Static Analysis of Spam Call Blocking Applications

Common Android APIs Used for Call Interception and Blocking

Yoon Hwan Jeong
Supervisor(s): Dr. Apostolis Zarras, Dr. Yury Zhauniarovich
EEMCS, Delft University of Technology, The Netherlands
23-6-2022

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

In order to combat increasing spam calls, many applications are developed and downloaded to block those calls. Some studies about performance of the applications were previously conducted, however, the actual processes the applications go through to intercept and block the spam calls are not well studied. This research presents a method to systematically explore Android APIs that are responsible for intercepting and blocking the spam calls.

1 Introduction

Nowadays, it is hard to imagine living without having a smartphone. According to Ericsson, there are more than 6 billion smartphone users worldwide and they predicted that there will be more than 7 billion users by 2024 (Ericsson, 2022). As the number of users grows fast, scam calls are now more than just annoyance. Hiya reported that a person receives 16 scam calls a month on average (Hiya, 2019). Not only the number of scam calls increased, but they also became more tricky to avoid. According to First Orion, scam callers knew some personal information about their victims in 75% of cases (Orion, 2019). It gets much harder to avoid scam calls when caller ID spoofing technique is used. For example, people who fell victim to scam calls and lost more than 1000$ reported that the scam calls were from a known business (Orion, 2019).

In response to the increasing number of scam calls, numerous applications were developed and some of them are downloaded more than hundreds of millions. Despite their number of downloads shows high needs of those applications in daily life, little is known about them. Pandit et al. conducted a study to investigate multiple data sources of blocklists and analyze their effectiveness (Pandit et al., 2018). They concluded that the blocklists could block more than half of scam calls but effectiveness decreases as level of caller ID spoofing increases. There were also other attempts to analyze the applications from user experience perspective, however, it is still unclear how those applications work under the hood.

In this research, few applications are selected and analyzed to answer the following question. What Android APIs are most commonly used to intercept and block calls? This research provides the contribution to systematically extract Android APIs used in an application and answering the proposed research question provides better understanding of the applications from technical perspective and can contribute to finding any privacy and security holes that they might have. In addition to answering the question, a tool that can automatically extract Android APIs used in the applications is also developed to help analyzing a large set of applications.

2 Background

An Android application is distributed as an archive file with an `.apk` suffix. It contains codes in binary files with a `.dex` suffix, a manifest file where components are registered, and other resources such as images. Android provides following components with distinct lifecycle.

- Activities
- Services
- Broadcast receivers
- Content providers

Several Android classes serve as entry points for applications to be called by Android system. Developers interact with the system by inheriting those classes. While an application is going through its lifecycle, callbacks are invoked by the system and the application is notified about what is happening on a device. While codes written by developers and libraries included are included in DEX files, Android system codes are not present in DEX files and provided at runtime.

DEX files are bytecodes for Dalvik VM. AndroGuard parses and decompiles DEX files and provides all classes, methods, fields and strings available in those files. Whenver a class, method, field, or string is referenced somewhere else, AndroGuard builds crossreferences. This allows finding what methods are called by a method and what methods calls that method, which essentially is a call graph. However, as mentioned above, Android classes are not included in DEX files, thus, AndroGuard cannot extract methods called by Android methods.

3 Methodology

The following steps were taken to extract Android APIs used in applications using AndroGuard (AndroGuard, n.d.) as a tool to decompile and analyze Dalvik Executable (DEX) files found in Android Packages (APK files). First, any calls to APIs belong to `android.telecom` and `android.telephony` packages are extracted. These packages are responsible for managing calls. Then, only APIs that actually intercept and block calls are filtered based on their descriptions on Android API reference (Android, n.d.). From those APIs, a call graph is constructed to find other APIs used and get an overview of process. A Python program was developed to automate these steps on a set of applications and find out what Android APIs are most commonly used among them during their processes of intercepting and blocking calls.

3.1 Limitations

Although this method is enough to answer the proposed question, three limitations were found during the process. First, extracted APIs are not guaranteed to be called while an applications is running because those are extracted by a static analysis. Second, many components of an application work by inheriting classes provided by Android and defining callbacks as discussed in section 2. There is no indication in bytecode whether a method is overridden or not and any classes not defined in DEX files are represented incompletely in AndroGuard, thus, it is hard to identify if the method is written by the application developers or overrides Android API. In addition, since the callbacks are called internally by Android, there can be no incoming edges to the callbacks when constructing a call graph unless they are explicitly called within the application. In order to decrease false negative errors, if a class inherits any Android classes in the previously mentioned packages, all of its methods are extracted. This is re-

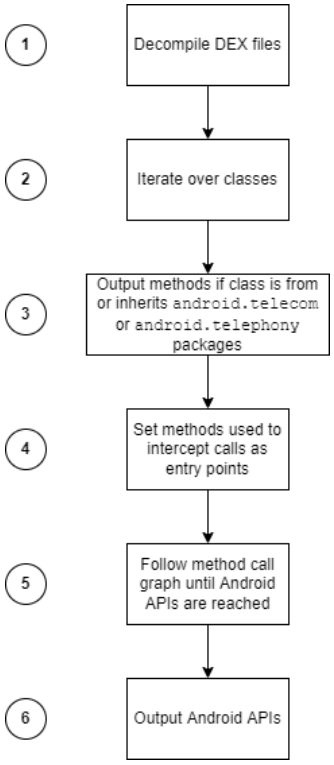

Figure 1: Flowchart of evaluation procedure

lated to the last limitation where a manual inspection is required for finding which APIs are used to intercept and block calls and checking whether methods override APIs or not. Some of these limitations can be solved by using more advanced frameworks, however, AndroGuard is chosen because of its ease of use and better performance from its naïve strategy of building cross-references.

4 Evaluation

A typical spam call blocking application works like following. First, Android system notifies the application that there is an incoming call. Then, the application decides whether to allow or reject the call based on information it has on the caller. Lastly, if the application has decided to block the call, it notifies Android system through APIs. This research is interested in finding Android APIs used in the first and last steps. 10 applications from Google Play (Google, n.d.) were analyzed as described in section 3. The complete list of applications can be found in Appendix A.

Figure 2 shows a collection of Android APIs found in DEX files of all 10 applications. These APIs are from the third step shown in Figure 1 so they can be from libraries as libraries are also included in DEX files and it is not guaranteed that the APIs used to intercept calls lead to the APIs used to block calls. Since there are multiple APIs with similar functionality, developers are free to choose which APIs they want to use in their applications. All extracted APIs per application can be found in Appendix B.

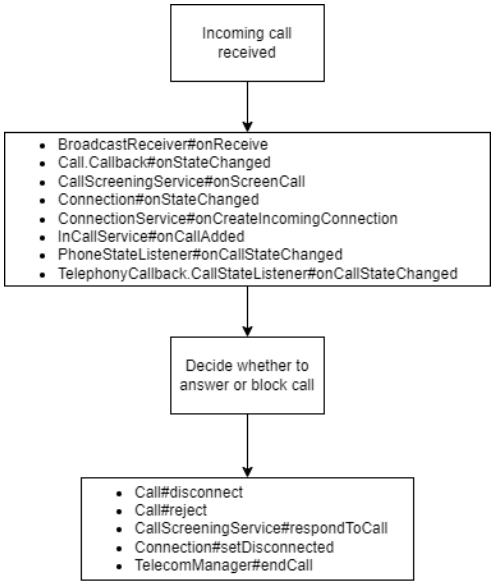

Figure 2: Overview of Android APIs used

4.1 Call Interception APIs

`BroadcastReceiver#onReceive` was found in all applications. This API can be used to intercept calls, however, it is a general-purpose API used to listen to an intent broadcast sent by Android system. Therefore, further inspections by either looking at source code or manifest are needed to conclude if it is actually used to intercept calls or not. Second most used API was `CallScreeningService#onScreenCall` which was used in 6 applications. Third most used API was `InCallSservice#onCallAdded` which was used in 5 applications. These two APIs are similar in functionality but `CallScreeningService` can allow or disallow incoming calls before they are shown to a user while `InCallService` provides more functionality for managing phone calls like an in-call user interface when the device is in a call and a means to initiate calls. `PhoneStateListener#onCallStateChanged` was used in 4 applications. This API is deprecated in API level 31 and `TelephonyCallback` was added in API level 31 to replace it. Only one applications was found to use `TelephonyCallback.CallStateListener #onCallStateChanged`. `ConnectionService #onCreateIncomingConnection` was used in one application. `Connection` can also represent VoIP (Voice over Internet Protocol) as well as typical calls.

4.2 Call Blocking APIs

All APIs mentioned above are callbacks that Android system calls so developers need to override those methods. Overrides of those methods that seem to be implemented by the applications are chosen as entry points for call graphs. During this process, packages that look like libraries and obfuscated packages are excluded and only APIs described above were considered, thus, it is possible that not all APIs shown in Figure 2 are reached via call graphs because of missed entry

points.

Most common APIs were from `CallScreeningService.CallResponse.Builder`. This class is used to build `CallResponse` which is used to call `CallScreeningService#respondToCall` and has `setDisallowCall` and `setRejectCall` that can be used to block calls. Next common API for call blocking was `Call#reject` and `Call#disconnect`. `TelecomManager#encCall` was deprecated in API level 29 and was only reachable from one application.

4.3 Other APIs

While traversing call graphs, other telecommunication related APIs were found. `Call.Details`, which is a class that holds details of a call, was used frequently. Especially, `getHandle` was used the most. Some applications also used `TelephonyManager` that provides various information about the telephony services on the device. `getSimCountryIso` and `getNetworkCountryIso` were found to be used. This can be a sign that some applications block different numbers based on the location.

5 Responsible Research

The ethical aspects and the reproducibility of the method were kept in mind while conducting this research. Regarding the ethical aspects, it was made sure that the results do not contain any closed information that can be used against those applications, since this research contains decompiling applications whose sources are not openly available.

In order to make the method reproducible, APK files and Python scripts used in this research are available on the GitHub repository. The AndroGuard version used in this research is from the master branch of AndroGuard's GitHub repository. The hash of the latest commit is `8d091cb` and was committed on November 24, 2020. Currently, AndroGuard is not actively maintained, thus, the Python scripts should remain usable unless AndroGuard is maintained again and breaking changes are made.

6 Discussion and Future Work

In some applications, only `BroadcastReceiver#onReceive` and other call intercepting APIs were extracted. This could mean that they use libraries to intercept and block calls or due to human errors while filtering entry points manually. Some applications were also found to interfere with SMS (short message service).

Since Android provides several APIs for intercepting calls, it is up to developers to choose which APIs suit their applications best. Furthermore, some applications were found to implement more than one API. There were some APIs that are deprecated as of current API level 32. However, deprecated APIs remain available and working without further official support, thus, developers can still use them but they are recommended to replace them with new APIs.

Some limitations discussed in subsection 3.1 can be improved. For example, Android APIs not being included in DEX files unless explicitly referenced can be fixed by also loading and decompiling Android JAR. This will allow identifying overridden methods systematically. For more accurate static analysis of callbacks, FlowDroid (Arzt, 2017) can be used. FlowDroid models Android component lifecycles by analyzing `AndroidManifest.xml` and creating dummy entry points. This allows obtaining a complete call graph.

7 Related Works

7.1 Asynchrony-Aware Static Analysis

As mentioned in section 2, an Android application is event-driven which makes it asynchronous. Android system invokes lifecycle callbacks in specific order, however, static analysis technique used by AndroGuard does not consider asynchronous nature of the application and may result in unsound analysis. Mishra et al. presented Android inter-component control flow graph (AICCFG) which represents control flow of Android applications and accruately models asynchronous nature of callbacks (Mishra et al., 2016).

7.2 Taint Flow Analysis

One contribution this research makes is to provide security analysis and make sure applications do not leak call activities of a user to somewhere else. One approach of detecting potential information leaks is a taint analysis. Klieber et al. described a new static taint analysis that tracks both inter-component and intra-component data flow in an Android application (Klieber et al., 2014).

8 Conclusions

In this research, common Android APIs used to intercept and block calls among 10 spam call blocking applications were statically analyzed and their descriptions are briefly discussed. Figure 2 shows all traces of Android APIs that were present in DEX files. They are slightly different to APIs presented in sections subsection 4.1 and subsection 4.2 because libraries were excluded while constructing call graphs. Furthermore, some APIs related to location were also found, which can be concluded that some applications use location in order to decide whether a call is blocked or not. Although there were some limitations as discussed in subsection 3.1, possible improvements are discussed in section 6.

A List of applications used in evaluation

- Showcaller: Caller ID & Block
- CallApp: Caller ID & Recording
- Caller ID, Phone Dialer, Block
- Call Control - SMS/Call Blocker
- Stop Calling Me - Call Blocker
- Spam Call Blocker - telGuarder
- Truecaller: Caller ID & Block
- Call Blocker - Stop spam calls
- Hiya - Call Blocker, Fraud Detection & Caller ID
- Should I Answer?

B Android APIs per application

B.1 Showcaller: Caller ID & Block

- BroadcastReceiver#onReceive

B.2 CallApp: Caller ID & Recording

- BroadcastReceiver#onReceive
- CallScreeningService#onScreenCall
- InCallService#onCallAdded
- Call$Callback#onStateChanged
- PhoneStateListener#onCallStateChanged

B.3 Caller ID, Phone Dialer, Block

- BroadcastReceiver#onReceive
- SmsMessage#getTimestampMillis
- SmsMessage#getOriginatingAddress
- SmsMessage#getMessageBody
- SmsMessage#createFromPdu

B.4 Call Control - SMS/Call Blocker

- BroadcastReceiver#onReceive
- CallScreeningService#onScreenCall
- Call$Callback#onStateChanged
- InCallService#onCallAdded
- TelecomManager#getDefaultDialerPackage
- Call$Details#getCallerDisplayName
- CallScreeningService$CallResponse$Builder #setDisallowCall
- CallScreeningService$CallResponse$Builder#build
- CallScreeningService$CallResponse$Builder #setRejectCall
- PhoneNumberUtils#isEmergencyNumber
- CallScreeningSerivce#respondToCall
- CallScreeningService$CallResponse$Builder #setSkipCallLog
- CallScreeningService$CallResponse$Builder #setSilenceCall
- Call$Details#getHandle
- CallScreeningService$CallResponse$Builder #setSkipNotification
- Call#getState
- Call#disconnect
- Call#reject
- DisconnectCause#getCause
- Call#getDetails
- Call$Details#getDisconnectCause
- InCallService#setAudioRoute
- Call#answer
- Call$Details#getVideoState
- InCallService#startActivity
- Call#registerCallback
- TelephonyManager#getNetworkOperatorName
- TelephonyManager#getSimCountryIso
- TelephonyManager#getLine1Number
- TelephonyManager#getDeviceId
- TelephonyManager#getSimState
- TelephonyManager#getSimSerialNumber

B.5 Stop Calling Me - Call Blocker

- BroadcastReceiver#onReceive
- CallScreeningService#onScreenCall
- CallScreeningService$CallResponse$Builder #setRejectCall
- Call$Details#toString
- CallScreeningService$CallResponse$Builder #setDisallowCall
- CallScreeningService$CallResponse$Builder#build
- CallScreeningSerivce#respondToCall
- Call$Details#getExtras
- CallScreeningService$CallResponse$Builder #setSkipNotification
- Call$Details#getIntentExtras

B.6 Spam Call Blocker - telGuarder

- BroadcastReceiver#onReceive

B.7 Truecaller: Caller ID & Block

- BroadcastReceiver#onReceive
- InCallService#onCallAdded
- ConnectionService#onCreateIncomingConnection
- PhoneStateListener#onCallStateChanged
- CallScreeningService#onScreenCall
- ConnectionRequest#getAddress
- ConnectionRequest#getExtras
- TelephonyManager#listen
- CallScreeningService$CallResponse$Builder#build
- CallScreeningSerivce#respondToCall
- CallScreeningService$CallResponse$Builder #setDisallowCall
- CallScreeningService$CallResponse$Builder #setSkipNotification
- Call$Details#getCallDirection
- Call$Details#getAccountHandle
- Call$Details#getIntentExtras
- PhoneAccountHandle#getComponentName
- Call$Details#getHandle

B.8 Call Blocker - Stop spam calls

- BroadcastReceiver#onReceive
- PhoneStateListener#onCallStateChanged
- TelephonyManager#listen
- SubscriptionInfo#getSubscriptionId
- SubscriptionManager#getActiveSubscriptionInfoList
- TelephonyManager#createForSubscriptionId
- TelecomManager#endCall
- TelephonyManager#getSimCountryIso

B.9 Hiya - Call Blocker, Fraud Detection & Caller ID

- BroadcastReceiver#onReceive
- CallScreeningService#onScreenCall
- InCallService#onCallAdded
- Call$Callback#onStateChanged
- Call#answer
- Call#reject
- PhoneNumberUtils#normalizeNumber
- Call$Details#getCallDirection
- CallScreeningService$CallResponse$Builder#build
- CallScreeningService$CallResponse$Builder #setDisallowCall
- CallScreeningSerivce#respondToCall
- Call$Details#getCallerNumberVerificationStatus
- Call$Details#getHandle
- Call$Details#getHandlePresentation
- Call#getDetails
- Call#getState
- Call#registerCallback
- SmsMessage#getDisplayMessageBody
- SmsMessage#getOriginatingAddress
- DisconnectCause#getReason
- Call$Details#getDisconnectCause

B.10 Should I Answer?

- BroadcastReceiver#onReceive
- PhoneStateListener#onCallStateChanged
- CallScreeningService#onScreenCall
- InCallService#onCallAdded
- InCallService#onConnectionEvent
- Call$Callback#onStateChanged
- TelephonyManager#getNetworkCountryIso
- TelephonyManager#getSimCountryIso
- Call$Details#getCreationTimeMillis
- GatewayInfo#getOriginalAddress
- GatewayInfo#getGatewayAddress
- Call$Details#getExtras
- Call$Details#getCallerDisplayName
- CallScreeningService$CallResponse$Builder #setSkipNotification
- Call$Details#getStatusHints
- CallScreeningService$CallResponse$Builder#build
- Call$Details#getCallDirection
- TelephonyManager#isNetworkRoaming
- Call$Details#getGatewayInfo
- Call$Details#getIntentExtras
- StatusHints#getLabel
- CallScreeningService$CallResponse$Builder #setDisallowCall
- CallScreeningService$CallResponse$Builder #setSkipCallLog
- CallScreeningService$CallResponse$Builder #setRejectCall
- Call$Details#getHandle
- Call#getState
- Call#answer
- Call#reject
- Call#getDetails
- Call#registerCallback
- Call#disconnect

References

AndroGuard. (n.d.). *AndroGuard.* https://github.com/androguard/androguard.

Android. (n.d.). *Android API reference.* https://developer.android.com/reference.

Arzt, S. (2017). *Static data flow analysis for android applications* (Doctoral dissertation, Technische Universität, Darmstadt). Retrieved from http://tuprints.ulb.tu-darmstadt.de/5937/

Ericsson. (2022). *Number of smartphone subscriptions worldwide from 2016 to 2027 (in millions).* https://www-statista-com.tudelft.idm.oclc.org/statistics/330695/number-of-smartphone-users-worldwide.

Google. (n.d.). *Google Play.* https://play.google.com.

Hiya. (2019). *State of the Phone Call: Half Yearly Report 2019.* https://assets.hiya.com/public/pdf/HiyaStateOfTheCall2019H1.pdf.

Klieber, W., Flynn, L., Bhosale, A., Jia, L., & Bauer, L. (2014). Android taint flow analysis for app sets. In *Proceedings of the 3rd acm sigplan international workshop on the state of the art in java program analysis* (p. 1–6). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi-org.tudelft.idm.oclc.org/10.1145/2614628.2614633 doi: 10.1145/2614628.2614633

Mishra, A., Kanade, A., & Srikant, Y. N. (2016). Asynchrony-aware static analysis of android applications. In *2016 acm/ieee international conference on formal methods and models for system design (memocode)* (p. 163-172). doi: 10.1109/MEMCOD.2016.7797761

Orion, F. (2019). *Scam Call Trends and Projections Report.* http://firstorion.com/wp-content/uploads/2019/07/First-Orion-Scam-Trends-Report_Summer-2019.pdf.

Pandit, S., Perdisci, R., Ahamad, M., & Gupta, P. (2018). Towards Measuring the Effectiveness of Telephony Blacklists. *NDSS*.