Analysing Android Spam Call Applications: Developing a Methodology For Dynamic Analysis

Atanas Pashov
Supervisor(s): Apostolis Zarras, Yury Zhauniarovich
EEMCS, Delft University of Technology, The Netherlands
22-6-2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

The aim of this research is to provide a structured approach for dynamically analysing Android applications, focusing on applications that block or flag suspected spam caller IDs. This paper discusses ways to determine how an application stores the data regarding the phone numbers, and what APIs it utilizes. In order to develop a methodology of what to search for while performing dynamic analysis, the research in this paper focuses on analysing one such Android application, using it as a template. Additionally, it provides insights and limitations on this analysed application, and potential improvements both on the techniques and tools used.

1 Introduction

Many bad actors have developed malicious tactics and methods to abuse the telephony infrastructure and disturb the mobile phone users due to the popularity of mobile telephony. Because of this many Android applications that detect and/or prevent spam calls have been created [1], however, there is little information about how they work or where they fetch their information from. This research paper will focus on analysing such applications and developing a methodology to try and get more information about how they work, and where they get their data from.

The already existing research on the topic focuses on building a database of potentially spam or scam callers. Four different ways to obtain such phone numbers are described in [9].

Other research focuses on the security aspects of the Android operating system, and how it is (theoretically) protecting its users. Section 7.10 in [12] describes the spam filters that Android utilizes and some notable third-party applications, but does not go in depth into how they work, what permissions/API calls they require, or where the data is taken from and stored.

Another way to detect spam calls is by utilizing algorithms. An algorithm that detects spam phone numbers based on call patterns is discussed in [8]. [11] provides an algorithm that classifies calls based on conversation, and [14] develops a "Humming call" smart phone application that helps with identifying spam calls.

However, there is little to no research on how existing production applications work (as it is also briefly discussed in [9]), including where they get their data from, what API calls the applications need and/or perform, and where the data regarding (potentially) spam/scam calls is stored.

Therefore, the main objective for this reseach is to develop a methodology that can be used to determine what Android API calls are performed by a pre-determined set of applications by performing dynamic analysis. Additionally, a script has to be developed that extracts the differences between subsequent runs of a given application with different inputs (different phone numbers that would be handled differently based on the functionality of the application).

2 Methodology

In order to dynamically analyse the API calls made by Android applications, ACVTool [10] was used [2]. After initial inspection of the reports generated by ACVTool (an example and description of the output of ACVTool can be found in Appendix D), it is obvious that a big number of the API calls made by the applications are irrelevant to the subject of the research (namely what API calls are needed for the call blocking and caller ID flagging nature of the applications, as described in Section 1). In order to ease the research process, a Python script was developed [3], such that the common API calls can be identified.

The process of analysing an application can be summarized as follows:

Using ACVTool:

1. Instrument a given application.
2. Upload the instrumented apk to the Android emulator.
3. Start ACVTool and simulate a phone call to the emulated device.
4. Generate a report.

This has to be done multiple times in order to generate reports for different cases, namely when the phone number is blocked, flagged as spam, or let through. These cases depend on the application that is being tested.

Then, using the report generated by ACVTool:

1. Extract the differences between the API calls from the several generated reports using the developed script. The tool will automatically extract the differences in all reports in the `reports/` directory, and will group them by application. So if there are multiple applications analysed, each having multiple reports, the script will extract the differences between the reports per application, and group the results by the application's corresponding directory.
2. Manually analyse the common API calls (this is not as important as the next step, but it is still important to try to go through some of the common libraries especially if there is obfuscation involved, as they likely contain useful information).
3. Manually analyse the different API calls.

Finally, the data extracted from the analysis, and the different techniques utilized and motivations are summarized in the next section. The next section also contains the findings and the developed workflow after analysing the first application referenced in Appendix A. More detailed notes and the extracted information can be found in the github repository [4].

[1] https://api.ctia.org/wp-content/uploads/2020/01/robocall-resources-for-android-2020.pdf

[2] https://github.com/pilgun/acvtool

[3] https://github.com/yoonhwanjeong/SpamCallBlockingApps/blob/dynamic_api_analysis/ref/extract_common_api_calls.py

[4] https://github.com/yoonhwanjeong/SpamCallBlockingApps/tree/dynamic_api_analysis/ref

Differences Extractor Tool

After the initial observations of the report generated by ACVTool, the need to somehow filter the irrelevant executed instructions from the decompiled code became apparent. This is the reason why a script that logs the decompiled smali code file names that have differences in the executed lines of code was developed.

The script utilizes the xml reports that were generated by ACVTool. For each apk analysed, the script compares the different reports generated and logs for which file the code coverage differs, and between which reports. This significantly lowers the number of files and lines of code that need to be manually analysed, as it removes the "expected" libraries that are executed in multiple runs of the same application. Usually these common libraries include low-level Android code that is irrelevant to the subject of research.

Still, there are a lot of cases, where code from these common libraries has to be analysed. This is mainly because a big part of the application code could be obfuscated and may have most of its strings and variable names removed, as was the case for the first application - Hiya. Therefore, in order to understand what information some variables hold, or what some executed methods do, deeper code analysis is usually required.

3 Dynamic analysis of the utilized Android system APIs

As described in Section 2 the dynamic analysis of an Android application involves a lot of manual code inspection. The results of this analysis and the developed methodology as a result of the performed research are described in the following subsections. Initially, I expected to be able to perform manual analysis on at least 5 different applications. However, this became unfeasible after I started with the first application, because in order to obtain any information, I had to spend a considerable amount of time and effords reverse engineering the obfuscated code of the application. Later, when I performed analysis on a second application, I found that it did not utilize nearly as much obfuscation, and the codebase was considerably smaller.

Still, the topic of the research was modified to include developing a methodology of reverse engineering these types of applications, as it would provide an approach that could be used later to analyse a bigger list of applications. This section, therefore, describes the workflow and results mostly of the first application - Hiya.

Experimental work

This section provides a discussion of the findings and motivations of the analysis performed on the application. Since decompiled code analysis is very time consuming, the research was performed on one application, namely the first provided in Appendix A. The following subsections summarize the findings after analysing the application and provides a discussion of the motivations and expected results from each of the performed actions.

Hiya: Initial Analysis

In the beginning of the analysis, several interesting library names were chosen for a quick code inspection, as they could contain some useful information about, for example, where the data was stored or how the application works. Especially the `com.hiya.client.database.db` library is interesting, because it contains references to a `Hiya.db` database and a lot of references to the `androidx.room` API. This is to be expected, as Room is an Android built-in library that provides a layer of abstraction over the default internal SQLite database [6].

More interesting is the `HiyaRoomDb_Impl$a.smali` decompiled file, because it contains several `CREATE TABLE` SQL statements, which reveal some internal database schema that is sumarized in Appendix C. `HiyaRoomDb_Impl.smali` also contains two interesting hashes, one of which is put into the `room_master_table` after it is initialized.

Finally, before getting into the deeper code analysis, I briefly looked over some of the libraries in the `com.hiya.stingray` package and found the following strings in some executed code segments:

Two interesting method names: "callLogDisplayType", "notificationType" were executed. Based on the names, it looks like they return how the UI should present the caller ID for this received phone call. It would be interesting to find out later where they get this data from, i.e. if it is from local storage, or from a remote database.

A "CallLifecycleHandler" object. Based on the class name, it sounds like this object handles the phone call's lifecycle, so it might be used to extract, analyse, and/or store some information about the call.

A format string: "Showing post call notification: reputation=$\%s$ identity=$\%s$ notification=$\%s$". The most interesting part is the logged "reputation" variable, because it would determine how the caller ID should be treated. So it should be further analysed where this "reputation" variable comes from.

These libraries are probably used to intercept the phone calls in real time and display the UI menu that appears whenever the phone is ringing. Regarding the Android APIs utilized to intercept the call, the `TelephonyManager` [7] and `BroadcastReceiver` [1] classes were used.

Analysing the Differences

After noting the initial observations from only one run of the application, it was run with 4 different phone numbers that can be found in Appendix B. Then, the script described in Section 2 was used to generate the differences between the four runs of the application. Each class was manually analysed, and the most interesting findings are summarized below:

There were some differences in some internal Android `WorkManager` libraries. This API is used to run code asynchronously by creating tasks and specify when they need to be run [4]. Following some of the classes referenced there, I found an Enum with values "CONNECTED", "METERED", "NOT_REQUIRED", "NOT_ROAMING", "UNMETERED", which belong to the `NetworkType` Android

enum [5]. As expected, the application would probably behave differently depending on what the network connection is.

There were also some differences in the code coverage in the scheduling internal library, and classes of the `WorkDatabase` and `ProcessUtils` types, but they are probably not relevant.

In an obfuscated class there is a method `this.isFraudOrSpam` that receives an instance of an enum as a parameter and returns a boolean. I also found a `this.toCallerId` that returns a `RoomCallerId` that has the same fields as one of the tables found earlier (`caller_ids` in Appendix C). So from the caller ID of the caller, the application determines if the call is a fraud or spam. It is still not clear where the data is fetched from.

There were some differences between some google libraries regarding number parsing, which is to be expected, as different phone numbers were used between the different application runs, and they need to be parsed differently based on the location. There were also some references to user configs and different languages, so this is probably related to locale settings. Some of these libraries probably deal with phone number parsing, which would explain why there are differences in the reports as different phone numbers would be parsed differently, depending on multiple factors, like geographical location, locale config of the device, phone number prefix, etc. It is interesting that in the report of the phone number that is flagged as suspected spam, a `NumberParseException` class from the i18n library is executed, and a custom `PhoneParserFailure` class returns an exception. This would indicate that this phone number has an unexpected format, but still is flagged as suspected spam. Therefore, there is some offline on-device phone number processing that is able to flag some caller IDs.

An interesting library used is the i18n internationalization library. It is used to differentiate between different phone types, but it can also be used to set up the region of the phone [3]. This is useful, in the use-case of the application, because some caller IDs might need to be handled differently, depending on the geographical region of the receiving phone. The application also differentiates between the following types of phone numbers: FIXED_LINE, FIXED_LINE_OR_MOBILE, MOBILE, PAGER, PERSONAL_NUMBER, PREMIUM_RATE, SHARED_COST, TOLL_FREE, UAN, UNKNOWN, VOICEMAIL, VOIP.

Most of the classes in the `com.hiya.stingray` package contain useful information. For example, a class inside the Stingray Manager iterates through the following enum values: SPAM, FRAUD, NEUTRAL, AUTO_BLOCK_PASS, AUTO_BLOCK_BLOCK, BLOCKED_STARTS_WITH, BLOCKED_BLACK_LIST, BLOCKED_AUTO_SPAM, BLOCKED_AUTO_FRAUD, BLOCKED_AUTO_PRIVATE, BLOCKED_CALL_SCREENER, ADD_BLACKLIST, REMOVE_BLACKLIST. Based on thses values it is apparent that the application also utilizes an internal database (to add and remove blocklisted phone numbers) together with some caller ID analysis (BLOCKED_STARTS_WITH), in order to categorize the phone numbers.

The application sets up a Realm databse [6] and implements a lot of realm data access objects (or DAOs). One of these objects that is saved in this database is a PhoneEvent object. It is saved in the test with the identified caller ID, but not in the test with the suspected spam caller. This is because this phone number is probably invalid, as it became clear when it was being parsed. This may be because of the location that the application perceived the phone to be in, combining it with the lack of a country code. Based on some error messages I found, this PhoneEvent object is saved, sent, and found in the `com.hiya.stingray.manager.l2` class.

While analysing the differenes in the one of the stingray packages, I found an interesting method name - `hashingCountryListProvider`, as it indicates that a list of country codes is utilized to perform some analysis on the phone numbers. Moreover, another variable name is `hashingCountriesDao`, therefore, since there is a data access object, there is also a database probably containing contry codes, stored on-device.

A similar object, called PhoneSendEvent, is also saved to the Realm database, and it has the following parameter names: time, phone, isContact, direction, termination, profileTag, phoneWithMeta, userDisposition, duration, clientDisposition, eventType, isBlackListed. The information stored in this object is apparently logging the phone calls that the device receives. These parameters indicate that information about the phone calls is stored in an application database, but I found no indication that this information left the device, so the application might perform some heuristic analysis, based on phone calls from a given caller ID, in order to determine if a call should be flagged. However, further research on the uses of this database table are required to determine how this information is used,especially because Realm is not the default internal Android database system and remote data synchronization could have been configured.

The methods in the stingray manager packages execute several interesting function calls, like for example `java/util/Locale;->getDisplayCountry()`. This might be used by the UI to display from what country the phone call originates, or it might also be saved in the internal Realm database (as it is in the `com.hiya.stingray.manager` package and not in any of the UI packages) to be used later. Also, an obfuscated library, referenced in a class from the stingray manager package, pattern matches country codes. It is probably used in combination with the country list mentioned above to categorize caller IDs based on contries. Additionally, there are several enums with interesting values, most notably ADD_BLACKLIST and REMOVE_BLACKLIST, which would indicate that users can also blocklist numbers they consider spam.

Analysing Code Containing Interesting Strings

After analysing the differences in the executed instructions, it was still not obvious where the data was taken from. Therefore, I decided to search for some potentially useful strings through the decompiled code, and analyse the classes where they were used. I had to also search for some of the classes

[5] https://developer.android.com/reference/androidx/work/NetworkType

[6] https://realm.io/

and enums that were found above to find where they were initialized or referenced.

One of the obfuscated functions, namely `g.g.b.c.f->v()`, that was found during the analysis of the differences in the executed instructions, returns whether something is fraud or spam. Therefore, after searching what initializes this object, and thus saves the data that `v()` returns, I found several potentially useful classes. One of them is a UI class that deals with displaying the phone numbers as it has parameters with the following names: "formattedPhone", "rawPhone", "contact", and references a ViewUtil object.

I also found the following class names and mapped them to the obfuscated variables used in the code: BlockManagerLazy, DeviceUserInfoManager, CallScreenerHelperLazy, RxEventBus, ContactManager. The most interesting would be the ContactManager, as I had encountered it before, and it could indicate that whether a phone number is in the device's contract list could influence how it was shown in the app. It would be interesting to further test how the contact information is used by the app, and whether it is sent somewhere over the network. I found an indication that this information may be sent over the Internet, which is discussed below.

I continued searching for where some of the more interesting functions are called. It was done by recursively searching throught the strings of the decompiled Android code for the name of the method, in order to determine where it is referenced. For example, the method that calls `this.isFraudOrSpam` is referenced in a method in a different obfuscated class that then calls `getReputationLevel()` and then uses the returned value to decide whether the call is SPAM or FRAUD. Also, apparently there are different FRAUD enums that are being referenced throughout the code base. Using this information I was able to conclude that the reputation level of a caller ID is used to determine whether the call is spam, fraud, or neither.

At this point I started going through the more interesting classes that are connected to getting the reputation of a caller ID, since this would indicate where the caller ID information is fetched from. There are a lot of DAO objects, like `RoomCallerId` that are initialized, and some enums, most notably ProfileIconType with possible values BUSINESS, WARN, PERSON, STOP, PREMIUM, NONE, and ReputationLevel with possible values OK, UNCERTAIN, SPAM, FRAUD. This step requires many recursive string searches and analysis of obfuscated object references.

It was important to find where the caller ID information is stored. I started searching for references to the enum and DAO classes, containing information about the caller IDs. Apparently, this information comes from the Room database, because there is a `SELECT * FROM caller_ids` string passed to a method from the `androidx.room` package. The implementation of the Room DB class that returns the caller IDs uses a Map object stored as a global variable to store and return the required values. The values in the Map were added from a different method in the same class, where they are taken from a parameter given to this method. Furthermore, there is a global variable that is used as an index for the variable passed as a method parameter.

The method that adds the values to the internal Map structure is called in 11 different classes, 6 of which are in the `androidx.work.impl` package. It makes sense that the Room DB implementation utilizes the internal Android WorkManager, so I did not spend too much time analysing them. However, all of the 11 classes look like internal database calls. I could not find any references to remote connections that fetch the caller ID information.

Additionally, one of the DAO objects contains a variable that is referenced by the string `reputationLevel`. The global variable that holds this value is only set in the constructor, and has one getter function that is referenced in multiple places in the application. The information about the reputation level of the caller in this package looks to be taken from calls to the internal Room database. Since this information could have been cached, I decided to further search for some common internal Android libraries that are associated with making requests over the Internet.

Analysing Code Referencing Certain Android Libraries

One such class would be `HttpURLConnection`, however, there are too many obfuscated places where this class is referenced. The Google and Android classes can probably be ignored, as they would contain mostly library functions that do not provide useful information. I looked over some of the classes and most are not executed in any of the reports. The most interesting things I found are summarized below.

There is a ping request to https://pagead2.googlesyndication.com/pagead/gen_204?id=gmob-apps. This URL seems to be connected to Google's advertisements, which would indicate the application uses Google AdSense [7]. However, the package name is obfuscated, so unfortunately as some Android APIs cannot be discovered from the package name alone, they need to be analysed as well.

Additionally, there are many references to Google Protocol Buffers, which is a protocol used to serialize and deserialize data to and from raw bytes [5]. They may be used by an Android or Google library, but it is also possible the application uses them to transmit some data, because as it is discussed below, the application also uses GSON [8].

Moreover, the `g.g.a.a.k.j` class performs some `m/z$a;->request()`'s and may throw a `HiyaExcessiveAuthRequestsException`, so it probably makes some internal Hiya HTTP authenticated requests. However, I could not find credentials being hardcoded anywhere, so they might be stored in a database or are heavily obfuscated.

Finally, I found a class that links the following strings to global variables and contains only getter and setters (which could inticate it is a DAO, or some data-store object): "attributionDTO", "displayCategory", "displayImageString", "displayLocation", "displayMessage", "displayName", "entityType", "localizedLineType", "profileTag", "reputationLevel". Each of them is an object that can be further analysed if necessary, but they appear to be connected to the UI part of the application.

[7] https://www.google.com/adsense/

[8] https://github.com/google/gson

Analysing Code Referencing Certain Strings

Another interesting string to search for is "JSON", because of how popular the JSON serialization protocol is to transmit data over the internet, as pointed out by [13]. The files that contain "JSON" reveal some information about the requests the application makes, but most of them are not connected to determining whether a phone call should be considered as spam. The only potentially relevant information is fetched with an HTTP GET request that retrieves `hash/hashCountries`. The hashed countries list may be used to determine whether a phone number is considered as spam based on the geographical location of the user and/or the location of the dialing phone number.

Other references of "JSON" include a file called `Hiya_Services.json`, a service for reporting called Zipkin [9], a package `revenuecat.purchases` that is used to manage paid subscriptions [10], and several HTTP requests. Most notably, the application makes the following requests: POST requests to `auth/token`, `phone_numbers/feedback`, `phone_numbers/events`; POST request to `phone_numbers/eventProfile` that probably sends an EventProfile object to some server, and expects a JSON response. `auth/token` is probably used to generate some session token, and `phone_numbers/feedback` probably sends users's feedback, if they want to share it.

However, the requests to `phone_numbers/events` and `phone_numbers/eventProfile` are really interesting. The EventProfile object contains the following variable names: "attribution", "displayBackground", "displayDescription", "displayDetail", "displayImage", "displayName", "profileDetails", "profileIcon", "profileTag", "reputationLevel", "verified", so it is probably connected to displaying the information on the user interface. The request to `phone_numbers/events` expects a list of `g.g.a.a.i.k.d` objects. This class contains the following variable names: "cachedProfileTag", "clientSignal", "clientTag", "disposition", "eventProfileEvent", "profileTag", of which the "disposition" and "eventProfileEvent" objects are the only ones that are not `String`s. The former is not interesting (it contains mainly just enums), but the latter is an object with the following variables: "direction", "duration", "isBlock", "isContact", "lastInteraction", "phone", "termination", "timestamp", "tokens", "type".

This object suggest that information about the received phone calls could be sent in an HTTP POST request. It is important to note that there is no indication that the methods containing the POST requests have been executed in any of the reports. Nevertheless, just the possibility to send an "Event" object is alarming from a privacy point of view, although the context, in which it is used, is not known.

Finally, the last analysed file that references "JSON" is an obfuscated class containing some cryptographic information, although most of it is not executed in the reports. First, there are two `Date` variables initialized in one of the two constructors: for years 1970 and 2048, so this is probably the timespan in which the certificates in the other methods are valid. The second constructor reads the following variables from its parameters: "userInfoProvider", "productInfoProvider", "idProvider", "measurable". Then, the other methods use a `java.security.PrivateKey` and a `java.security.cert.Certificate` objects for JOSE (Javascript Object Signing and Encryption) with `com.nimbusds.jose` [11]. This implies that the application needs to transmit some JSON objects securely over a network. The class uses `RSA` with `SHA-512` to generate public-private key pair to be used to securely share some secret probably with a remote server.

Another method builds a GSON object with the following fields: "createdAt", "productName", "installationId", "deviceId", "accountUserId", "userPhoneNumber", and then converts it to a string. Considering the field names, this object is probably used for logging some of the device's information to a server. Especially interesting is the "userPhoneNumber" field, because it implies the phone number of the user using the application is saved on a Hiya server and tied to the user's "accountUserId" and more importantly to the "deviceId", which should be private information.

The last two methods in this class contain: a hard-coded `PKCS8EncodedKeySpec` and `X509` strings, which are probably Hiya's public key and are used to generate private keys, because there is an exception with the string "Failed to generate private key based on the Hiya key." if the key generation fails; a base64 string "6bef0890a22741259d0d28035810f5cc" that is decoded and XORed with the second method parameter.

RoomCallerId and SQL

The last things I decided to search for are the RoomCallerId data class, as it is an object containing information about the caller ID, and the string "SQL", because I already found some SQL schemas, queries, and references to a database, so it would be useful to find what other information is saved. I found an insert SQL statement that is being built for the `caller_ids` table, which is probably for an internal database, as it was with the select statements discussed above. Furthermore, `RoomCallerId` objects are converted to `CallerId`, and one method iterates through Set objects given as parameters and references a `com.hiya.client.database.db.HiyaRoomDb` class to get `phone`, `phoneNumbers`, and `callerId`s. This is a similar pattern as the one observed for the select statements discussed above with the difference that there a Map object was used instead of a Set. I could not find any indication that the `HiyaRoomDb` object connects to a remote database, and all references to it suggest the use of an internal database.

The search for "SQL" produced mostly expected results: mostly references to SQLite [12] (the internal Android database system) and some `java.sql` classes like `Timestamp`, `Date`, and `Time`. An interesting database is referenced in the `com.hiya.client.repost.db.a` class, which opens "Elixir.db" and creates a table called `stored_requests` with

[9] https://zipkin.io/

[10] https://www.revenuecat.com/

[11] https://mvnrepository.com/artifact/com.nimbusds/nimbus-jose-jwt

[12] https://developer.android.com/training/data-storage/sqlite

type, body, and retry_count fields, and also contains the relevant select, update, insert, and delete queries.

This indicates that some HTTP requests are being built and stored in this database, although it is important to note this class is not executed in any of the reports. Other interesting table names I found are "events", "event_metadata", "transport_contexts", and "event_payloads", and their corresponding "create", "alter", and "drop" SQL queries. Neither of the SQL modifier statements were executed in any of the reports, but this is probably because there might be certain conditions that need to be met. For example, the "create" query might only be executed when the application starts, and the "drop" query might be executed whenever the application is updated. However, there is an SQLiteEventStore database object in the g.f.a.b.i.x.j.b0 class that operates on the "events" table. This "events" table seems to save different objects than the ones discussed with the phone_numbers/events request, because the names of the variables are different.

Some of the "events" database table's variable names are "context_id", "transport_name", "timestamp_ms", "uptime_ms", "payload_encoding", "code", "num_attempts", "inline", "payload". Initially, I thought that this database is connected to the PhoneEvent object, however after searching for the schema, I found it is used by the Google Firebase library [13], so it is not connected to the flagging functionality of the application.

Additionally, one class from the com.hiya.stingray package that is executed in each of the four reports uses the call logs from the internal database. Possibly, the application uses them to flag caller IDs with certain call patterns that could indicate they are spam. This class uses another internal Android package that compiles and executes SQL statements, and enables write ahead logging for the SQLite database. Therefore, it might be useful to further analyse where this internal library is referenced, in order to determine what other SQL statements are built, what class builds them, and on what database they are executed.

Finally, even though these results do not show exactly how the application operates, they can be used to further analyse the codebase by searching for where the classes are referenced, and which methods are executed in any of the reports.

I did not find where the information about the caller IDs was initially obtained from. All database references I could find use the local database, and I did not find any indication that the application made external calls to get information about the phone numbers. With that being said, I also tried intercepting the network traffic using BurpSuite [14] to find if numbers are being sent anywhere. Unfortunately, even after I installed the custom CA certificated on the emulated device, the application did not try to perform any network requests whenever the intercepting proxy was turned on. This also caused the Hiya overlay to not display any useful information regarding the caller ID, which would imply that the application does indeed require unintercepted internet connection to fetch its data. Furthermore, it appears the application is saving some information regarding the phone calls performed by the users, which might later be used to change the reputation level of the caller IDs, and may even be sent to a remote Hiya server to build their database of scam phone numbers.

[13] https://github.com/firebase/firebase-android-sdk/blob/master/transport/transport-runtime/src/main/java/com/google/android/datatransport/runtime/scheduling/persistence/SchemaManager.java

[14] https://portswigger.net/burp

4 Responsible Research

All data that was used and extracted during the research is publicly available [15]. Since the purpose of this research is to analyse the API calls performed by Android applications for educational purposes only, it is ethical to apply the reverse engineering techniques described in this paper, in order to uncover what APIs the spam call blocking applications require in order to function [2]. From an ethical point of view it is important to perform such application analysis in order to uncover any potential unconventional or malicious practices the applications could employ.

The APK files of the applications analysed, the reports generated by ACVTool, the script used to extract the differences between subsequent runs of the same application as well as its outputs are all available in the project repository. This makes the project and the methods in this paper fully reproducible.

5 Discussion

It is important to note that most of the information in this paper is based on speculation and a lot of educated guesses. This is because of the whole nature of reverse engineering and dynamic code analysis. In order to properly analyse an application, a lot more time would be required and more sophisticated tools for reverse engineering of Android bytecode would be necessary to be developed.

However, this paper and the extracted data still serve as an interesting insight on to the inner workings of one of the most popular Android spam call blocking applications. Additionally, the methodology described in the paper can be further utilized to find more information about the analysed application, or to extract information from any other Android application with a similar functionality.

6 Conclusions and Future Work

After performing in-depth dynamic analysis of the selected application, it became apparent to me that such analysis should only be performed for applications that are hard to analyse statically or using different methods. Reverse engineering, especially when obfuscation techniques were employed, becomes too time and efford consuming to be worth utilizing it for mass application analysis. Nevertheless, the research performed in this paper provides an interesting methodology approach that could be used whenever little information could be extracted from other analysis approaches, or when simply more information is needed for a specific application.

Initially, the plan was to perform a dynamic analysis on all 10 applications, however, after several weeks focusing on one application, it became apparent that more useful would be to

[15] https://github.com/yoonhwanjeong/SpamCallBlockingApps/tree/dynamic_api_analysis

focus on developing a structured approach to reverse engineer the list of applications rather than going over each of them breifly. Developing a methodology that can be utilized to perform a more in-depth analysis could also be useful for analysis of different types of Android applications, not only the spam flagging or blocking ones. Furthermore, initially I underestimated the amount of time and efford reverse engineering decompiled Android code would take.

Future research on this topic could try to integrate string search functionality from the decompiled source code with the xml report generated by ACVTool. In this way, the researcher would be able to search for strings in the code, be it method/class/library names, or hardcoded strings, that were actually executed. Currently, the workflow includes manually executing recursive `grep` for some string found in the reports to see where some functions were called, however, in a lot of cases, these functions not executed in any of the reports. Although it is also useful to read through code that was not executed, I found that most of the useful information is found in the instructions that were executed.

Finally, the ability to rename some of the packages, classes, variables, and types would be a really useful addition to ACVTool, or any reverse engineering tool. After extensive analysis of the decompiled code, it became apparent that keeping track of all packages and classes would be increasingly difficult the more obfuscated classes an application has. In the case of Hiya, most of the code had no sensible strings, so I had to manually keep track of class names like `g.g.b.b.a.e` and `g.g.b.b.a.g`. Even though not all applications are that difficult to analyse (like for example telGuarder had a lot smaller codebase, so it was easier to keep track of the classes), allowing variable and class renaming would greatly ease the reverse engineer's work.

A Application Analysed

- Hiya - Call Blocker, Fraud Detection & Caller ID: https://play.google.com/store/apps/details?id=com.webascender.callerid&hl=en&gl=US
- Spam Call Blocker - telGuarder: https://play.google.com/store/apps/details?id=com.telguarder&hl=en&gl=US

B Hiya Phone Numbers Tested

Number	How it is handled
(650) 555-1212	produces a name and location of caller
605-367-1378	produces a warning
0611945863112	produces "suspected spam"
201-200-0014	falgged but also identified caller ID

C Hiya Database Schema

caller_ids	
_id	**INTEGER**
entity_type	TEXT
phone_number	TEXT
display_name	TEXT
display_location	TEXT
display_image_url	TEXT
attribution_image	TEXT
attribution_url	TEXT
attribution_name	TEXT
profile_tag	TEXT
display_line_type	TEXT
entity_expired_time_millis	INTEGER
source_type	TEXT
last_access_time_millis	INTEGER
profile_icon_type	TEXT
reputation_category_id	INTEGER
category_name	TEXT
display_category_name	TEXT
line_type_id	TEXT
display_detail	TEXT
display_description	TEXT
language_tag	TEXT
display_background_url	TEXT
display_background_assettype	TEXT

local_override_ids	
_id	**INTEGER**
phone_number	TEXT
reported_name	TEXT
user_comment	TEXT
category_name	TEXT
reputation_category_id	INTEGER
profile_tag	TEXT
time_created	INTEGER

translated_strings	
_id	**INTEGER**
key	TEXT
translated_text	TEXT

postevent_data	
_id	**INTEGER**
type	TEXT
direction	TEXT
phone_number	TEXT
country_hint	TEXT
duration	INTEGER
is_missed	INTEGER
is_blocked	INTEGER
is_contact	INTEGER
timestamp	INTEGER
profile_tag	TEXT
block_reason	INTEGER

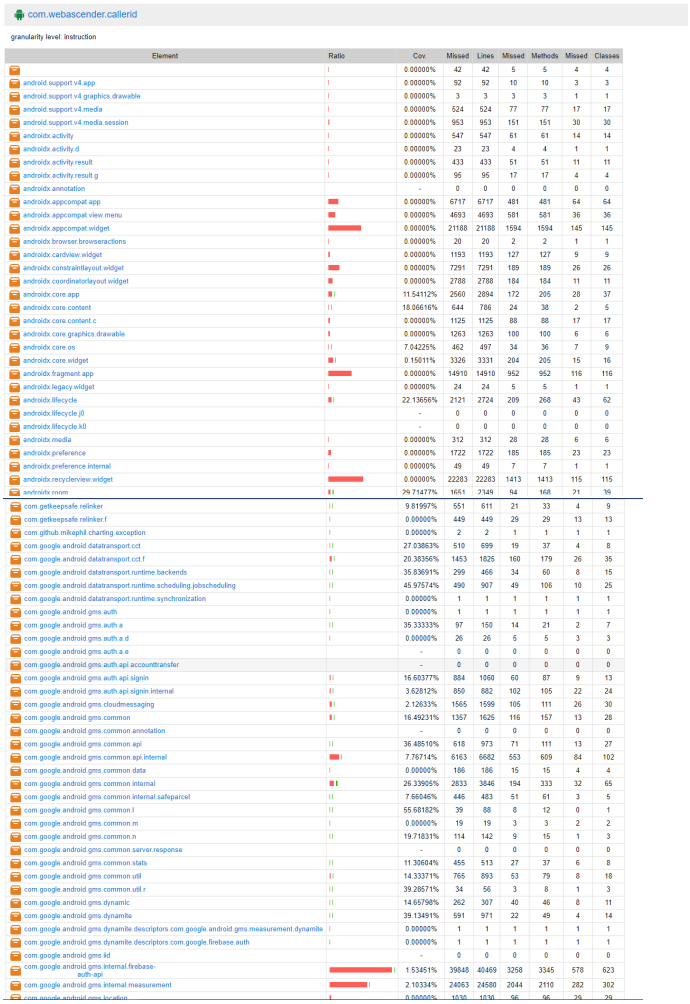
room_master_table	
_id	**INTEGER**
identity_hash	TEXT

block_numbers	
_id	**INTEGER**
phone_number	TEXT
normalized_number	TEXT
created_time_millis	INTEGER
is_partial	INTEGER
country_calling_code	INTEGER

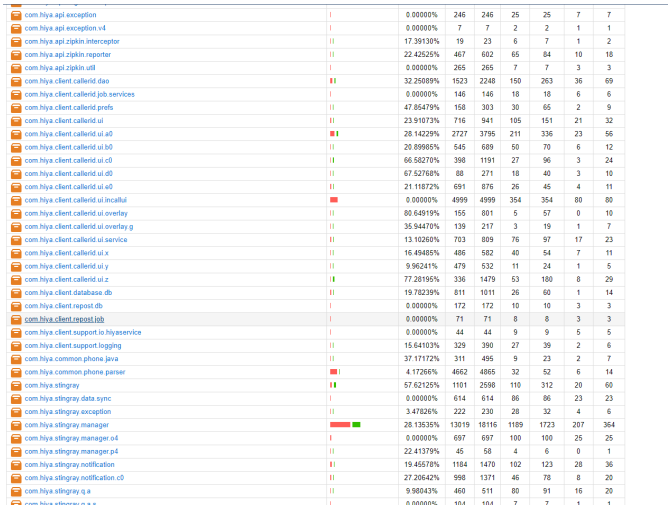
D ACVTool Example Output

The following images are screenshots of the reports generated by ACVTool.

The following images are some of the libraries in the generated ACVTool report. The name of the package is on the left, and some statistics about the number of executed and missed instructions is on the right side:



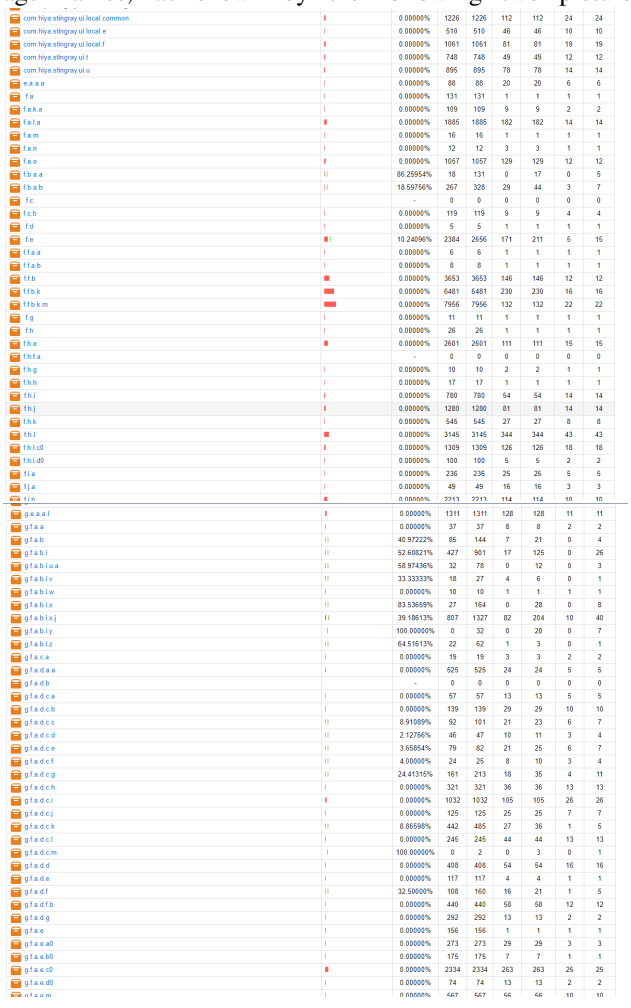
Element	Ratio	Cov.	Missed	Lines	Missed	Methods	Missed	Classes
android.support.v4.app		0.00000%	42	42	5	5	4	4
android.support.v4.graphics.drawable		0.00000%	92	92	10	10	3	3
android.support.v4.media		0.00000%	3	3	3	3	1	1
android.support.v4.media.session		0.00000%	524	524	77	77	17	17
androidx.activity		0.00000%	953	953	151	151	30	30
androidx.activity.d		0.00000%	547	547	61	61	14	14
androidx.activity.result		0.00000%	23	23	4	4	1	1
androidx.activity.result.g		0.00000%	433	433	51	51	11	11
androidx.annotation		0.00000%	95	95	17	17	4	4
androidx.appcompat.app		0.00000%	6717	6717	481	481	64	64
androidx.appcompat.view.menu		0.00000%	4693	4693	581	581	36	36
androidx.appcompat.widget		0.00000%	21188	21188	1594	1594	145	145
androidx.browser.browseractions		0.00000%	20	20	2	2	1	1
androidx.cardview.widget		0.00000%	1193	1193	127	127	9	9
androidx.constraintlayout.widget		0.00000%	7291	7291	189	189	26	26
androidx.coordinatorlayout.widget		0.00000%	2788	2788	184	184	11	11
androidx.core.app		11.54112%	2560	2894	172	205	28	37
androidx.core.content		18.06616%	644	786	24	38	2	5
androidx.core.content.c		0.00000%	1125	1125	88	88	17	17
androidx.core.graphics.drawable		0.00000%	1263	1263	100	100	6	6
androidx.core.os		7.04225%	462	497	34	36	7	9
androidx.core.widget		0.15011%	3326	3331	204	205	15	16
androidx.fragment.app		0.00000%	14910	14910	952	952	116	116
androidx.legacy.widget		0.00000%	24	24	5	5	1	1
androidx.lifecycle		22.13656%	2121	2724	209	268	43	62
androidx.lifecycle.j0		-	0	0	0	0	0	0
androidx.lifecycle.k0		-	0	0	0	0	0	0
androidx.media		0.00000%	312	312	28	28	6	6
androidx.preference		0.00000%	1722	1722	185	185	23	23
androidx.preference.internal		0.00000%	49	49	7	7	1	1
androidx.recyclerview.widget		0.00000%	22283	22283	1413	1413	115	115
androidx.room		29.71477%	1651	2349	94	168	21	36
com.getkeepsafe.relinker		9.81997%	551	611	21	33	4	9
com.getkeepsafe.relinker.f		0.00000%	449	449	29	29	13	13
com.github.mikephil.charting.exception		0.00000%	2	2	1	1	1	1
com.google.android.datatransport.cct		27.03863%	510	699	19	37	4	8
com.google.android.datatransport.cct.f		20.38356%	1453	1825	160	179	26	35
com.google.android.datatransport.runtime.backends		35.83691%	299	466	34	60	8	15
com.google.android.datatransport.runtime.scheduling.jobscheduling		45.97574%	490	907	49	106	10	25
com.google.android.datatransport.runtime.synchronization		0.00000%	1	1	1	1	1	1
com.google.android.gms.auth		0.00000%	1	1	1	1	1	1
com.google.android.gms.auth.a		35.33333%	97	150	14	21	2	7
com.google.android.gms.auth.a.d		0.00000%	26	26	5	5	3	3
com.google.android.gms.auth.a.e		-	0	0	0	0	0	0
com.google.android.gms.auth.api.accounttransfer		-	0	0	0	0	0	0
com.google.android.gms.auth.api.signin		16.60377%	884	1060	60	87	9	13
com.google.android.gms.auth.api.signin.internal		3.62812%	850	882	102	105	22	24
com.google.android.gms.cloudmessaging		2.12633%	1565	1599	105	111	26	30
com.google.android.gms.common		16.49231%	1357	1625	116	157	13	28
com.google.android.gms.common.annotation		-	0	0	0	0	0	0
com.google.android.gms.common.api		36.48510%	618	973	71	111	13	27
com.google.android.gms.common.api.internal		7.76714%	6163	6682	553	609	84	102
com.google.android.gms.common.data		0.00000%	186	186	15	15	4	4
com.google.android.gms.common.internal		26.33905%	2833	3846	194	333	32	65
com.google.android.gms.common.internal.safeparcel		7.66046%	446	483	51	61	3	5
com.google.android.gms.common.l		55.68182%	39	88	8	12	0	1
com.google.android.gms.common.m		0.00000%	19	19	3	3	2	2
com.google.android.gms.common.n		19.71831%	114	142	9	15	1	3
com.google.android.gms.common.server.response		-	0	0	0	0	0	0
com.google.android.gms.common.stats		11.30604%	455	513	27	37	6	8
com.google.android.gms.common.util		14.33371%	765	893	53	79	8	18
com.google.android.gms.common.util.r		39.28571%	34	56	3	8	1	3
com.google.android.gms.dynamic		14.65798%	262	307	40	46	8	11
com.google.android.gms.dynamite		39.13491%	591	971	22	49	4	14
com.google.android.gms.dynamite.descriptors.com.google.android.gms.measurement.dynamite		0.00000%	1	1	1	1	1	1
com.google.android.gms.dynamite.descriptors.com.google.firebase.auth		0.00000%	1	1	1	1	1	1
com.google.android.gms.ild		-	0	0	0	0	0	0
com.google.android.gms.internal.firebase-auth-api		1.53451%	39848	40469	3258	3345	578	623
com.google.android.gms.internal.measurement		2.10334%	24063	24580	2044	2110	282	302
com.google.android.gms.location		0.00000%	1030	1030	96	96	29	29



com.hiya.api.exception	0.00000%	246	246	25	25	7	7
com.hiya.api.exception.v4	0.00000%	7	7	2	2	1	1
com.hiya.api.zipkin.interceptor	17.39130%	19	23	6	7	1	2
com.hiya.api.zipkin.reporter	22.42525%	467	602	65	84	10	18
com.hiya.api.zipkin.util	0.00000%	265	265	7	7	3	3
com.hiya.client.callerid.dao	32.25089%	1523	2248	150	263	36	69
com.hiya.client.callerid.job.services	0.00000%	146	146	18	18	6	6
com.hiya.client.callerid.prefs	47.85479%	158	303	30	65	2	9
com.hiya.client.callerid.ui	23.91073%	716	941	105	151	21	32
com.hiya.client.callerid.ui.a0	28.14229%	2727	3795	211	336	23	56
com.hiya.client.callerid.ui.b0	20.89985%	545	689	50	70	6	12
com.hiya.client.callerid.ui.c0	66.58270%	398	1191	27	96	3	24
com.hiya.client.callerid.ui.d0	67.52768%	88	271	18	40	3	10
com.hiya.client.callerid.ui.e0	21.11872%	691	876	26	45	4	11
com.hiya.client.callerid.ui.incallui	0.00000%	4999	4999	354	354	80	80
com.hiya.client.callerid.ui.overlay	80.64919%	155	801	5	57	0	10
com.hiya.client.callerid.ui.overlay.g	35.94470%	139	217	3	19	1	7
com.hiya.client.callerid.ui.service	13.10260%	703	809	76	97	17	23
com.hiya.client.callerid.ui.x	16.49485%	486	582	40	54	7	11
com.hiya.client.callerid.ui.y	9.96241%	479	532	11	24	1	5
com.hiya.client.callerid.ui.z	77.28195%	336	1479	53	180	8	29
com.hiya.client.database.db	19.78239%	811	1011	26	60	1	14
com.hiya.client.repost.db	0.00000%	172	172	10	10	3	3
com.hiya.client.repost.job	0.00000%	71	71	8	8	3	3
com.hiya.client.support.io.hiyaservice	0.00000%	44	44	9	9	5	5
com.hiya.client.support.logging	15.64103%	329	390	27	39	2	6
com.hiya.common.phone.java	37.17172%	311	495	9	23	2	7
com.hiya.common.phone.parser	4.17266%	4662	4865	32	52	6	14
com.hiya.stingray	57.62125%	1101	2598	110	312	20	60
com.hiya.stingray.data.sync	0.00000%	614	614	86	86	23	23
com.hiya.stingray.exception	3.47826%	222	230	28	32	4	6
com.hiya.stingray.manager	28.13535%	13019	18116	1189	1723	207	364
com.hiya.stingray.manager.o4	0.00000%	697	697	100	100	25	25
com.hiya.stingray.manager.p4	22.41379%	45	58	4	6	0	1
com.hiya.stingray.notification	19.45578%	1184	1470	102	123	28	36
com.hiya.stingray.notification.c0	27.20642%	998	1371	46	78	8	20
com.hiya.stingray.q.a	9.98043%	460	511	80	91	16	20
com.hiya.stingray.q.e	0.00000%	104	104	7	7	1	1

The package names are not obfuscated, so it is easier to distinguish the internal Android API calls from the Hiya packages.

Unfortunately, there are also a lot of obfuscated package names, as shown by the following two pictures:



com.hiya.stingray.ui.local.common	0.00000%	1226	1226	112	112	24	24
com.hiya.stingray.ui.local.e	0.00000%	510	510	46	46	10	10
com.hiya.stingray.ui.local.f	0.00000%	1061	1061	81	81	19	19
com.hiya.stingray.ui.t	0.00000%	748	748	49	49	12	12
com.hiya.stingray.ui.u	0.00000%	895	895	78	78	14	14
e.a.a.a	0.00000%	88	88	20	20	6	6
f.a	0.00000%	131	131	1	1	1	1
f.a.k.a	0.00000%	109	109	9	9	2	2
f.a.l.a	0.00000%	1885	1885	182	182	14	14
f.a.m	0.00000%	16	16	1	1	1	1
f.a.n	0.00000%	12	12	3	3	1	1
f.a.o	0.00000%	1057	1057	129	129	12	12
f.b.a.a	86.25954%	18	131	0	17	0	5
f.b.a.b	18.59756%	267	328	29	44	3	7
f.c	-	0	0	0	0	0	0
f.c.b	0.00000%	119	119	9	9	4	4
f.d	0.00000%	5	5	1	1	1	1
f.e	10.24096%	2384	2656	171	211	5	15
f.f.a.a	0.00000%	6	6	1	1	1	1
f.f.a.b	0.00000%	8	8	1	1	1	1
f.f.b	0.00000%	3653	3653	146	146	12	12
f.f.b.k	0.00000%	6481	6481	230	230	16	16
f.f.b.k.m	0.00000%	7956	7956	132	132	22	22
f.g	0.00000%	11	11	1	1	1	1
f.h	0.00000%	26	26	1	1	1	1
f.h.e	0.00000%	2601	2601	111	111	15	15
f.h.f.a	-	0	0	0	0	0	0
f.h.g	0.00000%	10	10	2	2	1	1
f.h.h	0.00000%	17	17	1	1	1	1
f.h.i	0.00000%	780	780	54	54	14	14
f.h.j	0.00000%	1280	1280	81	81	14	14
f.h.k	0.00000%	545	545	27	27	8	8
f.h.l	0.00000%	3145	3145	344	344	43	43
f.h.l.c0	0.00000%	1309	1309	126	126	18	18
f.h.l.d0	0.00000%	100	100	5	5	2	2
f.i.a	0.00000%	236	236	25	25	5	5
f.j.a	0.00000%	49	49	16	16	3	3
f.j.b	0.00000%	2213	2213	114	114	10	10
g.a.a.a.l	0.00000%	1311	1311	128	128	11	11
g.f.a.a	0.00000%	37	37	8	8	2	2
g.f.a.b	40.97222%	85	144	7	21	0	4
g.f.a.b.i	52.60821%	427	901	17	125	0	26
g.f.a.b.i.u.a	58.97436%	32	78	0	12	0	3
g.f.a.b.i.v	33.33333%	18	27	4	6	0	1
g.f.a.b.i.w	0.00000%	10	10	1	1	1	1
g.f.a.b.i.x	83.53659%	27	164	0	28	0	8
g.f.a.b.i.x.j	39.18613%	807	1327	82	204	10	40
g.f.a.b.i.y	100.00000%	0	32	0	20	0	7
g.f.a.b.i.z	64.51613%	22	62	1	3	0	1
g.f.a.c.a	0.00000%	19	19	3	3	2	2
g.f.a.d.a.a	0.00000%	525	525	24	24	5	5
g.f.a.d.b	-	0	0	0	0	0	0
g.f.a.d.c.a	0.00000%	57	57	13	13	5	5
g.f.a.d.c.b	0.00000%	139	139	29	29	10	10
g.f.a.d.c.c	8.91089%	92	101	21	23	6	7
g.f.a.d.c.d	2.12766%	46	47	10	11	3	4
g.f.a.d.c.e	3.65854%	79	82	21	25	6	7
g.f.a.d.c.f	4.00000%	24	25	8	10	3	4
g.f.a.d.c.g	24.41315%	161	213	18	35	4	11
g.f.a.d.c.h	0.00000%	321	321	36	36	13	13
g.f.a.d.c.i	0.00000%	1032	1032	105	105	26	26
g.f.a.d.c.j	0.00000%	125	125	25	25	7	7
g.f.a.d.c.k	8.86598%	442	485	27	36	1	5
g.f.a.d.c.l	0.00000%	245	245	44	44	13	13
g.f.a.d.c.m	100.00000%	0	2	0	3	0	1
g.f.a.d.d	0.00000%	408	408	54	54	16	16
g.f.a.d.e	0.00000%	117	117	4	4	1	1
g.f.a.d.f	32.50000%	108	160	16	21	1	5
g.f.a.d.f.b	0.00000%	440	440	58	58	12	12
g.f.a.d.g	0.00000%	292	292	13	13	2	2
g.f.a.e	0.00000%	156	156	1	1	1	1
g.f.a.e.a0	0.00000%	273	273	29	29	3	3
g.f.a.e.b0	0.00000%	175	175	7	7	1	1
g.f.a.e.c0	0.00000%	2334	2334	263	263	25	25
g.f.a.e.d0	0.00000%	74	74	13	13	2	2
g.f.a.e.m	0.00000%	567	567	56	56	10	10

They are a lot harder to analyse, because there is no way of knowing which would be relevant, and which are lower-level

Android API calls.

The following images show the classes in the `com.hiya.stingray.manager` and `com.hiya.stingray.exception` packages, respectively:

com.webascender.callerid > com.hiya.stingray.manager

granularity level: instruction

Element	Ratio	Cov.	Missed	Lines	Missed	Methods	Missed	Classes
a.smali		0.00000%	7	7	3	3	1	1
a0.smali		0.00000%	7	7	2	2	1	1
a1.smali		0.00000%	5	5	2	2	1	1
a2.smali		0.00000%	9	9	3	3	1	1
a3.smali		0.00000%	1	1	1	1	1	1
a4$a.smali		2.27273%	43	44	4	5	0	1
a4$b.smali		0.00000%	10	10	3	3	1	1
a4$c.smali		0.00000%	7	7	4	4	1	1
a4.smali		29.58580%	119	169	8	13	0	1
b.smali		0.00000%	5	5	2	2	1	1
b0.smali		0.00000%	6	6	3	3	1	1
b1.smali		51.47059%	33	68	0	4	0	1
b2.smali		52.74725%	43	91	1	5	0	1
b3$a.smali		31.25000%	11	16	2	4	0	1
b3$b.smali		100.00000%	0	3	0	3	0	1
b3$c.smali		20.49180%	97	122	7	10	0	1
b3.smali		55.90551%	56	127	2	7	0	1
b4.smali		0.00000%	38	38	4	4	1	1
c.smali		0.00000%	22	22	2	2	1	1
c0.smali		0.00000%	9	9	2	2	1	1
c1.smali		0.00000%	36	36	1	1	1	1
c2$a.smali		0.00000%	7	7	3	3	1	1
c2.smali		5.70175%	430	456	29	31	0	1
c3.smali		100.00000%	0	10	0	4	0	1
c4$a.smali		0.00000%	34	34	4	4	1	1
c4$b.smali		0.00000%	52	52	4	4	1	1
c4$c.smali		0.00000%	46	46	4	4	1	1
c4.smali		0.00000%	401	401	4	4	1	1
d.smali		33.33333%	4	6	1	2	0	1
d0.smali		0.00000%	6	6	3	3	1	1
d1$a.smali		0.00000%	167	167	7	7	1	1
d1.smali		0.00000%	8	8	3	3	1	1
d2.smali		100.00000%	0	68	0	4	0	1
x3$a.smali		0.00000%	33	33	5	5	1	1
x3$b.smali		0.00000%	1	1	1	1	1	1
x3$c.smali		86.36364%	3	22	1	4	0	1
x3da.smali		0.00000%	59	59	2	2	1	1
x3db.smali		0.00000%	6	6	2	2	1	1
x3$d.smali		0.00000%	12	12	2	2	1	1
x3$e.smali		0.00000%	13	13	4	4	1	1
x3$f.smali		100.00000%	0	8	0	4	0	1
x3.smali		69.98445%	386	1286	14	36	0	1
y.smali		100.00000%	0	7	0	2	0	1
y0.smali		0.00000%	6	6	3	3	1	1
y1aa.smali		0.00000%	9	9	2	2	1	1
y1$a.smali		0.00000%	16	16	2	2	1	1
y1$b.smali		100.00000%	0	11	0	4	0	1
y1.smali		48.38710%	16	31	2	4	0	1
y2$a.smali		0.00000%	8	8	2	2	1	1
y2$b.smali		0.00000%	1	1	1	1	1	1
y2$c.smali		0.00000%	68	68	3	3	1	1
y2$d.smali		0.00000%	21	21	3	3	1	1
y2$e.smali		0.00000%	12	12	3	3	1	1
y2$f.smali		0.00000%	9	9	3	3	1	1
y2.smali		0.00000%	139	139	14	14	1	1
y3.smali		100.00000%	0	12	0	1	0	1
z.smali		0.00000%	23	23	2	2	1	1
z0.smali		0.00000%	6	6	3	3	1	1
z1.smali		100.00000%	0	14	0	4	0	1
z2$a.smali		8.69565%	21	23	2	3	0	1
z2$b.smali		57.14286%	3	7	2	4	0	1
z2.smali		25.92593%	80	108	6	8	0	1
z3aa.smali		0.00000%	8	8	3	3	1	1
z3ab.smali		0.00000%	13	13	3	3	1	1
z3ac.smali		0.00000%	38	38	3	3	1	1
z3$a.smali		0.00000%	18	18	3	3	1	1
z3.smali		0.00000%	146	146	15	15	1	1
Total	5097 of 18116	28.13535%	13019	18116	1189	1723	207	364

com.webascender.callerid > com.hiya.stingray.exception

granularity level: instruction

Element	Ratio	Cov.	Missed	Lines	Missed	Methods	Missed	Classes
a.smali		0.00000%	117	117	6	6	1	1
b.smali		-	0	0	0	0	0	0
c$a.smali		12.50000%	14	16	2	3	0	1
c.smali		17.14286%	29	35	3	6	0	1
d.smali		0.00000%	28	28	4	4	1	1
HiyaDbException.smali		0.00000%	11	11	5	5	1	1
HiyaGenericException.smali		0.00000%	23	23	8	8	1	1
Total	8 of 230	3.47826%	222	230	28	32	4	6

The first package has 365 files, while the second has only 7, so there are a lot of files to go through to get an idea of how the application works.

Finally, the following images show some of the decompiled code for the `com.hiya.stingray.manager.l2` class. The green lines represent executed instructions, while the lines with a transparent background are parts of the code that was not executed:

com.webascender.callerid > com.hiya.stingray.manager > l2.smali

granularity level: instruction

```
.class public Lcom/hiya/stingray/manager/l2;
.super Ljava/lang/Object;
.source "SourceFile"


.field private final a:Landroid/content/Context;
.field private final b:Lcom/hiya/stingray/manager/i1;


.method static constructor <clinit>()V
    .locals 0

.end method

.method public constructor <init>(Landroid/content/Context;Lcom/hiya/stingray/manager/i1;)V
    .locals 1

    const-string v0, "context"
    invoke-static {p1, v0}, Lkotlin/w/c/k;->g(Ljava/lang/Object;Ljava/lang/String;)V
    const-string v0, "appSettingsManager"
    invoke-static {p2, v0}, Lkotlin/w/c/k;->g(Ljava/lang/Object;Ljava/lang/String;)V
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V
    iput-object p1, p0, Lcom/hiya/stingray/manager/l2;->a:Landroid/content/Context;
    iput-object p2, p0, Lcom/hiya/stingray/manager/l2;->b:Lcom/hiya/stingray/manager/i1;
    return-void
.end method

.method public static synthetic b(Lcom/hiya/stingray/manager/l2;Lcom/hiya/stingray/manager/m3$a;Ljava/lang/Throwable;Ljava/lang/Object;)V
    .locals 0

    if-nez p4, :cond_1
    and-int/lit8 p3, p3, 0x2
    if-eqz p3, :cond_0
    const/4 p2, 0x0
    :cond_0
    invoke-virtual {p0, p1, p2}, Lcom/hiya/stingray/manager/l2;->a(Lcom/hiya/stingray/manager/m3$a;Ljava/lang/Throwable;)V
    new-instance p0, Ljava/lang/UnsupportedOperationException;
    :cond_1
    const-string p1, "Super calls with default arguments not supported in this target, function: failedToFindEvent"
    invoke-direct {p0, p1}, Ljava/lang/UnsupportedOperationException;->(Ljava/lang/String;)V
    throw p0
.end method

.method public final a(Lcom/hiya/stingray/manager/m3$a;Ljava/lang/Throwable;)V
    .locals 2

    const-string v0, "eventDestination"
    invoke-static {p1, v0}, Lkotlin/w/c/k;->g(Ljava/lang/Object;Ljava/lang/String;)V
    const/4 v0, 0x1
    new-array v0, v0, [Lkotlin/l;
    invoke-virtual {p1}, Ljava/lang/Enum;->name()Ljava/lang/String;
    move-result-object p1
    const-string v1, "destination"
    invoke-static {v1, p1}, Lkotlin/p;->a(Ljava/lang/Object;Ljava/lang/Object;)Lkotlin/l;
```

```
.method public e(Ljava/lang/String;)Ljava/util/Map;)V
    .locals 3
    .annotation system Ldalvik/annotation/Signature;
    value = {
    "(",
    "Ljava/lang/String;",
    "Ljava/util/Map<",
    "Ljava/lang/String;",
    "Ljava/lang/String;",
    ">;)V"
    }
    .end annotation

    const-string v0, "eventName"
    invoke-static {p1, v0}, Lkotlin/w/c/k;->g(Ljava/lang/Object;Ljava/lang/String;)V
    const-string v0, "attributes"
    invoke-static {p2, v0}, Lkotlin/w/c/k;->g(Ljava/lang/Object;Ljava/lang/String;)V
    iget-object v0, p0, Lcom/hiya/stingray/manager/l2;->b:Lcom/hiya/stingray/manager/i1;
    invoke-virtual {v0}, Lcom/hiya/stingray/manager/i1;->g()Z
    move-result v0
    if-nez v0, :cond_0
    new-instance v0, Landroid/os/Bundle;
    :cond_0
    invoke-direct {v0}, Landroid/os/Bundle;->()V
    invoke-interface {p2}, Ljava/util/Map;->entrySet()Ljava/util/Set;
    move-result-object p2
    invoke-interface {p2}, Ljava/util/Set;->iterator()Ljava/util/Iterator;
    move-result-object p2
    invoke-interface {p2}, Ljava/util/Iterator;->hasNext()Z
    :goto_0
    move-result v1
    if-eqz v1, :cond_1
    invoke-interface {p2}, Ljava/util/Iterator;->next()Ljava/lang/Object;
    move-result-object v1
    check-cast v1, Ljava/util/Map$Entry;
    invoke-interface {v1}, Ljava/util/Map$Entry;->getKey()Ljava/lang/Object;
    move-result-object v2
    check-cast v2, Ljava/lang/String;
    invoke-interface {v1}, Ljava/util/Map$Entry;->getValue()Ljava/lang/Object;
    move-result-object v1
    check-cast v1, Ljava/lang/String;
    invoke-virtual {v0, v2, v1}, Landroid/os/Bundle;->putString(Ljava/lang/String;Ljava/lang/String;)V
    goto :goto_0
    const-string p2, "app version"
    :cond_1
    const-string v1, "11.0.1-8647"
    invoke-virtual {v0, p2, v1}, Landroid/os/Bundle;->putString(Ljava/lang/String;Ljava/lang/String;)V
    iget-object p2, p0, Lcom/hiya/stingray/manager/l2;->a:Landroid/content/Context;
    invoke-virtual {p2}, Landroid/content/Context;->getResources()Landroid/content/res/Resources;
    move-result-object p2
    const-string v1, "context.resources"
    invoke-static {p2, v1}, Lkotlin/w/c/k;->c(Ljava/lang/Object;Ljava/lang/String;)V
    invoke-virtual {p2}, Landroid/content/res/Resources;->getConfiguration()Landroid/content/res/Configuration;
    move-result-object p2
    iget-object p2, p2, Landroid/content/res/Configuration;->locale:Ljava/util/Locale;
    const-string v1, "context.resources.configuration.locale"
    invoke-static {p2, v1}, Lkotlin/w/c/k;->c(Ljava/lang/Object;Ljava/lang/String;)V
    invoke-virtual {p2}, Ljava/util/Locale;->getDisplayCountry()Ljava/lang/String;
    move-result-object p2
    const-string v1, "country"
    invoke-virtual {v0, v1, p2}, Landroid/os/Bundle;->putString(Ljava/lang/String;Ljava/lang/String;)V
```

References

[1] BroadcastReceiver Android Class. https://developer.android.com/reference/android/content/BroadcastReceiver. [Online; accessed 06-June-2022].

[2] Ethics in Computing. https://ethics.csc.ncsu.edu/intellectual/reverse/study.php. [Online; accessed 30-May-2022].

[3] Google Android Localization API. https://developer.android.com/guide/topics/resources/localization. [Online; accessed 30-May-2022].

[4] Google Android WorkManager API. https://developer.android.com/jetpack/androidx/releases/work. [Online; accessed 30-May-2022].

[5] Protocol Buffers. https://developers.google.com/protocol-buffers/docs/overview. [Online; accessed 12-June-2022].

[6] Room. https://developer.android.com/jetpack/androidx/releases/room. [Online; accessed 19-June-2022].

[7] TelephonyManager Android Class. https://developer.android.com/reference/android/telephony/TelephonyManager. [Online; accessed 06-June-2022].

[8] Arka Bhowmik and Debashis De. mtrust: Call behavioral trust predictive analytics using unsupervised learning in mobile cloud computing. *Wireless Personal Communications*, 117:1–19, 03 2021.

[9] Sharbani Pandit, Roberto Perdisci, Mustaque Ahamad, and Payas Gupta. Towards measuring the effectiveness of telephony blacklists. 01 2018.

[10] Aleksandr Pilgun, Olga Gadyatskaya, Yury Zhauniarovich, Stanislav Dashevskyi, Artsiom Kushniarou, and Sjouke Mauw. Fine-grained code coverage measurement in automated black-box android testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(4):1–35, 2020.

[11] Chinmay R C, Mrinal Raj, Sarthak Mishra, and Shobha K. Record.ai - an ai based solution to classify calls based on conversation. In *2021 2nd International Conference on Smart Electronics and Communication (ICOSEC)*, pages 1096–1101, 2021.

[12] A. Shabtai, Y. Fledel, U. Kanonov, Yuval Elovici, and Shlomi Dolev. Google android: A state-of-the-art review of security mechanisms. *Neural Networks*, abs/0912.5, 12 2009.

[13] Audie Sumaray and S. Makki. A comparison of data serialization formats for optimal efficiency on a mobile platform. 02 2012.

[14] Chang Sung, Chi Kim, and Joo Park. Development of humming call system for blocking spam on a smartphone. *Multimedia Tools and Applications*, 76, 08 2017.