SEVENTH FRAMEWORK PROGRAMME
Trustworthy ICT

Project Title:

Enhanced Network Security for Seamless Service Provisioning in the Smart Mobile Ecosystem

Grant Agreement No: 317888, Specific Targeted Research Project (STREP)

DELIVERABLE

D2.1 Survey of Smart Mobile Platforms

Deliverable No.		**D2.1**	
Workpackage No.	**WP2**	Workpackage Title	**Development of Virtualized Honeypots for Mobile Devices**
Task No.	**T2.1**	Task Title	**Analysis of Smart Mobile Platforms**
Lead Beneficiary		**TUB**	
Dissemination Level		**PU**	
Nature of Deliverable		**R**	
Delivery Date		**30 April 2013**	
Revision Date		**6 April 2014**	
Status		**F**	
File name		**NEMESYS_Deliverable_D2.1.pdf**	
Project Start Date		**01 November 2012**	
Project Duration		**36 Months**	

Authors List

Author's Name	Partner	Email Address
Lead Author / Editor		
Steffen Liebergeld	TUB	steffen@sec.t-labs.tu-berlin.de
Co-Authors		
Matthias Lange	TUB	mlange@sec.t-labs.tu-berlin.de
Bhargava Shastry	TUB	bshastry@sec.t-labs.tu-berlin.de
Baltatu Madalina	TI	madalina.baltatu@it.telecomitalia.it
Rosalia D'Alessandro	TI	rosalia.dalessandro@telecomitalia.it
David García	HIS	dgarcia@hispasec.com
Laurent Delosières	HIS	ldelosieres@hispasec.com

Reviewers List

Reviewer's Name	Partner	Email Address
Stavros Papadopouplus	CERTH/ITI	spap@iti.gr
Konstantinos Demestichas	COSMOTE	cdemest@cn.ntua.gr

Contents

List of Figures 5

List of Tables 6

1 Introduction 10

1.1 Outline 10

2 Background 12

2.1 Smart Mobile Devices 12
2.2 Smart Mobile Platforms 12
2.3 Virtualization 13
2.3.1 Virtualization Terminology 13
2.3.2 CPU Virtualization 13
2.4 Device Virtualization 14
2.5 Honeypot 14
2.6 Intrusion Detection System 15
2.7 Shadow Honeypot 16
2.8 Scope of this project 16

3 Smart Mobile Platforms 17

3.1 Android 18
3.1.1 Android Architecture 19
3.1.2 Android Security Model 20
3.1.3 Android Patch Cycle 23
3.2 iOS 26
3.2.1 iOS Architecture 26
3.2.2 iOS Security Model 27
3.3 Windows Phone 29
3.3.1 Windows Phone Architecture 29
3.3.2 Windows Phone Security Model 30
3.4 Blackberry OS 30
3.4.1 Blackberry Application Security 30

3.4.2 Blackberry Updates . . . 31
3.5 Other Mobile Platforms . . . 31
3.5.1 Symbian . . . 31
3.5.2 Firefox OS . . . 31
3.5.3 Tizen . . . 31

4 Market Analysis 33

5 Applicability for Mobile Honeypot 36
5.1 Requirements for Mobile Honeypot . . . 36
5.2 Analysis of Smart Mobile Platforms . . . 36
5.2.1 Android . . . 37
5.2.2 iOS . . . 37
5.3 Target Platform for Mobile Honeypot . . . 37

6 Android Security Aspects 39
6.1 General Remarks . . . 39
6.2 Attack Vectors . . . 40
6.2.1 Android and its attack vectors . . . 41
6.2.2 Attack vectors detected . . . 44
6.3 Mitigation Methods . . . 44
6.3.1 Malware Detection . . . 44

7 Conclusion 47

List of Figures

3.1 HTC Dream, the first Android smartphone on the market. 18
3.2 Android Logo 19
3.3 Android system architecture. 20
3.4 Android patch cycle [70] 26
3.5 Tizen Architecture [35]. 32

4.1 Global smartphone shipments at the fourth quarter of 2012. 34
4.2 Mobile malware distribution at the end of 2012. 35

List of Tables

6.1 Malware list 46

Abbreviations

2G	Second Generation
3G	Third Generation
ADB	Android Debug Bridge
AMD	Advanced Micro Devices, Inc.
AOSP	Android Open Source Project
ARM	Advanced RISC Machines Ltd.
BSP	Board Support Package
BYOD	Bring Your Own Device
CPU	Central Processing Unit
CSS	Cascaded Style Sheet
DAC	Discretionary Access Control
DEF	Dalvik Executable Format
DMA	Direct Memory Access
GID	Group IDentifier
HTC	High Tech Computer Corporation
HTML	Hyper Text Markup Language
IMEI	International Mobile Equipment Identity
IP	Internet Protocol
JTAG	Joint Test Action Group
LLB	Low Level Bootloader
MAC	Mandatory Access Control
MMS	Multimedia Messaging Service
MVP	Mobile Virtualization Platform
MWR	MWR InfoSecurity
NX	No Execute
OK	Open Kernel Labs
PIE	Position Independent Executable
QNX	QNX micorkernel
RAM	Random Access Memory
RIM	Reseach In Motion
ROM	Read Only Memory
ROP	Return Oriented Programming
TI	Texas Instruments Inc.
UDID	Unique Device IDentifier
UID	Unique IDentifier
USB	Universal Serial Bus
XD	eXecute Disable
XN	eXecute Never
XNU	X is Not Unix

Abstract

Smart mobile devices–smartphones and tablets–are ubiquitous tools to manage and access people's online assets everywhere, every time. They store valuable personal information, such as passwords, emails, contact information and photos. With this wealth of valuable information, smart mobile devices became the target of attackers. With smart mobile devices entering the corporate domain with BYOD, it is of utmost importance to ensure the confidentiality, integrity and availability of the information on the devices.

Enacting countermeasures against attacks precludes knowledge about current threats. Up to now we do not have a systematic tool to collect threat intelligence.

In this report we aim to do the first step in creating of such a tool: Determine the target platform and the attack vectors that need to be covered. To that end we survey the current smart mobile platforms in terms of their architecture, security measures and openness. We follow up with a market analysis to determine the most important mobile platform today. Finally we do a thorough analysis on known threats to that platform.

1 Introduction

Smart mobile devices–smartphones and tablets–are of ever increasing importance for our daily life. They provide us with virtually ubiquitous access to our digital life. Be it online calendars that help us to get organized, be it our email accounts or messagers and social networks, with smart mobile devices we can use them everytime at every place.

In their role of being our digital hub, these devices became very valuable targets for malware. On the one hand they store valuable personal information such as the users contacts and access credentials for their email accounts, and on the other hand they are lacking security measures, such as frequent security updates and anti-virus software, that are being used on traditional desktop computers. In fact, smart mobile devices are being targeted by a lot of malware.

As smart mobile devices are now being used in corporate environments where confidentiality of information is even more important, their security becomes a very valuable asset. Mobile devices sport more communication channels than traditional desktop computers, each of which is a new potential attack vector.

It is up to the smart mobile device's operating system, the *smart mobile platform*, to ensure the confidentiality, integrity and availability of the user's private data. In this report we identify the most important smart mobile platforms, and analyze them with regard to their architecture, their security measures and vulnerabilities. We also analyze them with regards to the openess of the platform and its applicability to a mobile honeypot.

The goal of this report is twofold: First, we survey a number of smart mobile platforms to make an informed decision of which one we will target in our honeypot. Second, we identify interesting attack vectors that can be covered with the honeypot.

1.1 Outline

This report is structured as follows. We start with defining all the nomenclature that is needed for work package 2 in chapter 2. We put special emphasis on a definition for *smart mobile platforms*, and introduce the term *honeypot*. In chapter 3, we introduce all major smart mobile platforms, detail their architecture and security measures. In chapter 4, we have a detailed look at the current distribution of smart mobile platforms in the market.

Using this information, we define the prerequisites for our mobile honeypot and analyze the applicability of the most popular smart mobile platforms in chapter 5. In chapter 6, we dive into the security of Android to clarify existing attack vectors that can be covered with the honeypot. In particular, we describe Android's security architecture as well as known weaknesses, review scientific literature and explain existing attack vectors and give a short overview of existing malware. We conclude our report in chapter 7.

2 Background

In this chapter we introduce the concepts that are needed for the understanding of this report.

2.1 Smart Mobile Devices

For the scope of this report, smart mobile devices are smartphones and tablets. We exclude tablets that are being used like personal computers with a keyboard and a mouse, because these are usually operated by a traditional desktop operating system. An example is Windows 8, which forms a smart mobile platform while being used with touch input in the METRO environment, but clearly forms a desktop environment when the desktop mode is used.

Today's smart mobile devices are powered by system on a chips (SoCs). In the majority of SoCs there are ARM cores[1]. The devices have limited memory, usually 512MB up to 2GB maximum. They sport a large set of sensors, for example gyroscopes, accelerometers, barometers and light sensors. Smart mobile devices are equipped with WiFi, optionally Bluetooth and in the case of smartphones and some tablets also with basebands. Most smart mobile devices have one or two cameras as well. Most importantly, smart mobile devices are battery powered.

2.2 Smart Mobile Platforms

We define smart mobile platforms to be the operating systems of today's smart mobile devices. As such, they sport a touchscreen interface, are trimmed for mobile use (optimize battery runtime, react to sensor readings, account for embedded CPU and limited memory), and allow for the installation of third-party applications. Typical examples are iOS, Android and Windows Phone 7 and 8.

[1] SoCs based on x86 cores have a very limited market penetration. In Germany for example, only the Motorola Razor i sports an x86 based CPU.

2.3 Virtualization

For the scope of this report we refer to *virtualization* as the act of running an Operating System (OS) in a Virtual Machine (VM).

2.3.1 Virtualization Terminology

VM: A VM is an encapsulated environment that has its own (virtual) resources (CPU, memory and devices).

Guest OS: The guest OS if the OS that runs inside of a VM.

Host OS: The host OS runs the VMs. To that end, all VM resources are under full control of the host OS. The host OS is also known as the *virtualization layer* or *hypervisor*.

Virtualization can be classified into two categories:

Faithful virtualization establishes a VM that is identical to an existing physical machine. Therefore such a VM can run an OS that runs on the physical machine in an unmodified form. Faithful virtualization requires emulation of devices, which is very costly in terms of performance, also known as virtualization overhead.

Paravirtualization presents the guest OS with custom virtualization friendly devices, which eases the burden of the virtualization overhead

2.3.2 CPU Virtualization

The CPUs of today's smart mobile devices are usually ARM Cortex-A8 or Cortex-A9. These CPUs are not natively virtualizable[2]. This means that to implement virtual machines on these CPUs requires either binary rewriting, emulation or rehosting.

Binary rewriting allows to virtualize unmodified operating systems. It is rather complex to implement and has a significant performance overhead.

Emulation has a large performance overhead, even larger than binary rewriting. This overhead is prohibitive on today's smart mobile devices. Emulation can virtualize unmodified guest operating systems.

[2] Upcoming Cortex-A15 based designs sport hardware virtualization capabilities

Rehosting is the port of an OS kernel from the machine interface to the API of another OS. Examples are user-mode Linux [49] and L4Linux [16]. It requires modifications to the guest operating system, which is only possible with source code access. Rehosting can achieve good performance on non virtualizable CPUs.

2.4 Device Virtualization

For the guest OS to be able to leverage the full functionality of devices in the mobile device, the devices need to be virtualized. That is, we need drivers in the virtualization layer (back ends) and some way for the guest OS to access them.

Device virtualization can be done either with *paravirtualization* or with *faithful virtualization*. On today's smart mobile devices, hardware devices do not implement multiple personalities, and are therefore not virtualizable in hardware. Therefore faithful virtualization requires *emulation*. Emulation however introduces way too much overhead into the system, and is therefore not feasible.

Paravirtualization instead consists of custom virtualization friendly device models, which require custom drivers in the guest. Custom drivers require an access to the API of the guest system and a way to add the drivers to the system.

To implement back end device drivers, we need either full documentation of the target smart mobile device, or a working Linux driver.

2.5 Honeypot

Honeypots are a systematic tool in information security defence. For the scope of this work we will use the following definition:

A honeypot is a security resource whose value lies in being probed, attacked, or compromised. [13]

Honeypots are standalone systems that are used solely for attack detection and serve no productive use. That is, any traffic that reaches the honeypot is by definition not legitimate. The same argument goes for any outgoing traffic. Honeypots have been used extensively in computer networks to [65]:

- distract attackers from more valuable machines on a network.
- provide early warning about new attack and exploitation trends.
- allow in-depth examination of adversaries during and after exploitation of a honeypot.

Honeypots are passive entities. Honeyclients instead actively connect to other systems to check if they are trying to infect them.

Literature knows two major types of honeypots:

High-interaction honeypots present the full system, including the monitored services and the environment around, e.g. the full operating system and the set of services it comes with. This type of honeypot is often implemented with virtual machines.

Low-interaction honeypots implement and monitor a single service of the real system.

Honeypots can also be categorized for their intended use:

Production honeypots are being used to detect major attacks on networks. They gather exactly the data which is needed for response. They usually have relatively low overhead, and the data they produce is relatively scarce.

Research honeypots gather all data they can, and are being used to understand the kind of threats that are in the wild. Usually research honeypots have a very high overhead, collect large amounts of data and require manual forensics to understand the threats.

2.6 Intrusion Detection System

Anomaly detection refers to the problem of finding patterns in data that do not conform to expected behavior. [56]

Anomaly detection is closely related to intrusion detection.

Intrusion detection is the process of monitoring the events occurring in a computer system or network, analyzing them for signs of security problems. [51]

Intrusion Detection Systems(IDSs) are software or hardware systems that automate the process of monitoring the events occurring in a computer system or network, analyzing them for signs of security problems. [52]

In contrast to a honeypot, an IDS may be deployed directly on the production system. IDSs are prone to false positives as well as false negatives. False positives can occur because the rules for intrusion detection are too strict. False negatives instead occur when the rules are too weak. Fine tuning an IDS to have low false positives and false negatives, can be difficult. Honeypots do not have this problem as any traffic going in and out is by definition illegal.

2.7 Shadow Honeypot

A rather novel type of honeypot is called *shadow honeypot* [50]. The shadow honeypot is a hybrid of an honeypot, a production system and an anomaly detection system. The production system and its shadow honeypot share their internal state. The shadow honeypot is instrumented to detect possible attacks. All incoming traffic is subject to anomaly detection. Any abnormal traffic is sent to the shadow honeypot, where it is being processed. If the traffic is part of an attack, the event is logged, and the internal state changes of the honeypot are discarded. Legitimate traffic sent to the shadow honeypot will be validated and handled correctly. The shadow honeypot will also instruct the anomaly detection system of false positives, which can be used for fine-tuning.

2.8 Scope of this project

In this project we aim for a high-interaction honeypot. Its purpose is to gather as much information about attacks as possible to enable forensics, and to give us new insights into the threats our smartphones face. In particular, we aim at getting to know the new attack vectors that smartphones offer.

To that end, we strive to run the target smart mobile platform in a virtual machine. This will allow us to do snapshots of the whole system, and to control its communication with the outside, which is needed to contain attacks.

3 Smart Mobile Platforms

Roadmap

In this chapter, we survey several smartphone platforms that are either available in the market or are going to be introduced as of April 2013. The marketized platforms include Google's Android, Apple's iOS, Blackberry, Microsoft's Windows phone, and Nokia's Symbian platform; the two new entrants being Mozilla's Firefox OS, and Samsung's Tizen. The purpose of the survey is to characterize the smartphone platforms with regard to their overall architecture in general and their security model in particular. The survey is presented for two broad classes of platforms:

- Those with a sizable and growing market share: Android, iOS, Windows phone, and Blackberry,
- Those with a low smartphone market share or none at all: Symbian, Tizen, and Firefox.

For the former category, an elaborated comparison between their architectures is provided; for the latter, a brief description of the platforms is provided. By presenting an informative comparison between the security architectures of the surveyed smartphone platforms, this chapter not only serves as a handbook for the security-minded reader but lays the groundwork for selecting the target platform to be chosen for the honeypot prototype in this project.

Our overall criteria for selecting the target platform for honeypot development are the following:

1. High market share, so that findings from the project have a high impact,
2. Availability of platform source code, so that the platform could be instrumented according to honeypot's requirements.

Therefore, in Chapter 4, we evaluate the market share of platforms surveyed in this chapter as of April 2013 and conclude that Google's Android and Apple's iOS are the top two platforms with the most market share. Subsequently, in Chapter 5, we elaborate on other requirements for a honeypot, including the availability of platform source code.

Having filtered out Android and iOS from the list of surveyed smartphone platforms on the basis of their market share, we compare Android and iOS on the basis of their applicability for honeypot development. Such a comparison elaborates on the availability of the source code for the kernel, device drivers among other things. Based on the requirements for a mobile honeypot and comparison between iOS and Android against these requirements, it is concluded in Chapter 5 that Android is the most suitable platform for the development of a smartphone honeypot.

Finally, Chapter 6 delves deep into the selected platform and presents a focused and elaborated description of the platform to be used in the NEMESYS project for honeypot development: Google's Android. The interested reader, who wishes to understand the security aspects of the chosen platform may therefore skip to Chapter 6.

3.1 Android

Figure 3.1: HTC Dream, the first Android smartphone on the market.

Android is an open source platform and application environment for mobile devices. It was announced in November 2007 [11]. It is being developed by the Open Handset Alliance, which consists of Google, Original Equipment Manufacturers(OEMs) like HTC and Motorola, chipset manufacturers like Qualcomm and carriers like T-Mobile. The first device running Android was the T-Mobile G1, which was available to the market

Figure 3.2: Android Logo. The Android robot is reproduced or modified from work created and shared by Google and used according to terms described in the Creative Commons 3.0 Attribution License.

in October 2008 [48] (depicted in Figure 3.1). The logo and mascot of Android is an android robot as depicted in Figure 3.2.

In early 2013, the most recent version of Android is 4.2 *Jelly Bean*, which was released in October 2012. This version is installed on 1.2 percent of Android devices [36]. The two year old Android 2.3 Gingerbread is still installed on about 50 percent of the devices [19].

3.1.1 Android Architecture

The Android architecture is depicted in Figure 3.3. It is based on a custom version of the Linux kernel. To that end, Linux was enhanced with numerous techniques that facilitate Inter-process Communication (IPC), memory management and power management. Each major version of Android comes with its own Linux kernel. That is, older Android userland usually runs on newer kernels, but the same is not true for the opposite direction. Some of the Android code has been merged into Linux staging in version 3.3. This enables a mainline Linux kernel to boot Android userland[6].

The Android userland consists of several layers. The layer that directly interfaces with the kernel is implemented in C and C++, and includes numerous open source

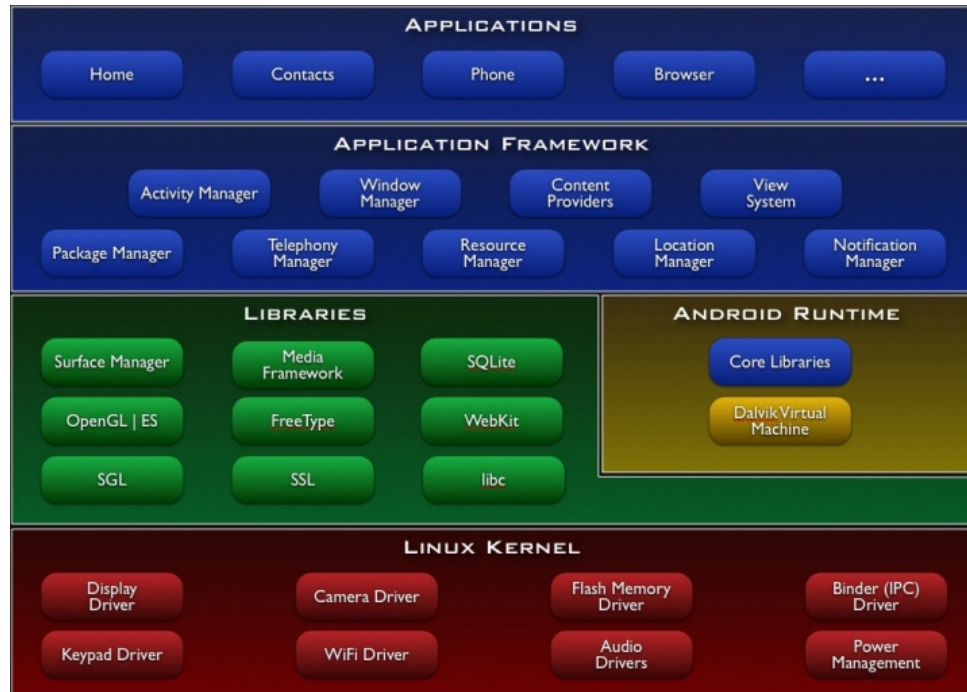

Figure 3.3: Android system architecture.

libraries, such as WebKit, libpng and libsqlite. Android uses a register-based process virtual machine called Dalvik to execute Dalvik Executable Format (DEF) code. Mobile Applications(Apps) for Android are mostly written in Java. Java code is compiled to Java Virtual Machine (JVM) bytecode which is compiled into DEF upon installation on the device.

3.1.2 Android Security Model

The Android security model can be divided into two main parts: Kernel level and application level security. Both aspects will be described in more detail in the following sections.

Linux Kernel Level and System Security

At the very basis of the Android security model is the Linux kernel. It is a very mature kernel that provides the traditional Unix security mechanisms including process isolation via address spaces and file permissions. It also features common Unix abstractions such as users and groups. Linux implements a discretionary access control (DAC) scheme, where each file has an owner. Access permissions are stored with the file (Access control lists) and belong to one of three groups: Owner, Group and Others. In Android, each application runs in its own *sandbox*. In Android, a sandbox is a process, which has its own unique user and group ID, and its own private data storage.

Linux implements a number of IPC mechanisms, for example sockets, signals, message queues and pipes. For Android, Google employed another IPC framework–*Binder*. Binder is a descendant from *OpenBinder* [8], which was developed by Be, Inc for their BeIA and Palm OS Cobalt OSs. Apart from being a communication mechanism, Binder also enforces access control using a form of object capabilities. Communication between applications in Android using Binder is always synchronous [34].

The Android system has several approaches to enhance system security. Its system partition that contains all core services, the application frameworks and the Linux kernel is mounted read-only, which should prevent an attacker from tampering with the core system. On many devices, Android is booted using a secure boot mechanism. On system startup, bootloader and kernel are subject to a check for validity before they are being run. This check can either be a checksum that is compared to a known good version, or a check if the binary was cryptographically signed with a known good private key. In theory, secure boot ensures that the system reaches a known good state after boot. In practice, secure boot is implemented by the OEMs. In many cases the implementation has flaws, or secure boot is omitted entirely to enable custom OSs [68, 58].

Since version 3.0, Android provides an option to have the disk encrypted [4], which is supposed to keep personal data confidential even if the phone is stolen or lost.

Furthermore, Android implements exploit mitigation methods. For example it makes use of hardware extensions that mark memory pages of the stack and heap as non-executable [7]. Such hardware extensions exist for modern x86 systems in the form of the NX (no-execute) bit[1], and on ARM since ARMv6 as XN (eXecute Never) bit [3].

Address Space Layout Randomization (ASLR) is a common technique in modern OSs. It requires that key areas of the layout of a process appear at random locations on each run. Doing so makes it hard for an attacker to guess the location of code and data to use for his malign purposes (e.g. return oriented programming, ROP). Since version 4,

[1] The acronym NX is used throughout literature, however, Intel Inc. uses the term *XD (eXecute Disable)* bit, whereas AMD Inc. markets it as *Enhanced Virus Protection*.

Android supports ASLR [7]. This first implementation did not randomize the application binary and the linker, which weakens the impact of ASLR because it leaves enough code untouched for an attacker to execute on [9]. In version 4.1, Google added support for position independent executables (PIE [7]. Since then also the location of the heap and the linker are randomized. However ASLR is not very effective on systems with 32bit address spaces, as these do not provide enough entropy. However, in combination with the non-executable bit, writing exploits is hard. A downside is that all processes which execute Apps share the address layout with Zygote, because they are forked from Zygote to optimize launch times.

Application Level Security

Apart from system level security measures, Android also implements an application level security system in its application framework. This system implements a mandatory access control (MAC) scheme. The framework defines a set of access permissions to protected APIs, for example SMS/MMS, camera and telephony. Each application defines which permissions it needs to operate in its AndroidManifest.xml file. On installation, the user is asked to either deny or grant these rights to the application. In Android, this decision is binary; Either the user grants the full set of requested permissions, or he cannot install the application at all.

All communication between applications and the system is done via IPC using Binder. In the application API, IPC is abstracted as *Intents*. Applications can send Intents either directly to a named component (explicit Intent) or without naming a component–implicit Intent. In an implicit Intent, the Android runtime system determines if there is a component that can handle the message. If there are more than one possible recipients, it presents the user with a dialog, letting him choose either one. Any application can register as receiver for Intents using *IntentFilters*. These need to be defined in the AndroidManifest.xml file. However, any application can register itself as a BroadCastReceiver for any type of Intent without definition in the AndroidManifest.xml file. Depending on the self-assigned priority it can receive the Intent before any other component. Malware can use this for example to filter or modify SMS before they reach the messages application [70].

Android applications need to be signed with a certificate whose private key belongs to the developer [47]. Whereas the system will not run applications that have not been signed, it allows for self-signed certificates, and does not present the user with any clues about the origin of the certificate [70].

Android applications are distributed using *Google Play*, which is an unmoderated market, where any developer can post applications after paying a small fee. Google

scans the market regularly and removes applications that contain malware [41]. Android allows the user to opt in to allow applications from other sources than Google Play. This is called *side-loading*. As of Android version 4.2, Android includes a malware scanner that scans side-loaded applications for malware. However, this scanner initially showed a detection rate of only 15 percent [27].

3.1.3 Android Patch Cycle

A system is vulnerable during the timeframe starting with the introduction of a vulnerability to a production system until the vulnerability is closed, either through deploying a patch fixing the issue, or other countermeasures, such as the shutdown of the affected service. During this timeframe, the system has a certain risk of being compromised. This risk depends on a number of factors such as the time of exposure, the number and type of vulnerabilities and whether these vulnerabilities are known to attackers. The length of the timeframe in combination with the risk of compromise is called *exposure* [62].

Android depends on a number of open source projects such as the Linux kernel, libsqlite, libpng and WebKit. It is in the nature of open source projects that vulnerabilities are found easily for example through review of the code. When a vulnerability is made public, its implications are discussed publicly through mailinglists and public bug trackers. Therefore any vulnerability in open source software is very well visible to attackers, which increases the system's exposure. Luckily, in many open source projects patches are provided twice as fast as in closed source systems [73]. It is therefore of utmost importance to deploy patches fixing the vulnerability quickly.

Android itself is also open source, but its development model differs from other open source projects. As it is vital for the system's security to keep exposure small, we will now discuss how the Android patch cycle works, and how it affects Android security.

Android Development

Android is developed by Google Inc. behind closed doors. Releases are usually introduced together with fresh Nexus devices. The source code of the Android userland is published to the official Android git repositories[2]. Usually the source code is published with a delay of several weeks after the Nexus device. It is up to Google to decide on source code publication[3]. For example the code of the 3.0 release, named *Honeycomb*

[2]The Android OS source code can be accessed online at `http://source.android.com`.

[3]The Linux kernel is licensed under the terms of the Gnu General Public Licence (GPL), which demands that any derivative work needs to be published in source code as well. This requirement does not hold for the Android userland, so there is no strict obligation to Google to publish the sources.

was not released at all[4].

The official Android releases are a blueprint of what Google envisions Android to look like. In reality, stock Android is only deployed by the Nexus devices. Nexus devices are created by Google together with an OEM. The official source code releases include all code needed to build Android for these devices only (some drivers are available in binary form only). All major OEMs enhance Android with their own *Skin*. A Skin is a customized user interface that includes custom user interfaces and applications. These customizations tend to be very intrusive.

The official Android release needs to be ported to new devices. That is, the OEM has to implement drivers for hardware, such as sensors and peripherals of the device. Due to the Android kernel's licence (GPL) any in-kernel code needs to be published in source code, whereas code running in user space does not have this constraint. Because device logic is often regarded a trade secret, OEMs implement device drivers with very small in-kernel drivers and a complex binary only user space component.

Each firmware is subject to extensive Quality Assurance (QA) on the side of the OEM. If a firmware update would render the phones of customers unusable due to a bug, this would have severe consequences for the public image of the OEM and thereby its future sales.

Usually, smart phones are sold by mobile operators–the *carriers*. Carriers add custom apps and designs to their devices–a process called *branding*. Branding is done to increase the visibility of the brand, and to set it apart from other carriers. Some carriers also modify the firmware to disable features like tethering. The firmware supplied by the OEM together with the addons is subject to extensive QA at the carrier.

The carrier also has to certify hardware and software of each device for correct operation in its network. Depending on the carrier, this process can take three to four months.

Android Patch Cycle

There are four parts of the system that can contain vulnerabilities: the base system containing the kernel and open source libraries, the stock Android runtime including basic services and the Dalvik runtime, the Skin supplied by the OEM and the branding. The skin and branding are usually not open source, which is why bugs contained therein are not that visible to attackers. Skin and branding are implemented in an upper layer of the Android software stack, which limits the potential impact of vulnerabilities therein. Further, both Skin and branding differ from device to device.

[4] The Honeycomb code is included in the official source code repositories, but it does not have a git tag for identification.

The Android base system and runtime however are published with full source code on a regular basis. Any exploit to these systems has catastrophic effects on the smartphone's security. Further, the code contained therein is deployed to all Android based smart phones. That is, bugs have a high visibility and exploits have a large impact. Therefore these layers contribute the most to the system's *exposure*, and it is vital to fix vulnerabilities quickly once they become public.

According to Google Inc, the response to a vulnerability works as follows [7]:

1. *The Android team will notify companies who have signed Non-Disclosure Agreements(NDAs) regarding the problem and begin discussing the solution.*
2. *The owners of code will begin the fix.*
3. *The Android team will fix Android-related security issues.*
4. *When a patch is available, the fix is provided to the NDA companies.*
5. *The Android team will publish the patch in the Android Open Source Project*
6. *OEM/carrier will push an update to customers.*

In practice, updates are very slow to reach the devices, with major updates taking more than 10 months [22]. Many vendors do not patch their devices at all, as the implementation of a patch seems too costly [23]. According to Google Inc.'s own numbers, the most recent version of Android is deployed to only 1.2% of devices [19]. To remedy this problem, Google announced an industry partnership with many OEMs pledging to update their devices for 18 months. This partnership is called the *Android Update Alliance*. However, there has been no mentioning of the alliance since 2012, and updates are still missing [22].

The Android patch cycle is shown in Figure 3.4. Once a vulnerability is found (A), it is being disclosed (B) and there will be a patch by the affected component's author (C). This process is the same for any project. However, for the patch to reach the actual device, in the Android ecosystem some more steps are needed. Google will add the patch to its core system, and publish the code in its official code repositories (D). The OEM will then import the patch in its internal code base (E). This is then sent to the carrier (F), and integrated in their code base. Finally, it is being deployed to the device (G) either via over the air update or as side-loaded firmware image.

Steps D, E and F require extensive QA, which takes a lot of effort and time and is very expensive. The carrier (step F) also has to certify the correct behaviour of the new firmware image in its network, which can take 3 to 4 months to complete.

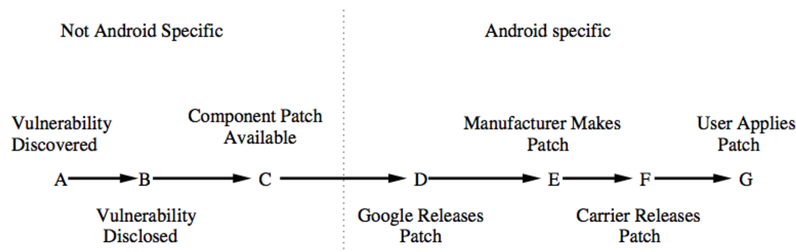

Figure 3.4: Android patch cycle [70]

Updating the devices to new major versions takes additional work from the OEM. Major Android releases usually require a new version of the underlying Linux kernel. This requires the OEM to port their in-kernel drivers to the new kernel version, because in-kernel drivers are usually not sent upstream. Further, major Android updates also have updates to the Application Programming Interface (API). This requires the OEM to update its Skin and the user-level parts of its device drivers. Thus major Android updates incur a high cost at the OEM's side, which is often not feasible from a business perspective.

3.2 iOS

iOS is Apple's mobile OS system running the iPhone, iPad, iPod touch and newer Apple TV devices. The first version of the iPhone OS, which was eventually renamed to iOS later on, was unveiled to the public on January 9th, 2007 together with the introduction of the first iPhone. Since iPhone OS 2.0 Apple allows the installation of third party applications [28].

As of the writing of this report, the current version of iOS is 6.1. Since its release in October 2012 it has grabbed roughly 62 percent of the installed base. The previous iOS release, including all minor revisions accounts for 32 percent. The remaining 6 percent is almost covered by iOS 4. These numbers show a high adoption rate of new OS releases [29].

3.2.1 iOS Architecture

The iOS architecture largely resembles the Mac OS X one. The kernel is mostly the same but compiled for the ARM CPU architecture. Also the userland architecture is

mostly identical with Mac OS X. The launcher and some system libraries have been customized. Instead of loading the Finder the Springboard is loaded on iOS devices.

Lockdownd is a daemon always running on an iOS device. It is in charge of monitoring several activities. It also provides system information to clients such as UDID and IMEI. Lockdownd is running with root privileges and thus has been the target of several jailbreak attempts.

3.2.2 iOS Security Model

Apple tightly integrates hardware and software and thus is able to control and validate all activities from initial boot to application installation. The security model starts with a secure boot chain. The iOS kernel implements security features such as ASLR, supports the NX bit and encryption is used to protect the user data. Third party applications (apps) are reviewed by Apple prior to admitting them to the app store.

The following sections will detail the iOS security features. We will also discuss weaknesses of those mechanisms.

Secure Boot Chain

iOS devices implement a secure boot chain. This starts with some custom code burned into the boot ROM during manufacturing. The boot ROM also contains Apple's public key which is used to verify the following boot stage. The read-only code stored in the boot ROM verifies the low level boot loader (LLB) which in turn is signed by Apple. The LLB then loads and verifies the next stage boot loader called iBoot. iBoot eventually loads and verifies the iOS kernel (XNU).

If one step in the boot process fails to verify the next stage it will stop and the device will switch into a recovery mode. In that case the device can only be restored to a factory default using the iTunes software.

Kernel Security Mechanisms

The iOS kernel implements the execute-never bit of recent ARM architectures. This allows the kernel to mark memory pages as non executable such as for the stack or the heap.

As of iOS 6 the XNU kernel implements kernel ASLR. This means that the kernel image base address is randomized to prevent an attacker from utilizing kernel data located at fixed memory addresses. Second the address of the kernel_map which is used for allocating kernel data structures is randomized too.

Application Security and Sandboxing

iOS only allows the execution of signed code. This makes sure that each application can be accounted to an individual developer or company. Furthermore each application has to pass a review process at Apple before becoming available in the app store.

This review process has been under harsh critic as the rules for accepting or declining a particular application are not completely clear. In fact the review process was justified to prevent malware from becoming available through the app store but in reality it has been shown that this is no guarantee [17]. As a last resort Apple has the ability to remotely delete an application from the device. For that purpose Apple keeps a list of blacklisted applications[5], which is regularly checked by iOS to deactivate apps. So far no incidents have been reported.

When installed on iOS each application runs in its own sandbox. Apple's sandbox implementation is based on TrustedBSD [72]. The sandbox consists of a set of library functions for initializing and configuring the sandbox, a user space server for logging, a kernel extension using TrustedBSD for enforcing policies and a support kernel module which allows regular expression matching of policies. The TrustedBSD layer hooks into the system call layer and forwards system call arguments for pattern matching to the regex matching kernel extension.

Entitlements are another mechanism to protect user information from access by third-party apps. They are used to perform privileged operations such as access to iCloud. Entitlements are key value pairs which are digitally signed.

Data Security

Data security on iOS devices builds on top of hardware security features. Every iOS device has a unique ID (UID) and group ID (GID), which both are fused into the application processor during fabrication. This makes sure that those IDs cannot be tampered. Whereas the UID is unique per device, the GID is unique per SoC revision only. Both keys cannot be accessed by software directly. Only the AES256 crypto engine, which is built into the DMA path between the flash storage and the system main memory has access to those keys.

The UID allows data to be physically tied to one device. Transplanting the memory chips from one device to another doesn't allow deciphering of the content.

Besides the UID and GID all other cryptographic keys are generated using a random number generator. To store such keys and to allow them to be securely eraseable Apple

[5] The list of unauthorized applications can be found online: `https://iphone-services.apple.com/clbl/unauthorizedApps`.

directly accesses the flash storage at a very low level to erase a dedicated small number of block directly (Apple calls this *effaceable storage*). This avoids problems incurred by e.g. wear leveling algorithms.

The encrypted file system contains two separate partitions. One for the OS, the other is the user partition. The OS partition is mounted read-only and unnecessary services like remote login aren't included.

In addition to the storage encryption, iOS provides a per file encryption. This allows the device to encrypt or decrypt files on a per event basis. Upon creation a file is assigned to a certain class, which determines the policy of when the data is available. Also an individual encryption key is created which is wrapped with the class key and stored in the metadata of that file.

A major threat to mobile device security are lost devices. To avoid letting data get into wrong hands, iOS devices can be remotely wiped. This is achieved by deleting the block storage encryption key from the device stored in the effaceable storage. This renders all data unreadable. Additionally devices can be configured to automatically wipe in case of a series of failed passcode attempts.

One technology to search for lost iDevices is a service offered by Apple called *Find My iPhone*. It uses the location capabilities of the device which can be used to track a device. A similar feature is called *Find My Friends*, which locates nearby friends. These functionalities may expose privacy issues as the user has to implicitly trust Apple for not exploiting that data.

Access to the address book, photo library, calendar and location services is protected by a permission that has to be granted by the user to the application. If location is enabled for the camera application each photo carries the location information, potentially even when uploaded to the Internet like social media or cloud services.

3.3 Windows Phone

Windows Phone is Microsoft's successor to its Windows Mobile line of OSs. It was first launched in October 2010. The latest version is Windows Phone 8 which was released to the public in October 2012.

3.3.1 Windows Phone Architecture

Windows Phone is built around the Windows NT kernel. The used kernel version is similar to the version used in Windows 8. The kernel incorporates a hardware BSP to provide drivers for all the built-in devices such as the radio, Wifi and graphics.

3.3.2 Windows Phone Security Model

On Windows Phone each application runs as a least privileged host process in its own sandbox. To get access to certain functions an application needs a capability (access token). Capabilities are granted during install time to trusted applications only. Trusted means that the application has been signed with a digital certificate [18].

Applications for Windows Phone are developed using a managed language which is executed using the .NET Common Language Runtime (CLR). This ensures that the application code is verifiable with regard to type safety, and safe against buffer overflows.

Each application has its private data area that is only visible to that application. Reading or writing from or to other data areas is not possible.

3.4 Blackberry OS

The Blackberry OS is a mobile OS developed by Research in Motion (RIM) for its line of smartphones. Blackberry is popular for its native support of corporate email. Throughout the last years Blackberry was faced with a declining market share. The most recent version of BlackBerry OS has received the Common Criteria EAL4+ certification [15]. Documentation about the BlackBerry OS architecture and security is not published by RIM. Whereas this might add to a perceived increase in security, this *security through obscurity* scheme might hide some serious implementation errors, which could lead to serious vulnerabilities.

As of January 2013, RIM changed its name to BlackBerry [45]. At the same event, BlackBerry introduced a new smartphone platform, which is called BlackBerry 10. It is based on the QNX OS, and was created from the ground up for mobile touch enabled smartphones. It includes a technology allowing to use the phone both privately and in the business, with private and corporate data being properly isolated from one another [37]. Because no devices with BlackBerry 10 are available yet, we do not cover it in this report, but will only have a look at its predecessor.

3.4.1 Blackberry Application Security

Developers have to use the Blackberry API. To access certain functionality of the device the application must be signed. Applications are available through its own app store. They are implemented in Java and run in a custom JVM. The use of Java rules out certain types of vulnerability such as buffer overflows. However, the proprietary JVM might contain serious vulnerabilities in itself.

3.4.2 Blackberry Updates

RIM provides updates to its OS over the air if the mobile carrier supports that. However, the mobile carrier can decide to block over the air updates for its customers.

3.5 Other Mobile Platforms

In this section we will cover other mobile platforms around on the market.

3.5.1 Symbian

Until the end of 2010 Symbian was the most popular mobile platform in terms of market share. Symbian was originally developed by Symbian Ltd. before being transformed into an open-source project under the command of the Symbian Foundation. When Nokia announced in early 2011 that it will switch to Windows Phone the development of Symbian was discontinued.

3.5.2 Firefox OS

Firefox is an open-source mobile OS developed by the Mozilla Foundation. Previously it was called Boot to Gecko (B2G) and it aims at HTML5 applications to interact directly with the device's hardware.

Firefox OS builds on top of the Linux kernel and shares a significant amount of its code base with the Android AOSP project. The user interface is called Gaia and provides a lock screen and some basic applications e.g. for messaging and the camera. Gaia is entirely written in HTML, CSS and Javascript. Gaia and all other applications run using the application runtime Gecko which is a modified version of the Firefox rendering engine. Gecko provides APIs to access the smartphone hardware e.g. sensors and the camera.

3.5.3 Tizen

The architecture of Tizen is depicted in Figure 3.5. Tizen is based on the Linux kernel. It targets Html5 applications in favor of native applications. Its first public release was in January 2012. It is being developed by a technical steering group with Intel and Samsung. The project itself resides within the Linux Foundation. Tizen is the successor of *Meego* [42]. The first phones sporting Tizen are expected to come to market in 2013 [46].

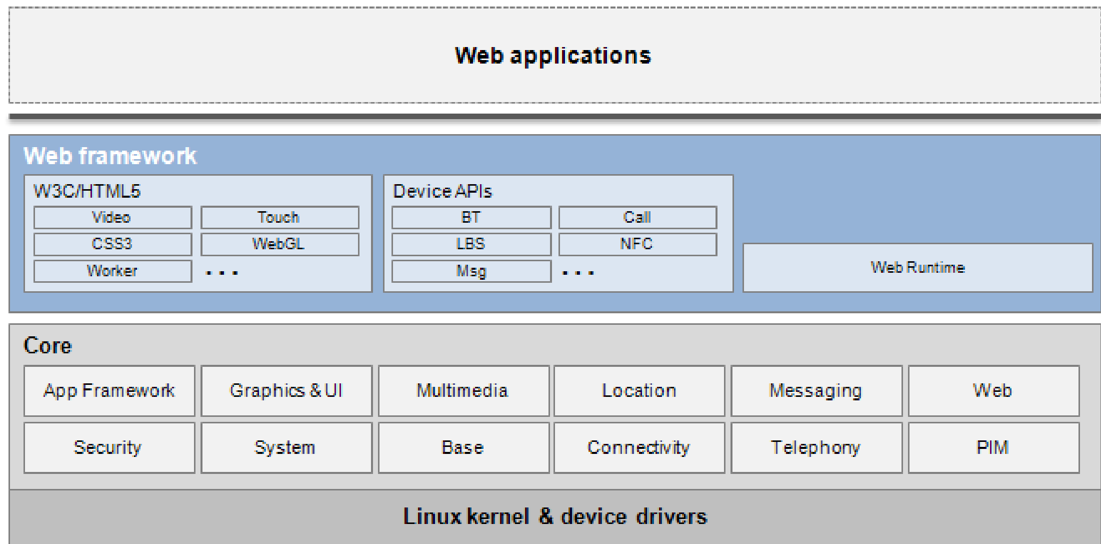

Figure 3.5: Tizen Architecture [35].

As Tizen is yet to come to market, it is most probably not an attractive target for malware. Therefore we will not consider it for the mobile honeypot.

4 Market Analysis

In October 2012, it was announced that, according to the latest research, the number of smartphones in use worldwide surpassed the 1 billion-unit mark for the first time ever in the third quarter of 2012 [10]. Nevertheless, smartphone penetration is still relatively low. Most of the world does not yet own a smartphone and there remains huge room for future growth, particularly in emerging countries such as China, India and the African countries. Also according to [10], it seems that the first billion smartphones in use worldwide took 16 years to reach, but the next billion is forecasted to be achieved in less than two years.

In the same time, it seems that the smartphones are spreading faster than any technology in human history [2] even in developing countries. Smartphones accounted for 36 percent of global mobile phone shipments in the first quarter of 2012, up from 25 percent a year earlier. If smart phones continue to gain at even this pace, feature phones will be largely a memory in another five years.

As far as the mobile operating systems are concerned, taken together Googles Android platform and Apples iOS accounted for a record 92smartphone shipments in the fourth quarter of the last year, according to new figures from the same analyst Strategy Analytics [1]. The analysts estimate that 152.1 million Android smartphones were shipped globally in the quarter, nearly double the amount shipped in the year ago quarter (80.6 million) taking Androids global smartphone share from 51% at the end of 2011 to 70.1% at the end of 2012. Apple iOS holds a 22% of the market share, while other mobile operating systems together arrive to 7.9%. This situation is illustrated in Figure 4.1 below.

Therefore, in 2012 Android has become the most popular mobile OS in the world. And, we might say quite inevitably, this also holds true as far as the malware spreading is concerned. Kaspersky [5] shows that during the last year, Android malware share also grew quite conspicuously, clamorously surpassing the malware targeting the rest of mobile devices. According to the statistics provided by the Kaspersky analysts, during 2012 mobile malware for Android arrived at 98.96% from all mobile malware registered by Kaspersky alone. This situation is illustrated in Figure 4.2.

Since similar results are displayed by other mobile malware research groups around the world, we feel quite confident that that the Nemesys choice to focus on Android as its principal platform for the honeypot scheme development is realistic (it reflects

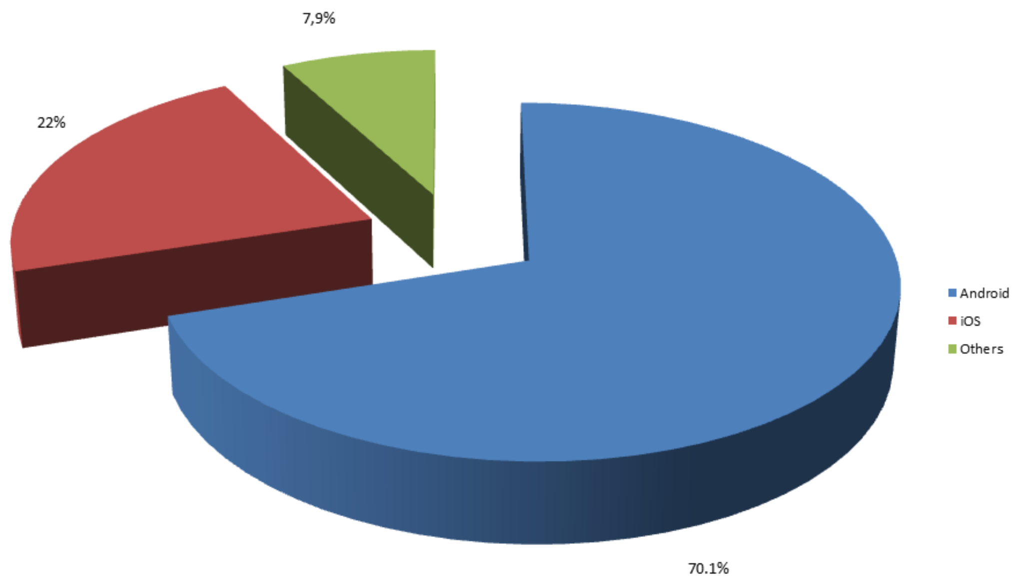

Figure 4.1: Global smartphone shipments at the fourth quarter of 2012.

the real world malware distribution) and, hence, can optimally meet the needs of the mobile users. For a more detailed view and statistics on mobile platforms and malware distribution and classification please refer to Nemesys deliverable D1.1 [69].

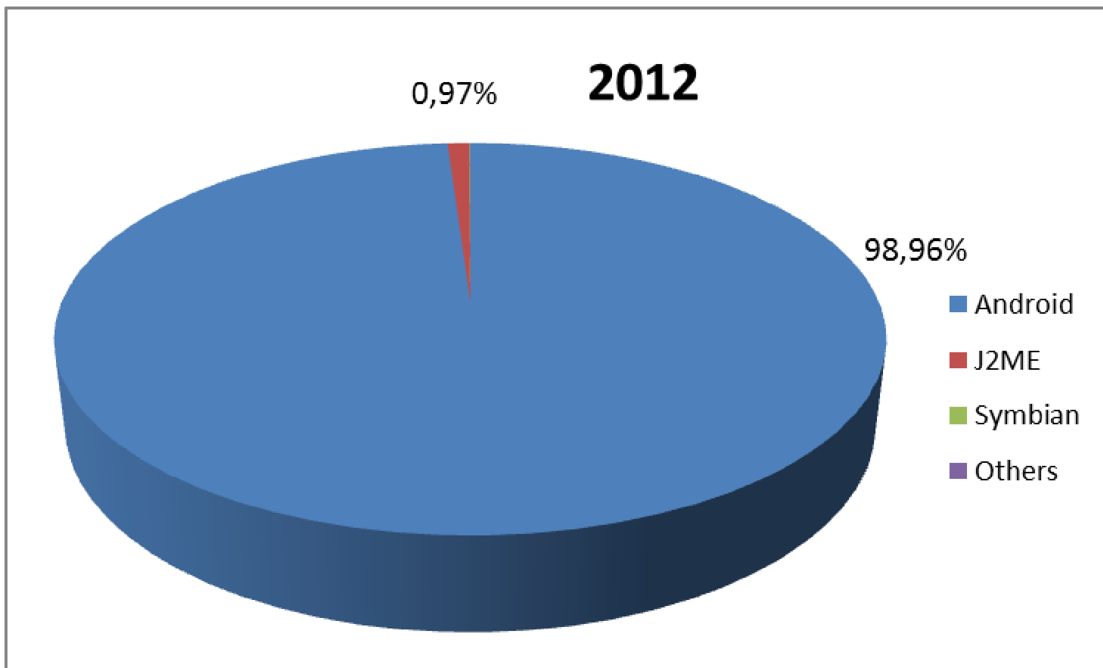

Figure 4.2: Mobile malware distribution at the end of 2012.

5 Applicability for Mobile Honeypot

With our projected mobile honeypot we aim to create a new tool to collect threat information for today's mobile devices. To that end, we plan to use virtualization to contain attacks and to enable snapshots and quick restore.

In this chapter we will first outline the prerequisites for our mobile honeypot. We will then proceed to analyze each of the smart mobile platforms that we characterized in the previous section in terms of their applicability for the mobile honeypot.

5.1 Requirements for Mobile Honeypot

To be able to learn about current threats, we want to catch as much current attacks as possible. To that end, we want to cover the smart mobile platform that is most popular, both in market share and in attractiveness for attackers. If much information about the platform is available then this is a plus because it allows us to more acurately analyze attacks.

Technically, the use of virtualization precludes some requirements on contemporary smartphones. As described in section 2.3.2, current System on a Chip (SoC) are based on the ARM instruction set, which is not virtualizable. This means that virtualization can only be implemented with either emulation, binary rewriting or rehosting. Because of limitations in performance and battery power, only rehosting makes sense on today's smartphones. Rehosting however requires in depth modifications to the OS kernel that is to be virtualized. For that reason we require access to the smart mobile platform kernel's source code. The same is true for device drivers.

5.2 Analysis of Smart Mobile Platforms

As detailled in chapter 4, only two smart mobile platfroms have a significant market share: Android and iOS. We will therefore focus our analysis on these two platforms.

5.2.1 Android

The Android OS is very well suited for the development of a mobile honeypot. Its full source code starting with the kernel is available. Its kernel has been virtualized in many projects, the most prominent being the L4Android project, which also works on systems that are not naively virtualizable [64]. Other notable solutions are VMWare MVP [53] and OK:Android [40].

Android is used on a plethora of different devices, which will enable us to pick a device that has all drivers available.

In-kernel drivers are available, and allow for modification and virtualization as well. The user-level parts of the drivers are not available with source code, but in many cases they can still be used in custom systems.

Source code of Skins and branding of devices that are actually in the field is not available. This can be a problem when faking an actual device.

Because of its open nature, Android has been the system of choice for security research for many years. Therefore it is already well understood, and academic literature is extensive, which can be of great help for the honeypot development.

5.2.2 iOS

Although the iOS kernel is based on the open source project XNU [33] it is not available as open source[1]. Only the sources of the Mac OS X kernel are available. Because we cannot access or modify the iOS kernel sources, we would need to employ binary rewriting or emulation to run iOS in a virtual machine on current smartphones, which come with non-virtualizable CPUs. Further, we cannot reuse the proprietary drivers without implementing device emulation, which is very laborious. Moreover, the iOS userland contains several open source components but large parts of the remaining framework are proprietary. In summary, iOS is not suited for a mobile honeypot.

5.3 Target Platform for Mobile Honeypot

Our analysis shows that Android is the best target for our projected mobile honeypot for the following reasons:

- It is currently the only open platform with a significant deployment. Its open nature, and the availability of its source code both help in understanding the system, and enable modifications, which will be needed for virtualization.

[1] http://theiphonewiki.com/wiki/Kernel#Source_Code

- Android is also the best understood platform in research, with lots of available literature. This will ease understanding the platform.
- Several projects showed that Android can be virtualized on today's smartphone hardware.
- Android is run on many commercial-off-the shelf smartphones, which allows us to pick one that has all the drivers and low-level debug infrastructure and that can be acquired at reasonable prices.
- Android is the most widely used smart mobile platform, and its market share easily surpassed that of iOS. We expect that the Android market share will increase, an assumption that is supported by numbers.
- Most known malware for mobile devices target Android by a large margin.
- We have a good understanding of the attack vectors of Android.
- There is a large library of known Android malware, which can be used to verify the functionality of our mobile honeypot.

6 Android Security Aspects

In this chapter we will give an analysis on the security of the Android OS. We will first give some general remarks about the Android OS, and will then distill attack vectors out of a large library of malware samples.

6.1 General Remarks

Android security is layered, with each layer's security depending on the layer below. The most basic layer is the Linux kernel. On top there is the service layer. The application layer sits on top of the service layer. For our analysis of the Android security model, we start from the bottom layer and work ourselves through to the topmost layer.

The Linux kernel, which forms the bottommost layer, is a monolithic kernel. That is, every kernel component, such as file systems, network protocol stacks and device drivers lives in the same security domain. A minimal configuration consists of about 3 million Source Lines Of Code (SLOC), with the kernels installed on the devices being much bigger. If this code contains any exploitable vulnerability, an attacker can successfully subvert the platform and render its security mechanisms useless. In general, the number of bugs in code seems to be roughly proportional to its complexity [61]. This situation does not seem to improve even when security is in focus [67]. In fact, in a study using static code analysis, Coverity Inc. found 88 critical bugs in a recent version of the Android kernel, some of which are remotely exploitable [57].

The Android application sandbox utilizes Unix users and permissions. As in any Unix system, the user *root* has all permissions to modify the file system and tamper with the kernel (e.g. by loading or unloading kernel modules). If the attacker succeeds to subvert any service running as root, or raising the privilege of a process via a privilege escalation attack, the platform is at his hands. Users choose to *root* their system to be able to fully control their device, for example to enable functionality that was disabled by the network operator, such as tethering.

A recently uncovered vulnerability [32] sheds light on the security implications of the Android driver model. Because of the Linux kernel's license, all in-kernel driver code needs to be published under the terms of the GPL. As driver logic is often considered a trade-secret, many OEMs implement only very small driver stubs in the kernel, and place

a larger binary in userspace that implements all the device logic. In order to communicate with the in-kernel driver stub, these userspace device drivers need a speedy interface. As this is sometimes implemented using shared memory, the userspace device driver, and the kernel interface are additional attack vectors that allow kernel access.

Another set of problems result from the Android permission model. Kelley et al. conducted a study and found that users are generally unable to understand and reason about the permission dialogues presented to them at application installation time [63].

In [54] Barrera et al. conducted an analysis of the Android permission model on a real-world data set of applications from the Android market. It showed that a small number of permissions are used very frequently and the rest is only used occasionally. It also shows the difficulty between having finer or coarser grained permissions. A finer grained model increases complexity and thus has usability impacts. The study also showed that not only users may have difficulties understanding a large set of permissions but also the developers as many over-requesting applications show.

Felt et al. performed a study on how Android permissions are used by Apps. They found that in a set of 940 Apps about one-third are over-privileged, mostly due to the developers being confused about the Android permission system [59].

Google Inc. employs a combination of malware scanner and emulator to detect malware before it enters the Google Play market. This is called *Bouncer*. Bouncer runs a submitted application in an emulator an checks for abnormal or malicious behaviour. Recently researchers showed that it is easy for malware to avoid detection within Bouncer [30].

6.2 Attack Vectors

G. Turbin [60] defines the attack vector as follows: *with respect to computer systems, an attack vector is simply the method used to gain access to the device to deliver a malicious payload (malware). Hackers and fraudsters exploit systems by using attack vectors that most easily allow access to the system. An attack can use one or more vectors. Because defences against attacks are eventually developed, criminals constantly seek to exploit systems through new attack vectors that are more vulnerable.*

Exploits are used for automatically installing malware on the mobile device, crashing the device making it unavailable, or elevating the privileges. An exploit can be used in every layer of a smartphone ranging from the kernel layer to the application layer. By adding more functionality to the device such as Near Field Communication (NFC) for instance, we increase the number of potential exploits and therefore the smartphone might become more vulnerable.

Phishing [44] is an attack aiming to steal private information such as usernames, passwords, and credit card details by mimicking a trusted entity in an electronic communication.

Pharming [43] is an attack which consists in redirecting the user to a fraudulent website which mimics a trusted entity. Generally, such attack is conducted by changing the victim's host file which contains the mapping between a domain name server and its IP address. The host file is used by the Dynamic Name Resolution (DNS) client of the victim when looking up the IP address of a server. In most of the cases, the attacker also changes the mapping between the antivirus companies domain name and their IP address to prevent the victim from testing the trustworthiness of the fraudulent websites.

Application repackaging consists in decompiling a famous application, inserting some malicious payload, re-compiling, and uploading the application to a repository (Google play or 3rd party). When users download this application, they think that the application is the original one and they get infected. Y.Zhou et al. [74] propose a technique to detect the repackaged applications and identify their corresponding malicious payloads.

First, we present a list of potential attack vectors that have been detected in Android system since its creation then in the second part, we enumerate a list of known Android malware on the market with their associated attack vectors.

6.2.1 Android and its attack vectors

There are different types of attack vectors: physical attacks, attacks by Social Engineering, and attacks by exploiting the smartphone ranging from the kernel layer to the applications layer.

Physical attacks

A physical attack consists in accessing physically the device in order to install or remove software. For instance, by means of a USB cable, an attacker can access physically to an Android device and install or remove software even though the device is blocked [71]. For this, Android has a module called Android Debug Bridge (ADB) [20] which enables a developer to get the logs, install, and remove Android applications from his computer. There are two ways of infecting the device either with ADB enabled or not.

In the former case, if ADB is enabled and the attacker has physical access to the victim device, it is possible to plug the USB cable and install a malicious application via "adb" tool by entering the command "adb install package_name.apk" from a computer terminal. Moreover an attacker can exploit the plugged device by uploading a shellcode to the device via the command "adb push shellcode" causing an exploit. Once the

smartphone is exploited, the attacker can install a rootkit and therefore obtain root privileges on the device, i.e. the highest privileges.

In the latter case, it is still possible to infect it via recovery mode. Recovery mode is a mechanism present in almost every Android smartphone so as to repair the phone in case its system image gets corrupted. It consists in dividing the disk in two partitions: one for the system image and another one for the recovery image (factory settings). When the system image gets corrupted, the user can restore the corrupted image with the recovery image. The attacker can use the recovery mode to infect the smartphone. For this, he first needs to build a malicious image and overwrite the recovery image. Then, via a specific hardware key combination (e.g pressing "home" and "volume up" buttons at the same time), the attacker can trigger the recovery mode which overwrites the system image with the recovery image, i.e. the malicious image. However, this attack is not stealthy since all the user data is wiped out. This mechanism is also used to install custom images, e.g. CyanogenMod [38].

Another attack leverages the fact that RAM does not loose its content directly after power off, but that its content fades slowly. The fadeout can be slowed down by cooling the device. In a so called *cold boot attack*, the adversary freezes the device, reboots it into his own firmware, and can then read the memory contents of the previous session. This can be used to obtain cryptographic keys, for example those that are used for disk encryption. Such an attack was recently shown to work [39].

Many devices sport a JTAG interface, which can be used to obtain any memory content at runtime. If the attacker is able to figure out the location of the JTAG interface, and attach his JTAG devices without powering the device off, he can read all memory and access the cryptographic keys within.

Social Engineering

The main attack vector is the Social Engineering i.e. using people naivety to infect mobile devices. It encompasses all the techniques that consist in exploiting the human being such as phishing, pharming, application repackaging, etc. Social engineering does not require an exceeding technical investment, i.e. no new exploit development is needed to infect the smartphone.

Most of people using mobile devices even though aware of the presence of mobile malware, cannot recognize malicious applications. In order to protect the user from installing malware, Google has introduced the notion of permissions for mobile devices. When the application wants to access a resource of the system (e.g., network), it must ask the authorization of the system, i.e., mandatory access control policy. This mechanism prevents malware from using resources whose access rights are prohibited. The rights

are granted by the user upon installing the application. Unfortunately most of people do not check this list or do not understand the real meaning of those permissions and therefore install those applications.

The presence of non free Android applications has enticed the creation of black markets where applications are easier to download, i.e. do not require any authentication and are free of charge. It is the case for instance of the market "market.b3er.org" used by the Android application "Blackmart App" [25] that offers similar services to the application Google Play, i.e. browsing and installing Android applications. However, a lot of those applications have been re-packaged and therefore are malicious.

Exploits

NFC has been shown as being vulnerable and a potential attack vector. Miller et al. [24] have injected and run some malicious code in an Android smartphone with NFC. For this, they have exploited the memory corruption bug of and Android 2.3 Gingerbread to gain the control of the NFC daemon with a specific designed Radio Frequency IDentification (RFID) tag. The most worrying is that NFC is enabled by default for the Android Ice Cream Sandwich and Android Jelly Bean which are the last Android versions.

The MWR Labs hacking team [26] has also exploited the vulnerability of the NFC. They have used a zero day vulnerability on the NFC in order to hack the Samsung Galaxy S3. With this exploit, the team had access to the user data such as e-mails, Short Message Service (SMS), the address book, the photo gallery, and access to third-party application data.

SMS can also trigger an attack on the smartphone [12]. For instance with the Android 1.5, a malformed SMS message could cause the crash of the smartphone. The user is therefore forced to restart the mobile phone. A continuous reception of such SMS can cause a Denial of Service (DoS) in which the mobile phone becomes unavailable.

Drivers added by each manufacturer also suffer from attacks. That is the case for instance of the third party kernel drivers developed by Samsung [31]. This exploit enables the user to access the memory of the device and inject some code, and run the code. This bug is exploited to root the phone i.e., elevate the privileges of users that by default have normal privileges on Android.

Some malware exploit Android firmware vulnerabilities to elevate their privileges. It is the case for instance of the application BaseBridge [14] which exploits a breach in ADB for elevating its privileges.

Android browser has also been revealed vulnerable. G. Kurtz et al. [21] have presented a Remote Access Tool (RAT) which can infect the smartphones with Android 2.2. For this, they used a bug in the Android WebKit browser to inject some malicious code.

This vulnerability can be exploited to install malware.

6.2.2 Attack vectors detected

For knowing the most used attack vectors, we have selected 38 unique malware Android applications that are the most common malware on the market. By unique, we intend that the malware in the list are not variant of each other. In Table 6.1, the list of selected malware is displayed with their attack vectors and their behaviors. By behaviors, we have selected the SMS Premium Sending (SPS), the communication with the Command and Control (C&C) server, the Stealing of private information (S) such as the phone ID for instance, and the Installation or Suggestion (IS) other applications to the user. Finally, we consider Social Engineering (SE).

We expect to have a lot of infections by Social Engineering since it is the easiest attack vector to infect a smartphone. This expectation is confirmed by our analysis of 38 malware. We also notice that some malware use some exploits in order to elevate their privileges. Most of the malware steal private information, transform the smartphone in a bot, and send Premium SMS while few malware suggest or install other applications. However, we have not detected any viruses or worms for Android but we can expect to see them in the near future.

6.3 Mitigation Methods

6.3.1 Malware Detection

In [66] the authors use Snort to detect malicious network traffic and a hardened Android kernel to successfully remedy malicious apps. The countermeasures range from blocking the network traffic coming from that app through termination of the process, removal of the application package to removal of all files owned by the application.

In [55] Bickford et al. discovered that running a malware detector on a mobile device may drain the battery very fast. They determined a sweet spot in the trade-off between energy consumption and security.

Malware name	Type	Attack vector(s)	Behavior			
			SPS	C&C	S	IS
AnserverBot	Trojan	SE		✓		
Basebridge	Trojan	SE & exploit on ADB	✓			
CruiseWind	Trojan	SE	✓	✓		
Dogowar	Trojan	SE	✓			
Droiddeluxe	Trojan	SE			✓	
Droidream	Trojan	SE & exploit on firmware		✓	✓	✓
DroidKungFu	Trojan	SE & exploit on firmware		✓		
Ewalls	Trojan	SE			✓	
FakeRun	Trojan	SE				
FakeInstaller/Boxer	Trojan	SE	✓			
FinFisher	Trojan	SE			✓	
Foncy	Trojan	SE	✓			
Geinimi	Trojan	SE		✓		
ggtracker	Trojan	SE	✓		✓	
Gingermaster	Trojan	SE & exploit on firmware		✓	✓	
Golddream/spygold	Trojan	SE		✓	✓	
hipposms	Trojan	SE	✓			
hongtoutout	Trojan	SE				
lena/dkfbootkit	Trojan	SE		✓		✓
logastrod	Trojan	SE	✓			
Loozfon	Trojan	SE			✓	
lovetrap/cosha	Trojan	SE	✓		✓	
mobiletx	Trojan	SE			✓	
moghava	Trojan	SE				
Nickibot	Trojan	SE		✓	✓	
nickyspy	Trojan	SE			✓	
ozotshielder/kmin	Trojan	SE	✓		✓	
SMS.AndroidOS.FakePlayer.a	Trojan	SE	✓			

Malware name	Type	Attack vector(s)	Behavior			
			SPS	C&C	S	IS
Tapsnake	Trojan	SE		✓		✓
VDLoader	Trojan	SE				✓
plankton/tonclank	Trojan	SE		✓	✓	✓
pjapps	Trojan	SE	✓	✓	✓	
roguesppush/autospsubscribe	Trojan	SE	✓			
smshider	Trojan	SE			✓	✓
sndapps	Trojan	SE			✓	✓
spitmo/zitmo	Trojan	SE				✓
yzhcsms	Trojan	SE	✓			
zsone	Trojan	SE	✓			

Table 6.1: Malware list

7 Conclusion

In this report we presented a survey of current smart mobile platforms. The goals of this report were to decide on a target platform for the mobile honeypot, and to distill attack vectors that the honeypot can cover.

To that end we surveyed a number of smart mobile platforms. We then investigated their market share. Given this information we defined the prerequisites of the virtualized mobile honeypot. Using these, we decided to target the Android platform, because it is the most popular mobile platform, and it is the target of the majority of malware. Further, is in large parts open source, which enables us to virtualize it.

Having decided on the target platform, we further investigated the attack vectors of Android that the honeypot can cover. Our analysis of attack vectors revealed that there are three categories: physical attacks, social engineering and exploits. We argue that there is no effective means against physical attacks. Especially the JTAG interface can usually not be turned off in software. So there is essentially nothing that we can do. However, physical attacks require that the attacker get physical access to the device, which limits the effectiveness. For the scope of this work package, we will not address physical attacks.

Social engineering is an attack vector that is very interesting. It sheds light on the role of the user. It is the user who cannot interpret the requested permissions on App installation. It is the user who turns to black markets to obtain potentially malicious pirated versions of legitimate paid Apps. Therefore, to assess the current threat situation of smart mobile devices, we cannot ignore the user.

The third attack vector is software vulnerabilities. The impact of a vulnerability is based on the location of the vulnerability. In the worst case, the vulnerability is located in the kernel. If the attacker manages to write a reliable exploit that allows him to run her own code in kernel context, then all the platforms security measures are void. Thus, for our honeypot we must assume the kernel to be vulnerable.

Glossary

API	Application Programming Interface.
App	Mobile Application.
ASLR	Address Space Layout Randomization.
C&C	Command and Control.
CLR	Common Language Runtime.
DNS	Dynamic Name Resolution.
DoS	Denial of Service.
GPL	Gnu General Public Licence.
IDS	Intrusion Detection System.
IPC	Inter-process Communication.
IS	Installation or Suggestion.
JVM	Java Virtual Machine.
NDA	Non-Disclosure Agreement.
NFC	Near Field Communication.
OEM	Original Equipment Manufacturer.
OS	Operating System.
QA	Quality Assurance.
RAT	Remote Access Tool.
RFID	Radio Frequency IDentification.
SE	Social Engineering.
SLOC	Source Lines Of Code.

SMS Short Message Service.
SoC System on a Chip.
SPS SMS Premium Sending.

VM Virtual Machine.

Bibliography

[1] Android and apple ios capture a record 92 percent share of global smartphone shipments in q4 2012. http://blogs.strategyanalytics.com/WSS/post/2013/01/28/Android-and-Apple-iOS-Capture-a-Record-92\-Percent-Share-of-Global-Smartphone-Shipments-in-Q4-2012.aspx. Online, visited 12/04/2013.

[2] Are Smart Phones Spreading Faster than Any Technology in Human History? http://www.technologyreview.com/news/427787/are-smart-phones-spreading-faster-than-any-technology-in-human-history/. Online, visited 12/04/2013.

[3] ARM1136JF-S and ARM1136J-S Technical Reference Manual: Execute never bits in the TLB entry. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0211k/Cachfici.html. Online, accessed 03-01-2013.

[4] How Android encryption works. http://source.android.com/tech/encryption/android_crypto_implementation.html. Online, accessed 02-01-2013.

[5] Kaspersky Security Bulletin 2012. The overall statistics for 2012. http://www.securelist.com/en/analysis/204792255/Kaspersky_Security_Bulletin_2012_The_overall_statistics_for_2012. Online, visited 12/04/2013.

[6] KS2012: Status of Android upstreaming. https://lwn.net/Articles/514901/. Online, visited 02-01-2013.

[7] Memory Management Security Enhancements. http://source.android.com/tech/security/#memory-management-security-enhancements. Online, accessed 02-01-2013.

[8] OpenBinder Official Documentation. http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html. Online, visited 02-01-2013.

[9] The Duo Bulletin: A look at ASLR in Android Ice Cream Sandwich 4.0. https://blog.duosecurity.com/2012/02/

a-look-at-aslr-in-android-ice-cream-sandwich-4-0/. Online, accessed 03-01-2013.

[10] Worldwide Smartphone Population Tops 1 Billion in Q3 2012. http://blogs.strategyanalytics.com/WDS/post/2012/10/17/Worldwide-Smartphone-Population-Tops-1-Billion-in-Q3-2012.aspx. Online, visited 12/04/2013.

[11] Industry Leaders Announce Open Platform for Mobile Devices. http://www.openhandsetalliance.com/press_110507.html, November 2007.

[12] Google fixes SMS crashing bug in mobile OS. http://www.theregister.co.uk/2009/10/12/google_android_security_update/, 2009.

[13] Open Source Honeypots: Learning with Honeyd. http://www.symantec.com/connect/articles/open-source-honeypots-learning-honeyd, November 2010.

[14] Android.Basebridge. http://www.symantec.com/security_response/writeup.jsp?docid=2011-060915-4938-99&tabid=2, 2011.

[15] BlackBerry 7 OS Awarded Common Criteria EAL4+ Security Certification. http://www.berryreview.com/2011/11/14/blackberry-7-os-awarded-common-criteria-eal4-security-certification/, November 2011.

[16] L4Linux - Running Linux on top of L4. http://os.inf.tu-dresden.de/L4/LinuxOnL4/, January 2011.

[17] Researcher plants rogue app in Apple's App Store. http://www.computerworld.com/s/article/9221615/Researcher_plants_rogue_app_in_Apple_s_App_Store, November 2011.

[18] Windows Phone Platform Security. https://www.developer.nokia.com/Community/Wiki/Windows_Phone_Platform_Security, July 2011.

[19] Android Dashboard. https://developer.android.com/about/dashboards/index.html, December 2012.

[20] Android Debug Bridge - Android Develper Documentation. http://developer.android.com/tools/help/adb.html#move, 2012.

[21] Android smartphones infected via drive-by exploit - Update. http://www.h-online.com/security/news/item/Android-smartphones-infected-via-drive-by-exploit-Update-1446992.html, 2012.

[22] Arstechnica: The checkered, slow history of Android handset updates. http://arstechnica.com/gadgets/2012/12/the-checkered-slow-history-of-android-handset-updates/, December 2012.

[23] Arstechnica: What happened to the Android Update Alliance? http://arstechnica.com/gadgets/2012/06/what-happened-to-the-android-update-alliance/, June 2012.

[24] Black Hat hacker lays waste to Android and Meego using NFC exploits. http://www.extremetech.com/computing/133501-black-hat-hacker-lays-waste-to-android-and-meego-using-nfc-exploits, 2012.

[25] Download Blackmart Alpha 0.49.93 Android APK. http://www.crazyforandroid.com/2012/10/download-blackmart-alpha-0-49-93-android-apk/, 2012.

[26] Exploit beamed via NFC to hack Samsung Galaxy S3 (Android 4.0.4). http://www.zdnet.com/exploit-beamed-via-nfc-to-hack-samsung-galaxy-s3-android-4-0-4-7000004510/, 2012.

[27] Google's Android malware scanner detects only 15 percent of malicious code in test (update). http://www.theverge.com/2012/12/10/3751202/google-android-malware-scanner-test, December 2012.

[28] iOS Security. https://ssl.apple.com/ipad/business/docs/iOS_Security_May12.pdf, May 2012.

[29] iOS: Where can I find an current statistics for iOS version usage? http://www.quora.com/iOS/Where-can-I-find-an-current-statistics-for-iOS-version-usage, March 2012.

[30] Researchers find ways to bypass Google's Android malware scanner. https://www.infoworld.com/d/security/researchers-find-ways-bypass-googles-android-malware-scanner-194882, June 2012.

[31] The Samsung Exynos kernel exploit - what you need to know. http://www.androidcentral.com/samsung-exynos-kernel-exploit-what-you-need-know, 2012.

[32] xdadevelopers: [ROOT][SECURITY] Root exploit on Exynos. http://forum.xda-developers.com/showthread.php?t=2057818, December 2012.

[33] XNU, the kernel . http://www.puredarwin.org/developers/xnu, January 2013.

[34] Android Developer Documentation: Binder. http://developer.android.com/reference/android/os/Binder.html, January 2013.

[35] Architecture of Tizen. https://source.tizen.org/documentation/architecture-overview, January 2013.

[36] Arstechnica:. http://arstechnica.com/gadgets/2013/01/ice-cream-sandwich-and-jelly-bean-get-a-big-bump-from-the-holidays/, January 2013.

[37] BlackBerry 10 review. http://www.techradar.com/reviews/pc-mac/software/operating-systems/blackberry-10-1090522/review, January 2013.

[38] CyanogenMod. http://www.cyanogenmod.org/, 2013.

[39] FROST: Forensic Recovery Of Scrambled Telephones. https://www1.informatik.uni-erlangen.de/frost, 2013.

[40] General Dynamics: OK:Android. http://www.ok-labs.com/products/ok-android, April 2013.

[41] Google Play: Security on Android. http://support.google.com/googleplay/bin/answer.py?hl=en&answer=1368854, January 2013.

[42] Meego. https://meego.com/, January 2013.

[43] Pharming. http://en.wikipedia.org/wiki/Pharming, 2013.

[44] Phishing. http://en.wikipedia.org/wiki/Phishing, 2013.

[45] Press Announcement: Research in Motion changes its name to BlackBerry. http://press.blackberry.com/press/2013/research-in-motion-changes-its-name-to-blackberry.html, January 2013.

[46] Samsung confirms availability of Tizen Phones in 2013. http://www.tizenphones.com/samsung-confirms-availability-of-tizen-phones-in-2013/, January 2013.

[47] Signing your applications. http://developer.android.com/tools/publishing/app-signing.html, January 2013.

[48] T-Mobile G1. http://www.gsmarena.com/t_mobile_g1-2533.php, January 2013.

[49] The User-mode Linux Kernel Home Page. http://user-mode-linux.sourceforge.net/, April 2013.

[50] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 9–9, Berkeley, CA, USA, 2005. USENIX Association.

[51] R. Bace. *Intrusion detection.* Sams, 2000.

[52] R. Bace and P. Mell. Nist special publication on intrusion detection systems. Technical report, DTIC Document, 2001.

[53] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The vmware mobile virtualization platform: is that a hypervisor in your pocket? *ACM SIGOPS Operating Systems Review*, 44(4):124–135, 2010.

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

[55] J. Bickford, H. A. Lagar-Cavilla, A. Varshavsky, V. Ganapathy, and L. Iftode. Security versus energy tradeoffs in host-based mobile malware detection. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 225–238, New York, NY, USA, 2011. ACM.

[56] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[57] Coverity Inc. Coverity Scan 2010 Open Source Integrity Report. `http://www.coverity.com/html/press/coverity-scan-2010-report-reveals-high-risk-software-flaws-in-android.html`, 2010.

[58] Droid Life: A Droid Community Blog. Motorola Eases Up on Locked Bootloader Stance, Plans to Unlock Portfolio in 2011? `http://www.droid-life.com/2011/04/26/motorola-eases-up-on-locked-bootloader-stance-plans-to-unlock-portfolio-in-2011/`, April 2011.

[59] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[60] George Turbin. The Sky IS Falling: The Need for Stronger Consumer Online Banking Authentication. `http://www-304.ibm.com/jct03006c/businesscenter/fileserve?contentid=82467`, 2005.

[61] L. Hatton. Reexamining the fault density component size connection. *Software, IEEE*, 14(2):89 –97, mar/apr 1997.

[62] J.-H. Hoepman and B. Jacobs. Increased security through open source. *Commun. ACM*, 50(1):79–83, Jan. 2007.

[63] P. Kelley, S. Consolvo, C. Lorrie, J. Jung, N. Sadeh, and D. Wetherall. An conundrum of permissions: Installing applications on an android smartphone. Workshop on Usable Security, 2012.

[64] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4android: a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 39–50, New York, NY, USA, 2011. ACM.

[65] I. Mokube and M. Adams. Honeypots: concepts, approaches, and challenges. In *Proceedings of the 45th annual southeast regional conference*, ACM-SE 45, pages 321–326, New York, NY, USA, 2007. ACM.

[66] Y. Nadji, J. Giffin, and P. Traynor. Automated remote repair for mobile malware. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 413–422, New York, NY, USA, 2011. ACM.

[67] A. Ozment and S. E. Schechter. Milk or wine: does software security improve with age? In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[68] P. Chou. HTC to open bootloaders. `https://www.facebook.com/HTC/posts/10150307320018084`, May 2011.

[69] N. Project. State-of-the-Art for security threats and attacks against mobile device, 2013.

[70] T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: a survey of current android attacks. In *Proceedings of the 5th USENIX conference on Offensive technologies*, WOOT'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[71] T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: a survey of current android attacks. In *Proceedings of the 5th USENIX conference on Offensive technologies*, WOOT'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[72] R. N. M. Watson. New approaches to operating system security extensibility. Technical Report UCAM-CL-TR-818, University of Cambridge, Computer Laboratory, Apr. 2012.

[73] B. Witten, C. Landwehr, and M. Caloyannides. Does open source improve system security? *Software, IEEE*, 18(5):57 –61, sep/oct 2001.

[74] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.