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 D7.2: Intermediate Report on Cyberattacks on Ultra-portable Devices

Abstract: This deliverable reports our preliminary research results in the area of cyberattack detection/mitigation on ultra-portable devices. We first present a review of the State-of-the-Art in attacks and defenses in smart devices and continue with reporting on the research we have been conducting.

Contractual Date of Delivery	August 2012
Actual Date of Delivery	September 2012
Deliverable Dissemination Level	Public
Editor	Sotiris Ioannidis
Contributors	All SysSec Partners
Quality Assurance	Magnus Almgren, Valentin Tudor, Zhang Fou

[†] 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
IPP-BAS	Principal Contractor	Bulgaria
Technical University of Vienna	Principal Contractor	Austria
Chalmers University	Principal Contractor	Sweden
TUBITAK-UEKAE	Principal Contractor	Turkey

Contents

List of Figures

1 Introduction

The purpose of this deliverable is twofold: (i) we want to present the work that has been conducted by the research community in general on attacks and defenses on ultra-portable devices, and (ii) we want to describe the efforts that the partners of the consortium have been doing in the area of cyberattacks on ultra-portable devices.

In the following two chapters, we discuss and analyze the work of other research groups, that are conducting State-of-the-Art research in the area. We then present three systems, that we have been working on for the duration of the project so far. Those systems are BinderProfiler, Paranoid Android and Andrubis. Lastly, we close with some final thoughts in Chapter 7.

2 Attacks

2.1 Privilege Escalation

Davi et al. [18] described a conceptual weakness in the Android permission model which allowed applications to access resources without the necessary permissions.

In the Android permission model applications request permissions for resources at installation time and the user is prompted to either grant these permissions or deny them. In the Android platform each application runs in a different virtual machine to ensure that malicious applications stay isolated from the rest of the system, a mechanism called sandboxing. Even though application run in isolated environments they offer a mechanism for inter-process communication. Process are allowed to send messages to other processes and get back results. This feature allows applications to better integrate with the platform and each other. For example an application that wishes to send the user to visit its homepage will send a message to the browser application containing the URL of the homepage. The browser will send the user to the homepage and when the user closes the browser they will be directed back to the original application.

Applications can limit their exposure to other applications. An application can request from the system to only receive messages from applications holding a relevant permission P_1. It is paramount for the security of the system that applications with access to sensitive resources e.g. address book, location, limit their exposure to other applications and they do not just serve every message they receive. Unfortunately, limiting the exposure of this sensitive interface is left to the developer of the application and it is not enforced by the platform.

The attack called *privilege escalation* is based on the fact that a malicious (unprivileged) application can take advantage of benign (privileged) applications to perform malicious actions. Assuring a privileged applica-

tion, e.g. the browser, has been granted access to the Internet. It is perfectly normal for the browser to access the Internet, but the application has not limited the access to its interfaces. Thus any other application can request the browser to visit a URL or download a file. The user has also installed in the system a malicious application but without granting any permission to it. Even though it seems that the malicious application cannot perform any damaging actions, it is in fact able to connect to a malicious server and download content by exploiting the lack of filtering of the browser.

2.2 SOUNDCOMBER

SOUNDCOMBER [42] is a trojan designed to eavesdrop the user and extract valuable information while remaining stealthy. SOUNDCOMBER requests only permission to access the microphone, a seemingly benign permission. When the user is making a call the trojan records the call and performs signal processing to extract valuable information. The prototype is designed to steal the user's credit-card number. Filtering the information and extracting only useful pieces of information instead of transmitting the whole conversation increases the stealthiness of the malware, since it reduces bandwidth usage.

SOUNDCOMBER avoids to ask permission to access the Internet. Asking for both Internet and microphone permission could have made the user suspicious. Thus a second application is needed, with access to the Internet, to transmit the data to the malicious server. SOUNDCOMBER can perform a *privilege escalation* attack or use a colluding application. In the first case the malware can ask the browser to visit a specifically crafted URL e.g `http://target?number=N` with N the credit card number, thus informing the server of the credit card number. In case of a colluding application the colluding application can open a socket to the server and transfer the data. In this scenario, SOUNDCOMBER can use covert channels to transfer the data to the second application rendering it even more stealthy. Some of the proposed channels are the vibration setting, the volume setting, the screen and file locks.

SOUNDCOMBER uses a database of known *profiles* to recognise conversation segments of interests and then uses speech processing to exfiltrate valuable data. Since, the data is transmitted by a second application the trojan requires only permission to access the microphone.

2.3 iClearshot: Automated shoulder surfing on touchscreens

Portable devices increase the risk of shoulder surfing because they are used in public places (more often than regular personal computers). Touchscreen mobile devices, which are the majority (e.g., iPhone), are particularly threatened by shoulder surfing attacks, because they directly expose also the soft keyboard, making it feasible for an attacker to spy on them. Attackers could indeed steal sensitive information by simply following the victim and observe his or her portable device.

In [37, 36] the authors study and motivate the feasibility of this type of attack, which is easier for human attackers on touchscreens than on regular mobile phones. In addition to human attacks, which outcome is precise yet tedious to gather, they propose a fully-automatic shoulder surfing attack against modern touchscreen keyboards, which requires no prior information (e.g., training) on the specific device, with the exception of the keyboard layout. Other work [41] investigated this threat concurrently to this research and obtained equivalent performance and precision, although they require a training phase, which may not always be acceptable. Both of these efforts focus on touchscreen keyboards that display magnified keys in predictable positions.

In the adversarial model proposed in [37, 36] the attacker points a camera (e.g., portable camera, or surveillance camera) toward the target touchscreen while the victim enters a text. No visibility of typed text is required. The attack works even when fingers partially cover the magnified keys, as it typically happens while typing. Key magnification is often enabled by default in popular touchscreen phones and sometimes cannot be deactivated (except in some older versions of Android OS).

The attack first detects the touchscreen image by leveraging a template of the target keyboard. When a match is found in the current frame, it rectifies and crops the screen area. Then, it isolates the high-contrast areas of the rectified image that are different from the template and previous frames. Then, it filters out noisy areas and identifies the best-matching key. The authors evaluated this attack against the Apple iPhone—although it can work with other layouts and different devices—and show that it recognizes up to 97.07% (91.03% on average) of the keystrokes, with only 1.15% of errors, at 37 to 51 keystrokes per minute: About eight times faster than a human analyzing a recorded video.

3 Defenses

3.1 TAINTDROID

TAINTDROID [22] is an extension to the Android platform which protects against privacy leaks using information flow. TAINTDROID monitors the flow of sensitive information inside the Android and the 3rd party, when sensitive information leaves the system TAINTDROID logs the event, including the application that was responsible for the transmission, the destination server and the content of the message.

The first step to perform information flow, and more specifically dynamic taint analysis, is to identify the *sources* of sensitive information in the system. Fortunately Android's documentation makes it straightforward to identify the method calls that return sensitive data, e.g. calls to the location manager will return the current coordinates of the device which constitute private information. By identifying the sources of sensitive information you can inject 'meta' information, named taint, to each piece of data to know that it is private.

The second step to dynamic taint analysis is to propagate the taint as the private information is processed during program execution. TAINTDROID offers taint propagation in different granularities. During code execution inside an application TAINTDROID taints sensitive information in the variable-level. Thus the variable holding the coordinates from the GPS is marked as tainted, when this variable is used as an operand in an operation the variable that will hold the result of the operation will also be marked as tainted. Moreover, TAINTDROID handles gracefully the cases where sensitive data is written to files and sensitive data is sent to other processes using IPC messages. Conceptually, TAINTDROID can track sensitive information and its by-products, through intra and inter process propagation.

As a final step, TAINTDROID needs to identify attempts of sensitive data leaving the device. TAINTDROID taps into the Java library at the point where the native socket library is called.

To evaluate TAINTDROID, the authors studied 30 popular applications, from which 20 were found to potentially compromise user's privacy. A performance evaluation showed an 27% overhead over the vanilla Android platform.

3.2 PIOS

PIOS [21] is a tool for automatically detecting applications that leak users data without their consent in the iOS platform. PIOS finds privacy leaks using data flow analysis, which means that it identifies flows from functions with access to sensitive data (*sources*) to functions that can send data to third parties (*sinks*). Not all flows from sources to sinks are privacy leaks, if the user has given his consent then it is a legitimate action. To account for this case PIOS assumes that if the application interact with user between the point where private data has been accessed and the potential transmission point then the user has been warned and he has given his consent.

PIOS performs its analysis in three steps. As a first step, PIOS constructs the interprocedural control flow graph (CFG) from the binary of the application. Constructing the CFG can be challenging for the following reasons. i) iOS applications are written primarily in *Objective-C*, an object oriented language. Thus finding the class and method invocations require knowledge on the class hierarchy to account for inheritance. ii) All messages from one object to another pass from a central dispatch function that can obscure the destination of the message. As a second step, PIOS uses standard reachability algorithms to identify, in the CFG, all paths from sources of sensitive data to sinks without user interaction in the path. Finally, to confirm that a path from a source to a sink is actually being used by the application, PIOS performs an additional data flow analysis to locate if the sensitive data from the source reaches one or more of the parameters of the sink.

The authors analyzed 1407 applications with PIOS from the official as well as unofficial repositories, and concluded that applications, even from the unofficial repositories, respect user privacy. One exception being the device ID number which is leaked by almost 15% of the applications.

3.3 *Saint*

Saint [38], *Secure Application INTeraction* framework, is an extension to the traditional Android permission framework. *Saint* gives applications more fine-grained control over their interfaces. More specifically it aims to solve

the following weaknesses of the current Android permissions system: (i) applications have limited control over who can access their interfaces. (ii) applications have limited control over how their interfaces are used. (iii) applications have limited control, at runtime, to dynamically change who can use their interfaces. *Saint* defines two new types of policies to work in conjunction with the existing Android permission system.

Install time policies regulate granting of application defined permissions. When an application requests a permission during its installation *Saint* ensures that the application satisfies the policy for granting that permission, otherwise the installation is blocked. The policy for a permission is defined by the application that declares the permission. Conceptually, an application can declare a set of permissions that other applications must hold to access its interfaces and also the policy under which these permissions are granted.

Run time policies mediate the interactions between the applications realtime. *Saint* policy enforcement intercepts messages in the IPC middleware of the Android platform. When the sender initiates an IPC communication *Saint* ensures that both the caller's and the callee's security requirements are met.

The authors have integrated *Saint* in Android 1.5. One of the required changes for the integration was a new installer that can handle the new policies. Also, a mediator for enforcing the policy when appropriate was necessary. *Saint* enforces its policies in the following cases: 1. Starting Activities. 2. Binding Services. 3. Receiving broadcast Intents. 4. Accessing Content Providers.

3.4 *Aurasium*

Aurasium [47] is a novel technique to enhance security in the Android ecosystem. One of *Aurasium*'s strengths is that it does not require any changes in the underlying operating system. *Aurasium* works by automatically rewriting application to contain policy code that restrict each action. The main observation is that all interactions between the application and the OS happen through well-defined communication channels. *Aurasium* rewrites the application to interpose policy code in all such channels. Some of the policies that *Aurasium* can check are the following: 1. By interposing `ioctl()` calls the system is capable of mediating all IPC calls, thus blocking the application from calling premium numbers. 2. By interposing `getaddrinfo()` and `connect()` *Aurasium* can effectively block the application from ever accessing the Internet. By monitoring the return data of the `ioctl()` it is possible to locate applications trying to access the IMEI or IMSI of the device. Upon finding a possible breach *Aurasium* prompts the user about the desired action and stores the decision for the future.

Aurasium poses a small overhead in the application, for accessing device information for instance the overhead is 35%. By rewriting the application *Aurasium* effectively creates a sandbox around the application without needing to modify the OS. The rewriting increases the application by around 50 KB.

3.5 *QUIRE*

QUIRE [19] is designed to solve the following issues: i) most applications can access the Internet, making it difficult for a remote server to trust the source of the information. ii) malicious applications can take advantage of other, benign, applications with more privileges to perform malicious actions. *QUIRE* tries to defend against these attacks by allowing applications to reason about the call chain and data provenance of requests, coming either from another application using IPC or to a remote server though RPC.

QUIRE taps into the Android binder to allow endpoints that protect sensitive data to reason about the source of the request by looking at the call chain of the applications that causes this request. *QUIRE* modifies the Android binder to automatically build and propagate information about the source of each request.

3.6 *Crowdroid*

Crowdroid [15] is a system that takes advantage of crowd sourcing to obtain traces of application executions. Based on a sufficient number of traces a clear profile of the application's behaviour can be created and can be used to differentiate between benign and malicious applications. *Crowdroid* consists of three components:

- Users, the crowd, install a lightweight application that monitors the behaviour of the application in the smartphone and reports the data back to a central server. This application collects the system calls produced by each application using Strace.

- The traces of each user are sent to a central server where they are parsed and organized by application. Thus grouping the observed behaviour of an application by different users.

- Using clustering algorithms the data sets are analyzed and applications are grouped as malicious or benign.

3.7 Cells

Smartphones are increasingly ubiquitous, and many users carry multiple phones to accommodate work, personal, and geographic mobility needs. The Cells project revolves around a virtualization architecture for enabling multiple virtual smartphones to run simultaneously on the same physical cellphone in an isolated, secure manner. Cells introduces a usage model of having one foreground virtual phone and multiple background virtual phones. This model enables a new device namespace mechanism and novel device proxies that integrate with lightweight operating system virtualization to multiplex phone hardware across multiple virtual phones while providing native hardware device performance.

The idea is that you can run unsafe application in one virtual phone without jeopardizing the security of the other virtual phones. This gives us something equivalent to 'red' zones and 'green zones'.

Cells virtual phone features include accelerated 3D graphics, complete power management features, and full telephony functionality with separately assignable telephone numbers and caller ID support. The Cells prototype supports multiple Android virtual phones on the same phone. Cells imposes modest runtime and memory overhead, works seamlessly across multiple hardware devices including Google Nexus 1 and Nexus S phones, and transparently runs Android applications at native speed without any modifications. The Cells research was commercialised in a start-up company called Cellrox[1].

3.8 *MockDroid*

MockDroid [12] is a modified version of Android permitting users to fake granting a resource to an application. The application thinks that it is given access to a particular resource, but the resource appears unavailable when the application tries to access it.

MockDroid gives the user the ability to install software without having to grant access to all the requested resources. Moreover, based on what information the user is feeling comfortable to share and the functionality the users uses to use from the application, the user can fine-graine the permissions wishing to actually grant and those wishing to 'mock'. Some examples of the mocking functionality are the following:

Location The user can cause the GPS receiver to always return *No location fix* or fixed coordinates.

Internet The application is allowed to open socket, but they will always time-out.

[1] http://www.cellrox.com/

Calendar/Contact The application may try to read the address book, but no information will be returned, and writes will not update the address book.

Device ID The application can access a mock ID instead of the real.

Broadcast Intents Applications are tricked to believe that they can send and receive broadcast intents. But in reality, no generating intents will reach other application and no intent will ever be received by this application.

3.9 *XManDroid*

XManDroid [14] is a solution to mitigate the *privilege escalation* attacks in Android. The authors consider two attack vectors. The first is the *confused deputy attack* where a malicious application with limited privileges uses a benign application to perform malicious actions. The second attack vector is *colluding applications* where two or more malicious application, each with limited permissions, collaborate to perform malicious actions.

The user defines policies that limit the communication between applications with dangerous permission combinations. An example of a policy would be, 'Applications with can read the user contact database must not communicate with an application that has network access'. At run time a reference monitor examines each ICC (*inter-component communication*) to ensure that it follows the policy.

The system caches known decisions on whether communications should be granted or denied to improve performance. Evaluation with real applications show that the overhead is below humans perception.

3.10 *Kirin*

Kirin [25] is a security framework for Android which allows to reason about the security state of the whole device. *Kirin* allows the user of the device a set of policies, called invariants, that must hold in order to install applications. Some examples of *Kirin* policies are the following: 1. An application must have an explicit permission to make an outgoing call. 2. An application holding a dangerous permission must have no unprotected components.

Kirin extracts the policy of newly downloaded applications and turns them into Prolog statements. It then merges the new statements with the existing policy knowledge. The result of the merge represents the state of the policy after the application installs. On that knowledge *Kirin* evaluates the set of invariants to ensure that they still hold. If the invariants hold then the installation is allowed to proceed, otherwise the installation stops.

4 BinderProfiler

4.1 Introduction

Android malware has received a lot of attention. For the work presented here, we used the Android Malware Genome Project [51], which we consider as the State-of-the-Art malware sample. Furthermore, there are many proposals for identifying malware in mobile devices. `Ded` [24] decompiles Dalvik programs to Java and use the many available Java tools for performing static analysis to the recovered code. They study more than 21 million lines of source code and they identify potential misuses by legitimate applications. A similar effort has been done in PiOS for iOS [21]. Source analysis is a well studied technique and has many advantages. However, many times, it is hard to analyze the code, which may be either obfuscated or native in a proprietary library. We believe that source analysis is a microscopic technique for identifying malicious activity, where our approach is macroscopic, since we analyze the high-level actions of an applications. We argue that there are many cases where hiding information at the microscopic level is feasible (obfuscating the code), but hard at the macroscopic level, where concrete actions must take place.

Tainting and dynamic analysis have been also explored for tracking information flow and the behaviour of applications in the mobile environment [22, 40, 13, 45, 15, 46] and Taintdroid [22] is probably the most complete and mature tainting framework for Android at the time of writing. It provides a system-wide tainting system for Android with realistic overhead. However, it suffers from propagating the tainted information in native code, as well as from well known problems associated with tainting [44].

Finally, there are many research efforts for enhancing and optimizing the permission model of Android, which is fundamental for the security of the platform. Saint [38] enhances Android's permission model with policies, which is more powerful than static permissions enforced at installation

time. Saint policies can assist in trusted communication between applications and components. For a short example, consider an application which is linked with an Ad framework. The application could outsource fake Ad clicks by hijacking the communication between itself and the component offering the Ads functionality. On the other hand, Kirin [25] attempts to resolve potentially dangerous combinations of permissions at install-time and warn the user. The authors also provide the implementation of a service offering application certification based on Kirin. Aurasium [47] enforces policies through user-level sandboxing. The authors automatically repackage Android applications with custom code, which is able to resolve offensive actions similar to the ones we identify here (e.g. calling or texting premium numbers). The great advantage of Aurasium is that it needs no system modifications, since applications are automatically extended to support the framework.

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.

4.1.1 Contributions

Our work makes the following contributions:

1. We present *BinderProfiler*, a novel system that classifies Android applications based solely on observed traffic produced by IPC activity.

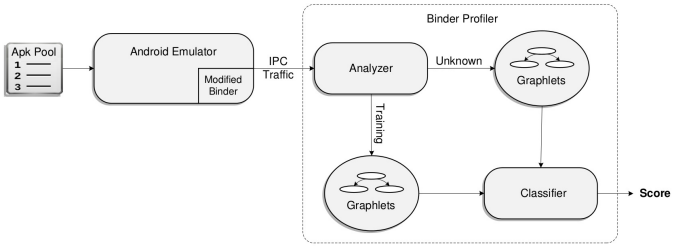

Figure 4.1: Schematic overview of the architecture. We feed a modified Android emulator, running *BinderProfiler*, with applications taken from a pool of malware and legitimate ones. We record all IPC traffic generated. We then analyze all recorded IPC activity and we export graphlets. We form a library of known classified graphlets, which we use to further classify unknown software.

2. We introduce application graphlets for characterizing Android software. Application graphlets are graphs, which encapsulate the high-level behavior of an application.

3. We evaluate our system in terms of accuracy and performance.

4. We design an on-line public service, which utilizes *BinderProfiler* for profiling Android applications before installation.

4.2 Malware Classification

BinderProfiler can be effectively used for producing the profile of an Android application in terms of interprocess communication. From a security perspective, this profiling can be leveraged for distinguishing malware from legitimate software. In this section, we discuss how *BinderProfiler* can be used for malware classification.

4.2.1 Training

BinderProfiler generates application graphlets. A graphlet depicts the esoteric high-level functions, expressed in the form of IPC communication, of a running application. Our intuition suggests that Android malware is characterized by the accumulation of certain subsequent activities (e.g. touching the address book, sending SMS messages or calling premium numbers), which inevitably will trigger IPC traffic. This intuition mainly stems from recent studies about malware behavior in Android [51]. Thus, we seek of a malware classification algorithm based solely on application graphlets or information that can be exported from them.

Depth	Nodes
1	system_server, android.app.IActivityManager
2	com.android.phone, com.android.internal.telephony.ISms, 1066156686
3	system_server, android.app.IActivityManager, com.zft android.provider.Telephony.SMS_RECEIVED, android.permission.BROADCAST_SMS
full	system_server, android.app.IActivityManager, android.intent.action.VIEW, vnd.android-dir/mms-sms, sms_body, *, vnd.android-dir/mms-sms

Table 4.1: Example payload entities used for creating taxonomies. Each taxonomy is characterized by its depth, *d*, describing the maximum amount of intermediate nodes, i.e. payload entities, which we take into account for connecting two individual nodes. A taxonomy with great depth is more descriptive, but less general for further capturing unknown malware. A taxonomy with less depth is more general and it fails to distinguish malware from legitimate applications.

An application graphlet is the IPC profile of an application. It is questionable whether this information is enough for distinguishing malicious from legitimate behavior. The classification algorithm we present in this section is not perfect, but recall that we are tolerant in false positives mainly for two reasons. First, our system can be used in combination or as part of an existing system [1, 5]. Second, *BinderProfiler* acts more as a warning system assisting the security analyst, or even the end-user, in characterizing an application's behavior, rather than as a sensitive detector which blindly decides whether an application is malware or not.

Application graphlets are graphs with nodes representing processes or payload entities. These graphs are formed by connecting individual nodes with each other when there is IPC between them. Comparing two such graphlets, in the general case, is non-trivial. Moreover, we are interested in particular legitimate actions, which if we encounter in groups, we consider them as offensive. For example, getting the address book and sending an SMS produces a graph, which can be hardly considered offensive. However, getting the address book, accessing the network, and sending a series of SMSs in a short time window, produces a graph that is unlikely to be associated with legitimate behavior, or at the very least should raise concerns about the goals of the application. Thus, we need to use partial information extracted from graphlets for deciding about the probability of a given application encapsulating malicious features. Ideally we want to accomplish that with minimum overhead. Although, our system operates in off-line mode, we envision a service that will test multiple Android applications in paral-

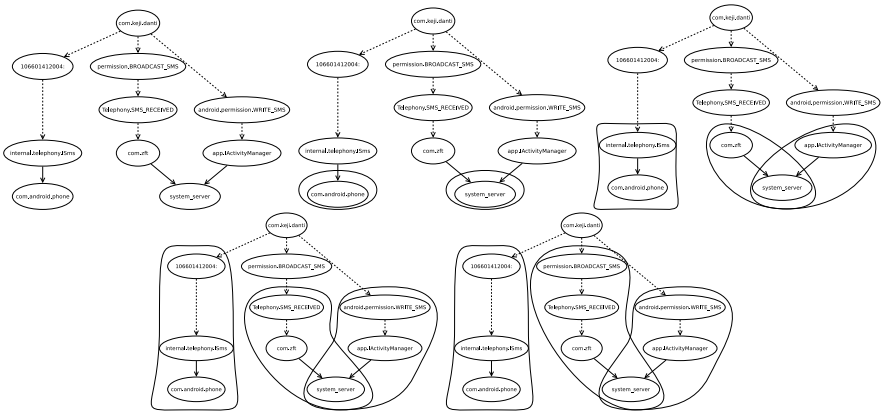

Figure 4.2: Taxonomies expressing paths of different level. We begin with the original graphlet (top and left). From left to right, the second graphlet depicts a taxonomy with depth 1. Only processes that receive a message are taken into account. The third graphlet depicts a taxonomy with depth 2, where processes that receive a message, as well as the first payload entity (usually the class name, which receives the message), are taken into account. In the same fashion, graphlets in the second row, from left to right, represent paths of depth 3 and 4, respectively.

lel, and thus incur low computation overhead. This also makes our system suitable for inline run-time operation.

To summarize, we leverage *BinderProfiler* for malware classification in the following way. First, we produce the application graphlets of a series of malicious applications and legitimate ones. We use the Android Malware Genome Project as a malware source [51]. Collecting legitimate applications is a harder task, since there is no official set composed by provably non-malicious applications. We collect all legitimate applications from Slideme market [8], since Google prevents crawling the official Android market. Slideme is a popular unofficial market for Android software and we expect it incorporates security auditing technologies. Although, it has been demonstrated that such technologies can be bypassed [3], we believe that currently, malware authors have not reached such a level of sophistication. Nevertheless, we select the most popular ones for reducing the probability in getting malware, that has not yet been identified and reported. Each graphlet is produced by running the application for three minutes on a modified Android emulator running *BinderProfiler*. We selected a three-minutes time for analysis period, since, as we discussed in Section 4.3, we observed that the most of IPC activity occurs during the first few minutes of a running process (see Figure 4.6). The specification of the platform is exactly the same as the one analyzed in Section 4.3.

Second, we group all graphlets as benign or malicious. Now, recall how the graphlet is composed. Two connected nodes represent two processes that exchange an IPC message. All IPC messages encapsulate a payload. We parse the payload and export all entities that reflect an API call. Parsing an IPC message is not trivial as 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. These entities are further nodes that interconnect the two initial nodes (see Figure 4.3). We use these entities for producing taxonomies based on the popularity of paths that occur more frequently in the malware set than the legitimate one. Each taxonomy is characterized by its depth, *d*, describing the maximum amount of intermediate nodes, i.e. payload entities, which we take into account for connecting two individual nodes. A taxonomy with great depth is more descriptive, but less general for further capturing unknown malware. A taxonomy with less depth is more general and it fails to distinguish malware from legitimate applications.

An example of how taxonomies are extracted based on different path depth, d, is shown in Figure 4.2, where we depict 4 different taxonomies, each one having depth from 1 to 4. We begin with the original graphlet (top and left). From left to right, the second graphlet depicts a taxonomy with depth 1. Only processes that receive a message are taken into account. The third graphlet depicts a taxonomy with depth 2, where processes that receive a message, as well as the first payload entity (usually the class name, which receives the message), are taken into account. In the same fashion, we depict two cases, where taxonomies represent paths of depth 3 and 4, respectively.

We use taxonomies for extracting weights. More precisely, we observe which graph paths are dominating in the malware set and which are rare in the legitimate set, and we assign each path with a positive weight. We also take into account the reverse behavior, i.e. paths that are dominating in the legitimate set and are rare in the malicious one. We assign each such path with a negative weight. The largest positive weight is assigned to the path that occurred the most times in the malicious set and least times in the legitimate one. The minimum negative weight is assigned to the path that occurred the most times in the legitimate set and least times in the malicious one. We, finally, calculate the *application frequency*, *f*, i.e. the percentage of applications a particular path was observed in. We do this both for the malware and legitimate application set. All collected weights, and application frequencies are used in scoring functions, discussed later. A schematic overview of the architecture is shown in Figure 4.1.

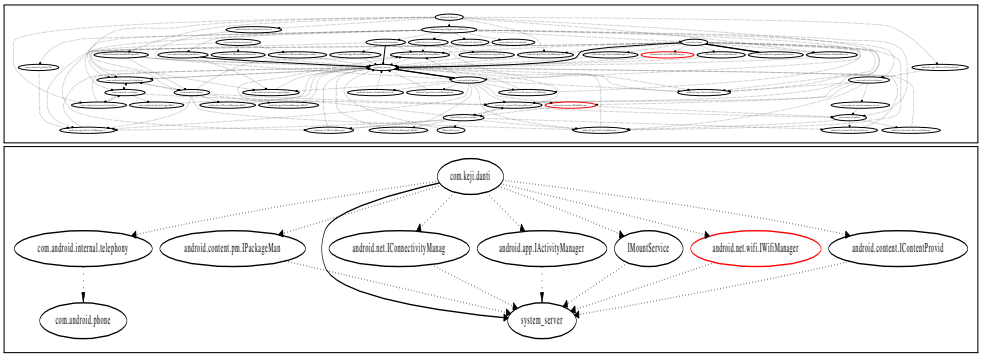

Figure 4.3: An example application graphlet. At the top we depict the graphlet of the malware identified as `com.keji.danti`. At the bottom we zoom in a particular area of the whole graphlet. If two nodes are connected with a solid line then these nodes have exchanged a message we are not able to parse. Otherwise, the message is parsed and decomposed in printable entities. These entities are drawn as nodes between the communicating nodes and interconnected with dashed line. For example, `com.keji.danti` is communicating with `com.android.phone` through a message that encapsulates the `com.android.internal.telephony` service. Nodes, highlighted with red are part of a path that contributes in malware classification.

4.2.2 Classification Algorithm

We use two scoring functions for classifying unknown applications, one application frequency aware, denoted as $F_{af}(n_i) \leftarrow (w_i, f_i, d)$, and one non application frequency aware, denoted as $F_{naf}(n_i) \leftarrow (w_i, d)$. Notice, that each function depends on the collected weights from the training phase and from the taxonomy's depth, d. For a particular depth, the analytic expressions of these scoring functions are the following:

$$F_{af}(n_i) = \sum_i \frac{w_i n_i}{1 - f_i},$$

$$F_{naf}(n_i) = \sum_i w_i n_i.$$

All weights are normalized to 1. The sum is over all paths composing a non classified graphlet. If a path has no weight then we remove it from the sum (we implicitly assume a weight of zero). We use these functions to evaluate graphlets of applications that are not pre-classified. For each graphlet, the final score is an indicator of how similar it is with the graphlet with either a malware one or a legitimate app. The higher the score, the greater the probability of an application containing malicious functionality.

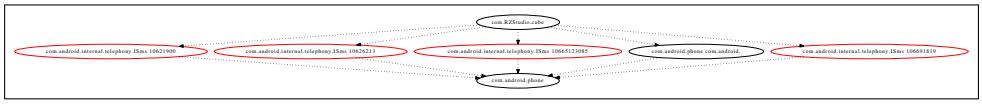

Figure 4.4: The graphlet of RZStudio as generated by *BinderProfiler*. Notice, that the fundamental actions that have been already documented [2] can be easily identified just by looking at the exported graphlet.

4.3 Architecture

In this section we present *BinderProfiler*. We begin with some background information about Android and IPC, and then we give a short overview of the system. We discuss how we implemented it for Android, we present application graphlets, and how we use them as descriptors for identifying malicious patterns in application execution. We close this section by presenting our system's two modes of operation, namely blind and deep.

4.3.1 Background

Instead of using the traditional IPC techniques offered by the Linux Kernel, the Android implemented Binder, based on OpenBinder [7]. 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 on 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. Binder uses these two classes to transmit method calls and parameters from the client process to the server process. More specifically, 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

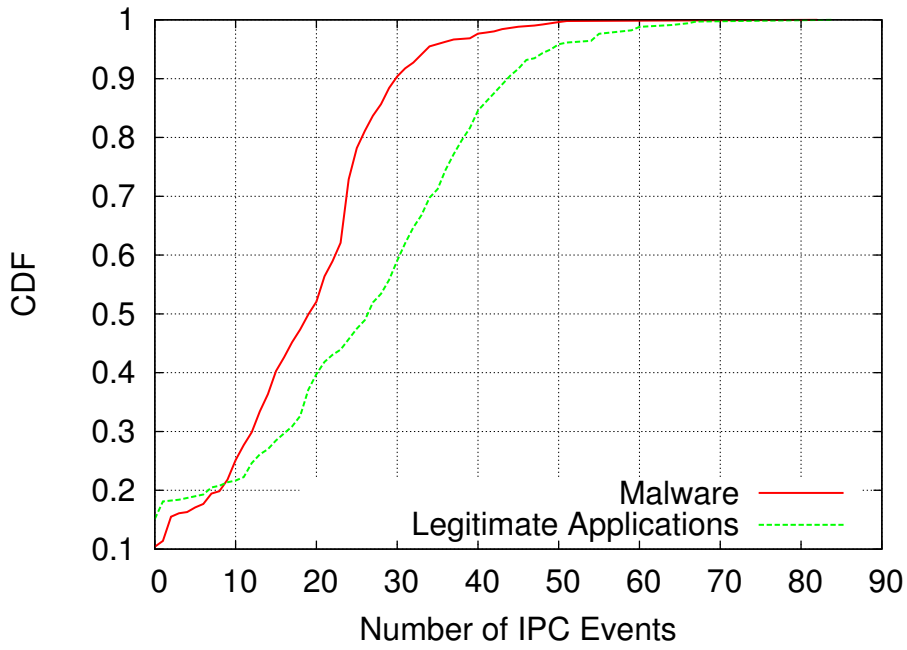

Figure 4.5: IPC activity of each analyzed application, excluding events related to GUI. Observe, that all applications generate tens of IPC events in the running period. Also, legitimate applications are more active. Recall, that all applications run in an emulator, which incorporates Android Monkey [9]. We speculate that some of the malicious applications are repackaged applications [48] with some of the originally functionality turned off.

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

Besides Binder, which is implemented and accessed in Java, there is middleware written in C++ that mediates the interaction between the Java objects and the Binder kernel module. Finally, Binder includes a custom kernel component that passes messages between processes. 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.

4.3.2 *BinderProfiler* Overview

BinderProfiler is based solely on traffic produced by Binder. Applications that need extra resources, for example access to the video or to the SMS function-

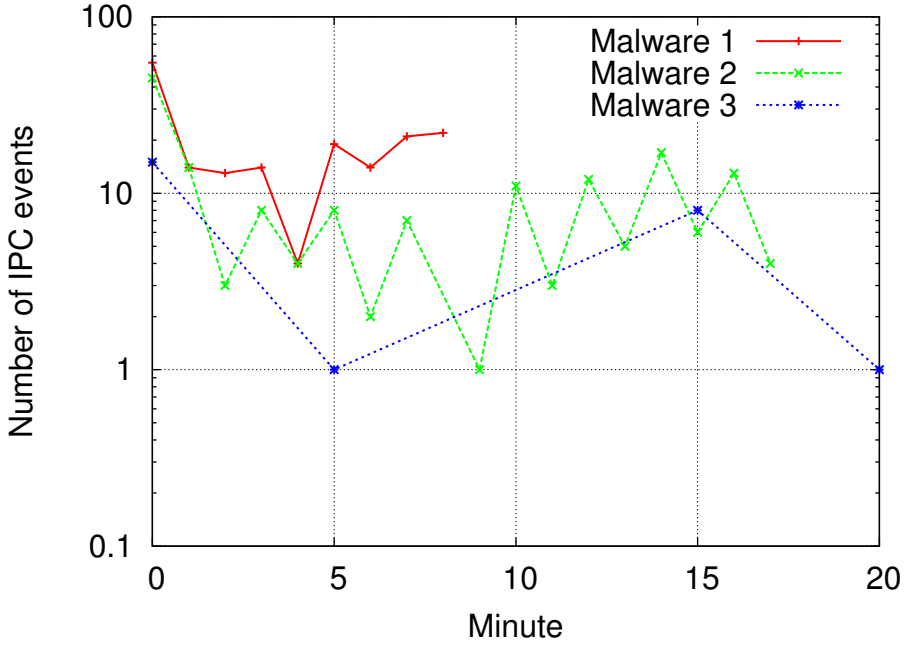

Figure 4.6: IPC activity as recorded for three randomly selected malicious applications for a running period of 30 minutes. We depict IPC messages exchanged per minute. Observe, that most of the IPC activity takes place during the first few minutes. This is the main reason we selected a three-minutes period analysis for our further experiments.

ality of a smartphone, produce messages towards the service that provides the particular functionality. These messages, along with their responses, are delivered through Binder. Our system runs on a modified Android kernel, which passively logs all 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 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 Binder. In the same fashion, an application that issues calls or SMSes towards premium numbers, will acquire the functionality by requesting, through 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.

4.3.3 Implementation

We implemented *BinderProfiler* on the Android emulator. It can be argued that malware can detect whether it is running on the emulator or on an actual device. However, we selected this approach for convenience, since it is easier to build and debug Android on the emulator than on an actual smartphone. Now that our system has been finished and tested, porting *BinderProfiler* on a device is trivial.

We used Android emulator version 18.0 (build_id MASTER-306762) running Android 4.0.3 (API level 15). Unfortunately, the variety of different Android versions, makes developing a security testbed hard. Some of the malware we run were crashing or not running at all. We assume that this is because the particular malware was originally written for a different Android version. One can argue that this hardens the task of malware authors, as well. This is evident, since many families of malware exist, presumably translating the same application for different Android versions [51].

To collect IPC traffic the application must perform some computation. Usually, Android applications incorporate a user interface and almost all actions are delivered through it. We expect that malware may perform malicious activities without the user doing anything special with the application. However, our system can be used more broadly for characterizing applications in general, not simply from a security perspective, and most importantly, our system needs a training phase (see Section 4.2), which incorporates both malware and legitimate applications as samples. We use Android Monkey [9], an automated UI exerciser for triggering functionality in each application. This tool has also been used for similar purposes in other papers [13].

4.3.4 Application Graphlets

The core idea of *BinderProfiler* is *application graphlets*. Initially, we were inspired from BLINC [31], a system for characterizing network hosts by inspecting headers of exchanged network flows. Since Android is heavily based on communicating services over IPC, we seek of a similar framework for characterizing Android applications. An application graphlet is the visual representation of an application interacting with other services over IPC calls. We depict such an example graphlet in Figure 4.3. We present the complete graph as well as a area zoomed in area. This graphlet visualizes the IPC activity of a known malware running on *BinderProfiler* for three minutes.

Two nodes are connected with each other if they exchange a message through Binder. The connection is directed; the direction is from the node sending the message to the node receiving it. Notice, that the graphlet has solid and dashed lines. If two nodes are connected with a solid line then these nodes have exchanged a message we are not able to parse. Otherwise, the message is parsed and decomposed in printable entities. These entities are drawn as nodes between the communicating and interconnected nodes with dashed line. For example, in the zoomed portion of Figure 4.3, `com.keji.danti`, the under inspection malware, is communicating with `com.android.phone` through a message that encapsulates the `com.android.internal.telephony` service. Thus, we draw `com.android.internal.telephony` in between the `com.keji.danti` and `com.android.internal.telephony`, and we connect all nodes with a dashed directed line. Nodes, that are drawn in red color are nodes with high weights in the classification algorithm. We discuss this in more detail in Section 4.2.

We can make many important observations simply by inspecting this example application graphlet. First, the graphlet is of high complexity. Recall that we only run the malware for three minutes. The size of the graph suggests that IPC traffic generated by an application may be sufficient for profiling the application. Second, many actions performed by the application can be easily identified semantically. For example, notice how accessing the telephone functionality of the device is visualized through the interconnected nodes `com.keji.danti` and `com.android.internal.telephony`. In this work, we argue that the application graphlet expresses the higher-level semantic functionality of an application, which is hard to obfuscate or hide. For example, observe in Figure 4.4 a part of the application graphlet exported by `RZStudio`, a malware that has been analyzed by security experts. The results of the analysis [2] can be instantly identified in its application graphlet. Certainly, a malware can be stealthier, i.e. perform all actions slowly, but this reduces the aggressiveness of the malware.

To get a clearer picture for IPC activity, we plot all IPC events recorded for each analyzed application, excluding events related to GUI, in Figure 4.5. Observe, that all applications generate tens of IPC events in the running period. Running period is three minutes. Also, legitimate applications are more active. Recall, that all applications run in an emulator, which incorporates Android Monkey [9]. We speculate that some of the malicious applications are repackaged applications [48] with some of the originally functionality turned off. For selecting an adequate running period, we do the following: We randomly select three malicious applications and analyze them for a running period of 30 minutes. We depict IPC messages exchanged per minute in Figure 4.6. Observe, that most of the IPC activity takes places during the first few minutes. Thus, we select a three-minute analysis period for our further experiments.

4.3.5 Blind and Deep Mode

All communication through Binder involves the exchange of custom messages. Some of them are hard to parse. Even if we are currently able to parse some of them, applications can be modified to exchange encrypted messages, and thus, evade *BinderProfiler*. We were tempted to simply use the end-points of the communication, and not to take into account the payload of messages. We refer to this as *blind mode*. As expected, in blind mode a large training phase needs to be used. This has several disadvantages. First, it is difficult to collect a large amount of applications, compared to other application domains such as passive network monitor. Second, many Android processes delegate other services for performing the actual actions. These delegated services act as proxies (see for example `system_server` in Figure 4.3). This delegation hides the actual end-points. It may be possible to perform time correlation analysis for matching which messages belong to a set of two communicating end-points; we plan to explore this in future work.

Despite, blind mode appearing inefficient for characterizing Android software in practice, it can be effectively used for identifying colluding applications, even if these applications communicate with encrypted messages [42, 27].

5 Paranoid Android

The idea behind *Paranoid Android* [40] is to transfer the malware detection from the actual smartphone to *virtualized replicas* running in server. Such server will not be subject to same physical constrains in terms of processing power and energy consumption as the actual device.

A small lightweight process, the *tracer*, records all the necessary information so that the execution of a process can be replicated in the server. The *tracer* works similar to the popular Linux utility ptrace. To reduce the transmission overhead a set a optimizations is used. 1. Record only system calls that produce non-determinism. System call for open a socket or a file are not recorded. 2. Use a proxy to store inbound traffic temporarily. It is common for a mobile device to read data from the network, normally these data should have be to forwarded to the cloud server. To avoid such overhead the mobile device uses a proxy where inbound traffic is stored, the cloud server can then download the data from that proxy. 3. Perform compression in the generated stream of system calls. To defend against attackers who would try to take control of the device and then erase incriminating logs before the phone sends the trace to the server, the authors employ HMAC signing and use of one time cryptographic keys.

The server can perform a series of advanced malicious detection techniques since it does not impede the users satisfaction. The series contains, but not limited to: 1. Virus scanner performing file scanning using known signatures. 2. Dynamic taint analysis which marks input coming from untrusted sources and tracks its propagation. The system requires from 2-64Kib/s of bandwith and an additionan 15% CPU overhead.

In the context of the SysSec project, VU University Amsterdam recently initiated a major overhaul of the Paranoid Android system. Rather than using a `ptrace`-based tracer as a basis for the record and replay functionality, the new version of Paranoid Android moves all of the instrumentation into the kernel. It employs a technique for efficient recording and replaying pi-

oneered by Columbia University (known as Scribe [32]) to make the entire Paranoid Android system much more performant.

In addition, a taint analysis implementaion for the ARM processor (used by Android Phones) was written from scratch. The reason for the rewrite is that the previous system was buggy and not portable. The new implementation is much leaner and can be easily ported to other architectures.

6 Andrubis

6.1 Introduction

With over 700.000 new device-activations per day, Android is undoubtedly the most popular operating system for smartphones and tablets, rivaled only by Apple's iOS. Naturally, cyber criminals are also aware of this significant spread. The fact that, unlike iOS, Android allows installation of apps from arbitrary sources, is an additional incentive for them to focus on subverting the supply of apps with malicious software. Reports by Anti-Virus companies back the increasing interest in malware for Android with concrete numbers. In the second half of 2011, the number of backdoors alone has risen by 285% [33, 17].

Google has reacted swiftly: in February 2012 they revealed the existence of *bouncer* [35], a service that transparently checks apps in the Google Play Store for malware. They further report that this service has led to a decrease of the share of malware in the Play Store by nearly 40%. However, Android users are not limited to the official Google Play Store when it comes to installing software. Apps are available from various sources – these can either be bulk archives which can be retrieved via torrents or one-click-hosting services, or complete alternative app markets that come with a dedicated installer app and host their own repositories. The possibility to install arbitrary applications is one of the major differences between Android and iOS and reflects the credo of Google and Apple respectively. Naturally, especially bulk archives are very unlikely to be checked for malware before they are released and so the question arises whether they really contain a large share of malware.

Analyzing or detecting malware follows the same basic principle that research on x86 malware relies on. On one hand, *static analysis* yields information immediately by just looking at a sample, while *dynamic analysis* actually executes the sample and provides details on its true behavior with

the disadvantage of being slower and more resource intensive. A large body of research [52, 15, 30, 28] uses these methods, while none of them provides a comprehensive technical solution that combines them to provide a thorough feature set for a sample. However, post-analysis techniques such as clustering tend to generate more meaningful results if they are applied to a rich feature set.

As a consequence, we significantly extended, combined and automated state of the art analysis solutions for Android. Our dynamic analysis is multi-layered: On the Dalvik-VM-level, we enhance the well known *DroidBox*[20] to record additional information. To also cover system-level events, we instrumented the Qemu-based Goldfish emulator to keep track of system calls and native library activity. As some characteristics are only exposed if they are triggered by specific interaction with the sample, we also provide targeted stimuli during the analysis. To be able to customize the set of stimuli for each sample we leverage information from prior static analysis.

Our contributions can be summarized as follows:

- We introduce ANDRUBIS, a fully automated dynamic analysis framework that includes both static and multi-layered dynamic approaches to analyze unknown Android applications.
- We provide a detailed analysis of different sources for Android applications and compare their behavior to known malware apps.
- Using ANDRUBIS, we analyze and cluster more than 27.000 samples from different sources. We further give an insights whether specific sources are prone to deliver specific strains of mobile malware. We also provide a set of properties which were identified as common elements in mobile malware, paving the way for an automated filtering procedure of suspicious applications.
- To provide our solution for the research community, we integrated the system as part of the well-known Anubis analysis framework. It is open for submissions under `http://anubis.iseclab.org`.

6.2 System Description

The basic idea behind our framework parallels several other approaches developed for x86 or, more specific, Windows systems. The core component is a virtual machine which is running the sample under scrutiny and records every action down to the last detail. However, due to the special structure of the Android operating system, some of the components are different. In this section, we discuss every major component, how they are integrated into the system and how they differ from their x86 counterparts.

6.2.1 Sandbox

Unlike Windows malware, Android is based upon the ARM architecture, a fact which heavily influences the sandboxing system as a whole. As it automatically implies the use of an emulator if used under a large-scale analysis environment, the choices when choosing the sandbox itself are quite limited. For ANDRUBIS, we utilized *DroidBox*[20], a Qemu-based virtualization environment which was initially designed to run arbitrary Android applications and monitor behavior that happens within the operating system. Since Android is based on Java, it closely monitors the Java Virtual Machine (JVM) and essentially records actions happening within this environment. For a comprehensive analysis, however, these capabilities are not sufficient.The original emulator has been extended with the following facilities.

- **Tainting**: To track privacy sensitive information ANDRUBIS uses *TaintDroid* which enables us to detect sensitive information leaving the phone through taint analysis. This part comes with the usual restrictions applicable to data tainting and is limited to JVM level. Therefore, tainting of native libraries is not supported.

- **Deployment**: For a large-scale automated approach it is imperative to automatically deploy and undeploy samples within the analysis environment. As each sample uses its own virtual machine and its resources, we have a limited amount of time to analyze each sample.

- **VMI**: To overcome the shortcomings of DroidBox and Taintdroid, we implemented a VMI (virtual machine introspection) based solution to track system calls of potentially harmful native libraries. A detailed discussion of this feature is presented in 6.2.7.

To mitigate potentially harmful effects of our analysis environment, we took precautions to prevent samples from executing DoS attacks, send Spam mails or propagate itself over the network. This part is essentially based on our experience with x86 malware analysis and proved to be effective in the past. The rest of the sandboxing system (host environment, network setup, database, etc.) is comparable to conservative analysis systems and thus not described further. In Figure 6.1, the structure of our system and its elements are shown in detail.

6.2.2 Static Analysis

Before actually executing a sample in ANDRUBIS, the first advantage when analyzing Android apps comes into effect. Android applications are packed in *Android Application Package* files (APK) which must contain an AndroidManifest.xml file. This description file is mandatory and cannot be bypassed

by any means. Without this information, the file cannot be installed or executed. In a first step, we parse Meta information from the manifest, like requested permissions, services, broadcast receivers, activities, package name and SDK version. This information can not only be used to assist in automating the dynamic analysis, it is also possible to identify permissions which are dangerous or commonly used by malware. Furthermore, it gives us an idea on how many permissions are requested by the app in the first place, compared to what was actually used during execution. Without anticipating our evaluation results, the set of used and requested permissions greatly differ between malware and benign samples. Compared to approaches where this data was used to distinguish between malware and goodware [10], we merely use static analysis results for a guided execution and analysis in our dynamic part.

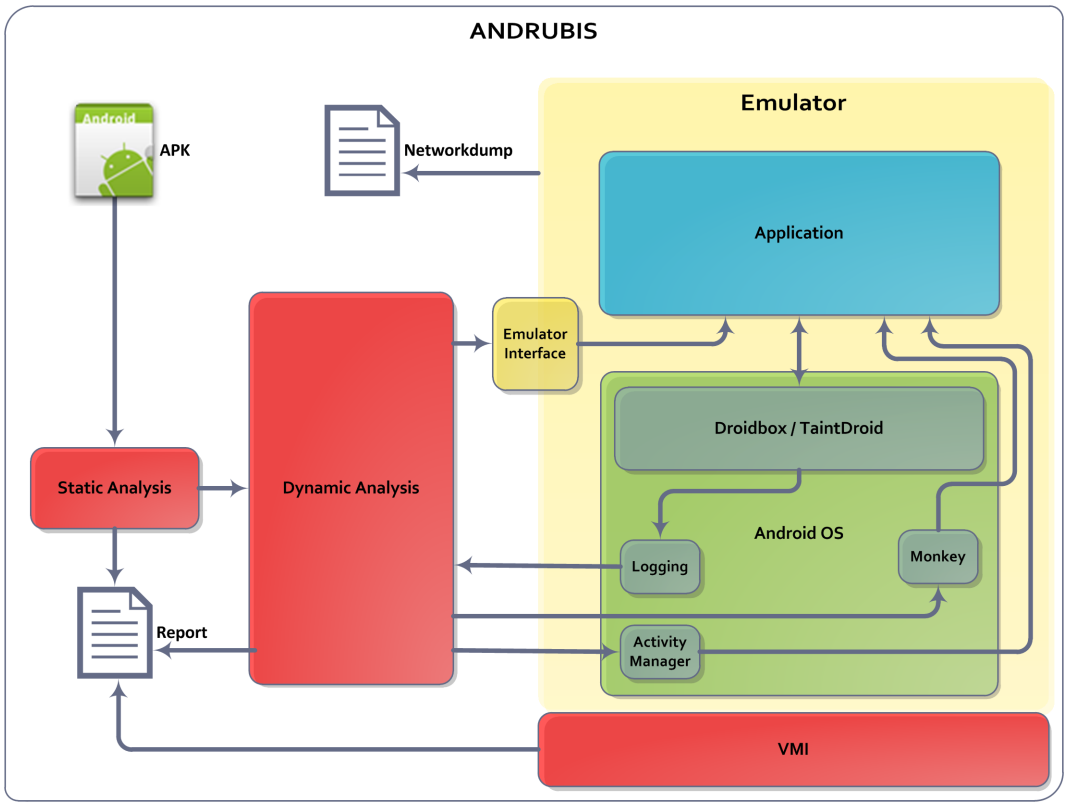

Figure 6.1: Architecture of ANDRUBIS

6.2.3 Stimulation

The elements presented so far comprised the static parts of our framework. The first relevant dynamic property we like to discuss is event stimulation.

Stimulation is essentially about code coverage. One major drawback of dynamic analysis in general is the fact that not all execution paths are certain to be traversed within one analysis run. In traditional programs, this problem is even more severe, because the only function which is guaranteed to be called is the main function or the main dialog. What happens from there depends on the program itself. Fortunately, the Android OS again provides the facility to partially overcome this problem. Defined within the application's manifest file (AndroidManifest.xml), a list of services, broadcast receivers and activities can be found. Some of them are not necessarily defined within the manifest but can be registered programmatically.
After the initialization of the emulator, ANDRUBIS installs the application which should be analyzed and starts the main activity. At this point, all predefined and programmatically registered entry points are known, which enables ANDRUBIS to perform the following stimulation events.

6.2.3.1 Activities

An `activity` provides a screen for the user to interact with. Like services, activities have to be registered in the AndroidManifest.xml and cannot be added programmatically. These activities define the interaction sequences presented to the user and come with a defined layout, which must be known in advance. By parsing the manifest, ANDRUBIS can invoke each activity separately, effectively iterating all existing dialogs within an application. However, this method comes at a price. Poorly written applications have the potential to crash if they are not thoroughly tested. Rarely used and badly implemented activities, like *About* screens, for example, are capable of ending the analysis run prematurely. More precisely, it ends the analysis of the current activity. As soon as the next activity is triggered, a new process is spawned by the Dalvik VM for the analyzed app. While this does not hinder dynamic execution of a sample, the changing process ids negatively impact id-specific analysis results like JNI invocations for instance.

Although we found no malware sample in our analysis which uses this technique to disrupt dynamic analysis, we acknowledge the fact that it could be used to distort our analysis results. In our current version, erroneous activities are simply omitted if the error is recoverable. Besides, applications that crash are immediately tagged with bad reviews if they appear on the market. For a malware-infested application that is certainly not desirable.

6.2.3.2 Services

Background processes on the Android platform are usually implemented as `services`. Other than activities, they come without a graphical component and are designed to provide some background functionality for a program. Naturally, they are interesting for malware writers as well, as they can be

used to implement data transfers to botmasters, upload personal information or send received SMS to an adversary. Again, all services used by an application must be listed in the manifest. Their existence, however, does not automatically mean the service is started under every circumstance. To save battery life and conserve memory, services have to be started on-demand, with a lifetime defined by the programmer. ANDRUBIS iterates and starts all listed services automatically after the application is deployed. This is done with the *Android Debug Bridge* (ADB) provided by the `ActivityManager`.

Stimulation Event	Target
Activities	All activities from manifest
Services	All services from manifest
Broadcast Receivers	All Broadcast Receivers from manifest, *registerReceiver()* from android
Common events	send/receive SMS, connect/disconnect WLAN/3G, GPS lock, send/receive phone call, boot completed phone-state-changed
Random events	Random input stream by Application Exercise Monkey

Table 6.1: Stimulation events

6.2.3.3 Broadcast receivers

Another possibility to enter an Android application, is by utilizing a `broadcastreceiver`. They can be used to receive events from the system or other applications on the Android platform. Just like services and activities, they can be registered in the manifest but that is not mandatory. To provide the possibility to react to certain events and realize communication with other applications dynamically, they can be registered and unregistered at runtime. For example a broadcast receiver for the `android.intent.action.BOOT_COMPLETED` event could be registered to start an application after the phone has finished its starting sequence. Similar to the previous stimuli, broadcast receivers from the manifest are always called within ANDRUBIS. All information to create and broadcast the necessary event is available from the manifest. Again, the `ActivityManager` is used to cast the message globally.

The case is a bit different for dynamically registered broadcast receivers. To find out if a program registers one, we intercept calls to `registerReceiver()`. As a result, we are provided with a list of dynamically registered events that should be triggered. To transport information between the caller and the receiver, however, Google implemented so-called `extras`, simple key-value pairs that can be filled as needed. While the content of

the extra field for Android-specific events (e.g. alarms, wifi connect, etc.) is predefined, a programmer could use it to transport arbitrary, but semantically important data. Automatically and reliably creating such an extra field would require a tremendous amount of effort. To derive all possible elements of this dictionary, a complete call graph of the application and a considerable amount of back tracing would be necessary. Besides, the gain when triggering dynamically registered, non-Android broadcast receivers is questionable at least. In our first approach, we filled extra fields with empty lists and stimulated the receiver, risking to crash the application if a possible null pointer exception was not properly handled by the programmer. If that happened, we restarted the sample and continued with another broadcast receiver. While this approach works in principle, it causes unwanted side effects. Restarting a process, for instance, changes the process id, which in turn complicates a proper system-level analysis. Therefore, we decided to use an alternative method to stimulate dynamically registered broadcast receivers.

6.2.3.4 Common events

A far superior method compared to stimulating broadcast receivers with a targeted event is to emulate some common events a sample is bound to react to. In contrast to directed stimuli, these events are implemented on emulator-level and therefore also trigger receivers from the Android OS itself. That, in turn, avoids causing inconsistent states the OS would have to recover from. By broadcasting these common events (e.g. SMS-received, GPS-lock, boot-completed, phone-state-changed, etc.), we are able to trigger most functions even if they propagate data by custom broadcast receivers. A list of currently implemented common events can be found in Table 6.1.

6.2.3.5 Application Exercise Monkey

The last elements that need to be stimulated are actions based on user input (e.g. button clicks, file upload, entry fields, etc.). For this purpose, we use the Application Exercise Monkey, which is a part of the Android SDK and generates semi-random user input. Originally designed for testing Android applications, it randomly creates a stream of user interaction sequences that can be restricted to a single package name. Its hit ratio for buttons is quite astonishing and we found it triggers a good amount of functionality that we could otherwise not see. For a sophisticated interface stimulation, however, some sort of GUI fuzzing would be needed which leverages static analysis results and derives GUI layout and interface elements from this information. This part is currently being developed but not yet ready to be deployed and therefore listed under future work.

6.2.4 Tainting

Data tainting is a double-edged sword when it comes to malware analysis. On one hand, it is the perfect tool to keep track of interesting data, on the other hand it can be tricked quite easily if a malware author is aware of this mechanism within an analysis environment [16]. By leaking data through implicit flows, for instance, it would be possible to circumvent tainting. A shortcoming which could be mitigated by a complete static analysis, for instance. Furthermore, enabling data tainting always comes at the price of additional overhead to produce and track taint labels. Still, the possibility to track explicit flows, like device IMEI to network for instance, is a valuable property of a dynamic analysis framework. ANDRUBIS leverages *TaintDroid* [23] to track different kinds of sensitive information across application boarders in the Android system. The introduced overhead in processing time of approximately 15% [23] is also acceptable for our purpose. As a result, ANDRUBIS can log tainted information leaving the system through sinks (network, SMS or files). Table 6.2 shows which kinds of sensitive information ANDRUBIS tracks during the execution phase. The scope column denotes the scope of the taint source. We basically distinguish between three different options:

- User: Data that is primarily used to identify the user operating the device. A good example is the address book with all the contacts which is sent over the network by some apps.

- Device: Independent of the user, each device running Android can be identified by several properties. The best example is probably the device IMEI, a unique number that is often used by ad libraries to count and report installation numbers.

- Carrier: Information about the mobile carrier. Just like unique device identifiers, this information uniquely describes the used SIM card and therefore the user operating the phone.

We distinguish between these three scopes so we can describe which type of privacy concern may arise when analyzing a sample.
One restriction when using *TaintDroid* is that it refuses to let an application drop to native code execution. The reason for that is quite simple. Tainting mechanisms only trace labels within the Dalvik VM. When native code is executed, there is almost no restriction on what the application can do, as long as it is within the permissions granted. Related studies [52] claim, that around 5% of all available applications use native code during their execution. What may sound as a negligible amount is in truth a good indicator for malware. Apps that heavily rely on native code to fulfill their task have a good chance of being malware simple because the possibilities for the

malware author are so much greater. As a result, prohibiting native code execution is unacceptable. Consequentially, we have removed the restriction that forbids the loading of custom native code (i.e. native code which is not located in the */system/* directory) from the original *TaintDroid* implementation. ANDRUBIS logs the use of native code and therefore can warn the auditor that the results of the taint analysis may not be trustworthy. Since the operations performed in the native code library are of particular interest for a thorough analysis, we decided to instrument the Qemu framework and utilize virtual machine introspection (VMI) to monitor the resulting actions. A detailed description of our native code analysis is provided in the next section.
The following listing exemplarily shows how ANDRUBIS traces information leakages:

```
<data-leaks>
 <network-leak seconds="59.0080029964" tag="TAINT_CONTACTS,
     TAINT_SMS, TAINT_CALL_LOG, TAINT_BROWSER">
    <host><![CDATA[gi60s.com]]></host>
    <port>80</port>
    <data><![CDATA[code=67ffc&data={"contacts":[{ "name":"Qm9i","
        numbers":"MDEyLTM0NS02Nzg5Ow==",{ "name":"QWxpY2U=","
        numbers":"MDgwLTAwOC0xNTs="}],"sms":[{ "address":"
        MDgxNTEyMzQ1Njc4OQ==","type":"1","date":"1329836847907","
        body":"SGVsbG8gV29ybGQh"},{ "address":"MDEyMzQ1Njc4OQ==","
        type":"1","date":"1329836407315","body":"
        SGVsbG8hIFRva2VuIGFzZGxmaWkyODIyNzdq"}],"recent":[{ "
        number":"0815123456789","type":"1","date":
        "1329836847797","duration":"0"},{ "number":"0123456789","
        type":"1","date":"1329836383176","duration":"0"},{ "number
        ":"08000815","type":"2","date":"1329836360814","duration":
        "3"}],"url":[{ "url":"aHR0cDovL3d3dy5nb29nbGUuY29tL3VybD9z
         YT10JnNvdXJjZT13ZWImY2Q9MSZ2ZWQ9MENCMFFGakF
        BJnVybD1odHRwJTNBJTJGJTJGd3d3LmlzZWNsYWIub3
        JnJTJGJmVpPUxySkRUOURJQmNYT3NnYTZpcjNnQkEmd
        XNnPUFGUWpDTkdETUFKdG5wcUhfZElESVBURE1LejRG ZUdUbkE="}
    ...
    ]]>
    </data>
  </network-leak>
</data-leaks>
```

Listing 6.1: Example information leakage section in ANDRUBIS report

In this example, private information like contacts, the call log, etc. were sent over the network to a server (gi60s.com) on port 80. In this particular case, however, the application is a backup solution that stores contact information, SMS, etc. on the provided Webserver.

Scope	Taint Source
Device	Location
User	Address Book (ContactsProvider)
Device	Microphone Input
Carrier	Phone Number
User	GPS Location
User	NET-based Location
User	Last known Location
Device	Camera
Device	Accelerometer
User	SMS
Device	IMEI
Carrier	IMSI
Carrier	ICCID (SIM card identifier)
Device	Device serial number
User	User account information
User	Browser data, history or bookmarks
Any	Other database data
Any	File content
User	Installed packages
User	Call history
User	Email data
User	Calendar data
Device	System settings

Table 6.2: Tracked taint tags

6.2.5 Logging

In ANDRUBIS, all interesting and dangerous API calls, along with the activities on native level, are logged and processed as XML files. We patched the key API functions, like reading or writing of network streams directly on the emulator and on JVM level to provide this information. Currently, these are the actions logged by ANDRUBIS:

- Incoming/outgoing network data
- File read and write operations
- Started services
- Loaded classes through DexClassLoader
- Loaded native code libraries

- Information leaks via the network, file and SMS
- Circumvented permissions
- Cryptography operations performed using Android API
- Listed broadcast receivers
- Sent SMS and phone calls
- Native system calls including call parameters

Our basic assumption was, that malware writers are certainly able to find a way and circumvent the permission system. One thinkable method would be to use a local exploit that drops a root shell and grants the needed permissions. We deal with this scenario by comparing the statically analyzed permissions to the called APIs and their required permissions. Therefore, if an operation was used without having the required permission beforehand, this action would be triggered. In all of our over 27,000 analyzed samples (and a good portion of those are malware), we did not encounter such a case even once. A fact that speaks in favor of Android's security structure. Once this action is triggered, it is a good indicator that mobile malware is becoming more sophisticated.

6.2.6 Network Analysis

Capturing network traffic is one of the essential parts when dealing with modern malware - C&C communication is undoubtedly one of its corner stones. In addition to tracking sensitive information to network sinks via tainting, we also record all the network activity during analysis regardless of the performed action or the application causing it. The dumped network trace is post-processed by customized BRO [39] scripts. We extract high-level network protocol features that are suitable for identifying interesting samples. Currently, we focus on well-known protocols such as HTTP, DNS, FTP and IRC. In general, network traffic is one of the most important features for establishing a malware-detection metric. If a piece of malware is supposed to do something else than destroy local data it is bound to request Internet permissions. Sending Spam, leaking private data or connecting to a botmaster is not possible otherwise. Therefore, the fact alone that an application is **not** requesting Internet permissions rules it out as being interesting malware. According to studies performed in production environments [29], more than 98% of x86 malware samples established a TCP/IP connection. Given the incentives of malware authors, we don't expect this behavior to change for mobile malware.

6.2.7 System Level Analysis

The last element of ANDRUBIS we want to describe in more detail is the aforementioned possibility to track native code execution. By default, Android apps are Java programs, being distributed as an apk file, which is basically a jar container. Hence the default way of programming for Android and executing Android applications is by running Java byte code within the Dalvik JVM. Analyzing a sample at the Java-bytecode level is an absolute necessity and the previously described parts of the system are dedicated to that purpose. However, Android apps are not limited to Java byte code. Via the Java Native Interface (JNI) it is possible to use native code system-level libraries. This functionality is mainly intended for performance-critical use-cases such as displaying 3D graphics. But apps are not limited to load the Android OS' native libraries, they can just as well load their own native libraries and thus execute their own system-level code. Naturally, such code would not be covered by a mere observation at the JVM-level, like the one implemented by Droidbox. Actually, most of recent research on Android malware only deals with the JVM-level and would thus miss malicious activity at the system-level. There are a couple of ways to implement system-level instrumentation in Linux, such as using LD_PRELOAD, ptrace or a loadable kernel module. We decided to use the most transparent and non-intrusive way - virtual machine introspection. With virtual machine introspection our analysis code is placed outside of the actually running Android OS, right in the codebase of the Goldfish emulator. To capture system-level behavior, we ultimately need to know what the library code loaded via JNI actually does. The first step towards that goal is to intercept the Android dynamic linker's actions. This allows us to track down when shared objects are loaded via JNI and gives us the memory region of the text segment as well as the addresses of the exported functions. System call tracking bundled with this information enables us to associate system calls with invocations of certain functions of loaded libraries. The result is a complete list of system calls done by the emulator as a whole. This data is processed and structured to reflect only system calls invoked by the app under scrutiny.

6.3 Evaluation

The evaluation of the system we presented in the previous sections consists of multiple parts that all aim to answer one basic question:"Is the system fit to produce the needed data for automated malware analysis of Android apps?" To answer this question, we did not restrict our evaluation to a single procedure. For example, a possibility would have been to simply extract used and requested permissions for malware and benign samples, span a vector space with each permission as a dimension and produce a malicious-

ness percentage for new samples according to their Euclidean distance from malware groups. This method would ignore data leaks completely, one of the main contributions of our approach. Consequently, we evaluate the system from different angles with particular focus on comparing the maliciousness of the different sources we used to gather our samples.

6.3.1 Data sets

A critical part was how to choose the data set for the evaluation. We aimed for a high diversity in our collected samples and therefore did not restrict our gathering process to Android markets or the Google Play Store. We also downloaded application archives via BitTorrent networks and one-click hoster (OCH) like rapidshare, uploaded.to etc. Table 6.3 depicts the actual numbers for each source.

	GP	VT	PS	DD1	DD2	T1	T2	T3	Malware	Total
APKs downloaded	1260	615	14141	2425	1341	2872	1982	9586	191	34413
Size (in GB)	1.6	0.8	22	2.8	2.2	4.4	3.1	11	0.4	48,3
Unique md5 hashes	1260	615	14141	1277	1331	1940	1960	4551	187	27262
PS overlap	0	1	14141	1	1	2	15	4	4	-

Table 6.3: Dataset size (GP = Genome Project, VT = Virus Total, PS = Play Store, T = Torrent, DD = Direct Download)

Feature group	GP	VT	PS	DD1	DD2	T1	T2	T3	Malware
File activity	94%	82%	70%	52%	47%	70%	66%	62%	79%
Network activity	76%	34%	61%	20%	17%	28%	27%	25%	59%
Phone activity (SMS)	4%	29%	0%	0%	0%	0%	0%	0%	10%
Native library load	18%	9%	10%	7%	6%	18%	14%	12%	18%
Data leak	50%	17%	17%	7%	4%	8%	9%	8%	40%
Crypto Operations	24%	4%	2%	1%	2%	6%	6%	5%	14%

Table 6.4: Share of samples per dataset that exhibited certain dynamic feature groups (GP = Genome Project, VT = Virus Total, PS = Play Store, T = Torrent, DD = Direct Download)

All samples, with the exception of the Genome Project, were downloaded from May to June 2012. Genome project samples stem from a larger period starting in August 2010 and are verified to contain malware from 50 different malware families [50]. We used this set as our malware baseline. One disadvantage is embodied by the age of some malware samples. Especially malware that relies on a functioning server is problematic to analyze if this server was already shut down. Nevertheless, our baseline showed a good overall activity as explained in the next section.
Our second malware indicator is provided by Virus Total [43]. Almost every modern virus scanner already contains signatures for Android apps. Virus Total, which utilizes 40 different scan engines to produce their result, is used as a control mechanism in this evaluation. For all 1260 samples within the Genome Project for example, Virus Total reported that 1218 (97%) contained malware. The remainder of the samples were collected as follows:

- Virus Total: By courtesy of Virus Total [43], we downloaded over 600 Android samples from their database, where multiple scanning results indicated the app to be malware.
- Play Store: This is a snapshot of some of the most recent apps published on the Google Play Store that were crawled during these two months.
- DD1/2: These samples indicate direct downloads from various one-click hosters. DD1 originated from crawled forum entries aggregated by http://filestube.com. The original sources stem from various forum entries. DD2 originated from a single site called http://iload.to, before the administrators decided to take the site down to avoid legal issues.
- Torrents 1/2/3: These samples stem from downloads from http://thepiratebay.se, http://torrentz.eu and http://isohunt.com respectively. Each torrent with more than 10 seeders was downloaded. To avoid distribution of copyright-protected content, we prohibited our torrent client from uploading any data at all. As a result, the download speed for some samples was rather slow as ”leechers” get punished by limited downstream.
- Malware: This is a small collection of manually gathered malware samples we encountered during our studies. They also stem from various sources which are too many to list here. This collection was designed to act as our manually vetted malware baseline before we received the samples from the Genome Project.

Torrents and direct downloads were kept separate on purpose. It allows us to see possible deviations in our results not only between different sources

but also within the sources themselves. One noteable difference between the dataset is the percentage of paid apps. All samples crawled from the official Google Play store are free to download. Direct downloads, torrents and even some of the malware samples, however, partially contain non-free commercial apps. Before downloading the samples, we presumed that external sources exclusively distributed non-free apps. This assumption was wrong, however. Most of the uploads are collections of apps and games the publisher deems useful. And that comprises freeware apps as well as paid ones.

6.3.2 Quantitative Results

With this dataset, we conducted our first tests. During our evaluation phase, we constantly upgraded our system to provide new features for a public release. For our evaluation environment that meant we had to freeze our system on a certain point to get homogenous results. Before deployment, we had a limited throughput of about 1500 samples per day, which also explains the size of our dataset. The complete analysis of the dataset took roughly a month. For each sample, we executed three steps.

1. We performed static analysis
2. We performed dynamic analysis with a timeframe of 5 minutes per sample
3. We submitted the sample to Virus Total directly after execution. If no report existed for the specific sample, we re-queried Virus Total again after 48 hours and 7 days.

Table 6.4 shows to which extent the samples exhibited certain feature groups.

Data Leaks	GP	VT	GP	DD1	DD2	T1	T2	T3	Malware
IMEI	45.6%	14.3%	9.4%	1.4%	1.5%	2.7%	2.0%	2.0%	28.3%
IMSI	26.2%	6.3%	0.8%	0.3%	0.0%	0.5%	0.3%	0.2%	20.3%
Phonebook	0.8%	0.3%	0%	0.2%	0.0%	0.1%	0.1%	0.0%	2.5%
Phone Number	15.0%	8.0 %	0.7%	0.2%	0.2%	0.2%	0.2%	0.2%	11.4%
Location	1.6%	1.4%	2.1%	1.2%	0.8%	1.6%	1.2%	1.1%	1.7 %

Table 6.5: Share of samples per dataset that leaked sensitive information over the network (GP = Genome Project, VT = Virus Total, PS = Play Store, T = Torrent, DD = Direct Download)

Overall, the Genome Project has very high overall activity in the main feature groups such as file and network activity, even though some of the samples remained dormant while we executed them. We limited the phone-specific facilities in the Table to the SMS service, as none of the samples

in all datasets initiated phone calls while we executed them. Sending SMS does not seem to be a very widespread feature as it is not represented in the Torrent and DD datasets.

At this point we are further able to analyze claims of previous research papers. In [52] 4.52% of all investigated apps used native code libraries, in [49] it was given with roughly 5%. The values we depict in our Table are 18% for the Genome Project and 10% for market apps. At a finer granularity, we can also distinguish between total libraries loaded, system native libraries loaded and non-system libraries loaded. Here the respective values are 17.88% / 7.17% / 12.08% for the Genome Project and 9.63% / 8.19% / 1.99% for the market. The last digit represents custom libraries which are far more dangerous than those provided by the Android system itself. Overall, we can say that native library usage has drastically risen in general. The reason for system library usage is simple. More developers are creating games and graphically enhanced apps. For this purpose they have to load the system's openGL library, which is of course implemented natively to utilize the onboard GPU. Custom libraries are, however, a good indicator for malware and are heavily used in the samples we analyzed. We conclude that dynamically loaded native code is ostensively used for updating purposes.

Another observation supporting this assessment is represented by the Java-based method to dynamically load modules. Instead of using JNI to invoke native code libraries, it is also possible to implement classes and load them with the DEX `classloader`. This practically never happened in samples from the market, direct downloads or torrents ($< 0.1\%$) but in over 16% of all malware apps. Furthermore, these elements are exclusive, meaning that either a sample loaded a native library or a java class, but never both. In total, a third of all malware samples loaded non-system code at runtime. Something that happened in less than 2% for other samples. Overall, dynamically loading code on either native or Dalvik level is a strong hint that a sample is malicious.

The next observation we want to discuss deals with data leaks. Of particular interest is sensitive information that was transmitted over the network. For the sake of clarity, we therefore omit results where information was sent via SMS messages or directly written to a file. The same information was collected for iPhone applications in 2011 [21]. Here, the authors found that 21% of all applications they analyzed, leaked the device id. The main reason for that is that freeware programs often use ad libraries to create revenue. The most popular library is Admob which in turn is owned by Google. On Apple smartphones, the device id is used to identify installations. Since Google requires every Android user to create a Google account to use the Play Store anyway, they don't need arbitrary means to track app installations on their devices. That explains the low number of apps leaking the device IMEI over the network. The detailed figures on privacy leaks shown in Table 6.5 additionally give a hint on how the various sources are composed.

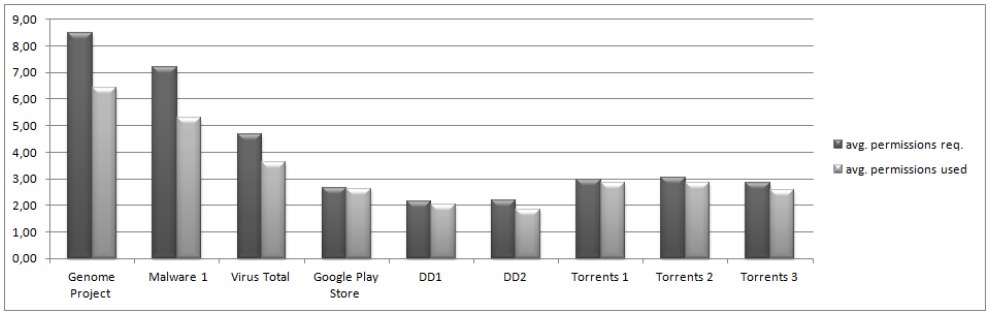

Figure 6.2: Average permissions

The first hint that external sources like OCH and torrents are not particulary dangerous was shown in the previous table. Most of the apps did not load any native or Dalvik code. This claim is also supported by the numbers for privacy leaks. In general, malware primarily leaks device identifiers (IMEI and IMSI) and phone numbers. Personal information like current location or even phone book entries seem to be less interesting. The International Mobile Subscriber Identity (IMSI) is almost never leaked in traditional samples. The IMEI on the other hand is seen in 9.4% of all samples from the traditional market. As mentioned before, these are mostly third party advertisement libraries that need to track devices to estimate installations. Interestingly, values for IMEI are mostly below 2% for direct downloads and torrents. That, in turn, supports the claim that these channels are not primarily used to distribute malware but to share pay-apps. Furthermore, our evaluation shows very consistent values across external sources.

As expected, Android apps distributed via OCH and torrents do not show very distinct behavior. They are simply collections of useful tools, if possible in their full version without advertisement.

6.3.2.1 Permissions

The next feature used in related papers [26] to detect malware-specific abnormalities is the set of permissions an app uses. During our evaluation we noticed two properties that were different between malware and control sets. First, malware apps tend to request a lot more permissions than benign samples. Figure 6.2 shows in detail, how the distribution across our sets looks like. Apps downloaded from the Google store, for instance, request a third of the number of permissions that samples from our ground truth do.

The concrete numbers for download sources again support our assumption that neither OCH-hosted files, nor torrents are overly infected with malicious apps.

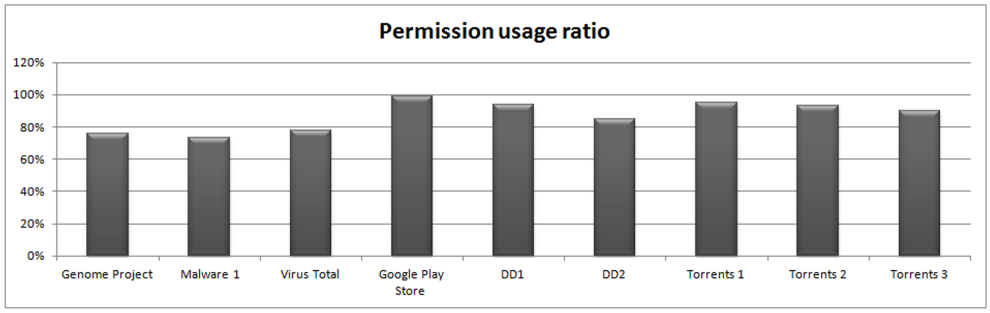

Figure 6.3: Permission usage ratio

Figure 6.3 shows which permissions were actually used during dynamic execution. The assumption was, that malware samples request more permissions during installation than are actually needed so later they have the possibility to load other code parts that use these permissions. We could only partially prove this assumption. The official market exhibits a 99% ratio of used versus requested permissions. Almost each application used all the requested permission during its 5 minutes execution window. We see this as proof, that our stimulation engine described in 6.2.3 does a very satisfactory job in nudging broadcast receivers and activities. The only non-malware set that dropped below 90% in this ratio was DD2. Upon further investigation we discovered that this set contains a large amount of games. Games are, in general, hard to guide. Even with a GUI stimulator, we are not confident that all actions can be triggered in a reliable way.

In comparison, the malware sets did exhibit a lower percentage in their permission usage ratio. With 73-78%, however, it is not significant to distinguish between malware and goodware. The total amount of requested permissions is a far better hint. On a side note, the total amount of existing permissions is 55. ANDRUBIS records permission uses on a finer grained level, amounting to 263 possible permissions and their corresponding invocation.

We assume that the main reason for the high permission utilization of malware samples also lies in our stimulation engine. By triggering all broadcast receivers and activities, we also trigger functionality that would otherwise lie dormant during ordinary execution. Finally, a look into the frequencies of requested permissions shows that dynamic behavior is almost completely in line with the static results presented in [50]. Even on this much larger sample base and with dynamic analysis, the suspicious permissions are the same.

The same pattern can be observed for the frequency of registered Broadcast receivers. Table 6.7 shows that market samples above all watch for a user being present. To save battery power, most apps switch to idle mode when the phone is locked or the screen is turned off. Malware, on the other

Play Store		Genome Project	
Permission Name	Percentage	Permission Name	Percentage
INTERNET	86%	INTERNET	98%
ACCESS_NETWORK_STATE	54%	READ_PHONE_STATE	94%
WRITE_EXTERNAL_STORAGE	33%	ACCESS_NETWORK_STATE	82%
READ_PHONE_STATE	22%	WRITE_EXTERNAL_STORAGE	67%
ACCESS_COARSE_LOCATION	21%	ACCESS_WIFI_STATE	65%
ACCESS_FINE_LOCATION	21%	READ_SMS	64%
VIBRATE	17%	RECEIVE_BOOT_COMPLETED	55%
ACCESS_WIFI_STATE	11%	WRITE_SMS	53%
WAKE_LOCK	11%	SEND_SMS	43%
CALL_PHONE	8%	RECEIVE_SMS	39%

Table 6.6: Top 10 used Permissions

hand, often registers as a service which is running in the background and does not care for user input in may cases. The most prevalent event they listen for is `BOOT_COMPLETED`, which triggers as soon as the phone is switched on and reports ready for operation.

Play Store		Genome Project	
Permission Name	Percentage	Permission Name	Percentage
USER_PRESENT	14%	BOOT_COMPLETED	55%
INSTALL_REFERRER	12%	UMS_DISCONNECTED	15%
BOOT_COMPLETED	5%	SCREEN_ON	10%
APPWIDGET_UPDATE	4%	SMS_RECEIVED	9%
CONNECTIVITY_CHANGE	2%	PHONE_STATE	7%
REGISTRATION	2%	CONNECTIVITY_CHANGE	5%
SERVICE_STATE	2%	NEW_OUTGOING_CALL	4%
PURCHASE_STATE_CHANGED	1%	USER_PRESENT	4%
SMS_RECEIVED	1%	ACTION_POWER_CONNECTED	4%
BATTERY_CHANGED	1%	UNINSTALL_SHORTCUT	3%

Table 6.7: Top 10 registered Broadcast Receivers

6.3.3 Qualitative results

While the quantitative analysis from the previous section provided insights into the structure of the various sources, it did not deal with concrete applications and their specifics. In this subsection, we discuss how we used ANDRUBIS to get a more detailed understanding of our samples.

6.3.3.1 AV labels

One side effect of the Anti Virus (AV) labels we use for our evaluation is that we get an overview on the current AV landscape. In principle, AV scanners are fit to scan Android samples and detect signatures for known

malware. Since APK files are nothing more than zip archives, they can be unpacked and inspected just like their PC counterparts. Therefore, repacked and newly distributed malware can be detected if a signature exists.

Following up on the results of our quantitative analysis, we submitted all investigated samples to Virus Total [43] and summarized the results in Table 6.8. Virus Total performs a scan with 42 different signature-based scan engines from various AV-companies.

The results are categorized in known and new malware. Known malware denominates samples that were submitted before we analyzed them while new malware are samples we submitted first. For both categories, we give the absolute number of samples that produced either between one to four, or more than four hits. The reason is simply that samples can hardly be classified as malicious if just one of 42 AV engines produces a hit. In this case, the label is often a false positive or, for instance, labeled as Adware or other, less malicious tags. A good example is the relatively high number of malicious apps in set *Torrents 3*. Out of the listed 21 malware apps, 13 are either programs for rooting a device or flashing firmware images. The rest are various malware apps, including one named *KasperskyMobile_Security* which turns out to be of the *AndroidOS.Kiser* family.

Overall, we see the same picture as in the previous results. Infection rates are between 1‰and 5‰. Market samples are expectedly even less malicious with only 6 reliable hits in our set (0.5‰). In conclusion it is safe to say that malware on Android systems is not yet widely spread. Confirmable infections are individual cases, even in torrents and files downloaded from OCH. The very low infection ratio on the official market is explainable by the introduction of Google's bouncer [35]. Instead of allowing each and every app to be published on the official market, Google introduced a vetting process not unlike Apple's. If an app does not meet the requirements, it is simply rejected.

An interesting number is the first time submission rate. Two third of all market samples were previously not known to Virus Total. This reflects on the prevalence of equipment to analyze Android executables and of course sample feeds.

6.3.3.2 Zero Day Malware

To estimate the amount of zero-day malware contained in our datasets, we also utilized Virus Total. We regard samples that were unknown by Virus Total at the time of analysis and recognized as malware after we re-checked them on a weekly basis as zero-day malware. For our complete dataset that amounted to three samples total. All three samples were found in the Google Play Store. Two of them were marked as Adware. The samples themselves were obviously designed to include a good amount of ad libraries, while the actual content or functionality was next to nothing (e.g.

just a link to an external site). The third hit was more interesting. The original app is called challenging alarm clock, designed to let the user solve a puzzle before the alarm turns off. In the background, however, it leaks the user's IMEI over the network, adds bookmarks to the browser and connects to a remote server. It belongs to the Plankton family known from various older malware samples and the Genome Project amongst others. We have seen this particular malware in various versions and reported the current one to Google. At the time of this writing, it was still accessible.

VirusTotal	GP	VT	PS	DD1	DD2	T1	T2	T3	Malware
Submitted	1260	615	14141	1277	1331	1940	1960	4551	187
First time submission	362	0	9304	199	78	221	227	414	0
First time submission rate	29%	0%	66%	16%	6%	11%	12%	9%	0%
Known Malware (>0 hits)	1232	615	440	13	17	28	17	66	236
Known Malware (>4 hits)	1214	615	6	5	2	2	5	21	220
New Malware (>0 hits)	362	0	394	0	0	2	4	1	0
New Malware (>4 hits)	359	0	3	0	0	0	0	0	0

Table 6.8: Virus Total results for the datasets

6.3.3.3 DEX Class Loader

One of the strongest hints for malicious activity was found to be dynamic loading of dex classes while executing. We found this behavior in 4 apps which were not in one of our malware sets (i.e. Genome Project, Malware 1 and Virus Total). In the following we give an overview on the functionality of each finding.

- Revival 2: A game by herocraft that loads game content on demand. No malicious activity was found.
- Art of war 2 lite: A game by herocraft that also loads game content as dex classes. No other malicious activity was found.
- Stolen in 60 Seconds: Again a herocraft game.
- Aareader: A Chinese reader for different kinds of documents. It leaks private information like IMEI and IMSI to a chinese server and dynamically loads classes. Upon investigation we discovered, that the developers included several versions in their app instead of using the market for versioning. On runtime, the app decides which version to use and updates the corresponding classes on demand.

While none of the investigated samples can be rated as outright malicious, the last example certainly poses a privacy threat. After all, the user has no control on what is updated. Safe for the initially granted permission set, not

boundaries are exerted over the loaded code. In the other three cases, it is questionable if loading game updates as dex files is good practise. The usual methodology is, to update the game via the Play Store and only download non-executable game data on demand.

6.3.3.4 Clustering

One of the biggest advantages when dynamically executing programs is the possibility to create behavioral profiles with the data. In contrast to other approaches [15, 52, 50], we use the term *behavioral* for operations observed while a sample is executed. While requesting permissions is certainly a behavioral aspect as well, we entitle these actions as static for clarity's sake. Thus, a profile with only static components is strictly speaking not a behavioral profile.

After finishing dynamic analysis for all our samples, we decided to utilize a clustering algorithm and investigate the largest clusters. If applied correctly, the clusters should expose families with common properties. For ordinary approaches [34], 27.000 elements are simply too much. Usually, distance calculation for each element within the feature cloud results in a computational complexity of $O(n^2)$. To overcome this limitation and process larger sample sets, we utilized an approximate, probabilistic approach [11]. This clustering algorithm is based on locality sensitive hashing (LSH), and provides an efficient solution to the approximate nearest neighbor problem (ϵ-NNS). LSH can be used to perform an approximate clustering while computing only a small fraction of the $\frac{n^2}{2}$ distances between pairs of points. Leveraging LSH clustering, we are able to compute an approximate, single-linkage hierarchical clustering for our complete dataset.

As already mentioned above, we cluster the malware samples in our dataset based on their dynamic behavior observed during analysis as well as static features extracted from the APK files. The dynamic behavior includes features such as reading and writing to files, sending SMS, making phone calls, the use of cryptographic operations, the dynamic registration of broadcast receivers, loading dex classes and native libraries and leaking information to files, the network and via SMS. Additionally, network-related dynamic features are generated by parsing the captured network dump with bro rules. Static features include activities, services and broadcast receivers parsed from the manifest as well as required permissions and URLs. We define the distance between two samples as the Jaccard distance between their profiles.

We evaluated our clustering approach in two different configurations:

- Clustering only based on behavior observed during dynamic analysis
- Clustering based on dynamic and static features

With the already categorized Malware from the Genome Project as well as Anti-Virus labels from Virus Total we have a ground truth that allows us to find variants of similar malware samples from other sources. It also allows us to discover previously unknown samples when they are placed in the same cluster due to similarities in behavior and/or static features. We picked the most interesting clusters and provide a short discussion on their properties in the following two sections.

Dynamic features

The largest clusters that were grouped by dynamic features are defined by advertisements. Applications that include the same ad library for displaying advertisements connect to the same server and therefore feature similar dynamic results. Unsurprisingly, the largest cluster features apps using Admob as their ad library. An interesting side-effect of these results is to see the approximate share of advertisement for each provider.

An interesting cluster is represented by 38 apps. While 65% of the corresponding samples belong to the already classified malware family *droidkungfu*, the remaining 35% stem from the official market. The cluster's determining factor is a network connection to `app.waps.cn`, in most cases connected with leaking the IMEI or the phone number. Spot checking the market samples revealed that these apps are mostly in Chinese and probably not overly concerned with privacy.

Comparable elements can be seen in a smaller cluster with 23 elements. For each sample we observed the app leaking the phone number and other database content (OTHERDB) to `ade.wooboo.com.cn`. 69% of the correlating samples stem from our malware collections, while 31% can be found in the market.

Static and dynamic features

When combining static results and dynamic behavior to a more complete profile, the growing feature size enables us to watch for larger clusters with more defining features.

With 216 elements, we found a set of apps that all belong to the *basebridge* malware family. These samples are primarily distinguished by the large set of permissions they request, 15 per app on average. Elements from this cluster are relatively easy to spot but we could not find a *basebridge* variant in the wild. All samples from that cluster belong to one of our malware sets.

Taken as a whole, the combined clustering provides a lot of interesting results, especially as a means to reduce the set of apps that have to be screened manually. Discussing all of them in detail, however, would be out of scope in this context. With a reference set, the data provided by both, static and dynamic analysis elements can be leveraged to deduce a malware rating scheme or at least provide a reduced list of suspicious apps to be screened by a human analyst.

7 Conclusions

In this deliverable we gave an overview of the work that the larger research community as well as the partners of the SysSec consortium are conducting in the area of cyberattacks on ultra-portable devices. The proliferation of these devices has been growing at an amazing pace. It is estimated that Android users number in the hundreds of millions [4]. Unfortunately, this great success of the platform has been followed by a simultaneous rise of malware specifically targeting Android. According to F-Secure, the growth of Android malware is exponential [6]. We expect to see even more sophisticated malware for ultra-portable devices in the near future. The majority of the research work, is focusing on devices running the iOS operating system by Apple or the Android operating system by Google.

This, consquently, primarily affects devices such as smart-phones and tablets, and that is what this survey has focused on. Adoption of these operating systems by possibly other light-weight devices, will most likely cause them to be vulnerable to the same or similar attacks. By the same token, the defensive technologies described in this report, will also apply.

Bibliography

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

[2] AVG Community Powered Threat Report. http://aa-download.avg.com/filedir/press/AVG_Community_Powered_ Threat_Report_Q3_2011.pdf. Last visited in July 2012.

[3] ComputerworldUK: Google Bouncer beaten at Black Hat 2012. http://www.computerworlduk.com/in-depth/security/3372385/google-bouncer-beaten-at-black-hat-2012/. Last visited in July 2012.

[4] Gartner Says Sales of Mobile Devices in Second Quarter of 2011 Grew 16.5 Percent Year-on-Year; Smartphone Sales Grew 74 Percent. http://www.gartner.com/it/page.jsp?id=1764714. Last visited in August 2012.

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

[6] Mobile Threat Report, Q1 2012. http://www.f-secure.com/weblog/archives/MobileThreatReport_Q1_2012.pdf. Last visited in July 2012.

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

[8] Slideme Marketplace. http://slideme.org/. Last accessed in July 2012.

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

[10] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 73–84, New York, NY, USA, 2010. ACM.

[11] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauscheck, Christopher Kruegel, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. In *16th Symposium on Network and Distributed System Security (NDSS)*, 2009.

[12] A.R. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2011.

[13] Thomas Bläsing, Aubrey-Derrick Schmidt, Leonid Batyuk, Seyit A. Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection.

In *5th International Conference on Malicious and Unwanted Software (Malware 2010) (MALWARE'2010)*, Nancy, France, France.

[14] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical report, Center of Advanced Security Research Darmstadt, 2011.

[15] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 15–26, New York, NY, USA, 2011. ACM.

[16] L. Cavallaro, P. Saxena, and R. Sekar. Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. Technical report, Technical report, Secure Systems Lab at Stony Brook University, 2007.

[17] G Data. Gdata malware report. avsecurity.ro/GData_MWR_2_2011_EN.pdf, 2011.

[18] L. Davi, A. Dmitrienko, A.R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. *Information Security*, pages 346–360, 2011.

[19] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, 2011.

[20] DroidBox. http://code.google.com/p/droidbox. Accessed: 2012-04-01.

[21] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2011.

[22] 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.

[23] 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, 2010.

[24] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[25] 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.

[26] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM.

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

[28] Peter Gilbert, Byung-Gon Chun, Landon P. Cox, and Jaeyeon Jung. Vision: automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services*, MCS '11, pages 21–26, New York, NY, USA, 2011. ACM.

[29] Jan Goebel, Thorsten Holz, and Carsten Willems. Measurement and analysis of autonomous spreading malware in a university environment. In Bernhard M. Hmmerli and Robin Sommer, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 4579 of *Lecture Notes in Computer Science*, pages 109–128. Springer Berlin / Heidelberg, 2007.

[30] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, pages 101–112, New York, NY, USA, 2012. ACM.

[31] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. Blinc: multilevel traffic classification in the dark. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '05, pages 229–240, New York, NY, USA, 2005. ACM.

[32] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. *SIGMETRICS Perform. Eval. Rev.*, 38(1):155–166, June 2010.

[33] Panda Labs. Panda labs annual report. press.pandasecurity.com/wp-content/uploads/2012/01/Annual-Report-PandaLabs-2011.pdf, 2011.

[34] T. Lee and JJ Mody. Behavioral classification. In *15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference*, 2006.

[35] Hiroshi Lockheimer. Android and security. http://googlemobile.blogspot.co.at/2012/02/android-and-security.html, 2 2012.

[36] Federico Maggi, Alberto Volpatto, Simone Gasparini, Giacomo Boracchi, and Stefano Zanero. A fast eavesdropping attack against touchscreens. In *7th International Conference on Information Assurance and Security (IAS)*, pages 320–325, 2011.

[37] Federico Maggi, Alberto Volpatto, Simone Gasparini, Giacomo Boracchi, and Stefano Zanero. Poster: Fast, automatic iphone shoulder surfing. In *Proceedings of the 18th Conference on Computer and Communication Security (CCS)*. ACM, October 2011.

[38] 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.

[39] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.

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

[41] Rahul Raguram, Andrew M. White, Dibyendusekhar Goswami, Fabian Monrose, and Jan-Michael Frahm. ispy: automatic reconstruction of typed input from compromising reflections. In *ACM Conference on Computer and Communications Security*, pages 527–536, 2011.

[42] 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.

[43] Hispasec Sistemas. Virus total. *http://www.virustotal.com*, 2004.

[44] Asia Slowinska and Herbert Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In *EuroSys*, pages 61–74, 2009.

[45] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna. Challenges for Dynamic Analysis of iOS Applications. In *Proceedings of the Workshop on Open Research Problems in Network Security (iNetSec)*, Luzerne, Switzerland, June 2011.

[46] Liang Xie, Xinwen Zhang, Jean-Pierre Seifert, and Sencun Zhu. pbmds: a behavior-based malware detection system for cellphone devices. In *Proceedings of the third ACM conference on Wireless network security*, WiSec '10, pages 37–48, New York, NY, USA, 2010. ACM.

[47] Rubin Xu, Hassen Saidi, and Ross Anderson. Aurasium: Practical Policy Enforcement for Android Application. In *Proceedings of the 21st USENIX conference on Security*. USENIX Association, 2012.

[48] 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.

[49] 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.

[50] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

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

[52] 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.