ANANAS – A Framework For Analyzing Android Applications

Thomas Eder, Michael Rodler, Dieter Vymazal, Markus Zeilinger
Department Secure Information Systems
University of Applied Sciences Upper Austria
{*thomas.eder, michael.rodler*}*@students.fh-hagenberg.at*
{*dieter.vymazal, markus.zeilinger*}*@fh-hagenberg.at*

Abstract—**Android is an open software platform for mobile devices with a large market share in the smartphone sector. The openness of the system as well as its wide adoption lead to an increasing amount of malware developed for this platform. ANANAS is an expandable and modular framework for analyzing Android applications. It takes care of common needs for dynamic malware analysis and provides an interface for the development of plugins. Adaptability and expandability have been main design goals during the development process. An abstraction layer for simple user interaction and phone event simulation is also part of the framework. It allows an analyst to script the required user simulation or phone events on demand or adjust the simulation to his needs. Six plugins have been developed for ANANAS. They represent well known techniques for malware analysis, such as system call hooking and network traffic analysis. The focus clearly lies on dynamic analysis, as five of the six plugins are dynamic analysis methods.**

Keywords-**Smartphone security, Android malware, automated malware analysis**

I. Introduction

Android is an operating system and open software platform for mobile devices based on the linux kernel. Since its first appearance in 2008, it became a big success story. In the third quarter of 2012, 72.4% of all mobile devices sold to end users were powered by Android [1]. This makes it the most installed OS on recently sold mobile devices [2]. The currently most used Android versions are, 2.3 (*Gingerbread*) with a share of 39.8%, and the versions 4.0 (*Ice Cream Sandwich*), 4.1 and 4.2 (*Jelly Bean*) with a share of 50% [3].

The increasing number of smartphones based on Android and the openness of the system, led to an increase of malware developed for the platform. In their threat report for the fourth quarter of 2012 [4], McAfee mentions that their "mobile malware zoo" accomodates a total of 36,669 samples by the end of 2012. 95% of these samples were gathered in 2012. McAfee also observed that the growth of mobile malware almost doubled in each of the last two quarters of 2012. From all mobile malware samples that have been gathered, 97% are targeting Android. Android malware was even used to carry out targeted attacks, as Kaspersky reported in a security alert [5]. This increase of malicious applications targeting the Android platform and its users, needs to be addressed urgently. To keep up with this huge growth rate of malware targeting Android, tools for automated and semi-automated malware analysis are much needed.

In this paper, ANANAS, a framework for **An**alyzing **An**droid **A**pplication**s** (APKs), focused on automated static and dynamic malware analysis, is presented. While other projects aim at developing different methods and tools for Android malware analysis, ANANAS is a framework, which allows the implementation and integration of several different analysis methods into one powerful platform. It was developed with the following goals in mind:

- To provide a core, which takes care of common needs for dynamic analysis, such as starting a clean environment for each analysis.
- To be useable for all Android versions.
- To provide adjustable user interaction and phone event simulation.
- To allow the integration of plugins for static or dynamic analysis, which can be called at different stages of the analysis process.
- Results should be available in several formats at different levels of detail.
- To be able to tailor the framework to specific needs by providing high configurability.

By providing the ANANAS framework to the community, we hope to ease and facilitate the development of new and improved malware analysis methods. Together with static and dynamic analysis plugins, ANANAS becomes a comprehensive tool for analyzing Android applications.

The remainder of this paper is structured as follows: In section II we take a look at related projects. The architecture of ANANAS is described in section III and section IV explains the components in greater detail. In section V, several plugins developed for the framework are presented. To complete this paper, in section VI results of an experimental evaluation are shown.

II. Related Work

A popular tool for dynamic analysis of Windows applications is Anubis [6]. Recently it was extended to allow the analysis of Android applications (codename Andrubis) [7]. It leverages several existing tools, like DroidBox [8], TaintDroid [9], apktool [10] and androguard [11] for static and dynamic analysis of Android applications (APKs). Unfortunately, not much about its inner working or architecture is public. However, a comparison between Andrubis and ANANAS is drawn in the experimental evaluation section.
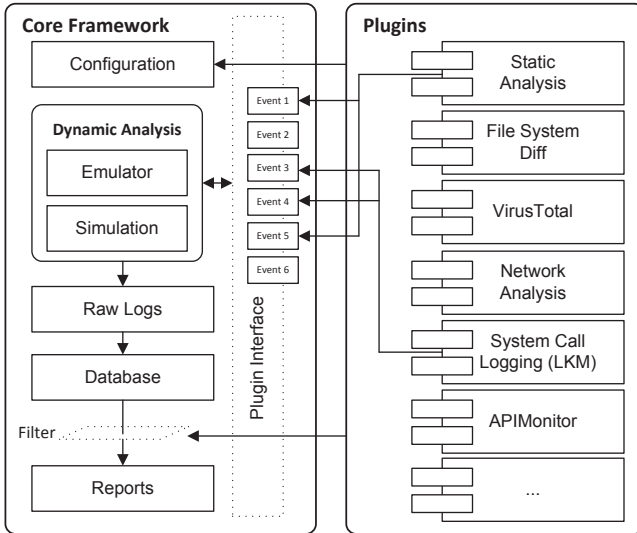
Figure 1.   ANANAS Architecture

Another tool similar to ANANAS and Anubis is Mobile-Sandbox [12]. It combines static and dynamic analysis techniques for analyzing Android applications. Unfortunately, there were no dynamic analysis results publicly available at the time of writing.

DroidScope [13] is a dynamic malware analysis tool that is built on top of QEMU and relies on introspection of an emulated system running the application in question. It is able to reconstruct the OS-level view (e.g. system calls) and the Java-level view (e.g. instructions in the Dalvik VM, Android's Java Virtual Machine). It is similar to ANANAS in the way that it uses dynamic analysis techniques and exports APIs, which can be used to develop plugins.

Recently, Rastogi et al. published their work on AppsPlayground [14], which is a framework for automated dynamic security analysis of Android applications. Among other things, they focus on detection evasion and automated exploration techniques for increased coverage of the applications code and therefore to trigger the malicious behaviour of an application. The detection/analysis techniques of AppsPlayground are similar to those of ANANAS but the approaches used for the implementation are different. ANANAS for example avoids intrusive changes to the code of the original Android system to be able to adopt new Android versions more quickly. In contrast, AppsPlayground uses TaintDroid, which modifies the original Android version heavily.

III.   ANANAS ARCHITECTURE

Figure 1 shows the basic architecture of ANANAS. It is composed of the core framework (written in Python) and several analysis plugins, which implement different analysis methods. The configuration system is shared between the framework and the plugins, which allows plugins to alter the configuration of the framework. The plugins can register to several events that are raised within the framework to run their code. An analysis run yields raw

logfiles that are produced by the plugins and saved into a database. To generate a readable report, the framework uses filters defined by the plugins.

The framework provides common services that are needed for the dynamic analysis. One requirement of dynamic malware analysis is that a clean, emulated environment is initialized for every analysis run. To emulate a smartphone, the emulator, which is shipped together with the Android SDK, is used. The framework itself is independent of the used Android version.

The analysis of Android malware differs from the analysis of Windows based malware because most Android malware is only triggered by certain phone events or user interaction. It is therefore crucial to simulate interaction with the emulated smartphone during the analysis phase. As the malicious behaviour of different malware families is triggered by different events, the simulation has to be adjustable. ANANAS achieves this by introducing a scripting language for user and phone event simulation.

The modularity and extendability of ANANAS is accomplished through the implementation of an event-based plugin system. Individual analysis methods are implemented as plugins, which can register to several events that are specified within the core framework. Every time a certain event is raised, methods for all plugins registered to this event are called by the core framework. Currently, six plugins are implemented, which are described in greater detail in section V.

Plugins can supply custom filtering mechanisms for further reducing the verbosity of the report when searching for very specific behaviour within an Android app. The filters follow a blacklist approach (exceptions can be defined) and are applied during report generation.

The result of every analysis is a report containing general information about the Android application and the results of the different plugins. The results of each plugin pass through several different stages and formats. Primarily, each plugin saves its results to a raw log file (JSON formatted). This log file offers the most detailed view on the analysis run. By saving the results into a database in the next step, an analyst gets the chance to use SQL queries for comparing results and generating statistical data. Finally, the plugin specific filters are applied to generate a condensed report in XML format, which can be used for a first examination.

An overall design goal during the development of ANANAS was to keep every part of it as configurable as possible. This includes the core framework, the plugins, the simulation and the filter process. Each plugin can specify its own settings in a separate section of the main ANANAS configuration file. The filtering mechanisms and the simulation part are configured in separate files. This gives the analyst the possibility to optimize the framework depending on individual needs and thereby determine the verbosity of the results.

IV.   THE CORE FRAMEWORK

In this section, the core of ANANAS is described by outlining the workflow of an analysis run. Figure 2
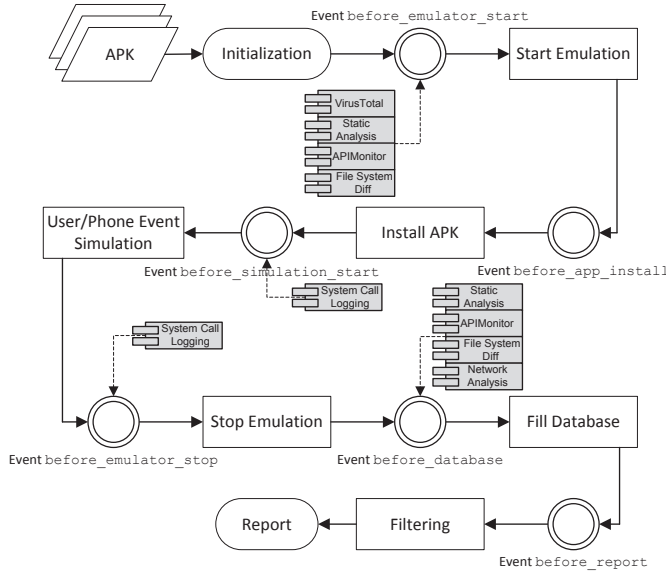
Figure 2.   ANANAS workflow

provides a high level overview of the workflow.

A. Initialization

The first thing ANANAS does at startup is to load its default configuration, which is appropriate for most use cases. If a custom configuration file is provided, these settings overrule the default configuration. Applications may behave differently on different Android versions and also security controls can change from version to version. Therefore, an important configuration setting for the dynamic analysis is the Android version (ANANAS currently supports Android versions 2.3, 4.0, 4.1 and 4.2).

Then, ANANAS creates a clean environment for emulation and simulation purposes. The emulator is started with a temporary image, which is copied from a clean source. This approach allows an easy comparison between the possibly infected and the clean image/filesystem.

As the last initialization step, the framework creates a unique directory for the analysis, where all data (e.g. configurations, results, screenshots, ...) generated during the analysis is stored. All plugins should save their results to this directory.

B. Load Plugins

The next step for the framework is to load the plugins. To ensure that an error in one plugin does not affect the overall analysis, plugins are being disabled for the current analysis run if any kind of error occurs.

Plugins can register on predefined events on which their methods get called by the framework. Each plugin can register methods to any event with a certain priority. Plugins with higher priorities get called first. The framework currently specifies the following events:

- *before_emulator_start,*
- *before_app_install,*
- *before_simulation_start,*
- *before_emulator_stop,*

- *before_database* and
- *before_report.*

These events occur at different steps during the dynamic analysis and therefore offer different perspectives on the analysis process and the analyzed app (have a look at section V for further details).

C. Call Plugins: Part 1

After the loading procedure, the plugins registered to the first event *before_emulator_start* are called. Then, the emulator is started with a temporary image for the configured Android version. By raising the *before_app_install* event, plugins that need to do work before the app is installed, are invoked. Then, the app that should be analyzed is installed on the emulator. After another event named *before_simulation_start* and subsequent calling of registered plugins, the simulation phase is starting.

D. User & Phone Event Simulation

User and phone event simulation is one of the most important components of the ANANAS framework, because malicious behaviour of most Android malware is triggerd by certain user or phone events (as shown in section III and [15]).

To allow a configurable simulation, a scripting language, consisting of consecutive command and no control structures, is parsed by the framework. It makes heavy use of the Android Debug Bridge (ADB) Tool and the telnet interface of the Android emulator for executing its tasks. The following simulation actions are currently available:

- Starting and stopping installed Android applications,
- simulation of incoming and outgoing calls and text messages and changes to GSM state,
- changes to the smartphone's battery state,
- simulation of the emulator's location,
- user input using the *monkey* program shipped with the Android SDK,
- unlocking and locking of the screen and
- execution of ADB commands and shell commands on the emulated system.

Commands within this script are for example *callfrom '+431234567'* to simulate an incoming call from the specified number or *changeLocation 'x-coordinate' 'y-coordinate'* to change the geo location of the emulated smartphone. An example for these commands can be found in figure 3.

With the help of the command *screenshot*, a screenshot of the current emulator's screen can be saved at any time. ANANAS ships with some predefined simulation scripts that should reveal most of the application's behaviour and are adjustable to specific needs.

E. Call Plugins: Part 2

After the simulation script reaches its end, the event *before_emulator_stop* is invoked. This is the last chance for plugins to interact with the emulator, as after calling all plugins that are registered for this event, the emulator is

```
unlockscreen
sleep 3
startservices
startapp
screenshot
monkey 500
smsfrom "+49123456789" "Hi there."
callfrom "+49123456789"
setBatteryCapacity 5
setBatteryPowerState CHARGING
changeLocation "65.966667" "-18.533333"
```

Figure 3.   An example simulation script.

stopped. For example, plugins can pull files needed from the emulator's filesystem for later processing.

If a plugin needs to process its results in any way before they are finally saved to the database, it can register for the next event named *before_database*. It is called directly before the final results of the plugins are written to the database.

Finally, the event named *before_report* is called, giving the plugins a last chance to do any processing or cleanup work. The report is solely generated from the results that are saved in the database, so any processing of results that don't occur on database level won't have any effect on the report.

F. Filtering & Report Generation

Before the report is generated, each plugin has the possibility to apply filters. Filters are used to determine which of the results that are saved in the database are worked into the report. As some plugins might generate a lot of log data, the accuracy of the filters is crucial for the quality of the report and its usefulness to the analyst.

Filtering follows a blacklist approach, whereas certain values can be excluded from the blacklist using a separate whitelist, that is only applied on blacklisted entries. For example, in the case of a system call log, this allows blacklisting each *open* system call on paths starting with */data/* but excluding anything in the */data/data/* directory from blacklisting and thereby including it in the report. Everything that doesn't show up in the blacklist is considered valuable and will be passed to the report generation.

Filters can be based on regular expressions or substring matches. For fine grained filtering, filters can also be written in Python.

As the last step of the analysis process, the ANANAS framework generates the XML report based on the filtered results of the plugins. The report also contains additional information such as the APK's filenames, hashes, timestamps and information about plugins used. A shortened example of such a report can be seen in figure 4.

V. PLUGINS

In order to turn ANANAS into a useful analysis tool, plugins for various analysis methods were developed. They represent well-known techniques for static and dynamic malware analysis. Some of them were solely developed by the ANANAS development team. In order not to reinvent

```
<analysis>
  <filename>test.apk</filename>
  <hashes><hash
      type="md5">d331c96...</hash></hashes>
  <package>ananas.analysis</package>
  <starttime>...</starttime>
  <endtime>...</endtime>
  <plugins>
    <plugin enabled="True" name="lkm"
        state="loaded" />
    ...
  </plugins>
  <errors>...</errors>
  <results>
    <!-- plugin results -->
    <screenshots>...</screenshots>
    <virustotal>...</virustotal>
    <static>...</static>
    <apimonitor>...</apimonitor>
    <filesystemdiff>...</filesystemdiff>
    <syscalltrace>...</syscalltrace>
  </results>
</analysis>
```

Figure 4.   Shortened example report.

the wheel, several plugins were created that leverage existing tools and act as a wrapper. The following plugins were developed for ANANAS:

- Filesystemdiff
- Network analysis
- Systemcall logging

The following plugins wrap existing tools:

- APKIL/APIMonitor
- Static analysis
- VirusTotal query

In the following sections, each plugin is described shortly.

A. File System Diff

The goal of File System Diff is to show changes in the emulator's filesystem which might be caused by the analyzed Android application. It does so by comparing the emulator's filesystem before the emulator boots and after it has been powered off. The plugin registers to the hooks *before_emulator_start* and *before_database*, where it mounts the emulator image into the analysis host's filesystem and creates a list of directories and files including their hash values. These lists are then compared to detect created, modified and deleted files. Files that only existed temporarily or were changed during the execution of the emulator cannot be found by using this method.

In ANANAS, we try to avoid the usage of tools that require root privileges. Thus, the ext4 filesystem is used for emulator images so, that they can be mounted with a slightly modified version of *ext4fuse* in userspace as an unprivileged user.

B. Network Analysis

Network traffic that occurs during the analysis is saved in a pcap file. Connections are extracted from the stored network traffic and provided in a readable format for later

use in the report. For simplicity, in case of TCP and UDP traffic a connection is identified as a tuple of two IP address and port combinations. Traffic caused by other protocols is identified by the two IP addresses and the protocol used. In addition, IP packets for each connection are counted to give an overview of the total amount of network traffic.

To get a high-level view, two application level protocols are further examined: DNS and HTTP. Therefore, queries made to DNS servers are extracted and the type and content of the query is shown. Furthermore, HTTP traffic is identified and relevant details such as URLs, header fields and parameters in GET and POST requests are extracted.

C. Systemcall Logging/LKM

To provide a low-level view of the interaction between the analyzed application and the Android system, a loadable kernel module is used to hook several system calls and log their usage. It allows ANANAS to trace all system calls that have been executed on the emulated system during the analysis. This is important to detect inter process communication (IPC), which is heavily used in Android. One example of IPC is the handling of the Short Message Service (SMS). SMS messages received by the system are broadcasted to applications (including the built-in SMS application), which requested to receive this type of IPC message. A trick that is used by malicious applications is to instrument a web browser via IPC to open a certain URL and exfiltrate data. This is why the logged system calls cannot be restricted to the analyzed application. Instead, all system calls on the whole system must be considered.

For each system call, the name and the parameters of the system call and its return value are logged. If an argument to a system call is a pointer to a string, the string is recorded. The plugin includes hooks for mostly filesystem related system calls, such as system calls needed to create, read, write, remove files and change permissions and ownership of files. System calls used for changing user and group IDs are also logged.

Since the logging of system calls for all applications leads to very big log files, a good filtering mechanism must be put in place. The filters were written and improved by comparing the analysis results of several malicious and non-malicious applications. System calls that represent normal behaviour on an Android system are filtered because they do not provide valuable insight for the analyst. Some popular root exploits for the Android system were whitelisted in order to make sure they show up in the report. ANANAS's LKM plugin doesn't try to detect these exploits actively, but by whitelisting the exploits it is ensured that they are not accidentally blacklisted.

D. APKIL/APIMonitor

APIMonitor/APKIL [16] was developed by Kun Yang at the Honeynet Project during the Google Summer of Code 2012 as an improvement for DroidBox [8]. By interposing APIs in the APK file and inserting monitoring code into the APK, API call logs can be retrieved.

The plugin registers to the event *before_emulator_start* and uses APKIL to modify the APK before it is installed to the emulator. It is important to notice that all other plugins, such as the static analysis plugin, which are in some way dependent on the APK file, still use the unmodified version. The modified APK then logs its API calls to logcat, Android's logging system. As the framework redirects the logcat files to the local result directory on the analysis host, the plugin doesn't have to retrieve the logcat files manually. However, it registers to the hook *before_database* to process the logcat files and search for relevant APKIL entries.

E. Static Analysis

When analyzing applications, static analysis can be a very useful tool to gather information about the application. Especially the file *AndroidManifest.xml*, which is part of every APK, is a precious source containing the application's permissions, activities, services, content providers, broadcast receivers as well as other useful information like the package name.

ANANAS' static analysis plugin's capabilities are to analyze the *AndroidManifest.xml* and extract the application's permissions, services, receivers and the package name. As the application's manifest is compressed, apktool is used to convert the manifest to a readable format before extracting the information wanted. Apktool is a tool, which can decode resources within an APK and disassemble the compiled dex files to smali [10].

The static analysis plugin extracts strings from the disassembled dex files. These strings are then scanned for URLs. The knowledge, which URLs an Android application contains and probably connects to, can be a valuable information, since a lot of malicious Android applications try to send personal information from the users' smartphones to external servers or try to contact their C&C servers.

The next step of the static analysis plugin is to check each resource within the APK for its filetype. The purpose of this is to identify binaries or native libraries that ship with the APK to be executed later and might contain exploits. For malware analysts, it might be suspicious if an application contains a file called *image.png*, which is detected as an executable.

To do its work, this plugin registers to the event *before_emulator_start* and starts the static analysis in a background thread in order not to delay the dynamic analysis. The plugin waits for the background thread to finish on the *before_database* event.

Although obfuscation and similar techniques can limit the benefit of static analysis for malware detection, it is still a valuable source of information, especially when it comes to analyzing Android applications. As long as the specification of Android applications doesn't change, the *AndroidManifest.xml* will always contain easily extractable and valuable information about the APK.

F. VirusTotal Query

VirusTotal is a platform that allows the user to submit a file or the hash of a file and receive the results of several antivirus scan engines.

This plugin provides the possibility to automatically include the results of VirusTotal into the final report. It does so by searching for the APK's hash on VirusTotal using the hook *before_emulator_start*. If VirusTotal isn't aware of the APK's hash, the plugin also provides the possibility to upload the APK and fetches the results of the analysis later.

VI. Experimental Evaluation

The evaluation of ANANAS was done based on samples within the *Android Malware Genome Project* (AMGP) [17]. The whole set of 1,260 samples was analyzed using the framework to evaluate the robustness of the system. ANANAS proved to be capable of analyzing this amount of samples without showing any fatal errors that would lead to a crash of the framework or a stop of an analysis run.

For the first experimental evaluation of the framework's detection capabilities, we analyzed a small subset of six randomly chosen samples from within the *AMGP* supplemented by two more recent samples. The resulting reports were manually evaluated. A more extensive evaluation on a larger amount of samples has to be conducted in the future.

In this section, we first present the results of this experimental evaluation. Problems and challenges faced during the evaluation are discussed subsequently. Finally, we draw a comparison between ANANAS and Andrubis [7] regarding the tested subset of samples within the *AMGP*.

A. Observation of Malicious Behaviour

The *AMGP* categorizes the samples by their payload functionality. The categories are *privilege escalation*, *remote control*, *financial charges*, and *personal information stealing* [15]. Apart from *remote control*, two samples of each category were analyzed with ANANAS. The *remote control* category was excluded due to the absence of reachable command & control servers for the samples within the *AMGP* at the time of doing the evaluation. Additionally, two more recent samples were analyzed (*Android/SystemSecurity.A* and *Trojan:Android/Maistealer.A*).

Each sample was analyzed with ANANAS using an Android 4.1 based emulation environment. All plugins were enabled and an extensive user and phone event simulation script was used. The results of the experimental evaluation are shown in table I. The detection of malicious behaviour results in a *Yes* for the respective category. If parts of the malicious behaviour or at least an indicator for such can be found in the report, this is described with *Partly*. A detection failure results in a *No*. A hyphen (-) symbolizes that the sample contains no malicious payload for the respective category.

As an example for the evaluation process, we discuss the analysis and results of two samples (DogWars and GGTracker) in greater detail below.

1) Android.Dogowar: Android.Dogowar (DogWars) is a trojan horse, which sends text messages to all contacts saved on the device. It also sends text messages to a hardcoded number. This malware is a repackaged version of a legit game called *Dog Wars*. [18]

The AMGP categorizes this sample's malicious payload into the category Financial Charges (subcategory SMS). This means, that the sample charges the user financially by sending text messages. ANANAS can detect this malicious behaviour. More precisely, the APKIL plugin reveals the actions of this trojan. The first action of DogWars is to query the contacts application that ships with Android for contacts stored on the device. Each contact for which a phone number exists will be forwarded a text message with the content *I take pleasure in hurting small animals, just thought you should know that*. Also, a text message to the number 73822 with the content *text* is sent. All this action occurs without the user's knowledge, leaving the user with the bill for the sent SMS.

The network analysis plugin shows that `POST` and `CONNECT` HTTP requests to the URL `http://kagegames.com/dw/process_cmds3.php` are sent. The network analysis plugin also shows that no personal data left the phone through these connections. As this malware is a repackaged version of a legit game, these connections might be part of the original game's behaviour.

2) GGTracker: The analyzed version of GGTracker hides inside a battery management application and its main functionality is to sign-up the phone to premium SMS subscription services [19]. The AMGP classifies the malicious payload of GGTracker into Financial Charges (subcategory SMS) and Personal Information Stealing (subcategories SMS and Phone Number).

ANANAS is able to detect the information stealing part of the payload. It is not possible for ANANAS to detect the financial charges that occur via premium SMS subscription services, as the server used for subscribing to these services wasn't reachable anymore at the time of writing this paper.

A first analysis of the sample showed that DNS queries for `ggtrack.org` and `www.amaz0n-cloud.com` failed. For the purpose of generating better results, the emulator's hosts file was modified to redirect these domains to a local server, where netcat was used as a sink for incoming connections to port 80. This enabled us to record the HTTP requests sent by GGTracker and therefore get better analysis results.

The static analysis plugin showed the two URLs, `http://ggtrack.org/SM1c?device_id=` and `http://www.amaz0n-cloud.com/droid/droid.php`, the sample connected to. The APKIL plugin revealed that the sample requested access to the SMS database several times but couldn't show that the SMS would leave the phone. The File

Application	Privilege Escalation	Financial Charges			Personal Information Stealing		
		Phone Call	SMS	Block SMS	SMS	Phone Number	User Account
GGTracker	-	-	No/No	No/No	Yes/No	Yes/Yes	-
DogWars	-	-	Yes/Yes	-	-	-	-
Gingermaster	Partly/Partly	-	-	-	-	Yes/No	-
AnserverBot	-	-	-	-	No/No	-	-
SndApps	-	-	-	-	-	-	No/No
DroidKungFu	Partly/No	-	-	-	-	Yes/No	-
Pjapps	-	-	No/No	No/No	-	Yes/Yes	-
Trojan:Android/SystemSecurity.A*	-	-	No/No	No/No	-	Yes/Yes	-
Trojan:Android/Maistealer.A*	-	-	-	-	No/No	Partly/Partly	-

Table I

OBSERVATION OF MALICIOUS BEHAVIOUR POSSIBLE IN ANANAS / IN ANDRUBIS. SAMPLES MARKED WITH * ARE NOT PART OF THE MALWARE GENOME PROJECT.

System Diff plugin showed that the following three files were added by the sample in its directory in `/data/data/t4t.power.management/`:

- `shared_prefs/carrier.xml`,
- `shared_prefs/t4t.power.management _preferences.xml` and
- `shared_prefs/phone.xml`.

The names of the files `carrier.xml` and `phone.xml` alone might raise concerns about sensitive data, which could be stored in these files.

The most beneficial plugin for the analysis of this sample is the network analysis. It clearly shows that the emulator's phone number is sent to `ggtrack.org/ SM1c` via a `GET` request multiple times. It also shows that the simulated incoming SMS that have been received are forwarded to `www.amaz0n-cloud.com/droid/ droid.php`. This is done via a `POST` request, which contains the phone number of the receiving as well as the phone number of the sending device, the carrier, the content of the message, and the version of Android that is currently running on the receiving device.

B. Challenges During the Evaluation

The biggest challenge during the evaluation was that most command & control servers for older malware samples were not reachable anymore. This was especially problematic with personal information stealing samples. We can often see that the data is accessed by the analyzed sample, but because the command & control server is not reachable the data is never sent. By using DNS redirection and a netcat listener as a sink for HTTP requests from GGTracker, we were at least able to receive the requests of this malware sample.

The unreachability of C&C servers is also a problem with malware that fetches and loads code at runtime and carries no malicious payload itself. Some malware acts only as an installer for other Android applications. The actual malicious payload is contained in the application to be installed. While an application can dynamically load code, it cannot install other applications without prompting the user for permission. ANANAS currently does not simulate a user approving an application installation and therefore the actual malicious payload may never be installed and executed.

Another challenge for the dynamic analysis is to trigger every malicious behaviour of an application. While ANANAS ships with some predefined simulation scripts which reveal most of the malicious activity, there is still behaviour that is not triggered (e.g. blocking of an incoming text message from a phone number that is specified in the application's code).

The usage of APIMonitor can also be a problem during the analysis. The AnserverBot malware family for example checks itself for integrity before the malicious payload is executed, as described by Zhou and Jiang in [20]. The sample is not activated at all if APIMonitor is used. With a disabled APIMonitor plugin some activity can be observed, although the described malicious behaviour is still not triggered.

C. Comparison of ANANAS to Andrubis

To compare the quality of the reports generated by ANANAS to the reports that have been generated by Andrubis, the test samples have been uploaded to Andrubis. It is worth noting, that each sample has already been analyzed by Andrubis prior to this evaluation. If C&C servers were still reachable at that time, results may be different. The reports of Andrubis and ANANAS were compared manually. As ANANAS focuses on dynamic analysis, only the dynamic parts of the reports are compared against each other. However, it is obvious that the static analysis of Andrubis is superior to the static analysis plugin of ANANAS.

Unfortunately, no detailed description of the inner workings of the Andrubis system could be found. Therefore, it is not always clear why and how some behaviour can or can not be detected with Andrubis. We suppose that the differences in the detection of malicious behaviour between ANANAS and Andrubis mostly result from the use of different analysis techniques. Also, differences in the implemented detection evasion techniques or user and phone event simulation may be responsible for the different results.

Table I shows the results of the comparative tests with Andrubis. It is clear that both ANANAS and Andrubis are not able to detect every malicious behaviour with their dynamic analysis modules. The challenges for the dynamic

analysis that lead to detection failures might be similar for both frameworks.

VII. CONCLUSION & FUTURE WORK

In this paper, ANANAS, an extendable framework for the analysis of Android applications, as well as its plugins have been introduced. The main contributions of this paper are:

- ANANAS, which is a ready to use framework for dynamic and static analysis of Android applications. It is highly configurable and independent of the emulated Android version.
- An abstraction layer for simple user interaction and phone event simulation, which is easy to adjust.
- A clean and defined interface for plugin developers to facilitate the development of new plugins, regardless of whether they are intended for static or dynamic analysis.
- Six plugins for static and dynamic analysis of Android applications, whereby three of them were developed solely for ANANAS and three are wrappers around existing tools.

The evaluation showed that it is important to simulate different phone events and user input during the dynamic analysis, since otherwise certain behaviour of the application cannot be triggered.

It also showed that none of the implemented analysis methods was capable of detecting every malicious behaviour. Therefore, it is not sufficient to build a tool providing a single analysis method. It is important to have an expandable framework in which new or improved analysis methods can be integrated as plugins. ANANAS aims to be such a framework.

By analyzing all samples within the Android Malware Genome Project, ANANAS proved to be a robust framework. The developed plugins also proved to be effective by recognizing most of the application's malicious behaviour.

Possible improvements of the ANANAS framework, which are left for future work, are to evade detection of the analysis environment by malware or the improvement of plugins. To evade the detection of the Android emulator, the IMSI and IMEI of the emulator could be altered, but effective detection evasion still needs more research. It is also a possibility to create new plugins for ANANAS or adopt existing tools for the analysis of Android applications as they are released.

Another topic, which needs more attention, is the development of effective filters for the plugins. The problem that has to be solved is to make sure that all relevant output of a plugin that can be mapped to malicious behaviour is shown in the report, while also making sure that non-relevant output does not show up. As some plugins, e.g. the *LKM* plugin, are very verbose, it is tempting to blacklist most of the output to make the final report smaller. The possibility to accidently blacklist relevant output is thereby tremendous.

One problem that occurs during the analysis, especially of older malware samples, is that the C&C servers are not reachable anymore. A simple simulation of such C&C servers could improve the generated results a lot.

Since the Android system is still quite young compared to other popular operating systems, a lot of work has to be done to address security concerns. The analysis of potentially harmful applications is one important aspect in this field to which this paper contributes.

REFERENCES

[1] Gartner, Inc. Gartner says worldwide sales of mobile phones declined 3 percent in third quarter of 2012; smartphone sales increased 47 percent. [Online]. Available: http://www.gartner.com/newsroom/id/2237315

[2] IDC Corporate. Android marks fourth anniversary since launch with 75.0market share in third quarter, according to idc. [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=prUS23771812#.UWAPDZNkOSo

[3] Dashboards — Android Developers. [Online]. Available: http://developer.android.com/about/dashboards/index.html

[4] McAfee Labs, "McAfee Threats Report: Fourth Quarter 2012," Tech. Rep. [Online]. Available: http://www.mcafee.com/sg/resources/reports/rp-quarterly-threat-q4-2012.pdf

[5] Android trojan found in targeted attack. [Online]. Available: https://www.securelist.com/en/blog/208194186/Android_Trojan_Found_in_Targeted_Attack

[6] Anubis: Analyzing unknown binaries. [Online]. Available: http://anubis.iseclab.org/

[7] M. Lindorfer. Andrubis: A tool for analyzing unknown android applications. [Online]. Available: http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzing-unknown-android-applications-2/

[8] Droidbox. [Online]. Available: http://code.google.com/p/droidbox/

[9] 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*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924971

[10] Apktool. [Online]. Available: http://code.google.com/p/android-apktool/

[11] Androguard. [Online]. Available: http://code.google.com/p/androguard/

[12] 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*, ser. SAC '13. New York, NY, USA: ACM, 2013, pp. 1808–1815. [Online]. Available: http://doi.acm.org/10.1145/2480362.2480701

[13] 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 conference on Security symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 29–29. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362822

[14] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: automatic security analysis of smartphone applications," in *Proceedings of the third ACM conference on Data and application security and privacy*, ser. CODASPY '13. New York, NY, USA: ACM, 2013, pp. 209–220. [Online]. Available: http://doi.acm.org/10.1145/2435349.2435379

[15] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (SP), 2012 IEEE Symposium on*, 2012, pp. 95–109.

[16] Apkil/apimonitor. [Online]. Available: http://code.google.com/p/droidbox/wiki/APIMonitor

[17] Android malware genome project. [Online]. Available: http://www.malgenomeproject.org/

[18] B. Cai. Android.dogowar technical details. [Online]. Available: http://www.symantec.com/security_response/writeup.jsp?docid=2011-081510-4323-99&tabid=2

[19] T. Strazzere, "GGTracker Technical Tear Down," Tech. Rep. [Online]. Available: https://blog.lookout.com/wp-content/uploads/2011/06/GGTracker-Teardown_Lookout-Mobile-Security.pdf

[20] X. J. Yajin Zhou, "An analysis of the anserverbot trojan," Tech. Rep., 9 2011. [Online]. Available: http://www.csc.ncsu.edu/faculty/jiang/pubs/AnserverBot_Analysis.pdf