Detecting Misuse of Google Cloud Messaging in Android Badware

Mansour Ahmadi
Department of Electrical and Electronic Engineering
University of Cagliari, Italy
mansour.ahmadi@diee.unica.it

Battista Biggio
Department of Electrical and Electronic Engineering
University of Cagliari, Italy
battista.biggio@diee.unica.it

Steven Arzt
Technische Universitat Darmstadt, Germany
steven.arzt@ec-spride.de

Davide Ariu
Department of Electrical and Electronic Engineering
University of Cagliari, Italy
davide.ariu@diee.unica.it

Giorgio Giacinto
Department of Electrical and Electronic Engineering
University of Cagliari, Italy
giacinto@diee.unica.it

ABSTRACT

Google Cloud Messaging (GCM) is a widely-used and reliable mechanism that helps developers to build more efficient Android applications; in particular, it enables sending push notifications to an application only when new information is available for it on its servers. For this reason, GCM is now used by more than 60% among the most popular Android applications. On the other hand, such a mechanism is also exploited by attackers to facilitate their malicious activities; e.g., to abuse functionality of advertisement libraries in adware, or to command and control bot clients. However, to our knowledge, the extent to which GCM is used in malicious Android applications (badware, for short) has never been evaluated before. In this paper, we do not only aim to investigate the aforementioned issue, but also to show how traces of GCM flows in Android applications can be exploited to improve Android badware detection. To this end, we first extend Flowdroid to extract GCM flows from Android applications. Then, we embed those flows in a vector space, and train different machine-learning algorithms to detect badware that use GCM to perform malicious activities. We demonstrate that combining different classifiers trained on the flows originated from GCM services allows us to improve the detection rate up to 2.4%, while decreasing the false positive rate by 1.9%, and, more interestingly, to correctly detect 14 never-before-seen badware applications.

CCS Concepts

•Security and privacy → Malware and its mitigation; Mobile and wireless security; •Computing methodologies → *Supervised learning by classification;*

SPSM'16, October 24 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2994459.2994469

Keywords

Google Cloud Messaging, Android Security, Adware, Botnet, Classification, Badware Detection, Malicious

1. INTRODUCTION

Mobile applications are often developed as client interfaces for accessing services provided by remote servers. In this setting, one of the challenges for developers is to timely notify mobile applications, i.e., the clients, on any event that updates the status of the application; e.g., messaging applications like WhatsApp need to notify their clients when they receive a new message. It is clearly computationally convenient that an application is notified only when new information is available on its server (i.e., through a *push* notification), rather than frequently checking if there is a new message (i.e., using a *pull* technique). One of the most used services that allows implementing push notifications for Android applications is Google Cloud Messaging (GCM).

Thanks to its efficiency and simplicity, GCM has also attracted the attentions of attackers. In fact, there are preliminary evidences of the use of this library in several unwanted applications like adware and bots, which we generically refer to here as *badware*, based on the ENISA threat taxonomy [1]. One possible case of GCM misuse is when it is transitively used in adware as many advertisement libraries (adlibrary) use GCM. There is a belief that this type of software is not exactly badware, and the boundary between adware and free benign applications using built-in adlibraries is rather blurred than clearly defined [7]. However, users do not tend to have adware in their mobile devices. Applications displaying ads are often undesired, because they drain battery life, consume unnecessary bandwidth, and can slow down the app [21, 32]. In addition, they may also exhibit sophisticated malicious behaviors like rooting the device.[1] In addition to being used in adware, GCM exhibits a number of desirable properties for attackers, rather than pull services like HTTP, to be engaged as a command and control (C&C) channel. The potential misuse of GCM in botnets was reported by the

[1]http://www.androidauthority.com/new-android-adware-nearly-impossible-to-remove-654197/

security community in 2012 [46], but the first real variant of botnet exploiting GCM for C&C was reported by Kaspersky in 2013 [23]. Less than a year later, another report by AndroTotal discussed the interest of attackers to exploit GCM channels in a malicious manner [5]. We discuss the GCM mechanism in more detail in Section 2, along with examples of how it can be exploited in malicious Android applications.

The only existing way to thwart GCM-based attacks is blocking the app's GCM registration ID at the GCM servers. However, this requires one to first detect the badware channel, and no solution to assess the degree of suspiciousness of GCM channels has been developed yet. One possibility could be to monitor network traffic of GCM channels, and detect anomalous behaviors. Although such a solution may enable detection of 0-day (i.e., never-before-seen) botnet channels while operating at the server side, GCM messages might be encrypted to circumvent tracking and detection. This motivates our proposal of a client-side solution, presented in Section 3, in which we model the functionality of GCM regardless of the content of messages, to be effective also against message encryption. In Section 4, we empirically show that characterizing GCM services is useful to achieve a more accurate detection of bot clients, as well as unwanted adware. Our results show that the detection rate can be increased up to 2.4%, while the false positive rate can be decreased up to 1.9%.

To summarize, this paper provides the following two main contributions.

(*i*) We build a model of GCM communications to evaluate the extent to which this popular mechanism is misused in Android applications. To this end, we provide a flow analysis for GCM services to be able to automatically detect flows originated from GCM entrypoints.

(*ii*) We show how the extracted flows from GCM services can help one to more effectively detect badware using GCM, where GCM contributes in the realization of the malicious activities. Our approach for badware detection is based on machine learning and, in particular, on a multiple classifier system (MCS) architecture.

We conclude our paper by discussing related work (Section 6) and future research directions (Section 7).

2. BACKGROUND

To better understand the whole GCM mechanism and how this service can be misused by an attacker, in this section we first discuss how GCM works, and then report an example of a dissected GCM badware.

2.1 Google Cloud Messaging

Google announced Cloud-to-Device-Messaging (C2DM) system in Google I/O, 2010 as a push mechanism for Android applications. It gradually became more efficient and was renamed to Google Cloud Messaging (GCM) in Google I/O 2012[2]. The new version of GCM has received a lot of improvement such as being cross-platform (support iOS and Chrome) as well as having simplified APIs, and was re-branded to Firebase Cloud Messaging (FCM) in Google I/O 2016. Based on reports presented in Google I/O 2016, Google receives around 2 millions queries per second, and more than 1 million apps have been registered by GCM.

[2]https://developers.google.com/android/c2dm/

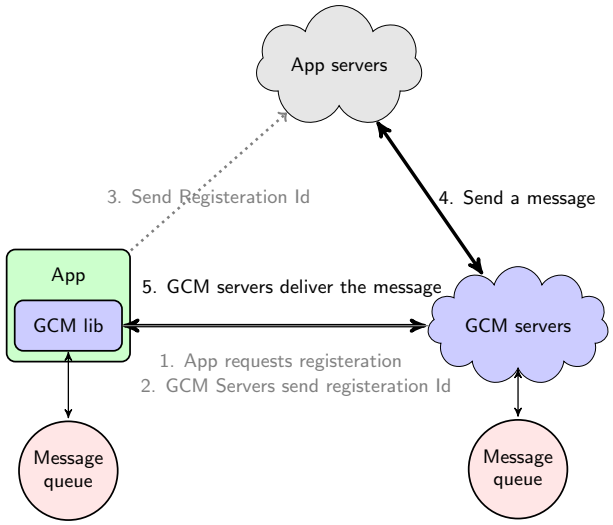


Figure 1: Google Cloud Messaging Mechanism.

Figure 1 shows how the whole mechanism works. First, the application needs to register itself to the GCM servers. After it receives a registration ID from the GCM servers, it sends the registration ID to its server for further communications. Whenever the App server needs to notify its clients, it can send data up to 4KB to a specific registration ID or a group of IDs through the GCM servers. When no Internet access is available on the client device, the messages accumulate in a queue on the server, and synchronize with the client device when it returns online. The connection protocol between App servers and GCM servers can be either HTTP or XMPP[3]. GCM provides a set of APIs for sending messages from servers to applications efficiently and reliably. These APIs can be categorized into 4 classes: *Registration*, an application needs a registration ID to communicate; *Send*, server can send messages to a particular device (registration ID); *Multi-cast*, it is possible to send messages to thousands recipients with a single request; *Time to live*, setting TTL on each request allows GCM to know when to expire a message.

2.2 GCM Badware

We describe the two possible cases of use of GCM in badware, namely, bot clients and adware, to better motivate how modeling GCM services can be beneficial in a detection system.

Botnet. Many Android bot clients use unencrypted channels like HTTP to accept messages from command and control bot masters [6]. However, the bot masters can also take advantage of secure popular public services for attacks. Three types of secure services that are frequently exploited by Android bot clients are email over SSL, GCM, and social networks (e.g., Twitter) [40]. By using these services, attackers can launch C&C attacks in a secure way, which is not easily detectable by normal TCP and HTTP traffic analysis. Furthermore, defenders cannot employ simple server blacklisting to mitigate such threats, because the email or GCM servers are used for badware as well as benign applications.

To better explain the misuse of GCM for C&C purposes,

[3]It is a persistent, asynchronous, and bidirectional connection.

Figure 2 presents a part of a decompiled Maxit backdoor sample [4]. It shows that after the bot client receives a GCM message, the content of the message is accessible through the Intent parameter of onMessage method (step 1). Then, the data is retrieved from the Intent by getExtras method and passed to Process_Message method which separates the command (step 2, 3) and executes consequent actions based on the command. After doing the action, the bot client sends an SMS to the attacker through sendTextMessage (step 5), which is located in SendSMSNow method (step 4). The SMS contains the received original GCM message, a retrieved data from SharedPreferences (e.g., IMEI) and the package name, which are split by "|". Since the attacker might receive many SMS messages, one probable answer to why the bot client sends the original GCM message along with the other information is to make the content of SMS easier for attacker to comprehend. It is obvious that as the received GCM message goes directly to sendTextMessage, there is an explicit data flow between the Intent parameter of onMessage in GCMIntentService and the sendTextMessage API. However, if the action is decided based on an if condition, the flow would be implicit. For example, in (step +), if the command equals to IMEI, the malware retrieves the identification number of the device from SharedPreferences so that there is an implicit flow between onMessage and getSharedPreferences.

Adware. Opposite of the aforementioned deliberate misuse of GCM, it might happen that GCM is exploited by badware indirectly. For instance, many adlibraries such as Revmob, Airpush, Leadbolt, Domob and Cauly [2] use GCM to notify users whenever a new advertisement has to be shown. As the purpose of adware is showing advertisement to receive benefits, these adlibraries might be embedded in adware to display unwanted ads when the user is online. As a result, GCM is unintentionally exploited as part of such malicious activities.

3. SYSTEM DESIGN

The architecture of the proposed system is depicted in Figure 3. First, we look into Android applications requesting GCM permission. Second, Flowdroid is used to extract the flows that are originated from GCM. As Flowdroid did not natively analyze GCM flows, we adapted it in order to support GCM callbacks (§3.1). Third, the output of Flowdroid is used to extract a number of features that describe the explicit flows. These features are subdivided into two sets, namely GCM and Non-GCM categories based on the corresponding services (§3.2). Accordingly, a classification function can be learnt by associating each flow to the type of applications it has been extracted from, i.e., badware or goodware. Classification is performed in different ways to verify the effectiveness of GCM features (§3.3). In the classification step, we build different models where each model provides a likelihood (between 0 and 1) denoting the degree of maliciousness of an app, which is subsequently thresholded to make the final decision on whether the application is badware or not.

3.1 Modeling GCM service

GCM base classes were not supported in FlowDroid [8] because GCM is a part of Google Play Services, and not of the Android Framework. Hence, in order to handle data flows in GCM classes, lifecycles of GCM classes have to be modeled in FlowDroid. Two common classes that have been employed in many applications in the past, are GCMBaseIntentService and GCMListenerService. The FirebaseMessagingService class has been introduced recently so it takes some time to be integrated in some applications. The GCMBaseIntentService class has been deprecated since September 2014, but there are still a lot of applications that have implemented this class. The GCM classes and methods are listed in Table 1. The methods are used for different purposes like registration, error handling, and message reception. The application needs to declare a GCMReceiver, which is a kind of BroadcastReceiver so that it delivers messages to GCM base classes. Two important methods that are called during receiving messages are onMessage in the GCMBaseIntentService class, and onMessageReceived in the GCMListenerService. Data flows from parameters of these services can represent command-action behaviors.

To model lifecycles, FlowDroid builds a custom entry point. This entry point is essentially a main() method that emulates the behavior of the Android operating system and framework. As a consequence, the data flow tracker itself can process the app as a standard Java program with a main() method, albeit it still uses the Android framework through calls to library methods. In the basic version of FlowDroid, this dummy main method contains calls the lifecycle methods of activities, services, content providers, and broadcast receivers. Our extension adds calls to the specific methods of the GCM service classes.

One could argue that the GCM base classes such as GCM Receiver are implemented as normal classes inherited from BroadcastReceiver. Therefore, correctly modeling broadcast receivers would be sufficient, because the implementation of GCMReceiver already fully specifies how and when methods such as onMessageReceived are called. With this approach, the GCM framework would be treated as part of the app and would be analyzed together with it. For performance reasons, we, however, chose a different approach. We treat the GCM framework classes as black boxes and instead add explicit models for their interfaces. In other words, we consider the GCM framework as a part of the Android operating system and abstract away from it, effectively reducing the size and complexity of the code to be analyzed.

3.2 Feature Extraction

Since FlowDroid supports detecting the desired flows, as a next step, proper sources and sinks should be provided for, and then run on various applications to extract existing flows. There are two possible ways that source of flows can be defined, i.e., parameters of callbacks, and APIs that retrieve information. As far as we aim to understand what actions are performed when the GCM callbacks are invoked, we consider callbacks as sources in our evaluation to be able to show the power of flows originated from GCM callbacks compared to the rest of Android callbacks. It is worth to mention that, although considering source APIs can provide more information about the semantics of applications, it makes the feature extraction step much slower so we avoided to use them in the proposed system. For sinks, we consider all sink APIs proposed by SUSI [34], which were extracted from Android 4.2 and contain 8,287 APIs.

After the flows are extracted by FlowDroid, they are mapped to a feature vector in which features are flows and

[4]Badware's MD5: 157febb16d16e8bcb5ba6564a2f7d320

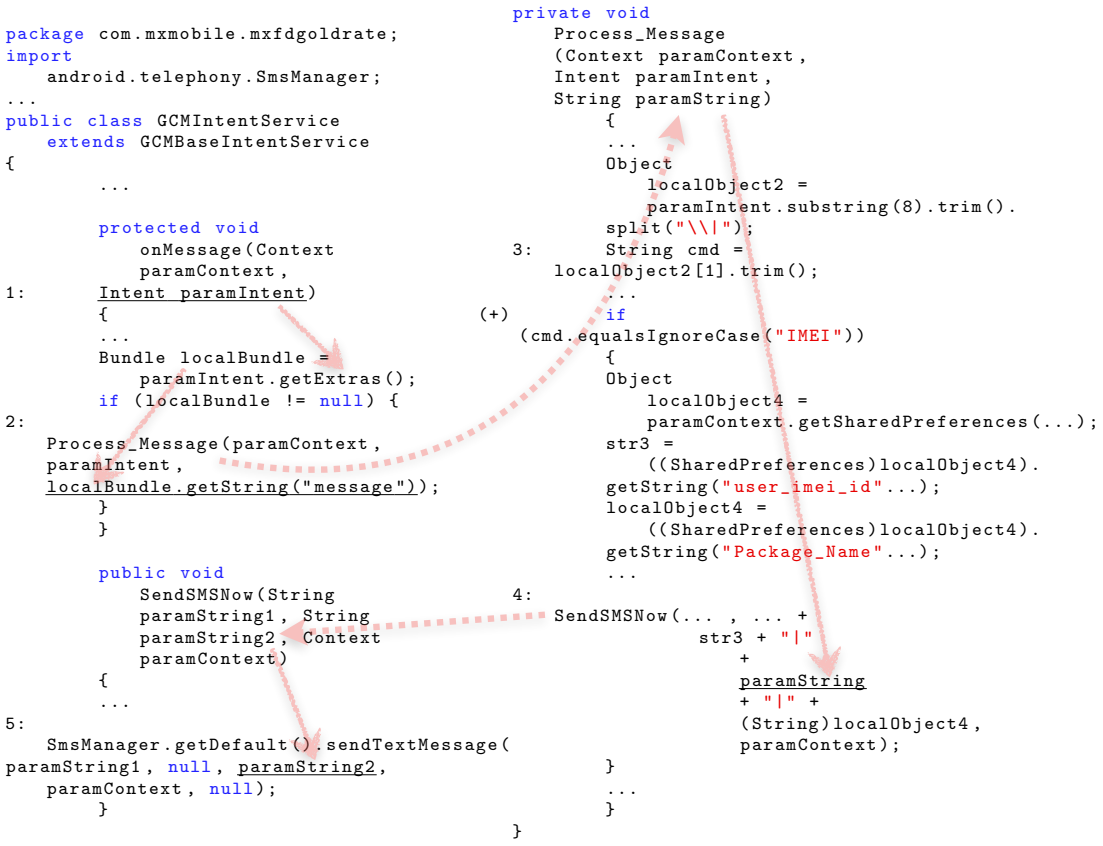


Figure 2: A part of Backdoor.AndroidOS.Maxit.a badware, which uses GCM for C&C.

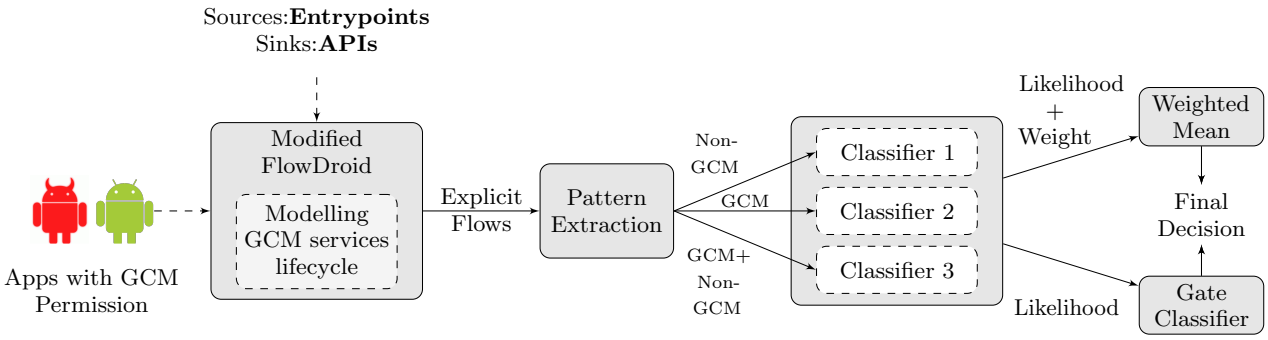


Figure 3: Overview of our approach.

Table 1: GCM services lifecycle.

Base Class	Methods
GCMBaseIntentService	void onDeletedMessages(android.content.Context,int) void onError(android.content.Context,java.lang.String) void onMessage(android.content.Context,android.content.Intent) void onRecoverableError(android.content.Context,java.lang.String) void onRegistered(android.content.Context,java.lang.String) void onUnregistered(android.content.Context,java.lang.String)
GCMListenerService (FirebaseMessagingService)	void onDeletedMessages() void onMessageReceived(java.lang.String,android.os.Bundle) void onMessageSent(java.lang.String) void onSendError(java.lang.String,java.lang.String)

their values are the total number of each flow in an application. In other words, we count how many flows there are between a specific pair of source and sink. In flows, the sources are the parameters of callbacks and the sinks are the name of APIs as well as their corresponding package. For example, in the sample in Figure 2, there is an explicit flow (by following the red lines) in which the source is `android.content.Intent` in `onMessage` method, and the sink is `sendTextMessage` in `SmsManager` package. Therefore, the feature is in the following format:

`onMessage(android.content.Intent)` $\rightsquigarrow$ `SmsManager.sendTextMessage`

We name such features as "Complete Source/Sink" in the evaluation section(§4.2). However, It is common that obfuscation techniques [36, 29] affect the name of some callbacks. Simple method renaming is applied by ProGuard, a popular tool shipped with the Android SDK. For example, we observe the fact in our experiments that `onMessage` callback has different names such as "`a`" or "`nybkaxzg`" in some applications, but the parameters type like `android.content.Intent` are intact. In addition, the sink might be package specific like `startActivity` API from `com.qihoo.psdk.app.QStatActivity` which is the name of an activity in an App. Therefore, we represented the flows to a short format in which we just consider the parameters of callbacks from the sources and API names from the sinks. Note that API method names are usually not affected by obfuscation techniques. So, for the same above example, the feature is in the following format:

`android.content.Intent` $\rightsquigarrow$ `sendTextMessage`

We call this type of features as "Abstract Source/Sink". Although the latter consideration looks loosing some information (e.g., all of `onError`, `onRegistered` or `onUnregistered` methods have `java.lang.String` parameter), we show that they can achieve better result using a smaller number of features compare to "Complete Source/Sink" representation. The underlying reason is that using a more compact (and less noisy) feature representation typically mitigates the so-called problem of *overfitting*, facilitating the task of learning an accurate classification function [11].

To evaluate the effectiveness of GCM features, we divide the features into two sets, i.e., GCM and non-GCM. If a flow is originated from GCM callbacks, we consider it as a GCM feature, otherwise, as a non-GCM feature. The following matrix shows an example of the final set of feature vectors, where each row is a feature vector for a goodware/badware, and each column is the frequency of a feature. To separate the features, we prefix them with "g" and "ng", which respectively refer to GCM and non-GCM flows.

$$\begin{Bmatrix} & g\_src_1\_snk_1 & ... & ng\_src_2\_snk_1 & ng\_src_3\_snk_4 \\ B_1 & 3 & ... & 1 & 0 \\ B_2 & 2 & ... & 4 & 1 \\ ... & ... & ... & ... & ... \\ G_1 & 1 & ... & 0 & 0 \\ ... & ... & ... & ... & ... \end{Bmatrix}$$

3.3 Classification

Regarding the nature of the task at hand, binary classification algorithms are powerful options to help us to discriminate badware from benign applications. Over the past years, a large number of classification techniques have been proposed by the scientific community, and the choice of the most appropriate classifier for a given task is often guided by previous experience in different domains, as well as by trial-and-error procedures. However, some classifiers like SVM and ensemble decision trees (e.g., Random Forest and Extra Trees [18]) have shown high performances in a variety of tasks [16].

To simplify the learning task and reduce the risk of overfitting, we exploit feature selection to reduce the feature set size by removing irrelevant and noisy features from our sets. In particular, as done in [4], we compute the so-called *mean decrease impurity* score for each feature, and retain those features which have been assigned the highest scores.[5]

We combine the obtained decisions of single models (see Fig. 3) using a multiple classifier system (MCS) [31, 24]. The underlying reason is that MCSs do not only often improve classification accuracy with respect to the combined classifiers, but also provide some degree of robustness against evasion attempts [19, 10]. One of the simplest and widely-used MCS fusion rule is the weighted average:

$$L_c = \frac{\sum_{i=1}^{n} (W_i \times L_{c_i})}{\sum_{i=1}^{n} W_i}\,, \qquad (1)$$

where n is the number of single classifiers, L is the likelihood of each single classifier, W is a weight assigned to a single classifier, and c refers to each class label. In this approach, a specific weight is assigned to each single classifier output, usually based on the performance of the classifier, and each weight is multiplied by the predicted class likelihood obtained by the classifier. Finally, the class labels are assigned based on the average of the achieved weighted likelihood. Another common MCS technique is passing the likelihood of single models to a gate classifier to make the final decision, which we call it two-tier classification technique. The gate classifier is thus trained in the same way as the flow classifiers, but its input is a feature vector whose components are the output likelihoods of the individual classifiers.

Overall, we build one classifier trained on the GCM features, one classifier trained on the non-GCM features, and a third classifier where all the features, GCM and non-GCM, are used. We observed a degree of complementarity among classifiers, as just a portion of the misclassified samples by one of the classifiers, is misclassified by the other classifiers, the rest of them being correctly classified by the other classifiers. Hence, this motivates the fusion of classification decisions by MCS techniques to combine the prediction at the score level. This fusion makes the final decision unbiased between the individual classifiers, which helps improving the final decision. Therefore, we combine the predictions of single classifiers with the weighted mean and the two-tier techniques.

4. EXPERIMENTAL ANALYSIS

In this section, we address the following research questions:

- How much discriminatory power do flows from GCM callback sources add to a badware classifier in contrast to only using non-GCM sources (§4.2)?
- Is the approach able to predict never-before-seen badware (§4.2.1)?

Before addressing these questions, we discuss the data and the experimental settings used in our evaluation (§4.1).

[5]Note that this technique is often referred to also as Gini impurity or information gain criterion.

4.1 Experimental Setup

To evaluate our approach, we have collected more than 15,000 goodware and 15,000 badware apps from McAfee and VirusTotal[6] sources. The McAfee dataset has been released to the authors on the basis of a research agreement during the period from 2014 to 2016. However, all of the gathered samples are first seen by VirusTotal between 2011 and early 2016. Since this paper focuses on analyzing the effects of modeling GCM data flows on badware detection, we filter out all apps that do not use GCM. We consider an app to use GCM if it uses the `com.google.android.c2dm.permission.RECEIVE` permission. We found that slightly less than 10% of our initial set of apps use GCM and were retained. However, checking whether the GCM permission is present is not sufficient, because the app might be overprivileged [15]. Therefore, as a complementary check, we discarded all those of applications that did not have at least one flow from a GCM-related source, i.e., a parameter of a GCM callback method. To obtain the flows, we ran `FlowDroid` for up to 10 minutes per app on a server with 64 Intel Xeon E5-4560 processor cores running at 2.7 GHz and 1 TB of memory. Note that we limited the maximum heap size allotted to FlowDroid to 250 GB. If the analysis did not complete within this time budget, the app was discarded as well. With these constraints, 1,058 benign and 1,044 badware apps remained for further analysis. Based on the naming convention[7] by VirusTotal, half of the badware are adware, and the rest are trojan.

We evaluate our approach on this set of samples through a 10-fold cross validation, to provide statistically-sound results. In this validation technique, samples are divided into 10 groups, called folds, with almost equal sizes. The prediction model is built using 9 folds, and then it is tested on the final remaining fold. The procedure is repeated 10 times on different folds to be sure that each data point is evaluated exactly once. For the data analysis, we used a laptop with a 2 GHz quad-core processor and 8GB of memory. The whole data analysis code was written in Python, and the main employed helper library is scikit-learn.[8]

Two metrics that are used for evaluating the performance of our approach are the False Positive Rate (FPR) and the True Positive Rate (TPR). FPR is the percentage of goodware samples misclassified as badware, while TPR is the fraction of correctly-detected badware samples. A Receiver-Operating-Characteristic (ROC) curve reports TPR against FPR for all possible decision thresholds.

4.2 Results

To better understand the effectiveness of our approach, we evaluate it on the set of Android applications described in Section 4.1. To recap the overall approach, we need three single classifiers to make three models on GCM flows, Non-GCM flows and the combination of GCM and Non-GCM flows. As a first step to better motivate the selection of single classifiers, we use all the three well-performed classifiers, namely SVM, Random Forest and Extra Trees (§3.3) to make five models (three single models plus two MCS models) to see which one provides a better performance. As a result, Extra Trees achieved the higher area under ROC curve so that we select it as the main classifier for the first step of classification (see Figure 4). As a consequent step, two MCS techniques, namely weighted mean (MCS-WM) and two-tier (MCS-TT) (§3.3) are applied to improve the performance. For the MCS-WM, based on the performance of single classifiers, we assign weights of one, two and three to GCM, Non-GCM and the combination of GCM & Non-GCM models respectively. For MCS-TT, the output of single classifiers are passed to a gate classifier, which is SVM in our approach.

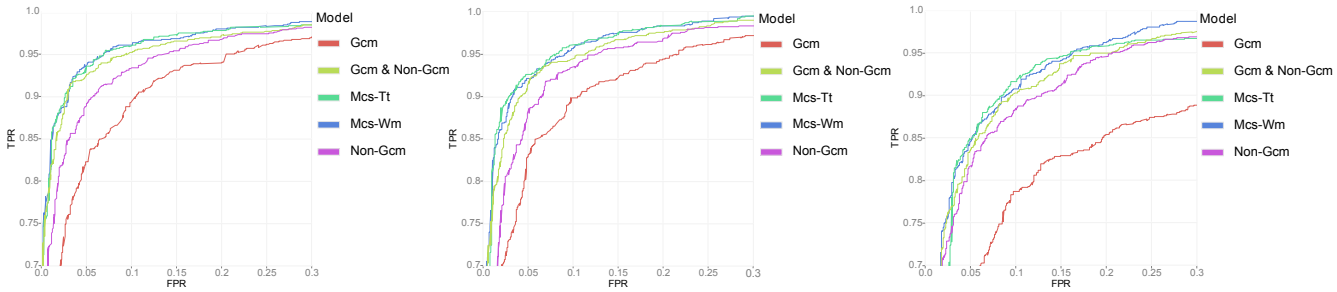
Our results are summarized in Table 2. To better discuss what we explained in Section 3.2 about the feature representation, we provide two sets of evaluations on "abstract source/sink" and "complete source/sink". In the "Measures" column, "# Features" shows the number of features used for classification while the numbers in parenthesis refer to the number of selected features. The value of each "FPR" is reported both in terms of the percentage, and in terms of the total number of misclassified goodware (in parenthesis). As is shown in the table, considering GCM-based flows alone is not a proper replacement for a traditional data flow analysis based on non-GCM flows. This is simply because GCM flows represent a small portion of the application behavior. Nonetheless, reported results clearly show that adding GCM flows to the normal flow set containing the non-GCM data flows can be helpful in detecting badware using GCM as part of the malicious behavior. In fact, when we combine GCM features with non-GCM features, they improve the performance, compared to when GCM features are ignored. In the case in which the features (GCM and non-GCM) are stacked, FPR decreases 1% and TPR is improved about 2%. Moreover, the improvement is more observable in the case of MCS-WM in which FPR and TPR respectively recover about 1.9% and 2.4%. In the case of MCS-TT, FPR decreases more, namely 2.2% while TPR has a small improvement of 1.2%. While these numbers seem small, static analyses on Android apps are usually performed on a very large scale, e.g., on complete app stores. If you consider the Google Play Store which contains over 2 million applications, improving the detection rate by 2.4% means that more than 24,000 new, previously undetected pieces of malware are discovered. Lowering the FPR by 1.9% means that 19,000 applications less are flagged as potentially malicious and, consequently, no longer need to be reviewed by human security specialists. On this scale, our proposed improvements greatly improve the state-of-the-art in Android app scanning.

Although there are still some misclassified samples (§4.2.1), we could successfully detect some bot samples based on GCM channel that were not tagged as malware by just relying on Non-GCM flows. Last but not least, the results by the proposed "abstract source/sink" features are preferable over "complete source/sink", because they need to consider a significantly lower number of features and can thus be computed more efficiently.

4.2.1 Misclassified Samples

We focus here on the proposed MCS architecture, which achieved the best results, and investigate some of the reasons behind its classification errors. As a first step, we checked again the groundtruth labels of all samples by VirusTotal three month after we gathered the last set of samples in our dataset and assigned a new groundtruth. In this way, we built the model with the original groundtruth and then checked the class of misclassified samples with the new groundtruth. In-

[6]http://www.virustotal.com
[7]https://github.com/ManSoSec/Auto-Malware-Labeling
[8]http://www.scikit-learn.org



(a) ROC of Extra Trees (b) ROC of Random Forest (c) ROC of SVM

Figure 4: ROC curves of different classifiers. The best result was achieved by Extra Tree classifier.

Table 2: Classification results of extra tree after feature selection on a set of 1058 benign and 1044 malicious apps.

Measures	Single Classifier on Flows			Multiple Classifiers	
	GCM	Non-GCM	Non-GCM + GCM	Weighted Mean	Two-Tier
Abstract Source/Sink : Parameter ⇝ API					
# Features	498 (74)	7,539 (1,162)	8,037 (1,215)	-	6
TPR	89.37%	90.71%	92.72%	93.10%	91.95%
FPR	9.735%(103)	6.049%(64)	5.009%(53)	4.159%(44)	3.781%(40)
Complete Source/Sink : Method.Parameter ⇝ Package.API					
# Features	3,322 (452)	36,219 (3,892)	39,541 (4,561)	-	6
TPR	89.94%	91.19%	92.53%	93.10%	92.53%
FPR	11.437%(121)	6.805%(72)	5.482%(58)	4.537%(48)	3.686%(39)

terestingly, we noticed that 4 out of 72 misclassified badware were not labeled as malicious based on the new groundtruth. Moreover, 14 out of 44 misclassified goodware are labeled as badware based on the new groundtruth where all of the 14 samples are labeled as adware. So based on the new groundtruth, we classified 18 (4+14) unknown samples correctly. Among the rest of misclassified badware, 32 of them are adware, and the rest are trojan. The misclassified benign samples need further analysis as there might be some other 0-day badware among them because there are many samples in our analysis from 2016. Another source of misclassification can be the use of obfuscation techniques like dynamic code loading, multi-level reflection, JavaScript and packing. We did not address those techniques in this paper as the main focus of this paper is modeling GCM services as complementary features.

Furthermore, we explored the features that might contribute the most in the misclassification by computing the median of the feature values in both the sets of correctly-classified and misclassified samples. Some of the features with the highest difference in the median between the two sets are summarized in Table 3. The table shows how the classifier might be misled by reducing or adding a specific flow. To point out some of the flows, the ones from GCM methods to `notify` and `Log.v` APIs have higher values in the undetected malware and lower values in the misclassified goodware. It is worth mentioning that the total number of flows alone cannot be representative of the class of applications because both the goodware and badware with almost the same number of flows are present in our dataset (see Figure 5). However, there are some goodware that contain higher number of flows and the fact is observable in the figure in the range of 10^3 and 10^4.

Table 3: Features that contribute the most in misclassification. Minus/plus refers to reduction/addition of a feature.

+/-	Feature
From correctly-classified goodware to false positives	
+	non-GCM : android.content.Intent ⇝ putExtra
+	non-GCM : android.os.Bundle ⇝ putString
+	non-GCM : android.os.Bundle ⇝ onCreate
-	GCM : android.content.Context ⇝ notify
-	GCM : android.content.Context ⇝ v
From true positives to undetected badware	
+	GCM : android.content.Context ⇝ notify
+	GCM : android.content.Context ⇝ v
-	non-GCM : android.os.Bundle ⇝ onCreate
-	non-GCM : android.content.Intent ⇝ putExtra
-	non-GCM : android.view.KeyEvent ⇝ onKeyDown

4.3 Discriminative Patterns

To be more informative, it is worth describing some of the important GCM features that contributed the most in the classification. Figure 6 shows the top 20 sink APIs that are performed after a GCM message received in the device and the message in a way passes to those APIs. Each of those APIs can reveal some useful information about an action that might reveal a sign about a malicious activity. As an example, 25 badware samples execute the `sendTextMessage` method based on the content of received GCM messages while no goodware shows this dependency between a received GCM message and an outgoing text message. Other kinds of suspicious actions are `openConnection` and `setRequestMethod` that exist in higher number of badware samples compared to goodware. These patterns can model downloading payload or performing DDoS attacks as the flow can show a command-

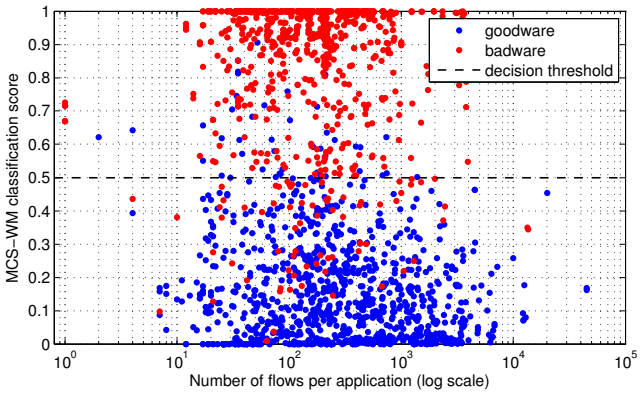


Figure 5: The realationship between the total number of flows in applications and the classification score.

and-control structure. The attacker sends a URL from the control server to his bots using GCM and the bots then perform requests against the received URL.

In adware, it is hard to say how each single pattern alone can divulge a malicious activity as the same adlibrary, embedded in adware, can be used in goodware as well. However, the experiments showed that combination of GCM features with others can contribute in the detection of adware, as this links the presence of the adlibrary to other facts about the application to provide more information about the context in which the library is used.

5. LIMITATIONS

Considering that our approach is built on top of FlowDroid, our system inherits its corresponding limitations. First, it has difficulty to track API calls that are employed by reflection techniques. Second, it cannot follow flows to the native code as it is a flow analysis system for Java. Third, dynamic code loading techniques should be another issue as FlowDroid is a static analysis technique and an attacker can download a code from internet as well as load a code from a local storage, and then load it during runtime. Moreover, FlowDroid doesn't handle inter component communications. While some recent papers have partially addressed the aforementioned issues [25, 35, 26, 33], there is a need to push forward handling these issues to the next stage.

Apart from static analysis limitations, there are possible evasion techniques against machine learning like mimicry attacks. For example, if the detection system didn't consider the semantic, an attacker can simply inject some dead code to evade the detection system [41]. Although we didn't evaluate our approach against these kinds of attacks, an adversary has to modify particular flows in application to evade our system, which is not easy and needs a lot of efforts.

6. RELATED WORK

A part of the security community proposes mitigation and access control solutions to protect Android against its potential vulnerabilities, such as privilege escalation, and information leakage. On the other hand, a vast majority of researchers have focused more on Android application analysis by considering two main issues. First, many approaches aim to assess if there is a vulnerability in applications because of neglected securely design and implementation. Second, others propose detection systems against badware. As far as the main purpose of our approach is application analysis, we describe some prominent approaches in the application analysis area with a highlight on those that focus on GCM and adlibraries.

Application Vulnerability Analysis. Application vulnerabilities are identified as the number one security threat because some developers don't have enough security knowledge. Therefore, researchers have been trying to detect vulnerabilities in different components of applications. MalloDroid [14] is one of the approaches that relies on static analysis to find misuses in communications via SSL in Android applications. Android applications vulnerabilities also include SQL injection, which is one of the major vulnerabilities affecting web application. ContentScope [22] showed that SQL injection attacks can be used to extract some private data from unprotected Content Providers. As other examples of vulnerability detectors for Android applications, SEFA [42] is a tool that analyzes in-component, cross-component, and cross-Apps vulnerabilities, while Woodpecker [20] focus on the detection of just in-component vulnerabilities. CHEX [28] is one of the most sophisticated in-component and cross-component analysis tools that can detect component hijacking vulnerabilities.

Badware Analysis. There are also a quite good number of approaches that proposed different static and dynamic analysis techniques for badware classification. They are mostly based on machine learning while the difference derives from the feature extraction step. Some of these approaches vet badware detection like Drebin [7], DroidAPIMiner [3], MudFlow [9], AppAudit [43], while the others just concentrate on badware family classification like DroidScribe [13] and Dendroid [39]. Moreover, there are some systems that generalize their approach for both malware detection and classification such as DroidMiner [44] and DroidSIFT [45]. Apart from the aforementioned systems that consider badware in generic cases, there are also some researches that target some specific kinds of badware like the one that provides a solution for detection of logic bombs [17] in Android applications, or another one that extracts potentially suspicious runtime values such as premium SMS numbers or blacklisted URLs [35] to thwart evasion techniques.

Advertisement Libraries Analysis. On the evaluation of advertising libraries, there are some approaches like AdDroid [30] and AdSplit [37] that contributed on isolating advertising libraries from host applications (e.g., to create fault isolation). A common way for adversary to monetize the adlibraries is to repackage free benign applications with injected adlibraries, and then earn ad revenue as explained by Zhoe et. al [47]. They proposed an approach to decouple the core of applications from other modules (e.g., adlibraries) based on program dependency graph to detect repackaged applications. Another work [38] examines the effects on user privacy in thirteen popular Android ad providers by reviewing their use of permissions.

GCM Services Analysis. There are just a few works that analyzed GCM services for security objectives. The most prominent example is the one that reported some vulnerabilities in GCM mechanism [27] by which an adversary can steal sensitive user data of popular applications like Facebook, and command the devices. A following work, called Seminal [12], provides an automation to find vulnerabili-

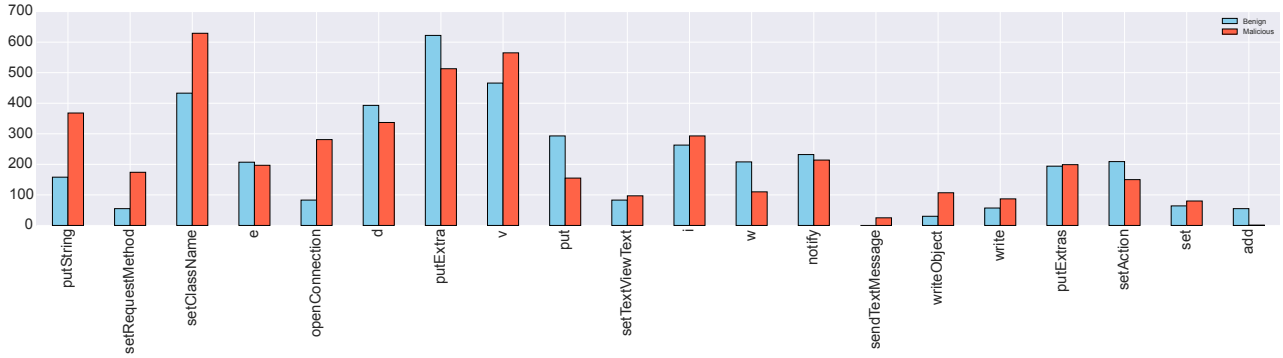


Figure 6: Top 20 discriminative actions based on mean decrease impurity. They are sinks in data flows from GCM received messages). `e`, `d`, `v`, `i`, `w` are `log` methods.

ties in applications using push notification services. Apart from vulnerability analysis, other researchers [46] showed how attackers might exploit push notification services like GCM to create a cloud-based push-styled mobile botnet. However, they didn't propose any concrete analysis/defense solution except advising either monitoring the network traffic or verifying the combination of GCM permission with others.

As an overall comparison with the previous approaches, opposite of the other works, this paper aimed to model the behaviors of GCM services in Android applications statically to more effectively discriminate badware from benign applications. Moreover, based on the best of our knowledge, we are the first to use the MCS paradigm for Android malware detection, which can help improving the performance of the single classifiers.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we modeled Google Cloud Messaging in Android applications to be able to detect flows from GCM services, which consequently helps analyzers to investigate security issues related to these services automatically. Consideration of the GCM services is important because of the advent of GCM in badware where GCM acts as a C&C channel. To assure how much this consideration can be beneficial, we evaluated the effect of data flows from GCM services for badware detection. Our results indicate that GCM features help to more effectively discriminate badware using the GCM mechanism from benign applications, compared to when they are ignored. The proposed approach benefits from the MCS approach which was proved to be more resilient to evasion in computer security, so we expect the same behavior. As a future plan, it is worth to extend this work to support every kind of push services as they might be exploited more extensively (e.g., Baidu Cloud Push service was abused in a badware[9]).

8. ACKNOWLEDGMENTS

We are thankful to Giovanni Murgia for his efforts on providing preliminary experimental results, as well as our shepherd, Daniel Xiapu Luo, and the anonymous reviewers for their invaluable comments to improve the paper. Moreover, we appreciate VirusTotal's collaboration for providing access to their Android applications. The research reported in this work has been supported in part by the German Federal Ministry of Education and Research (BMBF) and by the Hessian Ministry of Science and the Arts (HMWK) within CRISP.

[9]http://b0n1.blogspot.co.uk/2015/03/remote-administration-trojan-using.html

9. REFERENCES

[1] Enisa threat taxonomy. http://goo.gl/ATLpcA.

[2] Mobile advertisement platforms. http://www.mobyaffiliates.com/mobile-advertising-networks/.

[3] Y. Aafer, W. Du, and H. Yin. *DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android*, pages 86–103. 2013.

[4] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In CODASPY, pages 183–194, 2016.

[5] AndroTotal. (another) android trojan scheme using google cloud messaging. https://goo.gl/W7ebNx, 2014.

[6] M. Aresu, D. Ariu, M. Ahmadi, D. Maiorca, and G. Giacinto. Clustering android malware families by http traffic. In *MALWARE*, pages 128–135, 2015.

[7] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[8] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, pages 259–269, 2014.

[9] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *ICSE*, pages 426–436, 2015.

[10] B. Biggio, G. Fumera, and F. Roli. Multiple classifier systems for robust classifier design in adversarial environments. *Int'l J. M. Learn. Cyb.*, 1(1):27–41, 2010.

[11] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 1st ed., Oct. 2007.

[12] Y. Chen, T. Li, X. Wang, K. Chen, and X. Han. Perplexed messengers from the cloud: Automated security analysis of push-messaging integrations. In *Comp. & Comm. Sec. (CCS)*, pages 1260–1272, 2015.

[13] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro. Droidscribe:

Classifying android malware based on runtime behavior. In *Mobile Sec. Technologies (MoST)*, 2016.

[14] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *Comp. & Comm. Sec. (CCS)*, 2012.

[15] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Comp. & Comm. Sec. (CCS)*, pages 627–638, 2011.

[16] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim. Do we need hundreds of classifiers to solve real world classification problems? *Journal of Machine Learning Research (JMLR)*, 15:3133–3181, 2014.

[17] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. TriggerScope: Towards Detecting Logic Bombs in Android Apps. In *Sec. and Privacy (SP)*, May 2016.

[18] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, 2006.

[19] G. Giacinto, F. Roli, and L. Didaci. Fusion of multiple classifiers for intrusion detection in computer networks. *Patt. Rec. Lett.*, 24(12):1795–1803, Aug. 2003.

[20] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, 2012.

[21] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. In *Int. Conf. on Software Engineering (ICSE)*, pages 100–110, 2015.

[22] Y. Jiang and Z. Xuxian. Detecting passive content leaks and pollution in android applications. *Network and Distributed System Sec. Symp. (NDSS)*, 2013.

[23] Kaspersky. Gcm in malicious attachments. https://goo.gl/zcRLQi, Aug. 2013.

[24] L. I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. J. Wiley & Sons, Inc., 2014.

[25] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Int'l Conf. on Softw. Eng. (ICSE)*, pages 280–291, 2015.

[26] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein. Reflection-aware static analysis of android apps. In *Automated Softw. Eng., Demo Track (ASE)*, 2016.

[27] T. Li, X. Zhou, L. Xing, Y. Lee, M. Naveed, X. Wang, and X. Han. Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In *Comp. & Comm. Sec. (CCS)*, pages 978–989, 2014.

[28] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Comp. & Comm. Sec. (CCS)*, pages 229–240, 2012.

[29] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Comp. Sec.*, 51(C):16–31, June 2015.

[30] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Symp. on Information, Comp. & Comm. Sec. (ASIACCS)*, pages 71–72, 2012.

[31] R. Perdisci, D. Ariu, P. Fogla, G. Giacinto, and W. Lee. Mcpad: A multiple classifier system for accurate payload-based anomaly detection. *Comput. Netw.*, 53(6):864–881, Apr. 2009.

[32] I. Prochkova, V. Singh, and J. K. Nurminen. Energy cost of advertisements in mobile games on the android platform. In *Int'l Conf. Next Generation Mobile App., Services and Tech.*, pages 147–152, Sept 2012.

[33] C. Qian, X. Luo, Y. Shao, and A. T. S. Chan. On tracking information flows through jni in android applications. In *Dependable Systems and Networks (DSN)*, pages 180–191, 2014.

[34] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *Network & Distributed System Sec. Symp. (NDSS)*, Feb. 2014.

[35] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. *NDSS*, 2016.

[36] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: Evaluating android anti-malware against transformation attacks. In *Information, Comp. & Comm. Sec. (ASIA CCS)*, pages 329–334, 2013.

[37] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: Separating smartphone advertising from applications. In *USENIX Conf. on Sec. Symp.*, pages 28–28, 2012.

[38] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries. In *Mobile Sec. Technologies (MoST)*, 2012.

[39] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Syst. Appl.*, 41(4):1104–1117, Mar. 2014.

[40] Trendmicro. Android malware use ssl for evasion. https://goo.gl/OHeThO, Sep 2014.

[41] N. Šrndic and P. Laskov. Practical evasion of a learning-based classifier: A case study. In *Sec. and Privacy (SP)*, pages 197–211, 2014.

[42] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *Comp. & Comm. Sec. (CCS)*, pages 623–634, 2013.

[43] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective real-time android application auditing. In *Sec. & Privacy (SP)*, pages 899–914, May 2015.

[44] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications. In *European Symp. Research in Comp. Sec. (ESORICS)*, pages 163–182, 2014.

[45] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Comp. & Comm. Sec. (CCS)*, pages 1105–1116, 2014.

[46] S. Zhao, P. P. C. Lee, J. C. S. Lui, X. Guan, X. Ma, and J. Tao. Cloud-based push-styled mobile botnets: A case study of exploiting the cloud to device messaging service. In *ACSAC*, pages 119–128, 2012.

[47] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of "piggybacked" mobile applications. In CODASPY, pages 185–196, 2013.