Mobile Hacking

Ein kompakter Einstieg ins Penetration Testing mobiler Applikationen – iOS, Android und Windows Mobile

Bearbeitet von
Michael Spreitzenbarth

1. Auflage 2017. Taschenbuch. 12, 224 S. Paperback
ISBN 978 3 86490 348 9
Format (B x L): 16,5 x 24 cm

1 Einführung in mobile Betriebssysteme und deren Applikationen

Smartphones und Tablet-PCs sind aus dem täglichen Leben von Privatpersonen ebenso wenig wegzudenken wie aus dem Alltag in Unternehmen. Diese Geräte dienen längst nicht mehr nur dem Zweck der Kommunikation, wie es mit dem Telefon vor einigen Jahren noch der Fall war. Sie werden inzwischen vielmehr dazu benutzt, Zugriff auf sensible Firmen- oder Privatnetze (z. B. per VPN) zu erhalten oder teilweise sehr sensible Daten (wie z. B. Bilder und Angebote) zu verarbeiten.

Betrachtet man die Entwicklung der letzten Jahre, so sieht man, dass diese Geräte immer mehr Fähigkeiten erhalten und in immer mehr Szenarien eine Rolle spielen. Steuerung der kompletten Hauselektronik, Zugangskontrolle zu hochgesicherten Bereichen, Katastrophenfrühwarnung und der Ersatz des Notebooks in Unternehmen sind nur einige Beispiele, in denen diese Geräte und die darauf installierten Applikationen (oder kurz Apps) in Zukunft eine wichtige Rolle spielen werden.

Will man für solch kritische Szenarien ein mobiles Endgerät einsetzen, so landet man nach der initialen Betrachtung der Hardware immer wieder an dem folgenden Punkt: *Wie sicher ist eigentlich die Applikation selbst?*

Um diese Frage zu beantworten, gibt es eigentlich nur zwei mögliche Lösungsansätze: entweder Vertrauen in die Marketingfolien und -versprechen der Hersteller solcher Lösungen oder das Durchführen eigener Tests, um sicherzustellen, dass die ausgesuchte Lösung auch das leistet, was sie verspricht. Einen solchen Test nennt man Penetrationstest (oder kurz Pentest).

Betrachtet man die Schlagzeilen einschlägiger Magazine oder Webseiten (und ebenfalls die Beispiele, die in diesem Buch erwähnt werden), so sieht man sehr schnell, dass die erste Option (blindes Vertrauen in die Marketingversprechen) oft der falsche Ansatz ist, wenn man erwägt, sensible Daten mit einer Applikation zu verwalten oder zu verarbeiten. Aus diesem Grund werden im Rahmen dieses Buches Wege gezeigt, wie man selbst Applikationen überprüfen und deren Schwächen ans Tageslicht bringen kann.

Wir betrachten dazu ausführlich die Systeme Android und iOS, zeigen aber auch erste Einstiegspunkte in Windows Phone, da dieses System in vielen Einsatzszenarien eine immer wichtiger werdende Rolle spielt. Beginnen möchten wir

in diesem Kapitel mit der allgemeinen Einführung in die Systeme und den Aufbau ihrer Applikationen. Außerdem wird das Einrichten der Laborumgebung beschrieben, in der sich die später gezeigten Analysen durchführen lassen.

1.1 Android

Das Android OS, das ursprünglich für die Verwendung auf Smartphones und Tablet-PCs entwickelt wurde, findet in letzter Zeit auch immer mehr Verwendung auf Set-Top-Boxen, TVs oder als Car-Entertainment-System. Die Basis des Betriebssystems wird von der Open Handset Alliance unter der Führung von Google entwickelt und ist vollständig Open Source, lediglich die Anpassungen von Google (eigene Apps wie z. B. Google Maps und zugehörige Bibliotheken) sind nicht in ihrem Quellcode verfügbar. Die Komponenten, die das System ausmachen und auch im weiteren Verlauf des Buches von Interesse sind, werden in den folgenden Abschnitten genauer beleuchtet.

1.1.1 Die Entwicklung der Android-Plattform

Zu Beginn der Einführung in die Android-Plattform wird die bisherige Entwicklung der verschiedenen Android-Versionen aufgezeigt. Die gesamte zeitliche Entwicklung der Plattform, speziell deren Hauptversionen, ist in Tabelle 1–1 zusammen mit ihren Versionsnummern, Codenamen und Erscheinungsdaten dargestellt. Seit der Version 1.5 haben die Hauptversionen immer den Namen von populären US-Süßigkeiten. Dies hat sich erst durch die Kooperation mit Nestlé in Version 4.4 und dem Codenamen *KitKat* geändert, Google blieb zwar bei den Süßigkeiten, wechselte jedoch auf den europäischen Markt.

1.1.2 Die Architektur des Betriebssystems

Die Android-Architektur kann in vier verschiedene Schichten unterteilt werden: Basis der Architektur ist der Linux-Kernel, darüber liegt eine kombinierte Schicht aus Systembibliotheken und der eigentlichen Android-Laufzeitumgebung, dann folgt das Applikationsframework und als oberste Ebene die eigentlichen Applikationen (siehe Abbildung 1–1). Jede dieser vier Schichten stellt spezielle Interfaces und Systemressourcen für die darüberliegende Schicht zur Verfügung, sodass eine Interaktion zwischen den einzelnen Schichten ermöglicht wird.

Der Linux-Kernel

Wie auf der Abbildung 1–1 zu erkennen ist, bildet der Linux-Kernel in Version 2.6.x die Basis des Android-Systems. Dieser wurde um spezielle Module erweitert, um die Hardware der Android-Geräte zu unterstützen. Zusätzlich dazu wird der Kernel für die Verwaltung der laufenden Prozesse sowie des Speichers

Version	Codename	API	Veröffentlichung	Verwendung
1.0	Base	1	09/2008	s
1.5	Cupcake	3	04/2009	s
1.6	Donut	4	09/2009	s
2.0	Eclair	5 & 6	10/2009	s
2.1	Eclair	7	01/2010	s
2.2	Froyo	8	05/2010	s
2.3	Gingerbread	9 & 10	12/2010	s
3.0	Honeycomb	11–13	02/2011	t
4.0	Ice Cream Sandwich	14 & 15	10/2011	s,t
4.1	Jelly Bean	16	06/2012	s,t
4.2	Jelly Bean	17	11/2012	s,t
4.3	Jelly Bean	18	07/2013	s,t
4.4	KitKat	19 & 20	10/2013	s,t
5.0	Lollipop	21	10/2014	s,t
5.1	Lollipop	22	03/2015	s,t
6.0	Marshmallow	23	10/2015	s,t

Tab. 1–1: Liste der Android-OS-Versionen und deren Erscheinungsdatum seit der ersten offiziellen Veröffentlichung in 2008 (s = Smartphone, t = Tablet-PC)

verwendet und stellt einige der Sicherheitsmechanismen bereit, die in Android implementiert sind.

Prozesse mit ihrem zugehörigen Identifier (PID) sind eines der wichtigsten Konzepte des Linux-Kernels. Neben dieser PID speichert der Kernel weitere wichtige Informationen über laufende Prozesse – wie z. B. den Prozessstatus, den Thread, in dem der Prozess läuft, und Angaben darüber, welche Dateien verwendet werden (eine vollständige Liste stellt der Linux-Quellcode [10] bereit). Diese Daten werden in einer gesonderten Struktur – `task_struct` – abgelegt. Die PID ist im weiteren Zusammenhang sehr wichtig, da mit ihrer Hilfe während der dynamischen Analyse die Operationen den entsprechenden Applikationen zugeordnet werden können (mehr dazu in Abschnitt 4.2.1).

Systembibliotheken und Laufzeitumgebung

In der darüberliegenden Schicht befinden sich sie Systembibliotheken und die eigentliche Android-Laufzeitumgebung. Diese Bibliotheken sind in C bzw. C++ geschrieben und werden sowohl vom System selbst als auch von allen installierten

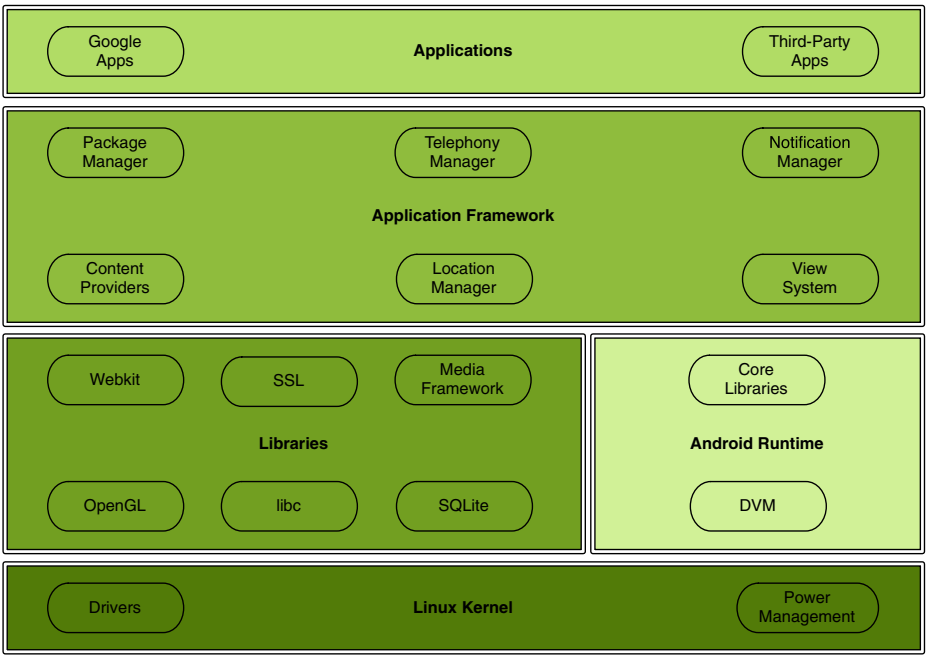

Abb. 1–1: Übersicht über die Architektur des Android-Betriebssystems

Apps verwendet. Die Android-Laufzeitumgebung enthält die *Dalvik Virtual Machine (DVM)* sowie die wichtigsten Java-Bibliotheken. Die DVM- und die Java-Bibliotheken wurden speziell an die Vorgaben eines mobilen Betriebssystems – geringer Stromverbrauch und geringe Rechenleistung – angepasst.

Applikationsframework

Die nächste Ebene ist das sogenannte Applikationsframework, das die *Application Programming Interfaces (API)* bereitstellt. Diese Ebene ist eine Art Übersetzungsschicht: Sie stellt eine hohe Anzahl an geräteabhängigen Schnittstellen zur Verfügung, die es einem Entwickler erlauben, auf alle Features des Endgerätes zuzugreifen, ohne dass er hierfür tiefere Kenntnisse der einzelnen Komponenten benötigt.

Die Applikationen selbst

Darüber liegen schließlich die eigentlichen Applikationen. Jede dieser installierten Applikationen ist zum großen Teil in Java geschrieben (mit optionalen eigenen Bibliotheken) und wird zur Laufzeit in einer eigenen DVM ausgeführt. Aktuell gibt es im offiziellen Google Play Store knapp über 1,6 Millionen dieser Applikationen (Apps) [16].

Android-Applikationen unterstützen auch die Verwendung von nativen Bibliotheken, die in C/C++ geschrieben sind. Sobald man jedoch die Applikation zur Verwendung auf einem Endgerät oder dem Emulator bauen lässt (z. B. durch *Eclipse* oder *Android Studio*), wird aus dem Java-Code ein in der DVM ausführbarer Bytecode, der in einer dex-Datei gespeichert wird. Dieser Bytecode unterscheidet sich an vielen Stellen von herkömmlichem Java-Bytecode, was sich dadurch auch auf die Analyse auswirkt. Zum Wichtigsten einer Android-Applikation gehört das *Android-Manifest*, das alle vom Nutzer abgefragten Berechtigungen sowie die zur Laufzeit nötigen *Intents*, *Listener* und *Receiver* enthält (mehr zu diesen Begriffen später in diesem Abschnitt). Das Android-Manifest wird zusammen mit dem DVM-Bytecode, den externen Bibliotheken und allen anderen Ressourcen, die für die Applikation benötigt werden, in eine Art ZIP- bzw. JAR-Paket zusammengeschnürt. Dieses Paket hat die Dateiendung `.apk` und wird vom Play Store (oder jeder anderen Installationsquelle) auf das Endgerät kopiert und dort installiert.

Im Allgemeinen besteht eine so paketierte Android-Applikation aus den folgenden Dateien und Verzeichnissen:

- **META-INF**: Verzeichnis mit folgenden Dateien:
  - **MANIFEST.MF**: das Manifest in kompilierter Form
  - **CERT.RSA**: das Zertifikat, mit dem die Applikation signiert wurde
  - **CERT.SF**: eine Liste aller Ressourcen und das SHA-1-Digest
- **lib**: Verzeichnis mit speziell für einen bestimmten Prozessor kompiliertem Code
  - **armeabi**: kompilierter ARM-Code
  - **armeabi-v7a**: kompilierter ARMv7-Code
  - **x86**: kompilierter x86-Code
  - **mips**: kompilierter MIPS-Code
- **resources.arsc**: eine Datei mit vorkompilierten Bibliotheken
- **res**: Verzeichnis mit Bibliotheken, die nicht in `resources.arsc` kompiliert wurden
- **AndroidManifest.xml**: ein weiteres Manifest mit wichtigen Metainformationen für die Applikation in codierter Form
- **classes.dex**: der kompilierte Java-Code der Applikation

Nachfolgend ein Beispiel einer realen Applikation, wie sie aus dem Play Store geladen wurde:

```
$: file DEMO.apk

DEMO.apk: Java Jar file data (zip)
```

Codebeispiel 1–1: Ausgabe des `file`-Befehls auf eine Android-Applikation zum Bestimmen des Dateityps

```
// die ersten 4 Byte sind bei jeder Android-Applikation 50 4b 03 04
$: hexdump -C -n 64 DEMO.apk

00000000 504b03040a000008 000039584f410000 |PK........9XOA..|
00000010 0000000000000000 0000250005006173 |..........%...as|
00000020 736574732f436f6e 6669677572617469 |sets/Configurati|
00000030 6f6e732f66646173 666a6b616c616664 |ons/fdasfjkalafd|
```

Codebeispiel 1–2: Anzeige der ersten 64 Byte einer Android-Applikation in Hex

```
$: unzip -l DEMO.apk

Archive:  DEMO.apk
  Length      Date    Time    Name
 --------  ---------- -----   ----
     2632  05-31-14 11:54   res/layout/activity_line.xml
     1820  05-31-14 11:54   res/layout/activity_main.xml
      392  05-31-14 11:54   res/xml/device_admin_sample.xml
     5068  05-31-14 11:54   AndroidManifest.xml
     2760  05-31-14 11:54   resources.arsc
    11566  05-31-14 11:47   res/drawable-hdpi/lineinfo_update.png
    14068  05-24-14 19:21   res/drawable-hdpi/ic_launcher.png
   718628  05-31-14 11:50   classes.dex
      687  05-31-14 11:54   META-INF/MANIFEST.MF
      740  05-31-14 11:54   META-INF/CERT.SF
     1203  05-31-14 11:54   META-INF/CERT.RSA
 --------                   -------
   759564                   11 files
```

Codebeispiel 1–3: Auflistung der Inhalte einer Android-Applikation

Bestandteile einer Android-Applikation

Wie bereits kurz erwähnt, gibt es mehrere essenzielle Bestandteile einer Android-Applikation. Neben dem gerade erwähnten Android-Manifest und den externen

Bibliotheken gibt es auch im eigentlichen Code der Applikation wichtige Bestandteile, die bei der Funktion, aber auch bei der Analyse eine wesentliche Rolle spielen. Hierzu gehören: *Activities*, *Services*, *Broadcast Receiver*, *Shared Preferences*, *Intents* und *Content Provider*. Da diese Komponenten im Folgenden sehr wichtig sind, werden sie an dieser Stelle ausführlicher beschrieben:

Android-Manifest: Jede Android-Applikation besitzt eine besondere Datei, die allgemeine Informationen über die Applikation beinhaltet, wie z. B. den Namen, die SDK-Version, die eigene Versionsnummer, angeforderte Berechtigungen sowie Hard- und Softwareanforderungen. Diese Datei heißt *AndroidManifest.xml* oder kurz *Android-Manifest*. Die codierten Informationen innerhalb dieser Datei werden während der Installation vom System ausgewertet, um so dem Nutzer die Berechtigungen anzuzeigen, die er akzeptieren muss, bevor die Applikation erfolgreich installiert werden kann. Des Weiteren sind in ihr auch sämtliche Activities enthalten, die als Einstiegspunkte in die Applikation dienen können. Die »MAIN«-Activity wird dabei als Einstiegspunkt verwendet, sobald ein Nutzer die App über den Launcher oder den Homescreen startet. Zusätzlich kann man im Android-Manifest erkennen, ob die Applikation auf externe Events wartet, um besondere Aktionen auszuführen (z. B. sobald eine SMS mit einem speziellen Text auf dem Gerät eintrifft, startet die Applikation und zeigt dem Nutzer eine Pop-up-Nachricht an). Wegen dieser Eigenschaften ist das Android-Manifest einer des besten Startpunkte für eine manuelle Analyse einer verdächtigen Applikation.

Activities: Activities stellen eine interaktive grafische Benutzeroberfläche bereit, indem sie Daten und Informationen der Applikation auf dem Display darstellen und Touch-Events des Nutzers an die Applikation zur Verarbeitung weitergeben. Jede dieser Activities muss im Android-Manifest deklariert werden, andernfalls ist eine Ausführung nicht möglich. Wechselt eine Activity in den Hintergrund, z. B. durch das Öffnen einer anderen Applikation, pausiert die Activity und alle Daten, die nicht als persistent deklariert wurden, werden gelöscht. Befindet sich eine Activity in diesem Modus, so kann das Android-System diese löschen, um Ressourcen freizugeben.

Intents: Intents sind ein spezieller Datentyp der Android-Architektur. Sie werden zur Kommunikation zwischen Komponenten einer Applikation oder zwischen Komponenten unterschiedlicher Applikationen verwendet. Jeder Intent besitzt zwei Attribute – Data und Action – und zusätzlich drei optionale Attribute – Types, Categories und Extras. Die Aktion beschreibt, was die Applikation mit den erhaltenen Daten macht. Das optionale Attribut Type ist nur nötig, falls die übermittelten Daten nicht in der *URI-Syntax (Uniform Resource Identifier)* vorliegen. In diesem Fall beschreibt der Type den MIME-Type der übermittelten Daten. Extras werden dann verwendet, wenn die zu übermittelten Daten nicht in das eigentliche Datenfeld passen. Das Attribut Category kann verwendet werden, um die Aktion genauer zu beschreiben.

Intents können in zwei Arten aufgeteilt werden: implizit und explizit. Explizite Intents haben einen spezifischen Receiver, der im Intent selbst definiert ist und meist in unterschiedlichen Komponenten einer Applikation verwendet wird. Im Gegensatz dazu werden implizite Intents an den Paketmanager des Android-Systems weitergegeben, das dann die passende Applikation sucht, die den passenden Broadcast Receiver definiert hat.

Broadcast Receivers: Android-Applikationen können eigene Activities als sogenannte Broadcast Receiver beim System registrieren. Dies hat zur Folge, dass – wenn eine andere Applikation oder das System selbst einen impliziten Intent via Broadcast-Nachricht versendet – der zugehörige Broadcast Receiver benachrichtigt wird und die damit verlinkte Activity gestartet wird.

Content Provider: Dieser Datentyp wird verwendet, um persistente Daten für andere Applikationen zugänglich zu machen. Content Provider sind der einzige Weg, Daten zwischen Applikationen auszutauschen, da es im System keinerlei Speicherbereich gibt, der von allen Applikationen gemeinsam benutzt werden kann. Applikationen verwenden SQL und relationale Datenbankschnittstellen, um die bereitgestellten Daten zu verarbeiten.

Services: Services sind Komponenten, die im Hintergrund und ohne Nutzerinterface ausgeführt werden und ebenfalls im Android-Manifest deklariert sowie dem Service Manager mitgeteilt sein müssen. Bei der Definition dieser Services kann auch festgehalten werden, wie die Applikationen bzw. die Komponenten der eigenen Applikation mit dem Service kommunizieren dürfen. Services können Dateisystemoperationen durchführen, Netzwerkverkehr überwachen oder anderweitige Informationen (wie z. B. den Standort des Gerätes) an andere installierte Applikationen weiterreichen. Verlangt eine Applikation Zugriff auf einen speziellen Service, so wird dies über den Binder IPC und den Service Manager umgesetzt. Dabei prüft der Service Manager die Zugriffsrechte und der Binder IPC agiert als direkte Kommunikationsschnittstelle zwischen Applikation und abgefragtem Service.

Shared Preferences: Diese werden von der Applikation verwendet, um kleinere Datenmengen zu speichern, die zur Funktion benötigt werden (z. B.: Spielstände). Sie liegen meist als XML in einem Verzeichnis namens `shared_prefs`. Sensible Daten sollten hier auf keinen Fall abgelegt werden, da sie an dieser Stelle anfällig für Diebstahl und Offenlegung sind.

1.2 Apple iOS

Apple iOS, das ursprünglich aus Mac OS X entstand und auch heute noch viele Gemeinsamkeiten damit hat, gibt es seit 2007 für iPhones und später auch für iPads und den AppleTV. Im Gegensatz zu Android ist es jedoch Closed Source. Die Komponenten, die das System ausmachen und auch im weiteren Verlauf des

Buches von Interesse sind, werden in den folgenden Abschnitten genauer beleuchtet.

1.2.1 Die Entwicklung der iOS-Plattform

Die Geschichte von iOS begann im Jahr 2005, als bei Apple entschieden wurde, ein Smartphone zu entwickeln, das auf dem vom Mac bekannten OS X beruhte. Die Vorstellung des ersten iPhone und damit auch von iOS war knapp 2 Jahre später im Januar 2007 auf der *MacWorld Conference and Expo*. Zuerst unterstützte das iPhone nur einige wenige Applikationen von Apple selbst und sogenannte Web-Apps, bei denen lediglich im Browser eine GUI angezeigt wird und die eigentliche Technik auf einem Server-Backend läuft. Ab 2008 folgte dann das erste SDK, womit es möglich war, auch native Applikationen als Dritthersteller zu entwickeln. Zeitgleich wurde auch der App Store eingeführt um die Applikationen an die Endkunden zu vertreiben.

Den offiziellen – und bis heute bekannten – Namen *iOS* bekam das System von Apple erst im Jahre 2010. Im selben Jahr wurden auch die Betriebssysteme von iPhones und iPads mit der Veröffentlichung von iOS 4.2.1 vereinheitlicht. Seit 2014 gibt es zusätzlich zu dem bekannten iOS auch noch *watchOS* für die Apple Watch sowie *tvOS* für den AppleTV mit einem Erscheinungsdatum ab 2015 (zuvor besaß der AppleTV ebenfalls iOS als Betriebssystem). Die gesamte zeitliche Entwicklung der Plattform, speziell deren Hauptversionen, ist in Tabelle 1–2 zusammen mit ihren Versionsnummern und Erscheinungsdaten dargestellt.

Wie man an der Tabelle sehr schön erkennen kann, versucht Apple jedes Jahr zu ihrer September-Keynote eine neue Hauptversion von iOS vorzustellen. Dieser Termin ist für alle Entwickler, aber auch Analysten, ein Datum, an dem sie sich die neuen Funktionen der Plattform, aber vor allem auch die Änderungen an bisherigen Funktionen und Sicherheitsmechanismen genau anschauen sollten. Auch viele der in diesem Buch verwendeten Werkzeuge und Techniken sind stark von der iOS-Version der verbundenen Endgeräte abhängig.

1.2.2 Die Architektur des Betriebssystems

Die iOS-Architektur kann in fünf verschiedene Schichten unterteilt werden: Basis der Architektur ist das sogenannte Core OS (Darwin), darüber liegen die Core-Services gefolgt von der Schicht, die für Grafik, Audio und Video zuständig ist. Eine Schicht darüber folgt Cocoa Touch und als oberste Ebene die eigentlichen Applikationen (siehe Abbildung 1–2). Jede dieser fünf Schichten stellt spezielle Interfaces und Systemressourcen für die darüberliegende Schicht zur Verfügung, sodass eine Interaktion zwischen den einzelnen Schichten ermöglicht wird.

Version	Veröffentlichung	Verwendung
1.0	01/2007	s
1.1	09/2007	s
2.0	07/2008	s
2.1	09/2008	s
2.2	11/2008	s
3.0	06/2009	s
3.1	09/2009	s
3.2	04/2010	s,t
4.0	06/2010	s,t,a
4.1	09/2010	s,t,a
4.2	11/2010	s,t,a
4.3	03/2011	s,t,a
5.0	10/2011	s,t,a
5.1	03/2012	s,t,a
6.0	09/2012	s,t,a
6.1	02/2013	s,t,a
7.0	09/2013	s,t,a
7.1	03/2014	s,t,a
8.0	09/2014	s,t,a
8.1	12/2014	s,t,a
8.2	03/2015	s,t,a
8.3	04/2015	s,t,a
8.4	06/2015	s,t,a
9.0	09/2015	s,t
9.1	10/2015	s,t
9.2	12/2015	s,t

Tab. 1–2: Liste der iOS-Versionen und deren Erscheinungsdatum seit der ersten offiziellen Veröffentlichung in 2007 (s = iPhone/iPod Touch, t = iPad, a = AppleTV)

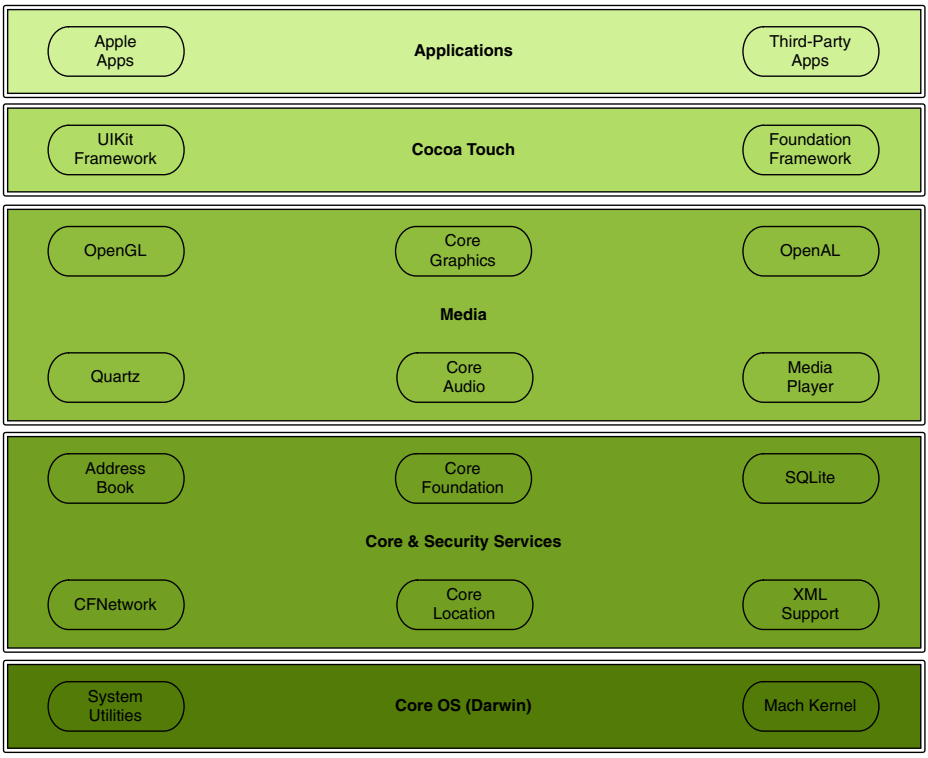

Abb. 1–2: Übersicht der Architektur des iOS-Betriebssystems

Das Core OS

Das Core OS von iOS ist auch unter dem Namen *Darwin* bekannt und ist im Wesentlichen ein vollwertiges UNIX-Betriebssystem, jedoch ohne grafische Oberfläche. Es basiert auf dem Nutzerbereich von FreeBSD und einem Mach-Kernel. Damit bietet es Benutzern und Applikationen eine Schnittstelle zur Kommunikation mit der verbauten Hardware und steuert das Speichermanagement sowie die Zugriffe auf das Dateisystem und die unterste Schicht des Netzwerkstacks. Das Besondere an Darwin ist, dass es sowohl unter Apples älteren PowerPC-Plattform lauffähig ist (und dort auch Verwendung fand) als auch unter der aktuellen Intel- und ARM-Plattform. Somit findet dieser Bereich von iOS auch auf den aktuellen mit Mac OS X betriebenen Systemen seinen Einsatz. Geräte mit iOS 9.x oder Mac OS X el Capitan setzen dabei auf die Version 15.x von Darwin.

Core- und Security-Services

Hier finden sich die Systembibliotheken von iOS, die sowohl vom System selbst als auch von allen installierten Apps verwendet werden. Hierzu gehören u. a. die

API, um mit dem iCloud-Speicherplatz zu interagieren, aber auch die nötigen Schnittstellen, um SQLite-Datenbanken zu lesen und zu schreiben oder um die Kommunikation über das Internet vorzunehmen. Des Weiteren sind hier auch die meisten Sicherheitsmechanismen verankert, wie z. B. Dataprotection (siehe dazu Abschnitt 1.2.3) und Automatic Reference Counting, auf das wir bei den Analysen von iOS-Applikationen noch zu sprechen kommen. In dieser Schicht werden auch *In-App-Käufe* und der Zugriff auf das Adressbuch sowie die *Keychain* (siehe dazu Abschnitt 1.2.3) gesteuert.

Aufrufe von Bibliotheken dieser Schicht sind während des Reversings sehr gut daran zu erkennen, dass ihre Namen mit `CF` beginnen (z. B. `CFReadStream` und `CFNetwork`).

Media

In dieser Schicht liegen alle Bibliotheken, die für die Anzeige von Grafiken jeglicher Art, das Abspielen von Videos, die Wiedergabe sowie Aufzeichnung von Audio und *AirPlay* zuständig sind. *AirPlay* ist im übertragenen Sinn die Streaming-Technologie von Apple. Sie erlaubt es, Video- und Audiodaten an über WLAN verbundene Geräte (z. B. den AppleTV) weiterzugeben (engl. streaming), und stellt diesen Dienst allen Applikationen zur Verfügung.

Cocoa Touch

Cocoa Touch ist vergleichbar mit der *Applikationsframework-Schicht* auf Android und stellt dem Entwickler einfache Schnittstellen zu den darunterliegenden Bibliotheken zur Verfügung. Es besteht aus den folgenden drei Frameworks:

- *Foundation* (relevante Basisklassen der C-basierten Programmierung, wie z. B. Strings und Arrays),
- *UIKit* (besteht aus den folgenden zwei Komponenten: AppKit [alles, was man zur Entwicklung des User-Interface benötigt, also z. B. Menüs und Buttons, aber auch Sprachanbindung] sowie Core-Data zur Erstellung von Objektgraphen und zur Ablage von Daten)

Klassen dieser Frameworks sind während des Reversings sehr gut daran zu erkennen, dass ihre Namen mit `NS` beginnen (z. B. `NSString` und `NSException`).

Die Applikation selbst

Darüber liegen schließlich die eigentlichen Applikationen. Jede dieser installierten Applikationen ist zum großen Teil in Objective-C geschrieben (mit optionalen eigenen Bibliotheken) oder neuerdings in Swift. Aktuell gibt es im offiziellen Apple App Store knapp über 1,5 Millionen dieser Applikationen (Apps) [16].

Die fertige Applikation wird über den App Store als `IPA`-Datei verteilt. Dieses Paket beinhaltet die kompilierte Applikation selbst, weitere Ressourcen, die zur Ausführung nötig sind, und Metadaten:

- **Payload/Applikation.app/:** enthält alle statischen Ressourcen und eingebundenen Bibliotheken sowie die kompilierte Applikation (das sogenannte *Mach-O-Binary*) selbst
- **iTunesArtwork:** Icon für den App Store und iTunes
- **iTunesMetadata.plist:** Metadaten wie z. B. Entwicklername oder Versionsnummer

Wie auch schon bei Android handelt es sich auch hier um einen Pakettyp, der mittels *unzip* entpackt werden kann.

Mach-O-Binary

Wie bereits kurz gezeigt, besteht eine iOS-Applikation aus verschiedenen Bestandteilen. Der für uns wichtigste Bestandteil ist das eigentliche Executable. Dieses liegt als *Mach-O*-Dateityp innerhalb des Paketes und ist – falls die Applikation aus dem App Store geladen wurde – solange verschlüsselt, bis sie auf dem Gerät ausgeführt wird. In diesem Moment wird die gesamte Applikation entschlüsselt und in den Speicher geladen (wie wir dies ausnutzen können, um an das entschlüsselte Binary für unsere Analyse zu kommen, zeigen wir in Abschnitt 5.3.3).

Bei iOS ist es häufig der Fall, dass innerhalb dieser Mach-O-Datei mehrere Executables liegen (nämlich eines für jede CPU-Architektur, auf der die Applikation lauffähig sein soll). Diese Dateien nennt man dann auch *Fat Binaries*. Wir werden später sehen, wie man eine einzelne Architektur extrahiert und so ein schlankeres Binary für die Analysen erhält (siehe dazu Abschnitt 5.2.6).

Ein solches Mach-O-Binary enthält die drei folgenden Bereiche (siehe dazu auch Abbildung 1–3):

Header: Hiermit beginnt das Mach-O-Binary. Es enthält wichtige Metadaten über die Datei selbst, dazu zählen u. a. Dateiformat, aber auch die Architektur(en), die enthalten sind.

Load-Kommandos: Dieser Bereich beschreibt das eigentliche Layout (initiales Layout der Datei im Speicher, Speicherort der Symboltabelle, Informationen zu den verschlüsselten Bereichen etc.) und die verlinkten Shared Libraries. Er besitzt für jedes Data-Segment eigene Load-Kommandos.

Data: Hier befinden sich die entsprechenden Segmente und ihre Code- bzw. Datenbereiche.

Die Struktur eines solchen Binaries – auch *Executable* genannt – ist für das manuelle Reversing sehr wichtig, wie wir in Abschnitt 5.4 sehen werden.

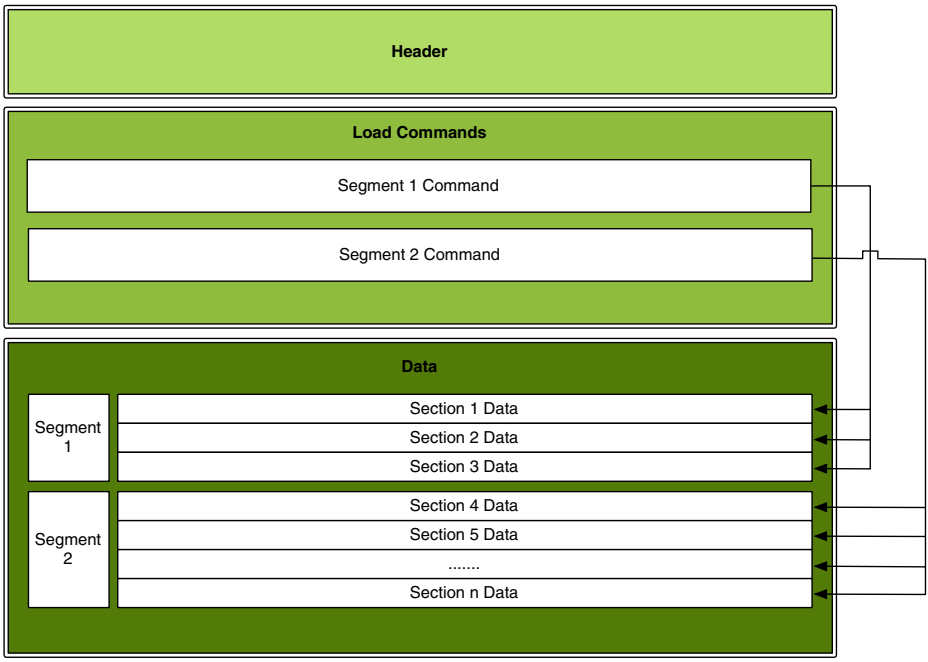

Abb. 1–3: Schematische Darstellung eines Mach-O-Binary

1.2.3 Besondere Sicherheitsmechanismen

Möchte man mit iOS als Analyseplattform arbeiten, so ist es wichtig, nicht nur die Eigenschaften des Systems und der Applikationen zu verstehen, sondern auch die Sicherheitsmechanismen, die Apple auf dieser Plattform eingeführt hat. Viele der im Folgenden beschriebenen Eigenschaften haben direkten Einfluss auf die Sicherheit der Applikation bzw. ihrer Daten und können wie im Fall der *Dataprotection-Level* und der *Keychain* vom Entwickler der Applikation beeinflusst werden. Aus diesem Grund ist es wichtig, die Unterschiede auch als Analyst zu kennen.

Secure Boot Chain

Die erste Stufe des Sicherheitsmodells von iOS – und somit auch die erste Hürde für Angriffe – ist die *Secure Boot Chain*. Also der Prozess, der das eigentliche Betriebssystem startet und dabei die gesamte Firmware lädt (für eine schematische Darstellung siehe Abbildung 1–4). Das Besondere hierbei ist, dass jede der relevanten Komponenten von Apple digital signiert wurde und diese Signatur beim Laden der Komponente überprüft wird. Dadurch kann eine Änderung oder gar der Austausch einzelner Komponenten erkannt und verhindert werden (Ähnliches

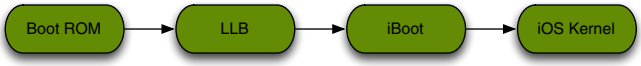

Abb. 1–4: Schematische Darstellung des Bootvorgangs unter iOS

verwendet neuerdings auch Android bei Systemen, die mit *KNOX* ausgestattet sind).

Schaltet der Nutzer das Endgerät an, so lädt der Prozessor als Erstes das *Boot ROM*. Dies ist ein spezieller Codeabschnitt, der während der Produktion in den Prozessorchip geschrieben wurde und nicht mehr verändert werden kann. Der wichtigste Bestandteil dieses Codes ist der Public-Key für Apples Root-CA.

Mithilfe dieses Public Key kann die Signatur des *Low Level Bootloader (LLB)* verifiziert werden, bevor dieser dann gestartet wird. Der LLB besitzt neben einer Vielzahl an Routinen zum Starten essenzieller Schritte auch die Fähigkeit, das *iBoot*-Abbild im Flash-Speicher zu lokalisieren und dessen Signatur zu prüfen. Schlägt dessen Prüfung fehl, startet das iOS-Gerät im sogenannten Recovery-Modus.

Gelingt die Prüfung, wird *iBoot* gestartet. Dieser zweite Bootloader nach *LLB* überprüft wiederum die Signatur des eigentlichen *iOS-Kernels*. Dieser lädt das eigentliche Betriebssystem mit allen Komponenten, die wir von der täglichen Nutzung kennen.

Was man hier sehr schön erkennen kann, ist, dass Apple alles versucht, um zu verhindern, dass bis zu diesem Punkt etwas manipuliert wird oder wichtige Module ausgetauscht werden, die eine spätere Verwendung von iOS selbst beeinflussen können. Das Umgehen dieser sicheren Kette ist auch die größte Herausforderung für Personen oder Gruppen, die einen *Jailbreak* für iOS entwickeln.

Dataprotection-Level

Wie schon erwähnt ist eines der wichtigsten Punkte beim Entwickeln von Applikationen die Verschlüsselung bzw. sichere Speicherung von Daten. Gerade wenn es um sehr sensible Daten wie Unternehmensdaten oder persönliche Nachrichten geht, sollte man darauf achten, ob sie wirklich vor unberechtigtem Zugriff geschützt sind. Apple bietet hierzu eine eigene API an: `NSFileProtection`.

Mit ihr hat der Entwickler einer Applikation die Möglichkeit, sich auf die Methoden und die Hardware von Apple zu verlassen, und muss sich keine eigenen Methoden oder gar Algorithmen ausdenken, um Daten sicher auf dem Gerät abzulegen. Hierzu hat Apple vier verschiedene Verschlüsselungsstufen bereitgestellt, aus denen der Entwickler wählen kann. Wie man in Abbildung 1–5 sehen kann, gibt es zwei verschiedene Werte, die in die Generierung der Schlüssel einfließen können: *Hardware* (also ein einzigartiger Schlüssel, der in der Hardware verankert ist) und *Passcode* (also der Passcode, den der Nutzer eingibt, um den

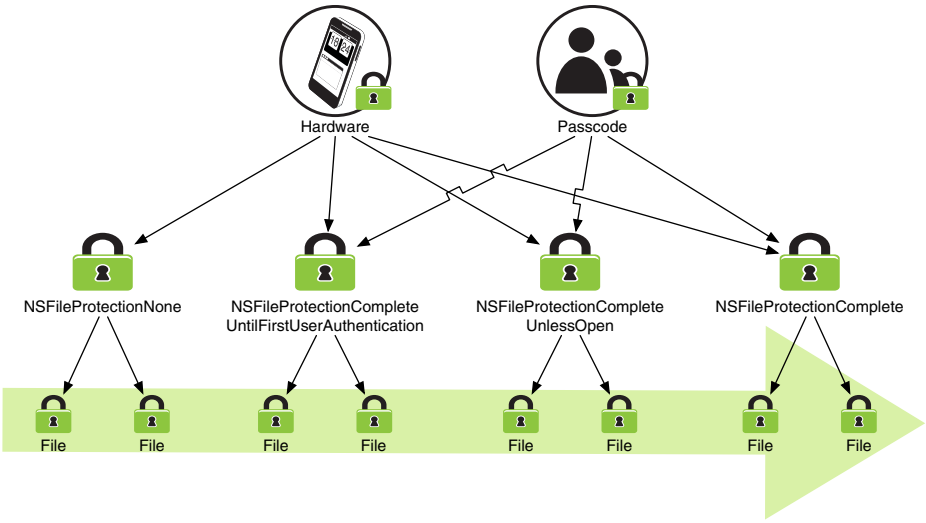

Abb. 1–5: Schematische Darstellung der Ableitung des Schlüsselmaterials

Bildschirm zu entsperren). Die Details zu den einzelnen Stufen werden im Folgenden noch kurz erläutert[1]:

NSFileProtectionNone: Diese Stufe bezeichnet Apple selbst auch gerne als »No Protection«. Die damit geschützten Daten sind auf einem iOS-Gerät so lange verschlüsselt, wie es ausgeschaltet ist. Der Schlüssel wird hierbei jedoch lediglich aus Variablen, die auf dem Gerät gespeichert sind, abgeleitet und bietet somit nahezu keinen Schutz vor Angreifern.

NSFileProtectionCompleteUntilFirstUserAuthentication: Hierbei wird ein Schlüssel aus Variablen des Gerätes selbst und dem Passcode des Nutzers abgeleitet. Der so generierte Schlüssel bleibt so lange im Speicher, bis es zu einem Neustart oder Ausschalten des Gerätes kommt. Dieser Schutz ist vergleichbar mit der Festplattenverschlüsselung eines Computers. Dies ist seit iOS 7.0 für alle Third-Party-Applikationen der Standardwert, sollte es vom Entwickler nicht explizit anders angegeben worden sein.

NSFileProtectionCompleteUnlessOpen: Diese Stufe ist sehr ähnlich zu der zuvor erwähnten. Der einzige Unterschied besteht darin, dass der Schlüssel der Datei aus dem Speicher gelöscht wird, sobald die entsprechende Datei geschlossen wird. Hiermit ist es immer noch möglich, mit geschützten Dateien zu agieren, während der Bildschirm des Gerätes gesperrt ist (wie auch zuvor), jedoch wird der Schlüssel aus dem Speicher entfernt, sobald er nicht mehr benötigt wird.

[1] Für tiefer gehende Details empfehle ich: https://www.apple.com/business/docs/iOS_Security_Guide.pdf

NSFileProtectionComplete: Dies ist die höchste Stufe der Sicherheit bei iOS-Dateien. Hier wird der Schlüssel sofort aus dem Speicher gelöscht, sobald der Bildschirm des iOS-Gerätes sich sperrt. Dies verhindert aber auch, dass eine Applikation bei gesperrtem Bildschirm mit den so geschützten Daten arbeiten kann.

Wie man am Pfeil in Abbildung 1–5 erkennen kann, steigt die Sicherheit der Verschlüsselung, je weiter man nach rechts geht, d. h., `NSFileProtectionComplete` bietet die beste Verschlüsselung. Aus den beschriebenen Eigenschaften und eventuellen Problemen ergibt sich direkt die Schlussfolgerung, dass ein Entwickler immer genau überlegen muss, wann er die Daten benötigt und welche Stufe der Verschlüsselung er somit anwenden kann. Hierbei sollte immer das Maximum gewählt werden, das den aktuellen Use Case noch ermöglicht.

Die Keychain

Die Keychain auf iOS ist vergleichbar mit einem Safe. Man kann in ihr schützenswerte oder hoch sensible Daten ablegen wie z. B. Passwörter, Zertifikate oder Crypto-Schlüssel. Wie auch schon bei der Ablage von Daten im Dateisystem, gibt es auch hier eine eigene API, die es erlaubt, vordefinierte Schutzlevel zu verwenden: `kSecAttr`.

Es gibt folgende Schutzlevel für Einträge in der Keychain (aufsteigend sortiert von »kein Schutz« bis »maximaler Schutz«):

kSecAttrAccessibleAlways: Diese Einträge in der Keychain sind immer verfügbar und zugreifbar.

kSecAttrAccessibleAfterFirstUnlock: Diese Einträge sind verfügbar, sobald das Gerät einmal entsperrt wurde, bis zu einem Neustart.

kSecAttrAccessibleWhenUnlocked: Damit versehene Einträge sind nur dann zugreifbar, wenn das Gerät entsperrt ist. Dies ist seit iOS 7.0 für alle Einträge der Standardwert, sollte es vom Entwickler nicht explizit anders angegeben worden sein.

kSecAttrAccessibleAlwaysThisDeviceOnly: Diese Einträge in der Keychain sind immer verfügbar und zugreifbar, können aber nicht auf ein anderes Gerät übertragen werden (z. B. im Rahmen eines Sync oder Restore eines Backups auf ein neues iOS-Gerät).

kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly: Hier gilt dasselbe wie bei `kSecAttrAccessibleAfterFirstUnlock`, die Einträge können aber nicht auf ein anderes Gerät übertragen werden.

kSecAttrAccessibleWhenUnlockedThisDeviceOnly: Hier gilt dasselbe wie bei `kSecAttrAccessibleWhenUnlocked`, die Einträge können aber nicht auf ein anderes Gerät übertragen werden.

kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly: Dies erlaubt es, nur dann auf Einträge in der Keychain zuzugreifen, wenn vom Nutzer ein Passcode gesetzt wurde und der Nutzer den Bildschirm mittels Passcode entsperrt hat (neu seit iOS 8). Wird dieser Passcode später wieder entfernt, können alle so markierten Einträge nicht mehr gelesen/entschlüsselt werden. Ein Übertragen auf ein anderes Gerät ist bei diesen Einträgen ebenfalls nicht möglich.

Für einen Vergleich der wichtigsten Schutzlevel zwischen dem Dateisystem und der Keychain siehe Tabelle 1–3.

Verfügbarkeit der Daten/Einträge	Dataprotection-Level	Keychain-Level
Endgerät entsperrt	`NSFileProtectionComplete`	`kSecAttrAccessibleWhenUnlocked`
Endgerät gesperrt	`NSFileProtectionComplete UnlessOpen`	n/a
Nach erstmaligem Entsperren	`NSFileProtectionComplete UntilFirstUserAuthentication`	`kSecAttrAccessibleAfter-FirstUnlock`
Sobald Passcode verfügbar	n/a	`kSecAttrAccessibleWhen PasscodeSetThisDeviceOnly`
Immer	`NSFileProtectionNone`	`kSecAttrAccessibleAlways`

Tab. 1–3: Übersicht der Schutzlevel für Daten auf dem Dateisystem und Keychain-Inhalte bei iOS

1.3 Windows 10 Mobile

Windows 10 Mobile ist die neueste Generation eines mobilen Betriebssystems von Microsoft. Microsoft baut seit dem Jahr 2002 solche Betriebssysteme und hatte sie früher unter dem Namen *Windows Mobile* veröffentlicht. In 2010 wurde dann das komplett überarbeitete und gänzlich neu-entwickelte System *Windows Phone* vorgestellt, das seither immer näher an die Desktop-Version angeglichen wird und seit 2015 unter dem Namen *Windows 10 Mobile* bekannt ist.

Die Komponenten, die das System ausmachen und auch im weiteren Verlauf des Buches von Interesse sind, werden wir uns in den folgenden Abschnitten genauer ansehen, und feststellen, wo für eine Analyse die wichtigen Daten liegen und die ersten Angriffsvektoren zu finden sind.

1.3.1 Die Entwicklung der Windows Phone-Plattform

Die Geschichte von Windows Phone begann bereits im Jahr 2002 mit der ersten Version von *Windows Mobile*. Dieses OS basierte auf *Windows CE 3.0* und

wurde speziell für tastaturlose PDAs (Personal Digital Assistant) entwickelt. Im folgenden Jahr wurden mehrere Versionen dieses Betriebssystems auf den Markt gebracht, die unterschiedliche Einsatzszenarien hatten. Dazu zählten u. a. PDAs mit Telefonfunktion und reine Smartphones ohne Touchdisplay.

2005 brachte Microsoft dann die Version *Windows Mobile 5* auf den Markt. Dieses OS hatte einige gravierende Änderungen zu den Vorgängern wie z. B. das *.NET Compact Framework* und verbesserte Versionen der Office-Apps. Zwischen 2007 und 2010 wurden verschiedene Versionen von *Windows Mobile 6* veröffentlicht. In den meisten Fällen wurde innerhalb dieser Versionen Bugs sowie Features, die die User Experience betreffen, geändert. Es gab jedoch auch einige Features im Bereich der Sicherheit des OS und der angeschlossenen Peripherie. Dazu zählten u. a. das Verschlüsseln der externen Speicherkarten, Remote Wipe auch für die Speicherkarte, verbesserte Zertifikatshandhabung und Gerätesperre sowie eine per Exchange-Policy aktivierbare Verschlüsselung der PIM-Daten (Personal Information Manager).

2010 kam mit dem eigentlichen Update auf *Windows Mobile 7* auch die Namensänderung zu *Windows Phone*. Dieses OS beruhte zwar immer noch auf *Windows CE*, hatte aber zusätzlich Bestandteile aus *Windows Compact* enthalten und schon versucht, das neue kachelbasierte Design auf die mobilen Endgeräte zu bringen. Mit der Allianz von Nokia und Microsoft kam dann im Jahre 2012 *Windows Phone 8* auf den Markt, das als erstes OS der mobilen Microsoft-Geschichte nicht mehr auf *Windows CE* aufbaute, sondern denselben Unterbau wie die vom Desktop bekannten Systeme Windows 8 und Windows RT hatte: *Windows NT*.

Windows 10 Mobile ist die neueste Version des mobilen OS von Microsoft und wurde speziell für den Einsatz auf Smartphones und Tablets entwickelt. Im Wesentlichen ist es eine angepasste Ausgabe des vom PC bekannten Windows 10. Microsoft versucht mit der in 2015 vorgestellten Version, eine Art Symbiose aus beiden Welten zu erschaffen. Eine echte Verbreitung dieses OS begann jedoch erst im Herbst 2015 mit dem Update einiger Windows Phone-8.x-Modelle. Der Marktanteil im 4. Quartal 2015 betrug lediglich 1,1 % [7], was deutlich zeigt, dass dieses OS bisher nur ein Nischenprodukt ist. Dies hat auch zur Folge, dass es wenig Forschung im Bereich der Analyse dieser Plattform gibt, was sich leider auch auf die im weiteren Verlauf dieses Buches beschriebenen Methoden auswirkt.

Eine Übersicht dieser Entwicklung ist noch einmal in Tabelle 1–4 dargestellt. Sie enthält zudem die Plattformen, auf denen die einzelnen OS-Versionen zum Einsatz kamen, und zeigt damit auch das Ende der *PocketPCs* – wie Microsoft seine PDAs stets nannte – im Jahre 2010.

Im Folgenden werden wir uns ausschließlich auf die neuere Generation des mobilen Betriebssystems mit Windows-NT-Kernel konzentrieren.

Version	Veröffentlichung	Verwendung
Windows Mobile 2002	01/2002	p
Windows Mobile 2003	06/2003	s,p
Windows Mobile 2003 SE	03/2004	s,p
Windows Mobile 5	05/2005	s,p
Windows Mobile 6	02/2007	s,p
Windows Mobile 6.1	04/2008	s,p
Windows Mobile 6.5	05/2009	s,p
Windows Mobile 6.5.3	02/2010	s,p
Windows Phone 7	02/2010	s
Windows Phone 8	06/2012	s,t
Windows 10 Mobile	01/2015	s,t

Tab. 1–4: Liste der Windows-Versionen und deren Erscheinungsdatum seit der ersten offiziellen Veröffentlichung in 2002 (s = Smartphone, t = Tablet, p = PDA/PocketPC)

1.3.2 Die Architektur des Betriebssystems

Die Architektur von Windows 10 Mobile kann in vier verschiedene Schichten unterteilt werden: Basis der Architektur ist der bereits erwähnte Windows-NT-Kernel, darüber liegen die WinRT APIs – die Schicht, die für Grafik, Audio und Video zuständig ist. Eine Schicht darüber folgt das *Component Object Model*, das den Anwendungsentwicklern verschiedene Runtimes zur Verfügung stellt und somit erlaubt, auf Basis der Kombination aus HTML, CSS und JavaScript oder XAML und einer der Programmiersprachen .NET, C# oder C++ Applikationen zu entwickeln. Als oberste Ebene folgen schließlich die eigentlichen Applikationen (siehe Abbildung 1–6). Jede dieser vier Schichten stellt spezielle Interfaces und Systemressourcen für die darüberliegende Schicht zur Verfügung, sodass eine Interaktion zwischen den einzelnen Schichten ermöglicht wird.

Windows-NT-Kernel

Der *Windows-NT-Kernel*, der ebenso die Basis aller Desktop-Betriebssysteme von Microsoft seit dem gleichnamigen *Windows NT* ist, besitzt selbst einen modularen Aufbau. Die unterste Ebene der Kernel-Schicht bildet der *Hardware Abstraction Layer*. Darauf bauen der eigentliche Hybridkernel und die später erwähnten Subsysteme auf. Der Hybridkernel kümmert sich dabei um die Vergabe des Arbeitsspeichers und der Rechenleistung auf der CPU und anderen verbauten Co-Prozessoren.

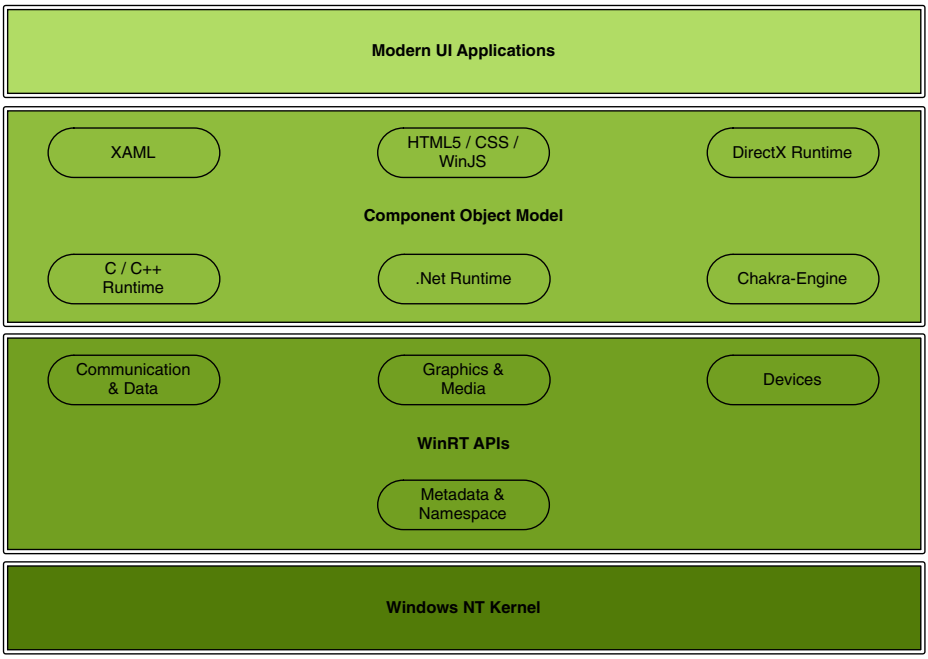

Abb. 1–6: Übersicht über die Architektur des Windows-10-Mobile-Betriebssystems

WinRT APIs

Direkt auf dem Kernel aufbauend befindet sich die *Windows Runtime*, auch *WinRT* genannt, mit den essenziellen Schnittstellen zwischen den eigentlichen Applikationen und dem *Hardware Abstraction Layer*. Hier sind all die APIs vorhanden, die eine Applikation im Hintergrund benötigt, um auf den lokalen Speicher zuzugreifen, SMS zu versenden, Kommunikation per NFC oder WLAN vorzunehmen oder auch einfach nur ein Video abzuspielen. Die wichtigsten Schnittstellen sind dabei die folgenden:

Windows RT Core: Sie übernimmt alles, was die Bereiche Authentifizierung, Kryptografie, Speichermanagement und das parallele Verarbeiten von Prozessen betrifft.

Communication & Data: Hier wird alles gesteuert, was lokalen sowie Cloud-basierten Speicher betrifft, aber auch die generelle Kommunikation über Netzwerke (GSM, Wi-Fi, NFC etc.).

Devices: In dieser API werden die Anfragen zu wichtigen Sensoren des Endgerätes gesteuert. Hierzu zählen u. a. GPS, Lichtsensor, aber auch die Kamera.

Graphics & Media: Diese API ist für das Anzeigen von grafischen Effekten, der eigentlichen Oberfläche und auch von Videos und Bildern zuständig.

Component Object Model

Das *Component Object Model*, auch bekannt unter der Abkürzung *COM*, ist eine Technik zur Erstellung von Softwarekomponenten, die unabhängig von der Programmiersprache eingesetzt werden können. Komponenten des COM ermöglichen Interprozesskommunikation und dynamische Objekterzeugung. Der Aufbau des COM ist dabei Client/Server-basiert, wobei der Server auch auf dem lokalen Endgerät laufen kann.

In dieser Schicht des OS sind eine ganze Menge verschiedener Runtimes implementiert, die für den Entwickler von Applikationen eine vereinfachte Abstraktion der APIs aus der WinRT-Schicht bieten. Möchte er also z. B. mit .NET eine Applikation programmieren, so findet er in dieser Schicht fertige APIs, die er einbinden kann, um z. B. auf die Kamera oder den lokalen Speicher zuzugreifen.

Innerhalb dieser Runtimes sind jedoch nicht alle verfügbaren WinRT APIs abgebildet. Möchte der Entwickler auf nicht zur Verfügung stehende Klassen und APIs aus dem darunterliegenden Framework zugreifen, muss die Applikation zwischen einem Client und einem Server aufgeteilt werden. An dieser Stelle greift das *Component Object Model* mit seinen vorhandenen Templates ein und unterstützt den Entwickler bei seinem Vorhaben.

Im Wesentlichen kann diese Schicht als Vermittler zwischen der Programmiersprache, in der die Applikation geschrieben ist, und den WinRT APIs gesehen werden.

Applikationen

Die oberste Schicht stellen wie üblich die Applikationen selbst dar. Diese laufen auch bei Windows 10 Mobile in einer Sandbox. Diese Sandbox sorgt dafür, dass Applikationen in einer isolierten und überwachten Umgebung ablaufen, wo sie nur begrenzten Zugriff auf das Betriebssystem, die darin enthaltenen APIs und andere Applikationen erhalten. Einzige Ausnahme sind hier Applikationen, die mit einem Enterprise-Zertifikat signiert sind. Diese können unter bestimmten Voraussetzungen diese Abschottung umgehen.

Das Modell bei Windows 10 Mobile ist – ähnlich wie es auch die Entwicklung bei iOS zeigt – darauf ausgelegt, alle Daten in der Cloud zu speichern und nur lokale Daten vorzuhalten, um auch offline arbeiten zu können oder um die Geschwindigkeit beim Arbeiten mit Applikationen zu erhöhen.

Microsoft setzt beim Verteilen der Applikationen auf zentrale Verfahren. Dazu gehören der *Windows Store*, *Configuration Manager* (ein Teil des Microsoft System Center), firmeneigene Mobile Device Management (MDM)-Systeme oder das Cloud-basierte *Windows InTune* (Microsofts eigenes Mobile Device Management [MDM]).

1.3.3 Besondere Sicherheitsmechanismen

Windows 10 Mobile bietet eine Menge Sicherheitsfeatures, die wir teilweise auch von iOS und Android kennen, aber auch von Systemen vergangener Zeiten wie *Symbian*. Man hat hier versucht, ein wirklich durchdachtes Sicherheitskonzept auf die mobilen Endgeräte zu bekommen. Einige der wichtigsten Komponenten davon werden im Folgenden kurz erläutert[2]:

Nutzerauthentifizierung: Neben den von anderen Systemen bereits bekannten Möglichkeiten, den Nutzer zu authentifizieren (wie PIN oder Passwort), bietet Windows 10 Mobile auch die Möglichkeit der Iris-Erkennung. Hierbei wird mit einer Kombination aus dem IR-Licht-Bereich und einer hochauflösenden Kamera die Iris des Nutzers fotografiert und gegen die gelernten Muster des legitimen Nutzers abgeglichen. Durch die Verwendung von Licht aus dem IR-Bereich wird es deutlich erschwert, dieses System mit bekannten Techniken [15] zu überlisten.

Enterprise Data Protection (EDP): Das Herzstück der Verschlüsselung von Windows-10-Mobile-Endgeräten ist eine TPM-gestützte Version von *BitLocker* mit definierbaren *Cipher Suites*. Hinzu kommt eine Trennung der Firmen- und Privatdaten, wie wir sie schon von iOS oder Android-for-Work kennen: Whitelist für Applikationen, die auf Firmendaten oder das Firmen-VPN zugreifen dürfen, Abkapselung der verschiedenen Bereiche (sodass ein Nutzer keine Firmendaten manuell in den persönlichen Bereich kopieren kann) und das automatische Taggen von Firmendaten bei der Verarbeitung.

Secure Boot: Windows 10 Mobile setzt hierbei auf eine Technik, wie wir sie bereits bei iOS gesehen haben. Sobald ein Windows-10-Mobile-Endgerät startet, prüft die UEFI-Firmware die digitale Signatur des Bootloaders, um sicherzustellen, dass dieser nicht manipuliert wurde. Zusätzlich wird geprüft, ob das Zertifikat, das für die digitale Signatur verantwortlich ist, von einer vertrauenswürdigen Stelle ausgestellt wurde – in diesem Fall von Microsoft selbst. Erst wenn diese Prüfungen erfolgreich absolviert wurden, wird der Bootloader geladen und ausgeführt. Im nächsten Schritt prüft der Bootloader die Integrität und Signatur des Windows-NT-Kernels und lädt ihn nur, falls auch hier wieder alle Prüfungen bestanden werden. Als letzten Schritt folgen nun die Prüfungen der restlichen Komponenten, die zum Starten von Windows 10 Mobile notwendig sind. Auch hier werden vor allem die digitalen Signaturen und die Integrität der Komponenten geprüft.

Health Attestation Service (HAS): *Device Health Attestation* verwendet das TPM (Trusted Platform Module) des mobilen Endgerätes zusammen mit Funktionen, die in der Firmware integriert sind, um essenzielle und kritische Sicherheitsfunktionen des BIOS und des Secure-Boot-Vorgangs zu überwachen.

[2] Für eine ausführliche Betrachtung empfehle ich: https://technet.microsoft.com/en-us/itpro/windows/keep-secure/windows-10-mobile-security-guide

Laut Microsoft werden diese Messungen und Überwachungen so durchgeführt, dass es für einen Angreifer oder eine Malware nahezu unmöglich ist, den Prozess zu manipulieren. Die so ermittelten Daten werden dann an den HAS gesendet und dort ausgewertet. Ist das überwachte Gerät Teil eines MDM, so können hier die Ergebnisse vom HAS abgerufen und ausgewertet werden. Manipulierte Endgeräte sind auf diese Weise schnell erkennbar und können mit vorkonfigurierten Prozessen abgehandelt werden (z. B. mit Remote Wipe, sobald HAS Hinweise auf eine Manipulation meldet).

1.4 Chancen und Risiken von mobilen Applikationen

Smartphones und Tablets sind aus dem Privatleben sowie aus dem Firmenalltag nicht mehr wegzudenken. Dabei haben sie schon lange als reine Kommunikationsmedien ausgesorgt und dienen heutzutage vielmehr als Ersatz für das Notebook oder den alten Arbeitsplatzrechner. Genau hier liegen auch die Chancen für mobile Apps: neue Einsatzszenarien für diese Art der Geräteklassen schaffen oder bekannte Einsatzszenarien auf die mobile Welt portieren.

Betrachtet man, wie diese Geräte in den letzten Jahren eingesetzt werden, so sieht man schnell, dass Zugriffe auf geschützte Netze – meist per VPN – und das Bearbeiten und Erstellen von Office-Dokumenten (Word, Excel, aber auch Präsentationen mit PowerPoint oder Keynote) keine Ausnahme mehr sind. Vor allem die immer größer werdenden Displays der Smartphones zielen darauf ab, auch unterwegs die Aufgaben erledigen zu können, für die man noch vor einigen Jahren das Notebook hätte auspacken müssen.

Mobile Future ist nicht nur ein sogenanntes Buzzword, sondern ein Trend, den immer mehr Firmen spüren und dem sie auch nachgehen möchten. Aus diesem Grund werden immer mehr Arbeitsabläufe in Apps implementiert oder zumindest integriert. Die Flut an vorhandenen Apps steigt dadurch täglich weiter an, was dazu führt, dass man wirklich für jeden Einsatzzweck eine passende App findet. Aber halten diese Apps auch immer, was sie versprechen? Gerade im Firmenumfeld oder wenn es um sensible private Daten geht, spielt die Sicherheit eine wichtige Rolle. Im Folgenden wird kurz aufgezeigt, welche Gefahren und Bedrohungen es in der Welt der Apps gibt, bevor in den späteren Kapiteln demonstriert wird, wie man herausfindet, ob die ausgewählte App auch wirklich das hält, was sie verspricht.

1.4.1 Bedrohungsszenarien durch schadhafte Applikationen

Durch die gesteigerte Nutzung von Smartphones sowie das geänderte Verteilungsmodell von Applikationen, konnten Kriminelle Smartphones als potenzielles Ziel für Malware identifizieren, um private Informationen zu stehlen und das

Smartphone für Premium-SMS-Services zu missbrauchen oder benötigte Bank-Informationen (mTAN) zu manipulieren. In diesem Abschnitt gebe ich einen kurzen Überblick über aktuelle mobile Bedrohungen und beschreibe, warum die Android-Plattform die am meisten betroffene ist.

Mobile Bedrohungen können in zwei Klassen kategorisiert werden: *webbasiert* und *applikationsbasiert*. Die webbasierten Bedrohungen auf Mobiltelefonen sind ein wachsender Angriffsvektor von Kriminellen. Diese Bedrohungen verlassen sich auf eine starke Nutzung von mobilen Browsern und deren Featurereichen Implementationen. Moderne Webbrowser unterstützen Funktionen wie embedded Video Players oder Video-Anrufe. Aufgrund der Beschaffenheit dieser Funktionen, z. B. Parsen von großen Mengen externer Daten, ist die Wahrscheinlichkeit groß, dass ausnutzbare Schwachstellen vorhanden sind. Darüber hinaus sind Angreifer in der Lage, Nutzer dazu zu bringen, einem Web-Link zu folgen, den sie via E-Mail oder Social Media erhalten haben, und dadurch das Smartphone zu infizieren, indem sie eine Browserschwachstelle ausnutzen.

Der zweite Typ von mobilen Bedrohungen sind applikationsbasierte Bedrohungen durch Drittanbieter-Applikationen in den mobilen Märkten. Um Applikationen auf den Smartphones zu installieren, haben Hardwarehersteller (wie Lenovo, Samsung etc.) sogenannte *Mobile Markets* wie z. B. Apples »App Store« and Googles »Google Play« entwickelt. Auf iOS-basierten Geräten kann Software nur vom App Store bezogen werden. Darüber hinaus bewertet Apple jede Software, die in den App Store geladen wird, und lässt nur Apps zu, die bestimmte (nicht näher spezifizierte) Sicherheitschecks bestehen. Auf Android-Geräten ist der Nutzer in der Lage, Applikationen von Drittanbieter-Märkten zu installieren. Besonders in Asien sind viele dieser Märkte entstanden. Typischerweise besteht bei Nutzung dieser Märkte ein großes Risiko, dass man sich bei der Installation von Applikationen schadhafte Applikationen einfängt, da die Marktbesitzer die angebotenen Applikationen nicht entsprechend prüfen.

Betrachtet man die schadhaften Applikationen aus solchen Märkten genauer, so kann man sie – nach Felt et al. [6] – in die folgenden drei Bedrohungstypen eingruppieren:

Malware: Angreifer wollen durch die Installation von Malware Zugang zum Gerät erhalten. Das Ziel: Daten stehlen, Aktionen fernsteuern oder das Gerät beschädigen. Malware wird installiert, indem man den Nutzer dazu bringt, eine vertrauenswürdig aussehende Applikation zu installieren, oder es wird – wie in sehr vielen Fällen – eine Schwachstelle im Gerät ausgenutzt, z. B. eine Sicherheitslücke im Webbrowser.

Spyware: Das Ziel von Spyware ist es, Informationen über das Opfer zu sammeln und diese dann an die Person, die die Spyware installiert hat, zu senden. Felt et al. behaupten, dass Spyware auf dem Gerät des Opfers installiert wird, indem der Angreifer physischen Zugriff auf das Smartphone hat. Betrachtet man jedoch Beispiele aus den letzten beiden Jahren, so sieht man, dass diese

Voraussetzung nicht mehr erfüllt sein muss und sich solche Tools auch über das Ausnutzen bekannter Lücken oder Social-Engineering-Techniken auf das Zielgerät aufbringen lassen können.

Grayware: Nutzer, die die Software selbst installiert haben, da sie dachten, sie wäre in Ordnung, sollen hiermit ausspioniert oder zu unseriösen Angeboten verleitet werden. Teilweise funktioniert die Software wie gedacht, da die Entwickler reale Funktionen wie versprochen eingefügt haben. Trotzdem sammeln sie Informationen über das System, wie z. B. das Adressbuch des Nutzers oder seine Browser-Historie. Das Hauptziel ist es, Daten für Marketingzwecke etc. zu sammeln oder zusätzliche Werbeeinblendungen vorzunehmen. Man bezeichnet diese Applikationen auch oft als *Pontentially Unwanted Applications (PUA)* oder *Adware*.

Die Verteilung dieser Bedrohungstypen auf schadhafte Applikationen, die in freier Wildbahn gefunden wurden, ist in Abbildung 1–7 zu sehen. Dabei ist zu beachten, dass die meisten schadhaften Applikationen mehr als eine dieser abgebildeten Eigenschaften vereint. Das beliebteste Ziel für Angreifer und Malware-Autoren ist immer noch der Diebstahl von sensiblen Daten wie z. B. Telefonbucheinträge oder Nutzernamen für Online-Plattformen. Direkt danach sehen wir die schadhaften Applikationen, die Eigenschaften eines Botnetzes beinhalten, dazu zählen z. B. Verbindung zu einem C&C-Server, um Befehle in Empfang zu nehmen oder Daten dorthin abfließen zu lassen. Neu dazugekommen sind in 2013 die kommerziellen Malware-Applikationen, die zum Ausspähen von Privatpersonen oder Firmen verwendet werden. Ebenso stark zugenommen hat in den vergangenen zwei Jahren die Art der schadhaften Applikationen, die auf das Abfangen und die Manipulation von mTAN-Nachrichten aus sind. In 2015 kam eine neue Bedro-

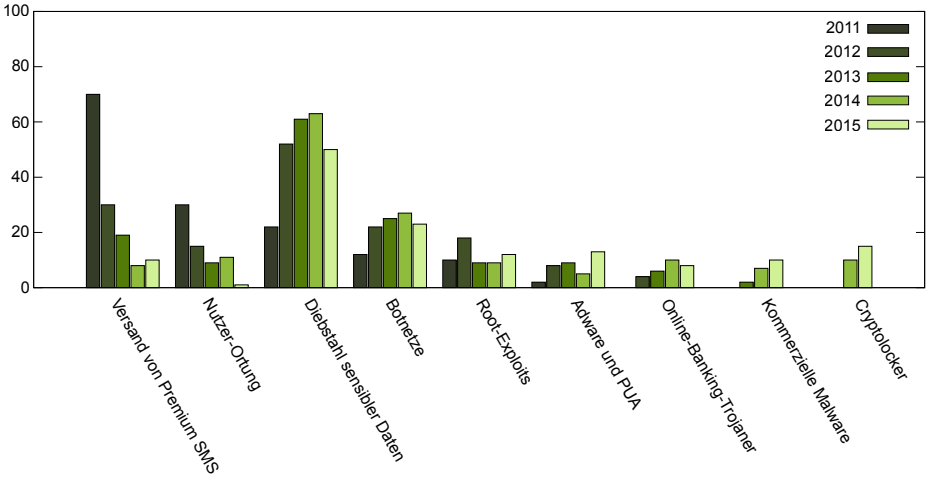

Abb. 1–7: Android-Malware-Verteilung in freier Wildbahn in den vergangenen Jahren

hung hinzu, die *Cryptolocker* bzw. allgemeiner gesagt die *Ransomware*. Diese Malware verschlüsselt das Endgerät oder die Nutzerdaten mit einem Passwort, das der Nutzer nicht kennt. Im Nachgang wird eine Meldung angezeigt, dass der Nutzer einen gewissen Betrag zahlen muss, um das Passwort für seine Daten zu erhalten. Der eigentliche Unterschied zwischen Ransomware und den Cryptolockern liegt hierbei im Detail: Bei gängiger Ransomware werden die Daten des Nutzers nicht verschlüsselt, sondern lediglich das Gerät gesperrt und eine Erpressungsmeldung angezeigt. Diese Masche ist aus der PC-Welt seit einigen Jahren bekannt und gefürchtet und nun auch im Android-Umfeld im Aufschwung. In Abbildung 1–7 haben wir diese beiden Arten von Schadcode unter dem Begriff der Cryptolocker zusammengefasst, da in den vergangenen Monaten fast keine Ransomware mehr erschienen ist, die nicht im Hintergrund die Daten des Nutzers verschlüsselt hat.

(...)