Andrubis – 1,000,000 Apps Later: A View on Current Android Malware Behaviors

Conference Paper · September 2014

DOI: 10.1109/BADGERS.2014.7

CITATIONS
127

READS
1,577

6 authors, including:

Christian Platzer
TU Wien
40 PUBLICATIONS **1,163** CITATIONS
SEE PROFILE

Victor van der Veen
Vrije Universiteit Amsterdam
14 PUBLICATIONS **432** CITATIONS
SEE PROFILE

ANDRUBIS - 1,000,000 Apps Later: A View on Current Android Malware Behaviors

Martina Lindorfer*, Matthias Neugschwandtner*, Lukas Weichselbaum*,
Yanick Fratantonio+,Victor van der Veen†, Christian Platzer*
*Secure Systems Lab, Vienna University of Technology, {mlindorfer,mneug,lweichselbaum,cplatzer@iseclab.org}
+Computer Security Lab, University of California, Santa Barbara, yanick@cs.ucsb.edu
†The Network Institute, VU University Amsterdam, v.vander.veen@vu.nl

***Abstract*—Android is the most popular smartphone operating system with a market share of 80%, but as a consequence, also the platform most targeted by malware. To deal with the increasing number of malicious Android apps in the wild, malware analysts typically rely on analysis tools to extract characteristic information about an app in an automated fashion. While the importance of such tools has been addressed by the research community, the resulting prototypes remain limited in terms of analysis capabilities and availability.**

In this paper we present ANDRUBIS, a fully automated, publicly available and comprehensive analysis system for Android apps. ANDRUBIS combines static analysis with dynamic analysis on both Dalvik VM and system level, as well as several stimulation techniques to increase code coverage. With ANDRUBIS, we collected a dataset of over 1,000,000 Android apps, including 40% malicious apps. This dataset allows us to discuss trends in malware behavior observed from apps dating back as far as 2010, as well as to present insights gained from operating ANDRUBIS as a publicly available service for the past two years.

I. INTRODUCTION

Android is undoubtedly the most popular operating system for smartphones and tablets with a market share of almost 80% [38]. Its widespread distribution and wealth of application (app) distribution channels besides the official Google Play Store, however, also make it the undisputed market leader when it comes to mobile malware: according to a recent estimate, as many as 97% of mobile malware families target Android [32]. Estimations by anti-virus (AV) vendors as to the number of Android malware in the wild vary widely. McAfee reports about 68,000 distinct malicious Android apps [50] and Sophos collected a total of 650,000 unique Android malware samples to date, with 2,000 new samples being discovered every day [59].

Google reacted to the growing interest of miscreants in Android by introducing *Bouncer* [44], a service that transparently checks applications submitted to the Google Play Store for malware. Google reported that this service led to a decrease of the share of malware in the Play Store by nearly 40% since its deployment in February 2012. However, a common practice among malware authors is *repackaging* popular apps with malicious code and publishing them in alternative app markets that do not employ effective security measures. In fact, in line with findings from F-Secure [32], we found alternative markets hosting up to 5-8% malicious apps [41].

Consequently, a significant amount of research has focused on analyzing and detecting Android malware, with numerous tools and services being proposed and operated by researchers [24,29,56,58,66] and security companies [8,10,16]. Automated and reliable solutions are required to deal with the growing number of mobile malware samples. Analysis capabilities and availability of proposed research prototypes, however, remain limited. A recent study on state-of-the-art Android malware analysis techniques showed that among the 18 analysis tools surveyed, many systems were not available online or were no longer being maintained [52]. In an evaluation on the susceptibility of Android dynamic analysis sandboxes against evasion, Vidas et al. [64] only found three publicly accessibly systems (including the one presented in this paper).

In order to provide a large-scale analysis solution to the research community we propose ANDRUBIS, a hybrid Android malware analysis sandbox that generates detailed analysis reports of unknown Android apps based on features extracted during static analysis and behavior observed through dynamic analysis during runtime. Similar to the spirit of AndroTotal [47], a service that allows researchers to scan Android apps with a number of AV scanners, we operate ANDRUBIS as a publicly available service and data collection tool that allows us to collect and share a comprehensive and diverse dataset of both Android malware and benign apps.

We built ANDRUBIS as an extension to the dynamic Windows malware analysis sandbox ANUBIS [3,21]. ANUBIS has collected a dataset of Windows malware samples that represent a comprehensive and diverse mix of malware found in the wild since 2007 [20]. ANDRUBIS itself has been online since June 2012 and has analyzed over 1,000,000 unique Android apps so far. Based on AV labels collected from VirusTotal [15], we estimate 40% of those apps are malware (not including adware). We further assess the age of apps in our dataset and categorize them by year starting in 2010 allowing us to identify trends in Android malware behavior. Similar to the dataset of ANUBIS, our dataset represents apps from a variety of sources, with apps collected from crawls of the Google Play Store and alternative markets, sample exchange with other researchers, torrents and direct downloads, and anonymous user submissions.

The tight integration of our analysis with the existing ANUBIS infrastructure for analyzing Windows malware provides two main benefits: (a) we can take advantage of existing sample exchange agreements as malware feeds often contain both Windows and mobile samples, and (b) adapt existing analysis techniques for the use with Android apps. For example, experiments applying clustering [22] to Android apps yielded promising results and showed that the feature set produced by ANDRUBIS is rich enough to allow researchers to build various post-processing methods upon [65]. This last aspect is of particular importance as we envision ANDRUBIS to be integrated with other analysis tools to foster sample exchange and provide deeper insights into Android malware behavior. ANDRUBIS has already been integrated with different tools, such as AndroTotal to provide an additional analysis report to AV scanner results. Similarly, ANDRUBIS provides a seed of malicious apps to AndRadar [41], which it uses to scan the Google Play Store and 15 alternative markets and that in turn allows us to collect valuable meta information for our dataset. Besides shedding light on publishing habits of malicious app authors we can gain insights on an app's distribution across markets and popularity according to user ratings and download numbers. In the future, we also hope to gain insights into the infection rates of user's devices by analyzing which apps are submitted through our mobile app interface from user's phones. Thereby we could verify reports of the small infection rates of less than 0.3% reported in related work [40,46,62].

In summary, we make the following contributions:

- We introduce ANDRUBIS, a fully automated analysis system that combines static and multi-layered dynamic approaches to analyze unknown Android apps.
- We provide ANDRUBIS as a large-scale analysis service to the research community, accepting public submissions at `https://anubis.iseclab.org` and through a mobile app [2].
- By collecting apps from a variety of sources we build a comprehensive and diverse dataset of over 1,000,000 Android apps, including over 400,000 malicious apps.
- We present insights gained from providing our service for the past two years and we discuss trends in malware behavior observed from apps dating back as far as 2010.

II. ANDRUBIS SYSTEM OVERVIEW

In this section we detail the building blocks of ANDRUBIS and how they contribute to forming a complete picture of an app's characteristics. ANDRUBIS follows the *hybrid analysis* approach and is based on both static and dynamic analysis complementing and guiding each other: results of the static analysis are used to perform more efficient dynamic analysis. Figure 1 shows an overview of the individual components of ANDRUBIS and how they relate to one another. Users can submit apps either through our web interface, automated batch submission scripts, or directly from their phone through a dedicated mobile app. We then subject each app to the following three analysis stages:

1) **Static Analysis.** During this stage we extract information from an app's manifest and its bytecode.
2) **Dynamic Analysis.** This core stage executes the app in a complete Android environment, and its actions are monitored at both the Dalvik and the system level.
3) **Auxiliary Analysis.** We capture the network traffic from outside the Android OS and perform a detailed network protocol analysis during post-processing.

A. STATIC ANALYSIS

Android apps are packaged in *Android Application Package* (APK) files, a ZIP archive based on the JAR file format. An APK file contains an app's bytecode stored in Dalvik Executable (DEX) format, resources, such as UI layouts, as well a manifest file (`AndroidManifest.xml`). The manifest is mandatory and without its information an app cannot be installed or executed. Thus, as a first step, we unpack the archive and parse meta information from the manifest, such as requested permissions, services, broadcast receivers, activities, package name, and SDK version. In addition we examine the actual bytecode to extract a complete list of available Java objects and methods.

We use the information gathered during static analysis to assist in automating the dynamic analysis, mainly during the stimulation of an app's components. Furthermore, an app requesting dangerous permissions can be indicative of malicious behavior. Therefore, we extract the permissions that are requested as well as the permissions that are actually used in the app's bytecode to later compare them to permissions used during runtime.

B. DYNAMIC ANALYSIS

Being designed for smartphones and tablets, Android is predominantly deployed on ARM-based devices. Since the underlying architecture should be of no difference to the apps, we decided to build our sandbox for the ARM platform, the typical environment for Android, and chose a QEMU-based emulation environment capable of running arbitrary Android OS versions. Since Android apps are based on Java, we instrument the underlying virtual machine (VM), called the Dalvik VM, and record activities happening within this environment. This allows us to monitor

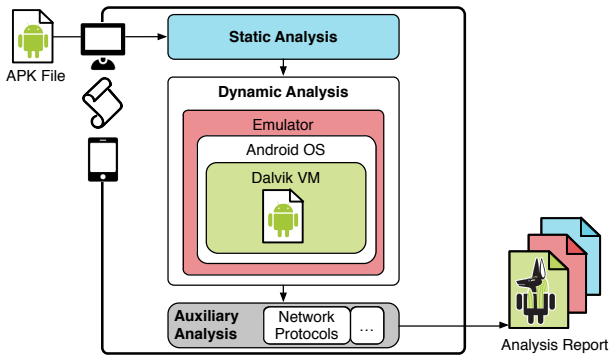

Fig. 1: System overview of ANDRUBIS.

the file system and network, as well as phone events, such as outgoing SMS messages and phone calls, and the loading of additional DEX or native code during runtime. For a comprehensive analysis, however, these capabilities are not sufficient. Therefore, we implemented the following additional analysis facilities:

- **Stimulation.** Due to the event-driven nature of Android, comprehensive input stimulation is invaluable for triggering interesting behavior from the app under analysis.
- **Taint Tracking.** To track privacy sensitive information ANDRUBIS uses taint tracking at the Dalvik level [30], which enables us to detect the leakage of sensitive information.
- **Method Tracing.** We record invoked Java methods, their parameters, and their return values. Combined with our static analysis, we can use method traces to measure the code covered during an analysis run, e.g., for evaluating and improving our stimulation engine.
- **System-Level Analysis.** To provide means for analysis beyond the scope of the Dalvik VM, we implemented an introspection-based solution at the emulator level. This enables us to monitor the system from outside the Android OS and to track system calls of native libraries and root exploits.

The output produced by the method tracer and the system-level analysis is not displayed in the public ANDRUBIS analysis report. As these tasks are quite resource-intensive and the log files are quite large, we only perform them on a subset of samples and provide them on an on-demand basis for researchers and analysts rather than ordinary users.

The remainder of the sandboxing system (network setup and traffic capturing, host environment, database, etc.) is comparable to conservative analysis systems. To mitigate potentially harmful effects of our analysis environment to the outside world while allowing apps under analysis to use the network, we took precautions to prevent apps from executing DoS attacks, sending spam e-mails or propagating themselves over the network. This part is based on our experience with Windows malware analysis and proved to be effective with ANUBIS in the past [21].

1) Stimulation: The purpose of stimulation is to exhaustively explore the functionality of an app. One major drawback of dynamic analysis in general is the fact that only a few of all possible execution paths are traversed within one analysis run. Furthermore, Android apps can have multiple entry points besides the main activity, which is displayed to the user when an app is launched, so that apps can react to system events or interact with each other. Luckily, since the app's manifest lists the various components (activities, services, and broadcast receivers), we can stimulate them individually. Additionally, we can initiate common events that malicious apps are likely to react to.

Our stimulation approach includes the following sequence of events: after the initialization of the emulator, ANDRUBIS installs the app under analysis and starts the main activity. At this point,

all predefined entry points are known from static analysis. During runtime ANDRUBIS keeps track of dynamically registered entry points, enabling it to perform the following stimulation events:

Activities. An activity provides a screen to interact with and defines the interaction sequences and UI layout presented to the user. Activities have to be registered in the manifest and cannot be added programmatically. Therefore, by parsing the manifest, ANDRUBIS has full knowledge about an app's activities and invokes each activity separately, effectively iterating all existing dialogs within an app.

Services. Background processes on the Android platform are usually implemented as services. In contrast to activities, they come without a graphical component and are designed to provide background functionality for an app. Naturally, they are also of interest to malware authors, as they can be used to implement communication with command and control (C&C) infrastructures of botnets, leak personal information, or forward intercepted text messages to an adversary. Again, all services used by an app must be listed in the manifest. Their existence, however, does not automatically mean the service is started: to save battery life and preserve memory, services have to be started on demand, with a lifetime defined by the programmer. For ANDRUBIS we utilize a customized *Activity Manager* to iterate and start all listed services of an app automatically after it has been installed.

Broadcast Receivers. Other possible entry points for Android apps are broadcast receivers. Broadcast receivers are basically event handlers used to receive events from the system or other apps on the Android platform. For example, a broadcast receiver for the `BOOT_COMPLETED` event can be registered to start an app after the phone has finished its boot sequence or a broadcast receiver for the `SMS_RECEIVED` event can be registered to intercept incoming SMS messages.

Just like services and activities, broadcast receivers can be registered in the manifest. However, for broadcast receivers this is not mandatory. In order to provide the possibility to react to certain events, or to provide communication with other apps dynamically, they can also be registered and deregistered at runtime. Therefore, we intercept the calls to `registerReceiver()` to obtain a list of dynamically registered event handlers that we can stimulate. Similar to the previous stimuli, ANDRUBIS uses the *Activity Manager* to invoke all statically registered broadcast receivers found in the manifest as well as the ones that have been dynamically registered.

Common Events. A far superior method compared to directly stimulating broadcast receivers with a targeted event is to emulate the events that apps might react to and especially malicious apps are likely to be interested in. Thus, we broadcast events such as boot completion, incoming SMS and phone calls, changes in the GPS lock, and changes in the WiFi and cellular connectivity. In contrast to directed stimuli, these events occur at the system level and thus also trigger receivers of the Android OS itself. That, in turn, avoids causing inconsistent states the OS would have to recover from when only invoking the event handler registered by an app.

Application Exerciser Monkey. The remaining elements that need to be stimulated are actions based on user input (button clicks, file upload, text input, etc.). For this purpose, we use the Application Exerciser Monkey, which is part of the Android SDK and generates semi-random user input. Originally designed for stress-testing Android apps, it randomly creates a stream of user interaction sequences that can be restricted to a single package name. While the triggered interaction sequences include any number of clicks, touches, and gestures, the monkey specifically tries to hit buttons. As some use cases might require repeatable analysis runs without any random behavior introduced by the monkey, we optionally provide a fixed seed in order to always trigger the same interaction sequences.

2) Taint Tracking: Data tainting is a double-edged sword when it comes to malware analysis. On one hand, it is the perfect tool to keep track of interesting data; on the other hand, it can be tricked quite easily if a malware author is aware of this mechanism within an analysis environment [25]. By leaking data through implicit flows, for instance, it would be possible to circumvent tainting. Furthermore, enabling data tainting always comes at the price of additional overhead to produce and track taint labels. Still, the possibility to track explicit flows of sensitive data sources, such as contacts, phone-specific identifiers, and the location, to the network is a valuable property of a dynamic analysis system. ANDRUBIS leverages *TaintDroid* [30] to track such sensitive information across application borders in the Android system. The introduced overhead in processing time of approximately 15% [30] is also acceptable for our purposes. As a result, ANDRUBIS can log tainted information as it leaves the system through three sinks: network, SMS, and files on disk.

3) Method Tracing: For an extensive analysis of Java-based operations, we extended the existing Dalvik VM profiler capabilities to incorporate a detailed method tracer. For a given app we log the executed Java methods on a per-thread basis. The method trace contains method names and their corresponding classes, the object's `this` value (if any), all provided parameters and their types, return values, constructors, exceptions and the current call depth. For non-primitive types, the tracer looks up and executes the object's `toString()` method, which is then used to represent the object.

Together with the output gained from system-level analysis (described in the next section), the fine-grained method traces can assist reverse engineering efforts, serve as input to machine learning algorithms, or they can be used to create behavioral signatures. Furthermore, by mapping the method trace to permissions utilizing a permission mapping, such as the ones provided by PScout [18] or Stowaway [34] we can determine the permissions an app actually used during runtime.

Our main incentive to integrate method tracing, however, is to measure the code covered during the individual phases of the stimulation engine. To this end, we first compile a list of executed method signatures. We then map this list against the list of functions extracted during static analysis based on their Java method signature excluding parameter types and modifiers, i.e., on their `<package>.<subpackage>.<class>.<method>` representation. Finally, we compute the code covered as the overall percentage of functions that were called during the dynamic analysis. However, apps may contain numerous functions that, during a normal execution, will never be invoked, such as localization and in-app settings or large portions of unused code from third-party libraries. Thus, for a less conservative and more realistic code coverage computation we can whitelist known third-party APIs or limit the computation to the main app package's code.

4) System-Level Analysis: In addition to monitoring the Dalvik VM, and in contrast to most related work on Android malware, ANDRUBIS also tracks native code execution. By default, Android apps are Java programs, being distributed as a DEX file within an APK file. Hence, the default way of programming for the Android platform and executing Android apps is by running Dalvik bytecode within the Dalvik VM. However, Android apps are not limited to Dalvik bytecode and can also execute system-level code by loading native libraries via the Java Native Interface (JNI). While this functionality is mainly intended for performance-critical use cases, such as displaying 3D graphics, apps are not restricted to loading the

native libraries shipped with the Android OS; instead they can also ship and load their own native libraries and, in turn, execute arbitrary system-level code. Naturally, the execution of this code takes place outside of the Dalvik VM and, thus, the behavior of this code is invisible to the analysis at Dalvik VM level. For malicious apps the use of native code is attractive as the possibilities to perform malicious activities, such as the usage of exploits to gain root privileges, are far greater than within the Dalvik VM – making system-level analysis indispensable for drawing a complete picture of an apps's behavior. In addition, Google recently introduced the new Android Runtime (ART) [37] that compiles Dalvik bytecode to native code at installation time. With the replacement of Dalvik with ART as the default runtime in upcoming Android OS releases [61], the capability to perform system-level analysis will gain further importance.

Being based on Linux, there are a couple of ways to implement system-level instrumentation in Android, such as using `LD_PRELOAD`, `ptrace` or a loadable kernel module. We decided to use the most transparent and non-intrusive way – virtual machine introspection (VMI). With VMI our analysis code is placed outside of the scope of the running Android OS, right in the emulator's codebase, and tracks the complete list of system calls performed by the emulator as a whole, including the OS. To capture the system-level behavior of the app under analysis, we ultimately need to extract the system calls executed by the library code that was loaded via JNI. To this end, we intercept the Android dynamic linker's actions in order to track shared object function invocations. System call tracking bundled with this information enables us to associate system calls with invocations of certain functions of loaded libraries. Android assigns a unique user ID (UID) to every app and runs the app as that user in a separate process – allowing us to associate system calls with apps based on the process UID. The result is a list of native code events caused by just the specific app under analysis.

C. Auxiliary Analysis

Network traffic is one of the most essential parts when establishing malware-detection metrics, with C&C communication being of particular importance. According to studies performed in production environments [36], more than 98% of Windows malware samples established a TCP/IP connection. Thus, in addition to tracking sensitive information to network sinks via taint tracking, we also capture all the network activity during analysis regardless of the performed action or the app causing it. This is necessary since apps not requesting and using the `INTERNET` permission themselves, can still use other installed apps like the browser, to send data over the network. Another way to transmit network data without requesting the appropriate permissions is by exploiting the Android OS and circumventing the permission system as a whole.

During post-processing we perform a detailed *Network Protocol Analysis* that extracts high-level network protocol features from the captured network traffic suitable for identifying interesting samples. Currently, we focus on the well-known and often used protocols DNS, HTTP, FTP, SMTP, and IRC.

III. Andrubis as a Service

In this section we present insights gained from offering Andrubis as a publicly available service for the past two years and the dataset of apps we collected along the way.

A. Submission Statistics

We base our analysis on a dataset collected over the span of exactly two years, between June 12, 2012 and June 12, 2014. We distinguish between *submissions* (all analysis requests Andrubis received), *tasks* (submissions for which the analysis was performed), and *samples* (unique apps based on their MD5 file hash). Overall, Andrubis received 1,778,997 unique submissions. Since Andrubis usually returns cached analysis reports in case an app is submitted multiple times (unless a user requests a re-analysis of a previous task), it performed analysis tasks for 1,073,078 (around 60%) of submissions. In total Andrubis received and analyzed 1,034,999 (58.18%) unique samples.

TABLE I: Users categorized by their number of submissions and proportion of all submissions.

Category		# of Users	% of All Submissions
Bulk	(10,000+)	15	95.82%
Large	(1,000-10,000)	13	2.47%
Medium	(100-1,000)	34	0.59%
Small	(10-100)	247	0.41%
Single	(1-10)	7,966	0.72%

To put the number of Android samples into perspective we compare them to overall submissions to Anubis. During our observation period Anubis received a total of over 22 million samples, Android apps thus amount to close to 5% of overall samples. However, since the submission interface only assigns submissions of ZIP archives containing `classes.dex` and `AndroidManifest.xml` to Andrubis, we only report numbers on APK files and not submissions of related files such as stand-alone DEX classes. A large number of samples comes from malware feeds as part of exchange agreements. We receive feeds with Android apps from nine sources, most of them submitting both Windows executables and Android apps – with the exception of AndroTotal almost exclusively submitting APK files. Other malware feeds from security researchers and AV vendors contain from as little as 1% to up to 37% Android apps. The largest sample feed contributing more than five million samples in the observation period contains around 10% Android submissions.

Figure 2 shows the weekly number of total submissions, submissions through sample exchanges, i.e., semi-regular feeds of samples, new samples and analyzed samples. Submissions peaked in August 2012 and January 2013, when we received bulk submissions from Google Play crawls and in July 2013 when one feed submitted a higher than usual amount of samples. In November and December 2013 AndRadar [41] started submitting a backlog of apps before switching to a regular feed of apps. Besides a power outage in January 2014 Andrubis has been operating reliably and analyzed up to the current maximum capacity of 3,500 new apps per day.

In order to estimate the number of different users using our service, we distinguish them either by their username, or in case of anonymous submissions, by their IP address. Users can register for an account in order to gain special privileges, such as a higher priority for their tasks or the ability to force the re-analysis of an app. The account management is shared with Anubis, but 152 registered users submitted at least one Android app. With anonymous submissions coming from 8,123 unique IP addresses we estimate that 8,275 unique users from 130 different countries are using Andrubis. The majority of submissions come from registered users, with 15 individual users amounting to over 95% of total submissions, and only 38,905 (3.76%) of submissions coming from anonymous sources. Table I categorizes users by their number of submissions from single submitters with less than 10 submission to "power users" with more than 10,000 submissions. The maximum amount of 557,559 submissions for a single user stems from one of the aforementioned malware feeds.

Figure 3 shows the number of different sources, i.e., the number of distinct users that submitted a particular app: around 70% of apps were submitted by only one user and only 1.5% of apps were submitted by more than three distinct users. In

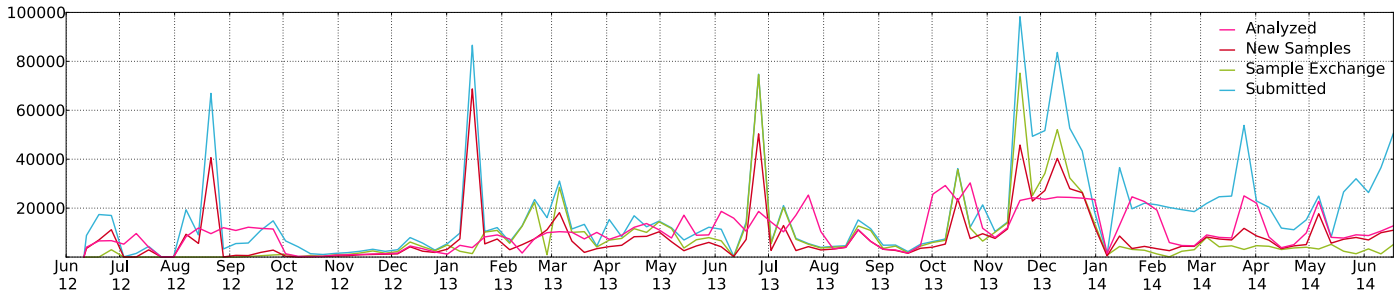

Fig. 2: Weekly number of total submissions, submissions through sample exchanges, new and analyzed samples.

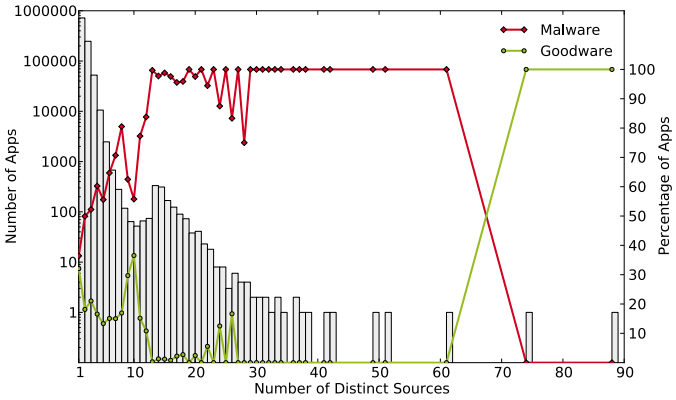

Fig. 3: Number of unique users submitting an app and percentage of goodware and malware submitted by N users.

general, malicious samples were submitted more frequently than goodware apps (with the exception of the top two apps): over 80% of apps submitted by more than five different users belong to the malware category. The popular game Flappy Bird was also the "most popular" app submitted to ANDRUBIS by 88 different users. The second most submitted app is the alpha version of our mobile interface to ANDRUBIS, with which users can submit apps directly from their phones (it is currently available for download on the ANDRUBIS's web interface). However, the remaining most popular apps (and all other apps submitted 26 times and more) are part of malware corpora, such as Contagio [6] and the Android Malware Genome Project [72].

B. Analysis Results and Limitations

Overall, ANDRUBIS successfully analyzed 91.67% of all apps. For the remaining samples, 0.34% failed due to bugs in our analysis environment and 7.99% of samples failed to install in our sandbox due to various reasons, such as the APK file being corrupt or the app exceeding the API level of the Android OS version installed in our sandbox. ANDRUBIS currently runs Android 2.3.4 Gingerbread and thus only supports apps with a minimum required API level ≤ 10. We know that 0.78% of apps require a newer OS version, and for 6.66% of samples we could either not parse the manifest or they did not specify an API level. However, this has no significant impact on malware analysis as of now. Instead, it is mainly a concern for goodware, of which 2.11% (6,099) require a higher API level, while only 0.10% (439) of malicious apps fail for this reason. Such a behavior by malware authors is expected: their malicious apps require a lower API level in order to maximize the potential user base for their apps, and, in turn, their profit. This is also confirmed by Figure 8 in the Appendix, which shows that malware authors are much slower in adopting new API levels than goodware authors.

C. Scalability

Currently, ANDRUBIS is capable of processing around 3,500 new apps per day, i.e., apps that have never been analyzed before and for which no cached report is available. The analysis of an app takes around 10 minutes, with 240 seconds analysis runtime in the sandbox plus an additional 387.27 seconds on average for pre- and post-processing. Pre-processing includes setting up the emulator and loading the Android OS snapshot, installing the app, parsing the manifest and performing static analysis on the APK. Post-processing includes extracting protocol information from the network traffic and preparing the final analysis report.

Judging from our experience running the Windows malware analysis service ANUBIS and similar to Andlantis [23], ANDRUBIS scales well by simply adding new workers to handle the analysis of new samples should submissions increase. However, already with the current throughput of over 100,000 apps per month, ANDRUBIS is capable of analyzing samples at market scale. For example, Google Play, the largest app store (by far), added, on average, 37,500 new apps per month in the last year, with peaks of up to 85,000 new apps in December [5]. When it comes to malware, Android still falls far behind the plethora of Windows samples circulating in the wild: Sophos estimates 2,000 new Android malware samples are being discovered each day [59], a number ANDRUBIS can handle in the current configuration and setup comfortably.

D. Sample Sources

One limitation of a public web interface allowing anonymous submissions is the lack of meta information associated with submitted apps. Since the majority of apps are submitted by registered users, however, we can associate them to sample exchanges, part of our own crawling efforts, or the integration of tools, such as AndRadar. Table II in the Appendix summarizes the number of apps from each source, as well as the proportion of benign and malicious apps (see the next section on how we separated goodware from malware). The apps in our dataset originate from the following eight sources:

Sample Exchange. These apps make up the majority of our dataset and come from sample sharing with other researchers. Most of the feeds are part of long-standing sample exchanges that started with Windows samples, but now also include Android samples, too.

Google Play. We initially crawled 100,000 apps from the Google Play US Store during May and June 2012. Additionally, since December 2013, we receive apps crawled from AndRadar that match a seed of malicious apps and are located in the Google Play Store. In April 2014, we started fetching the top apps overall, top new apps and top apps per category (limited to 500 entries each by Google Play) from the Google Play US and AT Store on a daily basis.

Alternative Markets. These apps are crawled by AndRadar from 15 alternative markets, including seven Chinese and one Russian market. This dataset is biased towards malware since AndRadar aims at locating malicious apps.

VirusTotal. We regularly download samples from VirusTotal. However, this dataset not only contains malware, but also a small percentage of samples labeled as adware as well as some samples not detected by any AV scanner.

Malware Corpora. This is a collection of manually gathered malware samples we encountered over time as well as samples from vetted malware corpora, such as the Contagio Mobile Malware Dump [6], the Android Malware Genome Project [72], and Drebin [17]. However, besides the relatively small Contagio set (470 samples) that is regularly updated, which, in turn, makes comparison hard, available malware corpora are already quite dated: the 1,200 samples (49 different families) from the Android Malware Genome Project were collected from August 2010 to October 2011, the 5,560 apps (179 families, including the Genome Project) from the Drebin dataset were collected in the period of August 2010 to October 2012.

Torrents. We downloaded apps from `isohunt.com`, `thepiratebay.se`, and `torrentz.eu` for which the torrent had at least ten seeders. To avoid distribution of copyright-protected content, our torrent client did not upload any data at all.

Direct Downloads. We downloaded a set of apps through direct downloads from various one-click hosters, including `filestube.com` and `iload.to`.

Unknown. These apps stem from anonymous user submissions and thus we do not have any information where they originate from.

E. Collected Dataset

The dataset gathered from samples submitted to ANDRUBIS allows us to perform a longitudinal analysis of Android app features in general and features specific to benign apps and malicious apps. First, however, we need to separate the dataset into subsets. Since the primary goal of ANDRUBIS is to provide researchers with a comprehensive static and dynamic analysis report of an app, not to automatically identify apps as goodware or malware, we have to rely on AV signatures as our ground truth:

Goodware. We classify apps as goodware if they do not match any AV signature from VirusTotal's AV scanners. Goodware apps make up 27.90% of our dataset.

Malware. We classify apps as malware if they match at least t AV signatures. We experimented with different settings for the threshold t and settled on at least 5 AV labels, ignoring all AV labels indicating adware. With thresholds $t>5$ a large portion of apps exhibiting malicious behavior, such as exploiting the Master Key vulnerabilities (see Section IV-A7), would have been missed. Malware apps make up 41.15% of our dataset.

All. In addition to goodware and malware our complete dataset contains 30.95% other apps that are detected by 1 to 5 AVs or that are classified as adware.

Estimation of Release Date. In order to perform any kind of longitudinal analysis on our dataset and categorize apps by the year of their release, we need to estimate the age of each sample. Besides this yearly division of our dataset we also would like to have a more precise estimate to allow for a fine-grained evaluation, such as the time it takes for us to receive and analyze samples after they have been released. We estimate the age of an app from four data points: (1) the last modification date of the APK file (`zip_modification_date`), (2) the release date of the SDK indicated by the minimum required API level (`sdk_release`), (3) the date a sample was first published in any of the markets monitored by AndRadar (`market_release`), and (4) the date a sample was first submitted to ANDRUBIS (`first_seen`).

For (1), the last modification date of the APK file, we parse the timestamp for the archive member that was modified last, usually the app's certificate, from the ZIP central directory file header. Naturally, this date can be tampered with, as evidenced by 273 apps feigning a modification date in 1980, the first year the ZIP file format supports for timestamps, further 9,703 before the first Android version was released in 2008, as well as 86 apps dated in the future, up to the year 2107. For (2), we parse the minimum required API level from an app's manifest and map it against the Android version history [1]. For (3), we have information from AndRadar for 68,197 apps in our dataset, since not all markets specify the date an app was uploaded and we do not want the overall release date of an app but the date when a specific version (based on the MD5 file hash) was released.

In general, we trust the modification dates extracted from the ZIP header as we only encountered relatively few outliers exhibiting unrealistic modification dates. However, we sanitize the `zip_modification_date` by checking the `sdk_release` as a lower bound for when the app could have been released in case the ZIP timestamp was predated, and the `market_release` as an upper bound when the app was first seen in the wild in case the app was postdated. In the normal case the app requests a specific API level after the corresponding SDK was released and the app is built before it is released to the public, e.g., an application market. In this case we estimate the release date (`apk_date`) as the date the ZIP was created:

$$\texttt{sdk\_release} < \texttt{zip\_modified} < \texttt{market\_release}$$
$$\texttt{apk\_date} = \texttt{zip\_modified}$$

For 10,000 apps (1.04%) the ZIP file was created before the corresponding SDK was released. This could be either due to the ZIP file header being tampered with or the app being part of an alpha/beta test of an unreleased SDK. Since an app cannot be installed on devices if it requires a higher API level than the currently available Android OS version, we assign the date of SDK release to the release date:

$$\texttt{zip\_modified} < \texttt{sdk\_release}$$
$$\texttt{apk\_date} = \texttt{sdk\_release}$$

In only six cases the market release date indicates that the app was published before the requested SDK level was released. This could be due to an error on the developers side, unintentionally requesting a higher API level than required. In this case we choose the maximum of the SDK release and the ZIP creation date:

$$\texttt{market\_release} < \texttt{sdk\_release}$$
$$\texttt{apk\_date} = \max(\texttt{sdk\_release}, \texttt{zip\_modified})$$

Around 5,000 apps (0.50%) were published in a market before the ZIP was last modified. Since this means that the ZIP header obviously has been tampered with, we set the release date to the market release as the first date we saw the app in the wild:

$$\texttt{market\_release} < \texttt{zip\_modified}$$
$$\texttt{apk\_date} = \texttt{market\_release}$$

We now can use the `apk_date` to estimate the *analysis delay*, the time it takes between an app being released and the app being submitted to ANDRUBIS. Figure 4 shows the CDF of the analysis delay for the first and second year of operation. Within the first year we only saw 15% of samples within one week of their release for both malware and goodware. In the second year this number significantly increased to over 40% for goodware apps, in part due to our crawling of the popular new apps from the Play Store on a daily basis. In the first year ANDRUBIS analyzed 60%-70% of all samples within the first three months. This number increased for apps of all categories to 80% in the second year. Finally, the number of apps analyzed within six months of their release increased from 2012 to 2013 by 10 percentage points for apps of all categories, to close to 90% of goodware and 95% of malware samples.

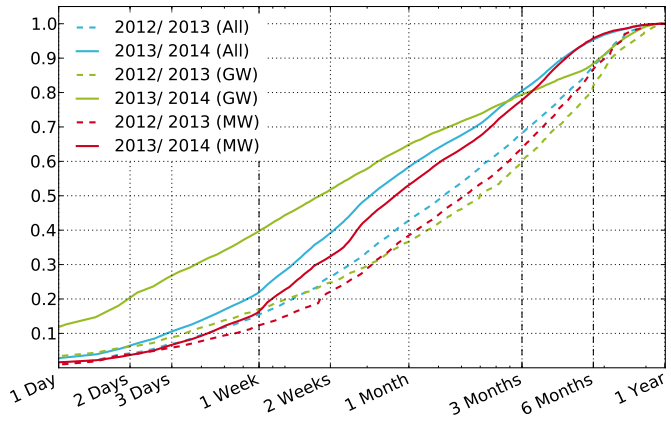

Fig. 4: CDF of time between APK creation and first submission to ANDRUBIS in the first (June 2012 - June 2013) and second year (June 2013 - June 2014) of operation.

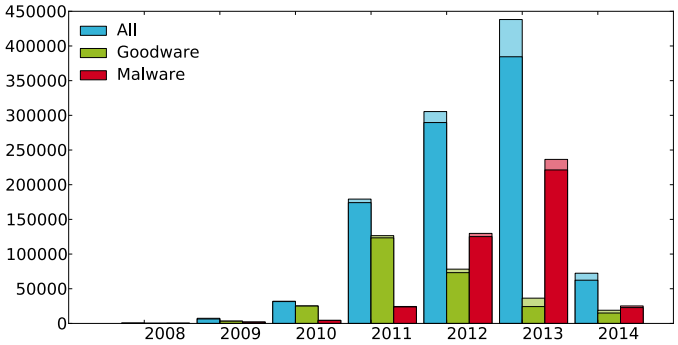

Fig. 5: Number of all, goodware and malware apps in our dataset per year. The share of successful analyses is highlighted in a darker shade.

Finally, for categorizing our dataset by release year, we also include the date ANDRUBIS first saw an app in the estimation and assign `min(apk_date,first_seen)` as the final app release date. This results in the dataset depicted in Figure 5, separated by app release year and category (*All*, *Malware*, *Goodware*). Android was released in September 2008, however, malware first surfaced in 2010 [39] and apps released before 2010 amount to less than 0.76% of all apps in our dataset, thus, we focus in our following evaluation on apps released between 2010 and 2014.

IV. ANDROID MALWARE LANDSCAPE

We already used the dataset described in the previous section in prior work for exploring WebView-related vulnerabilities [51]. Based on apps collected between July 2012 and March 2013 we determined that 30% of apps were vulnerable to web-based attacks by exposing native Java objects via JavaScript.

In the following, we give a summary of apps' static analysis features and behavior during dynamic analysis and identify trends for *All*, *Goodware* and *Malware* samples over the past four years from 2010 to 2014. As our observations show, dynamic analysis is increasingly able to capture behavior otherwise missed by static analysis. This is in part due to the increasing use of dynamic code loading amongst malicious and benign apps and their use of obfuscation techniques and/or DRM protection. Additionally, while 57.08% of malware samples employ reflection with no significant change over the years, use of reflection amongst all apps has increased significantly from 43.87% in 2010 to 78.00% in 2014, and even more in goodware (from 39.55% to 93.00%). Therefore, it is essential for large-scale evaluations to include dynamic analysis systems.

A. OBSERVATIONS FROM STATIC ANALYSIS

While dynamic analysis is gaining importance in forming a complete picture about an app's functionality, for some features evaluation of static features already provides valuable insights. In this section we take a look at permission requests and their usage according to static analysis, application names, developer certificates, resources sharing between apps, registered broadcast receivers, the use of third-party libraries and the exploitation of Master Key vulnerabilities.

1) Requested Permissions: Android apps can define and request arbitrary permissions: in fact, we observed almost 30,000 unique permissions being requested overall. Here, we focus on permissions defined and safeguarded by the Android OS. In addition to parsing all requested permissions from the manifest, we statically extract the usage of permissions from the app's source. While this approach ignores permissions that are requested, but only used in code dynamically loaded at runtime, we could use ANDRUBIS's method tracer (Section II-B3), to determine the permission usage during dynamic analysis in future experiments. For now the dynamic extraction of used permissions is in an experimental state and results are only available for a subset of samples.

We statically extract the usage of 143 permissions, covering the most interesting and commonly requested permissions as shown in Table III (in the Appendix). In line with previous findings on permission usage amongst malware, malicious samples generally request more permissions than goodware, but use less of them: malicious apps request 12.99 (11.57 when only looking at the subset of permissions we can statically extract) permissions on average, but use only 5.31 of them, goodware apps on the other hand request 5.85 (5.56) permissions on average and use 4.50 of them. One explanation for this behavior is that malware samples request more permissions during installation than needed so that they have the possibility to load other code parts that use these permissions later on. Permission requests by malware have also increased from an average of 11.46 (10.19) in 2010 to 15.33 (13.93) in 2014, with the average number of used permissions increasing only from 5.51 to 5.86. The number of requested permissions for goodware has increased from 3.74 (3.58) in 2010 to 9.38 (8.45) in 2014, while the number of used permissions also increased from 3.13 to 5.62. For individual samples, the permission usage ratio has declined for both goodware and malware, however, more significantly for goodware: samples in this category from 2014 only use 13.38% of requested permissions in their code – a possible side effect of the increased use of dynamic code loading (see Section IV-B4). Figure 6 illustrates this development.

Table III shows an overview of the most frequently requested permissions for malware and goodware. While the most commonly requested permissions for both malware and goodware are related to accessing the Internet, checking the network connectivity and reading device specific identifiers from the phone state, the majority of malware samples also requests SMS-related permissions. Furthermore, the possibility to manipulate shortcuts on the home screen can be used for phishing attacks and is frequently requested by malware as well. Another critical permission requested by malware is `SYSTEM_ALERT_WINDOW`, which allows an app to show windows on top of all other apps, overlapping them completely. It is used to display aggressive ads and by ransomware that draws a window over all other apps to keep the user from accessing any other phone functionality.

It is also important to note that not only individual but also combinations of permissions can be security-critical: while the `INSTALL_SHORTCUT` permission, which is requested by more than half of the malicious apps, is classified as dangerous, the same functionality can be achieved through the combination

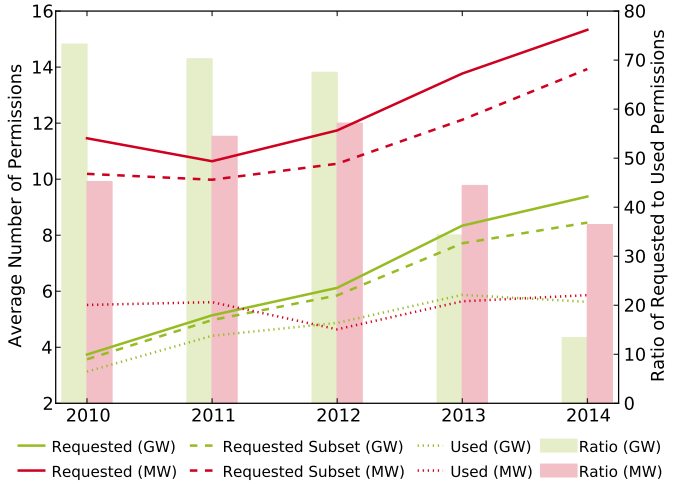

Fig. 6: Goodware (GW) and malware (MW) apps request an increasing number of permissions (overall as well as from the subset of permissions we statically extract), but permission usage stays constant – a side effect of the increasing use of dynamic code loading and obfuscation.

of the normal `READ_SETTINGS` and `WRITE_SETTINGS` permissions [67]. This is common practice amongst malware, with 10.88% of malware samples requesting both permissions, while only 0.20% of goodware samples do so.

During our evaluation of dynamic analysis features (see Section IV-B), we observed samples attempting to send SMS, connect to the Internet or accessing the SD card, without having the appropriate permissions – actions that will be prohibited by the Android OS. One explanation, besides a simple oversight, is developers mistyping the intended permission in some cases, for example as `andorid.permission.*`.

2) Application Names: The package name is the official identifier of an app, i.e., no two apps on a given device can share the same package name. Some markets, such as Google Play, also use it as a unique reference, but developers are not restricted from creating an app with an already existing package name. For malware authors reusing the package name of a legitimate app is also a way to masquerade as a benign app. Consequently, malware samples are far more likely to reuse package names than goodware samples: while 73.78% of goodware package names are unique, the same holds true for only 25.72% of malware's package names. Note, this number is likely to be slightly biased by submissions from AndRadar that explicitly locates apps in markets based on their package name to model and analyze how they spread [41].

A total of 8.50% of malware samples share their package name with legitimate apps from our goodware set – in total 4,059 distinct package names, half of which are currently available in the Google Play Store. Among the most frequently repackaged apps are Armor for Android Antivirus (`com.armorforandroid.security`, 387 samples), Steamy Window (`com.appspot.swisscodemonkeys.steam`, 93 samples), Opera (`com.opera.mini.android`, 68 samples), and Flappy Bird (`com.dotgears.flappybird`, 23 samples) – besides the paid Armor Antivirus all apps exceed 5 million downloads on the Google Play Store.

By far the most often shared package name, shared by 1,735 malicious apps with a single legitimate Google Play app, is `com.app.android`, however, more likely due to careless naming on the legitimate app's developers side. In general, authors of malicious apps tend to favor generic names and reuse them between samples, `com.software.app` and `com.software.application` being the most popular ones with 9,256 and 8,321 unique samples respectively. Starting in 2012, we observed malware authors adopting random looking package names, such as `ouepxayhr.efutel`, `ovbknnfm.xwscmnoi` and `rpyhwytfysl.uikbvktgwp`. F-Secure observed those package names being particularly popular amongst the *Android.Fakeinst* family [32]. However, contrary to the first impression, package names are not randomized on a per-app basis, as evidenced by up to 3,234 unique samples per name.

3) Certificates: Certificates are a corner-stone in Android security: each and every Android app has to be shipped with its developer's certificate and signed with his private key so that it can be installed. Android uses the certificate to enforce update integrity, i.e., it only allows updates signed with the same key, and it uses it to allow resource sharing and permission inheritance between apps from the same author [19].

Google does not impose any restrictions on the certificates used to sign Android apps and over 99% of all certificates are self-signed. We collected increasingly more apps signed by the same key, for goodware and malware alike. While, in 2010, 19.21% of all keys were used to sign more than one goodware app and 28.57% of the keys were used to sign more than one malicious one, this increased to 40% for both goodware and malware in 2014. This not only means that we are collecting more apps by the same developers, but also that blacklisting certificates used to sign malware is a viable option to keep malware from spreading. Especially widely used are four test keys distributed as part of the Android Open Source Project (AOSP): 8.92% of malicious samples are signed with one of these test key, however, the ratio significantly decreased from 65.29% of malicious apps in 2010 to 7.29% in 2014. Although those keys should not be used by legitimate apps, 2.26% of goodware apps are signed with a publicly available test key – making them vulnerable to attack: as we will show in the next section, if a user has such an app installed, malware signed with the same test key can potentially share permissions with the vulnerable app.

To our surprise we also found four samples, each labeled by more than 11 AV scanners as part of the *Android.Bgserv* malware family, that are signed with a valid Google certificate. These apps with the package name `com.android.vending.sectool.v1` are a malware removal tool by Google, mistakenly flagged by malware by numerous AV vendors [62].

4) Application Interdependencies: The Android system assigns, by default, a unique user ID (UID) to each app and runs it as that user in a separate process. Apps, however, can share their UID with other apps by specifying a `sharedUserId` in the manifest. This allows apps to share data, run in the same process, and even inherit each other's permissions [19], all under the prerequisite that apps are signed with the same key. Clearly, this feature also allows collusion amongst apps [48]: a malicious payload could be spread across multiple innocent looking apps. We saw this feature more commonly implemented in goodware than in malware: 1.14% of apps share their UID while only 0.29% of malicious apps do. This functionality becomes especially security critical when combined with an exploit for the powerful Master Key vulnerabilities (detailed in Section IV-A7). In theory, attackers could inject their code into apps not requesting any permissions at all but inheriting permissions from more privileged apps through a shared UID. Apps can even try to gain system privileges by exploiting an app signed with a platform certificate and sharing the UID with `android.uid.system`. Furthermore, with numerous apps being signed with the test key from the AOSP, crafting a malicious app inheriting the permissions from other apps is possible even without having to utilize an exploit to circumvent the app signing process. In fact, 6.79% of benign and 17.57% of malicious samples that share a

UID are signed with a public test key. This becomes especially critical when the Android OS itself is signed with a public key: according to DroidRay [69], a recent security evaluation of custom Android firmware, out of 250 firmware images, 56.80% were signed with a key pair from the AOSP. In our dataset, we identified 84 apps (4 of which were not detected by any AV scanners, the remainder was labeled as *Android.Fjcon*) that were capable of gaining system privileges through UID sharing with `android.uid.system`. All samples were signed with the same AOSP key pair used to generate the system signature for 134 (53.60%) of the firmware images evaluated by DroidRay.

5) Broadcast Receivers: Apps can register broadcast receivers for arbitrary custom events, however, we focus our analysis on broadcast receivers listening for system events. Broadcast receivers are by far more widely used in malicious apps than in benign apps: 82.18% of all malware samples register one or more broadcast receivers, while only 41.86% of goodware sample use this feature. Table IV (in the Appendix) lists the most frequently registered broadcast receivers for both categories. Goodware mainly watches for notifications to update their widgets, install referrers from the market and a user being present, probably to suspend idle mode quickly whenever a user unlocks the phone so that new data can be fetched and the app's status can be updated. Malware, on the other hand, often registers itself as a service, which is running in the background, and does not care for user input. More than half of all samples listen for the `BOOT_COMPLETED` event, which triggers as soon as the phone has booted the Android OS, and for the event that is published upon receipt of incoming messages, both text-based (`SMS_RECEIVED`) and data-based (`DATA_SMS_RECEIVED`). However, we only saw listeners for data-based SMS in 2012 and 2013 with 24.43% and 10.41% of malware samples listening for this event. We also see growing interest of malicious apps in the `CONNECTIVITY_CHANGE` and `AIRPLANE_MODE` receivers since 2012 with a peak of 17.93% and 14.99% in 2013 respectively. Furthermore, malicious apps started using Device Administrator Privileges, which makes them harder to uninstall. The latter are used by 11.94% of malware samples in 2014, which register for the `DEVICE_ADMIN_ENABLED` event.

6) Third-Party Libraries: We checked all apps in our dataset against a list of the 53 most popular *advertisement (ad) libraries* according to AppBrain [4]. Fewer malicious (17.45%) than benign (44.32%) apps come bundled with ad libraries, presumably in part because we excluded samples labeled as adware from our malware dataset. However, with ad fraud being one way to monetize malicious app installs, malicious samples include more ad libraries simultaneously: we saw a maximum of 13 ad libraries in a single goodware app and 14 ad libraries in a single malware app with 1.56 and 2.05 libraries on average respectively. Table V (in the Appendix) lists the most popular ad libraries for goodware and malware. Besides Google's AdMob being the most popular across both categories, albeit with diverging percentages of over 35% in goodware to only 5.7% in malware, there is little overlap. With mobile malware being particular prevalent in China [45], malicious apps mainly include Chinese ad networks. Malware also favors aggressive ad libraries, such as AirPush and Adwo, often classified by AV scanners as adware and banned from Google's Play Store [57] by policy because they push advertisements to the notification bar.

Social networking libraries are used in 11.14% of goodware apps (8.86% Facebook, 3.38% Twitter, 1.89% Google+), while the number of malicious apps including such libraries is a negligible 0.78% (0.66% Facebook, 0.13% Twitter, 0.09% Google+), possibly indicating those libraries are shipped with the original app that was targeted by repackaging to include malicious code.

The same as for social networking libraries holds true for the use of *billing libraries*: 3.58% of goodware and only 0.53% of malware apps make use of billing services (3.08% Google Billing, 0.57% Paypal, 0.17% Amazon Purchasing and 0.03% Authorize.net in goodware; 0.35% Google Billing, 0.19% Paypal and 0.05% Amazon Purchasing in malware). Billing services for in-app purchases are harder to monetize for malware since payment providers usually have refund policies.

7) Master Key Vulnerabilities: In 2013 researchers reported the Master Key vulnerability [35] in the Android app signing process, which allows an app's content, including its code, to be modified without breaking the signature – essentially allowing attackers to inject malicious code into any legitimate applications without repackaging them. This vulnerability stems from discrepancies between the handling of the ZIP file format between the signature verification and installation process in Android. Shortly after the original Master Key vulnerability was published, two similar vulnerabilities were discovered [27,28].

Bug 8219321, the original Master Key vulnerability, is based on the fact that the ZIP file format allows two files with the same file name, thus allowing attackers to hide an additional `classes.dex` file that is deployed by the installer instead of the original one that is checked by the signature verifier. We saw this vulnerability being exploited in 1,152 samples (0.11%), all from 2013 and 2014, and only in malware, possibly due to AV scanners automatically flagging apps as there is no legitimate reason for this behavior.

Bug 9695860 stems from a signed unsigned integer mismatch in the length of the extra field of the ZIP file header. In addition to allowing attackers to inject an app with a malicious `classes.dex`, the exploitation of this vulnerability also breaks analysis tools utilizing the unpatched version of the Python `zipfile` [12], such as Androguard in the default configuration. We saw 4,553 samples (0.44%) triggering the Python bug. However, we only found two samples with an extra field length triggering an integer overflow and thus the vulnerability, one of them being a proof of concept [9].

Bug 9950697 lies within the redundant storage of the length of the file name in both the central directory of the ZIP file as well as the local file header. Again this vulnerability allows attackers to specify a file name large enough for the installer to skip the original `classes.dex` file and install the injected one. However, we only observed this bug being exploited in 447 (0.05%) of all samples (starting already in 2011 with the majority of samples being from 2013), with 92 malware and 26 goodware samples respectively.

B. Observations from Dynamic Analysis

In contrast to static analysis, dynamic analysis lets us monitor an app's behavior during runtime – including behavior caused by dynamically loaded code. In addition, the obtained information is more comprehensive and includes full paths of file system accesses, called phone numbers, recipients and contents of SMS, leaks of sensitive information, as well as usage of cryptographic algorithms and a full profile of the app's network behavior.

1) File Activity: Apps can both read and write the internal storage as well as external storage from SD cards. Overall 72.49% of goodware and 95.99% of malware read files, and 83.11% of goodware and 94.70% of malware write to the file system during dynamic analysis in ANDRUBIS. When distinguishing file system access to the primary storage and access to the secondary storage, i.e., the SD card, it becomes apparent that SD card access is far more prevalent amongst malware: 22.02% of malicious apps read and 27.82% write files to the SD card, while only 2.91% of benign apps read and 6.69%

write to external storage. Starting in Android 3.2 (Honeycomb) Google restricted third-party apps from accessing the SD card by limiting the `WRITE_EXTERNAL_STORAGE` to the primary storage and requiring the `WRITE_MEDIA_STORAGE`, which is only granted to system apps, for write access to the SD card. However, this change was largely ignored by OEM and custom firmware developers [60]. In our dataset 93.08% of goodware and 97.69% of malware apps that write to the SD card request the first permission, while only 0.59% of goodware and 0.08% request both. Static analysis completely failed to determine the usage of the `WRITE_EXTERNAL_STORAGE` permission and thus the write access to the SD card in any app. Furthermore, despite Google's policy to restrict write access to SD cards, this behavior has been steadily increasing in goodware apps from 2.89% in 2010 to 16.64% in 2014. Writing to SD storage in malware has been a constant behavior in around 30% of malware. This is likely to increase even more in the future with new possibilities for monetization being explored: recently the Cryptolocker family started encrypting files stored on the SD card and demanding ransom for the decryption key [42].

2) Phone Activity: Concerning mobile-specific behavior, only very few applications initiated phone calls during dynamic analysis: 0.24% of goodware apps and only 0.04% of malware apps. For both malware and goodware, 98% of those apps requested the corresponding `CALL_PHONE` permission, however, static analysis failed to determine any usage of this permission from the apps' source.

While the percentage of apps sending SMS in the goodware dataset is as low as the percentage of apps initiating phone calls (only 0.26%), we observed 15.00% of malicious apps sending text messages. This comes as no surprise: sending SMS to premium numbers is a popular monetization vector of mobile malware [59]. Again, 98.57% (goodware) and 99.15% (malware) of those apps requested the necessary `SEND_SMS` permission, while static analysis revealed that 85.37% (goodware) and 81.79% (malware) of those apps actually use this permission in their source code – again showing the value and importance of dynamic analysis to uncover behavior from hidden or obfuscated function calls. Phone numbers tend to be shorter for malware, also indicating the use of premium numbers – goodware apps send SMS to 410 unique numbers with an average length of 7.18 digits, while the 1,943 distinct numbers malware sends SMS to is only 4.26 digits on average. Furthermore, we observed malware samples sending up to 120 SMS to premium numbers during four minutes of dynamic analysis.

3) Data Leakage: Data leakage is significantly more prevalent in malware than in goodware: overall, 14.28% of goodware apps leak information over the network, while 42.53% of malicious apps do so. When looking at the dataset as a whole, data leakage to the network overall occurred in 38.79% of all apps and significantly increased from 13.45% in 2010 to 49.78% in 2014. Both goodware and malware leak device specific identifiers, such as the International Mobile Station Equipment Identity (IMEI), International Mobile Subscriber Identity (IMSI), Integrated Circuit Card Identifier (ICCID) and the phone number. Goodware mainly leaks the IMEI, while a quarter of malware leaks the IMSI and almost 14% of malware leaks the user's phone number. Leakage of names and phone numbers from the user's address book is also more common amongst malware than it is amongst benign apps. Instead, goodware mainly leaks the location, an information source less commonly leaked by malware samples. Few samples in general leak information on installed packages, the contents of SMS, the call log, and browser bookmarks. Table VI (in the Appendix) summarizes the information sources most commonly leaked to the network by goodware and malware.

Data leakage via SMS occurred only in 0.04% of goodware and in 0.72% of malware samples. This number, however, has increased over the past years, with 1.87% of malware samples leaking identifiers such as the IMSI, IMEI, ICCID and the phone number, but also forwarding incoming SMS and the call log via SMS in 2014.

4) Dynamically Loaded Code: Android apps can load code at runtime to dynamically extend their functionality. However, this technique comes with severe security implications. While dynamic code loading is popular for legitimate reasons, such as loading external add-on code, shared library code from frameworks, or dynamically updating code during beta and/or A/B testing, it is especially interesting for malware. Since apps are typically inspected only once, either by an app market or by an AV scanner at installation time, malicious apps can download and load their malicious payload later at runtime to evade detection. Furthermore, the unsafe use of code loading techniques can also make legitimate apps vulnerable to code injection techniques, as shown by Poeplau et al. [54].

DEX Classes. One possibility to dynamically extend an app's functionality is to load modules at the Dalvik VM level through the DEX class loader. We observed this behavior for 2.97% of goodware and for 4.46% of malware apps, with a significant increase over the past two years. Static analysis successfully identifies the invocation of the `DexClassLoader` in 98.88% of goodware and 97.20% of malware respectively. On average, goodware loads 1.28 and malware loads 1.59 DEX classes. The maximum of different classes loaded is 37 for the Metasploit payload, 25 classes for samples from the *Android.SmsSpy* family and 9 classes for goodware in general.

Native Libraries. Overall, both goodware and malware apps load native libraries in equal proportions: we observed 8.60% and 8.50% of all benign and malicious apps loading native code during dynamic analysis, with a clear upward trend especially amongst goodware. The sources for the loaded native code and their impact differ: at a finer granularity, we distinguish between the number of system native libraries loaded and custom, non-system, libraries loaded. Custom libraries are far more dangerous than those provided by the Android system itself. The reason for system library usage is simple: games and graphically demanding apps make use of hardware-accelerated technologies found in modern graphics cards, like OpenGL or video decoding, for both performance reasons and increased battery life. Custom libraries, however, tend to be used by malware for a number of nefarious purposes, including the elevation of privileges through root exploits.

Goodware apps load 52.47% and 52.26% code from the system and the data directory respectively, contrary, only 19.46% of malware samples load native system libraries, while 84.19% load their own bundled native code or fetch it from remote servers. While for malware the percentage of system libraries loaded decreased from 2010 to 2014 by 13 percentage points and the usage of custom libraries increased by 20 percentage points, this trend is more severe for goodware: in 2010, 74.11% of goodware apps loaded native code from the system and only 29.57% loaded custom code; in 2014, 30.95% of apps loaded code from the system and 73.37% loaded it from the data directory.

Static analysis was far less successful in identifying native code loading compared to DEX class loading and only identified the `loadLibary()` call in 54.40% of goodware and 83.25% of malware. These numbers correspond to the number of apps shipping with unencrypted ELF libraries that can be identified based on their file signature: 54.29% in the case of goodware and 85.23% in the case of malware.

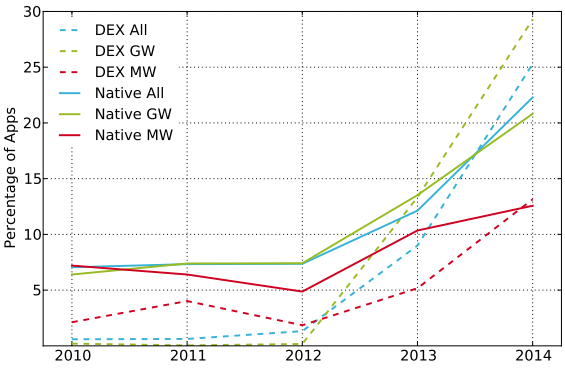

Fig. 7: Increasing use of DEX and native code loading overall, and in goodware (GW) and malware (MW).

Dynamic code loading significantly increased during our observation period, especially for goodware over the past two years, as shown in Figure 7: in 2014, 29.29% of benign apps loaded Dalvik and 20.82% native code, while 13.15% of malicious apps loaded Dalvik and 12.57% native code. Furthermore, loading native libraries and DEX classes is not an either-or decision: 1.25% of all malware (5.43% in just 2014) and 0.45% of all goodware (4.92% in 2014) combine those techniques to load both native and Dalvik code.

5) Cryptographic API Usage: Another interesting case study is the use of cryptographic protocols. During dynamic analysis, we observed the usage of the Java crypto API in 5.63% of malicious apps, in contrast to only 1.10% of goodware apps. Interestingly, for those apps we could statically determine the use of cryptography in 99.21% of cases for goodware, but for only 43.24% of malware – either due to this part of the code being obfuscated and loaded dynamically at runtime. Overall, static analysis revealed the use of `javax/crypto/*` in 44.83% of goodware apps, increasing from 11.12% in 2010 to 79.18% in 2014. For malware, we did not see such a development with the Java crypto API being used by only 29.84% overall, likely due to malware shipping their own implementations in order to evade detection.

The most popular algorithms observed during dynamic analysis of goodware are AES (66.75%), PBEwithMD5andDES (15.03%), DES (11.98%), and RSA (5.08%). Malware, on the other hand, mainly used AES (74.82%), Blowfish (14.31%), DES (8.78%), and RSA (1.20%). We also observed a trend toward stronger cryptographic algorithms in malware: while DES was the predominantly used algorithm amongst malware in 2010 (98.44%), its usage declined significantly to 1.53% in 2013. Instead, in 2012 malware authors started adopting the stronger Blowfish algorithm, which is now being used by 31.58% of all malware apps from 2013, while we have not seen a single goodware app using Blowfish.

6) Network Activity: We observed network traffic in goodware and malware apps alike – 71.11% of goodware and 80.36% of malware, with almost 99% of those samples requesting but only 70.97% of benign and 61.43% of malicious samples using the `INTERNET` permission according to static analysis. This numbers decreased for malware in 2014 to only 94.40% requesting and 58.84% using the permission, indicating malware circumventing the permission system by performing network activity through other apps installed on the device, such as the browser, for example.

Almost all apps that use the Internet query domain names: 99.91% of malware and 97.34% of goodware perform DNS queries, but while one third (32.33%) of the queries by malicious samples fail and result in an invalid (NX) domain, only 10% of queries from goodware samples do.

UDP traffic is almost limited to DNS, with only a few samples using NTP. However, 55.33% of malware and 23.62% of goodware also establish TCP connections. This number increased for malware from 27.69% in 2010 to 58.65% in 2013, and decreased to 45.84% in 2014; for goodware it monotonically increased from 12.81% in 2010 to 43.50% in 2014. The most commonly observed network activity for malware occurred on port 443 (HTTPS, 44.09% of samples), port 80 (HTTP, 15.52%), and port 5224 (XMPP/Google Talk), 8245 (DynDNS), and 9001 (Tor) with less than 0.2% of samples each. For goodware we observed port 443 (HTTPS, 15.58%), port 80 (HTTP, 7.31%), and port 1130 (CASP, 0.46%).

Other protocols were hardly ever used: we only observed 77 apps in our whole dataset establishing FTP connections and 14 samples using IRC. We saw, however, 352 samples from 2013 and 2014 establishing SMTP connections and sending emails. The majority of those samples are classified by AV scanners as malware and they leak sensitive information such as the contents of the address book and incoming SMS via email to addresses from Chinese freemail providers, such as NetEase (`163.com`, `126.com`) and Tencent (`qq.com`).

7) Cross-Platform Malware: In 2013 Android malware started to download a malicious Windows payload (*Backdoor.MSIL.Ssucl*) and saving it together with an `autorun.inf` file in the root directory of the phone's SD card, hoping it would be automatically executed on Windows computers once the phone was connected to the PC via USB [26]. We only saw this behavior in 11 apps overall, nine of which were different versions of the goodware samples iSyncr and RealPlayer that placed their Windows installer together with `autorun.inf` on the SD card. Only 19 goodware samples embedded executables. The only malicious samples we saw exhibiting this behavior were from the *Android.UsbCleaver* [31] family. Overall, we detected 447 malware samples with a total of 27 different embedded executables that are flagged by at least one AV scanner.

There have been reports of Windows malware attempting to infect Android devices, and even installing the Android Debug Bridge (ADB) to do so [43]. We have only seen 119 Windows samples in ANUBIS attempting to drop APK files, 16 of which also tried to access the ADB (currently not installed in our Windows environment). The majority of those files, however, failed to download completely or seem to belong to rooting utilities. VirusTotal has labels for 56 out of the 99 dropped APKs, with 33 not being detected by any AV scanners, 20 detected, as root exploits and the remaining three belonging to *Android.AndroRat* and *Android.FakeAngry*.

V. RELATED WORK

For Windows malware, Bayer et al. [20] performed a similar analysis to ours on a dataset of 900,000 Windows samples ANUBIS received within its first two years of operation. Here, however, we focus on related work on the Android malware landscape.

Android security and the detection and characterization of Android malware in particular has been an extremely active field of research in the past years. Felt et al. [33] analyzed a total of 46 iOS, Symbian and Android malware samples collected between 2009 and 2011 to provide one of the first surveys on mobile malware and their author's incentives. The Android Malware Genome Project [72] was a further attempt to systematize Android malware behavior and provided a publicly available dataset used in many following evaluations. The dataset, however, is now showing its age: the samples being collected between 2010 and 2011 behave significantly different than apps from 2012 to 2014, as we have shown in our evaluation (Section IV). Another available malware dataset is the one used by Drebin [17] for classifying Android malware. This dataset also includes the

Genome Project and the most recent samples were collected in 2012. Further studies on malware behavior mainly focused on the practice of repackaging and the pervasiveness of repackaged apps in alternative app stores [70,71].

TaintDroid [30] was the first work to propose taint tracking for monitoring data flow dependencies and data leakage in Android apps and is now at the core of many sandboxes, such as ours, to track data leaks. DroidScope [66] is a dynamic analysis system solely based on VMI. While this approach has advantages, such as whole-system taint analysis, the delicate reconstruction of Java objects and the like from raw memory regions requires substantial adaption effort with each Android OS update.

SmartDroid [68] and AppsPlayground [55] aim at improving the stimulation of apps during dynamic analysis. They try to drive the app along paths that are likely to reveal interesting behavior through targeted stimulation of UI elements. Their approaches can be seen as intelligent enhancements of the Application Exerciser Monkey and our custom stimulation of activity screens. They are largely orthogonal to our work, which focuses on stimulating broadcast receivers, services and common events, instead of UI elements.

Concerning systems for the large-scale dynamic analysis of Android applications, Bläsing et al. [24] proposed AASandbox, the first dynamic analysis platform for Android based on system call monitoring. ANANAS [29], on the other hand, is a dynamic analysis framework focusing on extensibility through modules. DroidRanger [73] pre-filters applications based on a manually created permission-fingerprint before subjecting them to dynamic analysis. In contrast to this approach, we analyzed every app, yielding full behavioral profiles to base our evaluation on. Furthermore, DroidRanger performs monitoring through a kernel module instead of VMI and focuses only on system calls used by existing root exploits. Finally, DroidRanger does not employ stimulation techniques. None of the above tools are publicly available.

Dynamic analysis systems that are publicly available are CopperDroid [7,56], Tracedroid [14,63], SandDroid [13], and Mobile-Sandbox [11,58]. CopperDroid performs out-of-the-box system call monitoring through VMI and reconstructs Dalvik behavior by monitoring Binder communication. Tracedroid generates complete method traces by extending the Dalvik VM profiler and was subsequently integrated into ANDRUBIS, but it is also available as a standalone service. SandDroid performs monitoring of the Dalvik VM, but does not allow any network connections to the outside and therefore misses behavior in apps checking for Internet connectivity [64]. Mobile-Sandbox monitors native code through `ltrace` in addition to instrumenting the Dalvik VM. However, both SandDroid and Mobile-Sandbox seem to be unable to cope with their submission load: SandDroid has only analyzed around 25,000 samples to date and samples we submitted have been stuck in the input queue for almost nine months, while Mobile-Sandbox reports a backlog of over 300,000 samples with no samples seemingly being analyzed. We emphasize that, to the best of our knowledge, ANDRUBIS is the only dynamic analysis sandbox operating on a large-scale, providing a thorough analysis on both Dalvik and system level, and typically returning a report in ten minutes or less.

VI. LIMITATIONS

One limitation of any dynamic analysis approach is evasion. As long as a sandbox is not capable of perfectly emulating a system, a possibility to detect it exists. Petsas et al. [53] and Vidas et al. [64] recently explored the possibility to fingerprint Android sandboxes, and found that all, including ours, are susceptible to evasion. Sandbox detection techniques range from static characteristics of the specific Android OS installation to information from sensors, to the detection of the underlying virtualization technology. One proof of concept [49] is able to detect any QEMU-based environment based on binary translation: QEMU (and other emulators) usually take a basic block, translate it, and execute the whole resulting basic block on the host machine. Unfortunately, this property allows for an easy detection of emulated code, since the basic block cannot be interrupted by the guest operating system's scheduler. As a countermeasure, we enabled QEMU single-step mode, which makes ANDRUBIS undetectable by this evasion technique. However, this mode introduces an analysis overhead of 29% compared to 7% with Dalvik monitoring and 18% with QEMU VMI [65]. Generally, dealing with analysis evasion is a never-ending arms race between security researchers and malware authors.

A further limitation of dynamic analysis is code coverage. While we try to increase behavior seen during analysis through various stimulation techniques, a more intelligent user interface stimulation than the random input stream by the Android Exerciser Monkey could provide more complex and user-like input and, in turn, trigger much more behavior from the apps under analysis.

Currently public submissions to ANDRUBIS are limited to a file size of 8MB. This limit, however, is simply a limitation of our web interface and not a fundamental limitation of our analysis. We are currently evaluating to increase this limit, while keeping storage requirements at an acceptable level without having to discard apps after analysis.

Finally, a limitation of any analysis system allowing submissions from anonymous sources is the lack of metadata and ground truth. We have no indication when and where samples were found or how widespread they are in the wild. We tried to mitigate this in part by collecting metadata from markets with AndRadar [41]. Lacking ground truth, we have to rely on AV signatures to classify our dataset in goodware and malware, but we are experimenting with machine-learning approaches to automatically classify samples with higher accuracy than related work.

VII. CONCLUSION

In this paper we presented ANDRUBIS, a fully automated large-scale analysis system for Android apps that combines static analysis with dynamic analysis on both Dalvik VM and system level. ANDRUBIS accepts public submissions through a web interface and a mobile app and is currently capable of analyzing around 3,500 new samples per day. With ANDRUBIS, we provide malware analysts with the means to thoroughly analyze Android apps. Furthermore, we provide researchers with a solid platform to build post-processing methods upon based on an app's static features and dynamic behavior. For example, leveraging machine-learning approaches one can use our analysis results to tackle the problem of judging whether a previously unseen app is malware significantly more accurate than prior work.

ANDRUBIS has analyzed over 1,000,000 Android apps to date. On an evaluation of this dataset spanning samples from four years, we showed changes in the malware threat landscape and trends amongst goodware developers. Dynamic code loading, previously used as an indicator for malicious behavior, is especially gaining popularity amongst goodware, and, in turn, loses significant information value when distinguishing between benign and malicious apps. Due to this development, static analysis tools alone are increasingly unable to completely capture an app's behavior, making dynamic analysis indispensable for a comprehensive analysis for a large number of apps.

In future work, we plan to explore the network behavior of Android malware further to identify C&C communication patterns and shared infrastructures with Windows malware. Furthermore, we are exploring the option of releasing a comprehensive malware dataset, once we sorted out legal and confidentiality issues.

ACKNOWLEDGMENTS

We would like to thank VirusTotal for the service they provided for our evaluation. The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement n. 257007 (SysSec) and from the FFG – Austrian Research Promotion under grant COMET K1.

This work also has been carried out within the scope of u'smile, the Josef Ressel Center for User-Friendly Secure Mobile Environments. We gratefully acknowledge funding and support by the Christian Doppler Gesellschaft, A1 Telekom Austria AG, Drei-Banken-EDV GmbH, LG Nexera Business Solutions AG, and NXP Semiconductors Austria GmbH.

REFERENCES

[1] "Android Version history by API level," http://en.wikipedia.org/wiki/Android_version_history#Version_history_by_API_level.

[2] "Andrubis Submission App," https://play.google.com/store/apps/details?id=org.iseclab.andrubis.

[3] "Anubis," http://anubis.iseclab.org.

[4] "AppBrain Stats: Android Ad networks," http://www.appbrain.com/stats/libraries/ad.

[5] "AppBrain Stats: Number of Android applications," http://www.appbrain.com/stats/number-of-android-apps.

[6] "Contagio," http://contagiominidump.blogspot.com.

[7] "CopperDroid," http://copperdroid.isg.rhul.ac.uk.

[8] "ForeSafe Mobile Security," http://www.foresafe.com.

[9] "Fuzion24/Zip File Arbitrage: Exploit for Android Zip bugs: 8219321, 9695860, and 9950697," https://github.com/Fuzion24/AndroidZipArbitrage.

[10] "Joe Sandbox Mobile," http://www.joesecurity.org/joe-sandbox-mobile.

[11] "Mobile Sandbox," http://mobilesandbox.org.

[12] "Python Bug Tracker: Issue 14315," http://bugs.python.org/issue14315.

[13] "SandDroid," http://sanddroid.xjtu.edu.cn.

[14] "Tracedroid," http://tracedroid.few.vu.nl.

[15] "VirusTotal," http://www.virustotal.com.

[16] "VisualThreat," http://www.visualthreat.com.

[17] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket," in *Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS)*, 2014.

[18] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android Permission Specification," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[19] D. Barrera, J. Clark, D. McCarney, and P. C. van Oorschot, "Understanding and Improving App Installation Security Mechanisms Through Empirical Analysis of Android," in *Proceedings of the 2nd ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2012.

[20] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel, "A View on Current Malware Behaviors," in *Proceedings of the 2Nd USENIX Conference on Large-scale Exploits and Emergent Threats (LEET)*, 2009.

[21] U. Bayer, C. Kruegel, and E. Kirda, "TTAnalyze: A Tool for Analyzing Malware," in *Proceedings of the 15th European Institute for Computer Antivirus Research (EICAR) Annual Conference*, 2006.

[22] U. Bayer, P. Milani Comparetti, C. Hlauscheck, C. Kruegel, and E. Kirda, "Scalable, Behavior-Based Malware Clustering," in *Proceedings of the 16th Annual Network & Distributed System Security Symposium (NDSS)*, 2009.

[23] M. Bierma, E. Gustafson, J. Erickson, D. Fritz, and Y. R. Choe, "Andlantis: Large-scale Android Dynamic Analysis," in *Proceedings of the 3rd IEEE Mobile Security Technologies Workshop (MoST)*, 2014.

[24] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. Camtepe, and S. Albayrak, "An Android Application Sandbox System for Suspicious Software Detection," in *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE)*, 2010.

[25] L. Cavallaro, P. Saxena, and R. Sekar, "Anti-Taint-Analysis: Practical Evasion Techniques Against Information Flow Based Malware Defense," Secure Systems Lab at Stony Brook University, Tech. Rep., 2007.

[26] V. Chebyshev, "Mobile attacks!" http://www.securelist.com/en/blog/805/Mobile_attacks, February 2013.

[27] P. Ducklin, "Anatomy of a file format problem - yet another code verification bypass in Android," http://nakedsecurity.sophos.com/2013/11/06/anatomy-of-a-file-format-problem-yet-another-code-verification-bypass-in-android, November 2013.

[28] ——, "Anatomy of another Android hole - Chinese researchers claim new code verification bypass," http://nakedsecurity.sophos.com/2013/07/17/anatomy-of-another-android-hole-chinese-researchers-claim-new-code-verification-bypass, July 2013.

[29] T. Eder, M. Rodler, D. Vymazal, and M. Zeilinger, "ANANAS - A Framework For Analyzing Android Applications," in *Proceedings on the 1st International Workshop on Emerging Cyberthreats and Countermeasures (ECTCM)*, 2013.

[30] 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 *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[31] F-Secure, "Android Hack-Tool Steals PC Info," http://www.f-secure.com/weblog/archives/00002573.html, July 2013.

[32] ——, "Threat Report H2 2013," http://www.f-secure.com/static/doc/labs_global/Research/Threat_Report_H2_2013.pdf, March 2014.

[33] 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.

[34] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[35] J. Forristal, "Android: One Root to Own Them All," in *Black Hat USA*, 2013.

[36] J. Goebel, T. Holz, and C. Willems, "Measurement and Analysis of Autonomous Spreading Malware in a University Environment," in *Proceedings of the 4th International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA)*, 2007.

[37] Google, "Introducing ART," https://source.android.com/devices/tech/dalvik/art.html, 2014.

[38] IDC, "Android and iOS Continue to Dominate the Worldwide Smartphone Market with Android Shipments Just Shy of 800 Million in 2013," http://www.idc.com/getdoc.jsp?containerId=prUS24676414, February 2014.

[39] B. Irinco, "First Android Trojan in the Wild," http://blog.trendmicro.com/trendlabs-security-intelligence/first-android-trojan-in-the-wild, August 2010.

[40] C. Lever, M. Antonakakis, B. Reaves, P. Traynor, and W. Lee, "The Core of the Matter: Analyzing Malicious Traffic in Cellular Carriers," in *Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS)*, 2013.

[41] M. Lindorfer, S. Volanis, A. Sisto, M. Neugschwandtner, E. Athanasopoulos, F. Maggi, C. Platzer, S. Zanero, and S. Ioannidis, "AndRadar: fast discovery of android applications in alternative markets," in *Proceedings of the 11th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2014.

[42] R. Lipovsky, "ESET Analyzes First Android File-Encrypting, TOR-enabled Ransomware," http://www.welivesecurity.com/2014/06/04/simplocker, June 2014.

[43] F. Liu, "Windows Malware Attempts to Infect Android Devices," http://www.symantec.com/connect/blogs/windows-malware-attempts-infect-android-devices, January 2014.

[44] H. Lockheimer, "Android and Security," http://googlemobile.blogspot.com/2012/02/android-and-security.html, February 2012.

[45] Lookout Mobile Security, "State of Mobile Security 2012," https://www.lookout.com/_downloads/lookout-state-of-mobile-security-2012.pdf, 2012.

[46] A. Ludwig, E. Davis, and J. Larimer, "Android - Practical Security From the Ground Up," in *Virus Bulletin Conference*, 2013.

[47] F. Maggi, A. Valdi, and S. Zanero, "AndroTotal: A Flexible, Scalable Toolbox and Service for Testing Mobile Malware Detectors," in *Proceedings of the 3rd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2013.

[48] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun, "Analysis of the Communication Between Colluding Applications on Modern Smartphones," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, 2012.

[49] F. Matenaar and P. Schulz, "Detecting Android Sandboxes," http://www.dexlabs.org/blog/btdetect, August 2012.

[50] McAfee Labs, "McAfee Threats Report: Second Quarter 2013," http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q2-2013.pdf, August 2013.

[51] M. Neugschwandtner, M. Lindorfer, and C. Platzer, "A View To A Kill: WebView Exploitation," in *Proceedings of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2013.

[52] S. Neuner, V. van der Veen, M. Lindorfer, M. Huber, G. Merzdovnik, M. Mulazzani, and E. Weippl, "Enter Sandbox: Android Sandbox Comparison," in *Proceedings of the 3rd IEEE Mobile Security Technologies Workshop (MoST)*, 2014.

[53] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware," in *Proceedings of the Seventh European Workshop on System Security (EuroSec)*, 2014.

[54] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications," in *Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS)*, 2014.

[55] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic Security Analysis of Smartphone Applications," in *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[56] A. Reina, A. Fattori, and L. Cavallaro, "A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors," in *Proceedings of the 6th European Workshop on System Security (EuroSec)*, 2013.

[57] D. Ruddock, "Google Pushes Major Update To Play Developer Content Policy, Kills Notification Bar Ads For Real This Time, And A Lot More," http://www.androidpolice.com/2013/08/23/teardown-google-pushes-major-update-to-play-developer-content-policy-kills-notification-bar-ads-for-real-this-time-and-a-lot-more/, September 2013.

[58] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Mobile-sandbox: Having a Deeper Look into Android Applications," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC)*, 2013.

[59] V. Svajcer, "Sophos Mobile Security Threat Report," http://www.sophos.com/en-us/medialibrary/PDFs/other/sophos-mobile-security-threat-report.ashx, 2014.

[60] C. Toombs, "External Blues: Google Has Brought Big Changes To SD Cards In KitKat, And Even Samsung Is Implementing Them," http://www.androidpolice.com/2014/02/17/external-blues-google-has-brought-big-changes-to-sd-cards-in-kitkat-and-even-samsung-may-be-implementing-them, February 2014.

[61] ——, "Updates To AOSP Confirm Dalvik Runtime Will Be Removed From Android, ART Officially Takes Its Place," http://www.androidpolice.com/2014/06/19/updates-aosp-confirm-dalvik-runtime-will-removed-android-art-officially-takes-place, June 2014.

[62] H. T. T. Truong, E. Lagerspetz, P. Nurmi, A. J. Oliner, S. Tarkoma, N. Asokan, and S. Bhattacharya, "The Company You Keep: Mobile Malware Infection Rates and Inexpensive Risk Indicators," in *Proceedings of the 23rd International Conference on World Wide Web (WWW)*, 2014.

[63] V. van der Veen, "Dynamic Analysis of Android Malware," *Internet & Web Technology Master thesis, VU University Amsterdam*, 2013.

[64] T. Vidas and N. Christin, "Evading Android Runtime Analysis via Sandbox Detection," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.

[65] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer, "Andrubis: Android Malware Under The Magnifying Glass," Vienna University of Technology, Tech. Rep. TR-ISECLAB-0414-001, 2014.

[66] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21st USENIX Security Symposium*, 2012.

[67] Y. Zhang, H. Xue, and T. Wei, "Occupy Your Icons Silently on Android," http://www.fireeye.com/blog/technical/2014/04/occupy_your_icons_silently_on_android.html, April 2014.

[68] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, and W. Zou, "SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications," in *Proceedings of the 2nd ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2012.

[69] M. Zheng, M. Sun, and J. C. Lui, "DroidRay: A Security Evaluation System for Customized Android Firmwares," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.

[70] W. Zhou, Y. Zhou, M. 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.

[71] 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.

[72] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[73] 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 Network & Distributed System Security Symposium (NDSS)*, 2012.

APPENDIX

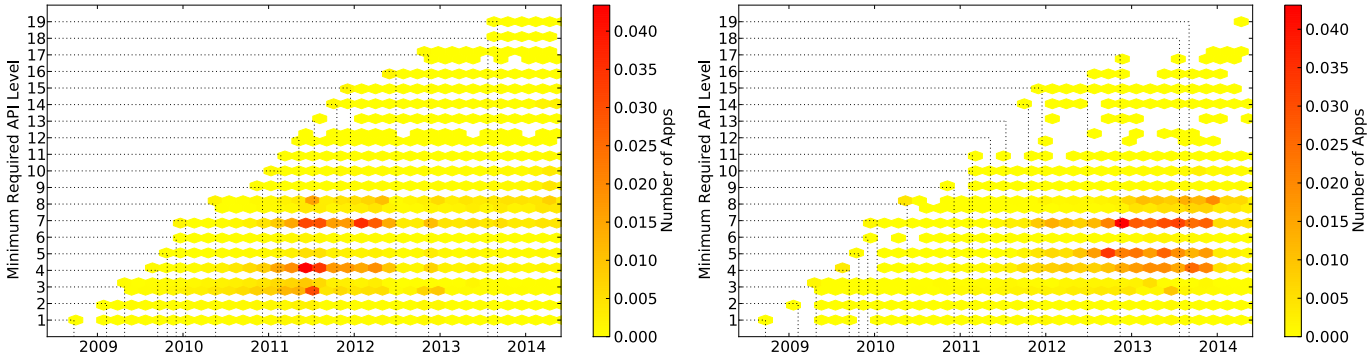

Fig. 8: Heatmap of API level adoption after Android SDK releases for goodware (left) and malware (right): goodware authors adopt new API levels much faster than malware authors, who try to maximize the potential user base for their apps.

TABLE II: Sources for apps in our dataset: sample exchange feeds have a high proportion of malware, while the Google Play Store and apps from torrents and direct downloads have low infection rates. Interestingly, not all samples from malware corpora are detected by AV scanners.

Category	Sample Exchange	Google Play	Alternative Markets	VirusTotal	Malware Corpora	Torrents	Direct Downloads	Unknown
All	683,842	125,602	60,951	37,499	5,997	17,916	1,704	159,040
Goodware	5.2%	88.73%	18.15%	0.20%	0.04%	88.50%	96.36%	78.65%
Malware	55.3%	1.60%	27.51%	98.65%	97.87%	1.60%	1.59%	7.56%

TABLE III: Most frequently requested permissions in goodware and malware by the percentage of apps in each set.

Goodware		Malware	
83.97%	INTERNET	95.37%	INTERNET
61.54%	ACCESS_NETWORK_STATE	91.42%	READ_PHONE_STATE
43.65%	WRITE_EXTERNAL_STORAGE	82.79%	WRITE_EXTERNAL_STORAGE
38.09%	READ_PHONE_STATE	71.99%	ACCESS_NETWORK_STATE
23.59%	ACCESS_COARSE_LOCATION	69.91%	SEND_SMS
22.51%	VIBRATE	60.67%	RECEIVE_SMS
21.56%	ACCESS_FINE_LOCATION	55.66%	INSTALL_SHORTCUT
19.32%	WAKE_LOCK	51.40%	WAKE_LOCK
18.05%	ACCESS_WIFI_STATE	48.73%	READ_SMS
12.11%	READ_CONTACTS	45.62%	RECEIVE_BOOT_COMPLETED
11.83%	RECEIVE_BOOT_COMPLETED	40.15%	ACCESS_WIFI_STATE
8.30%	CALL_PHONE	32.92%	WRITE_SETTINGS
8.15%	CAMERA	30.05%	READ_CONTACTS
7.66%	GET_TASKS	25.74%	CALL_PHONE
7.45%	SEND_SMS	24.70%	ACCESS_COARSE_LOCATION
6.72%	GET_ACCOUNTS	24.30%	ACCESS_FINE_LOCATION
6.31%	WRITE_SETTINGS	23.83%	VIBRATE
6.11%	WRITE_CONTACTS	23.04%	GET_TASKS
5.02%	SET_WALLPAPER	20.15%	WRITE_SMS
4.96%	CHANGE_WIFI_STATE	20.12%	CHANGE_WIFI_STATE
4.57%	INSTALL_SHORTCUT	19.21%	SYSTEM_ALERT_WINDOW
4.47%	RECEIVE_SMS	19.11%	CHANGE_NETWORK_STATE
4.03%	RECORD_AUDIO	17.81%	GET_ACCOUNTS
4.00%	READ_CALENDAR	13.93%	INSTALL_PACKAGES
3.79%	READ_LOGS	13.23%	UNINSTALL_SHORTCUT

TABLE IV: Most frequently registered broadcast receivers in goodware and malware by the percentage of apps in each set.

Goodware		Malware	
11.29%	BOOT_COMPLETED	56.32%	BOOT_COMPLETED
8.93%	APPWIDGET_UPDATE	41.73%	SMS_RECEIVED
8.74%	INSTALL_REFERRER	14.56%	CONNECTIVITY_CHANGE
6.74%	SCREEN_OFF	13.49%	DATA_SMS_RECEIVED
6.69%	USER_PRESENT	11.95%	AIRPLANE_MODE
4.17%	CONNECTIVITY_CHANGE	10.18%	PACKAGE_ADDED
2.40%	PACKAGE_ADDED	4.24%	NEW_OUTGOING_CALL
2.38%	IN_APP_NOTIFY	2.66%	USER_PRESENT
2.25%	SMS_RECEIVED	2.14%	BATTERY_CHANGED
1.43%	PHONE_STATE	1.72%	DEVICE_ADMIN_ENABLED
0.91%	MEDIA_BUTTON	1.60%	INSTALL_REFERRER
0.78%	PACKAGE_REMOVED	1.50%	APPWIDGET_UPDATE
0.70%	SERVICE_STATE	1.43%	PHONE_STATE
0.65%	SCREEN_ON	1.40%	BATTERY_CHANGED_ACTION
0.64%	MEDIA_MOUNTED	1.03%	PACKAGE_REMOVED
0.61%	NEW_OUTGOING_CALL	0.90%	UNINSTALL_SHORTCUT
0.60%	BATTERY_CHANGED	0.90%	INSTALL_SHORTCUT
0.47%	PACKAGE_REPLACED	0.90%	SIG_STR
0.40%	DEVICE_ADMIN_ENABLED	0.88%	ACTION_POWER_CONNECTED
0.36%	DEVICE_STORAGE_LOW	0.70%	SCREEN_OFF
0.32%	STATE_CHANGE	0.61%	PICK_WIFI_WORK
0.27%	TIME_SET	0.51%	TIME_SET
0.25%	WAP_PUSH_RECEIVED	0.41%	WAP_PUSH_RECEIVED
0.25%	ACTION_POWER_CONNECTED	0.39%	SCREEN_ON
0.24%	MEDIA_UNMOUNTED	0.36%	BATTERY_LOW

TABLE V: Most popular advertisement libraries in goodware and malware by the percentage of apps in each set.

Goodware		Malware	
36.76%	AdMob (Google)	5.74%	AdMob (Google)
5.61%	Flurry	3.90%	WAPS
4.00%	Millenial Media	2.94%	Kuogo
2.92%	MobClix	2.92%	domob
2.72%	AdWhirl	2.67%	Adwo
1.94%	InMobi	2.02%	AirPush
1.77%	MobFox	1.97%	YouMi
0.91%	MoPub	1.43%	Vpon
0.78%	Adlantis	1.27%	Wooboo
0.74%	Admarvel	1.15%	MobWIN
0.67%	Smaato	0.91%	Millenial Media
0.63%	YouMi	0.84%	Flurry

TABLE VI: Information most commonly leaked to the network by goodware and malware by the percentage of apps in each set.

Goodware		Malware	
12.86%	IMEI	39.68%	IMEI
1.70%	IMSI	25.88%	IMSI
1.51%	PHONE_NUMBER	13.89%	PHONE_NUMBER
1.12%	LOCATION	4.34%	ICCID
1.12%	LOCATION_GPS	1.40%	CONTACTS
0.60%	ICCID	0.40%	PACKAGE
0.08%	PACKAGE	0.11%	SMS
0.06%	CONTACTS	0.11%	CALL_LOG
0.05%	SMS	0.10%	LOCATION
0.02%	CALL_LOG	0.10%	LOCATION_GPS
0.01%	BROWSER	0.07%	BROWSER
0.01%	CALENDAR	0.00%	TAINT_CAMERA