Analysis of components in the manifest file of spam call blocking applications on Android

Colin Busropan
Supervisors: Dr. Apostolis Zarras, Dr. 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

Spam calls are a widespread problem as people receive about 14 spam calls per month on average. In response, tens of applications are available in Google's Play Store that aim to block these calls. While these apps have hundreds of millions of installations, there's a lack of research into these applications. Research has been done on the user experience of these apps, but the inner workings of the apps have not been explored. Analyzing the inner workings can provide useful insights to combat spam calls more effectively. In this paper, we analyze the elements of the AndroidManifest.xml file of a set of 10 spam call blocking applications to identify aspects that are related to the blocking of spam calls and to find commonalities among the apps. Occurrences of specific elements and their attributes are summarized in this paper. Permissions related to managing phone calls and accessing the internet were found to be common, and services were identified that screen and manage phone calls. 2 apps were found to allow other apps to use their phone number information database, but this analysis did not identify usages by other apps. A complete analysis of all available spam call blocking apps could show the general usage of these and other components.

1 Introduction

In 2021, people received about 14 spam calls per month on average (Hiya, 2022b). Unknowing victims sometimes lose thousands of euros while also suffering from the mental stress afterwards (van der Veldt, 2022). Spam calls are becoming a more prevalent problem as scammers use more sophisticated methods to automate these calls, making it easier to target more people. In the US alone, more than 68 million Americans have been estimated to have fallen for phone scams between May 2021 and May 2022, having lost almost $40 billion USD in total (Truecaller, 2022b).

In response to this problem, tens of applications are available in Google's Play Store to automatically block these malicious phone calls. However, there is not much research available on these applications. Sherman et al. (2020) analyzed spam call blocking applications but focused on the user experience. Pandit et al. (2018) performed a study on existing blocklists, which are lists containing a large number of malicious phone numbers, to gain more insight into their efficacy in blocking scam calls. Still, there is no available study that explores what applications actually use to decide if a call is malicious and whether or not applications make use of these blocklists. Most research in this area instead focuses on the techniques to block spam calls, such as the work by Salehin and Ventura (2007) and Li et al. (2018).

Questions that still remain are about what these applications have in common, what they commonly use to decide whether a phone call is spam or not, and where they get their information about a phone number from. Answering these questions could for example help future developers or researchers in creating a spam call blocker that is more effective than others. In general, it could also provide researchers with more insight into spam call-blocking applications before they perform research into other aspects of these applications.

In this research, the AndroidManifest.xml file of 10 spam call blocking applications will be investigated to find the commonalities and differences of these applications, and to find specific aspects related to the blocking of spam calls.

The paper is organized as follows. Section 2 provides background information about the components in the AndroidManifest.xml file. Section 3 discusses related research on manifest files and spam call blocking applications. Section 4 describes the methodology and the tools that were used. Section 5 reports the apps that have been analyzed and the findings for each of the analyzed components in the manifest file. Discussion of the findings has also been incorporated in Section 5. Section 6 reviews the reproducibility of this research and ethical aspects of reverse-engineering. Finally, Section 7 highlights the conclusions and explores the limitations of this work.

2 Background

2.1 APK and manifest file

An Android application is distributed in the form of an Android Package. This is an archive file with the file extension *.apk*, and is referred to as an APK.

The AndroidManifest file is an XML file within the APK that describes essential information to build and run the application on Android. In this paper, we concentrate on the following elements in the manifest file:

- **Activities** are essentially the different views of an application. All activities must be declared in the manifest file for them to be run by the system.
- **Services** are described as "a facility for the application to tell the system about something it wants to be doing in the background" and as "a facility for an application to expose some of its functionality to other applications" (Google, 2022d). An example is playing music in the background, even when the user switches to another application.
- **Broadcast receivers** are components of an app that specify to the Android system that it wants to be notified when a certain broadcast is sent, such as the device starting to charge.
- **Content providers** encapsulate data managed by the application and provide other applications structured access to it.
- **Required permissions** are those that the app requires to operate, such as camera or contacts access.
- **Declared permissions** are those that the app can define to protect its components from being accessed.
- **API Levels** indicate different versions of the Android platform. The minimum API level, the target API level,

and the maximum API level can be indicated in the manifest file to check compatibility with a device. However, indicating a maximum API level is not recommended since new Android versions are designed to be backward-compatible, and indicating a maximum may unnecessarily prevent an app from being installed even though it should be compatible.

2.2 Component attributes

Activities, broadcast receivers, content providers, and services are the 4 different types of app components in Android apps. These components can declare attributes in the manifest file and this subsection describes the attributes that were analyzed in this research. Not all attributes are present in each component type.

- **directBootAware** indicates if the component can run before the user unlocks the device.
- **enabled** indicates if the component can be instantiated by the system.
- **exported** indicates if the component is available to other apps.
- **foregroundServiceType** is an attribute for services that run in the foreground and that specify a particular use case, such as *phoneCall* or *microphone*.
- **process** indicates the name of the process in which the component should run if it is not in the default process created by the application.
- **permission** indicates the name of the permission that the calling entity must have to target the component.
- **multiprocess** indicates if an activity can run in multiple processes.
- **syncable** indicates if the data from a content provider is to be synchronized with data on a server.

Content providers can also protect their data by specifying a `readPermission` or `writePermission` that clients must have. The `grantUriPermissions` can specify if these restrictions can temporarily be overcome for one-time access.

2.3 Intent filters and actions

Intent filters and *actions* are also analyzed in this research. Activities, broadcast receivers, and services can contain intent filters to receive specific intents. An `Intent` is a messaging object that can request an action from another app component. If the contents of an intent match an intent filter, the corresponding activity or service can be started, or a broadcast can be delivered to the broadcast receiver.

An `action` is part of an intent and specifies the action that took place in case of a broadcast intent. For activities and services, it specifies the action to perform.

3 Related work

The manifest file of Android applications is analyzed in many research works. Often times it is analyzed to detect or characterize malware in applications, such as the work done by Dashevskyi et al. (2020) on apps that mine cryptocurrency.

Furthermore, the work done by Jha et al. (2017) focuses on detecting mistakes in the manifest file of more than 10,000 apps. They developed a tool that takes a manifest file as input and detects errors such as incorrect attribute values for given components.

With respect to the research on spam call blocking, Pandit et al. (2018) analyzed multiple data sources with phone numbers that were reported as spam and analyzed the effectiveness of those data sources in blocking future unwanted calls. They also compared them to publicly available blacklists from spam call blocking applications *Truecaller* (Truecaller, 2022a) and *Youmail* (Youmail, 2022), but did not dive deeper into the workings of these applications.

Sherman et al. (2020) did analyze spam call blocking applications but focused on the efficacy of the user interface and user experience in alerting users of spam calls. Although some activities were investigated, they did not analyze the properties of activities in the manifest file, nor that of other components.

4 Analysis of the AndroidManifest file

The first step before analyzing the manifest file of an app is to obtain its APK file. There are various websites available to obtain APK's and in this research, *apkcombo.com* is used. The manifest file of an app can then be obtained from its APK file. For the analysis of the APK, Androguard (Desnos et al., 2018) is used, which is a tool to interact with Android files in an interactive Python shell. To manually inspect the manifest file, Apktool (Wiśniewski & Tumbleson, 2022) is used to obtain the decompiled file, although the raw content is also available through Androguard. The former was used for the convenience of getting the file directly instead of creating a file manually from the raw content. Furthermore, the Android documentation (Google, 2022a), other third-party documentation, Play Store information, and information on the websites corresponding to the apps, were used to understand the purpose of the obtained data.

5 Findings

5.1 App selection

10 Android apps were selected by searching for apps that mention terms related to 'spam call blocker' in Google's Play Store in April 2022. The apps selected are shown in Table 1 with their unique package name, since there could be multiple apps with a similar app name in the Play Store. Some apps were not selected due to them being paid apps, having solely a manual blocklist, or being exclusive to customers of a certain phone carrier. The number of downloads indicated in the Play Store and the ratings given by users of these apps are also summarized in Table 1.

The following findings in this subsection were obtained from the websites of the apps to identify data sources containing phone number information that the apps may use or provide. A closer look can then be given to specific instances of components in the analysis of the manifest file.

Information on *Hiya*'s website (Hiya, 2022a) showed that any developer could integrate some of *Hiya*'s spam call blocking functionalities in their app. However, none of the

App	Downloads	Rating (_/5.0)	`versionCode`	`versionName`	Package name
Showcaller	10M+	4.3	266	2.2.5	com.allinone.callerid
CallApp	100M+	4.2	1806	1.806	com.callapp.contacts
Caller ID	10M+	4.4	151	1.6.5	com.callerid.block
Call Control	5M+	4.1	40142	2.12.1	com.flexaspect.android.everycallcontrol
Stop Calling Me	1M+	4.3	382	2.3.21	com.mglab.scm
telGuarder	1M+	3.7	748	1.0.41	com.telguarder
Truecaller	500M+	4.5	1221006	12.21.6	com.truecaller
Call Blocker	100K+	4.4	129	2.4	com.unknownphone.callblocker
Hiya	10M+	4.2	110001	11.0.1-8647	com.webascender.callerid
Should I Anwer?	1M+	4.4	191	0.7.191	org.mistergroup.shouldianswer

Table 1: The number of downloads, Play Store ratings, `versionCode`, `versionName`, and package name of the analyzed applications.

other apps' manifest components showed usage of *Hiya*'s functionalities.

Similarly, *Call Control* was found to provide an API and a content provider to request information about a caller (Call Control, 2022), but none of the other apps were found to make use of this. Subsection 5.7 elaborates on this content provider.

Moreover, *Should I Answer?* (ShouldIAnswer, 2022), *telGuarder* (telGuarder, 2022), and *Call Blocker* (unknownphone.com, 2022) allow anyone to see a report about a phone number on their website to assess the trustworthiness, but they were not found to provide an API.

5.2 Required permissions

The number of permissions required by each app is shown in Table 4, with *Truecaller* requiring the most permissions.

All analyzed apps used the permissions in Table 2. `ACCESS_NETWORK_STATE` and `INTERNET` are required permissions to access the internet, which are for example necessary to retrieve (up-to-date) blocklists or ads. The `CALL_PHONE`, `READ_CALL_LOG` and `READ_PHONE_STATE` permissions are required in order to make new calls, view previous calls, and monitor the status of calls and cellular network information. `READ_CONTACTS` is also necessary to view the user's contacts and know which contacts should never be blocked. `RECEIVE_BOOT_COMPLETED` allows the apps to start running as soon as the system has finished booting up, and as such can start receiving phone calls. `WAKE_LOCK` is in place to prevent the phone from sleeping, for instance during a call. Lastly, the `BIND_GET_INSTALL_REFERRER_SERVICE` permission allows installation information to be retrieved from the Play Store, such as the date of installation and version.

Permission
android.permission.ACCESS_NETWORK_STATE
android.permission.CALL_PHONE
android.permission.INTERNET
android.permission.READ_CALL_LOG
android.permission.READ_CONTACTS
android.permission.READ_PHONE_STATE
android.permission.RECEIVE_BOOT_COMPLETED
android.permission.WAKE_LOCK
com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE

Table 2: The permissions that were required by all apps.

Other permissions were also required frequently. 9 out of the 10 apps declared the `BILLING` permission, enabling the apps to sell digital products and content. The *CallApp* and *telGuarder* app were the only apps that surprisingly did not require the `ANSWER_PHONE_CALLS` permission. The `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION` permission were required by respectively 4 and 3 apps, as seen in Table 3. Interestingly, *Showcaller* did not require location permissions, even though its privacy policy indicates that the location of the device can be collected.

	ACCESS_COARSE_LOCATION	ACCESS_FINE_LOCATION
CallApp	X	X
Call Control	X	
telGuarder	X	
Truecaller	X	X
Hiya		X

Table 3: The location permissions required by the apps. The other 5 apps did not require these permissions.

Furthermore, the *Truecaller* app required camera access, which is likely due to the fact that this app also provides a chat service in which images can be sent. Some apps also serve as an SMS app and therefore require permissions to read and send SMS messages. Interestingly, the `maxSdkVersion` attribute of the `CALL_PHONE` permission for *telGuarder* was set to 27, meaning that the system will not grant this permission for devices with a higher API level.

5.3 Declared permissions

Only two apps were found to have declared permissions, *Truecaller* and *CallApp*. *Truecaller* declared 9 permissions, such as `com.truecaller.permission.USE_NUMBER_SERVICE` and `com.truecaller.permission.SDK_ACTION_HANDLER`, which is likely required for apps that make use of their mobile number verification service (Truecaller, 2022c). *CallApp* only declares `com.callapp.contacts.permission.MAPS_RECEIVE` as permission.

5.4 Targeted API Levels

All analyzed apps target an Android API level released in or after 2018. API levels 29 (2019) and 30 (2020) were targeted

	Permissions	Services							Broadcast receivers					
App	amount	amount	permissions	`direct BootAware`	`enabled`	`exported`	`foreground ServiceType`	external process	amount	permissions	`direct BootAware`	`enabled`	`exported`	external process
Showcaller	34	30	12	3	28	3	0	2	15	2	0	10	2	0
CallApp	52	36	20	3	34	4	2	0	20	3	0	15	6	0
Caller ID	34	27	8	2	25	4	0	2	18	3	0	13	1	0
Call Control	29	25	10	2	24	6	0	0	21	3	1	16	3	1
Stop Calling Me	37	17	5	1	15	3	0	0	14	1	0	9	3	0
telGuarder	15	9	3	5	9	1	0	0	12	0	0	7	0	0
Truecaller	79	72	33	1	69	10	4	1	60	10	2	54	8	0
Call Blocker	15	9	3	1	9	2	0	0	11	1	0	6	1	0
Hiya	25	23	13	3	22	3	0	0	23	4	0	17	6	0
Should I Answer?	16	12	4	2	12	3	1	0	10	3	1	10	4	0

Table 4: Summary of the number of instances of permissions, services, and broadcast receivers present in each app, with the number of instances of attributes appearing in those components. Specifically, the following is indicated: total number of required permissions, the number of instances with the `exported`, `directBootAware`, and `enabled` attribute set to `true`, and the number of services that specified a `foregroundServiceType` or run in a non-default process.

most frequently, as can be seen in Table 5. The minimum API level was 21 (2014) for 7 apps, 23 (2015) for 2 apps, and 16 (2012) for 1 app.

App	Minimum API level	Targeted API level
Showcaller	21	31
CallApp	21	29
Caller ID	16	30
Call Control	21	29
Stop Calling Me	21	32
telGuarder	21	30
Truecaller	23	30
Call Blocker	21	28
Hiya	21	29
Should I Answer?	23	29

Table 5: The minimum and the targeted Android API level of the 10 apps.

5.5 Services

The 10 most commonly declared services are all services from external libraries. These include libraries for app usage measurements, database management, and advertisements.

The number of services and number of instances of attributes that are declared by each app are shown in Table 4, again with *Truecaller* declaring the largest amount of services. The `directBootAware` attribute was enabled for a few services, which were mainly for *Firebase* (Google, 2022c), a development platform for back-end functionalities.

7 services in total declared at least one `foregroundServiceType`, indicating a specific use case for a foreground service. *Truecaller* contained 4 services with both `phoneCall` and `microphone` as `foregroundServiceType`. The `phoneCall` service type was also present in *CallApp* and *Should I Answer?*. Lastly, *CallApp* declared a service with the `dataSync` service type.

All permissions that were required by services are shown in Table 6, with their occurrences. The most appearing permission `BIND_JOB_SERVICE` is a required permission for any job service, which are services that execute jobs. Notably, the `BIND_SCREENING_SERVICE` is a permission required for services that screen incoming calls, meaning that the service can choose to block a call by letting it go to voicemail or let the call go through to the user. This type of service also allows developers to provide a custom user interface containing identifying information of a call. Not all apps have such a service, so they may be using other means to achieve this functionality. Similarly, the `BIND_INCALL_SERVICE` is a required permission for apps that implement the `InCallService`, a service for managing phone calls. Not all apps implemented this service, which means that these apps cannot function as the default dialer/phone app.

Service permission	Occurrences
android.permission.BIND_JOB_SERVICE	79
com.google.android.gms.auth.api.signin.permission. REVOCATION_NOTIFICATION	6
android.permission.BIND_SCREENING_SERVICE	6
android.permission.BIND_INCALL_SERVICE	5
android.permission.BIND_NOTIFICATION_LISTENER_SERVICE	4
android.permission.BIND_ACCESSIBILITY_SERVICE	3
android.permission.SEND_RESPOND_VIA_MESSAGE	2
com.google.android.gms.permission.BIND_NETWORK_TASK_SERVICE	2
android.permission.BIND_REMOTEVIEWS	1
com.truecaller.permission.USE_NUMBER_SERVICE	1
android.permission.BIND_CHOOSER_TARGET_SERVICE	1
android.permission.BIND_TELECOM_CONNECTION_SERVICE	1

Table 6: All permissions required for services.

Intent filter action	Occurrences
com.google.firebase.MESSAGING_EVENT	14
android.telecom.CallScreeningService	6
android.telecom.InCallService	5

Table 7: The top 3 intent filter actions for starting a service.

5.6 Broadcast receivers

Only a handful of broadcast receivers that were defined in the manifest file appeared in more than 1 app. The only broadcast receiver declared in all apps was Google's `AppMeasurementReceiver`, which is used to monitor how users engage with the app. Furthermore, all apps except *Should I Answer?* declared broadcast receivers that prevent background work from being done unless the system broadcasts a message such as the battery not being low anymore. The following 6 broadcast receivers were related to this: `androidx.work.-impl.background.systemalarm.ConstraintProxy-$BatteryChargingProxy`, `ConstraintProxy$Battery-NotLowProxy`, `ConstraintProxy$StorageNotLowProxy`, `ConstraintProxy$NetworkStateProxy`, `Reschedule-Receiver`, and `ConstraintProxyUpdateReceiver`.

5 apps also seemed to make use of Facebook's API as they declared a broadcast receiver for when an access token expires.

All apps declared some custom broadcast receivers, but none of them declared a broadcast receiver that listens for broadcasts from another spam call blocking app.

Broadcast receivers with the frequency of attributes are summarized in Table 4. *Truecaller* again defined the most broadcast receivers, of which 2 were `directBootAware`. These components were related to *Huawei*'s API. The other 2 `directBootAware` receivers were custom broadcast receivers defined by the app.

A number of broadcast receivers were protected by permissions, which can be found in Table 8. For instance, the `BROADCAST_SMS` permission is required such that the app can be certain that an `SMS_RECEIVED` broadcast originated from the system. Similarly, the most frequent permission `com.-google.android.c2dm.permission.SEND` is for broadcast receivers facilitating communication between apps and server applications through Google Cloud Messaging.

Permissions	Occurrences
com.google.android.c2dm.permission.SEND	9
android.permission.DUMP	5
android.permission.BROADCAST_SMS	4
android.permission.INSTALL_PACKAGES	4
android.permission.BROADCAST_WAP_PUSH	3
com.truecaller.permission.PROCESS_PUSH_MSG	2
com.truecaller.permission.SDK_ACTION_HANDLER	1
com.truecaller.permission.ACTION_HANDLER	1
com.hiya.hiyaconfig.permission	1

Table 8: The permissions that broadcast senders need to have for broadcast receivers to receive the broadcast.

Table 9 lists the top 3 system event subscriptions for broadcast receivers. The `CONNECTIVITY_CHANGE` event is important to monitor the network connectivity, and the `PHONE_STATE` event monitors the phone state and will, for example, indicate if an incoming call arrives.

Android system event	Occurrences
android.intent.action.BOOT_COMPLETED	19
android.net.conn.CONNECTIVITY_CHANGE	13
android.intent.action.PHONE_STATE	10

Table 9: The top 3 system event subscriptions of broadcast receivers.

5.7 Content providers

Most of the common content providers that were found seemingly did not actually provide content and were added through dependencies on libraries. The most common content provider is Firebase' `FirebaseInitProvider`, which is actually in place to initialize its SDK (Firebase, 2016). They claim that using content providers for this purpose is advantageous since content providers are the first components initialized when an app is started. The SDK can then do its work as long as the app process is alive.

Another advantage that they claim, is manifest merging. When an Android library project defines a content provider in its own manifest file, it is merged into the final manifest file. Developers will then only need to declare a dependency to the library, and they won't have to write any code to initialize the SDK. This is also the reason why external content providers appear in the app's manifest file.

Two other common content providers, `MobileAdsInit-Provider` and `FacebookInitProvider`, likely serve a similar purpose as `FirebaseInitProvider`.

As shortly mentioned before, *Call Control* declares a content provider that can provide information about a caller. This `com.callcontrol.datashare.DataShareProvider` can be accessed by other apps to see whether a phone call or SMS message may be malicious, and what the caller ID may be (Call Control, 2019). Normally, to use content providers, applications have to specify a 'read access' permission for that content provider in their manifest (Google, 2022b). However, the developers described that this `DataShare` content provider requires a specific `<meta-data>` element to be added in the manifest instead. Therefore, a manual inspection was done in the manifest file of the other applications to find out if they made use of this content provider, but none of them were found to make use of this.

Furthermore, the number of content providers and the presence of attributes are summarized in Table 10. For some of the content providers, the `multiprocess` attribute indicated that multiple instances of the content provider are created if the app runs in multiple processes. None of the content providers, however, specify a specific process they should run in, indicated by the `process` attribute. The `directBootAware` attribute was again set mainly for *Firebase* components. The `exported` content providers for the *Call Control* app were in place for the aforementioned `DataShare` functionality, and were not protected by permissions. A number of content providers also indicated that temporary access to its data can be granted through the `grantUriPermissions` attribute.

5.8 Activities

The activities common in a number of apps were intended for functionalities of Google and Facebook such as billing, logging in, and advertisements. The `MainActivity` in the *Call-Control* app specified intent filters for 5 actions related to the `DataShare` functionality mentioned in subsection 5.7. The online documentation about these actions indicated that other activities can be accessed, such as the blocked list, lookup, or number reporting activity.

Furthermore, the top 3 intent filter actions are shown in Table 11. The `DIAL` action is intended to open the number dialer interface of an app such that the user can initialize a

Content providers											
App	amount	directBootAware	enabled	exported	grantUriPermissions	multiprocess	permission	process	readPermission	syncable	writePermission
Showcaller	5	1	5	0	1	1	0	0	0	0	0
CallApp	11	1	11	0	1	2	0	0	0	0	0
Caller ID	7	1	7	0	2	2	0	0	0	0	0
Call Control	9	1	9	2	1	2	0	0	0	0	0
Stop Calling Me	7	1	7	0	1	0	0	0	0	0	0
telGuarder	6	1	6	0	0	1	0	0	0	0	0
Truecaller	19	0	18	2	6	2	0	0	2	0	1
Call Blocker	4	0	4	0	0	1	0	0	0	0	0
Hiya	9	1	9	0	3	2	0	0	1	0	0
Should I Answer?	4	0	4	0	0	1	0	0	0	0	0

Table 10: Summary of the number of instances of content providers present in each app, with the number of instances of attributes appearing in those components. Specifically, the following is indicated: the number of instances with the `directBootAware`, `enabled`, `exported`, `grantUriPermissions`, `multiprocess`, and `syncable` attribute set to `true`, and the number of times that a `permission`, `process`, `readPermission`, and `writePermission` was declared.

call, with optionally a phone number that was supplied in the intent.

Finally, the total number of activities can be seen in Table 12, along with the number of instances of attributes. The `InCallActivity` in *Should I Answer?* was the only activity that was direct-boot aware. A small fraction of the activities were `exported`, allowing components of other apps to launch the activity. None of the activities were protected by permissions.

Intent filter action	Occurrences
android.intent.action.VIEW	35
android.intent.action.MAIN	11
android.intent.action.DIAL	8

Table 11: The top 3 intent filter actions for starting an activity.

Activities						
App	amount	directBootAware	enabled	exported	multiprocess	permissions
Showcaller	84	0	82	3	0	0
CallApp	118	0	116	4	0	0
Caller ID	38	0	36	0	0	0
Call Control	27	0	27	2	0	0
Stop Calling Me	18	0	17	1	0	0
telGuarder	28	0	28	1	0	0
Truecaller	209	0	208	15	0	0
Call Blocker	21	0	21	1	0	0
Hiya	37	0	37	2	0	0
Should I Answer?	30	1	30	0	0	0

Table 12: Summary of the number of instances of activities present in each app, with the number of instances of attributes appearing in those activities. Specifically, the following is indicated: the number of instances with the `directBootAware`, `enabled`, `exported`, and `multiprocess` attribute set to `true`, and the number of times that a `permission` was declared.

6 Responsible Research

It is important that research is transparent, truthful, and does not omit details to present 'good' results. One way to make this clear is to show that the research is reproducible, such that the results can be verified. Section 4 already described the tools that were used, but as tools can change their behavior through software updates, the version numbers of the tools are provided in this section to show what is needed to reproduce the results. The used Androguard version was 3.3.5 and the used Apktool version was 2.6.1.

Similarly, the applications that were analyzed, can also be updated which could mean that the manifest file is different from the version analyzed in this research. Therefore, these app versions are indicated in Table 1. The `versionCode` is the internal version code that is used to check the downgrade or upgrade relationship between two versions. This prevents users from downgrading their app. The `versionName` is displayed to users, for instance in the Play Store, usually to give an indication of the significance of the changes with respect to other versions. The package names are also given in Table 1 to uniquely identify the app in the Play Store.

Furthermore, it is important to discuss the ethical aspects of decompiling and reverse-engineering an application, as this is done for each app. While this research aims to provide useful information to improve spam call blocking and consequently protect people from falling victim to malicious callers, the developers and the companies behind the spam call blocking applications are important stakeholders too. Decompilation or reverse-engineering can threaten the investments of companies by revealing important functionality and therefore laws exist to protect them (The European Parliament and the Council of European Union, 2009). Although in this research decompilation of the applications is performed, I believe this research does not reveal any important details as this is a rather high-level analysis of the applications.

7 Conclusions & Future work

7.1 Conclusions

The main objective of this research was to analyze the manifest file of a set of 10 spam call blocking applications to see what they have in common and what components they contain related to the blocking of spam calls. Various elements were found that are reasonable for spam call blocking applications to have. All apps required permissions to manage phone calls and to use the internet, except for 1 app that did not require the permission to answer phone calls. 6 apps were also found to declare services to screen an incoming phone call, meaning that they can choose to block a call instead of letting the call go through to the user. Moreover, a common system event that apps subscribed to was the `PHONE_STATE` event, which will be sent when the call state changes, such as the arrival of an incoming call.

Furthermore, most content providers did not actually seem to be utilized in the way content providers were designed to be, and instead were in place to initialize an SDK. However, an interesting content provider declared by *Call Control* was found, which can be accessed by other apps to query information about a phone number to get an indication of the trustworthiness of the caller. None of the apps analyzed were found to make use of this service.

Lastly, a common intent filter action for activities was in place to open the phone dialer activity of the app, such that a phone call can be initialized with a phone number supplied with the intent.

7.2 Future work

The performed analysis would be more complete if it were to be done with a larger set of applications. For example, the usage of *Call Control*'s content provider could be investigated by analyzing the manifest file of more apps. Similarly, it was found that *Hiya* offers its spam call blocking service to developers, but none of the analyzed apps were found to use this either. Analyzing a larger set of applications could highlight usages of these and possibly other services.

Additionally, this analysis could be put in a broader context if it was compared to apps that are not related to blocking spam calls. It is possible that some components found in these spam call blocking apps are more prevalent in these apps than other apps, and these components may then be interesting to investigate further.

Lastly, as this work is a high-level analysis of the elements in the manifest file, a sensible next step is to investigate the usages of the elements on a lower level. The possible usages of elements have been suggested in this research, but finding out what exactly they are used for, could be of great interest.

References

Call Control. (2019). Call control datashare [Online; accessed 06. June. 2022]. https://github.com/CallControl/Call-Control-DataShare/wiki

Call Control. (2022). Call control protect api [Online; accessed 09. June. 2022]. https://www.callcontrol.com/documentation/#call-control-protect-api

Dashevskyi, S., Zhauniarovich, Y., Gadyatskaya, O., Pilgun, A., & Ouhssain, H. (2020). Dissecting android cryptocurrency miners. *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*. https://doi.org/10.1145/3374664.3375724

Desnos, A., Gueguen, G., & Bachmann, S. (2018). Androguard [Online; accessed 19. May. 2022]. https://androguard.readthedocs.io/en/latest/

Firebase. (2016). How does firebase initialize on android? [Online; accessed 06. June. 2022]. https://firebase.blog/posts/2016/12/how-does-firebase-initialize-on-android

Google. (2022a). Android documentation [Online; accessed 30. May. 2022]. https://developer.android.com/reference

Google. (2022b). Content provider basics [Online; accessed 07. June. 2022]. https://developer.android.com/guide/topics/providers/content-provider-basics#RequestPermissions

Google. (2022c). *Firebase*. https://firebase.google.com/

Google. (2022d). Service [Online; accessed 19. May. 2022]. https://developer.android.com/reference/android/app/Service

Hiya. (2022a). Hiya protect [Online; accessed 08. June. 2022]. https://www.hiya.com/products/protect

Hiya. (2022b). State of the call - 2022 [Online; accessed 22. Apr. 2022]. https://f.hubspotusercontent30.net/hubfs/6751436/2022/Reports-and-Studies/State-of-the-Call-2022/2022_SOTC_report.pdf

Jha, A. K., Lee, S., & Lee, W. J. (2017). Developer mistakes in writing android manifests: An empirical study of configuration errors. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 25–36. https://doi.org/10.1109/MSR.2017.41

Li, H., Xu, X., Liu, C., Ren, T., Wu, K., Cao, X., Zhang, W., Yu, Y., & Song, D. (2018). A machine learning approach to prevent malicious calls over telephony networks. *2018 IEEE Symposium on Security and Privacy (SP)*, 53–69. https://doi.org/10.1109/SP.2018.00034

Pandit, S., Perdisci, R., Ahamad, M., & Gupta, P. (2018). Towards measuring the effectiveness of telephony blacklists. *NDSS*.

Salehin, S. M. A., & Ventura, N. (2007). Blocking unsolicited voice calls using decoys for the ims. *2007 IEEE International Conference on Communications*, 1961–1966. https://doi.org/10.1109/ICC.2007.326

Sherman, I. N., Bowers, J., McNamara Jr, K., Gilbert, J. E., Ruiz, J., & Traynor, P. (2020). Are you going to answer that? measuring user responses to anti-robocall application indicators. *NDSS*.

ShouldIAnswer. (2022). About the service [Online; accessed 09. June. 2022]. https://nl.shouldianswer.net/about

telGuarder. (2022). Protect your phone [Online; accessed 09. June. 2022]. https://www.telguarder.com/

The European Parliament and the Council of European Union. (2009). Directive 2009/24/ec of the european parliament and of the council of 23 april 2009 on the legal protection of computer programs. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32009L0024

Truecaller. (2022a). *Truecaller*. https://www.truecaller.com/

Truecaller. (2022b). Truecaller insights 2022 u.s. spam & scam report [Online; accessed 09. June. 2022]. https://truecaller.blog/2022/05/24/truecaller-insights-2022-us-spam-scam-report/

Truecaller. (2022c). Truecaller sdk documentation [Online; accessed 03. June. 2022]. https://docs.truecaller.com/truecaller-sdk/

unknownphone.com. (2022). Who is calling me? [Online; accessed 09. June. 2022]. https://www.unknownphone.com/

van der Veldt, M. (2022). Tu delft students scammed out of thousands of euros [Online; accessed 22. Apr. 2022]. https://www.delta.tudelft.nl/article/tu-delft-students-scammed-out-thousands-euros

Wiśniewski, R., & Tumbleson, C. (2022). Apktool - a tool for reverse engineering android apk files. https://ibotpeaches.github.io/Apktool/

Youmail. (2022). *Youmail*. https://www.youmail.com/