# A Measurement Study of Google Play

Nicolas Viennot
Computer Science Department
Columbia University
New York, NY, USA
nviennot@cs.columbia.edu

Edward Garcia
Computer Science Department
Columbia University
New York, NY, USA
ewg2115@columbia.edu

Jason Nieh
Computer Science Department
Columbia University
New York, NY, USA
nieh@cs.columbia.edu

## ABSTRACT

Although millions of users download and use third-party Android applications from the Google Play store, little information is known on an aggregated level about these applications. We have built PlayDrone, the first scalable Google Play store crawler, and used it to index and analyze over 1,100,000 applications in the Google Play store on a daily basis, the largest such index of Android applications. PlayDrone leverages various hacking techniques to circumvent Google's roadblocks for indexing Google Play store content, and makes proprietary application sources available, including source code for over 880,000 free applications. We demonstrate the usefulness of PlayDrone in decompiling and analyzing application content by exploring four previously unaddressed issues: the characterization of Google Play application content at large scale and its evolution over time, library usage in applications and its impact on application portability, duplicative application content in Google Play, and the ineffectiveness of OAuth and related service authentication mechanisms resulting in malicious users being able to easily gain unauthorized access to user data and resources on Amazon Web Services and Facebook.

## Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Performance of Systems**]: Measurement techniques; C.5.3 [**Computer System Implementation**]: Microcomputers–Portable devices; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval–Information filtering; J.7 [**Computers in Other Systems**]: Consumer products; K.6.2 [**Management of Computing and Information Systems**]: Installation Management–Performance and usage measurement; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection–Authentication

*SIGMETRICS'14,* June 16–20, 2014, Austin, Texas, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2789-3/14/06 ...$15.00.
http://dx.doi.org/10.1145/2591971.2592003.

## Keywords

Android; Authentication; Clone Detection; Decompilation; Google Play; Mobile Computing; OAuth; Security;

## 1. INTRODUCTION

The Google Play store allows users to download and use a vast amount of third-party applications. Millions of users register personal information both with Google and third-party services to download and use these applications on their personal Android phones and tablets. Hundreds of thousands of developers upload content to the Google Play store and millions of users download the content despite the fact that the content is largely unchecked.

However, little is known at an aggregate level about the hundreds of thousands of applications available in the Google Play store. This is due in large part to the lack of scalable tools available for discovering and analyzing Android applications in the Google Play store. Application source code is also only available to the respective third-party developers. Not even Google has access to the source code, as applications are submitted directly as compressed binary packages by application developers to Google Play. Furthermore, Google imposes various mechanisms to prevent others from crawling and indexing Google Play store content. For example, discovery of applications in the Google Play store is limited as only the first 500 applications belonging to any category or matching any search term can be found by browsing the store's web interface. Some applications also require specific hardware features or other existing applications and libraries to be available on the end-user device. Such applications are only available if the Google Play interface is accessed with an account registered on a device with the prerequisites available.

To explore Google Play content, we have created PlayDrone, the first scalable Google Play store crawler and application analysis framework. PlayDrone uses four key techniques. First, PlayDrone leverages common hacking techniques to easily circumvent security measures that Google uses to prevent indexing Google Play store content. These techniques include simple dictionary-based attacks for discovering applications, and decompiling and rebuilding the Google Play Android client to use insecure communication protocols to communicate with the Google Play servers to capture, understand, and reproduce the necessary protocols. Second, PlayDrone leverages higher-level languages and frameworks to provide highly concurrent, distributed processing with modest implementation effort. PlayDrone is written in Ruby and uses the Sidekiq [31] asynchronous processing framework and the Redis [33] key-value store. Its performance scales easily by simply adding servers to the cluster, enabling PlayDrone to efficiently crawl the Google Play store on a daily basis even as its content continues to grow. Third, PlayDrone stores each application's metadata and decompiled sources in a Git repository. This provides a simple versioning system for PlayDrone to track and manage multiple versions of each application and analyze how Google Play store content evolves over time. Finally, PlayDrone leverages the Elasticsearch [19] distributed real-time search and analytics engine using an indexing schema based on the Google Play store API to make it easy to analyze and explore the Google Play store metadata and content.

We have used PlayDrone to crawl the Google Play store and analyze over 1,100,000 Android applications, including decompiling the source code for over 880,000 free Android applications and analyzing over 100 billion lines of decompiled code. We demonstrate the usefulness of PlayDrone for analyzing application content by exploring four previously unaddressed issues in understanding Android applications. First, we provide a characterization of Google Play application content at scale. We discuss the relationship between application ratings and download frequency, discuss how applications are categorized in Google Play and how the choice of self-categorization can affect application visibility. We show how Google Play store content evolves over time, providing a measure of how often applications are released, updated, and removed. We also show that a small percentage of free applications account for almost all downloads.

Second, we perform the first large-scale source code analysis of library usage in Android applications. We show how library usage differs between popular and unpopular applications, including that native libraries are heavily used among the most popular applications. As a result, Android systems which only support Java-based applications are inadequate to support the most widely-used Android applications [12, 40]. We show that over half of the free Android applications use advertising libraries and discuss the size of the different advertising networks. We also show that cross-platform frameworks and application generators make up a very small fraction of the overall Google Play application content.

Third, we describe a new simple approach for efficiently detecting similar Android applications in the Google Play store. We use the structure of Android applications to analyze similarity by considering application assets and resources rather than requiring detailed source code analysis. This provides a more scalable approach than code analysis approaches with comparable results. Our results show that roughly 25% of Google Play store application content is duplicative, including various types of spam, application rebranding, and application cloning.

Finally, we present the first study of secret authentication key usage and its problems in Android applications. We show that developers often store secret authentication keys in their Android applications without realizing their credentials are easily compromised through decompilation. These secrets are publicly available in Google Play. We show these keys can be used by malicious users to steal server resources or user data available through services such as Amazon Web Services (AWS) or Facebook. Unlike compromised applications that only affect users who download and run them, these server vulnerabilities affect users without even running the applications. Our results demonstrate developer confusion may subvert the effectiveness of the widely used OAuth open source standard for authentication. We notified and worked with service providers to prevent these attacks, including providing Google with code to help them scan for secret keys in applications as part of the Google Play application publication process to protect users and developers.

This rest of this paper is organized as follows. Section 2 describes how PlayDrone intefaces with the Google Play API. Section 3 describes the PlayDrone crawler architecture. and Section 4 measures its scalable performance. Section 5 characterizes Android applications in Google Play. Section 6 discusses library usage in Android applications. Section 7 describes our approach for efficiently detecting similar Android applications and our measurements of similarity among applications in Google Play. Section 8 presents a study of secret authentication key usage and its problems in Android applications. Section 9 discusses related work. Finally, we present some concluding remarks.

## 2. INTERFACING WITH GOOGLE PLAY

To crawl the Google Play store, PlayDrone needs to communicate with the Google Play store, which requires use of a Google account for all the necessary functionality. Using only a few Google accounts to crawl the entire store might risk having the accounts disabled by Google, so we decided to harvest a large number of Google accounts. To do this quickly and efficiently, we had to address two problems. First, registering for a Google account requires solving CAPTCHAs. Second, registering for a Google account requires phone verification when the same IP attempts to register more than five accounts on a given day.

We addressed both issues by using a crowdsourcing Internet marketplace service to cheaply use other human users to register for Google accounts from a diverse set of IPs. Any such service could be used, including dedicated CAPTCHA solver services such as Death by Captcha [15]. We used Amazon's Mechanical Turk [1] for this purpose and deployed a website, `http://playdrone.io`, for users to submit the registered Google account information back to us. Mechanical Turk is a service where registered users are paid small dollar amounts to carry out trivial manual jobs. We posted a task description on Mechanical Turk with the following simple instructions: (1) Start your browser in incognito/guest mode. (2) Go on `https://accounts.google.com/SignUp`. (3) Fill out the requested information except "Mobile phone" and "Current email address" as they are not necessary. (4) Go to `http://playdrone.io/accounts/new` and enter the email and password of the account you created. (5) Answer with the returned confirmation code. The return confirmation code allows the user to be paid for the work. Before the code is given, `playdrone.io` validates the submitted Google account information to ensure that it is not duplicative and can authenticate with Google services. Mechanical Turk prevents users from carrying out the same task twice, so four copies of the task were created to benefit from dedicated users. We paid 10 cents per account, resulting in the creation of more than 500 Google accounts in just a few hours for a little more than $50. Note that Google accounts can be found on the black market for a similar price.

Google exposes an internal, non-documented API to its Android Play clients to access the store and download applications over the air. PlayDrone replicates the behavior of legitimate Android Play clients, each using a previously harvested Google account associated with Galaxy Nexus device profiles. PlayDrone interacts with Google Play servers through four different APIs. The first one is the *checkin* API to associate a Google account with an Android device, necessary to access the three other Google Play APIs. Based on the device used, Google Play may make available a different set of applications for the device. For example, some applications may only be available to devices in certain geographic locations. The *search*, *details*, and *purchase* APIs are used to discover applications, fetch application details, and retrieve binary download links, respectively. We were fortunately able to leverage additional information from non-Google sources [34, 24] to implement most of the APIs.

However, because of the lack of documentation and source code for the checkin API and our desire to make use of multiple Google accounts efficiently, we had to reverse engineer that API ourselves. Google makes it difficult to derive the checkin API by ensuring that communications between the Google Play client and servers are over SSL, preventing the capture of the wire protocol. However, since the Google Play client and related service applications are compiled to Dalvik bytecode, they were straightforward to disassemble with baksmali [26]. We changed all `https` strings to `http` ones and recompiled the client to send and receive unencrypted communications via an SSL proxy to the Google Play servers. This man-in-the-middle attack allowed the capture of a real device registration over the wire and the ability to reproduce it to reverse engineer the checkin API. Based on this API, we created a tool to register a fake Android device given an email password pair corresponding to a Google account. During the registration process, device capabilities and metadata need to be sent to the Google servers, including more than 50 data fields such as the mobile network provider, an IMEI number, the WiFi MAC address, and OpenGL capabilities. We extracted this information from a legitimate T-Mobile Galaxy Nexus device; all of our search results are therefore restricted to what would be accessible on such a device. The tool then uses this information, but randomly generates valid IMEI and MAC addresses to prevent device blacklisting by Google. A similar approach could be used for reverse engineering the other APIs if needed.

## 3. CRAWLER ARCHITECTURE

Given a set of Google accounts and the APIs for communicating with the Google Play servers, the PlayDrone crawler discovers and downloads Android applications with their metadata. Figure 1 shows the six components of the PlayDrone crawler architecture: a Sidekiq job scheduler for distributing work to multiple machines, a Redis key-value store to store the jobs, an Amazon EC2 proxy, Git version control repositories, an Elasticsearch distributed search and analytics engine, and an Nginx web server frontend. These components work together to provide four key benefits. First, since crawling and analyzing the evolution of Google Play on a daily basis requires a fair amount of CPU power and storage space, PlayDrone is designed using a higher-level language that makes it simple to build a powerful distributed system that scales out by just adding more servers. PlayDrone is written in Ruby, which provides an excellent higher-level language ecosystem that is simple to use and allows PlayDrone to leverage existing, well-tested mechanisms such as Sidekiq, Redis, and Elasticsearch to further simplify its implementation. Second, to circumvent attempts by Google to limit crawling of the Google Play store, PlayDrone uses various techniques to hide its activities, including using multiple Google accounts, rate limiting the number of requests for each account, and proxying requests through a third-party service provider. Third, to analyze the behavior of the Google Play store and its applications over time, PlayDrone leverages Git to store and track multiple versions of each Android application and metadata to allow the system to analyze how applications evolve as they are updated from one version to another. Finally, PlayDrone is designed with an easy-to-use web interface supported by Nginx and Elasticsearch to make it simple to search and perform various forms of analysis on the Android applications and their metadata. For example, Figure 2 shows the use of PlayDrone to identify how the Gmail application code has been updated.

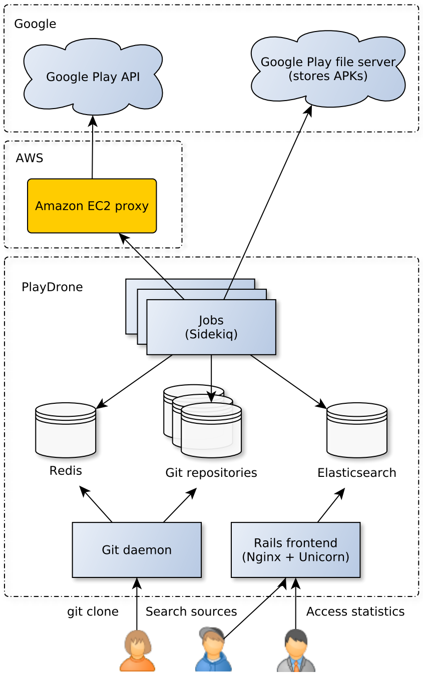
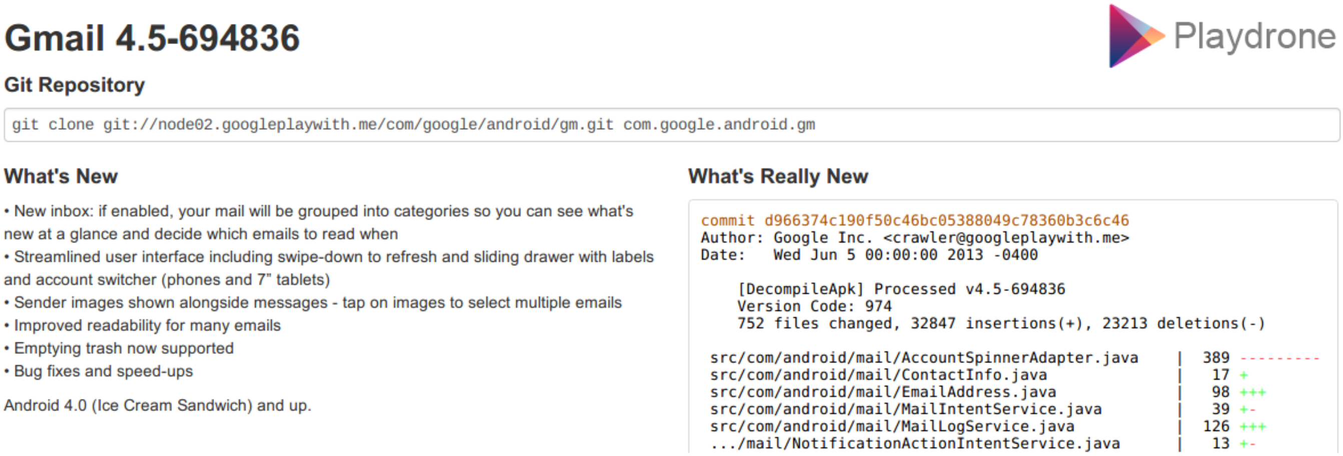


Figure 1: PlayDrone crawler architecture.

PlayDrone uses the Redis key-value store [33] and the Sidekiq background processing framework [31] to efficiently implement master-slave distributed computing. Sidekiq assigns jobs to different slave machines. PlayDrone uses two Sidekiq job queues, one for discovering Android applications in Google Play, and the other for downloading and processing applications. Redis runs on a master machine to store the job queues and track the use of Google accounts, which applications need to be processed, and what machines have been assigned to process which applications. Although only a single master machine is used, PlayDrone leverages Redis very efficiently so that the architecture can scale out to support hundreds of slave worker machines.

Discovering applications in Google Play is not straightforward because Google does not provide any public list of all the available applications in Google Play and limits



Gmail 4.5-694836

Git Repository

```
git clone git://node02.googleplaywith.me/com/google/android/gm.git com.google.android.gm
```

What's New

- New inbox: if enabled, your mail will be grouped into categories so you can see what's new at a glance and decide which emails to read when
- Streamlined user interface including swipe-down to refresh and sliding drawer with labels and account switcher (phones and 7" tablets)
- Sender images shown alongside messages - tap on images to select multiple emails
- Improved readability for many emails
- Emptying trash now supported
- Bug fixes and speed-ups

Android 4.0 (Ice Cream Sandwich) and up.

What's Really New

```
commit d966374c190f50c46bc05388049c78360b3c6c46
Author: Google Inc. <crawler@googleplaywith.me>
Date:   Wed Jun 5 00:00:00 2013 -0400

    [DecompileApk] Processed v4.5-694836
    Version Code: 974
    752 files changed, 32847 insertions(+), 23213 deletions(-)

 src/com/android/mail/AccountSpinnerAdapter.java    |  389 ---------
 src/com/android/mail/ContactInfo.java              |   17 +
 src/com/android/mail/EmailAddress.java             |   98 +++
 src/com/android/mail/MailIntentService.java        |   39 +-
 src/com/android/mail/MailLogService.java           |  126 +++
 .../mail/NotificationActionIntentService.java      |   13 +-
```

Figure 2: PlayDrone's web interface showing the Gmail application and its Git diff.

the search results returned from querying Google Play to no more than 500 applications. To overcome these problems, PlayDrone uses a dictionary attack method involving roughly a million words as search terms to search Google Play to find applications. To cover a broad range, words are used from multiple languages, including English, German, French, Spanish, Swahili, Japanese, Italian, Danish and Swedish. A Sidekiq job is created for each search term, making the discovery queue roughly a million jobs in length. For each search job, PlayDrone sends a search request to Google Play through its proxy. Because each search request is a separate job, requests that need to be retried are isolated in the event of a network issue or other problem. It is interesting to note that the Google Play API does not return any search results when hit directly from the PlayDrone servers in Canada, but proxying the connection through an Amazon EC2 public IP in North Virginia causes the API to successfully return results. We do not know if the IPs we are using are banned, or if the filtering is based on IP geolocation, but this anecdotal evidence demonstrates the benefit of our proxy approach. For each search request, Google Play returns a list of applications in batches of 20 applications, with a link to the next page if there are more results to be fetched; the pagination stops at 500 results. Each application in the list includes a link to a details page for the application, which provides a description of the application. When PlayDrone finds an application that it has not seen before, it stores the application unique identifier in Redis, and adds the application to the Sidekiq processing queue to be downloaded.

When a new application identifier is discovered, PlayDrone downloads and processes the application. PlayDrone uses Redis to atomically assign the application to a machine for processing and instantiates a Git repository for the application on the assigned machine. PlayDrone fetches the application's details page, from which it extracts all of the application's metadata and downloads and stores the application binary package (APK) into its Git repository. Application metadata includes a list of related applications, which PlayDrone uses to discover applications not identified via the dictionary-based method. Only free application APKs are downloaded to avoid the costly expense of downloading all paid applications. Future processing of the application is done on the machine where its Git repository resides. This simple distribution mechanism takes better advantage of file locality as opposed to relying on a distributed file system, which would result in much worse file system performance using Git and would add unnecessary complexity to the system.

PlayDrone provides a plugin architecture to allow a user of the system to write plugin middleware to perform various forms of processing and analysis on applications once they have been downloaded. For example, we wrote a plugin for decompiling APKs into readable Java sources to enable easier comprehension of application behavior. The decompile plugin uses apktool [35] to deflate the XML files and dex2jar [17] with a command-line version of JD-Core [37] for Java decompilation. The resulting Java sources are quite readable and complete, though not directly suitable to recompile back into an APK. As another example, we wrote a plugin for parsing the `./res/values/public.xml` file to extract resource names and compute the MD5 hashes of asset and resource files in the application to facilitate detection of similar applications as discussed in Section 7.

PlayDrone stores the raw application metadata and decompiled sources from crawling the Google Play store in its respective Git repositories, with each commit tagged with the crawl date or the application version when applicable. PlayDrone stores source code and application metadata in Elasticsearch, a distributed search and analytics engine [19]. Elasticsearch has a simple web interface, allowing fast searching of the data and various forms of simple analysis before writing a middleware plugin to perform more complete analysis of the data. A different index is used for each day of metadata from the Google Play store, making it possible to visualize the evolution of the Google Play store. For space reasons, only the most recent decompiled sources are indexed in Elasticsearch; older versions are stored in the respective Git repositories. Users can reindex data in Elasticsearch from the Git repositories as they store all the raw data collected from Google Play.

## 4. CRAWLER PERFORMANCE

Because of the use of the Ruby ecosystem, the effort to build and deploy PlayDrone was fairly low. The entire application was less than 2000 lines of Ruby and HTML. As a (unfair) comparison, GNU `cat` is 550 lines of code. PlayDrone deployment is also quite simple and the sys-

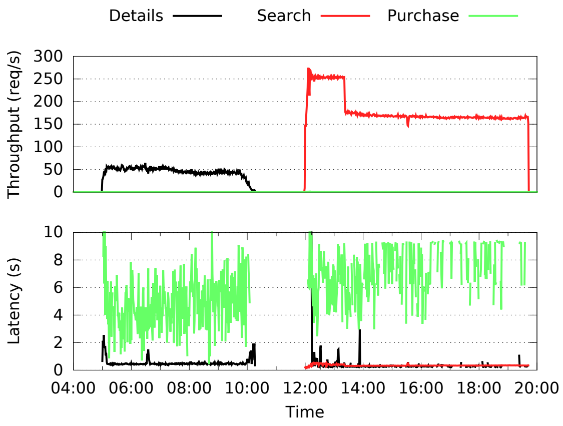


Figure 3: A typical daily crawl from May 21, 2013 showing the throughput and latency of the Google Play API.

tem setup is entirely automated with Chef [10] and Capistrano [9], which are tools written in Ruby. The amount of code we had to write to manage the entire system was less than 550 lines of code, including configuration file templates (e.g. upstart). PlayDrone is simple enough to be understood and used by others, and yet can provide powerful insights into Google Play. We have made PlayDrone source code available on GitHub [38] for others to use.

We deployed PlayDrone on ten servers on April 24, 2013, each with Intel Xeon E3 turbo boosted at 3.8Ghz, 32GB of RAM, and 2x2TB drives. The initial crawling took around two days, limited by CPU due to the source decompilation. Once the initial crawling was complete on April 26, 2013, the total size occupied by the Git repositories was 3.9TB with 790,000 applications, an average of about 5MB per application. Subsequently, crawling Google Play for new applications and application updates as well as identifying deleted applications took only several hours, enabling PlayDrone to crawl the entire Google Play store on a daily basis. Note that subsequent crawls after the initial crawl operate in exactly the same manner as the initial crawl, but PlayDrone does not need to download and decompile APKs for applications that have not changed. After two months of crawling, the total size of the compacted Git repositories reached 5.3TB with roughly 960,000 applications, among which 70,000 applications were removed from the Google Play store but archived by PlayDrone. We decommissioned the crawler on June 22, 2013 to save resources. In November 2013, we redeployed the crawler to validate that our crawling method was still valid and analyze the evolution of the Google Play store five months later.

Figure 3 shows the throughput and latency of the Google API during a daily crawl using PlayDrone. A daily crawl updates all the metadata of each known application, and discovers and downloads new applications. From 05:00 to 10:00, PlayDrone updates the metadata of all known applications in the system. The *Details* API endpoint is called once for each application. The response from Google servers includes various metadata including the current number of downloads, the current version of the application, and a list of related applications. Because all application metadata is stored in Git, PlayDrone bottlenecks at 50 requests per second (req/s) due to disk I/O. Later in the afternoon from 12:00 to 20:00, PlayDrone performs a dictionary search to discover additional applications. During the first 90 minutes, the *Search* API is called at full capacity, artificially rate limited to 250 req/s to avoid getting our Google accounts flagged; we empirically measured the rate limit enforced by Google at 1000 req/min per account. The throughput reaches maximum capacity because many words do not generate any results and the average API response is small to parse. After 90 minutes, the response size gets larger and PlayDrone bottlenecks on CPU, parsing these lengthy responses. When the *Details* API returns with a new version of the application that PlayDrone has not previously downloaded, the *Purchase* API endpoint is called to retrieve the download link, and proceed to downloading the APK.

**Number of applications**

| | **June 22, 2013** | **November 30, 2013** | |
|---|---|---|---|
| Free apps | 691,517 | 884,217 | (+28%) |
| Paid apps | 195,703 | 223,259 | (+14%) |
| **All apps** | 887,220 | 1,107,476 | (+25%) |

**Cumulative download counts (min-max)**

| | **June 22, 2013** | **November 30, 2013** | |
|---|---|---|---|
| Free apps | 22G-85G | 31G-116G | (+37%) |
| Paid apps | 111M-428M | 126M-488M | (+14%) |
| **All apps** | 23G-85G | 31G-117G | (+37%) |

Table 1: Number of applications and cumulative download counts on June 22, 2013 and November 30, 2013.

Table 1 shows the number of applications PlayDrone discovered and downloaded from Google Play. By June 22, 2013, we indexed 887,220 applications. By November 30, 2013, we indexed 1,107,476 applications, which is to the best of our knowledge the most extensive coverage of the Google Play store. Table 1 also shows cumulative download counts based on the download count ranges reported by Google Play. From June to November, The Google Play store grew by 25% in the number of applications, and 37% in download counts. Based on the last official report from Google indicating Google Play having 1 million applications as of July 24, 2013 [39] and the rate of growth of Google Play content shown in Figure 4, we estimate that our method covers over 90% of the Google Play applications with a Galaxy Nexus on T-Mobile profile. To increase coverage, we could check in other types of Android devices to fetch, for example, applications that are restricted to tablets, or applications reserved for a specific mobile carrier. Unless otherwise indicated, the analysis of Google Play in the remainder of this paper focuses on the June 22, 2013 data collection.

## 5. GOOGLE PLAY CHARACTERISTICS

Using PlayDrone, we present aggregated characteristics of Android applications based on a comprehensive index of Google Play application sources and metadata. Table 2 shows the list of 887,220 applications available in the Google Play store indexed by PlayDrone, separated into their respective categories and whether they are free or paid. Game applications are listed and categorized separately since they are shown in a separate top-level directory in Google Play. Categories are listed from most to least number of applications, and each application can only belong to one category in Google Play. Overall, there are more than 3.5 times as many free applications as paid applications. The aggregate download counts of paid applications accounts for only 0.05% of total downloads from the store as shown in Table 1.

| Applications | | | |
|---|---|---|---|
| **Category** | **Free apps** | **Paid apps** | **Total apps** |
| Personalization | 59,477 | 33,682 | 93,159 |
| Entertainment | 72,685 | 16,772 | 89,457 |
| Education | 41,115 | 16,985 | 58,100 |
| Lifestyle | 48,763 | 11,269 | 60,032 |
| Tools | 47,608 | 12,092 | 59,700 |
| Books & Reference | 34,990 | 22,703 | 57,693 |
| Business | 41,701 | 2,675 | 44,376 |
| Travel & Local | 28,473 | 13,379 | 41,852 |
| Music & Audio | 33,221 | 4,550 | 37,771 |
| Sports | 19,906 | 4,889 | 24,795 |
| Productivity | 18,575 | 5,557 | 24,132 |
| Health & Fitness | 18,078 | 5,707 | 23,785 |
| News & Magazines | 21,919 | 1,260 | 23,179 |
| Social | 17,548 | 1,858 | 19,406 |
| Finance | 16,731 | 2,191 | 18,922 |
| Communication | 14,725 | 2,999 | 17,724 |
| Media & Video | 15,014 | 2,438 | 17,452 |
| Shopping | 11,547 | 678 | 12,225 |
| Photography | 8,407 | 2,331 | 10,738 |
| Medical | 7,137 | 3,405 | 10,542 |
| Transportation | 8,099 | 1,340 | 9,439 |
| Comics | 3,798 | 1,721 | 5,519 |
| Libraries & Demo | 3,760 | 256 | 4,016 |
| Weather | 2,810 | 563 | 3,373 |
| Total | 596,087 | 171,300 | 767,387 |

| Games | | | |
|---|---|---|---|
| **Category** | **Free apps** | **Paid apps** | **Total apps** |
| Brain | 36,533 | 8,938 | 45,471 |
| Casual | 24,370 | 5,901 | 30,271 |
| Arcade | 22,517 | 6,309 | 28,826 |
| Cards | 5,589 | 1,619 | 7,208 |
| Sports Games | 3,821 | 1,167 | 4,988 |
| Racing | 2,600 | 469 | 3,069 |
| Total | 95,430 | 24,403 | 119,833 |
| **Grand Total** | 691,517 | 195,703 | 887,220 |

Table 2: Applications in Google Play as of June 22, 2013.

We also measured that the top 10% of most downloaded applications accounts for over 96% of the total downloads, and the top 1% of most downloaded applications accounts for over 78% of the total downloads as of June 22, 2013. As of November 30, 2013, the top 1% of most downloaded applications accounts for over 81% of the total downloads. This suggests that a decreasing number of applications accounts for almost all application usage in Google Play, indicating the increasing difficulty of releasing a popular application.

Other than games, personalization, a somewhat vague category name, represents the largest category of applications, with over 90,000 applications. To find out more about this category, we ranked the most recurring terms in the titles and descriptions of applications, discarding common nondescriptive words such as not, the, can, it, or, etc. The top three words among personalization applications were *wallpaper*, *please*, and *like*, accounting for 64,341 (69%), 35,953 (39%), and 26,563 (29%) applications, respectively. To compare with other categories, the proportions of applications that contain these words across the rest of Google Play were 4%, 12% and 12%, respectively. This suggests that the personalization category may be infected with many useless applications that users would consider as spam. This result also suggests that wallpapers deserve their own category.

The problems with the personalization category are just one of the problems with the application categorization used by Google Play. Category names are confusing and overlapping. Since an application can only belong to one category, the social and photography categories are mutually exclusive, so a user browsing the latter category will find no sign of Instagram, arguably the most popular photo sharing application. Similarly, business and productivity applications are categorized separately, health and fitness applications cannot correspond to lifestyle ones, and music and audio applications must be categorized separately from media and video applications. Given the large differences in the number of applications in each category and the already confusing category names, Table 2 may be useful to developers in deciding which category to use for applications to increase their visibility by using a less populated category.

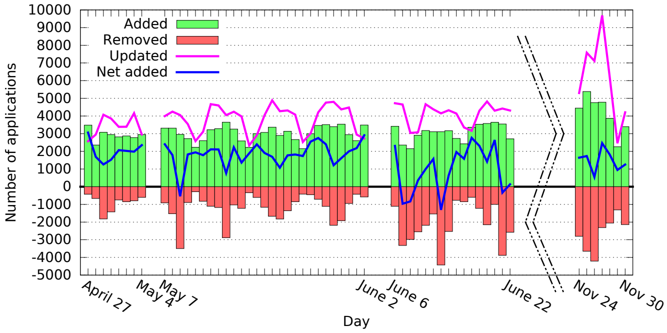


Figure 4: Evolution of Google Play from April 26, 2013 to November 30, 2013. Missing bars represents lack of crawling.

Figure 4 shows how the content of Google Play evolves over time in terms of how often applications are released, updated, and removed from Google Play. Google removes applications that do not comply with their terms and conditions. On most days, more applications are added than removed, and more applications are updated than added. During the May-June 2013 period, roughly 3000 new applications arrived daily on Google Play. This is far more than the Top New listing in Google Play, which is limited to 500 applications and provides an incomplete picture of new application content in Google Play. The November 2013 crawl shows a 30% increase in application release and update rate. The Google Play store is growing even faster than earlier in the year, motivating the need for automated auditing and quality control solutions.

Figure 5 shows a distribution of the average rating versus download count for applications in Google Play. Download counts are shown in bucketed ranges provided by Google Play; exact download counts are not available. Free and paid applications are shown separately. There are no paid applications with more than 5 million downloads. Users can rate an application with stars from 1 to 5, 5 being the highest possible rating, and these ratings are aggregated by Google Play per application to compute an overall average rating for each application. For example, Figure 5 shows that for applications with less than 500 downloads, there are applications with an overall rating as low as 1, applications with an overall rating as high as 5, and on average, paid applications have an overall rating of 4 with free applications having an overall rating of greater than 4. One might expect that applications with higher ratings would have higher download counts, but in fact the average of the overall ratings across all applications in any bucket of download counts was between 4 and 4.5. What did change was that as the download

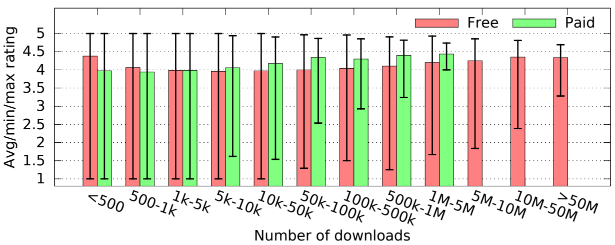


Figure 5: Ratings vs download counts for free/paid applications on June 22, 2013.

counts increased, the rating for the lowest rated application in a given download count bucket generally increased. For example, there are free applications with only 1 star overall ratings with less than 50 thousand downloads, but there are no applications with less than a 2.5 star overall rating with 10 million or more downloads. This increase in the minimum overall rating as the download count increases is even more pronounced for paid applications. Surprisingly though, there are still quite a few applications with very low ratings despite their high download counts.

Table 3 shows the top ten applications with the worse and best ratings which have at least a million downloads. The ten worst rated list shows three applications that come preinstalled, namely the T-Mobile one, the Motorola fitness watch application, and the HRS hotel application preinstalled on some HTC phones. Another interesting case is the Outlook.com application that advertises in its description to be the official Microsoft application for outlook.com. The developer, "Microsoft + SEVEN", has only one application released under his account, while the official Microsoft account, "Microsoft Corporation", has 18 applications. We were surprised that Microsoft hired SEVEN to develop their outlook.com application and did not release it under the official Microsoft account as releasing official applications under other accounts trains users to be more vulnerable to phishing attacks. The worst rated application with at least one million downloads is DroidScale, which gives the users the ability to turn their phones into a scale, enabling them to weight regular household objects. We decompiled the sources, to find that the weight is simply measured with `Random.nextDouble()`, a randomly generated number. An ad-free version of the application is even offered for $0.99 that has 1000-5000 downloads.

The top ten best rated list shows half of the applications on the list being related to the Holy Quran. Such high ratings can be explained by the target audience that prefers to rate the content rather than the application. For example, reading comments in the TvQuran application reveals unhappy users having sound issues or readability issues while still giving 5 stars. User comments and ratings are publicly available through Google+, so certain users may not be comfortable putting a bad rating on a Holy Quran application. We also observe that some applications such as Slots Deluxe or Cool Wallpapers have 10 times more ratings count than other applications in the same download counts bucket. While there might be some aggressive incentive from the application telling the user to rate the application, it is hard to draw any conclusions as Google does not give exact download counts.

| Application | Downloads | # Ratings | Rating |
|---|---|---|---|
| TvQuran | 1M-5M | 13,675 | 4.93 |
| Бияеты ПДД 2013 РФ | 1M-5M | 15,738 | 4.92 |
| Holy Quran Maher Moagely | 1M-5M | 6,341 | 4.91 |
| Slots Deluxe - Slot Machines | 1M-5M | 108,431 | 4.90 |
| [illegible] | 1M-5M | 19,567 | 4.89 |
| Alchemy Classic HD | 1M-5M | 37,706 | 4.89 |
| Zombies...OMG! | 1M-5M | 46,236 | 4.89 |
| Quran - [illegible] | 1M-5M | 17,666 | 4.89 |
| My Prayer - [illegible] | 1M-5M | 33,893 | 4.88 |
| Cool Wallpapers HD | 1M-5M | 210,320 | 4.87 |
| GoToMeeting | 1M-5M | 4,696 | 2.41 |
| Outlook.com | 10M-50M | 78,049 | 2.39 |
| TAMAGO hd | 1M-5M | 5,706 | 2.31 |
| MOTOACTV | 1M-5M | 4,191 | 2.30 |
| Screen Capture - No Rooting | 1M-5M | 2,963 | 2.28 |
| Wet Lesbian | 1M-5M | 2,865 | 2.23 |
| Ameba | 1M-5M | 35,933 | 2.21 |
| HRS App | 1M-5M | 5,778 | 1.99 |
| T-Mobile More For Me | 5M-10M | 1,763 | 1.84 |
| DroidScale | 1M-5M | 5,450 | 1.67 |

Table 3: Top 10 of the best and worse rated applications with at least 1 million downloads on June 22, 2013.

# 6. APPLICATION LIBRARY USAGE

Using PlayDrone to decompile applications, we present the first large-scale source code analysis of library usage in Android applications. One important question regarding library usage is how often native libraries are used in the context of Android's Java applications to improve their user experience. A native library contains code compiled directly for ARM and is invoked from the Java part of the application to improve performance or access low-level system calls. Figure 6 shows the number and percentage of Android applications that use native libraries versus download counts. For non-popular applications, those with less than 50,000 downloads, 14% of them on average have at least one native library. However, for popular applications, native library usage increases significantly such that among applications with more than 50 million downloads, the vast majority of them, 70% of them on average, have at least one native library. For example, Instagram uses seven different native libraries to optimize image processing and encoding performance, and Facebook uses nine different native libraries to access low-level system functionality, such as getting and setting the file descriptors limit of the current process.

As an application rises in popularity, developers are perhaps more willing to spend time and money to use native libraries to optimize the user experience of the application. Although there are efforts to run Android on non-ARM platforms for offloading and other reasons [12, 40], these systems rely on Java bytecode portability and do not support native library execution. Our results suggest that such approaches are problematic in that they will be unable to run the most popular Android applications. Despite Java's portability, these results indicate that the wide use of native libraries in popular Android applications may increasingly tie Android to ARM-based systems.

Table 4 shows the breakdown of the most popular Java libraries used among free applications, separated into non-popular (<50k downloads) and popular applications (≥50k downloads). Applications may use more than one library, so the sum of the percentages may exceed 100%. The breakdown shows that ad libraries are most widely used, with al-

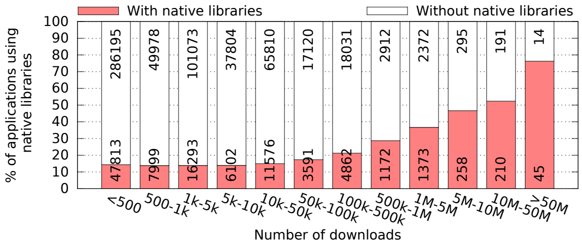


Figure 6: Native libraries usage compared to application popularity on June 22, 2013.

most half of the non-popular applications using ad libraries while almost two-thirds of the popular applications use ad libraries. Ad libraries are most likely more common in popular free applications because there is a greater focus on monetization for successful applications than many non-popular free applications. Google is the most popular advertising platform, with almost half of the popular applications using Google Ads. Among applications that use ad libraries, Google dominates with roughly 75% market share. This general trend of increased monetization of popular applications and Google's dominant role is also seen for billing libraries used for in-app purchases. Other than advertising, social libraries are the next most popular with almost 15% of free applications using those libraries. The official Facebook SDK is the most widely used, followed by Twitter4J, an unofficial Twitter library.

Table 4 also shows the percentage of free applications built using cross-platform frameworks or application generators. Almost 15% of non-popular applications are built using these frameworks and generators, but only about 3% of popular applications use these frameworks and generators. Beginning developers may find it useful to use application generators to simplifying development, or use cross-platform frameworks to be able to deploy an application on both Android and iOS platforms. However, the measurements suggest that these generators and frameworks most likely lack the necessary functionality and degree of control for building more sophisticated applications with richer user experiences that are more likely to become popular and widely used.

## 7. SIMILAR APPLICATIONS

Leveraging PlayDrone, we introduce a simple approach to identify similar applications in Google Play for the purposes of detecting duplicative content and application clones. Similar applications are those that appear to share the same source code origin, share common design and layout patterns, and offer comparable application level experiences to the end user. Previous studies have shown Android application clones to be vectors in spreading malware [43] as well as instruments to divert users and advertising revenues from legitimate applications [23]. While prior research has focused on code analysis to identify similar applications, this can be problematic for obfuscated code or applications where the core logic is written in multiple programming languages, such as in PhoneGap and Adobe Air applications. Additionally, code analysis methods are often too computationally expensive to scale to analyze all of Google Play.

**Advertising platform**

| **Name** | **Non-popular apps** | | **Popular apps** | |
|---|---|---|---|---|
| Google Ads | 225,344 | (35.73%) | 25,946 | (49.47%) |
| Google Analytics | 64,799 | (10.28%) | 7,522 | (14.34%) |
| Flurry | 34,040 | (5.40%) | 6,477 | (12.35%) |
| Millennial Media Ads | 23,120 | (3.67%) | 3,480 | (6.64%) |
| MobFox | 19,709 | (3.13%) | 1,219 | (2.32%) |
| InMobi | 17,432 | (2.76%) | 3,128 | (5.96%) |
| RevMob | 18,064 | (2.86%) | 1,018 | (1.94%) |
| Urban Airship Push | 14,657 | (2.32%) | 525 | (1.00%) |
| Mobclix | 12,315 | (1.95%) | 1,866 | (3.56%) |
| Smaato | 12,290 | (1.95%) | 241 | (0.46%) |
| AirPush | 10,773 | (1.71%) | 657 | (1.25%) |
| SendDroid | 9,907 | (1.57%) | 742 | (1.41%) |
| Adfonic | 9,170 | (1.45%) | 435 | (0.83%) |
| Jumptap | 8,968 | (1.42%) | 570 | (1.09%) |
| HuntMads | 7,275 | (1.15%) | 135 | (0.26%) |
| TapIt | 7,131 | (1.13%) | 259 | (0.49%) |
| Umeng | 5,742 | (0.91%) | 805 | (1.53%) |
| TapJoy | 3,358 | (0.53%) | 2,645 | (5.04%) |
| AppLovin | 5,124 | (0.81%) | 824 | (1.57%) |
| MoPub | 4,187 | (0.66%) | 1,167 | (2.23%) |
| LeadBolt | 3,517 | (0.56%) | 475 | (0.91%) |
| **Total** | 302,611 | (47.98%) | 34,348 | (65.49%) |

**Social**

| **Name** | **Non-popular apps** | | **Popular apps** | |
|---|---|---|---|---|
| Facebook SDK | 77,489 | (12.29%) | 6,206 | (11.83%) |
| Twitter4J | 41,606 | (6.60%) | 2,057 | (3.92%) |
| **Total** | 92,495 | (14.67%) | 6,990 | (13.33%) |

**Cross-platform framework**

| **Name** | **Non-popular apps** | | **Popular apps** | |
|---|---|---|---|---|
| PhoneGap | 36,915 | (5.85%) | 606 | (1.16%) |
| Adobe Air | 12,761 | (2.02%) | 619 | (1.18%) |
| Titanium | 8,316 | (1.32%) | 138 | (0.26%) |
| **Total** | 57,991 | (9.20%) | 1,363 | (2.60%) |

**Application generator**

| **Name** | **Non-popular apps** | | **Popular apps** | |
|---|---|---|---|---|
| Bizness Apps | 10,011 | (1.59%) | 3 | (0.01%) |
| App Inventor | 9,560 | (1.52%) | 152 | (0.29%) |
| Andromo | 6,294 | (1.00%) | 156 | (0.30%) |
| iBuildApp | 4,149 | (0.66%) | 25 | (0.05%) |
| Mobile by Conduit | 3,989 | (0.63%) | 21 | (0.04%) |
| **Total** | 34,003 | (5.39%) | 357 | (0.68%) |

**Bug tracking**

| **Name** | **Non-popular apps** | | **Popular apps** | |
|---|---|---|---|---|
| BugSense | 59,550 | (9.44%) | 4,251 | (8.11%) |
| Acra | 25,658 | (4.07%) | 1,450 | (2.76%) |
| **Total** | 84,896 | (13.46%) | 5,663 | (10.80%) |

**Billing**

| **Name** | **Non-popular apps** | | **Popular apps** | |
|---|---|---|---|---|
| Google Billing | 27,846 | (4.42%) | 6,312 | (12.04%) |
| Paypal | 16,943 | (2.69%) | 374 | (0.71%) |
| Authorize.net | 8,464 | (1.34%) | 1 | (0.00%) |
| Amazon Purchasing | 3,356 | (0.53%) | 1,044 | (1.99%) |
| **Total** | 44,798 | (7.10%) | 6,686 | (12.75%) |

**Audio/graphics engine**

| **Name** | **Non-popular apps** | | **Popular apps** | |
|---|---|---|---|---|
| FMOD | 8,199 | (1.30%) | 1,705 | (3.25%) |
| Unity3D | 8,158 | (1.29%) | 1,601 | (3.05%) |
| AndEngine | 7,098 | (1.13%) | 1,080 | (2.06%) |
| libGDX | 6,311 | (1.00%) | 1,395 | (2.66%) |
| Corona SDK | 3,750 | (0.59%) | 396 | (0.76%) |
| **Total** | 23,774 | (3.77%) | 4,222 | (8.05%) |

Table 4: Application libraries usage on June 22, 2013.

Our scalable approach comes from a simple observation: humans can typically just look at the screenshots of applications to determine if applications are similar. Humans recognize patterns by looking at the UI layouts or the similarity of images. Android applications are structured such that these visual aspects are embodied in resources and assets, such as images, sounds, UI layouts, or application settings.

Resources and assets are two different ways to embed visual elements in Android applications, the former having a locale-aware naming hierarchy through the 'R' Java class while the latter provides raw access to files. Based on this observation, we leverage the structure of Android applications to use a feature set of resource names and asset signatures, the latter generated by taking the MD5 hash of each asset of an application excluding its icon and XML files. This feature set is easy to identify and compute even for obfuscated application code, making it fast enough to use with daily crawls of Google Play.

Using PlayDrone to study 610,000 free applications downloaded and decompiled on May 5, 2013, we found roughly 58 million unique resource names and 45 million unique asset signatures. Because the most common resource names and asset signatures occur in widely-used application libraries, their frequency is high and they are poor indicators of application similarity. To address this issue, we use a simple blacklist approach with a cutoff parameter $C$ that ignores resource names and asset signatures appearing in more than $C$ applications. With a cutoff of 300, 45,000 resource names and 14,000 asset signatures are ignored, which represent 0.08% of the unique resource names and 0.03% of the unique asset signatures considered.

To determine whether two applications $App_A$ and $App_B$ with respective feature sets $A$ and $B$ are similar, we use the Jaccard index $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$. The resulting score is a real number $[0, 1]$. If the Jaccard index is above a certain threshold $T$, the two applications compared are considered similar. We compute the Jaccard index separately for resource names and asset signatures, ignoring the blacklisted ones. Similar applications are grouped into clusters. For simplicity, we assume that applications typically derive from one other application, so that each application should only be included in at most one cluster. We therefore merge clusters whenever an application has multiple matches belonging to different clusters. Each cluster is assigned a victim application, which is the application with the most downloads, under the assumption that it is also the one most likely to have been the duplicated. We then merge the clusters based on resource names with those based on asset signatures. All of this is accomplished by first indexing all resource names and asset signatures in Elasticsearch, then querying Elasticsearch to match applications. The former is done once per APK and takes a couple of hours while the latter takes around 20 minutes on our cluster of ten machines. The number of similar applications is the sum of the size of each detected cluster, excluding their victim application.

Figure 7 shows the number of similar applications detected when varying the score threshold $T$ ranging from 0.6 to 1.0. A value of 1.0 represents an exact match of resources and assets, while 0.8 allows similar applications to have some differences. We compare the effectiveness of using both resource names and asset signatures versus only using one or the other. Asset signatures alone detect fewer similar applications because many applications have no assets at all. Figure 8 shows the distribution of clusters by sizes. The distribution shows that clusters with sizes larger than 300 are infrequent, suggesting 300 as a suitable cutoff to exclude common application libraries. We base the rest of our discussion on using $C = 300$ and $T = 0.8$. Using these parameters, there were 158,204 free duplicative applications in Google Play, roughly 25% of the free application content.

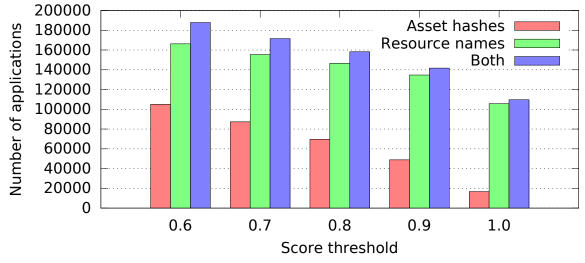


Figure 7: Similar apps vs score threshold (cutoff = 300).

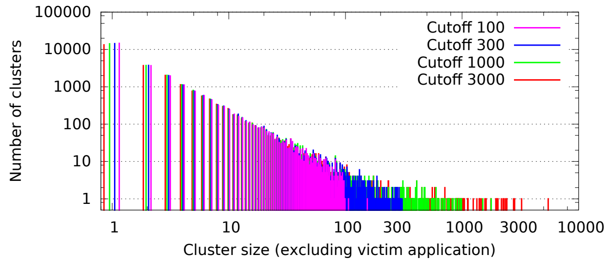


Figure 8: Number of clusters of similar apps vs cluster size (threshold = 0.8).

We then divided the similar applications taking into account developer information. Each application uploaded to Google Play is signed by a developer using a private key. We compared the ownership of each similar application by examining both the developer name associated with the application account and the certificate included in the application package. If either of these attributes match, applications are labeled as rebranded. If neither author attributes match, applications are labeled as clones. Out of 158,204 similar free applications, 115,896 were rebranded and 42,308 were clones. Sources of rebranding included changing the languages displayed in the user interface and reusing code from one application as a template for a new application, especially for wallpaper, trivia, and travel information applications. Sources of cloning included use of automated tools and wizard services, copying open source applications, contracting the same third party to develop applications for a range of clients in a specific industry, and plagiarism, though the latter is difficult to detect without knowing the relationships among developers. For a developer of an original application who therefore does know these relationships, PlayDrone can be useful to detect application cloning in various ways, from identifying plagiarized applications to monitoring statistics over all clones to gauge the health and popularity of an open-source project.

To evaluate the accuracy of our approach, we took a random sample of 400 applications flagged as similar and manually compared them to their corresponding victim application. Out of 400 applications identified as similar, manual inspection indicated that 5% were false positives that included similar resources and assets but actually differed on account of visible design and functionality differences. We also compared PlayDrone's method of similarity detection to a code analysis tool we built based on extracting a feature set of Android SDK methods from the DEX bytecode of an application [41]. Running the same 400 applications through the code analysis tool, only 79% of the manually verified applications were correctly identified.

PlayDrone's better performance can be explained by classifying code cloning techniques [32]. While most code analysis methods are able to identify similar applications with variations in identifiers, literals, types, whitespace, layout and comments (Type-1 and Type-2), they are less reliable in detecting similar applications with changed, added, or removed statements (Type-3) and ineffective at detecting similar applications implemented through different syntatic variants (Type-4). Because PlayDrone's detection technique is agnostic to the complexity of the code transformations used for cloning, it is the first system that can identify similar Android applications across all four clone types, including Type-3 and Type-4 clones. This makes PlayDrone not only fast and efficient, but also a more robust system for detecting application similarity in Google Play.

# 8. AUTHENTICATION TOKENS

The rise of the Web 2.0 architecture has seen a proliferation of cloud service APIs. Service to service communication is usually authenticated with secret tokens that are known only by the involved parties. When implemented as intended, secret tokens are never shared and are stored on trusted servers where they can be properly safeguarded. However, as these service to service protocols have been adapted to mobile applications, we have discovered using PlayDrone that developers are now embedding secret tokens directly into applications. While developers may believe their application sources are well guarded, the ease of decompilation and the widespread availability of mobile applications makes recovering secret tokens relatively simple. We discuss how we used PlayDrone to discover secret tokens used with Amazon Web Services (AWS) and several OAuth providers and demonstrate the potential for abuse of these tokens by malicious actors.

## 8.1 Discovering Tokens

We used PlayDrone's search engine to quickly probe application source code by searching words such as “secret” and found a large number of insecure tokens used for various services because developers often use constant names with the substring “secret” to identify their secret tokens. Our search results show that services often use tokens with service-specific formats. For example, the AWS API is accessed with an `AccessKeyId` which starts with the substring “AKIA”. Figure 9 shows a source search for all strings starting with “AKIA” revealing many AWS tokens. To extract authentication tokens, we created a flexible framework that searches for secret tokens in the decompiled Java source files of applications using regular expressions. Tokens usually come in pairs, typically a client ID and its corresponding secret key functioning similar to a username and password. For example, in the case of AWS, requests are signed using a 40 character string `SecretAccessKey`. To discover AWS credentials, we configured PlayDrone to find pairs of strings matching `AKIA[0-9A-Z]{16}` and `[0-9a-zA-Z/+]{40}` that are at most `5` lines apart.

Table 5 summarizes various authentication tokens for widely-used services that we found using PlayDrone. *Total Candidates* denotes the number of tokens found across the entire Google play store from the June 22, 2013 snapshot. *Unique Candidates* denotes the number of different tokens. The difference between total candidates and unique candidates can be explained by developer keys reuse, and various libraries directly embedding tokens in their SDKs. *Unique % Valid* denotes how many of the unique tokens from the June 22, 2013 snapshot were still valid on November 11, 2013. To test the validity of tokens, we sent authentication requests to their respective providers. Note that these results represent a conservative measure of the number of tokens in applications in Google Play as the simple search method does not detect tokens in obfuscated code because of its reliance on regular expression pattern matching.

## 8.2 Amazon Web Services

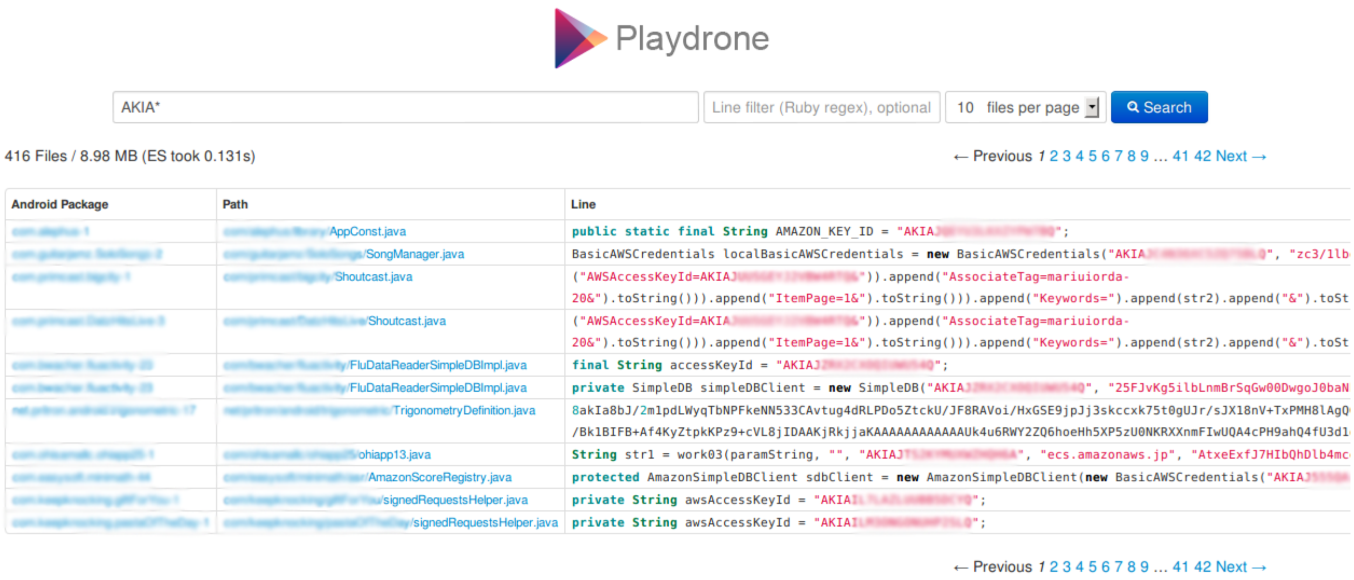
AWS provides various cloud computing resources that can be purchased by developers using AWS accounts and accessed by the developers' applications using AWS tokens associated with the respective AWS accounts. As shown in Table 5, we found 308 unique AWS tokens from the June 22, 2013 snapshot. Five month later, we tested the validity of these tokens by sending an AWS API request to count the number of storage buckets in the AWS Simple Storage Service (S3). We found 94% of the tokens were still valid five months later. These read-only API calls were carefully chosen to preclude any impact to AWS customer's data or resources. Amazon provides documentation describing best practices and a variety of ways to configure AWS tokens with different levels of privilege [2]. Despite this documented flexibility, we were surprised to find that even though some developers only intended their applications to use AWS tokens to access AWS Simple Database or Flexible Payment Services, the tokens embedded in the applications were root-level credentials providing access to all the other AWS services, including creating and shutting down Elastic Compute Cloud (EC2) instances or freely accessing S3 data.

Exposure of the AWS tokens can provide access to existing AWS resources, potentially leading to a range of confidentiality, integrity, and availability attacks, as well as the capability to allocate new resources at the owner's expense. With 288 valid tokens, an attacker could potentially setup a botnet of AWS EC2 instances. While AWS has a number of mechanisms to thwart such activities [30], usage patterns on AWS are elastic and inherently unpredictable, which may make it hard to detect stolen resources. Unless billing alerts are manually configured, billing statements will not reflect usage until the end of the billing cycle. Amazon recognizes the risks of embedding secret keys in Android applications and actively advises developers against this practice in their Android SDK documentation [5]. Additionally, AWS provides mechanisms for Android developers to securely leverage AWS from their application, such as AWS Token Vending Machine [4] and AWS Web Identity Framework [3]. The problem is that developers often find it simpler to embed tokens in their applications without being aware of best practices and understanding the resulting security risks.

Because of the potential for malicious use of the AWS tokens in Google Play, we reached out to Amazon to warn them of this security risk. Amazon responded quickly by identifying their affected customers based on the list of tokens we provided, and reaching out and working with their customers to resolve the security issues, though some mistakenly assumed that Amazon itself was scanning for secret keys in Android applications [8, 11, 28]. We also reached out to Google to ask them to scan for AWS and other tokens in applications as part of the Google Play application publication process to help protect users and developers. Google

|  | Amazon | Facebook | Twitter | Bitly | Flickr | Foursquare | Google | LinkedIn | Titanium |
|---|---|---|---|---|---|---|---|---|---|
| **Total candidates** | 1,241 | 1,477 | 28,235 | 3,132 | 159 | 326 | 414 | 1,434 | 1,914 |
| **Unique candidates** | 308 | 460 | 6,228 | 616 | 89 | 177 | 225 | 181 | 1,783 |
| **Unique % valid** | 93.5% | 71.7% | 95.2% | 88.8% | 100% | 97.7% | 96.0% | 97.2% | 99.8% |

Table 5: Credentials statistics from June 22, 2013 and validated on November 11, 2013. A credential may consist of an ID token and secret authentication token.



AKIA*

10 files per page

416 Files / 8.98 MB (ES took 0.131s)

← Previous 1 2 3 4 5 6 7 8 9 … 41 42 Next →

| Android Package | Path | Line |
|---|---|---|
| [redacted] | [redacted]/AppConst.java | `public static final String AMAZON_KEY_ID = "AKIA[redacted]";` |
| [redacted] | [redacted]/SongManager.java | `BasicAWSCredentials localBasicAWSCredentials = new BasicAWSCredentials("AKIA[redacted]", "zc3/1lb` |
| [redacted] | [redacted]/Shoutcast.java | `("AWSAccessKeyId=AKIAJ[redacted]")).append("AssociateTag=mariuiorda-`<br>`20&").toString())).append("ItemPage=1&").toString())).append("Keywords=").append(str2).append("&").toSt` |
| [redacted] | [redacted]/Shoutcast.java | `("AWSAccessKeyId=AKIA[redacted]")).append("AssociateTag=mariuiorda-`<br>`20&").toString())).append("ItemPage=1&").toString())).append("Keywords=").append(str2).append("&").toSt` |
| [redacted] | [redacted]/FluDataReaderSimpleDBImpl.java | `final String accessKeyId = "AKIAJ[redacted]";` |
| [redacted] | [redacted]/FluDataReaderSimpleDBImpl.java | `private SimpleDB simpleDBClient = new SimpleDB("AKIAJ[redacted]", "25FJvKg5ilbLnmBrSqGw00DwgoJ0baN` |
| [redacted] | [redacted]/TrigonometryDefinition.java | `8akIa8bJ/2mlpdLWyqTbNPFkeNN533CAvtug4dRLPDo5ZtckU/JF8RAVoi/HxGSE9jpJj3skccxk75t0gUJr/sJX18nV+TxPMH8lAgQ`<br>`/Bk1BIFB+Af4KyZtpkKPz9+cVL8jIDAAKjRkjjaKAAAAAAAAAAAUk4u6RWY2ZQ6hoeHh5XP5zU0NKRXXnmFIwUQA4cPH9ahQ4fU3d1` |
| [redacted] | [redacted]/ohiapp13.java | `String str1 = work03(paramString, "", "AKIAJ[redacted]", "ecs.amazonaws.jp", "AtxeExfJ7HIbQhDlb4mc` |
| [redacted] | [redacted]/AmazonScoreRegistry.java | `protected AmazonSimpleDBClient sdbClient = new AmazonSimpleDBClient(new BasicAWSCredentials("AKIAJ[redacted]` |
| [redacted] | [redacted]/signedRequestsHelper.java | `private String awsAccessKeyId = "AKIAI[redacted]";` |
| [redacted] | [redacted]/signedRequestsHelper.java | `private String awsAccessKeyId = "AKIAI[redacted]";` |

← Previous 1 2 3 4 5 6 7 8 9 … 41 42 Next →

Figure 9: PlayDrone's web interface to search decompiled sources showing Amazon Web Service tokens found in 130 ms.

automatically scans for some vulnerabilities, but plans to add checks and automated notices to developers for these specific issues as part of the Google Play application publication process. At Google's request, we provided some of our tools to help them develop these checks.

## 8.3 OAuth Tokens

Applications often request access to users' data to perform actions on their behalf. The standard protocol used by service providers to give access to users' data is OAuth. A third-party can register his application with an OAuth provider to receive OAuth client credentials consisting of a (`client_id`, `client_secret`) key pair. OAuth credentials are typically used in two ways. One way is to issue requests to the OAuth provider on behalf of the application, for example to ban a specific user or consume a rate-limited API (e.g. search). Another way is to request a user-specific `access_token` to perform actions on the user's behalf. For example, to acquire an access token, the third-party provides a link such as "Login with Facebook" on his website that would initiate the OAuth authentication process, including asking a user to grant permissions requested by the third-party application. Upon user acceptance, the third-party client receives an access token that can be used to read the user's Facebook friend list or post on his public feed.

When implemented correctly, the OAuth authentication protocol never reveals the tokens associated with a client's OAuth credentials. The tokens are stored on the third-party's server where it can be properly safeguarded. Requests can then be proxied through the third-party's server where the tokens reside. Unfortunately, developers often adapt this protocol to mobile applications by embedding OAuth tokens directly into their mobile applications without realizing their credentials are easily compromised through decompilation. Once an attacker acquires a secret OAuth token, a wide range of attacks can be performed as the targeted third-party application is open to impersonation. For example, an attacker can perform denial of service attacks on rate limited services, access and modify application settings, expose private user information, and launch phishing attacks in an attempt to get users' access tokens. Table 5 shows the total number of OAuth credentials we found in Android applications on Google Play for Facebook, Twitter, Bitly, Flickr, Foursquare, Linkedin, Google+, and Appcelerator's Titanium cloud services. After five months, over 90% of most of the OAuth credentials were still valid.

Focusing on Facebook and Twitter, we discovered 1,477 Facebook credentials and 28,235 Twitter credentials among all the free applications in Google Play. Finding 20 times as many Twitter tokens than Facebook tokens is surprising as Table 4 shows that the Facebook SDK is used twice as much as the Twitter4J library. One possible explanation is that until recently, Twitter encouraged developers to embed their secret tokens directly in client applications and even provided an official tutorial detailing this practice for Android using Twitter4J [36]. In contrast, as shown on the Facebook developer page [21], Facebook stresses the importance of never storing OAuth secret tokens in client applications: *Note that [this OAuth request] must never be made in client-side code or in an app binary that could be decompiled. It is important that your app secret is never shared with anyone. Therefore, this API call should only be made using server-side code.* To avoid the need of storing secret tokens on a mobile device, Facebook leverages the presence of the official Facebook Android application on Android devices. The Facebook SDK allows third-party applications to use Android intents to proxy requests for user access tokens through Facebook's Android application, where the user is already authenticated. To retrieve a user's access token with the Facebook SDK, the Android application identifier of a third-party application must be registered on its Facebook application settings page. This allows Facebook's Android application to respond to the third-party application's access token request by verifying the application identifier in the intent to the one officially registered online. Only the application identifier is needed and no secret key is compromised since it is retrieved at runtime. This technique relies on the robustness of Android since the source of the Android intent is assumed not to be spoofable.

Despite all the measures Facebook takes to make writing secure application easy, Table 5 shows that numerous developers still embed OAuth tokens in their applications and even seasoned developers have trouble following Facebook's simple security guidelines. For example, the popular Airbnb application still contained their Facebook, Google, LinkedIn, Microsoft, and Yahoo secret tokens from June 22, 2013 until well past November 11, 2013. Airbnb is the leader in peer-to-peer apartment rentals with more than 10 million users who are required to register on Facebook or Google+ to verify their identity. This is problematic in the case of Facebook because its API is too flexible by default, permitting user context queries to be performed in an application context. The default application settings allow an application to perform actions on behalf of a user at any later time once the user has authenticated the application. Even if an application does not subsequently retrieve a user access token, it still retains whatever permissions the user granted the first time the OAuth authentication process was run for the application. For example, we were able to access the email and friends list of Airbnb users using the URL: `https://graph.facebook.com/<user_id>?fields=email&access_token=<oauth_client_id>|<oauth_secret_key>`. Fortunately, the Airbnb application did not have permission to post on users' walls, otherwise we could write arbitrary content on millions of Facebook walls.

We notified Facebook of this problem and they immediately revoked the Airbnb OAuth credentials on Facebook. In a matter of hours, Airbnb published a new version of their application in Google Play, properly using the Facebook SDK for authentication and removing all secret tokens from the application. We also provided Facebook with a list of other Facebook tokens that we identified and Facebook promptly disabled Facebook access for those applications as well to protect Facebook users from potential unauthorized access. The affected applications would have to be updated using secure methods to regain functionality, resulting in service disruption for many users until updated applications are published in Google Play and users download the updated applications. Our results show that developers often ignore best practices, so it is important for OAuth providers to provide protection mechanisms such as limiting the service scope of tokens to help mitigate this security problem.

## 9. RELATED WORK

Besides Google, several companies maintain and publicize regular statistics about Google Play applications. AndroLib [6], AppBrain [7], and MixRank [29] offer services to help discover new applications and regularly publish statistics about the applications they have collected. d'Heureuse et al. [18] provide a temporal study of application statistics for Apple, Google, Microsoft and BlackBerry markets based on metadata collected from web scraping. Unlike previous studies and services, PlayDrone provides a scalable tool to crawl Google Play on a daily basis and a framework to analyze Google Play content at scale, including analysis based on Android application source code.

Some previous work has considered aspects of some of the issues we have studied. A number of approaches have explored how to detect similar Android applications, though not at the scale of the entire Google Play store. These approaches are computationally complex as they are based on application code analysis, either using pairwise comparisons [16, 13, 27, 42] or comparing applications to a subset of their closest neighbors [14, 41]. In contrast, PlayDrone indexes application resource signatures with Elasticsearch to efficiently match applications with common features.

Much work has focused on the security of Android applications and the presence of malware in Google Play, but this work has focused on possible compromises of user data and privacy on the Android devices themselves [20, 22, 25, 44]. While PlayDrone can be used as a tool to enable similar studies, we show how PlayDrone can be used to analyze a completely different type of security threat. By simply analyzing Android application content, we show that malicious attackers can go beyond Android devices to compromise server resources without even having users execute vulnerable Android applications. We present the first study to investigate these type of server-side vulnerabilities from mobile client code, showing evidence of how mobile developers leak secret tokens used in OAuth authorization and the Amazon Web Services API.

## 10. CONCLUSIONS

We have built PlayDrone, a system that uses various hacking techniques to circumvent Google security to successfully crawl Google Play. PlayDrone scales by simply adding more servers and is fast enough to crawl Google Play on a daily basis, downloading over 1.1 million Android applications and decompiling over 880,000 free applications. We use PlayDrone to perform a large-scale characterization of Android applications in Google Play and demonstrate how application content evolves over time, how even highly downloaded applications can be poorly rated, and that despite the large number of applications in Google Play, only a small percentage of free applications account for almost all downloads. We further show that (1) native libraries are heavily used by popular Android applications, limiting the benefits of Java portability and the ability of Android server offloading systems to run these applications, (2) 25% of Google Play is duplicative application content, and (3) Android applications contain thousands of leaked secret authentication keys which can be used by malicious users to gain unauthorized access to server resources through Amazon Web Services and compromise user accounts on Facebook. We worked with service providers, including Amazon, Facebook, and Google, to identify and notify customers at risk, and make the Google Play store a safer place. These results demonstrate that PlayDrone can serve as a useful tool to better understand Android applications and improve the quality of application content in Google Play.

## 11. ACKNOWLEDGMENTS

Don Bailey, Eric Davis, and Larry Rudolph provided helpful comments on earlier drafts of this paper. This work was supported in part by NSF grants CNS-1162447, CNS-1018355, CNS-0905246, and CCF-1162021.

## 12. REFERENCES

[1] Amazon Mechanical Turk. `https://www.mturk.com`.

[2] Amazon Web Services. IAM Best Practices, May 2010. `http://docs.aws.amazon.com/IAM/latest/UserGuide/IAMBestPractices.html`.

[3] Amazon Web Services. Creating Temporary Security Credentials for Mobile Apps Using Identity Providers. AWS Security Token Service, June 2011. `http://docs.aws.amazon.com/STS/latest/UsingSTS/CreatingWIF.html`.

[4] Amazon Web Services. Authenticating Users of AWS Mobile Applications with a Token Vending Machine. AWS Identity and Access Management, July 2013. `http://aws.amazon.com/articles/4611615499399490`.

[5] Amazon Web Services. Getting Started with the AWS SDK for Android. AWS SDK for Android, Sept. 2013. `http://docs.aws.amazon.com/mobile/sdkforandroid/gsg/Welcome.html`.

[6] AndroLib. `http://www.androlib.com`.

[7] AppBrain. `http://www.appbrain.com`.

[8] R. Bala. Amazon Is Downloading Apps From Google Play and Inspecting Them. Y Combinator Hacker News, Mar. 2014. `https://news.ycombinator.com/item?id=7491272`.

[9] Capistrano. `http://capistranorb.com`.

[10] Chef. `http://www.getchef.com`.

[11] R. Chirgwin. Amazon Is Decompiling Our Apps in Security Gaff Hunt, Says Dev. The Register, Mar. 2014. `http://www.theregister.co.uk/2014/03/31/dev_lashes_out_at_amazon_for_decompiling_his_app`.

[12] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proceedings of the 6th European Conference on Computer systems (EuroSys 2011)*, Apr. 2011.

[13] J. Crussell, C. Gibler, and H. Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *Proceedings of 17th European Symposium on Research in Computer Security (ESORICS 2012)*, Sept. 2012.

[14] J. Crussell, C. Gibler, and H. Chen. AnDarwin: Scalable Detection of Semantically Similar Android Applications. In *Proceedings of 18th European Symposium on Research in Computer Security (ESORICS 2013)*, Sept. 2013.

[15] Death by Captcha. `http://www.deathbycaptcha.com`.

[16] A. Desnos. Androguard. `https://code.google.com/p/androguard`.

[17] dex2jar. `http://code.google.com/p/dex2jar`.

[18] N. d'Heureuse, F. Huici, M. Arumaithurai, M. Ahmed, K. Papagiannaki, and S. Niccolini. What's App?: A Wide-Scale Measurement Study of Smart Phone Markets. *Mobile Computing and Communications Review*, 16(2):16–27, Apr. 2012.

[19] Elasticsearch. `http://www.elasticsearch.org`.

[20] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium*, Aug. 2011.

[21] Facebook. Login Security. `https://developers.facebook.com/docs/facebook-login/security`.

[22] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A Survey of Mobile Malware in the Wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM 2011)*, July 2011.

[23] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. AdRob: Examining the Landscape and Impact of Android Application Plagiarism. In *Proceedings of the 11th International Conference on Mobile Systems, Applications, and Services (MobiSys 2013)*, June 2013.

[24] E. Girault. Google Play Unofficial Python API. `https://github.com/egirault/googleplay-api`.

[25] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys 2012)*, June 2012.

[26] B. Gruver. smali/baksmali assembler/disassembler. `https://code.google.com/p/smali`.

[27] S. Hanna, L. Huang, E. X. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2012)*, July 2012.

[28] M. Kotadia. AWS Admits Scanning Android App in Secret Key Hunt. iTnews, Apr. 2014. `http://www.itnews.com.au/News/381432,aws-admits-scanning-android-app-in-secret-key-hunt.aspx`.

[29] MixRank. `http://www.mixrank.com`.

[30] R. Mogull. My $500 Cloud Security Screwup-UPDATED. Securosis Blog, Jan. 2014. `https://securosis.com/blog/my-500-cloud-security-screwup`.

[31] M. Perham. Sidekiq. `http://sidekiq.org`.

[32] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Sci. Comput. Program.*, 74(7):470–495, May 2009.

[33] S. Sanfilippo. Redis. `http://redis.io`.

[34] A. Thiel. Android-market-api. `https://code.google.com/p/android-market-api`.

[35] C. Tumbleson. Android-apktool. `http://code.google.com/p/android-apktool`.

[36] Twitter. Implementing the Twitter OAuth flow in Android. `https://dev.twitter.com/docs/implementing-twitter-oauth-flow-android`.

[37] N. Viennot. Java Library for JD-Core. `https://github.com/nviennot/jd-core-java`.

[38] N. Viennot. PlayDrone sources. `https://github.com/nviennot/google-play-crawler`.

[39] C. Warren. Google Play Hits 1 Million Apps. Mashable, July 2013. `http://mashable.com/2013/07/24/google-play-1-million`.

[40] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang. Refactoring Android Java Code for On-Demand Computation Offloading. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2012)*, Oct. 2012.

[41] W. Zhou, Y. Zhou, M. C. Grace, X. Jiang, and S. Zou. Fast, Scalable Detection of "Piggybacked" Mobile Applications. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY 2013)*, Feb. 2013.

[42] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy (CODASPY 2012)*, Feb. 2012.

[43] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP 12)*, May 2012.

[44] 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 *Proceedings of the 19th Annual Symposium on Network and Distributed System Security (NDSS 2012)*, Feb. 2012.