bytecode as constant strings. Attackers can simply decompile the app and retrieve these credentials to access the respective server on their own.

DROIDSEARCH can evaluate regular expressions on constant strings extracted from the applications. With this feature, we uncovered 4 unique credentials in 4 Play Store apps which were sent over unencrypted `http` connections, including one user name *admin*. The respective web site, however, was no longer available for confirmation. For one website, the respective URL returned an HTTP-404 (not found) error. One URL still contained the XML file addressed by the URL and required the credentials we obtained from the app. The URL on which the fourth credential was used was also accessible without authentication.

4) Android Manifest Settings: The `AndroidManifest.xml` files defines basic settings for an Android application. DROIDBASE stores the complete contents of this file in the database. This allows DROIDSEARCH to evaluate XQuery expressions against the collected manifests which can also be combined with arbitrary other analysis data in DROIDBASE to form complex queries.

The manifest file defines, among other settings, whether the operating system shall allow a debugger to be attached to the app. While this feature is useful during development, it can be abused in a productive scenario to obtain sensitive data processed by the app. In our crawled subset of the Google Play Store, we found 12,206 applications which allow debugger attachment. This is 5.8% of the overall sample for this store. 27,720 apps (13,1%) explicitly disable it. All other applications do not explicitly set the debugger flag and rely on the system default which is `false` for non-debug builds.

Applications can register broadcast receivers in their `AndroidManifest.xml` file. If the respective callback is sent by either the operating system or a different application, it is sequentially dispatched to all receivers registered for the respective intent. The dispatch order is controlled by the `priority` values of the registered listeners. Applications can abuse this to be the first to receive an intent and then suppress it. Using DROIDSEARCH, we found 5,357 applications in the Google Play Store that register broadcast receivers with a priority greater than 10 which is 2.5% of our sample. A manual analysis shows, though, that most uses are benign.

5) Content Provider Leakage: Data stored on the smartphone such as contacts, e-mails, and accounts is usually managed through so-called *Content Providers* which are defined in the `AndroidManifest.xml` file of the managing application. If a provider is marked as exported and no required permission is defined for it, the stored information is freely accessible to malicious applications installed on the same phone [**chex**]. Data leakage is one of the most common malware patterns [**survey**].

Open content providers can be found using DROIDSEARCH's XPath feature by searching for content provider with the `exported` flag, but no definition of `authorities` or `readPermission`. 2,861 apps in the Google Play Store (1.4%) contain a content provider which is explicitly marked as *exported*. 7,358 applications (3.5%) contain a content provider explicitly marked as non-exported. 9,698 apps (4.6%) contain content providers for which the `exported` attribute is not specified which makes them exported on Android API versions 16 and lower. Though the behavior was changed after Android API version 16, applications configured for earlier versions retain the old default for backwards compatibility.

Note that one application can specify multiple content

providers with different settings, so there can be an overlap in the numbers reported above. In total, 12,774 apps (6.0%) apps contain externally-accessible content providers. 11,995 apps (5.7% of all apps or 93.9% of all apps with exported content providers) do not require any permissions for apps requesting data from their content providers.

In some cases it was intended by the developer to provide application data to other apps without requiring permissions. Other apps might implement custom security checks in their content providers without relying on the mechanisms provided by the operating system. We randomly picked 100 candidate applications returned by DROIDSEARCH and subjected them to a more precise analysis. We ran drozer, a security audit and attack framework, on these apps and were able to confirm security vulnerabilities in 15 of these 100 apps. We were even able to generate working exploits from the URIs extracted from these apps. The confirmed vulnerable apps included calendars, file explorers, games, and even financial apps.

6) Applications Using Root Access: The Android operating system does not allow applications to be run as root by default. This can be bypassed to enable benign uses such as full-system backups, but a root account can also be exploited for malicious purposes. Once an application has root access, it can no longer be isolated from other applications and the operating system. Therefore, "rooting" a device can easily introduce grave security issues for the device as a whole. One would therefore only expect a small number of applications requesting root access.

DROIDBASE stores all constant strings used in an application. We used DROIDSEARCH to search for calls to `su` executables such as `/system/xbin/su` installed by common rooting tools. DROIDSEARCH identified 16,444 of our analyzed 211,500 applications in the Google Play Store (about 7.8%) require or request root access to the device which violates the normal Android security model. A manual inspection over some of these applications identified apps with root access that for instance synchronize pictures with Facebook. From a first inspection of these apps, it was not clear why these apps should need to be run as root.

B. Malware

In this section, we present current Android threats and show how applications potentially containing such malicious behavior can be pre-filtered by DROIDSEARCH.

1) Known Malware in Public Stores: For each application, DROIDBASE stores a SHA-256 hash of the respective `apk` file. Using DROIDSEARCH, we compared the hashes of the applications we downloaded from the Google Play Store with the hashes of already-identified malware applications for Android taken from VirusShare [**virusshare**], the Malware Genome Project [**genome**], VirusTotal [**virustotal**], DroidAnalytics [**droidanalytics**], McAfee, and GData. DROIDSEARCH found an intersection of 1,994 APKs which were contained both in our malware sample set and in our sample crawled from the Google Play Store.

The majority of these known malware apps contain adware. It is well known from existing work [**androidmalware**] that advertisement frameworks within applications collect personal user data in order to build profiles for targeted advertising. To distinguish outright malicious applications from such "grayware", we subjected the 1,994 apps returned by DROIDSEARCH to a second filter that takes concrete virus scanner classifications into account. This post-processing yielded 40 well-known truly malicious applications (not adware or greyware) which we verified by hand. Some of them were still in the Google Play Store at the time of writing. We reported these findings to the Android security team.

Apparently, some Play Store accounts are dedicated to malware distribution. DROIDBASE stores the certificate along with every application. We used an XPath query on this data to list the certificate issuer names for our set of known malware applications. About 400 samples of our original intersection of 1,994 apps were signed with a key generated for a company called "Moskow Droid Development". 500 malware applications were signed by "Mobile Sevices 500". The developer "912-Studios" that created the *Easy Button FREE* spyware produced 21 applications in total, all of which are very similar spyware apps, as if built from a template. 5 of those apps are still in the Play Store. We found these relations by searching for the names of known malware authors in the certificates of the store apps using DROIDSEARCH. We propose blocking store accounts if a certain number of malware applications has been uploaded from them. Since creating an account requires a payment in Google Play, it is not easy for malware developers to switch to fake accounts as every blocked account costs money. New accounts must furthermore be paid with a different credit card once the old account (including the credit card used to pay for it) got blocked.

2) SMS Message Interception: Android applications with the `RECEIVE_SMS` permission can intercept incoming SMS messages. Banking trojans use this method to access mobile transaction numbers (mTANS) which they can then use for fraudulent transactions. Using DROIDSEARCH, we identified 6,201 apps requesting this permission in the Google Play Store (2.9% of the sample). 2,379 apps register a broadcast receiver for `android.provider.Telephony.SMS_RECEIVED` intents in their manifest file as we found using DROIDSEARCH's XPath query function on manfiest files. The other applications that bear the permission, but do not register a broadcast receiver might register the respective receivers at runtime or might simply be over-privileged and not actually use this permission. 1,032 apps are especially interesting as they not only register a broadcast for incomig SMS messages and have the necessary permission but also assign an unusually high priority (> 10) to this receiver. 18 of these apps call the `BroadcastReceiver.abortBroadcast` method that stops other apps from receiving the SMS message. DROIDSEARCH makes it possible to only analyze those 18 applications in detail while discarding others that are not as likely to be potentially malicious. These 18 apps however seem benign from a first manual inspection.

3) User Reviews: Many app stores allow users to rate applications they have previously downloaded and installed on their devices. This information shall help other users decide whether an application suits their requirements, is stable, and trustworthy. DROIDBASE stores these user reviews and ratings for applications from the Google Play Store together with

the respective application which we used to look for reviews containing the words "malware", "malicious" or "virus". Most ratings were related to users pointing that their anti-virus application classified the application as malware. Manual analysis showed that all of these applications contained advertisement libraries which were identified as `Adware` by the analysis tools.

Only a minority of our sample (2%) was commented with texts such as *"Virus spyware. Do not download."*, *"Virus If installed"* or *"It has a virus he or she can read your phone number and do stuff without you knowing"*. Such concrete warnings by users should give the app store operator hints on which applications should be more thoroughly checked for malicious behavior.

4) App Descriptions: Many application developers provide short summaries for their applications in which they describe functionality, supported devices, and other useful information. For the user, the longer and the more comprehensive the description is, the clearer is the impression she gets of the application.

Since many malware detection approaches presented in literature [**chabada**, **whyper**] rely on the app description, we evaluated how meaningful and reliable these descriptions actually are. Using DROIDSEARCH, we counted the number of words used in real-world app descriptions. On average, an app description in the samples we crawled from the Google Play Store uses 117 words. 4% of all descriptions contain less than 5 words, 14% contain less than 20 words, 38% contain between 50 and 150 words and only 26% contain more than 150 words. This shows that the number of words used in descriptions is mostly very low: One thus needs to question whether these descriptions are really meaningful to both users and malware detection approaches.

5) Stealing Personal Data From Storage: Some applications store sensitive user data on the phone's SD card. All applications with the `READ_EXTERNAL_STORAGE` permission can read from this storage. As there is no further protection, this implicit sharing can lead to data disclosure vulnerabilities such as in the well-known WhatsApp [**whatsapp**] messaging application. It stores all sent and received messages as well as personal data such as photos or videos on the SD card. Previous versions of the application did not protect the data in any way. The latest version of WhatsApp encrypts the data, but uses a hard-coded symmetric key which has become known by now. Therefore, the WhatsApp data stored on the SD card is again readable to all applications which are allowed access to the SD card [**whatsapphack**].

DROIDSEARCH is able to search for URLs and file names accessed in applications which we used to identify 4 apps in the Google Play Store that attempt to access the WhatsApp database. One of these applications has already been removed from the market, two applications claim to access WhatsApp data for benign reasons, and one application removed the access to WhatsApp data in its latest version. The two applications that still access the WhatsApp data should be analyzed by more precise tools or human security analysts which is however out of the scope of this paper.

6) Suspicious Package Names: Repackaging exisiting applications is a well-known method for spreading malware in application markets [**piggybacking2**]. Attackers take a popular benign application, add malware to it and re-upload it to the store where it then coexists with the original benign application. As stores might check whether a package name has been used by a different application yet, some malware authors use different packages than the original application, sometimes even randomly generated ones such as `Ienee9chi.ceebah0Se` for a clone of *ICQ Mobile*. As a countermeasure, market operators could regularly scan their markets for applications sharing the same name, but having different package names, and then block all but the real, benign package name. Together with a block of re-using the same package name for multiple applications, this makes it harder to convincingly advertise a repackaged application in the store. The application name is stored in the manifest file, which can be queried by DROIDSEARCH.

7) Attempted Use of System Permissions: Some permissions on the Android operating system are only available to applications signed with a manufacturer key. Nevertheless, some applications include requests to such permissions in their manifest file. Table I contains examples in which user-level applications attempt to request such system permissions, including the `brick` permission which can be used to disable a phone or the `FACTORY_TEST` permission which allows code to be run under the identity of the `root` user. Such applications are candidates for potential malicious behavior and should be subjected to further analyses.

TABLE I. REQUESTS OF SYSTEM PERMISSIONS

Permission	# of apps
READ_LOGS	5,577
DISABLE_KEYGUARD	4,251
MOUNT_UNMOUNT_FILESYSTEMS	3,912
INSTALL_PACKAGES	1,232
CALL_PRIVILEGED	795
DELETE_PACKAGES	709
WRITE_SECURE_SETTINGS	620
CONTROL_LOCATION_UPDATES	405
DEVICE_POWER	591
HARDWARE_TEST	209
MOUNT_FORMAT_FILESYSTEMS	222
REBOOT	177
DIAGNOSTIC	26
MASTER_CLEAR	18
FACTORY_TEST	17
BRICK	11

C. Market Statistics

In this section, we provide statistical data on applications available in the Google Play Store. While this data is not directly related to security vulnerabilities or malware, it gives important hints as to which analyses are important to consider in real-world store operations and academic research to cover widely used features of the Android platform.

1) File Extensions: DROIDBASE stores the name, extension, and mime type of all files contained in an APK file. Table II shows the top 15 file extensions used in apps in the Google Play Store in an overview. The `so` extension for native code libraries is for instance the 11th-most prevalent extension

which shows that native code analysis is an important target for security research. 718 apps contained 881 other `apk` files. Such application nesting is usually used to hide functionality from analysis tools as only the code contained in the outer APK code is usually analyzed. At runtime, this code then loads and executes the inner one. Such nesting is an important challenge for analysis tools.

TABLE II. FILE EXTENSIONS IN APPS

Rank	Extension	# of files	# of apps
1	png	26,346,358	208,994
2	xml	9,711,276	209,510
3	jpg	1,928,071	75,031
4	*no extension*	1,405,589	28,822
5	html	632,446	46,428
6	mp3	537,901	21,838
7	js	340,348	32,164
8	txt	294,880	37,440
9	gif	279,363	31,124
10	ogg	190,553	13,197
11	so	134,471	30,601
12	properties	123,204	34,764
13	css	107,226	24,735
14	ttf	81,486	23,099
15	json	77,975	11,562

DROIDSEARCH can also be used to detect mismatches between the file extension and the actual mime type of the respective file. This can, for instance, be image files that are not actually images, but ELF binaries containing native code with a root exploit. In the samples we crawled from Google Play Store, we found 635 files that actually contained native code, but did not carry the `so` extension. 461 files had empty file extensions, the remaining 174 ones had misleading ones such as `mp3`, `png`, or `jet`. We found these files by querying DROIDBASE for mismatches between the proclaimed file extension and the result of querying a lookup table of file extensions for the respective actual mime types which were identified using the Apache Tika library.

2) Language Features: DROIDSEARCH supports querying the API methods used by an application. We found that 94,686 apps use classloaders to dynamically load code which is 44.76% of our sample from the Google Play Store. For static program analyses this poses the challenge of not all code being directly available for analysis. Only few apps however directly contain secondary `dex` or `apk` files. Therefore, one must conclude that the other apps either generate the code they load dynamically at runtime or download it from an external location. In either case, making such code available for static analysis or app vetting processes in a general way is a complicated challenge.

Using a similar search for API methods, we found that 166,434 apps (78.67%) use reflection to call methods. Many static analysis approaches do not support such calls at all or only if the target method and class names are constant strings. With reflective calls being so prevalent, this design decision must be questioned.

30,601 apps (14.5%) use native code. These numbers are in line with existing work [**playdrone**] which reported 14% of apps to include native code, excluding the few big and popular applications with 50 million downloads or more which contained a larger amount of native code.

3) Permissions: In this section we compare the ten most frequently used permissions in malicious apps and in benign applications from Google Play, FDroid, FreewareLovers, SlideMe, ApkHiapk, and AppChina. A similar comparision was already done by Hoffmann et al. [**saaf**]. We used a larger sample set than previous work for this evaluation. We took 216,464 benign applications from various stores and 26,865 malware applications in total in comparsion to 136,603 benign applications from Google Play only and 6,187 malware applications used by Hoffmann et. al. We therefore obtain slightly different results than existing work as shown in Table III.

As Table III shows, the INTERNET permission is used most frequently. Out of the 211,561 apps we crawled from the Google Play Store alone, 178,545 apps (84.4%) requested the `INTERNET` permission. This is in line with Google's update to the Play Store [**playstoreUpdate**] after which the store no longer lists the Internet access permission on the primary screen since it is requested by almost all applications and thus does not provide much information to the user.

TABLE III. TOP 10 PERMISSIONS

	Market	Malware
1	INTERNET	INTERNET
2	ACCESS_NETWORK_STATE	READ_PHONE_STATE
3	WRITE_EXTERNAL_STORAGE	WRITE_EXTERNAL_STORAGE
4	READ_PHONE_STATE	ACCESS_NETWORK_STATE
5	ACCESS_FINE_LOCATION	ACCESS_WIFI_STATE
6	ACCESS_COARSE_LOCATION	WAKE_LOCK
7	WAKE_LOCK	RECEIVE_BOOT_COMPLETED
8	VIBRATE	SEND_SMS
9	ACCESS_WIFI_STATE	ACCESS_COARSE_LOCATION
10	RECEIVE_BOOT_COMPLETED	VIBRATE

4) Apps Sharing Same Package Name: On the Android operating system, the package name uniquely identifies an application. No two applications with the same package name may be installed on the same phone. Interestingly, the malware samples from VirusShare, GData, and McAfee contain a large number of shared package names such as `com.software.application` (more than 1,600 times), `com.soft.android.appinstaller` (more than 500 times), or `com.software.marketapp` (more than 100 times). These applications are neither updates nor repackaged applications, but are advertised using very different names in the store. Some `com.soft.android.appinstaller` applications are called *Skype*, others *Mobile Internet Browser 3.7*. We found these applications by selecting the counts of package names grouped by package name in DROIDBASE.

5) Typing Mistakes in Permission Names: Android applications must a priori declare all permissions they need in their manifest file. Beside the system-provided permissions for accessing sensitive resources such as the location data or the camera, applications are also free to declare own permissions to protect own resources and request permissions defined by other apps. If an application therefore attempts to request a system permission, but the permission name is misspelled (e.g., `andriod.` instead of `android.`), this will be considered as a request for a custom permission defined by some (potentially not yet installed) other application and the operating system will not fail the app installation. The originally intended system permission is of course not granted and one would expect the

application to fail with a security exception at runtime since the permission was requested for a purpose. Even more, such issues should be detected during testing before releasing the app to the market. Nevertheless, DROIDSEARCH found 756 applications with misspelled permission names in the Google Play Store which is 0.34% of all apps we downloaded from the store.

IV. LEGAL ASPECTS

DROIDBASE and DROIDSEARCH are intended for the scientific community and potential end-users who wish to inform themselves about the privacy implications of certain Android applications. From a legal perspective, the described approach might on one hand strengthen malware detection by professionals as well as user privacy, informational self-determination and data security by educating and enabling users to make informed choices, which apps they deem secure and appropriate for themselves. On the other hand, legal issues may arise with respect to copyright law and legal provisions concerning trade secrets. A conflict between these two legal objectives, data security and user privacy on the one side and intellectual property rights on the other side, is slowly emerging as an important issue in the field of informational self-protection [**dudpaper**].

All of the technical steps described are legal if store owners and app developers agree and see scientific analysis of apps as a normal part of product testing by third parties, as widely applied by anti-malware developers or consumer protection organizations. Some app stores such as FDroid even provide the source code of their apps. Legal problems may only arise where right holders do not consent with such actions.

App stores are online product catalogues, and as such might be protected data banks. A sui generis protection for data banks has been established in the EU, but not in the USA and would thus be applicable only to European app stores [**dreierschulze**]. Even if app stores are protected as data banks, (automatic) searching is not prohibited, in contrast to reproducing, distributing or publicly displaying them without the right holder's consent. Searching app stores with crawlers instead of by clicking through them manually might, however, violate the concrete store's terms of use. The terms and conditions of the Google Play Store for instance restrict use of the service to the interface provided by Google.

Downloading an app from the store means to reproduce it which is generally allowed only with the consent of the copyright holder. Since only cost-free apps are downloaded by our crawlers, one might assume anybody to be permitted to download the apps in question. Even if not, the download should generally not collide with copyright law, as long as it is done for strictly scientific reasons [**bundestag**] [**dreierschulze**].

Analyzing the functionality of the apps is carried out without decompiling them as information is gathered directly from the bytecode. In general, analyzing apps and publishing information for scientific reasons and for public education on malware and privacy threats should be no legal problem.

Legal details need to be analyzed in future research and may differ from case to case, but generally speaking, comparable approaches to malware detection are widely spread and do not generally produce legal actions.

V. RELATED WORK

While DROIDBASE and DROIDSEARCH provide a layer around conventional application stores, certifying application stores [**cassandra**] or store-based validation techniques [**bouncer**] aim at not allowing applications violating the store's policies into the store in the first place. This concept is however limited to policies known to the store designer and fails if researchers or end-user define new policies or refine existing ones when the apps are already in the store. Our approach on the other hand can find candidates that might violate the (new) policy and subject them to further analyses without having to re-certify or re-check the entire store.

Existing analysis tools such as FlowDroid [**flowdroid**] or CHEX [**chex**] greatly benefit from pre-filtering target applications using DROIDSEARCH as without pre-filtering, these tools usually do not scale to size of full markets. Analyzing 24,000 applications is reported to take approximetly 30 hours in AndroidLeaks [**androidLeaks**] while only finding potential leaks of private data in 7,414 apps. A pre-filter such as DROIDSEARCH can help rule out apps that obviously cannot leak data since they e.g., miss the required permissions.

For some dynamic analysis tools such as AppsPlayground [**appsPlayground**], large-scale analyses have been conducted, but no performance statistics have been reported. From our experience, we can assume that installing and test-driving thousands of apps is a time-consuming undertaking that could also be improved by applying a pre-filter such as DROIDSEARCH.

AndRadar [**radar**] is an Android malware tracking and monitoring system which focuses on alternative Android markets. It tracks the distribution of malicious applications in real-time. The system identifies malware by comparing applications to known malware seeds using the package name, certificate fingerprint, apk file hash, and method signatures. This functionality can be simulated by DROIDSEARCH as all required matchers are available in the DROIDSEARCH query builder. Executing these queries in real-time whenever new apps are crawled can easily be added to our system.

PlayDrone [**playdrone**] is a tool for conducting large-scale analyses on the Google Play Store. PlayDrone provides statistics on aspects such as the prevalence of native code in apps, the relation between app ranking and download count, or the most common libraries. It also finds secret keys in many productive applications. The system is however directly tailored towards specific analyses while DROIDBASE and DROIDSEARCH are inteded as a generic store-agnostic pre-filter for arbitrary static and dynamic analyses.

VI. CONCLUSIONS

In this paper, we presented DROIDSEARCH, an approach for making available to researchers semantic metadata on Android applications obtained from various application stores. Using DROIDSEARCH, we crawled more than 235,000 apps from Google Play, and almost 27,000 known malicious apps from various sources. Using various examples, we showed how DROIDSEARCH and its backend DROIDBASE can be used to pre-filter applications that may exhibit a certain malicious behavior or suffer from known security vulnerabilities. These

findings can then be subjected to an additional, more precise analysis if required. DROIDSEARCH thereby reduces the workload for the consecutive processing step, be it a more precise automated analysis tool or a human analyst by reducing the number of applications that still need to be examined.

We found 40 known malicious applications in Google Play, and found that more than 35,000 apps use a mixture of `http` and `https` connections for the same resource, effectively rendering the `https` protection useless. We furthermore found a number of username and password combinations hard coded into apps, as well as 11,995 applications providing access to possibly sensitive data through unprotected content providers.

Acknowledgements: We would like to thank our students Brian Pfretzschner, Max Kolhagen, Benedikt Hiemenz, Patrick Lowin and Huong Luu Thu for supporting us with DROIDSEARCH as well as DROIDBASE. We also would like to thank Jan Peter Stotz for supporting us with Appicaptor. This work was supported by the BMBF within EC SPRIDE and by the Hessian LOEWE excellence initiative within CASED.