2016 IEEE Security and Privacy Workshops

A Study of Grayware on Google Play

Benjamin Andow*, Adwait Nadkarni*, Blake Bassett†, William Enck*, Tao Xie†
*North Carolina State University
{beandow, apnadkar, whenck}@ncsu.edu
†University of Illinois at Urbana-Champaign
{rbasset2, taoxie}@illinois.edu

***Abstract*—While there have been various studies identifying and classifying Android malware, there is limited discussion of the broader class of apps that fall in a *gray* area. Mobile grayware is distinct from PC grayware due to differences in operating system properties. Due to mobile grayware's subjective nature, it is difficult to identify mobile grayware via program analysis alone. Instead, we hypothesize enhancing analysis with text analytics can effectively reduce human effort when triaging grayware. In this paper, we design and implement heuristics for seven main categories of grayware. We then use these heuristics to simulate grayware triage on a large set of apps from Google Play. We then present the results of our empirical study, demonstrating a clear problem of grayware. In doing so, we show how even relatively simple heuristics can quickly triage apps that take advantage of users in an undesirable way.**

I. Introduction

Ensuring the scarcity of harmful applications (apps) within markets, such as Google Play, is essential for maintaining user confidence. While popular media frequently reports on mobile malware, recent reports indicate that the problem is not as dire as once thought [1]. One difficulty for researchers is defining what constitutes malware. Often the intent of functionality within an app is unclear. This ambiguity has lead Google to publicize a categorization of Potentially Harmful Apps (PHAs) that they actively monitor [2]. However, Google's classification focuses mostly on characteristics of malware. In contrast, the focus of this paper is *grayware*: a class of PHAs that occupy the nebulous middle ground between genuine utility and malicious behavior. Although grayware may not perform actions that can be labeled as outright illegal or malicious, its actions may still negatively impact the user in terms of privacy, performance, and user efficiency.

Existing studies on mobile app characterization focus on malware [3], [4]; however, there has been limited discussion of grayware. While there has been discussion on the privacy leakage of location and phone identifiers [5], [6], the impinging functionality is typically secondary to the app (e.g., due to advertisements). Grayware is broader than privacy leaks. PC grayware includes not only spyware, but also user annoyances. Indeed, popular media reporting on mobile app security has discussed imposter apps [7], [8], [9], spyware [10], [11], and apps that do not implement functionality that they claim [12], [13]. However, PC grayware characterizations do not directly apply to mobile devices due to changes in the underlying operating system (OS) and software distribution.

This work is motivated by two research questions: **RQ1:** *What categories of grayware are relevant for mobile device stakeholders?*; and **RQ2:** *What analysis techniques can triage mobile grayware in app markets?* Clearly, **RQ2** depends on the answer to **RQ1**. To answer **RQ1**, we begin with a grayware definition for PCs and then perform a broad survey of apps on Google Play. We identify nine prevalent classes of mobile grayware. We use this characterization to define a suite of analysis techniques to answer **RQ2**.

Grayware is subjective by its very nature. Ultimately, a human is needed to verify the results of any automated analysis of grayware. Even Google's PHA analysis includes manual review [14]. Therefore, the goal of analysis should be to *triage*. That is, we seek to use human resources as effectively as possible. We are in part motivated by prior techniques to triage mobile malware [15]. Given the subjective nature of grayware, we expect program analysis alone is insufficient. Instead, we hypothesize that *text analytics* can effectively reduce human effort when identifying grayware. For each class determined by **RQ1**, we propose heuristics that combine text analytics with program analysis for effective triage.

We simulate triage by applying our analysis techniques to a large set of Google Play apps. In doing so, we make the following broad findings: (1) text analytics is a promising approach to triage grayware in app markets, (2) grayware appears within top search results for popular topics on Google Play; (3) some grayware apps have a significant number of downloads, and therefore may impact a large number of users; (4) user ratings are an ineffective metric to triage grayware.

This paper makes the following main contributions:

- *We develop lightweight heuristics to triage grayware on mobile devices.* We vary our methodology for each heuristic, which primarily combines text analytics with lightweight static analysis of disassembled code.
- *We conduct a breadth study of grayware* on Google Play. Our findings show that grayware is a significant concern that warrants development of sophisticated algorithms for accurate detection.

The remainder of this paper presents our mobile grayware characterization, heuristics for triaging mobile grayware, and evaluation on a large set of apps from Google Play.

II. Mobile Grayware

Grayware occupies the nebulous middle ground between genuine utility and apps intending to harm the user or other

DOI 10.1109/SPW.2016.40

stakeholders. TrendMicro defines PC grayware as apps containing annoying, undesirable, or undisclosed behaviors that cannot be classified as malware [16]. Prior literature has identified apps with functionality that is not malicious, but ethically questionable, such as leaking users' location and personal data [5] and aggressive advertising [17]. Mobile grayware subsumes and goes beyond these previous characterizations.

The underlying distinction between malware and grayware is the clarity of intention. For example, an app that performs actions to directly damage or disrupt a system is clearly malicious due to its intentions to harm the user (e.g., intentionally causing data loss, stealing money, or authentication credentials). Likewise, an app that intentionally attempts to bypass security or protection mechanisms is also malware. However, the intention of apps that collect personal data or simply annoy the user falls into a gray area. Identifying grayware by analyzing app behaviors is difficult, because a behavior itself does not define intent. For example, consider a spyware app that tracks the user's location, and a child-monitoring app that contains the same functionality. Both apps are expected to have the same behavior, but the intent is different. Making this distinction often requires a human to review the overall context in which the functionality occurs.

Apps that do not satisfy the requirements of all stakeholders shall be classified as grayware. A distinctive use case is highlighted by the bring-your-own-device (BYOD) scenario, where users perform personal and business activities on the same device. In this scenario, a one-click-root app installed by an employee may function as intended, but may weaken protection around enterprise data. In contrast, a keylogger installed by the employer may violate the employee's privacy.

A. Survey Methodology

Existing classifications of grayware on PCs [18] are insufficient for mobile devices. Overall, mobile grayware differs from PC grayware due to (1) different use cases (e.g., BYOD), (2) new OS protection mechanisms, and (3) the move to a centralized app distribution model. New OS protections require clear app installation, which limits the feasibility of droppers (Section II-C). Further, the centralized app distribution model minimizes app piggybacking, which is another common installation mechanism for PC grayware. Finally, apps are treated as security principals and cannot modify the configuration of other apps, limiting hijackers (Section II-C).

These insufficiencies lead us to our first research question, which revisits the grayware classification.

> **RQ1:** *What categories of grayware are relevant for mobile device stakeholders?*

We answered **RQ1** by empirically surveying existing apps. Note that our goal was to get a broad understanding for types of mobile grayware that exist, as opposed to extracting an exhaustive classification.

Methodology: From Google Play, we retrieved a list of app metadata, which was gathered by the August 2014 Playdrone archive snapshot [19], and then we randomly selected a subset of around 40k apps. For each app, we downloaded the user reviews and queried the reviews for keywords (e.g., "scam"), applying filters based on the average user ratings, and manually triaging the results for interesting apps. We supplemented this information with various news articles focused on misbehaving apps on the app markets, covering other platforms such as Apple's App Store and Microsoft's Windows Store.

B. Categories of Mobile Grayware

We found 11 categories of grayware, each with a unique behavioral signature. Of these categories, 9 are relevant to mobile devices, while 2 are more relevant to PCs.

Of the 9 mobile grayware categories, 2 are defined with respect to installation tactics. The remaining 7 categories are defined based on runtime behaviors. While many of these category names have historical precedence in PC grayware, we describe how they are relevant to mobile devices.

1) Gray Installation Tactics: There are 2 categories of grayware that use questionable tactics to obtain installation. Apps that fall into these categories are gray, because there is potential for the user to be deceived into installing them based on the presented identity or claimed functionality. That is, the actual identity or functionality is not what the user may have expected. We first discuss *imposters*, a name referenced by popular media [7], [8], [9]. We then discuss *misrepresentors*, a name that we define to characterize installation tactics.

1. Impostors: Imposters are apps that impersonate other apps or companies to gain installation. Impostors leverage social engineering tactics to trick users into downloading these impostors by using similar titles, developer names, icons, or descriptions. Tactics can be similar to DNS typosquatting [20], which is used for website impersonation. Imposters are gray, because there may be legitimate reasons for apps with these characteristics. Determining whether or not an app is an imposter is difficult, because two apps could share very similar names and icons without attempting to be an imposter. For example, a flashlight app may have titles, icons, and descriptions similar to a well-known app without attempting to impersonate it. Note that impostors are distinct from repackaged apps [21], as impostors do not generally copy code, but simply project an outward appearance similar to a well-known app.

2. Misrepresentors: Misrepresentors are apps that falsely claim to provide functionality to users. We define two subcategories based on the type of misrepresented functionality.

Viable misrepresentors claim to provide functionality that is possible to implement. For example, in April 2014, an app claiming to be an antivirus (Virus Shield [13]) was found changing a picture from an "X" to a checkmark instead of actually scanning the system for threats. However, claiming to implement functionality that is not provided is not inherently malicious. For example, apps may have free and premium versions with similar descriptions while the free version does not necessarily implement all the mentioned functionality.

Fictitious misrepresentors claim to provide functionality that is impossible to implement with or without additional

hardware. For example, on Google Play, we located blood pressure scanners that claim to calculate the blood pressure from the touchscreen interface, which is not possible using only the touchscreen. This subcategory is gray, because such apps exist for entertainment purposes.

2) Gray Runtime Behaviors: The remaining 7 categories are categorized as such due to their runtime behaviors. We use existing PC grayware names [18], [22] for consistency.

3. Madware: Madware, or "mobile adware", are apps that aggressively display advertisements to the user after installation. Ads play a major role in incentivizing developers to offer apps for no charge. However, ads transition into the gray area when they use aggressive advertising practices that affect the user experience. Symantec defined and released a report categorizing the behaviors of madware [17]; however, their definition contains categories that cross over into spyware behavior. Therefore, we define the following criteria to classify an app as madware: (1) advertisements in the notification bar; (2) installing icons on the home screen; (3) setting the wallpaper; (4) displaying system alert windows with advertisements; (5) advertisements that continue after the user exits the app; and (6) advertisement walls (i.e., blocking app functionality).

4. Dialers: Dialers are apps that automatically make telephone calls or send SMS messages without the user's knowledge. These calls or SMS messages may be targeted towards the user's contacts to send ads (e.g., to alert friends that the user installed an app). Another class of dialers may operate with the user's knowledge to programmatically annoy friends (e.g., CatFacts). Although wireless carriers began to block SMS messages to premium-rate phone numbers [23], unauthorized use is still problematic.

5. Prank Programs: Prank programs are apps that cause system interference to annoy or irritate the user. For example, prank programs may change the user's wallpaper, add icons to the home screen, or undesirably play noises through the speaker. A common use case is to play a practical joke on a friend by installing it onto their phone when it is unattended.

6. Scareware: Scareware uses shock, anxiety, or a perceived threat to get the user to perform some arbitrary action (e.g., download or uninstall software). For example, an antivirus app may tell the users that their device is infected to persuade them to buy a premium version. If the displayed information is not true, the app enters the gray area.

7. Rooting Tools: Rooting tools are apps that allow the users to gain root privileges on their device. Even without malicious intentions, such apps may violate the security requirements of stakeholders and are therefore gray. For example, the users may desire root access on their BYOD phone and download a rooting app, but the corporation may view such root access as a security violation. Note that Google also defines rooting tools as a PHA [2].

8. Spyware/trackware: Spyware tracks the user's activities and collects information without the user's authorization. For example, spyware may track the user's web browsing habits, monitor the user's location, or log the user's keystrokes. Prior research has extensively studied mobile apps that harvest privacy sensitive data [5], [24], [25], [26] and keyloggers using third-party soft keyboards [27]. The functionality of tracking or monitoring a user does not necessitate maliciousness. For example, child-monitoring software is technically a form of spyware; but since the parent installs the software on the system, it is not inherently malicious.

9. Remote Access Tools (RATs): RATs provide functionality for the remote administration of a device. RATs are not inherently malicious, as they can provide legitimate functionality. Unlike RATs in PCs, RATs in Android are only as powerful as third-party apps, and can perform only tasks such as data management, retrieval, and tracking.

C. Less Pertinent Grayware Categories

Two PC grayware categories are less pertinent to mobile devices. These categories manifest themselves only via social engineering or by exploiting vulnerabilities.

10. Droppers: Droppers retrieve and install additional undesired apps in the background without user consent. Apps on mobile devices are generally restricted from installing other apps without the user's consent. For example, third-party Android apps cannot acquire the "INSTALL_PACKAGES" permission unless they are signed with the same key as the system image. As a result, third-party apps cannot install other apps in the background without finding an exploit in the system or by tricking the user to confirm installation through social engineering. Note that apps delivering grayware payloads via dynamic code loading are not droppers, because a new app is not installed and the code executes under the app's identity. Since the gray behaviors are being performed by the app itself, such characteristic causes the app to be identified as another grayware class (e.g., spyware).

11. Hijackers: Hijackers manipulate system or app settings to reroute the user (e.g., via browser settings, bookmarks, network proxies). Unlike traditional OSes where apps operate with the user's ambient authority, Android apps are run in sandboxes and have limited access to security-sensitive APIs of the system and other apps. Therefore, to exist on Android, a hijacker must exploit a vulnerability or feature within another app. Exploiting such vulnerabilities would result in the app to be classified as malicious.

III. Background

This section provides a brief background of Android apps, and the analytics techniques used in our heuristics.

A. Android

Android is an application-centric operating system where users perform tasks and interact with the device via apps. Android apps are distributed in archive files called Android application packages (APKs). APKs contain a variety of data that can be leveraged for analysis. This data for an app includes the app's compiled bytecode (i.e., DEX), a manifest file called the *AndroidManifest.xml*, layout files, and other resource files, such as the *strings.xml*. The manifest file contains data, such as

the components declared by the app, the permissions requested and defined by the app, and the events registered by the app to receive from the system (e.g., intent actions). Developers are urged to place strings that are displayed to the user into a resource file named *strings.xml*.

One major change emerging in mobile platforms is the shift to a centralized app distribution model (e.g., Google Play). To download or install apps, users browse these app markets to find the app that suites their goal. There are many types of metadata that the user encounters while browsing these app markets. This metadata for an app includes information such as the app's title, the app's icon, the name of the developer, a description of the app's functionality, user comments, and user reviews.

B. Text Analytics

We next introduce the text analytics techniques that we use to process and derive meaning from app metadata.

Bag-of-Words model is a representation used in natural language processing that disregards the order of words (i.e., grammatical structure), but maintains the multiplicity of the words. Techniques that leverage the bag-of-words model can account for the rearrangement of words in a string. For example, when using this model, the bags produced by the titles of "Google Chrome" and "Chrome Google" are equivalent.

Stemming is a common text-preprocessing technique that reduces words to their word stem. For example, the words "argue", "arguing", and "argues" are all reduced to "argu". Stemming is useful for preparing natural language text to be used by techniques that consider the frequency of words, as stemming reduces the variances of multiple tenses of a word.

Stopword Removal is a text-preprocessing technique to remove frequent words from a body of text (e.g., "the", "and").

Latent Dirichlet Allocation (LDA) is a generative probabilistic model that discovers topics within a collection of unlabeled text in the given dataset. A topic in LDA is a group of words that occur frequently together. LDA is known to be sensitive to noise in the dataset; thus, preprocessing the text is required.

K-means Clustering is an unsupervised classification algorithm that partitions data into k clusters. Each one of the k clusters contains a centroid, which is located at the mean of all of the points in the cluster. K-means clustering can group similar bodies of text together by converting the text to vectors and inputting the vectors to the algorithm.

IV. Triaging Techniques

Having answered **RQ1** in Section II, we next turn to analysis techniques to help triage grayware in app markets, motivated by our second research question.

> **RQ2:** *What analysis techniques can triage mobile grayware in app markets?*

To answer **RQ2**, we propose a suite of lightweight techniques to triage mobile grayware on Google Play. We design techniques to identify 7 out of the 9 categories of grayware defined in Section II. We do not propose techniques for spyware and scareware, because both have been explored on mobile devices and have had triaging techniques proposed previously [5], [25], [28], [6], [29]. Note that in this section, categories with similar methodologies are presented together and are not necessarily presented in the same order as Section II.

Identifying grayware ultimately requires some form of human review, as discussed in Section II. Looking at program behavior alone is insufficient, because doing so does not incorporate the broader context used by a human to determine the *gray-ness* of an app. We hypothesize that a combination of text analytics and program analysis can effectively reduce the number of apps that a human must review. We next discuss triage techniques for each category. For each technique, we describe the steps that a human performs for confirmation.

A. Impostors

Impostors use user-facing app metadata to impersonate popular apps (e.g., title, icon, developer name). We identify impostors in two phases. First, we compute the similarity of apps' titles and developer names. Next, we use fuzzy hashing [30] to score the similarity of the icons.

Phase 1: Title Scoring: To compare the titles of two given apps, we use the bag-of-words model and score their relatedness using the cosine similarity. We do not compute the similarity if the two apps are from the same developer, as impostor apps from the same developer are counter-intuitive. First, we convert each app's title into a bag of words. Next, we create a set out of the unique words in both bags (i.e., titles). To represent each app, we construct a vector (whose size is the set's size), in which each index corresponds to a unique word in the set. We set the value at each vector index to the number of times the corresponding word appears in the title. Afterward, we compute the cosine-similarity score of the vectors. The score is a value between 0 and 1, with 1 being a perfect match.

Phase 2: Icon Scoring: In this phase, we inspect the app pairs whose titles match beyond a certain threshold. Specifically, in our experiments in Section V-B, we search for blatant impostors, i.e., matches with similarity score 1. We download the app icons and use the ssdeep [31] implementation of fuzzy hashing to determine the similarity of the icons.

Confirmation: If the icons match, we manually compare the app descriptions for semantic similarity to confirm our results. If the descriptions are also similar within human interpretation, one of the apps is an impostor. To correctly identify the impostor, we assume well-known apps (e.g., top 50 apps in each category of Google Play) are not impostors. Note that future work can design more sophisticated text analytics to quantitatively measure semantic similarity of app descriptions.

B. Fictitious Misrepresentors, Prank Programs, and Dialers

We next discuss the heuristics for fictitious misrepresentors, dialers, and prank programs. We use topic modeling, since

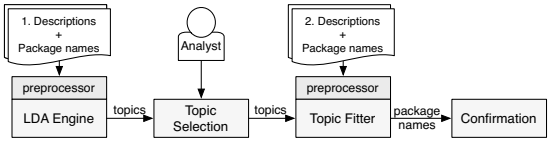

Fig. 1. The Topic-Extraction Pipeline

all of these categories rely on understanding apps' semantic topics. We first discuss the topic-extraction pipeline (shown in Figure 1) used for all three analysis techniques. Note that using topic modeling on app descriptions may not work well on apps with short descriptions, and descriptions may be intentionally designed to produce noisy results (e.g., keyword lists).

1) Topic-Extraction Pipeline (TEP): The TEP contains the following five steps for triaging the three categories of grayware: (1) preprocessing, (2) LDA topic modeling, (3) topic selection, (4) topic fitting, and (5) confirmation.

Preprocessor: The preprocessor takes app descriptions as input and outputs a set of refined descriptions. For each app description, the preprocessor performs the following steps. (1) Convert the description to lowercase and use regular expressions to remove HTML markup tags, URLs, email addresses, formatting characters, punctuation, non-ASCII characters, and stopwords. (2) Remove a list of popular words occurring within descriptions (e.g., Google, Facebook) and common words (e.g., free, app), (3) Stem the descriptions using the Snowball Stemmer [32]. (4) Remove most and least frequent words across all descriptions. Recall that preprocessing is necessary to remove noise. If many descriptions contain the same irrelevant words, LDA may incorrectly draw correlations. All lists used for analysis are available on our project website [33].

LDA-Topic Modeler: The LDA-topic modeler takes n topics and a set of refined descriptions from the preprocessor and outputs a set of topics. We use MALLET's implementation [34] of LDA with parameter values recommended by prior research [35], i.e., $\alpha = 50/n$, $\beta = 0.01$, and the number of iterations to 1000. Furthermore, to determine the optimal number of topics n to select, we iteratively vary n and maximize the logarithmic likelihood [35]. Once LDA extracts the topics, we output the top 20 words for each topic for manual topic selection.

Topic Selection: In the topic-selection phase, the analyst selects topics that represent functionality that cause an app to fall within the grayware categories. Therefore, in this phase, we manually read through the outputted topics, and select the subset of topics of interest.

Topic Fitter: The topic fitter takes the following data as input: (1) a set of descriptions and app names, (2) a set of topics, and (3) the selection threshold. Based on experimental exploration to optimize the effectiveness of confirmation, we set the selection threshold to 0.25 for all runs (i.e., there is a 25% probability that the description fits within the topic). For each description, the topic estimator uses the preprocessor to refine the set of descriptions, and estimates the probability that the description corresponds to the selected topics. If the probability is over the selection threshold, the topic estimator outputs the app name of the description for further verification.

Confirmation: The confirmation phase replicates what an analyst would do when verifying grayware. This phase consists of reading the description, examining the app's screenshots, and manually running the app if a decision cannot be made based on the first two criteria.

2) Fictitious Misrepresentors: Fictitious misrepresentors use user-facing text to claim to provide functionality that is not possible to implement without additional hardware. Our heuristic extracts topics from a training set of app descriptions that claim to implement impossible functionality (e.g., claim to be for entertainment purposes). Our heuristic then determines how a larger set of unseen apps fit within these topics.

We observed that there exist many apps that claim to be pranks or for entertainment purposes. Our intuition is that topics in these apps can be used as a baseline to identify fictitious misrepresentors. Therefore, to select our training set, based on keywords, we search for apps that include one of the following terms or phrases: "prank", "gag", "hoax", and "entertainment purpose". The training set is input to the TEP. During the topic-selection phase, we manually select topics that appear to be impossible to implement on mobile devices without additional hardware. These topics are available on our project website [33]. We pass the selected topics to the topic fitter along with a random selection of app descriptions. After the topic estimator outputs the results, we prune results that contain the keywords in our selection criteria (e.g., "gag"). The confirmation phase manually analyzes apps flagged by our heuristic to determine whether they are truly fictitious misrepresentors.

3) Prank Programs: Prank programs aim to annoy the user by interfering with the user interface or system functionality. From Section II, recall that in a multi-stakeholder environment, apps that violate the requirements of a single stakeholder can be classified as grayware (e.g., installing a prank program on a friend's device to annoy or scare them). We categorize such apps as grayware, because they outright claim to be pranks that can be installed on a victim's device, unknown to the victim, to cause fear, annoyance, or shock. They do not fulfill the requirements of the unwitting victim.

We triage prank programs by extracting the topics of apps that claim to be pranks and returning apps that cause system interference. Therefore, the training set for prank programs is identical to that of fictitious misrepresentors. Again, the training set is preprocessed and fed to the topic modeler.

The difference between the two heuristics is that the prank program heuristic is fed a different data set. Once the LDA topic modeler outputs the extracted topics, we manually select topics that appear to cause system inference and pass those topics to the topic fitter. These topics are available on our project website [33]. We pass the same set of descriptions used for training back to the topic fitter, because we want to analyze apps that explicitly claim to be pranks. In the confirmation phase, the reviewer manually analyzes the flagged apps to determine whether they are grayware.

```
Example (1). Classes from method invocations:
invoke-direct {v0}, Ljava/util/HashMap;-><init>()V
Example (2). Obfuscated class:
a/b/c/d
Example (3). Classes declared within the app (for the apk com.example):
Lcom/example/Class1
```

Fig. 2. App-Clustering Process: Code Examples

4) Dialers: Since dialers require access to sensitive interfaces (i.e., phone and SMS) to function, we select the training set by searching for apps that use the "CALL_PHONE" and "SEND_SMS" permissions. Similar to the first two heuristics, we pass the app descriptions to the preprocessor. Once the topics are outputted, we select the topics that do not appear to require phone or SMS access. These topics are available on our project website [33]. Similar to the prank-program heuristic, we pass the training set back to the topic estimator. In the confirmation phase, the reviewer manually analyzes the flagged apps to determine whether they are grayware.

C. Viable Misrepresentors

Since viable misrepresentors misrepresent the functionality that they provide, our heuristic aims to identify whether apps actually implement their claimed functionality. We collect groups of apps that claim to implement similar functionality to identify apps that do not fulfill the claim. For our study in Section V, we consider three types of apps: antiviruses, signal boosters, and performance boosters. To identify these apps, we use Google Play's search interface to find apps matching "antivirus", "signal booster", and "performance booster". While it is unknown what specific analytics algorithms Google uses, future work can explore text-clustering approaches.

For each group of functionality, we extract API-class names used in each app in the group to form feature vectors for the apps, and apply k-means clustering on the feature vectors to identify outliers; i.e., apps that do not implement the common features. The primary intuition behind this heuristic is that in order to implement a specific piece of functionality, apps need to access similar resources and use similar APIs (e.g., framework APIs). If we assume that most apps are not misrepresentors (a valid assumption if we consider top-rated apps), then by clustering apps along their API use, we can mark the outliers as grayware. Note that we choose class names instead of method names to minimize the size of the feature vector, and reduce potential noise in the clusters. Prior work has used similar techniques for clustering API invocations to detect code similarities and reuse [36].

Given a set of disassembled apps A, the clustering process is composed of the following four activities.

1. Extracting class names: We begin by extracting method invocations from the disassembled code, except those called within known ad library namespaces defined by AppBrain [37]. We extract the class names from method invocations, as highlighted in Example (1) in Figure 2.

2. Acquiring feature vector V: We consolidate the lists of class names acquired from each app into a feature vector V containing unique classes. We optimize V by filtering (1) classes that occur in fewer than 3 apps; (2) classes that are obfuscated, e.g., Example (2) in Figure 2; and (3) classes that are declared in the target app itself, e.g., Example (3) in Figure 2. For the above first group of classes being filtered, we empirically chose the threshold of 3 by examining the feature vector and finding those class names that had little impact on the app's primary functionality.

3. Initializing feature vectors v_a: Each app $a \in A$ is assigned a feature vector v_a, generated from the global feature vector V. Each element $v_a[c]$ of v_a is initialized to '0' or '1' based on the absence or presence of the class c in app a's class list.

4. Clustering using k-means & generating outliers: We perform k-means clustering using Weka [38] on the app-class matrix formed using feature vectors v_a of all $a \in A$ and V. As we need only one cluster of apps with similar functionality, we set k to 1. The outliers are apps that do not implement the common functionality. We compute the Euclidean distance of each app from the cluster's centroid. The outliers are apps farther than 2 times of the standard deviation from the centroid.

Confirmation: For every outlier app, we extract the corresponding APK and metadata from Google Play. We install the app and confirm that it fails to perform the said functionality, and also inspect the metadata for a misleading title, description, or both. In case further inspection is needed, we inspect the classes used by the outlier app, comparing it with an app close to the centroid.

D. Madware

To identify madware, we target aggressive forms of ads that escape the boundaries of the app. That is, an ad library that advertises even when the user may not be using the app can be classified as aggressive, and hence, madware. Uscilowski [17] characterized aggressive advertising on mobile phones. However, unlike Uscilowski, we only consider behaviors that visually appear to the user. Our heuristic leverages the first four aggressive advertising behaviors identified in Section II-B: (1) advertisements in the notification bar, (2) installing icons on the homescreen, (3) setting the wallpaper, and (4) displaying system alert windows with advertisements. The last two indicators are difficult to identify statically, so we exclude them from this study.

To identify ad libraries within apps, we use the list of 88 ad libraries defined by AppBrain [37]. We download and disassemble apps that include these libraries, and search for behaviors (within the ad libraries) that correspond to at least one of the four behaviors discussed above. Upon finding an aggressive behavior within the ad library, we categorize it as madware. We describe the results of the classification in Table III. For each madware library, we note the library package namespace. To identify apps as madware, we search for these namespaces in the disassembled apps. Similar techniques have been used by prior work [39].

Confirmation: The triage technique provides a list of apps that contain the madware libraries. For our study in Section V, we assume that this madware functionality is used by the apps. When deploying this technique to a real app market, apps containing madware libraries should be executed to confirm

that the madware behavior occurs before marking the app as grayware.

E. Rooting Tools and Remote Access Tools

To detect rooting tools and remote access tools (RATs), we perform keyword-based search of app descriptions for "1-click root" and "remote access tool". In the confirmation phase, the reviewer manually verifies the results. Note that this very simple heuristic is sufficient to detect rooting tools and RATs, because if any of these rooting or RAT behaviors is hidden within an app, the app would become malware.

V. Evaluation

In this section, we evaluate our grayware heuristics by simulating triage on a large set of Google Play apps. Our datasets and results are available on our project website [33].

A. Datasets

Viennot et al. [19] designed a large-scale crawler called Playdrone for Google Play. Archive.org ran the Playdrone crawler and uploaded APKs and metadata, including version codes, to the Playdrone Archive [40]. For experimental repeatability, we collect app metadata and APKs from the Playdrone Archive unless otherwise noted.

Every category of grayware described in Section II-B is different, and requires the evaluation of suitable heuristics with appropriate datasets. Therefore, in this paper, we use 4 separate datasets, each of which is used in evaluating one or more heuristic. The datasets are described as follows.

PD_1M: The PD_1M dataset consists of the metadata of 1,029,422 apps that we downloaded from the Playdrone Archive [40]. The dataset was previously collected from Google Play in October 2014.

PD_RNDM: The PD_RNDM dataset consists of the metadata of 100k randomly selected apps from the PD_1M dataset. We downloaded the APKs of these apps for analysis required to verify some heuristics (e.g., presence of madware). We successfully downloaded 99,827 APKs, of which 698 were downloaded from Google Play as they were unavailable on the PlayDrone Archive. We were unable to obtain 173 apps due to unsupported device restrictions, or the apps were removed from Google Play and not archived on PlayDrone.

GP_TOP50: To find impostors, we require popular apps that an impostor would benefit from impersonating. Hence, we collected metadata for the 50 most popular apps from Google Play, for each category (except games and comics), in both "free" and "paid" subcategories. Our dataset consists of the metadata for 2,500 apps, some of which may fall in more than one category. As we want a representative dataset for popular apps in each category, we refrain from removing overlaps.

GP_SPA: The GP_SPA dataset consists of 674 Android apps, extracted based on their proposed functionality. Of the 674 apps, there are 224 signal boosters, 236 performance boosters, and 214 antivirus apps. Each category of apps was selected from the top 250 results that were retrieved via keyword-searching Google Play for "antivirus", "signal booster", and "performance booster", respectively. Due to device restrictions and paid apps, we could not download 36 antivirus apps, 14 performance boosters, or 26 signal boosters.

Original App		*Impostor App*
The Coupons App	**Title**	The Coupons App
Most Popular Download	**Developer Name**	*Most Popular Downloads*
thecouponsapp.coupon	**Package Name**	*thecouponsapp.dailydeals*
	**Icon**	
10 - 50 million	**Downloads**	0.5 - 1 million

Fig. 3. The original Coupons app, and its impostor, which has the same title, icon, and description. The identical areas in both apps are shaded.

B. Experiments

In this section, we discuss our specific experiment setups for each heuristic and discuss the triage results. With the exception of identifying the madware libraries, all manual confirmation consists of analysis of disassembled source code.

1) Impostors: Since impostors attempt to impersonate popular apps, we use the app titles and developer names from the GP_TOP50 dataset as our representative set of popular apps. With this set, we look for impostors in the PD_1M dataset, using the heuristic described in Section IV-A.

Results: Out of the 1 million apps, our triage techniques reduced the manual review to 997 exact title matches, of which 22 app pairs had similar icons (triage reduction 1M $\rightarrow$ 22). Through manual confirmation of the matching pairs' descriptions, we found 8 impostors. While detailed results are available on our project website [33], Figure 3 shows an example of the *Coupons* app. These apps have the same title, icon, and description, but developer names differ by one character. Investigating further, we found that the impostor's developer website links to the Google Play web page of the real app. While the example demonstrates the usefulness of our technique, we acknowledge that in rare cases, a real-world developer may submit variants using different developer names. Such scenarios are beyond the scope of our study.

2) Fictitious Misrepresentors: The training set consists of 2,938 apps selected based on keyword-searching the PD_1M dataset using the criteria described in Section IV-B2. The evaluation set consists of the 100k preprocessed descriptions in the PD_RNDM dataset. We maximized the log-likelihood to get the optimal number of topics for LDA using the technique discussed by Griffiths and Steyvers [35], resulting in 650 topics. We used standard parameters for training the LDA model: $\alpha = 50/650$, $\beta = 0.01$, and 1000 iterations. During the topic-selection step of the pipeline, we manually read through the list of 650 topics outputted after the training phase and selected 32 topics that appeared to be misrepresentors.

Results: Out of the 100k apps of which the topic fitter estimated the probability, a total of 311 results were returned for manual confirmation (triage reduction 100k $\rightarrow$ 311). Out

of the 311 results, we manually verified that 18 apps claim that they implement functionality that is not possible.

Out of the 18 apps, the majority overstate the capabilities of the hardware on mobile devices (e.g., reading fingerprints from the touchscreen). The 10 apps that claim to read fingerprints from the touchscreen include lie detectors, blood pressure readers, biometric authenticators, mood detectors, and physical feature identifiers (e.g., age, appearance). During the manual inspection, we encountered 6 blood pressure readers that claim to calculate the user's blood pressure via the camera. We verified the legitimacy of these apps and our results account for this fact. The 4 apps that overstate the camera's functionality include claims of night vision, infrared detection, and x-ray vision. We found 3 apps claiming that the magnetometer can be used for detecting supernatural phenomenon. Finally, we found one app that claims to determine whether the user is sober by monitoring the accelerometer and gyroscope sensors.

3) Prank Programs: The training set and parameters for LDA are the same as the fictitious misrepresentors in Section V-B2. However, the main difference is that the training set is fed back into the topic fitter as described in Section IV-B3 During the topic-selection step of the topic-extraction pipeline, we manually read through the list of topics and select those that appear to cause system interference (i.e., 9 of 650 topics).

Results: Out of the 2,938 apps that the topic fitter estimated the probability, 104 results were returned for manual confirmation (triage reduction 1M $\rightarrow$ 104). We found that 79 out of the 104 apps appeared to cause system interference. Out of the 79 apps, 66 spoofed a cracked screen, 8 played annoying alarms, 3 popped-up computer-infection warnings, 1 popped-up fire alerts, and 1 randomly vibrated the phone in the background. We disregarded all interactive sound apps.

4) Dialers: The training set consists of 11,282 apps from the PD_1M dataset by using the selection criteria in Section IV-B4. The LDA-topic modeler runs with the following parameter values: the number of $topics = 260$, $\alpha = 50/260$, and $\beta = 0.01$. We selected topics that appeared not to be related to phone calls or SMS during the topic-selection phase. In total, we selected 4 of the 260 topics. Recall that the training set is also used for evaluation, as explained in Section IV-B4.

Results: Out of the 11,282 apps of which the topic fitter estimated the probability, a total of 58 results were returned for manual confirmation (triage reduction 1M $\rightarrow$ 58). Out of the 58 results, we manually found that 36 had legitimate reasons for requesting the "SEND_SMS" or "CALL_PHONE" permission. We could not rule out the remaining 22 apps as dialers without manually executing them, which we did not do for this study.

5) Viable Misrepresentors: To evaluate this category, we chose apps whose core functionality is executed in the background, which means users are less likely to notice whether the claimed functionality is implemented. Our evaluation set consisted of the three groups of apps from the GP_SPA dataset. For each group, we ran k-means clustering separately, as described in Section IV-C, with k=1. We considered outliers to be points more than two standard deviations from the centroid.

TABLE I
VIABLE MISREPRESENTORS STATISTICS

App Group	Total outliers	Fake apps	Misleading titles	False Positives: Dataset Selection	False Positives: Clustering
Antivirus	10	3	1	6	0
Performance Boosters	5	1	3	0	1
Signal Boosters	39	20	4	15	0

TABLE II
FAKE ANTIVIRUS APPS

Title (Package Name)	Description
Anti Virus & Mobile Security! (com.suzyapp.anti.virus.app.security)	"It checks for malware, vulnerabilities, and even cleans up trash."
Anti Virus Android (com.viruskiller.antivirusandroid545)	"This app Provides comprehensive protection for your Android phone or tablet."
Antivirus for Android (com.yoursite.afa1)	"...protects your android device from harmful viruses, malware, spyware..."

Results: Table I shows our results (triage reduction 200 $\rightarrow$ 5-39). For each group of apps, we recorded (1) the number of outliers, (2) the apps that advertised fake functionality (i.e., fake apps), (3) the apps that only advertised a misleading title, but mentioned their true functionality in their descriptions (misleading titles), (4) false positives, i.e., benign apps that were a part of the outliers either due to the broad app selection to form the GP_SPA dataset, or the k-means clustering itself.

After manual analysis of the fake apps, we observed that the nature of the fake apps is similar across all the groups. For conciseness, we only discuss results with respect to antivirus apps. We provide detailed results for all the groups on our project website [33].

As shown in Table II, the three found fake antivirus apps clearly describe their purpose as antiviruses, both in their title and descriptions. We manually ran these apps, and found they only display advertisements to other antivirus apps. We found that one of these apps (i.e., "Anti Virus Android") appears at the 16th position when Google Play is searched with the keyword "antivirus"; thus, it is prominently visible to the user as a top app in its category. The app has more than 100k downloads, and a 4-star user rating. We discuss the implications of these findings in Section V-C.

The encountered false positives occurred for two reasons: (1) the broad selection criteria for building the GP_SPA dataset, and (2) the feature vector used for the k-means clustering. The former resulted in the inclusion of legitimate apps that did not have misleading names or titles (e.g., "Antivirus Guides"). As seen in Table I, all but one of the false positives belong to this category, and can be eliminated by fine-tuning the search criteria. Furthermore, our technique can be improved by identifying and filtering out non-essential functionality and third-party libraries from the feature vector.

In case of the apps with misleading titles, but a roughly correct description, it is unclear whether there was a genuine attempt to fake functionality, or whether the misleading title was a mistake. In such cases, we take the conservative approach, and do not classify such apps as grayware.

6) Madware: As described in Section IV-D, we first analyzed popular ad libraries from AppBrain, and classified them as madware if they exhibited aggressive advertising behaviors.

TABLE III
MADWARE STATISTICS

Library Name	# Apps	Characteristics
mobileCore	1706	Notifications
AppBrainAppLift	716	Notifications
GetJar	123	Notifications
SkPlanet	91	Notifications
domob	89	Notifications
PapayaOffer	89	Notifications, AdWall
Tapcontext	81	Notifications
YouMi	61	Notifications
PontiflexOffers	38	Notifications
AdsMogo	10	Notifications
WAPS	7	Notifications
AppBucks	6	Notifications, Shortcuts
Kuguo	6	Notifications, Shortcuts
sellAring	1	Shortcuts, Replace dialtone
MoolahMedia	0	Notifications

We then disassembled apps from the PD_RNDM dataset, and looked for the presence of the ad libraries classified as madware using a path-based heuristic.

Results: Table III provides the list of madware libraries that we identified, and the number of apps that contain each library. Most of the ad libraries that we identified as madware displayed notifications to the user. Further, we found three instances of aggressive advertisement libraries that install advertisement shortcuts to the homescreen (i.e., AppBucks, Kuguo, sellAring). Our analysis of the PD_RNDM dataset returned 3,024 occurrences of madware out of the 100k apps. MobileCore alone is in 1.7K apps, i.e., more than 1% of the apps in our dataset show advertisements in the notification bar.

7) Rooting Tools and RATs: We conduct keyword-based search on the PD_RNDM dataset as discussed in Section IV-E.

Results: Out of the 100k descriptions, we did not locate any rooting tools or RATs, which may be due to dataset selection.

C. Summary of Findings

In this section, we discuss the effectiveness of our triaging techniques and the overall lessons learned from our findings.

Effectiveness of Triage: The results in Section V-B clearly demonstrate the effectiveness of our triaging techniques, and that text analytics is a promising approach for triaging grayware. We noted the triage reduction for each grayware category. In many cases, we reduced the workload from 1 million apps down to tens of apps. For smaller datasets, we reduced the workload from thousands of apps down to hundreds of apps. These reductions clearly indicate the value of text analytics in aiding in the identification of grayware in app markets.

Lessons from Triage: The results in Section V-B demonstrate the problem of grayware on Google Play. Our findings demonstrate that grayware is even present within some of the top-ranked apps displayed to the user. For example, a viable misrepresentor antivirus app is displayed within the top 20 results returned when users search for "antivirus" on Google Play. Since this app has been downloaded between 100k-500k times, this example demonstrates that grayware has the potential to impact a large number of users. Since many of these top-ranked grayware apps also have high ratings, not much confidence can be placed in the user's ratings/reviews to identify grayware. Our findings also show that grayware (e.g., imposters) may also negatively impact on the developer's brand and user experience. Some grayware may adversely impact the user's health and wellbeing (e.g., fake blood pressure), and are outright dangerous. Overall, the presence of grayware among the top apps on Google Play clearly indicates that it escapes the oversight of the app-market quality control.

VI. RELATED WORK

Malware Classification: Prior work in classifying threats in the mobile software ecosystem has focused on malware. Zhou and Jiang [3] systematically characterize over 1,200 malware samples, detailing their installation methods, activation methods, and nature of their payloads. Felt et al. [4] survey 46 malware samples across iOS, Android, and Symbian platforms. They characterize the incentives of these malware and formulate strategies for defending against them. In contrast, our work focuses on grayware, where classification of types and incentives has been poorly understood.

Malware Detection: Prior work in detecting unwanted apps has also mainly focused on malware. Zhou et al.'s approach [41], RiskRanker [42] and Crowdroid [43] use heuristics based on app behaviors and code properties to identify malware. MAST [15] triages app markets for malware by using multiple correspondent analyses on app behaviors and code properties. While these approaches contribute to building behavior-based signatures and heuristics, they focus on known malicious behavior, and may not be applicable for grayware identification.

Potentially Unwanted Apps: VetDroid [44] identifies PUAs using permission-based dynamic taint analysis to collect fine-grained information on the use of protected resources. While VetDroid focuses on malware, permission-use information could enhance some of our triaging techniques.

Privacy: Prior work has proposed techniques to detect potential privacy violations in mobile apps. TaintDroid [5] uses dynamic taint analysis to track sensitive information in Android apps. FlowDroid [24], AndroidLeaks [28], and PiOS [25] use static taint analysis for similar purposes. BAYESDROID [45] detects privacy leaks by transforming it into a Bayesian-classification problem.

Text Analytics for Security: Many of our grayware-triaging techniques rely on the analysis of text associated with apps (e.g., descriptions). Whyper [46] and AutoCog [47] use text analytics to justify permission requests based on an app's description. Chabada [48] detects suspicious apps by detecting anomalous permission requests by clustering apps based on their descriptions. Our heuristics for identifying misrepresentors operate at the API-level granularity and focus on overall app functionality rather than just protected behaviors.

VII. CONCLUSION

In this work, we investigated the topic of mobile grayware. We discussed how mobile grayware differs from PC gray-

ware, and provided a generalized characterization for mobile grayware behaviors. Based on these characteristics, we defined lightweight triaging heuristics that leverage text analytics and program analysis. Using these heuristics, we conducted a breadth study of grayware on Google Play. We found evidence that grayware appears to be a significant problem, warranting further investigation. Furthermore, we found that text analytics is a promising approach to triage grayware in app markets.

VIII. Acknowledgment

This material is based upon work supported by the Maryland Procurement Office under Contract No. H98230-14-C-0141. This work is also supported in part by NSF grants CNS-1513690, CNS-1222680, CNS-1253346, CNS-1434582, CCF-1349666, CCF-1409423, and CCF-1434596, CNS-1513939, and a Google Faculty Research Award.

References

[1] C. Lever, M. Antonakakis, B. Reaves, P. Traynor, and W. Lee, "The Core of the Matter: Analyzing Malicious Traffic in Cellular Carriers," in *Proc. of ISOC NDSS*, 2013.

[2] E. Kim, "Google Report: Android Security 2014 Year in Review," https://source.android.com/security/bulletin/Google_Android_Security_2014_Report_Final.pdf, 2015.

[3] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *Proc. of IEEE S&P*, 2012.

[4] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A Survey of Mobile Malware in the Wild," in *Proc. of ACM SPSM Workshop*, 2011.

[5] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proc. of USENIX OSDI*, 2010.

[6] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe Exposure Analysis of Mobile In-App Advertisements," in *Proc. of ACM WiSec*, 2012.

[7] "Android Users at Risk from Unapproved Imposter Apps," http://blog.trendmicro.com/android-users-at-risk-from-unapproved-imposter-apps/.

[8] "Scammers use Impostor Apps to Flood the Android Marketplace with Malware," http://gizmodo.com/5882776.

[9] "BBM for Android Fake App Scam Fools Blackberry Fans Again," http://www.cnet.com/news/bbm-for-android-fake-app-scam-fools-blackberry-fans-again/.

[10] "Commercial Spyware Invades Enterprises," http://www.csoonline.com/article/2886272.

[11] "Turning Off Your Smartphone Won't Save You from this Spyware," http://androidcommunity.com/turning-off-your-smartphone-wont-save-you-from-this-spyware-20150223/.

[12] "More Fake Antivirus Programs, Browsers found in Google Play and Windows Phone Store," http://www.pcworld.com/article/2156300.

[13] "The 1 New Paid App in the Play Store Costs 4 has Over 10000 Downloads a 4.7 Star Rating and It's a Total Scam," http://www.androidpolice.com/2014/04/06/the-1-new-paid-app-in-the-play-store-costs-4-has-over-10000-downloads-a-4-7-star-rating-and-its-a-total-scam/.

[14] "Creating Better User Experiences on Google Play," http://android-developers.blogspot.ro/2015/03/creating-better-user-experiences-on.html.

[15] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "MAST: Triage for Market-scale Mobile Malware Analysis," in *Proc. of ACM WiSec*, 2013.

[16] "TrendMicro Security Glossary," http://www.trendmicro.com/vinfo/us/security/definition/grayware.

[17] B. Uscilowski, "Mobile Adware and Malware Analysis," *Symantec Corp.*, vol. 1, 2013.

[18] "TrendMicro Spyware/Grayware," http://docs.trendmicro.com/all/ent/officescan/v10.5/en-us/osce_10.5_olhcl/osce_topics/spyware_grayware.htm.

[19] N. Viennot, E. Garcia, and J. Nieh, "A Measurement Study of Google Play," in *Proc. of ACM SIGMETRICS*, 2014.

[20] T. Vissers, W. Joosen, and N. Nikiforakis, "Parking Sensors: Analyzing and Detecting Parked Domains," in *Proc. of ISOC NDSS*, 2015.

[21] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "DroidMOSS: Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces," in *Proc. of ACM CODASPY*, 2012.

[22] Z. Chen, Z. Liang, Y. Zhang, and Z. Chen, "Evaluating Grayware Characteristics and Risks," *Journal of Computer Networks and Communications*, 2011.

[23] "T-Mobile, AT&T and Sprint Take Stance Against Fraudulent Premium SMS Charges," http://www.androidcentral.com/t-mobile-takes-stance-against-fraudulent-premium-sms-charges.

[24] 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 *Proc. of ACM PLDI*, 2014.

[25] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting Privacy Leaks in iOS Applications," in *Proc. of ISOC NDSS*, 2011.

[26] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Investigating User Privacy in Android Ad Libraries," in *Proc. of IEEE MoST Workshop*, 2012.

[27] F. Mohsen and M. Shehab, "Android Keylogging Threat," in *Proc of Collaboratecom*, 2013.

[28] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically Detecting Potential Privacy Leaks In Android Applications on a Large Scale," in *Proc. of TRUST*, 2012.

[29] N. Andronio, S. Zanero, and F. Maggi, "HELDROID: Dissecting and Detecting Mobile Ransomware," in *Proc. of RAID*, 2015.

[30] J. Kornblum, "Identifying Almost Identical Files using Context Triggered Piecewise Hashing," *Digital investigation*, vol. 3, 2006.

[31] "ssdeep," http://ssdeep.sourceforge.net/.

[32] M. F. Porter, "Snowball: A language for Stemming Algorithms," http://snowball.tartarus.org, 2001.

[33] "Grayware Project Website," https://sites.google.com/site/grayware2015/, 2015.

[34] A. K. McCallum, "MALLET: A Machine Learning for Language Toolkit," http://mallet.cs.umass.edu, 2002.

[35] T. L. Griffiths and M. Steyvers, "Finding Scientific Topics," *Proc. of the National Academy of Sciences*, vol. 101, no. suppl 1, 2004.

[36] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A Scalable System for Detecting Code Reuse among Android Applications," in *Proc. of Detection of Intrusions and Malware, and Vulnerability Assessment*, 2013.

[37] "AppBrain: Android Ad Network Statistics," http://www.appbrain.com/stats/libraries/ad.

[38] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA Data Mining Software: An Update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, 2009.

[39] I. J. M. Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. E. Hassan, "Impact of Ad Libraries on Ratings of Android Mobile Apps," *IEEE Software*, no. 6, pp. 86–92, 2014.

[40] "PlayDrone Archive," https://archive.org/details/android_apps.

[41] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets," in *Proc. of ISOC NDSS*, 2012.

[42] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and Accurate Zero-day Android Malware Detection," in *Proc. of MobiSys*, 2012.

[43] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-Based Malware Detection System for Android," in *Proc. of ACM SPSM Workshop*, 2011.

[44] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis," in *Proc. of ACM CCS*, 2013.

[45] O. Tripp and J. Rubin, "A Bayesian Approach to Privacy Enforcement in Smartphones," in *Proc. of USENIX Security*, 2014.

[46] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards Automating Risk Assessment of Mobile Applications," in *Proc. of USENIX Security*, 2013.

[47] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "AutoCog: Measuring the Description-to-permission Fidelity in Android Applications," in *Proc. of ACM CCS*, 2014.

[48] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking App Behavior Against App Descriptions," in *Proc. of ICSE*, 2014.