ANDRUBIS: Android Malware Under The Magnifying Glass

Lukas Weichselbaum[*], Matthias Neugschwandtner[*], Martina Lindorfer[*], Yanick Fratantonio[+], Victor van der Veen[†], Christian Platzer[*]

[*]Secure Systems Lab, Vienna University of Technology
{lweichselbaum,mneug,mlindorfer,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

NOTICE: An updated version of this report was published as **ANDRUBIS - *1,000,000 Apps Later: A View on Current Android Malware Behaviors*** *[20] at the International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS).*

***Abstract*—The smartphone industry has been one of the fastest growing technological areas in recent years. Naturally, the considerable market share of the Android OS and the diversity of app distribution channels besides the official Google Play Store has attracted the attention of malware authors. To deal with the increasing numbers 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 [8], [25], [26], [28], the resulting prototypes remain limited in terms of analysis capabilities and availability.**

In this paper we present ANDRUBIS, a completely automated, publicly available and comprehensive analysis system for Android applications. ANDRUBIS combines static analysis techniques with dynamic analysis on both Dalvik VM and system level, as well as several stimulation techniques to increase code coverage.

I. INTRODUCTION

With a market share of almost 80% [18], Android is undoubtedly the most popular operating system for smartphones and tablets, rivaled only by Apple's iOS. Naturally, cyber criminals are aware of this significant distribution. The fact that, unlike iOS, Android allows installation of apps from arbitrary sources without rooting the device first, is an additional incentive for criminals to focus on subverting the supply of apps with malicious code. Reports by antivirus (AV) companies back the increasing interest in malware for Android with concrete numbers: Sophos for instance reports collecting a total of 650,000 unique Android malware samples, with 2,000 new samples being discovered every day [27].

Google swiftly reacted to the growing interest of miscreants in Android: In February 2012 *Bouncer* [21] was revealed, a service that transparently checks apps submitted to the Google Play Store for malware. Google further reported that this service has led to a decrease of the share of malware in the Play Store by nearly 40%. However, Android users are not limited to the official Google Play Store when it comes to installing software. Apps are available from various sources – these can either be bulk archives which can be retrieved via torrents or one-click hosting services, or complete alternative app markets that come with a dedicated installer and host their own repositories.

Analyzing or detecting Android malware follows the same basic principle that research on x86 malware relies on. On the one hand, *static analysis* yields information immediately by just looking at a sample's application package and code, while *dynamic analysis* executes the sample in a sandbox and provides details on its behavior during runtime with the disadvantage of being slower and more resource intensive. A significant body of research [9], [17], [32] on Android malware uses these methods, while none of them provide a comprehensive technical solution that combines them to obtain a comprehensive feature set for a sample. However, post-analysis techniques such as clustering can produce more meaningful results if they are applied to a rich feature set.

As a consequence, we designed and implemented ANDRUBIS, an automated analysis solution for Android that records events on two different levels: Java code executed by the Dalvik Virtual Machine and native code executed at system level. As some characteristics are only exposed if they are triggered by specific interaction with the sample, we also provide targeted stimuli during the dynamic analysis. To be able to customize the set of stimuli for each sample we leverage information from prior static analysis.

As its name already suggests, we built ANDRUBIS as an extension to the public malware analysis sandbox Anubis [1], [6]. ANDRUBIS has been operating since June 2012 and has analyzed over 900,000 unique Android applications so far. ANDRUBIS achieves a throughput of around 3,500 analyses per day, with samples coming from market crawls, sample sharing with other researchers, and submissions through our web interface or directly from users' phones.

In summary, we present the following contributions:

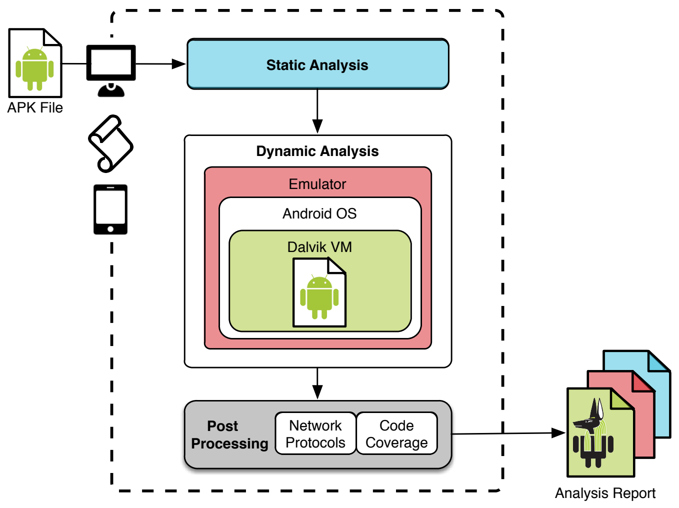

Fig. 1. Architecture of ANDRUBIS.

- We introduce ANDRUBIS, a fully automated analysis system that includes both static and multi-layered dynamic approaches to analyze unknown Android applications.
- We implement various stimulation techniques and verify their effectiveness by computing the resulting code coverage.
- We show by clustering our data set that the feature set produced by ANDRUBIS is rich enough to allow for researchers to build various post-processing methods upon.
- To provide our solution to the research community, we opened ANDRUBIS for public submissions under http://anubis.iseclab.org.

II. SYSTEM DESCRIPTION

ANDRUBIS is based on both static and dynamic analysis components complementing each other: results of the static analysis are used to perform more efficient and effective dynamic analysis. In the following we are going to detail the building blocks of ANDRUBIS and how they contribute to building a complete picture of an app's characteristics.

Figure 1 shows the architecture of ANDRUBIS and how the individual components relate to one another. During an analysis run, each app passes through the following stages:

1) **Static Analysis:** During this phase we extract information from an app's manifest and its actual bytecode.
2) **Dynamic Analysis:** This core phase executes the app in a complete Android environment, its actions being monitored at both the Dalvik VM as well as the system level.
3) **Post-Processing:** After the main analysis stages have completed, ANDRUBIS performs additional post-processing on the result, such as detailed analysis of the captured network traffic.

A. Static Analysis

Android applications are packaged in *Android Application Package* (APK) files that by definition contain a manifest file (`AndroidManifest.xml`). This description file is mandatory and without its information, the application cannot be installed or executed. As a first step, we unpack the archive and parse meta information such as requested permissions, services, broadcast receivers, activities, package name, and SDK version from the manifest.

In addition to this we examine the actual bytecode to extract a complete list of available Java objects and methods. We use the gathered information to assist in automating the dynamic analysis as well as identifying permissions, which are dangerous or commonly used by malware. Furthermore, this gives us an idea on how many permissions are requested by the app in the first place, compared to which permissions are actually used to implement the app's functionality.

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, as it is the typical environment for Android. We chose a Qemu-based emulation environment capable of running arbitrary Android applications and monitoring behavior that happens within the operating system. Since Android applications are based on Java, we closely monitor the underlying VM, called the Dalvik VM, and record actions happening within this environment. This allows us to monitor file system as well as phone events, such as outgoing SMS messages and phone calls. For a comprehensive analysis, however, these capabilities are not sufficient. Therefore, our emulator provides the following additional facilities:

- **Tainting:** To track privacy sensitive information ANDRUBIS uses taint tracking at the Dalvik level [12] which enables us to detect sensitive information leaving the phone through taint analysis.
- **Method Tracing:** We record invoked Java methods, their parameters and their return values. Combined with our static analysis, we use method traces to measure the code covered during an analysis run. This helps us to evaluate and improve our stimulation engine.
- **Emulator Introspection:** To provide a means for an 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 the outside and to track system calls of potentially harmful native libraries.

The rest of the sandboxing system (host environment, network setup, database, etc.) is comparable to conservative analysis systems. To mitigate potentially harmful effects of our analysis environment, we took precautions to prevent samples from executing DoS attacks, sending spam e-mails or propagating themselves over the network. This part is based on our experience with x86 malware analysis and proved to be effective in the past [6].

TABLE I. PERFORMED STIMULATION EVENTS.

Stimulation Event	Target
Activities	Activities declared in the manifest
Services	Services declared in the manifest
Broadcast Receivers	Broadcast receivers declared in the manifest and registered dynamically during runtime
Common Events	SMS, WiFi+3G connectivity, GPS lock, phone calls, phone state changes
Random Events	Random input stream by the Application Exerciser Monkey

C. Stimulation

The purpose of stimulation is to exhaustively explore the functionality provided by a program. One major drawback of dynamic analysis in general is the fact that only a few of the possible execution paths are traversed within one analysis run. Fortunately, the specifics of the Android OS provide some facilities to alleviate this problem. Since the application's manifest defines a list of the various application components (services, broadcast receivers, and activities), we can stimulate them individually. Furthermore, we can produce a set of common events that malware samples are likely to react to. Thus, our stimulation approach includes the following sequence of events: After the initialization of the emulator, ANDRUBIS installs the application under analysis and starts the main activity. At this point, all predefined entry points are known from static analysis. Further, ANDRUBIS keeps track of dynamically registered entry points, enabling it to perform the stimulation events listed in Table I.

Activities. An activity provides a screen for the user to interact with. Activities have to be registered in the manifest and cannot be added programmatically. These activities define the interaction sequences presented to the user and come with a defined layout, which must be known in advance. By parsing the manifest, ANDRUBIS can invoke each activity separately, effectively iterating all existing dialogs within an application.

Services. Background processes on the Android platform are usually implemented as services. Other than activities, they come without a graphical component and are designed to provide background functionality for a program. Naturally, they are also of interest to malware authors, as they can be used to implement communication with botmasters, leak personal information or forward intercepted text messages to an adversary. Again, all services used by an application must be listed in the manifest. Their existence, however, does not automatically mean the service is started under every circumstance. To save battery life and preserve memory, services have to be started on demand, with a lifetime defined by the programmer. For ANDRUBIS we patched the *Activity Manager* component to iterate and start all listed services automatically after the application is deployed.

Broadcast Receivers. Another possibility to enter an Android app is by utilizing a broadcast receiver. These can be used to receive events from the system or other applications on the Android platform. For example, a broadcast receiver for the `BOOT_COMPLETED` event can be registered to start an application 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, they can be registered in the manifest, although this is not mandatory. In order to provide the possibility to react to certain events and realize communication with other applications dynamically, they can be registered and unregistered at runtime. Therefore, we intercept calls to `registerReceiver()` in order to obtain a list of dynamically registered events that can be triggered. Similar to the previous stimuli, ANDRUBIS uses the *Activity Manager* to invoke all statically registered broadcast receivers found in the manifest as well as those ones that have been dynamically registered.

Common Events. A far superior method compared to stimulating broadcast receivers with a targeted event is to emulate some common events samples might react to. In contrast to directed stimuli, these events occur at the system level and thus also trigger receivers from the Android OS itself. That, in turn, avoids causing inconsistent states the OS would have to recover from. By broadcasting common events such as incoming SMS , we are able to trigger most functions even if they propagate data by custom broadcast receivers. A list of currently implemented common events can be found in Table I.

Application Exerciser Monkey. The remaining elements that need to be stimulated are actions based on user input (e.g., button clicks, file upload, entry fields, 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 applications, 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 can also provide a fixed seed in order to always trigger the same interaction sequences.

D. Taint Tracking

Data tainting is a double-edged sword when it comes to malware analysis. On the 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 [10]. 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 data sources such as address book entries to the network is a valuable property of a dynamic analysis system. ANDRUBIS leverages *TaintDroid* [12] to track sensitive information across application borders in the Android system. The introduced overhead in processing time of approximately 15% [12] is also acceptable for our purpose. As a result, ANDRUBIS can log tainted information as it leaves the system through three sinks: the network, SMS, and files.

E. Network Analysis

Capturing network traffic is one of the essential parts when dealing with modern malware – C&C communication is undoubtedly one of its corner stones. In general, network traffic is one of the most important features for establishing a malware-detection metric. According to studies performed in production environments [16], more than 98% of x86 malware samples established a TCP/IP connection. Therefore, applications that neither request network connectivity, nor cause any traffic are less likely to be malware. Thus, in addition to tracking sensitive information to network sinks via tainting, we also record all the network activity during analysis regardless of the performed action or the application causing it. This is necessary because even when an app does not request Internet permissions, it is possible to use other installed apps like the browser, to still send data over the network. Another possibility not to request Internet permissions, but still cause network traffic is by exploiting the Android OS and circumvent the permission system as a whole. Finally, we extract high-level network protocol features from the network traffic that are suitable for identifying interesting samples. Currently, we focus on well-known protocols such as HTTP, DNS, FTP, SMTP and IRC.

F. 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 dump executed 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.

The trace output is separated per process and thread ID and written to separate log files. Like the output produced by our system-level analysis (described in the next section), it is not directly displayed in our web report. As these listings are quite large, we provide them on an on-demand basis for researchers and analysts rather than ordinary users.

Together with the output gained from system-level analysis, the fine-grained method traces can be leveraged for reverse engineering purposes, as input to machine learning algorithms or to create behavioral signatures.

We also use the trace output to measure the code covered during the stimulation phase of ANDRUBIS. To this end we first construct a list of executed method signatures, which we then map against the list of functions found during static analysis. We map functions 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.

Another use for the method trace is the extraction of used permissions during analysis. By looking up each API function that was called during analysis in an API-to-permission mapping such as the one provided by Stowaway [14] or PScout [5], we can determine the permissions an app used during runtime.

G. System-Level Analysis

In addition to monitoring the Dalvik VM, ANDRUBIS also tracks native code execution. By default, Android apps are Java programs, being distributed as an APK file, which is basically a JAR container. 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. Via the Java Native Interface (JNI) it is possible to use native code system-level libraries. This functionality is mainly intended for performance-critical use cases such as displaying 3D graphics. But apps are not limited to load the Android OS' native libraries; they can also load their own native libraries and thus execute their own system-level code. Naturally, such code would not be covered by a mere observation at the Dalvik VM-level. Most of recent research on Android malware only deals with the Dalvik VM-level and would thus miss malicious activity at system level. However, for malicious apps the use of native code is attractive as the possibilities to perform malicious activities, including executing a root exploit, are far greater than within the Dalvik VM.

There are a couple of ways to implement system-level instrumentation in Linux, 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 actually running Android OS, right in the emulator's codebase. To capture system-level behavior, we ultimately need to know what the library code loaded via JNI does. To this end, we intercept the Android dynamic linker's actions in order to track shared object function invocations and monitor all system calls. System call tracking bundled with this information enables us to associate system calls with invocations of certain functions of loaded libraries. The result is a complete list of system calls performed by the emulator as a whole. In order to identify only system calls invoked by the app under analysis we use its UID – in Android a unique UID is assigned to every app. By filtering the native code events by their corresponding UID, we can thus only monitor actions caused by a specific app.

Finally system level analysis allows us to monitor the usage of exploits to gain root privileges. As an example, Figure 2 shows how we can retrace all the steps of the "rage against the cage"-exploit (RATC) [4] in our system-level log: RATC first enumerates running processes to find the PID of the system daemon `adbd`. It then spans $> 6{,}000$ processes until the Android OS' process limit is reached. Finally, it kills `adbd`, which is subsequently restarted by the OS and would normally drop its root privileges via `setuid` right afterwards. The latter, security-critical step is, however, prevented by the fact that RATC has already spawned the maximum number of processes.

```
# enumerate processes to find adbd (PID 47)
sys_open,(filename,'/proc/1/cmdline'),(flags,0x20000),(mode,0x0)
sys_read,(fd,0x6),(buf,0xbefaeb18),(count,0xff)
...
sys_open,(filename,'/proc/47/cmdline'),(flags,0x20000),(mode,0x0)
sys_read,(fd,0x6), (buf,0xbefaeb18),(count,0xff)

# fork processes until RLIMIT_NPROC
sys_fork,SystemMonitor: PID: 593 UID: 10044 name: rageagainstthec
...
sys_fork,SystemMonitor: PID: 7241 UID: 10044 name: rageagainsttheca

# kill adbd process (PID 47)
sys_kill,(pid,0x2f),(sig,0x9)
```

Fig. 2. Excerpt from the system-level log for the rage against the cage exploit.

III. Evaluation

The primary goal of ANDRUBIS is to provide researchers with a comprehensive static and dynamic analysis report of an application, not to automatically identify applications as goodware or malware. Thus, the evaluation of ANDRUBIS aims to answer one basic question: Is the system fit to produce the needed data for malware analysis of Android apps? By clustering a data set of 27,000 applications from multiple sources, including known malware, we show that the feature set produced by ANDRUBIS is rich enough to be integrated into post-processing methods for an automatic malware classification. Furthermore, we also evaluate the effectiveness of the stimulation both in terms of observed behavior and code coverage, as well as the overall performance overhead of ANDRUBIS.

A. Clustering

In order to evaluate whether the feature set produced by ANDRUBIS is indeed rich enough to allow for proper results using post-analysis techniques, we clustered an evaluation data set comprised of 27,000 apps from a variety of sources, including the Google Play Store and known malware corpora such as the Genome Project [31]. One of the biggest advantages when dynamically executing apps is the possibility to create behavioral profiles based on the monitored data in addition to the wealth of static features that can be extracted from Android applications. In contrast to other approaches [9], [31], [32], we use the term *behavioral* for operations observed while a sample is executed. While requesting permissions is seen as a behavioral aspect by the authors, we consider these actions as static as we can derive them without executing the app. Thus, a profile with only static components is strictly speaking not a behavioral profile.

We create a *behavioral profile* for each application based on features observed during dynamic analysis as well as static features extracted from the APK files. The dynamic behavior includes features such as reading and writing to files, sending SMS, making phone calls, the use of cryptographic operations, the dynamic registration of broadcast receivers, loading DEX classes and native libraries and leaking sensitive information to files, the network and via SMS. Additionally, network-related dynamic features are generated by parsing the captured network dump and extract contacted endpoints, ports and communication protocols. Static features include activities, services and broadcast receivers parsed from the manifest as well as required permissions and statically extracted URLs. We define the distance between two apps as the Jaccard distance between their profiles.

To overcome the computational complexity of exact clustering and process the behavioral profiles of 27,000 applications within a reasonable amount of time, we utilized the clustering approach Bayer et al. already applied to the clustering of Windows malware [7]. This clustering algorithm is based on locality sensitive hashing (LSH), and provides an efficient solution to the approximate nearest neighbor problem (ϵ-NNS). LSH can be used to perform an approximate clustering while computing only a small fraction of the $\frac{n^2}{2}$ distances between pairs of points. Leveraging LSH clustering, we are able to compute an approximate, single-linkage hierarchical clustering for our complete data set.

Under the assumption that the extracted feature set is rich enough, the clusters should expose applications with common properties. With the already categorized malware from the Genome Project as well as AV labels from VirusTotal we have a ground truth that allows us to identify clusters containing malware and find variants of similar samples from other sources. It also allows us to identify previously unknown samples when they are placed in the same cluster due to similarities in behavior and/or static features. We picked the most interesting clusters based on dynamic features alone and a combination of dynamic and static features and provide a short discussion on their properties in the following two paragraphs.

We first clustered samples using only dynamic features. The largest resulting clusters were defined by the behavior of advertisements. Applications that include the same ad library for displaying advertisements connect to the same server and therefore feature similar dynamic results. Unsurprisingly, the largest cluster features apps using AdMob as their ad library. An interesting side effect of these results is to see the approximate share of advertisement for each provider. Another strong characteristic exhibited by large clusters are information leaks. For one cluster represented by 38 apps 65% of the

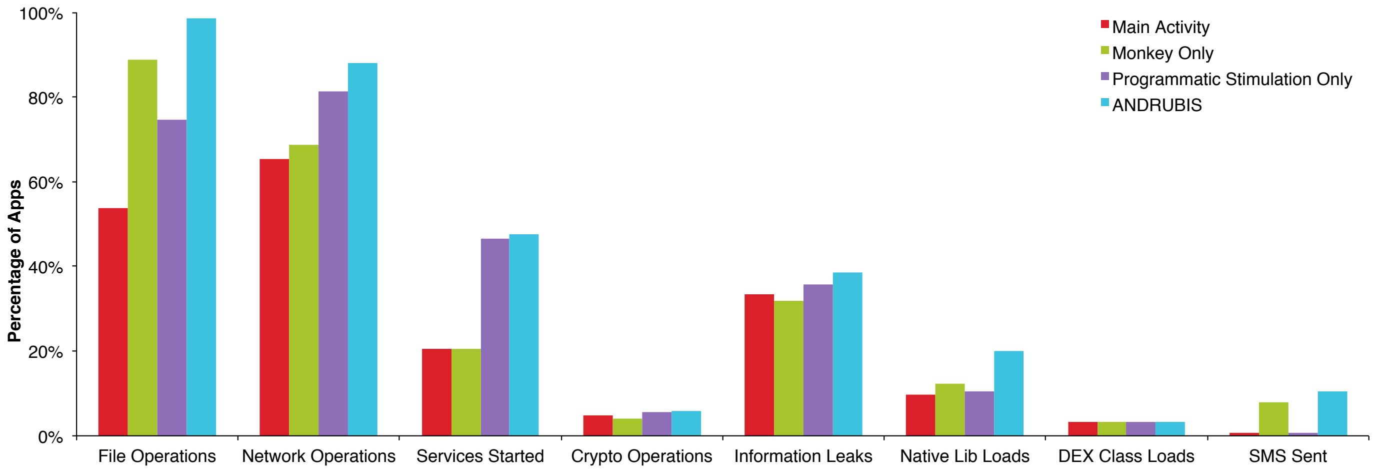

Fig. 3. Percentage of apps that showed specific operations when using each stimulation method.

corresponding samples belong to the already classified malware family *DroidKungFu*, the remaining 35% stem from the official app store. The cluster's determining factor is device ID leakage over the network. Samples of a comparable cluster of size 23 leak phone number and other database content. 69% of the correlating samples stem from our malware collections, while 31% can be found in the Play Store.

When combining static features and dynamic behavior to a more complete profile, the growing amount of features enables us to watch for larger clusters. With 216 elements, we found a set of apps that all belong to the *BaseBridge* malware family. These samples are primarily distinguished by the large set of permissions they request, 15 per app on average. All samples from that cluster belong to one of our malware sets.

Taken as a whole, the combined clustering can be used as a means to reduce the set of apps that have to be screened manually. With a reference set, the data provided by both, static and dynamic analysis elements can be leveraged to deduce a malware rating scheme or at least provide a reduced list of suspicious apps to be screened by a human analyst.

B. Stimulation

As an integral part of the analysis environment, we also evaluated our stimulation engine's effectiveness. For this purpose, we selected a set of 250 malicious and 250 benign apps. All benign apps were taken from the official Play Store. The malware samples are a random selection of AV-labeled samples. However, in order to select only apps that showed at least some interesting behavior, we first discarded apps showing no activity during dynamic analysis.

To better distinguish between programmatically introduced stimulation events and the GUI-based exerciser monkey, we ran separate tests with all permutations of these two stimulation methods. Figure 3 shows the percentage of apps that exhibit a specific behavior after stimulation. The first bar shows results when only the main activity of an app is invoked. This corresponds to a user starting the app. The second bar shows the result with the exerciser monkey in addition to invoking the main activity. The third bar shows the results of using our newly developed stimulator alone. Finally the forth bar puts all stimulation facilities together, as implemented in ANDRUBIS.

Taking the first category as an example, we see that only 54% of all apps perform file operations if we trigger the main intent after installation. With all elements from our stimulation engine, this percentage increases to 99%. The graph also shows that different stimulation methods are better suited for some events than others. Services, for instance, were almost exclusively triggered by our service iterator, while the exerciser monkey alone triggered a large portion of SMS activity. As expected, a combination of all techniques always surpassed behavioral coverage of a single technique.

The type of analyzed apps also causes differences in stimulation effectiveness. Games, for instance, are hard to stimulate with the monkey, while other apps are hard to activate programmatically. Figure 4 shows a three-minute analysis run of three different applications. MonkeyJump, a piece of malware distributed within a game lies dormant during the monkey phase, while App Manager Pro reacts positively to this form of stimulation. For NZ Subway & Bus Time both the programmatic stimulation and the monkey trigger a considerable amount of events. In conclusion, both GUI-based and programmatically triggered stimulation are necessary to achieve the best coverage.

C. Code Coverage

In order to understand the effectiveness of ANDRUBIS and its stimulation techniques in more detail, we further compute the obtained code coverage. As described earlier, we use static analysis to generate a complete function footprint of the target app. We then map each function invocation from the method trace output against this footprint and calculate the percentage of functions called during the individual stimulation phases of dynamic analysis.

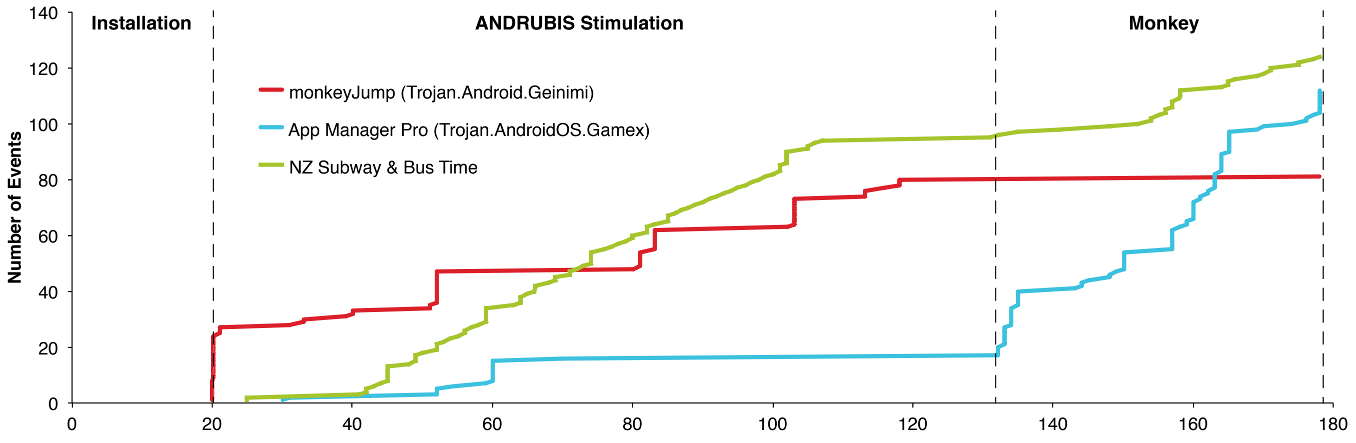

Fig. 4. Observed number of events for three applications during the phases of an analysis run.

TABLE II. CODE COVERAGE DURING INDIVIDUAL STIMULATION PHASES.

data set	broadcast	activities	services	monkey	sum
benign apps	0.79%	21.83%	0.56%	24.81%	27.74%
malicious apps	4.68%	15.14%	7.14%	19.17%	27.80%

TABLE V. BENCHMARK RESULTS FOR CPU AND I/O PERFORMANCE.

	Baseline (Qemu)	ANDRUBIS w/o VMI	ANDRUBIS	ANDRUBIS singlestep	Samsung I9001
AnTuTu	328	307	277	255	2134
Overhead	0%	7%	18%	29%	-

Table II lists the average code coverage per stimulation phase on our subset of 250 malicious and 250 benign applications. Overall, ANDRUBIS achieved an average code coverage of around 27.74% on benign applications and 27.80% on malicious applications. 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 ad libraries. Thus, for a less conservative code coverage computation we could whitelist known third-party APIs to get a better indication of the number of called functions that were written by the app authors themselves.

When looking at the coverage results for the individual stimulation phases, we can see that for both benign and malicious applications a large portion of the code coverage comes from activity stimulation and the exerciser monkey. Furthermore, while the stimulation of broadcast receivers and services has a negligible effect on benign applications, it triggers a considerable amount of functions in malicious applications. This is not surprising, as malware apps tend to register services and listen to broadcast events in order to operate without user interaction.

We also analyzed a handful of apps by hand using a custom system image that enables method tracing. Table III shows the result for 15 benign apps while Table shows the results for 15 malicious apps. In general, ANDRUBIS performs better for malicious applications, surpassing the code covered by manual analysis for some apps. This is likely caused by external stimulations, such as a reboot or the receipt of SMS that were not triggered during manual analysis. Overall, the differences between manual and automated analysis are below 10% for both benign and malicious applications. We plan to narrow this gap even further with a more targeted user interface stimulation than the random events caused by the exerciser monkey in the future.

D. Performance

Finally, we measure the performance of ANDRUBIS in different configurations and compare it to the performance of real hardware. Table V shows measurements with the AnTuTu Android benchmark, which we configured to rate CPU and I/O performance. The baseline for our measurements is the plain Qemu emulator running a vanilla Android image. Adding the instrumentation at the Dalvik level causes a negligible 7% overhead, with VMI monitoring the native code raising it to 18%. For reasons further explained in Section IV we have also measured the performance with the Qemu single-step mode enabled. Finally running the benchmark on a real-world device shows that the overhead additionally introduced by instrumentation of the Dalvik VM and the emulator is negligible when comparing the overhead introduced by the emulator alone with an actual smartphone: The Samsung I9001 is more than six times faster than the baseline.

IV. LIMITATIONS

Naturally, an automated analysis environment like ANDRUBIS comes with some limitations. One of the most severe problems for any VM-based approach is evasion. Even when executing x86 virtual machines on an x86 host, the possibility to detect certain features of the execution environment exists. Possibilities reach from iterating certain device properties to reveal the underlying virtualization technology, to querying for specific pixel colors of the Desktop background in order to detect a specific analysis system. Previous research has shown that despite being widespread, analysis evasion is not ubiquitously implemented in x86 malware [11], [19]. Whether this assumption holds true for mobile sandboxes is hard to estimate. In our opinion, the fact that Google introduced a feature to check third-party apps with Google Bouncer in

TABLE III. CODE COVERAGE OF ANDRUBIS COMPARED TO MANUAL ANALYSIS (BENIGN APPLICATIONS).

Application	Category	Manual	Calls (of total)	ANDRUBIS
`com.skylineapps.opentech`	Business	8.91%	27 of 303	+1.32%
`com.lftechs.tictactoe.free`	Games	34.52%	107 of 310	-4.19%
`ynd.tapmadness`	Games	24.08%	657 of 2728	-3.45%
`com.AndPhone.game.Defense`	Games	31.47%	772 of 2453	-12.27%
`com.via3apps.sensacio142`	Entertainment	35.24%	160 of 454	-32.38%
`com.baste.bender`	Entertainment	58.14%	125 of 215	-27.91%
`com.rpg90.seasons_cn`	Music & Audio	24.43%	472 of 1932	-5.33%
`com.omgbutton`	Music & Audio	37.70%	184 of 488	-17.83%
`com.brightai.middlesboroguide`	Sports	7.55%	216 of 2860	-0.80%
`com.snoffleware.android.rationalcalcfree`	Productivity	3.89%	166 of 4269	-0.56%
`org.steele.david.silentOnOff`	Productivity	56.83%	79 of 139	-9.35%
`com.accesslane.screensaver.shootinggallery.lite`	Screensaver	41.95%	146 of 348	-16.09%
`com.appspot.yongSubway_NZ`	Travel	100.00%	2 of 2	0.00%
`com.hetverkeer.info`	Travel	47.95%	105 of 219	-12.79%
`height.wallfeb28m`	Wallpapers	20.68%	97 of 469	-0.21%
Average code coverage		35.56%		-9.47%

TABLE IV. CODE COVERAGE OF ANDRUBIS COMPARED TO MANUAL ANALYSIS (MALICIOUS APPLICATIONS).

Application	AV Label (F-Secure, Kaspersky, Sophos)			Manual	Calls (of total)	ANDRUBIS
`com.keji.danti922`	BaseBridge.A	BaseBrid.a	Anserv-A	41.34%	296 of 716	-19.07%
`com.software.application`	Boxer.C	FakeInst.a	Boxer-D	14.15%	15 of 106	+10.21%
`org.zhou.cash.yy`	DroidKungFu.C	KungFu.a	KongFu-A	46.80%	476 of 1017	-30.07%
`tp5x.WGt12`	Fakeinst.L	FakeInst.ed	Opfake-E	39.77%	35 of 88	-11.05%
`org.cahlomi.dmugeiwawbrgt`	Frogonal.A	GinMaster.a	Frogonal-A	22.39%	245 of 1094	-18.65%
`com.gkiksfsle`	Frogonal.A	GinMaster.a	Frogonal-A	10.05%	584 of 5813	-9.16%
`org.snakemaxa.apps.app_uninstall`	Gamex	Gamex.a	Gamex-Gen	10.31%	234 of 2269	+1.06%
`com.bfsx.papertoss`	Gamex.A	Gamex.a	Gamex-Gen	22.31%	620 of 2779	-11.71%
`com.doidlonghair1`	GinMaster.A	GinMaster.a	Gmaster-A	15.83%	132 of 834	+18.35%
`com.load.wap`	JiFake.F	FakeInst.a	FkToken-A	52.00%	52 of 100	+9.76%
`com.zhenshi.Haidaogame`	Kituri.A	Placms.a	Kituri-A	17.10%	85 of 497	-4.30%
`com.zs.terence.calendar`	Kituri.A	Placms.a	Kituri-A	13.17%	64 of 486	-1.64%
`com.gamejing.box`	Kituri.A	Placms.a	Kituri-A	8.74%	41 of 469	-6.25%
`fhvm.vnnej`	OpFake.E	Opfake.bo	Opfake-F	28.30%	30 of 106	+0.94%
`ru.mskdev.andrinst`	SMStado.A	FakeInst.a	Boxer-D	67.74%	21 of 31	-33.92%
Average code coverage				27.33%		-7.03%

Android Version 4.2 [23] is a strong hint that malware writers will have to put more effort into evading analysis environments.

Additionally, a characteristic intrinsic to the design of efficient emulators allows a more fundamental detection mechanism to be utilized. An emulator usually takes a basic block, translates it, and executes the whole resulting basic block on the host machine. Unfortunately, this property allows for an easy detection of emulated code, since basic blocks cannot be interrupted by the (guest) operating system's scheduler. In [22] the authors leverage this knowledge to detect emulator-based sandboxes. They also introduce a proof of concept for their approach, targeting our system. We reacted to this detection approach by using single-stepping in our analysis.

A general limitation of dynamic analysis systems is the never-ending arms race between malware developers and security researchers. As long as a sandbox is not capable of perfectly emulating a system, a possibility to detect it will always exist. Therefore, raising the bar for attackers as high as possible is the only feasible thing to do.

V. RELATED WORK

Research in mobile malware has experienced a tremendous boom in the last few years. With the appearance of the first malicious apps, the research community launched various projects to shed some light on mobile malware.

Felt et al. [13] analyzed a total of 46 iOS, Symbian and Android malware samples in detail to provide one of the first surveys on mobile malware and their author's incentives. Along with the processed information, the authors provide a list of dangerous permissions these apps used. Burguera et al. [9] presented an approach to identify malicious applications that does not rely on requested permissions but uses syscall information. The authors use strace to extract vectors reflecting the number of invocations for each possible syscall from applications. To detect malware, the authors rely on k-means clustering over the available vectors. The authors of AdRisk [17] focused on detecting privacy and security risks in in-app advertisement libraries. They statically analyzed apps from the Google Play Store to identify the potential of included ad libraries to leak private information and execute untrusted code. We can support this result with our findings presented in Section III. Zhou et al. [30] analyzed official and third-party markets for repacked binaries and discovered that 5% to 13% of applications are repacked versions of existing applications from the original market. The repacked versions in their evaluation are mainly used to replace ad libraries and thus re-route ad revenues, but they also found repacked applications with additional malicious payloads. While the authors used fuzzy hashing to statically generate and compare app fingerprints, ANDRUBIS could also identify the additional malicious behavior during runtime.

Concerning systems for the large-scale dynamic analysis of Android applications, the vision paper of Gilbert et al. [15] was the first to propose a system like ANDRUBIS. With the only exception of using taint tracking instead of dependency graphs to determine the source of malicious actions, ANDRUBIS incorporates every element discussed in this work. For a thorough analysis, however, we extended the system to also track JNI invocations on emulator level.

DroidScope [28] is a dynamic analysis system that solely uses 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 will probably require a substantial amount of adaption whenever Google pushes an update.

DroidRanger [32] pre-filters applications based on a manually created permission-fingerprint before subjecting them to dynamic analysis. The authors use it to compare 200,000 apps from different markets. In contrast to this approach, we have analyzed every app with ANDRUBIS, yielding full behavioral profiles to base our evaluation on. Furthermore, DroidRanger performs system-level monitoring through a kernel module instead of VMI and focuses only on system calls used by existing Android root exploits. Finally, the dynamic analysis part of DroidRanger does not employ stimulation techniques.

Regarding stimulation of applications during the analysis, both SmartDroid [29] and AppsPlayground [24] try to drive the app along paths that are likely to reveal interesting behavior through targeted stimulation of GUI elements. Their approaches can be seen as intelligent enhancements of the existing Application Exerciser Monkey and our custom stimulation of activity screens. They are largely orthogonal to our work, which also focuses on stimulating broadcast receivers, services and common events.

Publicly available analysis systems for Android are Badger [2] and Mobile Sandbox [3], [26]. In contrast to ANDRUBIS, Badger performs only static code analysis to test for data leaks and lists permissions as well as identify used ad libraries. The Mobile Sandbox project is more mature as it claims to perform dynamic analysis as well. Unfortunately, we were not able to perform an in-depth comparison, as both systems seem to be unable to cope with their submission load – our samples are stuck in the input queues to these systems. We emphasize that ANDRUBIS's design allows for large-scale deployments that can easily handle a big workload.

VI. CONCLUSION AND FUTURE WORK

In this paper we presented ANDRUBIS, a fully automated large-scale analysis system for Android applications that combines static analysis techniques with dynamic analysis on both Dalvik VM and system level. The presented results are consistent with previous research and verify the system's soundness and effectiveness for analyzing Android apps. Furthermore, we implemented several stimulation techniques in order to trigger behavior during the analysis and verified their effectiveness by evaluating the resulting code coverage. To further enhance behavioral coverage, we plan to adopt methods similar to the ones of SmartDroid [29] and AppsPlayground [24].

We opened ANDRUBIS for public submissions with a current capacity of analyzing around 3,500 samples per day, resulting in a total of almost 900,000 analyzed apps to date. With ANDRUBIS, we provide malware analysts with the means to thoroughly analyze a given Android application. Furthermore, we provide researchers with a solid platform to build various post-processing methods upon. For example, machine learning approaches could use our analysis results to tackle the problem of judging whether a previously unseen app is malware or not.

Finally, we also provide an Android app to submit samples directly from a smartphone. It acts as a front-end for ANDRUBIS and features submission of an installed app to our system and displaying a summary of our analysis results for the user.

ACKNOWLEDGMENT

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] "Anubis," http://anubis.iseclab.org.
[2] "Badger Application Analysis," http://davidson-www.cs.wisc.edu/baa.
[3] "Mobile Sandbox," http://mobilesandbox.org.
[4] "Rage against the cage," http://thesnkchrmr.wordpress.com/2011/03/24/rageagainstthecage/, March 2011.
[5] 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.
[6] 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.
[7] 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.
[8] 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.
[9] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-Based Malware Detection System for Android," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.
[10] 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.
[11] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware," in *Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks (DSN)*, 2008.
[12] 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.

[13] 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.

[14] 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.

[15] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung, "Vision: Automated Security Validation of Mobile Apps at App Markets," in *Proceedings of the 2nd International Workshop on Mobile Cloud Computing and Services (MCS)*, 2011.

[16] 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.

[17] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe Exposure Analysis of Mobile In-App Advertisements," in *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*, 2012.

[18] 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, Feb 2014.

[19] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, "Detecting Environment-Sensitive Malware," in *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[20] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, "Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors," in *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

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

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

[23] J. Raphael, "Exclusive: Inside Android 4.2's powerful new security system," http://blogs.computerworld.com/android/21259/android-42-security, November 2012.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.