LUND
UNIVERSITY

An Android Application Sandbox for Dynamic Analysis

Patrik Lantz
dt06pl5@student.lth.se

Master's Thesis at Department of Electrical and Information Technology
Supervisor: Ben Smeets
Examiner: Thomas Johansson

1 November, 2011

Abstract

The number of Android devices on the market is increasing and so is its user base. Malware authors sees opportunities in this increase of smart-phones by the means of economical profit, stealing private information or simply controlling devices. Lately, this threat has escalated and to prevent malicious applications from spreading, several analysis tools have been released, introducing static analysis of Android packages. As of yet, there is no dynamic analysis tool publicly available. Therefore, this thesis project aims at implementing an Android application sandbox system with the intent to provide an initial understanding of the behavior of unknown packages through analysis during runtime.

By utilizing dynamic taint analysis to detect data leakage and inserting API hooks using physical modification of the Android framework, several interesting and potentially harmful operations performed by a package can be detected. Additionally, to get an overview of the operations performed during runtime, an analysis report is generated, much like the ones in traditional sandboxes. Furthermore, by visualizing the package behavior can facilitate in the interpretation of text-based reports as well as determining similarity between analyzed packages.

Keywords: android application, data leakage detection, dynamic analysis, dynamic taint analysis, malware, sandbox, treemap

Acknowledgement

This thesis was carried out at the Department of Electrical and Information Technology at Lund University. I would like to thank Professor Ben Smeets at the department and from Ericsson Security Research Group, for giving me the possibility to work on this project as my master thesis and for giving me valuable input. Additionally, I would like to thank the Honeynet Project for giving me the opportunity to work on this project during the Google Summer of Code program for 2011, which took place June-August. During this period I received valuable input and feedback from Anthony Desnos, member of French chapter in Honeynet and researcher at ESIEA.

Contents

List of Figures

List of Tables

Listings

Abbreviations

AES	*Advanced Encryption Standard*
ADB	*Android Debug Bridge*
ARM	*Advanced RISC Machine*
AVD	*Android Virtual Device*
API	*Application Programming Interface*
APK	*Android Package*
AWT	*Abstract Window Toolkit*
CD	*Check Digit*
DES	*Data Encryption Standard*
DEX	*Dalvik Executable*
DVM	*Dalvik Virtual Machine*
GPS	*Global Positioning System*
IMEI	*International Mobile Equipment Identity*
IMSI	*International Mobile Subscriber Identity*
ICC-ID	*Integrated Circuit Card Identifier*
IPC	*Inter-process Communication*
J2ME	*Java 2 Platform, Micro Edition*
JSON	*JavaScript Object Notation*
JNI	*Java Native Interface*
MCC	*Mobile Country Code*
MD5	*Message-Digest Algorithm 5*
MNC	*Mobile Network Code*
MSIN	*Mobile Subscriber Identification Number*
NDK	*Native Development Kit*
OS	*Operating System*
PC	*Personal Computer*
QEMU	*Quick EMUlator*
RC	*Release Candidate*
SE	*Standard Edition*
SD	*Secure Digital*
SHA1	*Secure Hash Algorithm 1*
SHA256	*Secure Hash Algorithm 256*
SDK	*Software Development Kit*
SQL	*Structured Query Language*
SMS	*Short Message Service*
SVM	*Support Vector Machine*
TAC	*Type Allocation Code*
UI	*User Interface*
UID	*Unique identifier*
XML	*Extensible Markup Language*

Chapter 1

Introduction

Today's smartphones are getting more powerful for each release and have become a general-purpose computing platform. Together with the fact that sensitive information is often stored on mobile devices, this requires a new approach to protect against possible attacks. One attack scenario already exists, where malware authors tailor malicious software for mobile platforms. With the marketplace, a central repository that enables third party applications to be released and downloaded, a new infection vector has emerged.

In recent time there has been an increase of malicious Android applications showing up on both official and unofficial markets [9]. If one had a tool that via sandboxing technique would provide an initial perspective on a package's behavior, one could reduce the risk of getting exposed to such malware. In this thesis project, an application sandbox is implemented to provide analysis of unknown packages and potential malware.

This paper starts out by describing our research purpose and goals. Chapter 3. continues with the background on topics such as malware, malware analysis, dynamic taint analysis, data leakage detection and the architecture of Android. Additionally, designated threats on Android devices are presented.

Chapter 4. presents the theories that were utilized and implementation details in this project. More precisely, the overall functionality is described and details are explained related to data leak detection, detection of certain critical events, analysis visualization and description of the tools that were exploited.

Analysis reports for test-cases and real-world malware are presented in Chapter 5 together with a comparison of malware visualization. Chapter 6 describes problems and whether the desired results were achieved and ideas for additional features. In Chapter 7, this paper is concluded.

Chapter 2

Research definition and goals

2.1 Purpose

As of today, malicious Android applications have been primarily discovered by accident due to suspicious behavior. These applications are analyzed by performing static analysis of the application by volunteers and professionals tied to various antivirus companies. The two main reasons why reverse engineering can become quite cumbersome are the use of code obfuscation and when the malicious code becomes complex. By analyzing an application during runtime, it is possible to retrieve an initial overview of the malware behavior and also to gather different types of information that is dynamically generated as opposed to static analysis. In traditional malware analysis, dynamic and static analysis complement each other very well, but on the mobile platform there is no such equivalent complement due to the lack of publicly available tools for dynamic analysis.

2.2 Goals

The goal of this project is to investigate the possibility for implementing a sandbox system capable of detecting certain operations performed by an Android application during runtime. Some of the operations may differ from the ones detected in traditional sandbox analysis performed on the PC platform. The main operations and events that can occur that are of particular interest are:

Sensitive data leaks: Detect leaks of sensitive information stored on the phone by the user. This involves contacts, calendar entries, emails, call history and SMS data. Additionally, phone-specific information like IMEI, phone number, installed applications and GPS coordinates is seen as sensitive data. These leaks need to be detected in outgoing network data, write operations to files and outgoing SMS.

Network traffic: Intercepting data leaks via the network is not the only interesting data that may be transferred. To facilitate in reversing a communication protocol used by a potential malware, all incoming and outgoing network communications need to be logged.

File operations: Android malware has been discovered where vulnerabilities are exploited to escalate the privileges on the phone. Detecting read and write operations on files is crucial for reverse engineering these applications. Another aspect is sharing of information between applications via files or IPC where sensitive data leaks may also occur.

Usage of cryptography API: Malware often makes use of cryptographic functions to obfuscate its code or when reporting home to a malicious server with credentials and other sensitive data from the host its running on. Android is open source and thus it could be modified to monitor the native cryptography implementations to detect when applications negotiate new cryptographic keys or perform encryption and decryption on data. This could monitor when sensitive data is encrypted to hide the leak or encryption of a communication protocol.

Bypassing application security policies: Circumventing Android permissions have been reported in early Android versions. Mapping an application's actions during runtime with the Android security policy permissions will help in finding possible violations of the policy. This could be used to help detect vulnerabilities in the framework that could be utilized by malware authors.

Phone calls and outgoing/incoming SMS: Unintentional actions like these can be used for spam or fraud attempts when a phone user is unaware. Even though it is the user's responsibility to investigate if an application's permissions are consistent with the description it is still a possibility that such applications can slide through the user's awareness.

Several issues arise here as (a) how to track sensitive data throughout the system and detecting it leaving at a specific output channel and (b) if all this mentioned operations and events can be detected and logged on an Android emulator. Regarding (b) there is also the subject of investigation of how to save the sandbox alerts to gather an analysis report outside the emulator.

Further goals include generating visualization based on the analysis result for a more human readable presentation which is missing in many sandbox system analysis reports. The visualization is more precisely to be used to classify samples to different malware families by utilizing treemap visualization. Another crucial aspect of analyzing sandbox system reports is to get an understanding of what sequence operations occur in. With a behavior graph showing the temporal order of the operations and mapping these with the text-based analysis report, this feature could be implemented. An analysis will also be carried out against real-world samples to test these implementations, specifically the malware classification to see how well this theory works when ported to classifying Android malware.

Chapter 3

Background

3.1 Android overview

3.1.1 Architecture

Android is implemented as a software stack, customized for mobile devices. Figure 3.1 [1] shows some of the most important components of this stack.

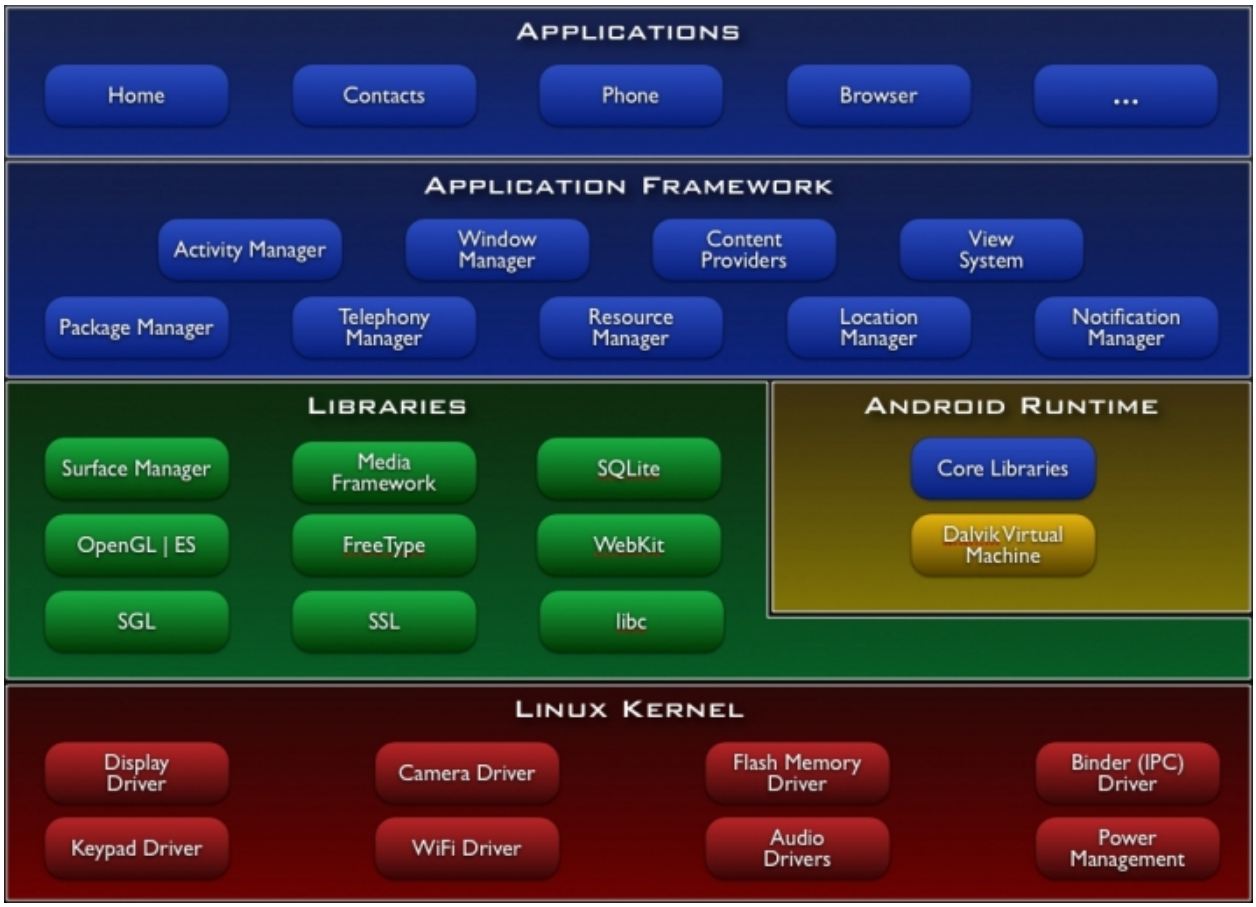

Figure 3.1. Android architecture overview

The core of the Android platform is a Linux kernel. The kernel's responsibility is handling device drivers, resource access, memory-, process-, and power management and other typical OS duties. The kernel also acts as an abstraction layer between the hardware and the rest of this software stack.

On top of the kernel are several native C/C++ libraries. Most of the application framework access these core libraries through the Dalvik Virtual Machine (DVM), which can be seen as a gateway to the Android platform. This access is based on Java APIs that are thin wrapper classes around the native code using the Java Native Interface (JNI)[1].

Programmers develop end-user applications on top of the main libraries in the application framework which provide access to the following features [1][5].

Activity manager: manages the lifecycle of an application as applications are started, suspended, resumed or destroyed (happens when the application exits), see figure 3.2. This also provides a navigation stack for the graphical views as the user is navigating through the different views within an application.

Content providers: enables applications to share its own data and access phone data such as contacts and SMS entries.

Resource manager: provides access to resources outside code such as strings, layout XML files and graphics.

View system: access to views that can be used to build the application and include buttons, lists, grids etc.

Telephony manager: information about telephony services on the device. Applications can also use this manager to determine services and states and access some subscriber information. This also enables the possibility for applications to register as listeners to receive changes of the telephony state.

Package manager: access information related to the packages installed on the device.

Location manager: access to the system location services. The services allow applications to obtain information about geographical location.

3.1.2 Android SDK

The Android software development kit supports most of the Java SE except for the AWT and Swing UI components, thus making almost all Java SE libraries available compared to J2ME[2] which is very stripped down.

Included in the SDK is an emulator to run, debug and test end-user developed applications. The emulator mimics most of the features of a real device except some limitations regarding camera and video capture, headphones, battery simulation and Bluetooth. The emulator is based on QEMU [10] which enables several operating systems to be executed on one machine and under different architectures. In this case the emulator runs an Android Linux version on an ARM[3] simulated processor.

[1] Framework for Java code that is used to call or to be called from native code such as C/C++ or assembler code.
[2] Java platform for embedded systems.
[3] ARM is a 32-bit reduced instruction set computer (RISC) instruction set architecture (ISA).

The SDK also contains several tools to assist developers, the ones significant for this thesis are:

android: manages virtual devices (AVDs), projects and installed components on a SDK.

monkeyrunner: provides an API to programatically control a Android device or emulator from outside of Android system.

Android Debug Bridge (adb): tool to enable communication with an emulator instance. This can be used to install applications to the emulator and transfer files to or from the emulator or device. Another feature is the possibility to issue command-line options to the operating system through a shell interface.

logcat: provides a mechanism for collecting and viewing all logs issued within the emulator by the Android system and applications.

The Android Virtual Device (AVD) is an emulator configuration that enables modeling of an actual device by defining hardware and software options that are then emulated. These options include: mapping to a system image, hardware features and dedicated storage area for simulating a SD card that contain user data. The system image contains the version-specific Android implementation that include the application framework and DVM.

3.1.3 Application fundamentals

The four essential building blocks of an application are; Activities, Services, Content providers and Broadcast receivers.

Component	Description
Activity	Represents a single screen in the user interface. Users implement this by subclassing the Activity class and implementing necessary lifecycle callbacks, see figure 3.2 [2].
Service	A component that runs in the background that is not a user interface. A service is typically started by an Activity component but can be started by other components too.
Content provider	Manages shared application data. This data is saved in the filesystem, SQLite[3] database or other persistent storage location. Through this component other applications can access data to perform queries or make modifications if the content provider allows it. Content providers are accessed via *ContentResolver* objects. Other ways of storing data is using *SharedPreferences* that write a key, value pair to a XML file.
Broadcast receiver	This component responds on system-wide broadcast messages. These messages usually contains information about system changes but applications can also issue broadcasts.

Table 3.1. Android application components

Each application must have an *AndroidManifest.xml* file which provides information about the application to the Android system. This information contains; names for Java packages, declares each component used by the application from table 3.1 and permissions used by the application, which is described more in detail in section 3.1.5. The structure of a Manifest file and its tags is listed in appendix A.

Activities, Services and sending broadcasts are launched using asynchronous messages, Intents. An Intent is an abstract description of an operation to perform. These messages contain information about action and data to operate on.

[4]Stripped down version of original SQL.

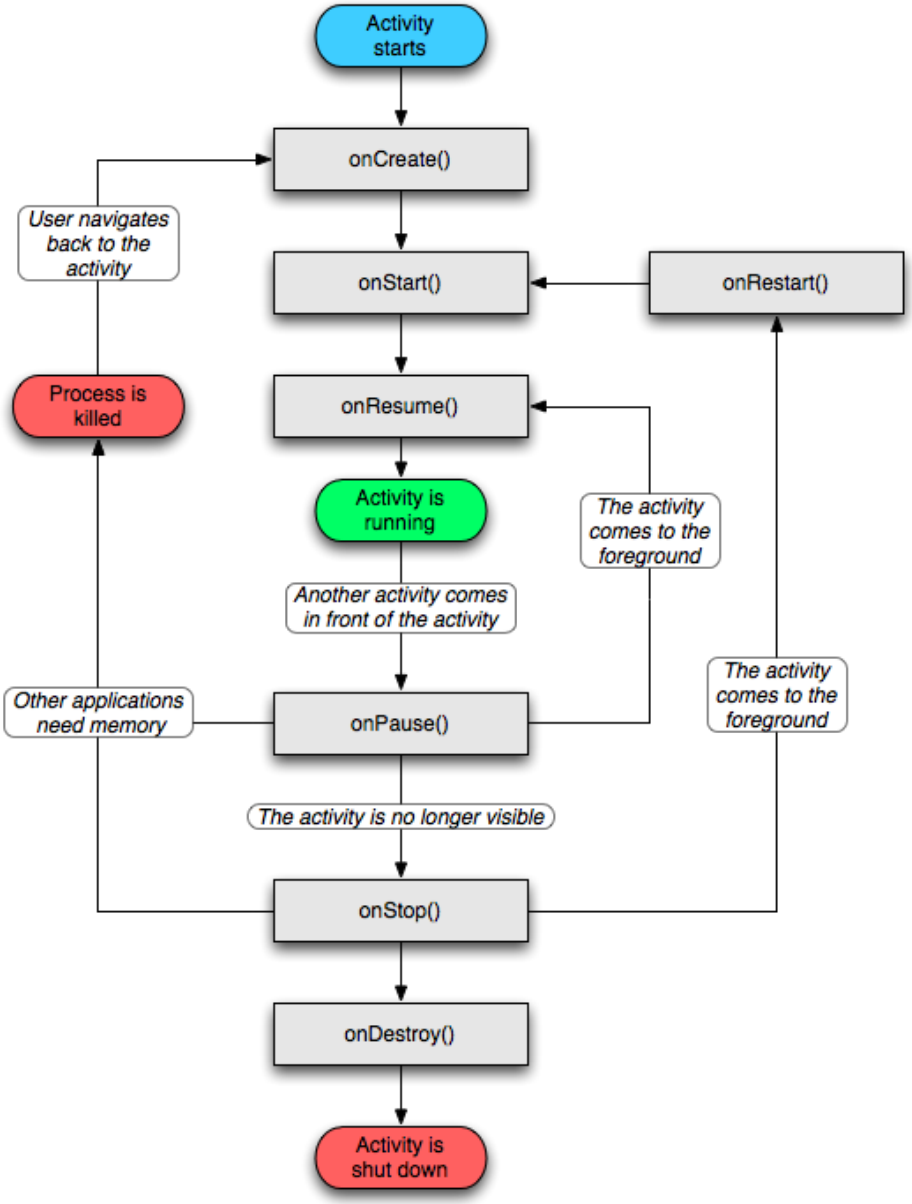

Figure 3.2. Lifecycle of an Activity and its callback methods

3.1.4 Package structure

Each application is compiled into an Android package with the *APK* file extension which is basically a zip archive. This package contains the compiled code, resources and additional data. This single file is considered an application ready to be installed on a device.

Some of the files that are included are AndroidManifest.xml described earlier and *classes.dex* which contain the classes compiled in Dalvik Executable (DEX) format understandable by DVM to run the application.

3.1.5 Permission policies

One of the building blocks of the Android security architecture is that no application, by default has permissions to perform operations that would impact other applications. The kernel separates applications from each other and the applications must therefore share resources and data. To

access certain resources or data the applications must have the correct permissions that are declared by the developer in the AndroidManifest.xml file using the *uses-permission* tag.

Except the official Android permissions [3], applications can declare and enforce their own permissions. This is specified using the *permission* tags in the AndroidManifest. For example, applications can specify a permission that allow others to start one of its Activity. There is also the possibility to restrict who can send broadcasts to a Broadcast receiver and specifying required permissions needed to receive a broadcast.

3.1.6 Dynamically loading and executing code

The SDK provides a class loader functionality using the *DexClassLoader* to execute code that is not installed as part of an application. This is done by loading classes from packages containing the classes.dex file. When successfully loaded, instances of these classes can be created and its methods can be invoked. This provides a way to download packages from a source and injecting payload into the application to extend its functionality. Using DexClassLoader it is also possible to start Activities and Services belonging to an application that has already been installed.

3.1.7 Dalvik Virtual Machine

DVM is a register-based virtual machine that execute the applications on the platform by interpreting the DEX file containing the compiled classes. The generated Java class files are transformed into the DEX file, however this does not contain Java bytecode, but an alternative instruction set used by the DVM. The Dalvik bytecode assigns for example local variables to any of the available register and the opcodes manipulate directly the registers instead of accessing elements on the program stack.

Each application executes within its own DVM interpreter instance where each instance is executed under its own unique UNIX user identities (UID) which is used to isolate applications within the Linux platform. Applications communicate via Parcels, containers for messages that are sent through Binder, an IPC mechanism in the Android system, see figure 3.3 [4].

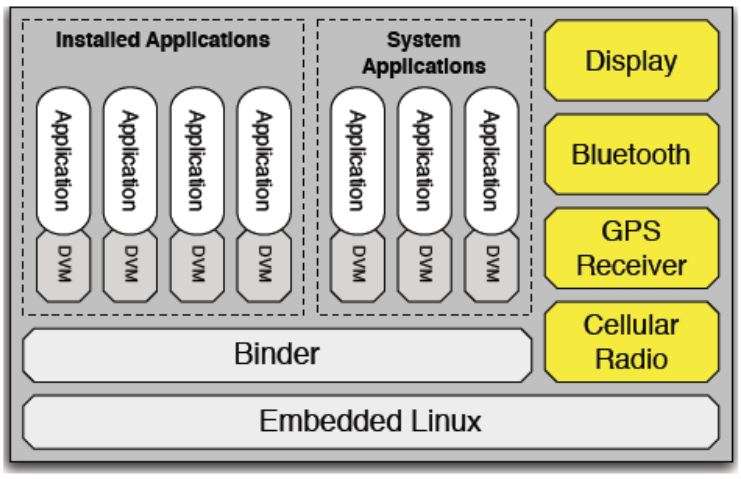

Figure 3.3. DVM and Binder illustration

3.2 Malware

3.2.1 Types

Malicious software is referred to as malware, classified by its nature as either computer virus, trojan horse, worm, backdoor or rootkit. Many more labels are applied for specialized malware, for instance, spyware, scareware, and dishonest adware. Table 3.2 summarizes some of the most common types [6][7].

Malware type	Description
Computer virus	Code that that inserts itself into another program and replicates, that is, copies itself and infects other computers. Nowadays often used as a generic term that also includes worms and trojans horses.
Worm	Self-replicating malware which copies itself to other nodes in a network without user interaction using vulnerabilities. Worms do not attach themselves to an application like a virus do.
Trojan horse	Malicious program which masquerades itself as being an application. Unlike viruses and worms, it does not replicate itself.
Rootkit	Software that enables continued privileged access to a computer while actively hiding its malicious activity from administrators by modifying the operating system functionality.
Backdoor	Specialized trojan horse that masquerades itself as an installed program to enable remote access to a system and bypassing normal authentication. Additionally, backdoors attempts to remain undetected.
Spyware	Software that reveals private information about the user or computer system to eavesdroppers.
Bot	Piece of malware that allows the bot master, i.e. the author to remotely control the infected system. A group of infected systems that are controlled are denoted as *botnets*, instructed by the bot master to perform various malicious activity such as distributed denial of services, stealing private information and sending spam.

Table 3.2. Common malware types

3.2.2 Infection vectors

Methods for which malicious software infects a system are denoted as infection vectors. These methods are categorized as push- and pull-based infection schemes. That is, connections are initiated actively by malicious code and awaits connection from a source, e.g. a client or user, respectively.

The following section describes briefly some vectors commonly used to infect desktop and server platforms.

Exploiting vulnerable network services

Software providing a service may be vulnerable to code exploits. This piece of code exploits programming flaws in the software to remotely execute code on the system. Finding such vulnerabilities that are shared among a large number of software enables for automatic exploitation. Thus, it is the preferred method of infection by worms by actively searching for vulnerable services in a network, making it a push-based scheme.

Drive-by downloads

This vector is most common through web browsers and requires users to visit a malicious web page. The embedded code in the page might exploit a vulnerability in the web browser, allowing it to start downloading malicious code from the web and executing it on the victim's system without

user interaction. This scheme is categorized as a pull-based infection vector since the user must initiate the connection to a web page in the first place.

Social engineering

All actions where a user is lured into executing malicious code on his system is classified as social engineering attacks.

3.2.3 Threats concerning Android

The number of Android devices on the market is increasing and so is the number of users. This attracts malware authors to target Android devices with the intentions of economical profit, stealing private data and infecting devices with botnet-like capabilities. Infection vectors are currently constrained to social engineering attacks, enticing users to download and installing malware on their devices.

Figure 3.1 shows the statistics of captured malware and their types during the period April-June, 2010 [8]. These captures involve malware targeting traditional PC and the Windows platform. Similar statistics representing Android malware, discovered and analyzed during first half of 2011, looks completely different with only two types of malware, see figure 3.4 [9]. The malware portion in these figures represent trojans where premium-rate SMS and botnet functionality is included in the malicious code. Even though the diversity of malware in Android is small, malicious applications are increasing, from 80 to 400 unique applications in the first six months of 2011 [9].

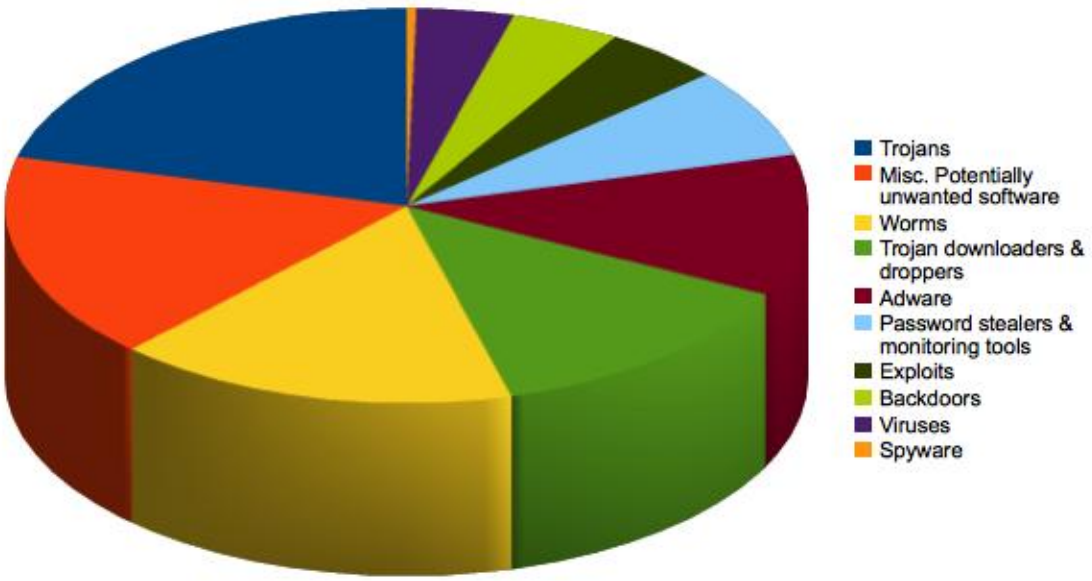

Figure 3.4. Malware types targeting the Windows platform

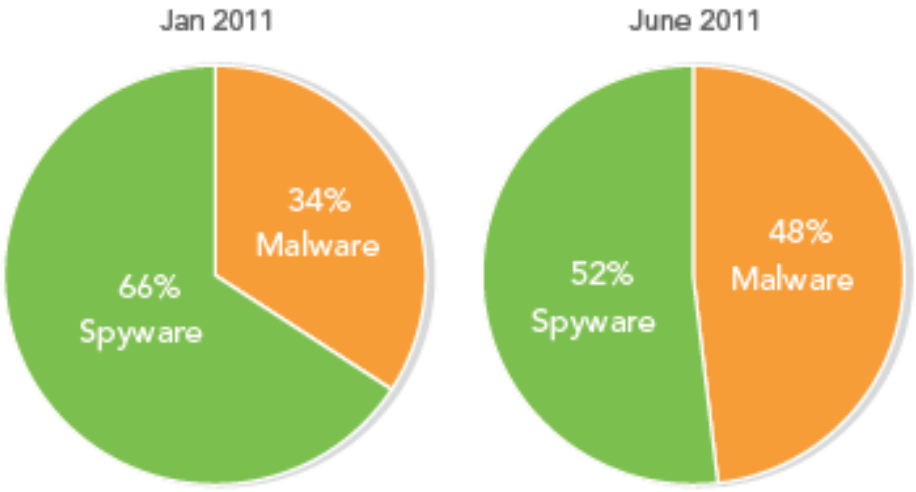

Figure 3.5. Discovered malware types targeting Android

This ongoing threat emerges from the design of the Android system and Google's policy on releasing applications. The design to isolate applications from each other implies that an application cannot steal or tamper with data belonging to another application. However, using the permissions, an application can be granted access to information from other device subsystems, for example, GPS system and database information such as SMS data and contact entries. So it is still possible for a malware application to operate within the isolation and still conduct many different categories of attacks and violations, including resource and data loss attacks.

The Android permission policy may seem robust but the problem is that this approach relies on the user making the decision whether the combination of permissions used by an application is safe or not. Many users may not have the necessary technical knowledge to make such decisions, and sometimes the users are simply lazy to conduct such inspections when downloading applications.

The permission system is not totally secure and this has been shown earlier [15]. An application can perform certain actions without specifying it in the permissions, thus circumventing the policy. These vulnerabilities have been patched by Google instantly but shows that exploitation is possible.

Another issue is that some critical Intents do not require any user interaction to verify the action for these Intents to be successful. For example, Intents to send SMS messages and phone calls can be performed unnoticed when the device is not used and sending SMS can even be performed in the background without any indication of this action.

Google's model for accepting applications to be released in the Android Marketplace follows a very open policy. This means that malware can be distributed easily, compared to iOS applications where a rigorous vetting is conducted. Additionally, applications can be released anywhere on the web. A simple search for $-inurl : htm - inurl : htmlintitle : "indexof" apk$ in Google reveal 1140000 hits with many of the pages in the search result containing directory listings of APK files ready to be downloaded. Legitimate packages can be patched with malicious code (mentioned in chapter 3.3.1) and released on these unofficial markets, for example as pirated packages.

3.3 Malware analysis

The main purpose of analyzing malware is to gain knowledge of its behavior to be able to setup mitigation strategies, this involves performing removals and in the end, to learn how to take down malware spreads. Antivirus software relies on signatures-based approaches to identify malware by searching for known patterns within the binary. This approach works if the patterns matched to

any previously known malware and is grouped into families if it resembles[5] any other malware. Another approach is based on heuristic analysis to detect new malware or variants of known malware.

Analyzing malware is traditionally performed using two different approaches, dynamic and static analysis. With static analysis a study is performed without executing the program. Dynamic analysis of malware cover analysis during runtime where an unknown application is executed and monitored in a secured, virtual environment also known as a sandbox.

There is a constant arms race between malware authors and analysts. Malware authors employ anti-analysis techniques to prevent a sample from being analyzed, these techniques target both dynamic and static analysis. Some of the most common anti-analysis techniques are presented in the next sections.

3.3.1 Static analysis

Analyzing files with this method involves using decompilers, disassemblers and analyzing the source code if available. There exist several tools to perform static analysis on Android packages. Disassemblers[6] for the DEX format can generate assembly-looking instruction shown in listing 3.1.

```
.super Ljava/lang/Object;

.method public static main([Ljava/lang/String;)V
    .registers 2

    sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;

    const-string        v1, "Hello World!"

    invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/
        String;)V

    return-void
.end method
```

Listing 3.1. Disassembled DEX format

Other tools such as Androguard [11], have features to perform diffing of applications, visualization of call graphs, reading the AndroidManifest[7] file inside a package and using assembler tools, packages can be patched with custom code. Several tools also exist for decompiling DEX format to reconstruct Java source code. Decompilers do not reconstruct the original code, but much less intelligible code which can pose problems interpreting it. Original code can also be obfuscated using tools like ProGuard to obstruct reverse engineering.

Malware targeting desktop and server platform have a much more complex obfuscation where self-modifying code and packers are used. A packer obfuscates or encrypts the original binary code and stores the data in a new executable. An unpacker routine deobfuscates or decrypts the data into original representation. This restoration takes place in memory which prevents leaking unpacked binary to disk. Shortly after unpacking, a jump or call is occurred to the address in the memory where the unpacked data resides and control is handed over to the code containing malicious functionality. This approach can be extended with polymorphic variants of a given binary

[5] If a slight variation of already known malware is encountered.
[6] Tools like smali and dedexer.
[7] Stored as binary XML format inside the package.

which can be created automatically using random encryption keys. So called metamorphic variants can in contrast to polymorphic binaries also mutate the unpacking routine.

3.3.2 Dynamic analysis

Analysts prefer to analyze a sample in a safe environment where a potential malware would not cause any damage on a workstation. This also allows for easy clean up of the operating system to perform new sample analysis. Several public sandboxes exist that provide an interface to submit samples for analysis, e.g. binary files and documents such as PDF. The sandbox record changes to the file system, registry keys, incoming/outgoing network traffic and API traces, making the results available in a formatted report. The methods to detect these operations can vary, but the most common are:

API hooking: a method to monitor or alter the behavior of an operating system or application by intercepting function calls. Hooks are usually inserted during runtime but can also be employed before execution. Using runtime modification, hooks are inserted at runtime. Physical modification can achieve hooking with binary rewriting or by modifying the API to monitor function calls before execution.

Difference-based: by taking snapshots of the file system or registry, before and after analysis, a comparison is made of these snapshots to locate modifications.

Notification: Notification routines are registered and automatically called by the system when certain events occur, such as creating directories or deleting files.

There are several pros and cons for each of these methods. Usually, the difference-based approach is the easiest to implement since no modifications have to be made to the API, but does not provide much detailed information. There is also no way to see changes in real-time or temporal order since this analysis is performed after executing the sample. Hook-based method typically provides the most verbose reports since the hooks have access to the arguments and return values of monitored API functions [12].

One issue with using virtualization lies in the implementation of the virtual machines as these reveal information that an operating system is executed in a virtual environment, for example an address in a memory register that could differ in a virtual machine compared to a non-virtual. Malware authors usually try to detect if the sample is being executed on a virtual machine, and if so, they terminate their program. While malware authors discover new ways of fingerprinting and revealing an emulator environment, analysts perform countermeasures against anti-analysis techniques.

3.3.3 Information-flow tracking

The goal of information-flow tracking is to track interesting data as it propagates through a system. One approach as how to accomplish this is dynamic taint analysis [13]. Data that is tracked is marked as *tainted* with a corresponding label, *tag* to distinguish tainted and clean data. Whenever this tainted data is processed, the taint tag is propagated. For example, an assignment statement propagate the taint tag of the source operand to the target. Such propagation logic must be decided beforehand to handle the propagation of tainted data correctly.

```
// x,y tainted
a = y + x;
```

Another important issue is to decide on the propagation policy, i.e. how to handle the taints when encountering statements as the one above. If y and x have two distinct tags, either one of the tags is chosen over the other or creating a new tag by merging the two tags. The bookkeeping of taint

tags is generally done by storing the tags in memory or internal tag maps. Another approach is writing the taint tags to file.

Two important concepts are *taint sources* and *taint sinks*. A taint source introduces a taint tag in the system and the sink is a component in the system that reacts in a specific way when encountering tainted data.

Depending on the scenario and application that should be analyzed, this method can be implemented on different levels. For interpreted languages such as Java, this instrumentation can be added to the instruction interpreter or compiler. Information flow tracking can also be added at a binary level and as a hardware implementation.

TaintDroid [14] is a research project aimed at detecting data leakage using dynamic taint analysis on Android devices. Taint sources are employed at various places in the Android framework, for example when an application retrieves the IMEI value of the device. This retrieved variable is assigned a tag describing what type of data is tainted. When this tainted data is about to leave the device, a taint sink reacts on the tainted data, emitting an alert.

The inner-workings of TaintDroid relies on an instrumented DVM and modified Binder IPC library, see figure 3.6 [14] . Multiple taint markings are merged into one taint tag and are stored adjacent to the tracked data in the memory. When data is retrieved from a taint source it is assigned a taint tag in the DVM virtual map (1, 2). This virtual map is used to assign and retrieve taint tags for specific variables using a mapping function. The DVM is instrumented with a tainted propagation logic to track tainted data whenever an application uses the data (3). To preserve the taint when an application communicates with other applications, taint tags are specified in Parcels and propagated through the system via Binder (4,5). The modified Binder parses the taint tags from the Parcel message and invokes the receiving application's DVM instance to store the tag in its taint map and track the tainted data using the propagation logic (6, 7). Whenever an application calls some library in the Android framework, for example socket write operations to transmitt tainted data (8), the taint tag is retrieved (9) and the application is reported as untrusted. Sinks and sources are interface library functions of TaintDroid that are added at various output channels and sources in the framework. The functions call native code to interact with the DVM instance, retrieving and assigning taint tags.

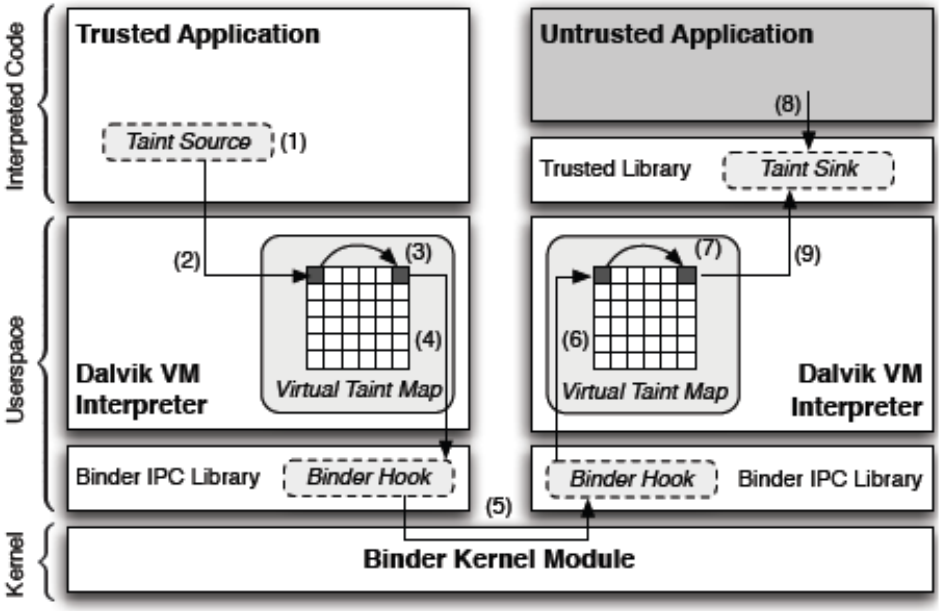

Figure 3.6. TaintDroid inner-workings

The tracked variable types in TaintDroid are method local variables, method arguments, class static fields, class instance fields, and arrays. For each of these types, a 32-bit bitvector stores the taint tag, allowing 32 simultaneous taint markings. Local method variables and arguments are stored in an internal stack by the DVM where each register is 32-bits. To allocate taint storage for these variables, the DVM was instrumented to double the size of the internal stack frame, thus taint tags are interleaved between the local variables and arguments and are accessed by multiplying the variable's register address by 2. For static fields, arrays and class fields the tags are stored adjacent to these inside the interpreters internal data structures.

Chapter 4

Methodology

This thesis started out by investigating an implementation strategy concerning the discussion in Chapter 2. Section 4.1 describes how this problem was approached and the rest of this chapter describes these strategies that were chosen in more detail regarding the theory and implementation.

The available development hardware was a Mac workstation running VirtualBox to deploy virtual machines. The modifications to the Android framework were carried out on a virtual machine running Linux to build the kernel and the Android system images which was not feasible on a Mac.

During the thesis work, test-cases were implemented to test the impact of the modifications made to the Android framework. These tests were made of an actual application containing malicious proof of concept code performing various operations with the purpose of triggering the emulator to broadcast logs originating from the monitored API methods and when an information leak was encountered. The test-cases were implemented in Eclipse on the Mac workstation as well as all host OS code which was implemented in Python. Checking for correct behavior of the modified framework was done by comparing the output from the emulator to the expected output.

4.1 Sandbox system

One approach as how to detect specific operations performed by an application is to intercept system calls generated by the application at the low level in the kernel space. This is achieved by monitoring the system calls that go through a hijacked syscall function and redirecting the arguments to the original syscall function to preserve functionality. The hijacked syscall function for which all calls go through could be implemented as a Linux kernel module. This approach has been tested in Android Application Sandbox (AASandbox) [16] for detecting suspicious software. However, this approach only stores the frequency of called syscall functions during sample analysis. The conclusion on whether the sample is a suspicious software comes only from this dataset and from a static analysis performed prior to running the sample. AASandbox does not log syscall arguments due to instability of the kernel causing a crash, related to resource limitations if too many logs are outputted from the kernel module.

In the thesis description a much more detailed information on the operations is desired, therefore the difference-based approach for dynamic analysis is also not considered. Information regarding the operations is crucial for this project, for example, to be able to retrieve the actual data sent on a socket write operation. This detailed information is also necessary in other scenarios, e.g. detecting leaks and any possible host and port numbers to remote servers an application is communicating with. To detect and log certain operations together with its data, another approach was considered, that is to modify the underlying Android framework which the applications make use of. This involves identifying methods in the framework that are of special interest and logging its arguments and return values, so called API hooking, see section 3.3.2.

To detect an information leak one could mirror the content of the phone's database and at various phone output channels compare the data that is outputted with the phone content. A comparison would involve text string matching and therefore there would have to be training of classifiers involved, e.g. using SVM[1]. However this approach could generate false negatives or even false positives and thus a much more precise and fine-grained method was desired.

Using TaintDroid and adding various sinks and taint sources at certain starting points will generate almost no false positives/negatives. This project was a starting point for this thesis and was extended to track more data and adding new sinks as well as modifying existing.

This approach consociates very well with strategy to perform API hooking by modifying the framework as the TaintDroid implementation is also based on a modified Android framework and would thus be considered the best approach regarding the issues discussed in Chapter 2. The TaintDroid implementation at the time the thesis started targeted Android version 2.1, one of the most common versions and compatibility with newer versions was out of the scope in this thesis.

4.1.1 Overview

The process of the analyzing a sample in the sandbox system is outlined in figure 4.1. The main component in this system is the emulator that is running on the host OS with a modified kernel and system image containing functionality to detect data leaks and monitoring API calls. At first, samples go through a static pre-check. When ready to be analyzed, the information extracted from the sample is passed to the monkeyrunner component to start executing the application. The emulator, in this case the guest OS, broadcast logs that are both system-wide and logs that are triggered at various events that occur when the sample perform certain actions. The issue on how to collect logs outside the emulator is solved by parsing logs in the host OS, where the logcat tool is exploited for this purpose.

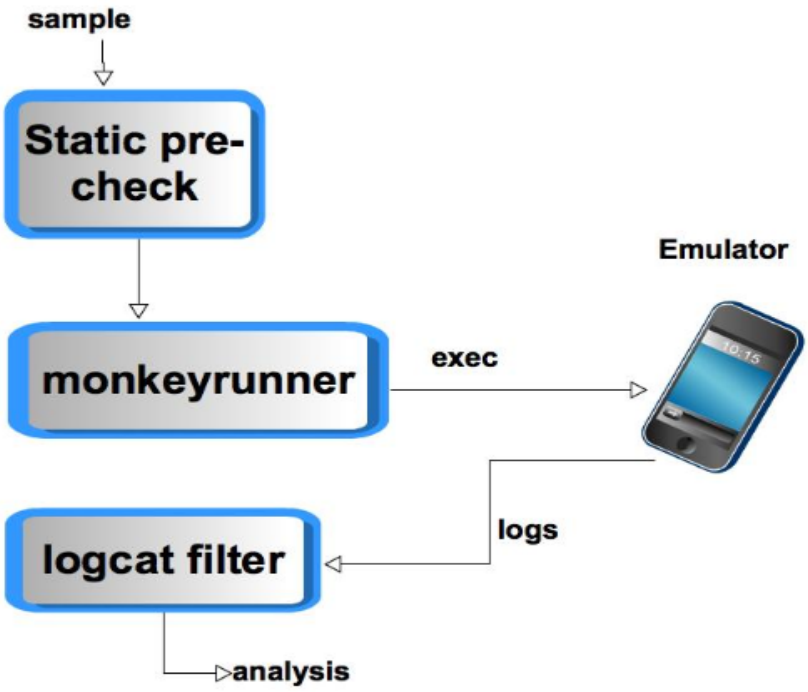

Figure 4.1. Analysis process

[1]SVM stands for Support Vector Machine, a method that analyze data and recognize patterns. This method is used for classification and regression analysis by taking a set of input and predicts for each set which class the data is a member of. In this example the classes would constitute of leak and non-leak data.

4.1.2 Static pre-check

This component performs a minor static analysis before executing the application as each package must be analyzed before performing the runtime analysis. Main reasons for this is that the monkeyrunner script needs certain parameters and the analysis is dependent on analyzing the contents of the AndroidManifest file. More precisely, the static pre-check extracts the Java package and the main Activity from the package's AndroidManifest file. Other data that is relevant for the analysis part is information regarding permissions and registered Intent receivers that are parsed from this file.

As each package is a zip file, it is uncompressed programmatically and the binary format AndroidManifest is read using code from the Androguard project outside the emulator in the host OS. The code in listing 4.1 shows how the unzipping and parsing is performed.

```
fd = open(apkName, "rb")
raw = fd.read()
fd.close()
zip = zipfile.ZipFile(StringIO.StringIO(raw))
for i in zip.namelist() :
  if i == "AndroidManifest.xml":
    buf = AXMLPrinter(zip.read(i)).getBuff()
    xml[i] = minidom.parseString(buf)
    for item in xml[i].getElementsByTagName('manifest'):
      packageNames.append(str(item.getAttribute("package")))
    for item in xml[i].getElementsByTagName('uses-permission'):
      permissions.append(str(item.getAttribute("android:name")))
    for item in xml[i].getElementsByTagName('receiver'):
      recvName = str(item.getAttribute("android:name"))
      recvs.append(recvName)
      for child in item.getElementsByTagName('action'):
        actionName = (str(child.getAttribute("android:name")))
        recvsaction[recvName] = actionName
    for item in xml[i].getElementsByTagName('activity'):
      activityName = str(item.getAttribute("android:name"))
      activities.append(activityName)
      for child in item.getElementsByTagName('action'):
        actionName = str(child.getAttribute("android:name"))
        activityaction[activityName] = actionName
```

Listing 4.1. Unzipping a package and parsing the Manifest

AXMLPrinter is a class implemented to interpret binary XML files and is utilized in the Androguard project. The format of the binary XML file is not well documented, however there exist several tools to perform the decoding. The implementation available in Androguard is a ported Python version that originated from a Java based implementation. The *getBuff* method returns a decoded XML buffer ready to be parsed using the standard library class *minidom*.

For each registered Receiver and Activity the pre-check also parses the *action* tag within these component's *intent-filter* tags. For registered Receivers, this tag specifies what kind of broadcast message the component receives. The Activity component's action tag specifies the Intent action that launches the Activity. The action name for the main launchable Activity is *android.intent.action.MAIN* and this Activity is the first one started when an application is executed.

4.1.3 monkeyrunner script

A sample is installed and executed using a monkeyrunner script. One advantage in using monkeyrunner compared to installing packages through the ADB is that this component automatically install packages and start a specific Activity or Service. Using ADB the user must issue two command line statements and then locate the newly installed application through the emulator UI and execute it. Listing 4.2 shows an example of how monkeyrunner scripts are implemented to automatically install and execute a package.

```
import sys
from com.android.monkeyrunner import MonkeyRunner, MonkeyDevice

device = MonkeyRunner.waitForConnection()
device.installPackage(sys.argv[1])
package = sys.argv[2]
activity = sys.argv[3]

# set the name of the starting component
runComponent = package + '/' + activity
# run the component
device.startActivity(component=runComponent)
```

Listing 4.2. Example of a monkeyrunner script

This code shows that for each package, the Java package name and name of the launchable Activity must be extracted from the sample which has already been performed in the static pre-check. These parameters are supplied through the command line input when this script is executed.

4.1.4 logcat filter

Logs that are broadcasted from the emulator are intercepted using the SDK tool logcat. The interception is achieved by pipelining the logcat output to a Python script that reads from *stdin*. By parsing the logcat output inside the script, using a delimiter and looking for strings containing a static prefix, the relevant logs can be collected and the actual log data can be read. Each sandbox log follows the JSON[2] encoding format. For each type of log that is collected, it is placed in its corresponding list structure, referenced by the timestamp relative to the starting time of the analysis for when the log was collected. These list structures are later used when analyzing the logs and generating the analysis report. Using JSON enables for easy decoding of the JSON format into a dictionary data structure using the Python standard library class *json*. Appendix B shows all types of JSON encoded logs that are collected from the emulator.

Several issues were encountered when choosing JSON as the log format. The Python standard library class for decoding JSON deserializes a string containing the log into a Python object, more precisely a dictionary structure. If the string is encoded with an ASCII based encoding other than UTF-8, the decoding breaks. Other obstructions involved collected data, e.g. HTTP responses that contained quotation marks that defected the JSON encoding. Other data that defected the encoding was path names as these sometimes were returned including trailing binary characters from the sandbox system. This was solved by encoding all *data* and *path* fields into hexadecimal values.

One severe problem was encountered with the logcat output, it seemed to truncate logs that were long. Looking at the log implementation in the kernel source code, revealed that the maximum length of one log was by default set to 4000 characters. This value was increased to 20k and the kernel was recompiled for this changes to have an effect, which solved this truncation problem.

[2]Lightweight text-based open standard for data interchange

4.2 Detecting information leaks

The initial work and issues were related to compiling the kernel and framework to be able to successfully test TaintDroid on an emulator. It is necessary to compile the kernel since TaintDroid implements extended attribute support in the filesystem to store more than one file taint tag at a time. Extended attributes are used to perform association of files with metadata that is not interpreted by the filesystem and adds additional information to a file, in this case the file taint tags that are stored in the extended attributes.

When successfully compiled and tested on an emulator, the next step involved identifying where all data is tainted initially and where the sinks were located in the framework. The implementation that was released by the research team behind TaintDroid was a limited version in the sense of what data that was tracked. The information that was tracked related to SIM card identifiers and device identifiers (IMEI, IMSI, ICC-ID[3] and phone number), location, accelerometer and media (microphone and camera) data. To fulfill the goals described in Chapter 2, several taint tags (see Appendix C), taint sources and sinks needed to be added.

TaintDroid provides an interface library to set a taint at a source and detecting taints within variables. A taint is set using:

```
Taint.addTaintString(mImei, Taint.TAINT_IMEI);
```

at a source in the Android framework where the variable *mImei* is retrieved by an application. This also enables tracking various variable types, for example integers, arrays and strings using different framework methods provided by TaintDroid. These methods also take an argument describing what information is tainted, in this case the constant $TAINT\_IMEI$ is used to indicate that an IMEI value is retrieved by an application. This information is used to verify exactly what type of data is tainted at a sink. Retrieving these taint tags is performed usually at a sink with the statement:

```
int tag = Taint.getTaintString(data)
```

where the integer *tag* will contain all taints within the String variable *data* and a value of zero if no taints are found. Producing a log is performed by calling the function $Taint.log(String)$, a native function that issues a system-wide log containing information found in the argument to this interface function. A prefix is added to the log in the native function providing a delimiter.

Several taint sources were added to track data retrieved from the database using *ContentResolvers*. This data was related to calls, contacts, SMS, browser bookmarks, calendar and email. Additionally, information regarding installed packages retrieved by applications were tainted.

To preserve the functionality to track tainted data that is encrypted using the native cryptography API, taint sinks and sources were added in commonly used algorithms such as DES and AES. A sink was also added in the framework where SMS messages are sent to detect information leaks through this channel.

[3] An international identifier of a SIM card.

4.3 API hooking using physical modification

Further modifications were made to the framework by also adding API hooks in various methods throughout the framework. The most crucial and interesting operations that needed to be monitored covered:

- sending SMS
- performing phone calls
- crypto key negotiations
- opened and closed network connections
- read and write operations on sockets
- accessed files and modifications
- started Services and use of DexClassLoader

For operations related to SMS and phone calls, the hooks logged the SMS message and the receiving numbers. Hooks were also added in the commonly used cryptography functions to detect what key that is negotiated when encrypting and decrypting data.

Applications that start Services can be interesting information to log and so is DexClassLoader. By monitoring started Services and use of DexClassLoader also provides some more characteristics of the application which can be used in the visualization.

Detecting file read and write operations is crucial so low-level file methods in the Android framework were modified to log the method arguments holding the data and logging file paths, file descriptors and operation type. At such low-level there is no way to know what file is being read from or written to based on only the file descriptor within the framework. This is solved by supplying the file descriptor and a random value as arguments to a native function that handles the querying and logging the path together with the random value. The code for querying is shown below.

```
char ppath[20];
char rpath[80];
pid = getpid();
snprintf(ppath, 20, "/proc/%d/fd/%d", pid, fd);
readlink(ppath, rpath, 80);
```

Next step involves logging the data, operation type and the random value from the Android framework. These two logs are then matched in the host OS by comparing the random values.

To get an understanding of the network communication, the low-level network socket methods were monitored to log opened and closed connections together with read and write operations of non-leak data for both TCP and UDP connections. Current implementation of the host OS code buffers the network read data since the API hooks are placed at a low-level in the socket implementations where data is received as byte streams. This buffering is ended when the logcat filter recieves a log with a *CloseNet* tag, indicating that the socket is closed. Matching the socket read data with an opened connection is done by comparing the host, port and file descriptor. When Android HTTP libraries make connections to servers they utilize the *Keep-Alive* header in the HTTP request, which forces the Android system to keep the sockets opened for reuse even when the HTTP connection is closed. This produces a problem when buffering since there is no way of knowing if all data has been transferred. This was solved by modifying the *Properties* class that hold all system properties to set the property Keep-Alive to false by default and hijacking the *setProperty* method. Applications that call the setProperty method with the arguments *("http.keepalive", true)* will not have any effect on this property.

All logs outputted from the sandbox system were modified to follow the JSON encoded format to be able to easily decode the logs in the host OS. To output the logs, the TaintDroid interface method *Taint.log(String)* was exploited.

4.4 Preventing evasion techniques

Evasion techniques are possible on the Android emulator due to usage of hard-coded values and thus a package can detect if it is running in an emulator environment. This issue was discovered prior to Honeynet releasing a mobile malware reverse engineering challange [17]. The sample in this challenge uses a check for the hard-coded values and if any of these are encountered, the application terminates. The hard-coded values of interest are shown in listing 4.3.

```
Build.BRAND = "generic"
Build.DEVICE = "generic"
Build.MODEL = "generic"
Build.PRODUCT = "generic"
IMEI = 000000000000000
IMSI = 3126000000
```

Listing 4.3. Hard-coded values in the emulator environment

These values can be retrieved very easily which is showed in the code sample in listing 4.4. This also requires adding the permission *READ_PHONE_STATE* in the AndroidManifest file to be able to read the device and subscriber id.

```
String brand =  Build.BRAND;
String device =  Build.DEVICE;
String model =  Build.MODEL;
String product = Build.PRODUCT;
TelephonyManager tm = (TelephonyManager) getSystemService
                      (Context.TELEPHONY_SERVICE);
String imei = tm.getDeviceId();
String imsi =  tm.getSubscriberId();
```

Listing 4.4. Retrieving hard-coded build and telephony id values

These strings and values are listed in a build file in the Android framework. By modifying these to values targeting a real device, this evasion technique could be prevented. The build file was modified with the values in listing 4.5 to represent a Samsung Galaxy S device, and the framework was compiled with the modified build file. Modifying IMEI and IMSI values was a little bit more complicated. To be able to generate correct IMEI and IMSI numbers, the values must follow a predefined format which could be easily checked by a malicious package, making the evasion technique still possible if correct values are not generated. The format of these values and the solutions to this problem is described in the two following sections.

```
Build.BRAND = "Samsung"
Build.DEVICE = "GT-I9000"
Build.MODEL = "GT-I9000"
Build.PRODUCT = "Galaxy␣S"
```

Listing 4.5. Build values after modifying the hard-coded values

4.4.1 IMEI

To prevent evasion based on the IMEI value, this value needs to be modified and the method in the framework returning this value must be hijacked. The International Mobile Equipment Identity number is used to identify valid devices in a mobile network. This could for example be used to prevent stolen devices to be used on a network. The structure is specified in 3GPP TS 23.003 [18]. This number consist of 14 digits and one check digit (CD) that acts as a checksum. The 8 first digits are known as Type Allocation Code (TAC) that are issued by a central body. The rest 6 digits represent the serial number that uniquely identify each equipment within the TAC. The CD is calculated using the Luhn [19] algorithm and is achieved by doubling every other digit in the IMEI and summing these doubled values with the remaining IMEI digits. The CD is chosen such that the sum should be divisible by 10. Table 4.1 shows an example of this, supposing the TAC and the serial number is 35724204323751.

IMEI	3	5	7	2	4	2	0	4	3	2	3	7	5	1	CD
Double	3	10	7	4	4	4	0	8	3	4	3	14	5	2	CD
Sum	3+(1+0)+7+4+4+4+0+8+3+4+3+(1+4)+5+2 = 53+CD														
CD	(53 + CD) mod 10 = 0 ⇒ CD = 7														
Final IMEI	3	5	7	2	4	2	0	4	3	2	3	7	5	1	7

Table 4.1. IMEI check digit calculation

The TAC value is also crucial in order to generate a valid number. This information identifies brand and model of the device and could reveal an incorrect IMEI, so it is necessary to choose a TAC value matching the Build values discussed in Chapter 4.4. The IMEI value is chosen according to this method to generate a number and the framework was modified to return a new and valid static value when calling the method *getDeviceId* in the *TelephonyManager* class.

4.4.2 IMSI

The method returning the IMSI value must also be modified to prevent evasion techniques based on this value. International Mobile Subscriber Identity is used as a unique identification to a network and is stored in the SIM card [18], however the emulator returns a bogus value. Choosing a valid value is crucial since the digits in IMSI represent the mobile country code and the network code which both can be looked up and verified that they match. The number contains maximum 15 digits, first 3 digits represent the Mobile Country Code (MCC) followed by a 2 or 3 digit Mobile Network Code (MNC), , see table 4.2. The remaining digits store the Mobile Subscriber Identification Number (MSIN) used to identify the mobile subscriber. The issue here lies in that the MNC value is dependent on the MCC so there is no room for improvisation when choosing these values, however the MSIN can be chosen randomly. The changes, similar to the IMEI value affected the static value returned by the *TelephonyManager* class when calling the *getSubsciberId* method.

IMSI	310005123456789	
MCC	310	USA
MNC	005	Verizon Wireless
MCIN	123456789	

Table 4.2. IMSI value

4.5 Sample analysis and visualization

4.5.1 Analysis and report

A taint tag, for example $0x40002$ may contain several tags. To get information about each tags stored within a taint, all possible tags are bitwise *AND*ed with this taint. If the output is nonzero for this tag, the tag is encountered within the taint. In this example an array of

$['TAINT\_CONTACTS','TAINT\_PACKAGE']$

is returned indicating that contact information and a list of installed applications have leaked at some taint sink. To detect any bypassed permissions, each operation is checked against a list of permissions and verified that the permission has been extracted from the AndroidManifest file. The following permissions are included in this check:

- *android.permission.INTERNET* - permission to connect to internet
- *android.permission.SEND_SMS* - permission to send SMS
- *android.permission.CALL_PHONE* - permission to make calls
- *android.permission.RECEIVE_SMS* - permission to receive incoming SMS
- *android.permission.READ_CONTACTS* - permission to read contact data and call log
- *android.permission.READ_SMS* - permission to read SMS data
- *android.permission.READ_PHONE_STATE* - permission to read IMEI, IMSI, etc.
- *com.android.browser.permission.READ_HISTORY_BOOKMARKS* - permission to read browser bookmarks

Analyzing bypassed permissions for data retrieved from databases is performed by checking if any of these are encountered within a leak and if a permission exists for retrieving this data. To read the system settings there is no permission needed.

The analysis report is generated from the collected sandbox logs after the analysis is stopped and is written to standard out. Each report contains general information as package name, MD5, SHA-1 and SHA256 hashes of the package. The format of a report entry from the output shown in listing 4.6.

```
[Section]
    [Operation]
        [timestamp] log data
```

Listing 4.6. Format of an entry in the analysis report

The timestamp is relative to the starting time of the analysis. Table 4.1 shows all sections, operations and log data that are generated in an analysis report.

Section	Operation	log data
File	read, write	path, file descriptor
Crypto	key, encrypt, decrypt	key byte buffer, encrypted/decrypted data, algorithm
Network	open, read, write	host, port, data
DexClassLoader	-	path to APK
Started services	-	class name
Broadcast recv	-	action name
Enforced perm	-	permission
Bypassed perm	-	permission
Info leak	-	sink, taint tag and sink specific information
Sent SMS	-	receiver number, message content
Phone call	-	number

Table 4.3. Analysis report content

4.5.2 Treemap

Treemaps display a tree structured graph and its data as nested rectangles. Each branch is assigned a rectangle that is divided into smaller rectangles representing sub-branches. The area of the rectangles is proportional to the dimension of the data in the leaf nodes and the color of each leaf node is used for visual clarity [20]. As seen in figure 4.2 [20], each node has a letter and a dimension.

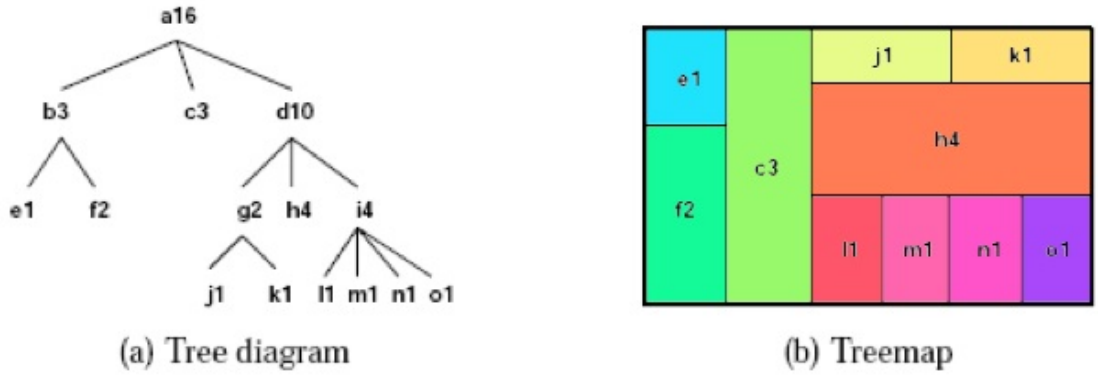

Figure 4.2. Treemap theory

For each node, some static color is applied in the rectangle and the area of the rectangle depends on the size of the dimension. The main motivation to use treemaps is what it was designed to be, a human visualization of tree-structures that can be applied for different dimensions of data. One approach that has been used earlier in sandbox systems is to apply treemaps to malware analysis to be able to classify malicious behavior [21]. This approach relies on assigning API calls belonging to a specific section to a sub-branch where the number of distinct API calls make up the dimension in the nodes. The wider a section becomes (on x-axis), indicates that a sample performs some specific operations more frequently and the height of a rectangle for a given section represents how frequently a specific operation is performed for that section.

Transforming this approach from PC to mobile platform requires specifying new section types and operations. A typical treemap generated is visualized in figure 4.3, where some new types of sections and operations are covered compared to traditional Windows platform. The new ones are sections related to phone calls, SMS and usage of native cryptography libraries. Additionally, in traditional Windows platform, sections such as loaded DLLs and started threads are represented,

but in this case DexClassLoaders and Services are included. These new sections cover operations like information leaks through SMS, network, file and also the different cryptography operations as well as sent SMS.

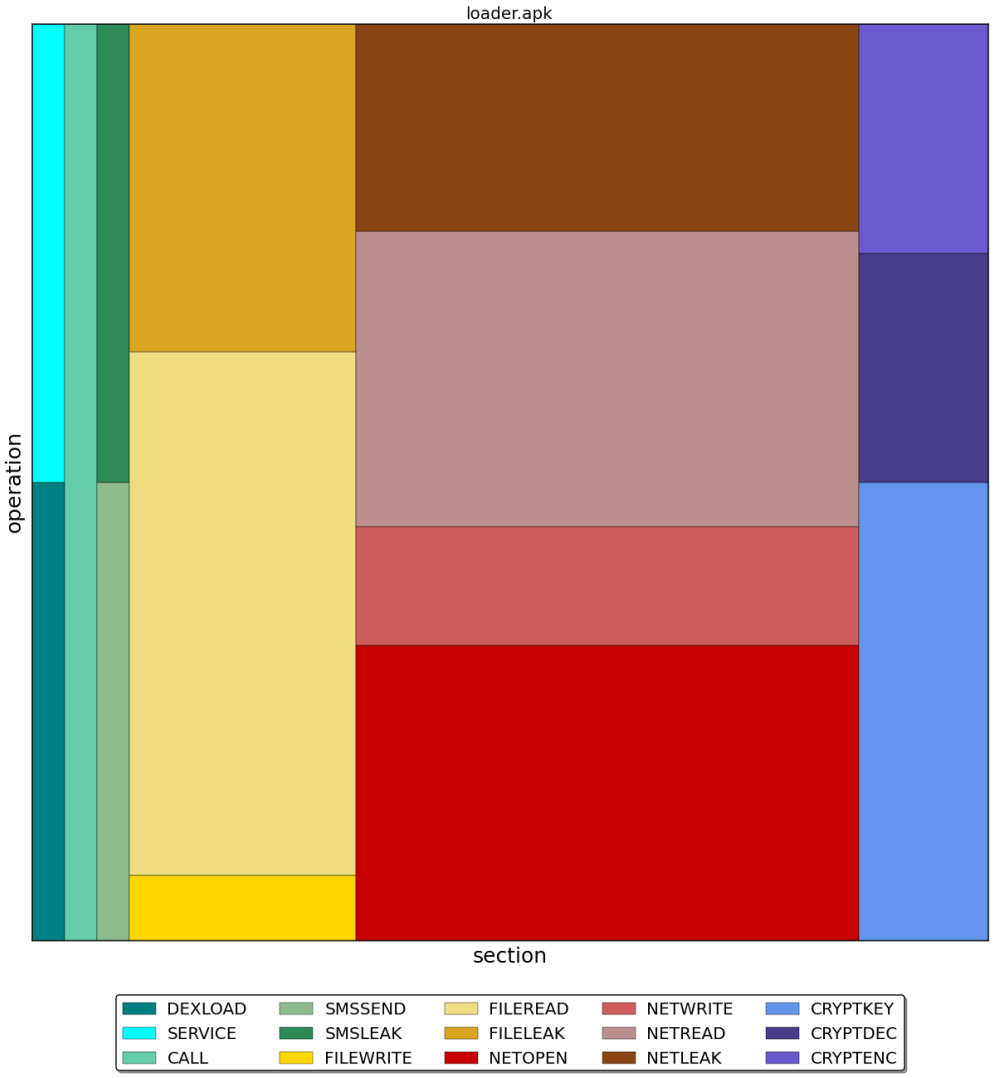

Figure 4.3. Treemap generated by test-cases

For the numeric computation and plotting environment two Python libraries, *PyLab* and *Matplotlib* were utilized. To generate the treemaps, a tree structure was constructed containing all the frequency of the logged operations. The tree-structure is supplied as an argument to the *Treemap* function that calculates the rectangles and draw these by traversing the tree in a iterative fashion. Algorithm 1 [20] shows how this code is implemented in practice.

Algorithm 1 Treemap(root, P[0..1], Q[0..1], axis, color)

```
drawRectangle(P, Q, color)
width ← Q[axis] − P[axis]
for i < root.nbrChildNodes do
    Q[axis] ← P[axis] + (size(child)/size(root)) × width
    Treemap(child[i], P, Q, 1 − axis, color)
    P[axis] ← Q[axis]
end for
```

The function also takes two arrays of length 2, P and Q holding (x,y) coordinates. This array pair represent the opposite corners (upper left and lower right corners) of the current rectangle. The *axis* value varies between 0 and 1 to indicate if cuts should be made vertically or horizontally.

4.5.3 Behavior graph

To get an understanding of in what order operations occur in, a visualization showing the temporal order of the operations [21] was implemented. The main reason for this is the difficulty to interpret the temporal order in a text-based analysis report.

The graph is outlined as follows: on the x-axis the timestamp for the operation is located and on the y-axis, the operation type. Each collected log from the sandbox system has a timestamp relative to the starting time of the analysis. The implementation of this visualization was done by taking the lists containing the collected sandbox logs, merging these into one list and sorting after their timestamp. This list is then iterated and each operation is plotted in the graph. The data point from the visualization can be mapped with the text-based report for more detailed information by comparing the timestamp. Running the test-cases generate a behavior graph as in figure 4.4.

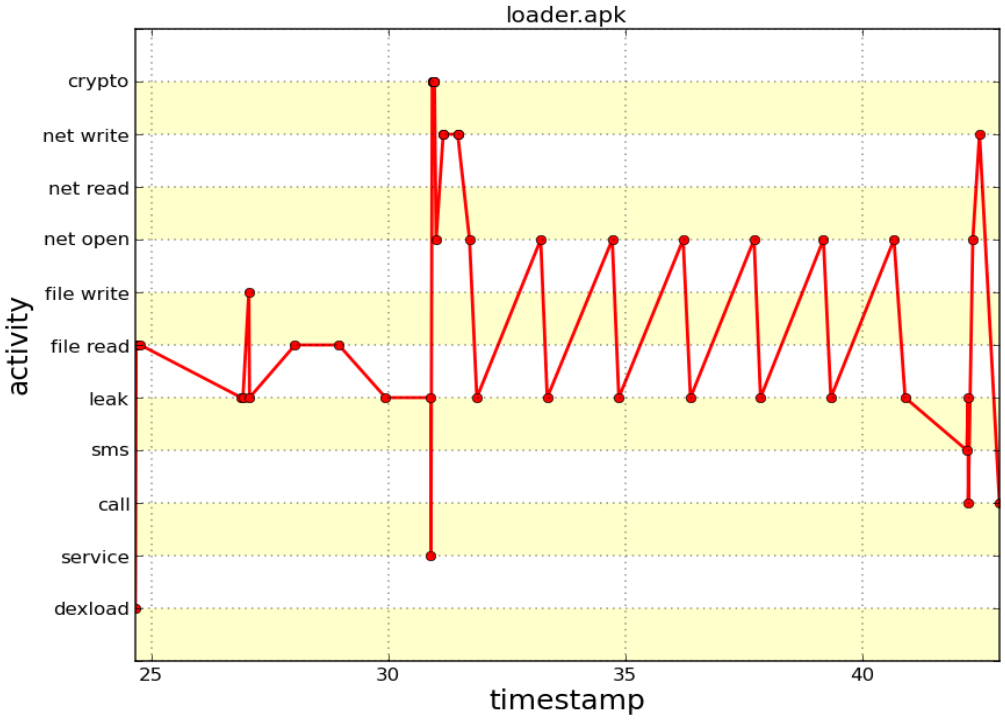

Figure 4.4. Behavior graph generated by test-cases

Chapter 5

Result

5.1 Test-cases

The code to test the framework modification consists of two APK files. The first APK *loader.apk* simply loads a previously installed APK and executes it via DexClassLoader. The second APK *DroidBoxTests.apk* performs the following operations, listed in chronological order:

1. Retrieves IMEI and IMSI values, phone number, contact names, call logs, browser bookmarks and a message stored within a SMS message. Finally, the application retrieves a system setting containing set alarms.
2. Saves the IMSI value and bookmarks using *SharedPreferences*.
3. Retrieves the installed packages
4. Writes a clean string to a file *myfilename.txt* and writes the tainted contact name to *output.txt*. After the write operations, the application reads the contact name and the string just written and concatenates the both strings for later use.
5. Starts a Service named *SendDataService* that sends a HTTP request with a non-tainted string *Hello* to a PHP script.
6. Encrypts the IMEI with AES, directly after the encryption this encrypted IMEI is decrypted. These operations are also performed with the DES algorithm. The DES encrypted IMEI value is stored for later use.
7. Connects and sends a string to an echo server at port 5007 using a TCP socket. Makes also a connection to an echo server to port 50010 and sends a string using UDP and reads the response from these connections.
8. Makes several HTTP connections to send the tainted phone number, the encrypted IMEI value, the SMS message content, data from the file content read in step 4, and sends a list of installed packages. Additionally it sends call logs and system settings. All these connections read the responses from the PHP scripts that data is sent to.
9. Finally, the test application sends an SMS and makes a phone call.

The visualization generated by these test applications are shown in figure 4.3 and 4.4. The analysis report is listed in appendix D where some HTTP headers and data has been truncated. Several bypassed permissions are listed since the Manifest file is parsed from loaded.apk and not DroidBoxTests.apk, both of these package's AndroidManifest files are listed in Appendix E and F. For the same reason as why the bypassed permissions are shown goes for the registered Broadcast receivers and Enforced Permissions from DroidBoxTests.apk AndroidManifest file and why these are *not* shown in the report.

5.2 Real-world malware

The malware that is analyzed is *Gone In 60 Seconds* [22] which is very simple but was one of the samples tested that was actually still fully functional since it was discovered recently. Directly after the package is installed and executed, it starts to upload sensitive data from the device such as contacts, call log, SMS data and browser bookmarks. When the upload is finished, the application asks to be uninstalled. The analysis report is listed in Appendix G and figure 5.1 shows the temporal order graph.

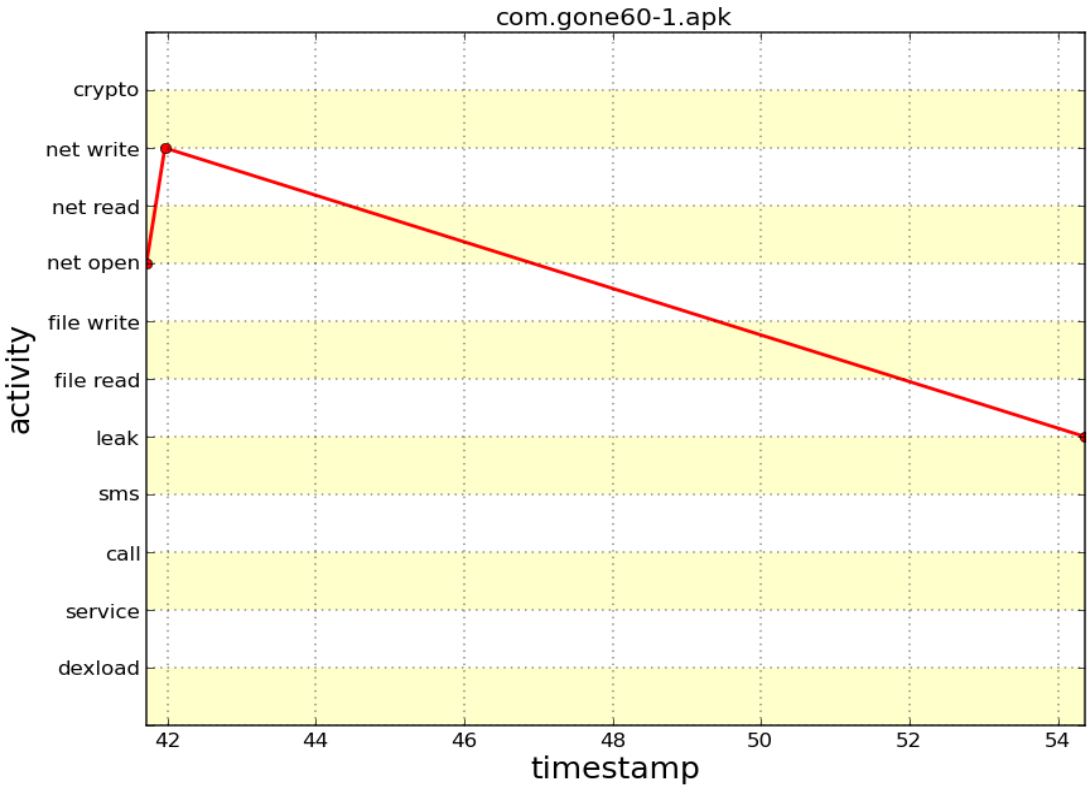

Figure 5.1. Behavior graph generated for real-world malware

5.3 Treemap visualization

The malware family represented in figure 5.2 is *DroidDreamLight* [23][24] which shows a comparison of three distinct malware samples in this family. The visualization has been generated after analysis of each one of these samples.

Figure 5.3 shows a comparison of four distinct samples. The sample on the top left is *Gone In 60 Seconds* and the sample to the right is denoted *GoldDream B* [25]. On the bottom, from left to right are *Geinimi* [26] and *DroidKungFu* [27].

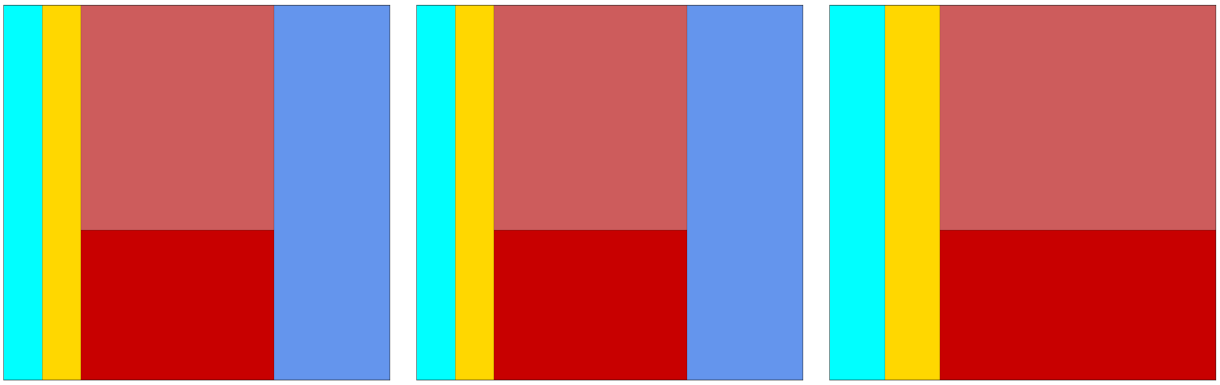

Figure 5.2. Comparison of three distinct samples classified to the same malware family

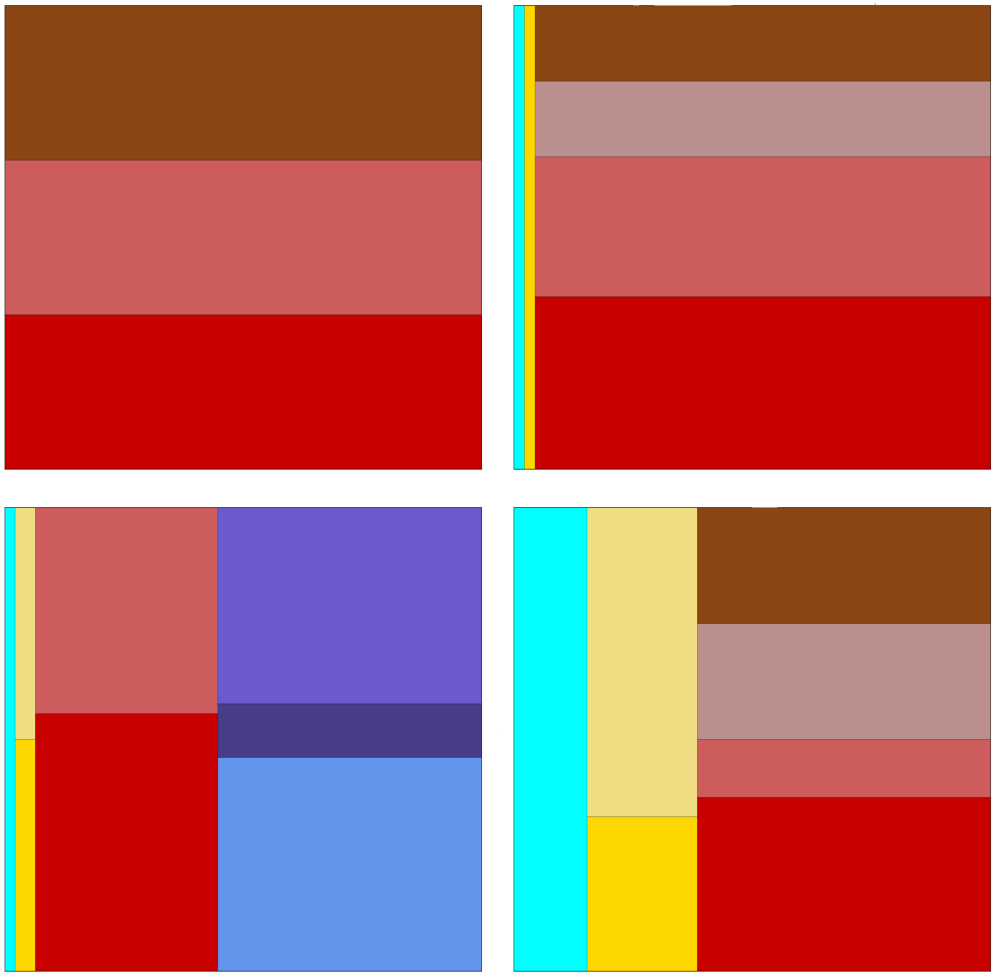

Figure 5.3. Treemaps of four distinct malware

Chapter 6

Discussion

6.1 Framework modifications

Checking if the taint source when retrieving email data is set properly could not be tested. As the email application in Android enforces a permission that only the system or applications signed with the same key can read from the database holding email content, this kind of leak could not be tested. The calendar application and its database access are part of the Google Apps package that is not available on the emulator, so this taint source and calendar data leaks could also not be tested.

Some of the entries in the analysis report suggests that duplicate information is sometimes logged from the sandbox system. This does not pose any problems as the duplicate logs are always issued each time the operation is performed. But, to not confuse analysts why there are for example two phone call entries in the report, this issue has to be fixed.

The changes made to the native cryptography API, regarding taint source, sinks and hooking implementation covered only the default algorithm implementations. The cryptography functions can utilize so called *Providers*, that is different cryptography implementations for all the algorithms which would bypass the taint sinks. This could be prevented by adding taint sinks and sources in the various implementations.

The sandbox system only takes in consideration what a sample performs within the framework using calls to the SDK APIs. Using native code the operations are performed in the Linux system as regular applications at user-level using the Native Development Kit (NDK) [28] and its APIs. Not all operations are possible when executing native code since there are limited APIs available, for example, there exists headers and libraries for interacting with the graphics library and C runtime libraries (libc). A sample could however perform file and socket operations using the NDK and leak sensitive data, if this data is supplied to a native function from Java code.

Regarding performance there is a noticeable slow down in execution time compared to an emulator running with an unmodified framework. The paper describing TaintDroid includes a performance benchmark on the CPU-bound[1] that shows a 14% performance overhead and also demonstrates its capabilities on a real device. The sandbox has a much larger overhead since all operations are logged at the sinks, not only tainted data compared to TaintDroid which only alerts when a sensitive data leak has occurred. The memory consumption is expected to increase since 32 different taints are stored in the interpreter which occupies 4 bytes for each 32-bit variable, however the sandbox does not increase this consumption.

[1]The time it takes to complete a task is determined by looking at the speed of the processor.

6.2 Visualization

The result in Chapter 5 shows that current malware is not yet very complex and do not perform as many operations as the test-cases that were implemented. Many of the samples that were tested simply performed one or two operations and terminated, most common for these were that they sent SMS to premium rate numbers. One must also realize that these samples were not the newest malware found in the wild and any possible leaks to command and control servers may have been omitted in the visualization due to closed or removed servers. To conduct a valuable and trustworthy classification from this visualization it is necessary to analyze fresh samples and storing the figures for future comparison to find a match of possibly repackaged samples. Other obstructions involve hidden malicious functionality that is not triggered during the analysis but rather when certain events occur. This issue is however also applied for traditional sandboxes on the PC platform and the reason why static analysis is important to fully analyze a sample.

Conducting a classification based only on the visualization is not feasible as for example different malwares with the only functionality being to send SMS could match each other. This way of classifying the samples could instead be seen as a preliminary classification. To establish a secure classification it is necessary to compare the numbers and other data from the analysis report and conducting a static analysis. For example, figure 5.2 shows that these samples start a Service, from the report it is noted that they are all named *CoreService* and they open connections to the same remote server. From this information retrieved, these samples could be classified to the same family although one of the visualization shows that one of the sample does not perform any cryptographic operations, which suggests that the other samples are updated versions.

6.3 Problems

Several problems were encountered with the SDK emulator that prevented implementation to perform automated sample analysis. The emulator boot procedure is time-consuming and the snapshot functionality that exists is not very stable to be used in a release. The current implementation is based on user-interactions to boot the emulator and stopping an analysis. The only automated actions in this version are the installation and execution of a package. One could however stop an analysis when enough logs have been collected or when no new logs are collected during a time period but in this way the users can control the analysis by sending text messages, changing GPS coordinates and making phone calls to the sandbox emulator to detect any reactions from the samples.

To restore the emulator to default state it is necessary to delete the emulated SD card and creating a new which currently the user must take care of. This is necessary to perform if applications that have been tested previously are not uninstalled via the emulator interface since some applications are automatically started when the emulator has booted up, generating sandbox logs when Services are spawned.

To detect leaks of user-data it is necessary to populate the databases in the emulator. During the tests, data was manually added via the interface. If this project would be used on a larger scale, an initialization package must be implemented that executes prior to analyzing a sample, populating the different databases with data.

6.4 Future work

Suggestions for major future work cover functionality to monitor native code, discovering further evasion techniques and patching the evasions. Additionally, work to port this project to Android version 2.3.

The last-mentioned suggestion is soon possible as TaintDroid is being ported by the research team behind it to run dynamic taint analysis on version 2.3. The reason why TaintDroid needs

to be ported is that Android versions newer than 2.1 uses a different DVM interpreter since the compilers in the new versions utilize just-in-time (JIT) compilation. JIT is a hybrid of dynamically and pre-compiled interpretation of binaries, while the version 2.1 interprets a pre-compiled DEX format prior to executing it.

The visualization could also be improved by tracing more API function calls that may not necessary be valuable for the analysis report or behavior graph, but for the treemap visualization. In this way more characteristics of the sample can be visualized.

Other important work includes adding taint sinks in SSL sockets that may be utilized by malware authors in the feature. At last, as discussed earlier the framework modifications must be improved to not log duplicate operations for cleaner reports and adding more taint sinks and sources in the various cryptography implementations.

Chapter 7

Conclusion

In this thesis project, an application sandbox system has been implemented by modifying the Android framework and Linux kernel to provide dynamic analysis of unknown packages. By exploiting the SDK tools and implementing host OS code to collect logs from the sandbox system, the operations performed by an unknown Android package can be presented. Furthermore, to help interpreting the analysis results, visualization have been implemented to visualize the temporal order behavior and similarity of packages in terms of their API function calls and behavior. The operations that are of particular interest to detect includes data leakage detection, sent SMS, phone calls, usage of cryptography API and logging file and socket operations.

The test results show that the sandbox system captures these operations in the framework and that visualization can facilitate in order to understand in what order operations occur in and to provide information used to classify malicious samples into malware families. However, relying on the sandbox system by the means as the only analysis technique utilized while analyzing unknown packages will not reveal all malicious functionality within a package but should be considered more as a tool used for initial analysis before performing a deeper, focused static analysis.

An evaluation is also presented, pointing out what features that must be improved and ideas to extend this sandbox system.

Reference

[1] Android Developers (2011)
What is Android?
http://developer.android.com/guide/basics/what-is-android.html (13 Oct. 2011).

[2] Android Developers (2011)
Activity.
http://developer.android.com/reference/android/app/Activity.html (13 Oct. 2011).

[3] Android Developers (2011)
Manifest.permission.
http://developer.android.com/reference/android/Manifest.permission.html (13 Oct. 2011).

[4] Enck W., McDaniel P., Chaudhuri S. (August 2011)
A Study of Android Application Security.
Proceedings of the 20th USENIX Security Symposium.
http://www.enck.org/pubs/enck-sec11.pdf (13 Oct. 2011).

[5] Hashimi S.Y., Komatineni S. (2009)
Pro Android. pp.4-15. New York: Springer-Verlag

[6] D. Gollmann (2006)
Computer Security. 2nd edition pp.258. West-Sussex: Jonh Wiley & Sons

[7] Kaspersky Lab (2011)
Types of known threats.
http://support.kaspersky.com/viruses/common?qid=193238745 (13 Oct. 2011).

[8] Microsoft (2010)
Microsoft Security Intelligence Report.
http://www.microsoft.com/security/sir/default.aspx (13 Oct. 2011).

[9] Lookout Mobile Security (2011)
Lookout: Mobile Threat Report.
https://www.mylookout.com/mobile-threat-report (13 Oct. 2011).

[10] QEMU (2011)
QEMU machine emulator and virtualizer.
http://bellard.org/qemu/ (13 Oct. 2011).

[11] Androguard (2011)
Reverse engineering, Malware and goodware analysis of Android applications ... and more !.
http://code.google.com/p/androguard/ (13 Oct. 2011).

[12] Ligh M.H., Adair S., Hartstein B., Richard M. (2011)
Malware Analyst's Cookbook.
pp.284-285. Indianapolis: Wiley Publishing

[13] Schwartz E.J., Avgerinos T., Brumley D. (2010)
All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask).
Proceedings of the 2010 IEEE Symposium on Security and Privacy.
http://www.ece.cmu.edu/~ejschwar/papers/oakland10.pdf (13 Oct. 2011).

[14] Enck W., Gilbert P., Chun B-C., Cox L.P., Jung J., McDaniel P., Sheth A.N. (October 2010)
TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones.
Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation.
http://www.enck.org/pubs/osdi10-enck.pdf (13 Oct. 2011).

[15] Lineberry A., Richardson D.L., Wyatt T. (2010)
These Aren't the Permissions You're Looking For.
BlackHat USA 2010, DefCon 18 2010.
http://dtors.files.wordpress.com/2010/09/blackhat-2010-final.pdf (13 Oct. 2011).

[16] Blasing T., Batyuk L., Schmidt A-D., Camtepe S.A., Albayrak S. (October 2010)
An Android Application Sandbox System for Suspicious Software Detection.
Malicious and Unwanted Software (MALWARE), 2010 5th International Conference.
http://www.dai-labor.de/fileadmin/Files/Publikationen/Buchdatei/Thomas_AAS_Malware2010.pdf (13 Oct. 2011).

[17] The Honeynet Project (2011)
Forensic Challenge 9 - "Mobile Malware".
http://www.honeynet.org/node/751 (13 Oct. 2011).

[18] 3GPP (2011)
3GPP TS 23.003.
http://www.3gpp.org/ftp/Specs/html-info/23003.htm (13 Oct. 2011).

[19] University of New Brunswick (2002)
Secrets of the LUHN-10 Algorithm - An Error Detection Method.
http://www.ee.unb.ca/tervo/ee4253/luhn.shtml (13 Oct. 2011).

[20] Shneiderman B. (1991)
Tree visualization with Tree-maps: A 2-d space filling approach.
http://hcil.cs.umd.edu/trs/91-03/91-03.html (13 Oct. 2011).

[21] Trinius P., Holz T., Gobel J., Freiling F.C. (October 2009)
Visual Analysis of Malware Behavior Using Treemaps and Thread Graphs.
Visualization for Cyber Security, 2009. VizSec 2009. 6th International Workshop.
http://pi1.informatik.uni-mannheim.de/filepool/publications/THGF_vizsec2009.pdf (13 Oct. 2011).

[22] AVG Mobilation (2011)
Malware information: Gone in 60 Seconds.
http://www.droidsecurity.com/securitycenter/securitypost_20110927.html (13 Oct. 2011).

[23] AVG Mobilation (2011)
Malware information: DroidDreamLight.
http://droidsecurity.appspot.com/securitycenter/securitypost_20110601.html (13 Oct. 2011).

[24] Lookout Mobile Security (2011)
Security Alert: DroidDreamLight, New Malware from the Developers of DroidDream.
http://droidsecurity.appspot.com/securitycenter/securitypost_20110601.html (13 Oct. 2011).

[25] Jiang X. (2011)
Security Alert: New Android Malware GoldDream Found in Alternative App Markets.
http://www.cs.ncsu.edu/faculty/jiang/GoldDream/ (13 Oct. 2011).

[26] Lookout Mobile Security (2011)
Security Alert: Geinimi, Sophisticated New Android Trojan Found in Wild.
http://www.cs.ncsu.edu/faculty/jiang/GoldDream/ (13 Oct. 2011).

[27] AVG Mobilation (2011)
Malware information: DroidKungFu.
http://droidsecurity.appspot.com/securitycenter/securitypost_20110609.html (13 Oct. 2011).

[28] Android Developers (2011)
What is the NDK?
http://developer.android.com/sdk/ndk/overview.html (24 Oct. 2011).

Appendix A

AndroidManifest.xml structure

```
<?xml version="1.0" encoding="utf-8"?>

<manifest>

    <uses-permission />
    <permission />
    <permission-tree />
    <permission-group />
    <instrumentation />
    <uses-sdk />
    <uses-configuration />
    <uses-feature />
    <supports-screens />
    <compatible-screens />
    <supports-gl-texture />

    <application>

        <activity>
            <intent-filter>
                <action />
                <category />
                <data />
            </intent-filter>
            <meta-data />
        </activity>

        <activity-alias>
            <intent-filter> . . . </intent-filter>
            <meta-data />
        </activity-alias>

        <service>
            <intent-filter> . . . </intent-filter>
            <meta-data/>
        </service>

        <receiver>
```

```
            <intent-filter> . . . </intent-filter>
            <meta-data />
        </receiver>

        <provider>
            <grant-uri-permission />
            <meta-data />
        </provider>

        <uses-library />

    </application>

</manifest>
```

Appendix B

JSON encoded logs

```
{
    "FdAccess": { // File access
      "path": "efbfbd7f553432470300340000efbfbd065557...",
      "id": "282345"
    }
}

{
    "FileRW": { // File operation
      "path": "efbfbd7f553432470300340000efbfbd065557...",
      "data": "efbfbd7f557...",
      "operation": "read", // read or write
      "id": "282345"
    }
}

{
    "CryptoUsage": { // negotiating new cryptographic key
      "key": "0,42,2,54,4,45,6,7,65,9,54,11,12,13,60,15",
      "operation": "keyalgo",
      "algorithm": "AES"
    }
}

{
    "CryptoUsage": {  // encryption or decryption
      "data": "000000000000000",
      "operation": "encryption",
      "algorithm": "AES"
    }
}
```

```
{
    "OpenNet": { // opened connection to host
      "destination": "lth.se",
      "port": "80"
    }
}

{
    "SendNet": {  // sending data that does not contain leak
      "destination": "lth.se",
      "port": "80",
      "data": "GET␣/index.html␣HTTP/1.1...."
    }
}

{
    "RecvNet": {  // receive data from socket
      "source": "lth.se",
      "port": "80",
      "data": "efbfbd7f555703000000efbfb..."
    }
}

{
    "CloseNet": {  // socket closed
      "host": "lth.se",
      "port": "80",
    }
}

{
    "DataLeak": {  // data leak via socket
      "sink": "Network",
      "desthost": "pjlantz.com",
      "destport": "80",
      "tag": "0x8",
      "data": "GET␣/phone.php?phone=0735005281"
    }
}

{
    "DataLeak": {  // data leak via SMS
      "sink": "SMS",
      "number": "0735005281",
      "tag": "0x400",
      "data": "dbd4e36bd5295531800c9596724361c4"
    }
}
```

```
{
    "DataLeak": {  // data leak via file write
      "sink": "File",
      "operation": "write", // read or write
      "data": "dbd4e36bd5295531800c9596724361c4"
      "tag": "0x400"
    }
}

{
    "SendSMS": {  // sent SMS
      "number": "0735005281",
      "message": "Sending sms..."
    }
}

{
    "PhoneCall": {  // phone call
      "number": "0735005281"
    }
}

{
    "ServiceStart": {  // started Service
      "name": "droidbox.tests.SendDataService"
    }
}

{
    "DexClassLoader": {  // started APK file
      "path": "/sdcard/DroidBoxTests.apk"
    }
}
```

Appendix C

Taint tags

Tag	Value	Description	Added
TAINT_CLEAR	0x0	No taint	No
TAINT_LOCATION	0x1	Location	No
TAINT_LOCATION_GPS	0x10	GPS coordinates	No
TAINT_ACCELEROMETER	0x100	Data from accelerometer	No
TAINT_ICCID	0x1000	ICCID value	No
TAINT_OTHERDB	0x10000	Other database values	Yes
TAINT_EMAIL	0x100000	Email data	Yes
TAINT_CONTACTS	0x2	Contact name and number	Yes
TAINT_LOCATION_NET	0x20	Network location	No
TAINT_SMS	0x200	Numbers and messages	Yes
TAINT_DEVICE_SN	0x2000	Device serial number	No
TAINT_FILECONTENT	0x20000	Content of a file	Yes
TAINT_CALENDAR	0x200000	Calendar data	Yes
TAINT_MIC	0x4	Data recorded by microphone	No
TAINT_LOCATION_LAST	0x40	Last known location	No
TAINT_IMEI	0x400	IMEI value	No
TAINT_ACCOUNT	0x4000	Google account data	No
TAINT_PACKAGE	0x40000	Installed packages	Yes
TAINT_SETTINGS	0x400000	System settings	Yes
TAINT_PHONE_NUMBER	0x8	Number of the phone	No
TAINT_CAMERA	0x80	Data taken by camera	No
TAINT_IMSI	0x800	IMSI value	No
TAINT_BROWSER	0x8000	Browser bookmarks	Yes
TAINT_CALL_LOG	0x80000	Call history	Yes

Table C.1. Values and description of taint tags. Shows which new tags were added and the tags which functionality was added for

Appendix D

Analysis report for test-cases

```
[Info]
------
File name:  loader.apk
MD5:    811c541850f8367d74b46d6cb5e417f1
SHA1:   871b2b3c783414a23e06ade6096668843bfe9037
SHA256:
   b12f69d2b52bfce7b1feb10594494052bcd8db061db6be166548887891c8cee4
Duration: 47.5297720432s


[File activities]
-----------------

[Read operations]
-----------------
[24.6775109768]
Path: /sdcard/DroidBoxTests.apk
Data: P

[24.6806430817]
Path: /sdcard/DroidBoxTests.apk
Data: K

[24.6869540215]
Path: /sdcard/DroidBoxTests.apk
Data:

[24.6965999603]
Path: /sdcard/DroidBoxTests.apk
Data:

[24.7071151733]
Path: /sdcard/DroidBoxTests.apk
Data: Binary data..

[24.7661750317]
Path: /sdcard/DroidBoxTests.apk
Data: Binary data..
```

```
[28.0287871361]
Path: /data/data/droidbox.tests/files/myfilename.txt
Data: Write a line

[28.9519150257]
Path: /data/data/droidbox.tests/files/myfilename.txt
Data:

[Write operations]
------------------
[27.054803133]
Path: /data/data/droidbox.tests/files/myfilename.txt
Data: Write a line

[Crypto API activities]
-----------------------
[30.9224250317]
Key:{0, 42, 2, 54, 4, 45, 6, 7, 65, 9, 54, 11, 12, 13, 60, 15}
Algorithm: AES

[30.9309520721]
Operation:{encryption} Algorithm: AES
Data:{357242043237517}

[30.9378101826]
Key:{0, 42, 2, 54, 4, 45, 6, 7, 65, 9, 54, 11, 12, 13, 60, 15}
Algorithm: AES

[30.9429590702]
Operation:{decryption} Algorithm: AES
Data:{357242043237517}

[30.9491829872]
Key:{0, 42, 2, 54, 4, 45, 6, 8}
Algorithm: DES

[30.9567849636]
Operation:{encryption} Algorithm: DES
Data:{357242043237517}

[30.961507082]
Key:{0, 42, 2, 54, 4, 45, 6, 8}
Algorithm: DES

[30.9688510895]
Operation:{decryption} Algorithm: DES
Data:{357242043237517}
```

```
[Network activity]
------------------

[Opened connections]
--------------------
[31.0044879913]     Destination: pjlantz.com Port: 50007
[31.4600141048]     Destination: pjlantz.com Port: 50010
[31.7091829777]     Destination: pjlantz.com Port: 80
[33.210103035]      Destination: pjlantz.com Port: 80
[34.7188341618]     Destination: pjlantz.com Port: 80
[36.2142031193]     Destination: pjlantz.com Port: 80
[37.7101781368]     Destination: pjlantz.com Port: 80
[39.1731140614]     Destination: pjlantz.com Port: 80
[40.6604549885]     Destination: pjlantz.com Port: 80
[42.3231761456]     Destination: pjlantz.com Port: 80

[Outgoing traffic]
------------------
[31.1540911198]
Destination: pjlantz.com Port: 50007
Data:

[31.1601109505]
Destination: pjlantz.com Port: 50007
Data: Hello master via TCP

[31.460018158]
Destination: pjlantz.com Port: 50010
Data: Hello master via UDP

[42.4645011425]
Destination: pjlantz.com Port: 80
Data: GET /data.php?data=Hello HTTP/1.1

[Incoming traffic]
------------------
[31.2989680767]
Source: pjlantz.com Port: 50007
Data: Hello master via TCP

[31.6889610291]
Source: pjlantz.com Port: 50010
Data: Hello bot via UDP

[32.002215147]
Source: pjlantz.com Port: 80
Data: HTTP/1.1 200 OK Phone number received!

[33.4717371464]
Source: pjlantz.com Port: 80
Data: HTTP/1.1 200 OK IMEI received!
```

```
[34.9848711491]
Source: pjlantz.com Port: 80
Data: HTTP/1.1 200 OK SMS body received!

[36.5038011074]
Source: pjlantz.com Port: 80
Data: HTTP/1.1 200 OK File content received!

[37.9697849751]
Source: pjlantz.com Port: 80
Data: HTTP/1.1 200 OK Settings received!

[39.4508621693]
Source: pjlantz.com Port: 80
Data: HTTP/1.1 200 OK Call log received!

[41.0233950615]
Source: pjlantz.com Port: 80
Data: HTTP/1.1 200 OK Installed packages received!

[42.5844841003]
Source: pjlantz.com Port: 80
Data: HTTP/1.1 200 OK Non tainted message received!

[DexClassLoader]
------------------
24.6616630554       Path: /sdcard/DroidBoxTests.apk

[Broadcast receivers]
---------------------

[Started services]
------------------
30.8953449726       Class: droidbox.tests.SendDataService

[Enforced permissions]
----------------------

[Permissions bypassed]
----------------------
android.permission.INTERNET
android.permission.SEND_SMS
android.permission.CALL_PHONE
android.permission.READ_CONTACTS
android.permission.READ_PHONE_STATE
android.permission.READ_SMS
android.permission.READ_HISTORY_BOOKMARKS
```

```
[Information leakage]
------------------------

[26.8974089622]
Sink: File
Path: /data/data/droidbox.tests/shared_prefs/Prefs.xml
Operation: read
Tag: TAINT_BROWSER, TAINT_IMSI
Data:
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<string name="Book">http://www.lth.se/</string>
<string name="SharedValue">310005123456789</string>
</map>

[26.9308199883]
Sink: File
Path: /data/data/droidbox.tests/shared_prefs/Prefs.xml
Operation: write
Tag: TAINT_BROWSER, TAINT_IMSI
Data:
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<string name="Book">http://www.lth.se/</string>
<string name="SharedValue">310005123456789</string>
</map>

[27.0697660446]
Sink: File
Path: /data/data/droidbox.tests/files/output.txt
Operation: write
Tag: TAINT_CONTACTS
Data: John Doe

[29.938863039]
Sink: File
Path: /data/data/droidbox.tests/files/output.txt
Operation: read
Tag: TAINT_CONTACTS
Data: John Doe


[30.895277977]
Sink: File
Path: /data/data/droidbox.tests/files/output.txt
Operation: read
Tag: TAINT_CONTACTS
Data:
```

```
[31.8631529808]
Sink: Network
Destination: pjlantz.com
Port: 80
Tag: TAINT_PHONE_NUMBER
Data: GET /phone.php?phone=15555215554 HTTP/1.1

[33.347383976]
Sink: Network
Destination: pjlantz.com
Port: 80
Tag: TAINT_IMEI
Data: GET /imei.php?imei=92a871af351ba747d7789b67f09c817b HTTP/1.1

[34.8577671051]
Sink: Network
Destination: pjlantz.com
Port: 80
Tag: TAINT_SMS
Data: GET /msg.php?msg=Hi,+how+are+you HTTP/1.1

[36.3688721657]
Sink: Network
Destination: pjlantz.com
Port: 80
Tag: TAINT_CONTACTS
Data: GET /file.php?file=Write+a+line&John+Doe HTTP/1.1

[37.8458170891]
Sink: Network
Destination: pjlantz.com
Port: 80
Tag: TAINT_SETTINGS
Data: GET /settings.php?alarmset=Thu+9:00+am HTTP/1.1

[39.32903409]
Sink: Network
Destination: pjlantz.com
Port: 80
Tag: TAINT_CALL_LOG
Data: GET /call.php?logs=null1318421100106+123456789+.. HTTP/1.1

[40.8949520588]
Sink: Network
Destination: pjlantz.com
Port: 80
Tag: TAINT_PACKAGE
Data: GET /app.php?installed=com.android.soundrecorder:.. HTTP/1.1
```

```
[42.2236230373]
Sink: SMS
Number: 0735445281
Tag: TAINT_IMEI
Data: 357242043237517

[Sent SMS]
----------
[42.1967680454]
Number: 0735445281
Message: Sending sms...

[Phone calls]
-------------
[42.2276241779]     Number: 123456789
[42.8857491016]     Number: 123456789
```

Appendix E

loaded.apk Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.loader"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-sdk android:minSdkVersion="7" />
    <uses-permission android:name="android.permission.
       WRITE_EXTERNAL_STORAGE" />
    <application android:icon="@drawable/icon" android:label="@string/
       app_name">
        <activity android:name=".LoaderActivity"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.
                   LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Appendix F

DroidBoxTests.apk Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="droidbox.tests"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-permission android:name="android.permission.SEND_SMS" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.READ_SMS" />
    <uses-permission android:name="android.permission.READ_CONTACTS" />
  <uses-permission android:name="android.permission.CALL_PHONE" />
  <uses-permission android:name="android.permission.READ_PHONE_STATE" /
     >
  <uses-permission android:name="android.permission.RECEIVE_SMS" />
  <uses-permission android:name="com.android.browser.permission.
     READ_HISTORY_BOOKMARKS" />
  <uses-permission android:name="android.permission.READ_CALENDAR"></
     uses-permission>
    <uses-permission android:name="android.permission.WRITE_CALENDAR"><
       /uses-permission>
    <permission android:name="com.me.app.myapp.permission.
       DEADLY_ACTIVITY"
            android:protectionLevel="dangerous" />
    <application android:icon="@drawable/icon" android:label="@string/
       app_name">
        <activity android:name=".DroidBoxTests"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.
                   LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver android:name=".SMSReceiver">
            <intent-filter>
                <action android:name="android.provider.Telephony.
                   SMS_RECEIVED" />
            </intent-filter>
        </receiver>
```

```
        <service android:name=".SendDataService"/>
    </application>
</manifest>
```

Appendix G

Analysis report for real-world malware

```
[Info]
------
File name:  com.gone60-1.apk
MD5:    859cc9082b8475fe6102cd03d1df10e5
SHA1:   e54bbd6754cb23075421daf3021576f5830f8ada
SHA256:   922741596
   dde5081760706c653a7b2bd4634c832b648cd3d06f7edca8ea8d1b7
Duration: 69.5587289333s


[File activities]
-----------------

[Read operations]
-----------------

[Write operations]
------------------

[Crypto API activities]
-----------------------

[Network activity]
------------------

[Opened connections]
--------------------
[41.7149739265]
Destination: gi60s.com Port: 80

[Outgoing traffic]
------------------
[41.9642870426]
Destination: gi60s.com Port: 80
Data: POST /upload.php HTTP/1.1
accept: application/json
Content-Length: 5038
Content-Type: application/x-www-form-urlencoded
```

```
Host: gi60s.com
Connection: Keep-Alive
User-Agent: Apache-HttpClient/UNAVAILABLE (java 1.4)
Expect: 100-Continue

[Incoming traffic]
------------------

[DexClassLoader]
----------------

[Broadcast receivers]
---------------------

[Started services]
------------------

[Enforced permissions]
----------------------

[Permissions bypassed]
----------------------

[Information leakage]
---------------------
[54.363202095]
Sink: Network
Destination: gi60s.com
Port: 80
Tag: TAINT_CONTACTS, TAINT_SMS, TAINT_CALL_LOG, TAINT_BROWSER
Data: code=aac08&data=%7B%22contacts%22%3A%5B%7B+%22name...

[Sent SMS]
----------

[Phone calls]
-------------
```