Ein Android-Framework für Angriffsvektoren in mobilen Netzwerken

Bachelorarbeit

von

Fabian Mende

aus

Köln

vorgelegt am

Lehrstuhl für Technik sozialer Netzwerke
Jun.-Prof. Dr. Kalman Graffi
Heinrich-Heine-Universität Düsseldorf

Juni 2016

Betreuer:
Raphael Bialon, M. Sc.

Abstract

Die Datenübertragung über Funknetze nach Standard 802.11 (WLAN) wird immer bedeutsamer und damit zunehmend anfälliger für Störungen und als Ziel von Angriffen. Zum Schutz bedarf es genauer Kontrollen des Netzwerkes und der übertragenen Daten. Um die Überwachung einfach zu ermöglichen, wird im Rahmen dieser Bachelorarbeit ein Framework zum Mitschneiden des Netzwerkverkehrs mit einem Mobilgerät auf Android-Basis entwickelt. Dieses kommt, anders als andere Arbeiten zu dem Thema, ohne die gefährliche Anforderung von Root-Rechten aus und soll auf den meisten Geräten ab Android-Version 4 laufen. Die entwickelte App WTest braucht dazu lediglich eine externe WLAN-Karte mit RTL8187-Chipsatz, die mittels eines USB-Hostadapters angeschlossen wird. Das Framework ist modular erweiterbar und ermöglicht detailliertes Logging der internen Prozesse. Es kann während des Netzwerkmitschnittes gleichzeitig GPS-Koordinaten aufzeichnen, um die Daten besser verwertbar zu machen. Die einzelnen Funktionen sind in Services gekapselt und laufen somit robust und parallel zueinander, wobei sie über das Betriebssystem miteinander kommunizieren.

Danksagung

Viele Personen haben mich auf dem Weg bis zu dieser Arbeit unterstützt und ich möchte mich bei ihnen dafür bedanken.

Zuerst möchte ich meinen Betreuer Raphael Bialon für zahlreiche Tipps, Korrekturen und Ratschläge danken, mit denen er mich bei dieser Arbeit unterstützt hat. Danke auch für seinen Zuspruch und die Hilfe mit Hard- und Software im Verlauf der Arbeit.

Ebenfalls danke ich Andre Ippisch für den ein oder anderen Kniff rund um Android-Programmierung und ihre Möglichkeiten.

Besonderer Dank gebührt Jan Kammers für seine Hilfe mit den Grafiken, das Erstellen der tollen Icons für meine Software und die langjährige Freundschaft.

Meinen Freunden Sebastian und Domenique danke ich für ihre motivierende Freundschaft und all den Kampfgeist in Mathematik und Biologie, mit dem wir alle Herausforderungen gemeistert haben.

Großer Dank gehört meiner Familie für ihre Unterstützung, ausgiebiges Korrekturlesen und ihren Beistand in allen Lebenslagen.

Und ganz besonders danken möchte ich meiner Lebensgefährtin Stephanie Nicolaus, für ihre Geduld und ihr Vertrauen in mich und darauf, dass ich den richtigen Weg immer finden werde.

Danke, dass ihr immer für mich da seid, wenn ich euch brauche.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Codeverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Mobile Datennetze und insbesondere Wireless Local Area Networks (WLANs) nach Standard Institute of Electrical and Electronics Engineers (IEEE) 802.11 werden für die mobile Kommunikation und den Zugriff auf entfernte Daten und das Internet immer wichtiger. Auch dienen sie zunehmend der spontanen Ad-hoc-Verknüpfung verschiedener Geräte und dem Datenaustausch von zahlreichen Sensoren, Geräten und Diensten. Damit steigt automatisch auch die Bedeutung der Verfügbarkeit und der Sicherheit der verwendeten Netzwerke und den ihnen zugrunde liegenden Technologien. Durch die zunehmende Nutzung und Belastung der Netzwerke werden diese anfällig für Störungen durch Überlastung und die Überlappung der verwendeten Funkfrequenzen. Darüber hinaus werden die Netze, aufgrund ihrer gestiegenen Wichtigkeit für viele Dienste, mehr und mehr Ziel von Angriffen mit der Absicht, sie zu stören, zu unterbrechen oder die übertragenen Daten unerlaubt mitzulesen oder zu manipulieren.

Für einige dieser Probleme gibt es technische Lösungen, bei anderen lässt sich nur der Einfluss auf das Netzwerk durch hohe Robustheit oder Ausweichstrategien vermindern. Um die geforderten Ansprüche an das Netzwerk und die Umsetzung und Effektivität der Schutzmaßnahmen zu gewährleisten und zu prüfen, braucht es spezielle Software

und teilweise auch Hardware zur Messung und Überwachung der Netzwerke. Hierfür kann man einerseits selbst versuchen das Netzwerk anzugreifen, insbesondere aber muss man das Netzwerk und seine Reaktion auf Angriffe genauestens beobachten. Größtenteils lässt sich dies heute schon mit einem handelsüblichen Laptop und einer geeigneten Netzwerkkarte prüfen, andererseits reicht diese Ausrüstung häufig auch für potenzielle Angreifer aus. Meist verwendet man hierbei ein *Linux*- oder *Unix*-basiertes Betriebssystem und Software wie *aircrack-ng* [Air16], *mdk3* [mdk14], *Kismet* [Kis16] oder zum Mitlesen des Traffics *tcpdump* [tcp15] und *Wireshark* [Wir16].

In den letzten Jahren verbreiten sich auch Smartphones und Tablets immer mehr und werden deutlich leistungsstärker. Da viele und insbesondere technikaffine Personen bereits ein Smartphone besitzen und das Abhören von Netzwerken mit diesen Geräten deutlich bequemer und unauffälliger wäre, stellt sich nun die Frage, ob ein entsprechender Einsatz einfach zu bewerkstelligen ist. So könnten interessierte Sicherheitsforscher schon durch die Installation einer App ihr bereits vorhandenes Gerät in eine mobile Testplattform verwandeln.

Das Betriebssystem *Android* ist gut abgesichert und verhindert größere Eingriffe in die Netzwerkoptionen wirkungsvoll. So ist es nicht möglich, beliebige Sockets zu erzeugen oder die Netzwerk-Chips in den Monitor-Mode zu versetzen, also sämtliche empfangene Frames sichtbar zu machen. Diese Funktionen benötigen zum Beispiel tcpdump und Wireshark, sowie einige Angriffsvektoren.

In dieser Arbeit wird ein Framework entwickelt, mit dem sich Android-Geräte möglichst bequem und einfach zu mobilen WLAN-Testplattformen erweitern lassen. Hierbei liegt das Augenmerk darauf, dass keine riskanten Änderungen am Gerät notwendig werden, sondern das Gerät im Werkszustand genutzt werden kann. Das Framework soll auch als modulare Grundlage für mögliche Erweiterungen zu einem späteren Zeitpunkt dienen. Zumindest sollen Netzwerke per Monitor-Mode überwacht werden können, idealerweise auch aktiv angegriffen werden. Benötigt wird diese Anwendung unter anderem, um die vom Lehrstuhl entwickelte App *Opptain* und die von ihr verwendeten Konzepte auf Robustheit zu untersuchen. Opptain dient der Verknüpfung von Mobilgeräten zu einem opportunistischen Netzwerk mittels WLAN. Die Teilnehmer verbinden sich, wenn sie sich begegnen, ad-hoc zu einem größeren Netzwerk. Durch die sporadischen Kontak-

te der sich in Bewegung befindlichen Knoten, soll dies den Austausch von Daten über größere Distanzen ermöglichen, ohne zentrale Infrastruktur zu benötigen.

1.2 Aufbau der Arbeit

In dieser Arbeit wird ein Android-Framework für Angriffsvektoren in WLANs erstellt. Das besondere Augenmerk hierbei liegt auf dem Verzicht von Root-Rechten (siehe Abschnitt 3.2) auf dem ausführenden Gerät, sowie der guten Erweiterbarkeit des Frameworks.

In Kapitel 2 beschreibe ich verschiedene Lösungswege anderer Autoren, mit denen sie die Hauptanforderungen von Angriffsvektoren, nämlich den Monitor-Mode von WLAN-Karten und Packet-Injection, erreicht haben. Allerdings verwenden alle Methoden, außer einer, Root-Rechte um ihr Ziel zu realisieren.

Kapitel 3 erklärt grundlegende Begriffe, erläutert Schwierigkeiten der gewählten Voraussetzungen und beschreibt in der Arbeit verwendete Module anderer Entwickler.

In Kapitel 4 geht es im Detail um die gewählte Lösung und wie das Framework implementiert wird. Es wird auf die benötigten Anpassungen der verwendeten Software-Module und den Funktionsumfang des entwickelten Frameworks eingegangen.

Anschließend werden in Kapitel 5 die mit dem Framework durchgeführten Tests beschrieben und deren Ergebnisse ausgewertet. Hierbei werden verschiedene Messungen zu Durchsatz der Paketmitschnitte und sonstigen Leistungsparametern des Frameworks betrachtet.

Abschließend wird in Kapitel 6 zusammengefasst, auf welchem Stand das Framework ist und wie das Fazit der Tests bezüglich seiner Einsatzfähigkeit ist. Zuletzt wird ein Ausblick auf mögliche Erweiterungen des Frameworks und weiterführende Themen gegeben.

Alle verwendeten Abkürzungen finden sich im Abkürzungsverzeichnis und verwendete Fachbegriffe sind im Glossar erläutert. Namen und Befehle werden bei ihrem ersten Auftreten im Text *kursiv* geschrieben.

Kapitel 2

Verwandte Arbeiten

Zur Erweiterung eines Android-Gerätes um Möglichkeiten der Überwachung oder des aktiven Angreifens von WLANs gibt es verschiedene Lösungsansätze, von denen jedoch die meisten bei genauerer Prüfung sehr ähnlich sind. Vorschläge hierfür finden sich zum Beispiel in [Ans15]. Im Folgenden beschreibe ich Varianten von Kernel-Patches, eine Lösung durch ein komplettes Linux, sowie den Monitor-Mode durch eine per Universal Serial Bus (USB) verbundener WLAN-Karte mittels *Android PCAP*. Weitere vergleichbare, aber von mir nicht näher beschriebene Ansätze finden sich in [mic12], [Dra12] und [ciu15].

2.1 Kernel-Patches mit modifizierten Treibern

2.1.1 Packet-Injection und Monitor-Mode für HTC Hero

Für den in [bla11] beschriebenen Ansatz wird auf einem gerooteten Smartphone vom Typ *HTC Hero* der *Kernel*, also das Herzstück des Betriebssystems, im Recovery-Mode mit einem speziell präparierten Code geflasht. Mittels Skripten wird anschließend ein gepatchtes WLAN Modul geladen, mit dem dann die Packet-Injection aktiviert werden kann. Meist wird auch noch ein angepasstes *Debian* installiert, mit dem *aircrack-ng* und

Kismet aufgeführt werden können. Hiermit sind vielfältige aktive und passive Angriffe möglich.

Nachteil jedoch ist, dass ein spezielles Smartphone (HTC Hero) benötigt wird, auf welchem man den Rootzugriff aktiviert haben muss. Zusätzlich werden aufwendige Veränderungen am System notwendig und der Autor weist ausdrücklich auf die Instabilität der Lösung hin. Da für diese Arbeit mit einem ungerooteten Gerät gearbeitet werden und die Software einfach zu bedienen sein soll, ist dieses Vorgehen keine Option.

2.1.2 Aircrack-ng auf einem Galaxy Note2

Dieser Ansatz [Moi13] verwendet ein *Galaxy Note2* mit modifiziertem Kernel und speziellen Treiber für *CyanogenMod* 10.1. CyanogenMod [Cya16] ist eine angepasste Firmware und ein Betriebssystem, basierend auf einem Open-Source-Fork von Android. Dieses enthält viele Funktionen, die in dem standardmäßig auf dem Smartphone installierten Betriebssystem nicht enthalten sind. Durch die Installation von CyanogenMod 10.1 mit Root-Rechten wird es hier möglich, spezielle Treiber in den Kernel zu injizieren. Mit diesen lässt sich dann eine per USB angeschlossene WLAN-Karte vom Typ *Alfa AWUS036H* oder *TP-LINK TL-WN722N* betreiben. Das ermöglicht die Benutzung von aircrack-ng auf dem Smartphone, zum Beispiel zum Knacken der Wired Equivalent Privacy (WEP)-Verschlüsselung eines WLANs.

Auch hier wird sehr spezifische Hardware benötigt, welche mit Root-Rechten und einer veränderten Firmware betrieben wird. Dies ist mit viel Know-how und Aufwand verbunden und und schränkt die Einsatzmöglichkeiten stark ein. Damit ist auch dieser Ansatz kein valider Weg für die angestrebte Lösung, da die benötigten Eingriffe in das Betriebssystem nicht erwünscht sind.

2.1.3 BCMON

BCMON [IOF13] ist ein Projekt von drei israelischen Technikern, die eine Möglichkeit entwickelt haben, auf den WLAN-Chips *BCM4329* und *BCM4330* den Monitor-Mode zu aktivieren und unter bestimmten Bedingungen Raw-Packet-Injection zu ermöglichen. Wie in [bcm13] beschrieben, waren dabei zahlreiche Versuche des Reverse Engineerings nötig, um mittels *Assembler*, *C* und *Python* Patches für den Kernel zu entwickeln. Mittlerweile haben die Entwickler die nötigen Patches in der App BCMON [bcm12] zusammengeführt. Getestet wurde die App auf einigen wenigen Smartphone-Modellen mit dem passenden WLAN-Chipsatz (siehe [IOF13]). Um diese App installieren und ausführen zu können, benötigt man allerdings CyanogenMod (siehe Abschnitt 2.1.2) mit Root-Rechten (siehe Abschnitt 3.2). Ein weiterer Entwickler hat, aufbauend auf der BCMON Lösung, noch eine App namens *Reaver* entwickelt, mit der sich WiFi Protected Access (WPA)-Netzwerke knacken lassen [pur15]. Auch die Version *AircrackGUI 1.2* von *deviato* [dev13] verwendet BCMON, um aircrack-ng auszuführen. Natürlich bleiben auch bei diesen Erweiterungen die Geräteanforderungen von BCMON erhalten, da immer zunächst BCMON installiert und ausgeführt wird.

Die kritischen Probleme bei dieser Lösung sind der Bedarf von Root-Rechten sowie die Benutzung von CyanogenMod. Da in dieser Arbeit auf das Installieren von veränderter Firmware und Betriebssystemen verzichtet werden soll und ohne Root-Rechte gearbeitet wird, kommt auch dieser Ansatz nicht als Basis für mein Framework in Frage. Des Weiteren sind die beiden unterstützten WLAN-Chipsätze schon einige Jahre alt und schränken die Wahl der verwendeten Smartphones zunehmend ein.

2.2 Kali Linux Nethunter

Kali Linux Nethunter ist laut der Webseite [Kal16a] eine komplette Penetrationstest-Plattform auf Android. Das verwendete Linux basiert auf einem Nachbau der auf Penetrationstest spezialisierten *BackTrack* Linux-Distribution. Dadurch ist Kali Linux Nethunter von den in dieser Arbeit vorgestellten Lösungen die umfangreichste und wird von

Offensive Security professionell betreut und weiterentwickelt. Sie enthält Funktionen wie den Monitor-Mode, 802.11 Frame-Injection und das Ausführen einiger bekannter Exploits, also gezielter Angriffe auf Schwachstellen. Darüber hinaus enthält die Plattform viele weitere Module für Angriffe auf IT-Geräte, wie Tastaturen und USB-Geräte.

Die für das jeweilige Gerät angepassten Pakete und der freie Quellcode sind auf *Github* [Kal16b] verfügbar. Laut der dort veröffentlichten Liste läuft Kali Linux Nethunter auf allen aktuellen *Nexus*-Geräten sowie einigen anderen Smartphone- und Tablet-Modellen. Die Software wird mittels einem speziellen Recovery-Tool auf das gerootete Android oder CyanogenMod installiert und modifiziert dort das Betriebssystem. Zusätzlich zum Smartphone wird eine externe USB-WLAN-Karte benötigt, wovon verschiedene gängige Modelle unterstützt werden [Kal16c]. Ähnlich zu der relativ eleganten Lösung von Kali Linux Nethunter gibt es auch die etwas aufwändigere, in Artikel [zit13] beschriebene Möglichkeit, um sein Smartphone zu einer Angriffsplattform auszubauen.

Genau wie bei den anderen bisher vorgestellten Ansätzen, benötigt auch die Kali Linux Nethunter Software Root-Rechte, um den Kernel zu verändern und ein Linux zu starten. Damit übertrifft die Software vom Funktionsumfang zwar alle Erwartungen, fällt aber aufgrund der Ausgangsbedingungen ebenfalls heraus.

2.3 Android PCAP

Android PCAP [And13] ist eine Open-Source-Software von *Kismet Wireless* und der Versuch, das bekannte Packet CAPture (PCAP) unter einem werksmäßigen Android ohne besondere Rechte lauffähig zu machen. PCAP ist eine Open-Source-Programmierschnittstelle zum Mitlesen von Netzwerkverkehr. Unter Linux wird sie durch die Bibliothek *libpcap* implementiert [tcp15]. Mit ihr lassen sich Netzwerkmitschnitte in der standardisierten Form *.cap erstellen, welche dann zum Beispiel mit Wireshark visualisiert und interpretiert werden können. Zunächst hat Kismet Wireless den Kernel-Treiber für eine Netzwerkkarte mit dem Chipsatz *RTL8187* im Java-Userspace nachgebaut, um eine per USB-Hostadapter angeschlossene WLAN-Karte verfügbar zu machen. Diese

wird von der Software zusammen mit der Linux-Bibliothek libpcap verwendet, um die empfangenen Frames mitzuloggen. Mit dieser einfach zu bedienenden App lassen sich ausschließlich Mitschnitte erstellen und somit nur passive Angriffe ausführen, da der Anwender keine Sendemöglichkeiten hat. Die Software ist leider nur rudimentär entwickelt und kann lediglich auf einem eingestellten Channel Frames mitloggen. Hierbei funktionieren angebotene Funktionen teils nur sporadisch oder gar nicht. Auch ist die Anwendung nicht besonders stabil und neigt zu Abstürzen. Des Weiteren hat der Benutzer nur wenig Kontrolle darüber, was intern abläuft, da kaum Logeinträge erstellt werden und vieles dadurch schleierhaft bleibt.

Dennoch stellt Android PCAP den Ausgangspunkt für das in dieser Arbeit entwickelte Framework dar, da immerhin der Zugriff im Monitor-Mode auf eine WLAN-Karte ohne Root-Rechte funktioniert. Probleme mit der im Grundgerüst enthaltenen Android PCAP Software können dank des offenen Quellcodes teilweise behoben oder umgangen werden. Weitere Informationen zu Android PCAP finden sich im Abschnitt 3.3.

2.4 Zusammenfassung

Einige der vorgestellten Möglichkeiten, um ein androidbasiertes Gerät mit Monitor-Mode oder Packet-Injection zu versehen, sind sehr interessant und im Falle von Kali Linux Nethunter auch mächtig und relativ komfortabel. Jedoch benötigen alle, außer Android PCAP, Root-Rechte, da sie grundlegende Modifikationen des Betriebssystem und des Kernels beinhalten. Somit kommt für diese Arbeit nur Android PCAP als Ansatz in Frage und wird im folgenden Kapitel näher beschrieben.

Kapitel 3

Grundlagen

In diesem Kapitel werden zunächst einige in der Arbeit verwendete Begriffe und Konzepte näher erläutert. So werden die Grundlagen von Monitor-Mode und Packet-Injection erklärt und beschrieben wie PCAP und Java Native Interface (JNI) funktionieren. Anschließend beschäftigt sich Abschnitt 3.2 mit dem Hintergrund und der Gefahr von Rootrechten, und warum das erstellte Framework ohne sie laufen soll. Da von den verwandten Arbeiten lediglich Android PCAP als Ansatz in Frage kommt und es die Grundlage für das entwickelte Framework WTest darstellt, behandelt Abschnitt 3.3 im Detail die Grundlagen, Funktionsweise und Probleme von Android PCAP. Zuletzt folgt eine kurze Einführung in das Logging-Framework Logback, da dies für das applikationsinterne Logging von WTest verwendet wird.

3.1 Begriffsdefinition

3.1.1 Monitor-Mode

Nach [mon16] bezeichnet der Monitor-Mode einen von sechs möglichen Modi, in denen eine WLAN-Karte mit Standard 802.11 arbeiten kann. Sie leitet in diesem Modus alle empfangen Frames ohne Prüfung an das Betriebssystem weiter. Also auch Frames von

benachbarten Kanälen und mit defekter Prüfsumme. In diesem Modus wird nicht auf Service Set IDentifier (SSID) oder Internet Protocol (IP)-Adresse gefiltert, sondern man liest alle Frames, deren Funksignale empfangen werden. Die Frames werden komplett inklusive aller Header an das Betriebssystem weitergeleitet, anstatt wie in den anderen Modi nur den Ethernet-Frame weiterzureichen. Im Monitor-Mode darf die Karte nicht mehr senden, sondern nur noch empfangen. Dies ermöglicht es, unbemerkt Netzwerke und Funkverkehr zu belauschen und Pakete von Netzwerken zu empfangen, deren Passwort für die Authentifikation unbekannt ist. Wie [Par08] beschreibt, muss sich die Karte im Promiscuous Mode (ein weiterer der sechs Modi) hingegen erst mit einem Netzwerk assoziieren und nur der in diesem Netzwerk empfangene Verkehr wird an das System weitergeleitet. Auch beim Promiscuous-Mode müssen die Frames nicht genau an das empfangende Gerät gerichtet sein.

3.1.2 Packet-Injection

Unter Packet-Injection versteht man den Vorgang, in beliebige Netzwerkverbindungen eigene Pakete einzuschleusen, von denen der Empfänger glaubt, sie kämen von seinem Partner in der Verbindung. Diese Pakete kann man auf seinem Gerät mit beliebigen Inhalt befüllen und sendet hierbei häufig Kombinationen von Informationen, die auf dem eigenen Gerät eigentlich ungültig sind. So kann man, wie zum Beispiel in [KS04] beschrieben, andere Geräte in deren Namen von WLAN Access-Points abmelden. Packet-Injection wird daher häufig für Angriffe verwendet, aber es gibt auch einige halbwegs legitime Anwendungszwecke [Sch07]. Für Angriffe möchte man Packet-Injection häufig auch ausführen, wenn die eigene WLAN-Karte sich im Monitor-Mode befindet und eigentlich nicht senden darf.

3.1.3 PCAP

Packet CAPture (PCAP) ist eine Open-Source-Programmierschnittstelle zum Mitschneiden von Netzwerkverkehr. Unter Linux wird sie durch die Bibliothek libpcap implemen-

tiert [tcp15]. PCAP kann Netzwerkmitschnitte in Form von *.cap und *.pcap Dateien erstellen und diese wiedergeben. Die erstellten Mitschnitte können dann mit Programmen, wie dem auf PCAP aufbauenden Wireshark, angezeigt, gefiltert, manipuliert und interpretiert werden. Zum Mitschneiden von Netzwerkverkehr muss eine passende Netzwerkkarte zur Verfügung stehen, wobei diese den Monitor-Mode beherrschen muss [pca15]. Dann schaltet sich PCAP zwischen die Karte und das Betriebssystem und schreibt alle empfangen Frames in das Logfile. PCAP bildet auch die Grundlage für den Logger vieler Netzwerktools wie tcpdump, Wireshark und *Snort*.

3.1.4 JNI

Das Java Native Interface (JNI) ist eine Schnittstelle, um Java-Programmcode mit dem Code anderer Sprachen zu verknüpfen. Wie in [JNI16b] und [JNI16a] beschrieben, lassen sich hierüber Funktionen und Programme, welche in anderen Sprachen wie zum Beispiel C oder Assembler geschrieben sind, aus dem Java-Code heraus aufrufen und umgekehrt. Dazu lädt man in Java die jeweilige Bibliothek des anderen Programmes und deklariert die aufzurufenden Methoden als native. Nun generiert man zum Beispiel in C eine Adapterklasse und legt dort eine Methode mit dem zu der Java-Methode passenden, voll qualifizierten Namen an. Dann kann Java die erstellte Methode aufrufen und das aufgerufene Programm von dort auf bestimmte Methoden und Variablen in Java zugreifen. Den genauen Ablauf können interessierte Leser in [Ull12, Kapitel 21] nachlesen.

3.2 Root-Rechte unter Android

Für diese Arbeit wurde die Entscheidung getroffen, zu versuchen den gewünschten Funktionsumfang des Frameworks von Monitor-Mode und Exploit-Ausnutzung ohne Root-Rechte zu erreichen. Grund hierfür ist, dass ein Smartphone so benutzt werden soll, wie es ab Werk kommt, ohne erst kritische Eingriffe in das Betriebssystem und die Firmware vorzunehmen. Darüber hinaus ist das Erstellen eines Exploit-Frameworks mit Root-Rechten relativ simpel, da man mit Kali Linux Nethunter (siehe Abschnitt 2.2) ein kom-

plettes Linux mit allen Möglichkeiten zur Verfügung hat. Somit war es nun an der Zeit zu schauen, was man ohne die erweiterten Rechte erreichen kann, denn eine solche Anwendung wäre deutlich attraktiver. Viele Nutzer scheuen sich mit gutem Recht, ihr Gerät zu rooten und könnten in Zukunft durch eine einfache App erreicht werden.

3.2.1 Risiken durch Root-Rechte

Einige der mit Root-Rechten verbundenen Risiken werden in [Cum15] und [Joo15] beschrieben, woraus ich im Folgenden ein paar Beispiele entnehme. Schon beim Vorgang des Rootens kann vieles schief gehen und das Gerät sogar irreparabel beschädigt werden. Sobald man auf dem Gerät Root-Rechte erlangt hat, kann man sämtliche Daten des Gerätes verändern und bekommt Zugriff auf zahlreiche versteckte Funktionen. Allerdings kann dadurch, insbesondere bei unsachgemäßer Bedienung, auch viel zerstört werden. So kann man zum Beispiel das ganze Betriebssystem durch eine veränderte Version wie CyanogenMod austauschen und muss sich in Zukunft selber um Updates kümmern. Das Risiko für die Infektion mit Viren und Trojanern steigt an, da nun Apps aus unsicheren Quellen installieren werden können, wobei diese zusätzlich in geschützte Systembereiche schreiben und die erweiterten Funktionen nutzen dürfen. Auch ist es möglich, das Gerät schlimmstenfalls durch falsche Einstellungen unwiederbringlich zu zerstören oder zumindest zu beschädigen. Und in vielen Fällen verliert man den Support durch den Hersteller, wenn man sein Gerät einmal rootet. Zusätzlich kann man die Root-Rechte nicht nur einer App erteilen, sondern muss sie sich zunächst systemweit beschaffen. Dies alles sind gute Gründe, nach Möglichkeit auf Root-Rechte als Bedingung für eine App zu verzichten.

3.2.2 Herausforderung ohne Root-Rechte

Da Android auf Linux aufbaut, ist es laut der Definition [Lin16] von Raw-Socket und [Moo09] nur Benutzern mit Root-Rechten erlaubt, Raw-Sockets zu erstellen. Diese benötigt man teilweise aber für den Monitor-Mode und Packet-Injection, denn es sollen Pa-

kete auf allen Ports empfangen und gesendet werden können. Leider bietet auch Java laut Oracle [raw16] keine Raw-Sockets an. Somit müsste der Zugriff auf einen Raw-Socket von Android mittels JNI realisiert werden. Dies scheitert aber aufgrund des Verzichts auf Root-Rechte.

Des Weiteren unterstützen die Treiber im Android-Kernel für die WLAN-Chips keinen Monitor-Mode oder Packet-Injection. Aus diesem Grund patchen die in Abschnitt 2.1 vorgestellten Lösungen erweiterte Treiber in den Kernel, beziehungsweise tauschen den Kernel gegen eine für ihre Zwecke verbesserte Version aus. Um einen solchen Patch mit neuen Treibern vorzunehmen, benötigt man allerdings auch Root-Rechte, womit diese Option entfällt.

Die beiden einfachsten Möglichkeiten, sowie zahlreiche tiefer greifende Modifikationen des Betriebssystems zum Ermöglichen der Hauptfunktionalität des angestrebten Frameworks, kommen somit nicht in Frage. Aus diesem Grund ist das im Folgenden beschriebene Android PCAP ein besonders interessanter Ansatz das Problem zu lösen.

3.3 Android PCAP

3.3.1 Allgemeine Funktionsweise

Android PCAP [And13] ist eine App von Kismet Wireless für Smartphones mit Android ab Version 4. Mit ihr lassen sich Netzwerkmitschnitte im standardisierten PCAP-Format erstellen. Hierfür wird lediglich ein geeignetes Smartphone mit vollständiger USB On-The-Go (USB-OTG)-Unterstützung, ein USB-OTG-Kabel, sowie eine USB-WLAN-Karte mit RTL8187 Chipsatz benötigt. USB-OTG ist ein Standard, der es dem Smartphone erlaubt als USB-Host zu agieren und somit externe Geräte wie zum Beispiel USB-Sticks, USB-Tastaturen oder besagte Netzwerkkarten zu benutzen.

Das Besondere an Android PCAP ist der Verzicht auf Root-Rechte und gleichzeitig die Funktion eine WLAN-Karte in den Monitor-Mode zu versetzen. Hierfür hat Kismet Wi-

reless den Linux Kernel-Treiber des RTL8187 Chips teilweise in Java nachgebaut, sodass dieser im Userspace läuft. Hierbei wird die WLAN-Karte über die Android USB-Host Application Programming Interface (API) angesprochen. Mittels JNI wird eine importierte Linuxbibliothek von PCAP, die libpcap-1.3.0 geladen und zum Mitschneiden der empfangenen Frames verwendet.

3.3.2 Interner Programmaufbau

Der Quellcode ist Open-Source und steht auf dem Git-Portal von Kismet Wireless zum Download bereit. Bei genaueren Analysen des Codes zeigt sich die im Folgenden beschriebene Funktionsweise.

Zunächst startet, wie bei den meisten Android-Apps, die *MainActivity*, welche die Benutzeroberfläche initialisiert und anzeigt. Sie nimmt auch die Benutzereingaben entgegen und zeigt den Status der App an. Während des Starts wird ein separater Service für das eigentliche Logging mittels PCAP erstellt und gestartet. Der *PcapService* und die MainActivity kommunizieren miteinander über *Messages*, sobald die MainActivity sich beim Service registriert hat. Dieser Service prüft auf angeschlossene USB-Geräte und bekommt von der MainActivity mitgeteilt, wenn neue Geräte angesteckt oder angeschlossene entfernt werden. Wenn eines der geprüften Geräte eine gültige WLAN-Karte ist, wird diese als neue USB-Karte initialisiert und ein passender *Logger* erstellt. Dieser *PcapLogger* enthält die benötigten nativen Methoden für den JNI-Zugriff auf die mitgelieferte libpcap Bibliothek. Mit diesen Methoden kann die jeweilige Funktion der Adapterklasse in C aufgerufen werden, mit der die Bibliothek angesprochen wird. Wenn sich der Status des Services oder der angeschlossenen Karten ändert, informiert der PcapService die bei ihm registrierte MainActivity, damit diese die Benutzeroberfläche entsprechend anpassen kann.

Wenn der Benutzer nun den Logvorgang startet, sendet die MainActivity eine Message an den PcapService und dieser startet mittels PcapLogger die libpcap Funktionen. Über eine Helferklasse liest der PcapService regelmäßig die Menge der bisher geloggten Pakete aus und sendet ein Statusupdate an die MainActivity. Wenn der Benutzer den Logvorgang

stoppt, wird wiederum der PcapService benachrichtigt und dieser stoppt den Logger.

Zusätzlich kann der Benutzer verschiedene Parameter des Loggings konfigurieren, woraufhin sich eine Einstellungs-Oberfläche öffnet und bei deren Schließen der PcapService über die Änderungen benachrichtigt wird. So kann die WLAN-Karte entweder auf einen bestimmten Channel konfiguriert oder sogenanntes Channelhopping aktiviert werden, bei dem die Karte den Channel alle 250 ms wechselt. Beim Channelhopping kann ausgewählt werden, welche Channel angesprungen werden sollen. Über diese Basisfunktionalität hinaus hat Android PCAP noch die Funktion, bisher erstellte Logdateien zu löschen oder mit anderen Anwendungen zu teilen.

3.3.3 Probleme von Android PCAP

Wie sich im Laufe der Zeit bei der Arbeit mit Android PCAP herausstellt, läuft Android PCAP leider nicht besonders stabil und viele der Optionen funktionieren nicht. So bricht das Logging nach einem Moment ab, wenn das Display abgeschaltet wird und funktioniert sporadisch gar nicht. Letzteres liegt daran, dass die WLAN-Karte manchmal nicht erkannt wird oder sich in einigen Fällen nicht richtig initialisiert, beziehungsweise kalibriert. Auch funktioniert Channelhopping nicht, sondern die Karte verharrt in dem eingestellten Channel, ohne jemals zu wechseln. Bei der Änderung von Optionen durch den Benutzer konfiguriert sich die App nicht immer um. Ohne Änderung des Channels nach jedem Start der App bleibt sie aber nicht im gespeicherten Channel, sondern stellt immer Channel 6 ein. Zusätzlich weisen die Entwickler von Android PCAP auf den deutlich höheren Stromverbrauch durch die externe WLAN-Karte hin, der sich durch Verwendung eines Y-Kabels mit Anbindung einer Stromquelle aber reduzieren lässt. Diesen erhöhten Stromverbrauch werde ich in späteren Tests überprüfen.

Trotz aller Probleme von Android PCAP zeigt die Software, dass die Idee prinzipiell funktioniert. Die beschriebenen Bugs lassen sich dank Open-Source beheben und die Software in einem eigenen Framework erweitern. Daher werde ich Teile von Android PCAP in meinem Framework für das Loggen von Netzwerkpaketen nutzen.

3.4 Logback

Logback ist ein sehr mächtiges, flexibles und doch einfach zu benutzendes Logging-Framework für Java. Es ist eine Weiterentwicklung von *Logging Utility for Java (log4j)* und implementiert die API von *Simple Logging Facade for Java (SLF4J)*. Nach [SLF16] ist SLF4J eine standardisierte Schnittstelle verschiedener Logging-Frameworks zur einfachen Einbindung des Frameworks in die eigene Software.

Nachdem Logback in der Software zur Laufzeit einmal initialisiert ist, kann jede Klasse einfach auf das Logging zugreifen. In der Klasse können dann wie gewohnt Logeinträge geschrieben werden, die zentral an Logback gesendet werden. Hierbei kümmert sich Logback dann darum, nur die Einträge des gewünschten Loglevels zu schreiben und diese entsprechend der jeweiligen Konfiguration zu formatieren. Logback kann gleichzeitig in verschiedene Formate, Ziele und auf verschiedenen Leveln loggen und auf Wunsch asynchron, also nicht blockierend, arbeiten. Durch die zentrale Verwaltung des Frameworks können jederzeit Quellen oder Ziele für die Logeinträge hinzugefügt oder entfernt werden, ohne den laufenden Betrieb des sonstigen Loggings zu stören.

Interessierte Leser können unter [Log16] die komplette Dokumentation, sowie eine Vielzahl hilfreicher Artikel und Anleitungen rund um Logback finden.

Kapitel 4

Implementierung des Frameworks

4.1 Allgemeines zum Framework

Das von mir erstellte Framework WTest ist eine modular aufgebaute Basis, um WLANs zu überprüfen. Hierbei liegt das Augenmerk auf einfacher Handhabung der App, sowie guter Erweiterbarkeit der Funktionalität. Die internen Abläufe sollen für den versierten Benutzer so nachvollziehbar wie möglich sein und werden zur Laufzeit ausführlich protokolliert. Weitere Funktionen können leicht per zusätzlichem *Service* eingebaut und über den Hauptbildschirm konfiguriert werden.

Der aktuelle Funktionsumfang umfasst das Mitschneiden von Netzwerkverkehr in 802.11 WLANs mittels Monitor-Mode, bei gleichzeitigem Aufzeichnen der Global Positioning System (GPS)-Koordinaten und Dokumentation der internen Abläufe.

4.1.1 Voraussetzungen der App

Um die App zu betreiben, benötigt man ein Smartphone oder ein vergleichbares Gerät mit Android ab Version 4, welches USB-OTG vollständig unterstützt. Getestet ist dies

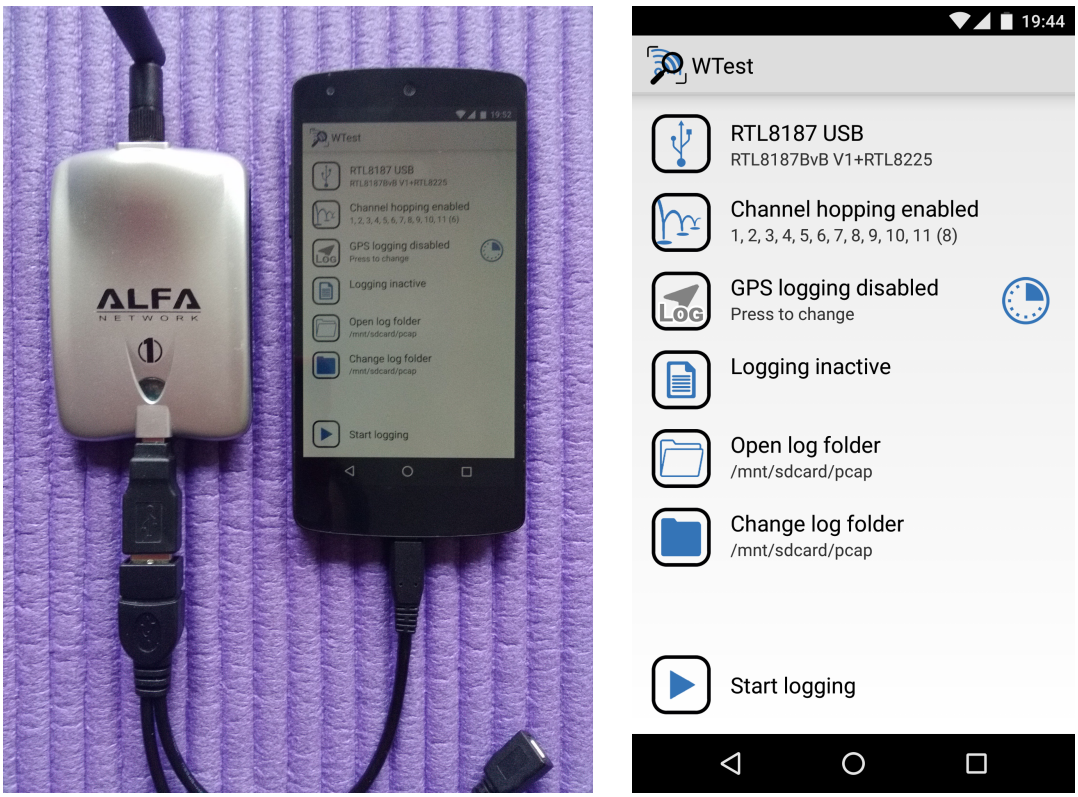
mit verschiedenen Nexus-Geräten, aber auch zahlreiche andere Modelle sollen funktionieren. Zusätzlich bedarf es zum Loggen von Netzwerkverkehr einer externen WLAN-Karte mit RTL8187-Chipsatz. Die WLAN-Karte wird mittels USB-OTG-Kabel mit dem Gerät verbunden, auf dem zuvor das Framework durch eine Android application package (*.apk)-Datei installiert wird. In Zukunft wird dies auch direkt über den *Google Play Store* möglich sein. Als WLAN-Karte eignet sich etwa die Alfa AWUS036H oder die *Alfa One*, welche schon für knapp 30 Euro erhältlich sind.

4.1.2 Grundlegende Aufteilung des Frameworks

Beim Starten der App wird zunächst die *Application*-Klasse geladen, in welcher das auf Logback basierende Loggingframework konfiguriert und gestartet wird. Nach diesem Initialisierungsschritt wird der zentrale Knotenpunkt des Frameworks gestartet - die *MainActivity*. Diese lädt die gespeicherten *Preferences*, startet die benötigten Services und kümmert sich um die Anzeige der Graphical User Interface (GUI).

Abbildung 4.1 zeigt den Aufbau mit Smartphone und WLAN-Karte sowie einen Screenshot der GUI. Hierbei läuft die App auf einem Nexus 5 mit Android 6 und der Alfa AWUS036H. In der GUI sieht man den Hauptbildschirm vor dem Logging. Durch Auswählen des Channelhoppings öffnet sich eine weitere Ansicht mit Einstellungsmöglichkeiten. Die anderen Zeilen öffnen einen kleinen Dialog zur Eingabe oder wechseln die Einstellung direkt.

Das Mitschneiden der Netzwerkpakete und Aufzeichnen der GPS-Koordinaten läuft jeweils in eigenen Services und ihren Helferklassen ab. Diese werden von der MainActivity gestartet und mittels Kommunikation über *Messages* gesteuert. Messages und *Intents* sind vom Android-System standardisierte Formate zur Kommunikation von Anwendungsteilen und können zusätzlich ein *Bundle* mit Daten enthalten. Sie werden von einem Sender an das Betriebssystem übergeben und in den jeweiligen Programmteilen über einen angepassten *Handler* empfangen. Der Vorteil von Services ist, dass diese in ihrer Ausführung den Hauptprozess nicht blockieren und auch Aktionen wie das Wechseln zwischen Apps und das Ausschalten des Displays überstehen, wobei die MainAc-



(a) Aufbau der Hardware (b) Screenshot der GUI

Abbildung 4.1: Aufbau der Hardware und Screenshot der GUI

tivity zeitweise gestoppt wird. Auch können auf diese Weise neue Funktionen einfach über zusätzliche Services in die App eingebaut werden.

Im Folgenden werde ich die grundlegenden Anwendungsteile wie MainActivity, internes Logging, sowie Netzwerkmitschnitt via PCAP und die GPS-Aufzeichnung genauer erläutern. Hierbei werde ich auch auf die jeweiligen Schwierigkeiten bei der Anpassung verwendeter Komponenten eingehen. Abbildung 4.2 zeigt eine Übersicht der Klassen von WTest mit ihren Zusammenhängen.

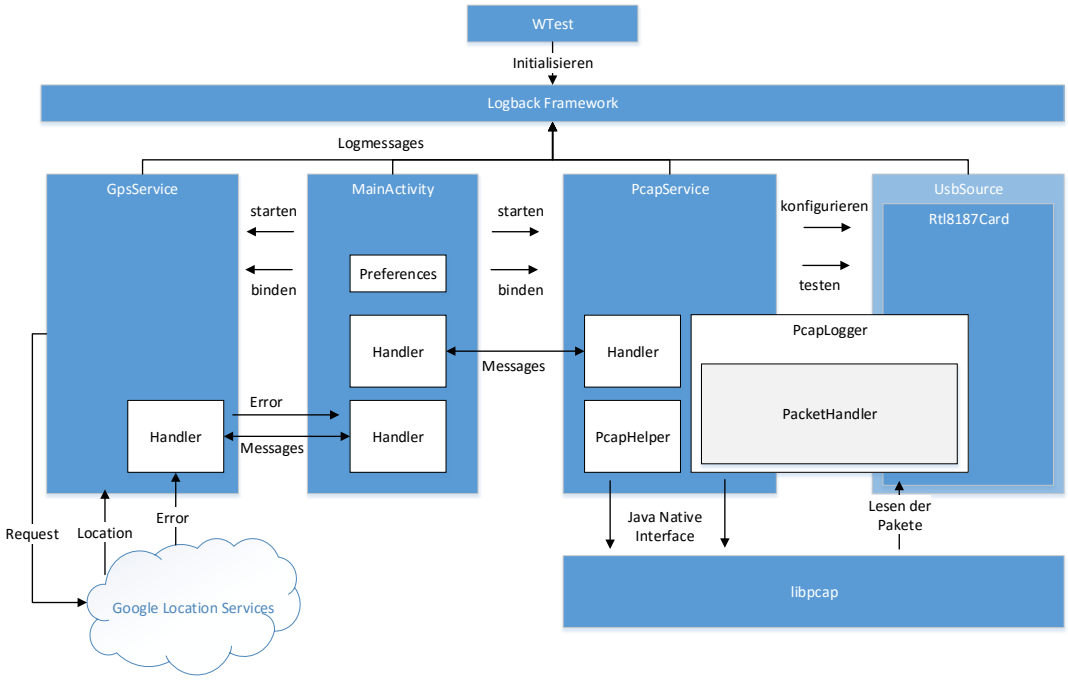


Abbildung 4.2: Übersicht der Klassen von WTest

4.2 Die MainActivity

Die MainActivity von *WTest* ist der zentrale Knoten, welcher die einzelnen Komponenten des Frameworks miteinander verbindet. Hierfür erfüllt die Klasse eine Vielzahl von Funktionen und greift auf verschiedene interne Hilfsklassen zurück. Trotz ihrer wichtigen Rolle als zentrales Steuerungselement der App kann die MainActivity des öfteren vom Betriebssystem gestoppt und wieder gestartet werden. Die Anwendung muss daher tolerant gegenüber diesen Änderungen sein.

4.2.1 Importieren des Codes aus Android PCAP

Nach dem Erstellen des Projekts importiere ich zunächst einige Teile des Codes aus der Hauptklasse von Android PCAP. Hierbei übernehme ich nur die für mein Framework

interessanten Teile. Sämtlicher Code zum Darstellen der Dateiliste in Android PCAP fällt zum Beispiel weg, da dieser umständlich und wenig funktional ist. Beim Übernehmen der Codefragmente achte ich auf ein einheitliches Namensschema für Variablen und Methoden. Im Anschluss setze ich die so entstandenen Fragmente zusammen und erweitere sie. Grundlegend orientiert sich die Funktionsweise nach wie vor an Android PCAP, beziehungsweise gültigen Android-Standards.

4.2.2 Kommunikation mit System und Services

Zur Steuerung und Kontrolle der Services verbindet sich die MainActivity mittels der Android-Methode *bindService()* mit dem jeweiligen Service. Anschließend kommuniziert die MainActivity mittels Messages mit den Services, in denen sie die vom jeweiligen Service festgelegten Kennungen zur Identifikation des Nachrichtentyps verwendet. Nach dem Binden registriert sie sich zunächst mit einer Nachricht beim Service, um von diesem in Zukunft als Empfänger von Statusupdates und Fehlermeldungen adressiert zu werden. Für die eingehenden Messages verwendet die MainActivity pro Service einen spezifischen *Messenger* und Handler, der mit den empfangenen Nachrichtentypen umgehen kann und entsprechende Aktionen auslöst. Bei jedem Starten und Stoppen der MainActivity muss sie sich bei den Services an- und abmelden. So ist sie mit den Services in Kontakt, wenn die *Activity* läuft und es werden keine ungültigen Messages an sie gesendet, wenn die Activity gestoppt ist.

Zusätzlich registriert die MainActivity beim Start einen *BroadcastReceiver*, der auf systemweite Ereignisse reagiert, die er per Intent erhält. Handelt es sich bei dem Ereignis um das An- oder Abstecken eines USB-Gerätes oder eine USB-Permission, so leitet es den Intent an den *PcapService* weiter, damit dieser das Gerät prüfen und gegebenenfalls initialisieren kann.

4.2.3 Interaktion mit dem Benutzer

Die MainActivity ist ebenfalls für die Interaktion mit dem Benutzer zuständig. Sie zeigt die Hauptbedienoberfläche und kümmert sich um deren Aktualisierung, wenn sich der Status ändert. Um Benutzereingaben zu registrieren, erstellt und verwaltet sie zahlreiche *OnClickListener*. Je nach ausgewähltem Bedienelement startet sie eine Activity mit den Einstelloptionen, zeigt ein Benachrichtigungs- oder Auswahlfenster an oder führt bestimmte Aktionen wie das Starten eines Services oder des Loggings aus.

Da unter Android graphische Oberflächen nur von bestimmten Klassen wie Activities gestartet werden dürfen, kümmert sich die MainActivity auch um die Fehlerbehandlung der möglicherweise in den Services auftretenden Fehler. Sie wird von den Services per Message informiert und startet dann die Fehlerbehandlung in Abhängigkeit des Fehlers. Dafür kann sie zum Beispiel andere Systemprogramme aufrufen und graphische Bedienelemente anzeigen oder den Benutzer auffordern Einstellungen zu verändern. Sobald das Problem gelöst ist, schickt die MainActivity eine Nachricht an den Service und informiert ihn über die Fehlerbehebung.

Darüber hinaus verwaltet die MainActivity die Einstellungen und Parameter des Frameworks in den *SharedPreferences* und bietet Optionen diese zu ändern. Sie kann, wenn ein Logvorgang läuft, das aktuelle Logfile mittels der Share-Funktion von Android mit anderen Kontakten und Diensten teilen. Auf Wunsch ruft sie einen Browser zum Anzeigen der bisher erstellten Logfiles auf.

4.3 Logback als Logging-Framework

Für das Logging der Abläufe im Framework verwende ich die von *tony19* auf Github veröffentliche [ton16a] Version 1.1.1-4 von *logback-android*. Sie ist eine auf das Logging unter Android spezialisierte Version von Logback. Durch die Anpassungen in diesem Fork ist es einfach möglich einen Logger zu erstellen, welcher zusätzlich zum normalen Logfile auch in das androidinterne *logcat* schreibt. Die Einrichtung ist sehr bequem

und unter [ton16b] gut dokumentiert. Dies ermöglicht es auch zukünftige Erweiterungen ohne großen Aufwand vorzunehmen.

Das Logging-Framework wird zusammen mit SLF4J in die Datei *build.gradle* hinzugefügt und kann dann verwendet werden. In der beim Appstart als erstes gestarteten Klasse WTest konfiguriere ich das Logging-Framework und starte den *rootlogger*. Anschließend können sich alle anderen Klassen in ihrer *onCreate()* Methode eine Instanz des Loggers holen und von da an Einträge auf beliebigen Loglevelen schreiben.

4.4 Monitor-Mode durch PCAP

4.4.1 Android PCAP als Grundlage

Um im WTest Framework den Monitor-Mode einer WLAN-Karte und somit das Mitschneiden von sämtlichem Verkehr auf der gewählten Funkfrequenz zu ermöglichen, verwende ich einen Fork des im Abschnitt 3.3 beschriebenen Android PCAP. Die von *Gabriel Nyman* erstellte Version ist auf Github [NKAS15] verfügbar und erweitert die Version von Kismet Wireless um eine Abschätzung der Signalstärke der empfangenen Pakete.

Lauffähig machen unter Android Studio und aktuellem Android

Bevor ich Teile von Android PCAP in meinem Framework verwende, prüfe ich diese zunächst auf ihre grundlegende Funktionalität. Hierzu importiere ich zunächst den Sourcecode in *Android Studio* und mache ihn dort kompilierbar. Dafür sind unter anderem einige Umstellungen in den *gradle*-Dateien notwendig, da diese beim Import des ursprünglich in *Eclipse* erstellten Projektes nicht korrekt generiert werden. Auch ist JNI noch nicht vollständig von Android Studio unterstützt, sodass man entweder ein experimentelles Plug-in nutzen oder *useDeprecatedNdk* aktivieren muss. Ich entscheide mich

bis zur Ausreifung des Plug-ins für letztere Möglichkeit.

Mit der dann kompilierbaren Version von Android PCAP teste ich, ob und mit welcher Hardware und welchen Betriebssystem-Versionen sich der Netzwerkverkehr mitschneiden lässt. Sowohl auf einem Nexus 5 mit Android 6, als auch auf Nexus 9 mit Android 5 funktioniert die App im Zusammenspiel mit der Alfa AWUS036H WLAN-Karte.

Anschließend ermittle ich, bis zu welcher Software Development Kit (SDK)-Version die App kompilierbar ist, sodass sie dann noch funktioniert. Im Ursprung wird die App mit SDK-Version 15 kompiliert, mit Abwärtskompatibilität bis 13. Durch meine Tests zeigt sich, dass die maximale Version zum Kompilieren bei 22 liegt, also Android 5.1. Ab Android 6 - SDK-Version 23 werden einige wenige Teile der App nicht mehr unterstützt und müssen durch neuen Code ersetzt werden. Für mein Framework wähle ich daher zum Kompilieren SDK-Version 21 - Android 5.0. Als Mindestkompatibilität setze ich SDK-Version 14 - Android 4.0, da es so gut wie keine Geräte mit der urprünglich unterstützten Version 13 gibt. Auf allen Geräten mit höherer Version läuft WTest durch die Aufwärtskompatibilität ebenfalls.

Herauslösen der benötigten Teile

Um die Logging-Funktion aus Android PCAP in WTest zu implementieren, übernehme ich die essenziellen Klassen für das reine Logging aus dem Projekt. Hierbei lasse ich, bis auf die Pfade, die meisten Klassen fast unverändert. Lediglich im PcapService nehme ich große Anpassungen vor. Zunächst verändere ich dort einige Variablen und Methoden und entferne überschüssigen Code. Anschließend mache ich in der Klasse im Laufe der Arbeit noch einige grundlegende Änderungen, welche ich in Abschnitt 4.4.3 und Abschnitt 4.4.4 näher beschreibe.

JNI unter Android Studio

Bei dem Versuch, den JNI Teil des Frameworks mit der PCAP Bibliothek von libpcap unter den angepassten Pfaden und Methodennamen für mehrere Architekturen neu zu kompilieren, treten in Android Studio aufgrund der fehlenden Unterstützung Fehler auf. Die je Architektur benötigte *.so-Datei kompiliere ich daher manuell mit dem Android Native Development Kit (NDK). Hierzu setze ich die Plattform-Version in der *Application.mk*-Datei auf 14, da dies die Mindestversion der App sein soll. Native Softwareteile sind aufwärts- aber nicht abwärtskompatibel und müssen daher mit der *MinSDKVersion* kompiliert werden. Festsetzen muss man die Version, da sonst einige Abhängigkeiten, die eigentlich per optionalem *include* heraus fallen dürfen, Probleme verursachen. Leider kann ich die Lauffähigkeit nur auf Android 5 und 6 überprüfen, da es Probleme beim Ansprechen des vorliegenden Android 4 Testgerätes über die *Android Debug Bridge (ADB)* gibt und die App daher nicht installiert werden kann.

4.4.2 Funktionsweise des PcapService

Das eigentliche Logging funktioniert in WTest noch äquivalent zum ursprünglichen Android PCAP. Hierfür führt der PcapService eine Liste mit bereits geprüften und initialisierten USB-WLAN-Karten. Dort werden sie durch die von der MainActivity weitergeleiteten Broadcasts hinzugefügt, wenn sie in der Prüfung als ein gültiges Gerät erkannt werden. Gültige Geräte sind von der Klasse *Rtl8187Card* und enthalten einen *PacketHandler* zum Behandeln der erstellten *Packet*-Instanzen. Der PacketHandler ist im *PcapLogger* implementiert, und sendet die Packet-Instanzen durch JNI-Methodenaufrufe an libpcap.

Bekommt der PcapService nun eine Message mit dem Logging zu starten, führt er durch seine Instanz des PcapLogger die native Methode *openPcap()* aus. Diese startet das Logging mittels der libpcap-Bibliothek. Hierfür greift der native Code mittels JNI auch auf die Packet-Klasse des Frameworks und die Java Klassen *ByteBuffer* und *IOException* zu. Zum Loggen wird der in der Klasse Rtl8187Card nachgebaute Treiber verwendet, wel-

cher Packets erstellt, die von libpcap in die *.cap-Datei geschrieben werden. Während des Loggings schickt der PcapService mit Hilfe des *PcapHelper* regelmäßige Statusupdates über die Menge der erfassten Pakete an die registrierte MainActivity. Zum Stoppen des Loggings wird, wieder mittels PcapLogger, die native Methode *closePcap()* ausgeführt.

Erhält der Service über seinen Handler eine Message seine Einstellungen zu ändern, liest er die SharedPreferences aus und passt sich an. So kann er das Channelhopping durch ein *Runnable* starten, stoppen oder die Frequenz verändern und die Auswahl der eingestellten Channel anpassen. Alternativ kann er die Karte fest auf einen Channel konfigurieren.

4.4.3 Beheben zahlreicher Bugs

Im ersten lauffähigen Zustand des WTest Frameworks und auch in den kurzen vorherigen Tests mit Android PCAP fallen mehrere Bugs auf. Manche führen zu spontanen Abstürzen der App, andere äußern sich in nicht funktionierenden Optionen. Im Folgenden beschreibe ich einige in WTest behobene Fehler, die in Android PCAP enthalten sind.

So stellt die App beim Initialisieren der WLAN-Karte nicht den richtigen Channel ein und schickt anschließend einen falschen Status an die MainActivity. Hier füge ich vor jedem Loggingvorgang eine Überprüfung des Channels ein und passe die Funktionen beim Initialisieren an. In der ursprünglichen Fassung der App wird auch nicht immer, wenn sich die Einstellungen für Channelhopping ändern, eine Anpassung des PcapService vorgenommen. Dies liegt daran, dass die Funktion nur zufällig durch ein fehlendes *break* in einer *switch-case*-Anweisung ausgeführt wird, wenn *USB-Permissions* an den Service gesendet werden. Dies führt zusätzlich zu gelegentlichen unerwünschten Neukonfigurationen. Hier behebe ich den Fehler und sende eine zusätzliche Message, wenn sich die MainActivity nach Änderung der Einstellungen erneut bei dem Service registriert. Die Registrierung findet immer nach der Änderung statt, da die MainActivity während des Anzeigens des Optionsmenüs vom Betriebssystem gestoppt wird und anschließend neu

startet.

Wenn man das Display während des Loggings ausschaltet oder es sich nach einiger Zeit von alleine abschaltet, wird die MainActivity gestoppt. Hierdurch wird bei Android PCAP das Logging abgebrochen oder die App stürzt ab. In WTest meldet sich die MainActivity beim Stoppen vom PcapService ab und beim Neustart wieder an. Dies beugt Abstürzen durch fehlgeleitete Messages vor. Zusätzlich bekommt der PcapService, solange er aktiv loggt, ein *WakeLock*, welches verhindert, dass die Central Processing Unit (CPU) einschläft und das Logging abbricht.

Ein weiteres Problem von Android PCAP ist, dass die Option des Channelhoppings keine Wirkung hat. Zwar läuft dauerhaft ein Runnable, aber dieses wechselt nicht den Channel der WLAN-Karte. Das löse ich durch das Eintragen der aktiven Rtl8187Card in die entsprechende Geräteliste. Dadurch entstehen jedoch Seiteneffekte, die an anderen Stellen im Code behoben werden, um die Funktionalität des Channelhoppings herzustellen.

4.4.4 Erweitern um neue Funktionen

Zusätzlich zu dem Beheben von Fehlern implementiere ich einige Verbesserungen der Funktionen im PcapService. So wird das Runnable für das Channelhopping nun nur noch ausgeführt, wenn Channelhopping auch aktiv ist und geloggt wird. Das Zeitintervall des Hoppings lässt sich nun ebenfalls flexibel über die GUI einstellen und ist nicht mehr fest auf 250 ms gesetzt. Natürlich schreiben der PcapService und die von ihm genutzten Klassen nun je nach gewähltem Loglevel auch zahlreiche Meldungen in das Anwendungslog, um den Programmfluss zu verdeutlichen und kontrollierbar zu machen.

4.5 Location-Logging in gpx

Das Location-Logging als optionale Funktion läuft in einem eigenen Service, der nur bei Bedarf gestartet wird. Die Klasse nennt sich *GpsService* und funktioniert in einigen

Punkten ähnlich wie der PcapService. So registriert sich die MainActivity nach dem Start beim Service und kommuniziert mit ihm mittels Handler und Messages.

Location Updates mittels Google Play Services

Für das Feststellen der aktuellen Geräteposition verwendet der GpsService die vom *Google Play Services* bereitgestellte Location-API in Version 8.4.0. Für deren Nutzung implementiert der Service die Interfaces *ConnectionCallbacks* und *OnConnectionFailedListener* des *GoogleApiClient*. Diese ermöglichen es, auf Verbindungsereignisse zu reagieren und dabei ereignete Fehler abzufangen. Zur Behandlung der Fehler werden diese Ereignisse per Message an die MainActiviy weitergeleitet, damit sie in Zusammenarbeit mit dem Benutzer gelöst werden können. Zusätzlich implementiert der GpsService die Methode *onLocationChanged()* des Interfaces *LocationListener*, mit der es die *Location*-Updates empfängt.

Beim Start des GpsServices erstellt dieser einen GoogleApiClient für die Location-API und verbindet sie, beziehungsweise behandelt eventuell auftretende Fehler, bis die Verbindung zustande kommt. Erhält der Service nun die Message von der MainActivity mit dem Logging zu starten, so erstellt er einen *LocationRequest* mit dem gewünschten Update-Intervall und möglichst hoher Genauigkeit. Treten hierbei Fehler auf, werden sie an die MainActivity zur Lösung weitergereicht. Ist der Request hingegen erfolgreich, beziehungsweise ist das Problem gelöst, wird mit dem Logging gestartet. Hierbei erreichen den Service nun regelmäßig neue Locations in der *onLocationChanged()* Methode. Diese prüft, wie lange das letzte Update zurück liegt und wie präzise die empfangene Location ist. Zu häufige Updates oder ungenaue Locations werden verworfen, alle anderen in das Logfile geschrieben.

Logging im GPX Standard

Das Logfile mit den GPS-Koordinaten schreibt der GpsService in dem von *TopoGrafix* erstellten *GPS Exchange Format (GPX)1.1* [GPX16] Standard. Vorteile des Standards

sind die gute Unterstützung in zahlreichen Geoinformationssystemen und einfache Konvertierung in andere Formate durch frei verfügbare Programme. Zusätzlich lässt sich das auf Extensible Markup Language (XML) basierende Format leicht erstellen und lesen. Vor dem Start des Loggings wird hierfür der passende Header geschrieben. Zum Abschluss des Loggings wird die Datei mittels schließender Tags beendet.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<gpx xmlns="http://www.topografix.com/GPX/1/1" version="1.1" creator="WTest.GpsService"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.topografix.com/GPX/1/1 http://www.topografix.com/GPX/1/1/gpx.xsd">
 <metadata>
  <name>/mnt/sdcard/pcap/filename.gpx</name>
  <desc> GPS Logfile created to track the location of the device logging.
       Loggingfrequency set to: 10 seconds.
       LoggingAccuracy will be written to pdop value as meters. </desc>
 </metadata>
 <trk>
  <name> GPS Track </name>
  <src> Logged by WTest.GpsService </src>
  <trkseg>
      ... Logged GPS Points here ...
  </trkseg>
 </trk>
</gpx>
```

Listing 4.1: Header und Ende der *.gpx-Datei

Zum Eintragen einer neuen GPS-Koordinate schreibt man jeweils einen simplen Eintrag nach dem folgendem Schema. Hierbei muss die Zeit im von GPX verwendeten ISO 8681 Format eingetragen werden. In das Feld *pdop* schreibt der GpsService die Genauigkeit der empfangenen Location in Metern.

```
<trkpt lat="50.9325818" lon="7.040128">
 <time>2016-05-23T17:43:05Z</time>
 <pdop>11.547</pdop>
</trkpt>
```

Listing 4.2: Eintrag einer GPS-Koordinate im *.gpx-Format

4.6 Limitationen und bekannte Probleme

Aus der verwendeten Hardware und dem aktuellen Stand der Software ergeben sich auch Einschränkungen der Lösung. So beschränkt der eingesetzte RTL8187-Chip die WLAN-Möglichkeiten auf die Standards *IEEE 802.11 a/b/g* und damit die Geschwindigkeit nach oben. Eine weitere Einschränkung ist die Ungenauigkeit der gespeicherten Signalstärke

empfangener Pakete. Wie [Shi14] und [NKAS15] beschreiben, handelt es sich hierbei lediglich um eine Abschätzung, die dem Vergleich der Pakete untereinander dient, aber keine genauen Messwerte wiedergibt. Dies lässt sich, mit ausreichender Einarbeitung in den Treiber, durch eine Erweiterung des betreffenden Codes beheben.

In seltenen Fällen kommt es zu Problemen beim Initialisieren der Netzwerkkarte, wenn die Kalibrierung fehlschlägt. In diesem Fall kann anschließend nicht geloggt werden und wenn man das Logging beginnt, erhöht sich die Anzahl der empfangenen Pakete nicht. Aktueller Workaround hierfür ist es, entweder die Karte neu einzustecken oder die App zu beenden und wieder zu starten, wobei die Karte neu initialisiert wird.

4.7 Zusammenfassung

Das von mir entwickelte Framework WTest besteht im Wesentlichen aus drei funktionalen Bereichen. Um das Zusammenspiel und die Steuerung der Komponenten, sowie die Interaktion mit dem Benutzer kümmert sich die MainActivity. Für das Netzwerk-Logging ist der auf Android PCAP aufbauende PcapService mit seinen Helferklassen zuständig. Das GPS-Logging im offenen GPX-Standard erledigt der GpsService mit Hilfe der Google Play Services. Für alle Klassen verfügbar ist das App-Logging durch Logback. Die einzelnen Bereiche kommunizieren untereinander mit Messages und können einzeln genutzt und erweitert werden. Durch die Auslagerung der wesentlichen Aufgaben des Frameworks in Services, blockieren keine aufwändigen Berechnungen die MainActivity und die App läuft auch im *Strict-Mode* von Android ohne Probleme.

Kapitel 5

Testszenarien und Ergebnisse

5.1 Allgemeine Testvoraussetzungen

Alle erstellten Tests werden mit der gleichen Hardware durchgeführt. Zum Einsatz kommen hier je nach Versuchsaufbau die Router Linksys WRT54GL, Linksys WRT610N, sowie FritzBox 7390. Als Messpunkte verwende ich einen per WLAN angeschlossenen Laptop, einen per Kabel angeschlossenen Linux Computer und die FritzBox. Die App WTest läuft hierbei auf einem Nexus 9 mit Android 5.0.2 und der Alfa AWUS036H. Als Testmethoden nutze ich hauptsächlich einfache Pings mit verschiedenen Einstellungen, sowie das auf Bandbreitentests spezialisierte Programm iPerf. Abbildung 5.1 zeigt den Aufbau des verwendeten Netzwerkes mit Messpunkten und Verbindungen während der Tests.

Für Vergleichsmessungen zur Leistungsfähigkeit des Netzwerks verwende ich noch weitere Laptops und Smartphones. Sie alle zeigen jedoch ähnliche Messwerte. Auch bei Änderungen im Aufbau des Netzwerks treten keine deutlichen Änderungen auf, sodass die Kabelverbindungen als Schwachstelle der späteren Tests ausgeschlossen werden können. Allerdings können gelegentliche Schwankungen im Durchsatz bei drahtlosen Übertragungen, durch in der Nähe befindliche WLANs und andere Störquellen, nie ganz verhindert werden. Siehe hierzu auch Abschnitt 5.3.1. Die Messdaten und Testprotokolle

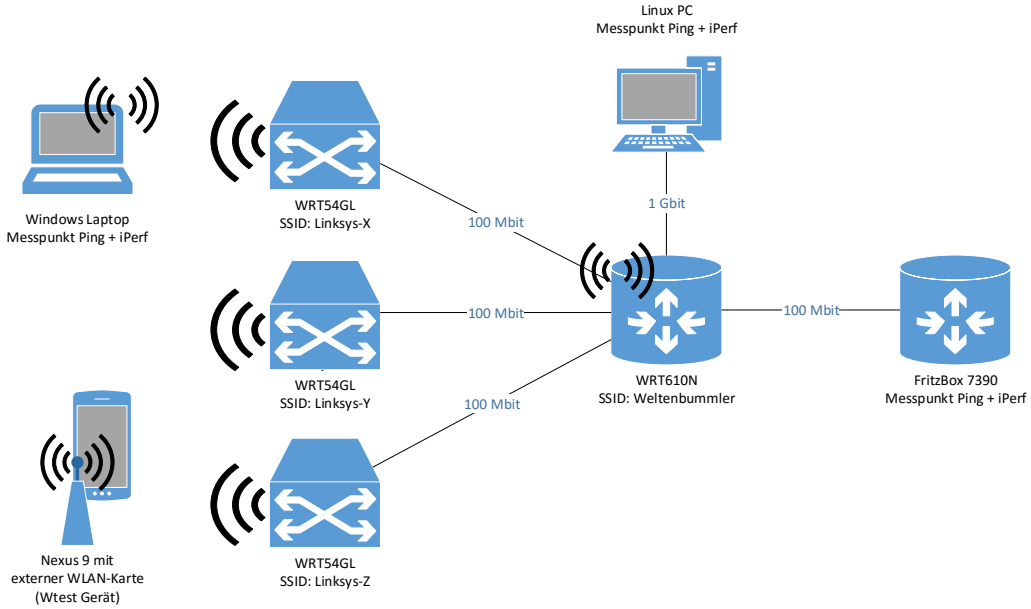


Abbildung 5.1: Netzwerkplan des verwendeten Testaufbaus

befinden sich auf der beigefügten DVD und im Git [Men16] der Arbeit.

5.2 Channel-Abstrahlung

Bei den Tests in diesem Kapitel geht es darum zu testen, ob das Channelhopping funktioniert und wie stark das Signal von benachbarten Channels ist. Dabei wird auch festgestellt, ob Verkehr auf Channel 13 noch lesbar ist, da sich WTest maximal auf Channel 11 einstellen lässt.

5.2.1 Test auf Channelhopping

Für den Test, wie gut Channelhopping funktioniert, konfiguriere ich einen der WRT54GL Router auf Channel 1 und einen zweiten auf Channel 11. Die SSIDs der Netze benenne ich dementsprechend Linksys-1 und Linksys-11 und aktiviere das Broadcasten ihrer SSIDs. WTest lasse ich alle 30 Sekunden zwischen Channel 1 und 11 hin- und herspringen und schneide den Verkehr über vier Runden mit. Bei anschließenden Auswertungen betrachte ich die Anzahl der empfangenen Beacon-Frames, mit denen die SSIDs angekündigt werden.

		Anzahl Beacon Frames	
Zeitintervall in s	*Aktiver Channel*	*Linksys-1*	*Linksys-11*
0-29	Ch. 1	275	0
30-59	Ch.11	0	306
60-89	Ch. 1	278	4
90-119	Ch.11	9	303
120-149	Ch. 1	303	3
150-179	Ch.11	1	291
180-209	Ch. 1	294	2
210-239	Ch.11	1	293

Tabelle 5.1: Auswertung der empfangenen Beacon Frames von Linksys-1 und -11

Tabelle 5.1 zeigt die Anzahl der von den beiden WLANs empfangenen Beacon-Frames in den Hop-Intervallen. Hierbei ist klar ersichtlich, dass jeweils überwiegend die Beacon-Frames von dem WLAN empfangen werden, auf dessen Channel WTest zu dieser Zeit eingestellt ist. Die Funktionalität des Channelhoppings ist somit gegeben.

Auffällig ist hierbei jedoch die Anzahl der vom jeweils anderen Channel empfangenen Beacon-Frames. Allerdings ist eine geringe Streuung von Frames im WLAN normal und somit kein Grund zur Sorge, wie auch [NZR08] detailliert beschreibt.

5.2.2 Mitlesen benachbarter Channel

In diesem Szenario geht es darum festzustellen, ob es gut möglich ist benachbarte Channel mitzulesen und damit insbesondere auch Channel 12 und 13 zu empfangen, auf die sich WTest nicht einstellen lässt. Nach [LLTY13] strahlen die Channel jeweils ab, da sich die verwendeten Frequenzen bei dem verwendeten WLAN nach IEEE 802.11b/g überlappen. Für den Test konfiguriere ich die WRT54GL-Router auf Channel 1, 6 und 13 und benenne die SSIDs dementsprechend. Die App WTest stelle ich auf Channel 11 ein und logge zweimal über drei Minuten mit.

Bei der anschließenden Auswertung zeigt sich, dass von Channel 13 jeweils knapp 1900 Beacon-Frames empfangen werden. Anhand der Sequenz-Nummer des ersten und letzten empfangenen Frames ergibt sich, dass Linksys-13 in dieser Zeit weniger als 2200 Beacon-Frames sendet. Somit werden von Channel 13 ungefähr 87 % der Beacon-Frames empfangen. Von den an der Funkfrequenz gemessen weiter entfernten Channels 1 und 6 werden in der gleichen Zeit hingegen jeweils weniger als 50 Beacon-Frames aufgezeichnet. Diese Zahl der gestreuten Pakete ist wie bei dem vorherigen Channelhopping-Versuch gering und im normalen Rahmen.

Bei einem Vergleichstest, in dem der dritte Router anstatt auf Channel 13 auf Channel 11, und somit auf den gleichen Channel wie WTest konfiguriert wird, zeigen sich ähnliche Ergebnisse. Auch hier werden ungefähr 87 % der gesendeten Beacon-Frames aufgezeichnet und auch die Zahl der von den anderen Channels empfangenen Frames steigt nicht signifikant an. Somit ist es durch die Channelüberlappung kein Problem, die Pakete benachbarter Channels mitzuschneiden und auch Channel 12 und 13 auszulesen.

5.3 Durchsatz-Messung

Bei den Durchsatz-Messungen geht es vor allem darum, festzustellen wie viel vom Netzwerkverkehr die App WTest mitlesen kann, wenn die Übertragungsraten im WLAN sehr hoch sind. Zum Testen nutze ich einfache Ping-Fluten sowie iPerf. Bei allen Tests ist der

räumliche Aufbau mit Position und Entfernung der Geräte gleich. Das Android-Gerät mit WTest befindet sich auf einer Achse zwischen Sender und Empfänger mit ungefähr einem halben Meter Abstand vom Laptop und knapp einem Meter zum Access-Point.

5.3.1 Vergleichsmessungen

Um die Tests mit der App in das richtige Verhältnis setzen zu können, nehme ich zunächst einige Messungen in der Testumgebung vor. Zunächst messe ich, wie gut die iPerf Gegenstelle per Kabel von dem Router mit dem später verwendeten WLAN Access-Point erreichbar ist. Dabei erreicht iPerf Geschwindigkeiten von knapp 70 Mbit/s. Bei Vergleichsmessungen mit verschiedenen Geräten per WLAN über den Access-Point sind es hingegen maximal 21 Mbit/s über längere Zeit. Hierbei macht es in der von mir verwendeten Testumgebung durchaus einen Unterschied, welchen Channel man einstellt. Die Ergebnisse sind wie erwartet auf Channel 1, 6 und 11 mit 21 Mbit/s am besten, da dort am wenigsten Störungen durch andere Frequenzen auftreten. Auf den Channels dazwischen sinkt der Durchsatz aufgrund von Überlappung der Funkfrequenzen auf bis zu 14 Mbit/s.

Schließt man die für meine WTest-Messungen verwendete Alfa AWUS036H WLAN-Karte an einen Laptop an, so erreicht sie im Schnitt nur 10 bis 20 Mbit/s. Dies legt den Verdacht nahe, dass beim späteren Betrieb per WTest, bei voller Auslastung des WLANs mit über 20 Mbit/s, im ungünstigsten Fall nur noch circa die Hälfte der Pakete empfangen wird.

5.3.2 Höchstlast-Tests mit iPerf

Bei diesem Test wird ermittelt, wie WTest auf hohen Datendurchsatz auf der mitgeschnittenen Frequenz reagiert und wie viele Pakete gelesen werden. Hierzu wird ein Router auf Channel 11 konfiguriert und der Laptop auf diesem Access-Point angemeldet. Zunächst wird über drei Runden mit je fünf Minuten per iPerf mit maximaler Bandbreite vom

Desktop-Computer an den Laptop gesendet und mit WTest mitgeloggt. Dabei werden im Schnitt 15 Mbit/s übertragen. Anschließend wird in die andere Richtung vom Laptop an die FritzBox gesendet, wobei 20 Mbit/s erreicht werden. Zum Vergleich von WTest wird immer auch auf dem Laptop mittels Wireshark der Verkehr aufgezeichnet.

Bei der anschließenden Auswertung der mitgeschnittenen Daten werden die Aufzeichnungen mit Hilfe verschiedener Filter gezählt und verglichen. Zunächst betrachte ich die Gesamtzahl der empfangenen Frames. Dann schaue ich auf die eindeutig per iPerf empfangenen Daten und zuletzt auf die unvollständigen Frames mit dem richtigen Ziel im 802.11-Header. Hierbei zeigen sich die in Tabelle 5.2 aufgelisteten Framezahlen.

	Testdurchlauf 1		*Testdurchlauf 2*		*Testdurchlauf 3*	
Displayfilter	*Laptop*	*WTest*	*Laptop*	*WTest*	*Laptop*	*WTest*
Sender: Desktop						
Gesamt	571 751	509 273	575 306	512 840	574 556	631 923
nach Port	571 120	132 838	574 719	115 058	574 014	224 446
nach Header	571 120	503 629	574 719	507 499	574 014	626 386
Sender: Laptop						
Gesamt	767 416	357 127	768 154	282 313	762 947	300 177
nach Port	766 738	108 825	767 774	70 839	762 437	72 953
nach Header	766 738	352 294	767 774	276 473	762 437	293 684

Tabelle 5.2: Höchstlast-Tests mit iPerf: Auswertung der Anzahl empfangener Frames

Als Referenz schneidet Wireshark auf dem Laptop alle intakten Frames mit, da per Transmission Control Protocol (TCP) übertragen wird und der Laptop als einer der Kommunikationspartner alle Pakete empfangen und bestätigen muss. Somit ändert sich hier beim Filtern auf Port oder Header auch nicht die Anzahl der gezählten Frames.

Bei den Versuchen mit dem Computer als Sender schneidet WTest knapp 90 % der vom Laptop gemessen Frames mit. In einem Fall sind es sogar über 100 %, was sich vermutlich mit doppelten Frames durch Fehlerkorrektur erklärt. Bei der Zahl der vollständigen

Pakete mit der verwendeten Portnummer sind es hingegen nur zwischen 15 und 30 % der Referenzmenge. Bei der Betrachtung der Frames mit dem passenden Ziel im 802.11-Header zeigt sich, dass die Anzahl wiederum bei fast 90 % liegt. Somit hat WTest anscheinend viele Pakete nicht vollständig mitgeschnitten, so dass nur der 802.11-Header lesbar bleibt.

Bei der Messung der Gegenrichtung, also der Übertragung von Laptop zu FritzBox, sinkt die Gesamtzahl der mitgeschnittenen Frames unter 50 %. Auch die intakten Pakete werden weniger und ihr Anteil vom Sollwert liegt nunmehr bei 9 bis 14 %. Die Anzahl der mitgeschnittenen Header liegt wie im vorherigen Versuch nur knapp unter der Gesamtmenge der empfangenen Frames. Somit werden bei diesem Versuch weniger Frames empfangen als bei der Übertragung von Computer zu Laptop. Mögliche Erklärung hierfür kann sein, dass durch die um ein Drittel höhere Übertragungsgeschwindigkeit der Grad der Überforderung der WLAN-Karte ansteigt. Zusätzlich scheint die Karte oder WTest anteilig deutlich mehr intakte Daten von dem Laptop als von dem Access-Point zu empfangen. Diese Feststellung zeigt sich deutlich in der Aufstellung der intakten Pakete nach Absende-IP in Tabelle 5.3. Mögliche Ursachen für diese Beobachtung können die etwas kleinere Distanz zum Laptop sein, die jedoch nur sehr gering ist, oder eine bessere Kompatibilität der WLAN-Adapter zueinander.

	intakte Pakete	*vom Access-Point*	*vom Laptop*	*Anteil Laptop*
Test 1.1	132838	36499	96339	73 %
Test 1.2	115058	19995	95063	83 %
Test 1.3	224446	92493	131953	59 %
Test 2.1	108825	16391	92434	85 %
Test 2.2	70839	11621	59218	84 %
Test 2.3	72953	11237	61716	85 %

Tabelle 5.3: Höchstlast-Tests mit iPerf: Aufteilung der intakten Pakete nach Absender

5.3.3 Tests mittels Pings

Auch in diesen Tests geht es darum, festzustellen wie viele Frames WTest mitschneidet. Hierbei kann durch die Größe der gesendeten Ping-Pakete und das Sendeintervall einfach zwischen verschiedenen Lastprofilen des Netzwerkes gewählt werden. Bei einem ersten Test wird durch eine Pingflut mit 1000 Byte großen Paketen alle 0.1 ms eine hohe Last im Netzwerk erzeugt, wobei einige wenige Pakete verloren gehen. Während des fünfminütigen Tests werden insgesamt ungefähr 960000 Internet Control Message Protocol (ICMP)-Pakete versendet. Davon empfängt WTest knapp 290000 Frames, von denen nur gut 120000 Pakete komplett intakt sind. Somit werden von der Gesamtmenge der zu loggenden Frames ungefähr 30 % empfangen und lediglich 15 % sind vollständig. Hierbei werden von den vollständigen Paketen fast 100 % vom Laptop empfangen und nur 224 vom Access-Point.

Die zweite Testreihe besteht aus jeweils zwei Runden mit je 10 Minuten pro Richtung. Es wird genau ein normaler Ping pro Sekunde gesendet und beantwortet. Die Gesamtzahl der ICMP-Pakete liegt somit bei ungefähr 1200 je Test. Da im Funkbereich jedoch auch andere Daten gesendet werden, empfängt WTest in dieser Zeit zwischen 15000 und 19000 Frames. Die gezählten ICMP-Pakete schwanken dabei zwischen 143 und 632, wobei ungefähr 87 % vom Laptop stammen. Schaut man auf die Anzahl der gezählten 802.11-Header mit dem Ziel Laptop oder Access-Point, so schwankt die gemessene Zahl zwischen 1132 und 1791. Somit werden wieder viele Pakete nicht komplett aufgezeichnet, sondern nur deren Header.

5.4 Stromverbrauch

Motiviert durch die Warnung der Android PCAP Entwickler, dass das Loggen mittels externer WLAN-Karte deutlich mehr Strom verbraucht, wird in den folgenden Tests der Akkubedarf von WTest überprüft. Dabei werden vier unterschiedliche Szenarien über jeweils drei Stunden getestet. Dafür wird der Akku des Nexus 9 auf 100 % aufgeladen und nach jedem Test der Akkustand abgelesen und wieder aufgeladen. Für Netzwerkverkehr

sorgen 10 Pings mit je 1000 Byte pro Sekunde, sowie der zufällige Funkverkehr anderer Geräte.

Zunächst wird das normale PCAP-Logging von Netzwerkverkehr, aber ohne GPS, getestet. Hierbei werden einmal sehr viele Frames und einmal wenige Frames geloggt um zu sehen, wie dies den Stromverbrauch beeinflusst. Anschließend wird zusätzlich zum PCAP Logging auch GPS aktiviert und eine geringe Menge Frames geloggt. Beim letzten Test wird die WLAN-Karte per Y-Kabel mit zusätzlichem Strom versorgt und diesmal eine mittelhohe Menge Frames, aber ohne GPS, geloggt. Dabei ergeben sich die nach dem Akkustand sortierten Messwerte in Tabelle 5.4.

	Anzahl Frames	*Akkustand*
PCAP + Zusatzstrom	795 000	87 %
PCAP geringe Last	290 000	71 %
PCAP hohe Last	1 400 000	66 %
PCAP + GPS	355 000	66 %

Tabelle 5.4: Akkustand des Nexus 9 nach dem Verbrauchstest

Es zeigt sich somit, dass der Stromverbrauch des Testgerätes mit der Zahl der geloggten Frames und dem Zuschalten von GPS ansteigt. Eine sehr deutliche Reduktion des Verbrauchs ist durch das Anhängen einer zusätzlichen Stromquelle für die Versorgung der WLAN-Karte zu erreichen, wodurch mehr als die Hälfte des Stroms im Geräteakku eingespart wird. Somit kann die Laufzeit bei der angestrebten mobilen Nutzung in Kombination mit einem Zusatzakku deutlich verlängert werden.

5.5 Zusammenfassung der Messergebnisse

Das Einstellen auf einen festen Channel sowie Channelhopping funktionieren, und durch die Frequenzüberlappung wird auch Verkehr von benachbarten Channels empfangen. Bei

den Lasttests mittels Ping und iPerf werden je nach verwendetem Sender und Netzwerkdurchsatz unterschiedlich viele Frames empfangen. Die Rate schwankt hier zwischen 20 und 100 %, wobei der interessantere Wert der vollständigen Pakete zwischen 10 und 50 % liegt. Somit werden häufig nur die Header der Frames gespeichert, sobald mehr Verkehr im Netzwerk herrscht. Allerdings fällt auch auf, dass die WLAN-Karte mit manchen Sendern deutlich besser funktioniert als mit anderen. Hierbei gab es reproduzierbare Unterschiede von bis zu 99 % bei der Anzahl der komplett empfangenen Pakete.

Der Test des Stromverbrauchs zeigt die erwarteten Messwerte, wobei der Gesamtverbrauch deutlich niedriger liegt, als nach der Warnung von Kismet Wireless zu befürchten wäre. Dennoch lohnt es sich, die WLAN-Karte für längere Logvorgänge mittels Zusatzstrom zu versorgen.

Kapitel 6

Zusammenfassung

Wie in Kapitel 2 beschrieben, gibt es zahlreiche Möglichkeiten ein Android-Gerät mit Hilfe von Root-Rechten in die gewünschte Testplattform zu verwandeln. Möchte man aber auf das Rooten des Gerätes mit gutem Grund verzichten, so gibt es bisher nur eine rudimentäre Umsetzung durch Android PCAP. Als Proof-of-Concept zu verstehen, bildet es grundlegende Funktionen eines WLAN-Treibers in Java nach.

Im Rahmen dieser Arbeit integriere ich ausgewählte Stücke des Codes von Android PCAP in das WTest Framework und kombiniere sie dort mit den Logging-Funktionen des logback-android-Frameworks und den Location-Tracking-Möglichkeiten von Android. Dabei vertiefe ich die Konzepte der Android-Programmierung und lerne die Funktionsweise der Google Play Services und des Java Native Interface kennen. Das entstandene Framework erweitert die Funktionen von Android PCAP und ist deutlich robuster als sein Vorgänger. Durch seinen modularen Aufbau, das interne Logging und den durch Kommentare besser verständlichen Code, eignet sich WTest auch als Ausgangspunkt für zukünftige Erweiterungen.

Nach dem Entwickeln des Frameworks werden verschiedene Tests auf Funktionalität der App und die Qualität der erfassten Netzwerkdaten gemacht. Hierbei werden Testreihen mit unterschiedlichen Sendern und Empfängern, sowie verschiedenen Lastprofilen im Netzwerk durchgeführt und ausgewertet.

6.1 Schlussfolgerung

Wie in den Tests gezeigt, funktioniert das grundlegende Konzept von WTest zum Mitschneiden von Verkehr in WLANs mittels Monitor-Mode. Somit ist es ohne Root-Rechte möglich, mit beliebigen Android-Geräten ab Version 4 WLANs abzuhören. Allerdings gehen hierbei, insbesondere bei höheren Belastungen und manchen Sendern, viele Pakete verloren, beziehungsweise es wird nur der Frame-Header des 802.11-Frames aufgezeichnet. Dies macht es häufig schwer, den Kommunikationsfluss richtig zu interpretieren, und die Nutzbarkeit der aufgezeichneten Daten sinkt deutlich. An dieser Stelle sind in der Zukunft weitere Tests mit anderen WLAN-Karten und Testgeräten notwendig, um festzustellen, ob es sich um ein grundsätzliches Problem handelt. Die bisherigen Verlustraten von 10 bis 80 % bei kompletten Paketen lassen das Framework nur für bestimmte Einsatzzwecke nutzbar erscheinen und erfordern stets eine sorgfältige Auswertung der Aufzeichnungen.

Im Gegensatz zu Android PCAP, aus dem die Möglichkeit zum Mitschneiden mittels libpcap per JNI und nachgebautem Treiber im Java-Userspace entnommen ist, läuft WTest aber deutlich stabiler und stromsparender. Auch sind in WTest zahlreiche Bugs von Android PCAP behoben und die Funktionen zur Konfiguration und das Channelhopping sind nun lauffähig. Darüber hinaus bietet WTest ein ausführliches appinternes Logging und die Möglichkeit GPS-Koordinaten mitzuspeichern. Dies macht zum Beispiel seinen Einsatz beim Wardriving möglich, da der mitgeschnittene Netzwerkverkehr mit Positionsdaten verknüpft werden kann. Als kleines Extra kann nun auch das Logverzeichnis gewechselt werden, damit der Benutzer zwischen Speicherkarte und internem Speicher wählen, aber auch je nach Test einen eigenen Ordner erstellen kann. Auch andere kleinere Verbesserungen des benutzten Android PCAP-Codes haben neue Funktionen wie die Einstellbarkeit des Channelhop-Intervalls gebracht. Die Sorge über den zu hohen Stromverbrauch beim Loggen konnte durch Tests zerstreut werden, wobei es dennoch sinnvoll ist, die WLAN-Karte für lange Messungen über ein Y-Kabel mit Zusatzstrom zu versorgen.

Um das Framework für zukünftige Erweiterungen zu rüsten, ist es modular aufgebaut und einfach per zusätzlicher Services und Zeilen in der GUI zu erweitern. Das interne

Logging kann hierbei einfach von anderen Klassen genutzt werden. Einige mögliche Erweiterungen zur Steigerung der Nutzbarkeit stelle ich im folgenden Kapitel vor.

6.2 Weiterführende Arbeiten

6.2.1 Testen der Klasse RawSocket

Um WTest mit der Möglichkeit von Packet-Injection zu erweitern, kann sich die Klasse *RawSocket.java* aus [Raw12] anbieten. Diese baut gewisse Funktionalitäten von Raw-Sockets für Android in Java mit Hilfe von JNI nach. Ob diese allerdings ohne Root-Rechte funktionieren und die App dann noch Google Play Store kompatibel ist, bedarf genauerer Untersuchung.

6.2.2 Erweiterungen des Treiber im Userspace

Eine andere Möglichkeit für Packet-Injection ist die Erweiterung des nachgebauten Treibers der RTL8187-Karte um die Funktion, Bitfolgen zu senden. Ob dies durch den verwendeten USB-Manager von Android möglich ist, muss hierbei geprüft werden. Für eine solche Erweiterung bedarf es allerdings einiger Einarbeitung in die hardwarenahe Programmierung und den Linux-Treiber der Karte.

6.2.3 Sonstige Verbesserungen der Software

Einfache Verbesserungen von WTest sind der Einbau einer kleinen Anzeige in der GUI für eventuell auftretende Warnungen und Fehlermeldungen. Aktuell werden viele Meldungen nur in das Log geschrieben und man kann sie dort nachlesen.

Des weiteren können die Stellen im Code ersetzt werden, die aktuell als deprecated markiert sind. Möchte man die App auf Android 6 kompilieren, sind diese Methoden nicht mehr vorhanden. Da es sich nur um wenige Zeilen Code handelt und für einige der Umstellungen gute Anleitungen vorhanden sind, ist dies in relativ kurzer Zeit realisierbar.

Ein aktueller Bug befindet sich noch in der Implementation des Treibers. In seltenen Fällen schlägt die Initialisierung der WLAN-Karte fehl und schreibt eine Meldung in das Logfile. Diesen Fehler kann man abfangen und versuchen die Karte neu zu laden, um den Fehler so zu umgehen. Zur Zeit muss entweder WTest neu gestartet oder die Karte neu verbunden werden.

6.2.4 libpcap-1.3.0 durch höhere Version austauschen

In der gegenwärtigen Version von WTest wird die libpcap-1.3.0 genutzt. Auf den Entwicklerseiten von libpcap ist das aktuelle Release hingegen Version 1.7.4. Man kann versuchen, diese aktuellere Version in die App zu integrieren und somit vielleicht die Performance zu verbessern. Dafür muss man zunächst die eingebaute libpcap mit der originalen 1.3.0 vergleichen und schauen, welche Teile verändert wurden. Vermutlich sind die wesentlichen Änderungen in den Adapterklassen des JNI. Wenn dem so ist, kann man prüfen, ob ein neuere Version von libpcap die gleichen Methodensignaturen hat und sie gegebenenfalls austauschen.

6.2.5 Weitere Performance-Tests

Einige weiterführende Performance-Tests mit WTest können die Aussage der bisherigen Tests in den richtigen Rahmen setzen. So ist zu prüfen, wie die Ursprungsversion von Android PCAP sich in den Szenarien verhält und wie sich die Leistung von WTest gegenüber dem Vorgänger verändert hat. Auch bieten sich Tests mit anderen Android-Geräten und WLAN-Karten an, um festzustellen, ob sich die Leistung hierdurch verbessern lässt. Insbesondere hochwertigere WLAN-Karten könnten einen Unterschied ma-

chen, denn die Alfa AWUS036H erreicht auch an einen PC angeschlossen nur mäßige Übertragungsraten. Diese Übertragungsraten deuten auf eine schwache Sende- und Empfangsleistung der Karte oder Limitationen durch den verwendeten Chip hin. Dies kann man in einer Testreihe mittels libpcap auf einem PC verifizieren und gleichzeitig prüfen, ob dort ebenfalls viele Pakete unvollständig aufgezeichnet werden.

6.2.6 Testen auf welchen Modellen WTest läuft

Ein weiterer sinnvoller Test wäre das systematische Überprüfen, auf welchen Geräten und mit welcher Android-Version WTest läuft. So konnte bisher nicht auf der Mindestversion 4.0 getestet werden und lediglich Nexus 5 und 9 wurden verwendet.

Literaturverzeichnis

[Air16] *Aircrack-ng.org mainpage*. http://www.aircrack-ng.org/, 2016. Online, zuletzt geprüft: 31.05.2016

[And13] *Android-PCAP von kismetwireless*. https://www.kismetwireless.net/android-pcap/, 2013. Online, zuletzt geprüft: 01.06.2016

[Ans15] *Use rooted android to capture all traffic on wan using monitor/promiscuous mode*. http://android.stackexchange.com/questions/120759/use-rooted-android-to-capture-all-traffic-on-wan-using-monitor-promiscuous-mode, 2015. Online, zuletzt geprüft: 03.06.2016

[bcm12] *bcmon google project information*. https://code.google.com/archive/p/bcmon/, 2012. Online, zuletzt geprüft: 03.06.2016

[bcm13] *Wardriving from your pocket - Using Wireshark to Reverse Engineer Broadcom WiFi chipsets*. https://www.dropbox.com/sh/le8zeczpddf3nx0/fdXn4LSxGI, 2013. Online, zuletzt geprüft: 03.06.2016

[bla11] BLACKPLATYPUS: *Packet injection and monitor mode support*. http://forum.xda-developers.com/showthread.php?p=18713483#post18713483, 2011. Online, zuletzt geprüft: 02.06.2016

[ciu15] CIUFFY: *Android CyanogenMod Kernel Building: Monitor Mode on Any Android Device with a Wireless Adapter*. http://null-byte.wonderhowto.com/how-to/android-cyanogenmod-kernel-building-monitor-mode-any-android-device-with-wireless-adapter-0162943/, 2015. Online, zuletzt geprüft: 03.06.2016

[Cum15] CUMPLIDO, Tao: Android rooten? In: *Heise* (2015). . http://www.heise.de/download/special-android-rooten-vorteile-nachteile-und-cyanogenmod-152621.html, Online, zuletzt geprüft: 05.06.2016

[Cya16] *About - CyanogenMod.org* . http://www.cyanogenmod.org/about, 2016. Online, zuletzt geprüft: 02.06.2016

[dev13] DEVIATO: *AircrackGUI 1.2 for bcm4329 and bcm4330 chipsets*. http://forum.xda-developers.com/showthread.php?t=2233282, 2013. Online, zuletzt geprüft: 03.06.2016

[Dra12] DRANSFELD, Damon: *Android WiFi Hacking Penetration Suite – dSploit*. http://www.tacticalcode.de/2012/10/android-wifi-hacking-penetration-suite-dsploit.html, 2012. Online, zuletzt geprüft: 03.06.2016

[GPX16] *GPX 1.1 Schema Documentation*. http://www.topografix.com/GPX/1/1, 2016. Online, zuletzt geprüft: 15.05.2016

[IOF13] ILDIS, Omri; OFIR, Yuval; FEINSTEIN, Ruby: *Monitor mode for Broadcom WiFi Chipsets*. http://bcmon.blogspot.de/, 2013. Online, zuletzt geprüft: 03.06.2016

[JNI16a] *Java Native Interface Specification - Design*. http://docs.

oracle.com/javase/7/docs/technotes/guides/jni/spec/design.html, 2016. Online, zuletzt geprüft: 04.06.2016

[JNI16b] *Java Native Interface Specification - Introduction*. http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/intro.html, 2016. Online, zuletzt geprüft: 04.06.2016

[Joo15] JOOS, Thomas: Android-Rooting: Vorzüge, Gefahren und Möglichkeiten. In: *Computerwoche* (2015). . http://www.computerwoche.de/a/android-rooting-vorzuege-gefahren-und-moeglichkeiten,2060387, Online, zuletzt geprüft: 05.06.2016

[Kal16a] *Kali Linux NetHunter*. https://www.kali.org/kali-linux-nethunter/, 2016. Online, zuletzt geprüft: 03.06.2016

[Kal16b] *Kali NetHunter Git - Documentation*. https://github.com/offensive-security/kali-nethunter/wiki, 2016. Online, zuletzt geprüft: 03.06.2016

[Kal16c] *Wireless Cards and NetHunter*. https://github.com/offensive-security/kali-nethunter/wiki/Wireless-Card, 2016. Online, zuletzt geprüft: 03.06.2016

[Kis16] *Kismet Wireless Documentation*. https://www.kismetwireless.net/documentation.shtml, 2016. Online, zuletzt geprüft: 31.05.2016

[KS04] KROMA, Pierre; SCHREIBER, Sebastian: *Heise - Deauthentication Flooding*. https://www.heise.de/security/artikel/Deauthentication-Flooding-271244.html, 2004. Online, zuletzt geprüft: 04.06.2016

[Lin16] LINUX (Hrsg.): *raw(7) - Linux man page*. http://linux.die.net/man/7/raw:

Linux, 2016. Online, zuletzt geprüft: 05.06.2016

[LLTY13] LI, Mingming; LIU, Haiyang; TAN, Haifeng; YANG, Miao: Performance and Interference Analysis of 802.11g wireless network. In: *International Journal of Wireless & Mobile Networks (IJWMN) Vol. 5*. http://airccse.org/journal/jwmn/5513ijwmn11.pdf, October 2013 (No. 5).. . Online, zuletzt geprüft: 10.06.2016

[Log16] *Logback Documentation, Articles and Presentations*. http://logback.qos.ch/documentation.html, 2016. Online, zuletzt geprüft: 06.06.2016

[mdk14] *mdk3 Package Description*. http://tools.kali.org/wireless-attacks/mdk3, 2014. Online, zuletzt geprüft: 31.05.2016

[Men16] MENDE, Fabian: *Git Repository der schriftlichen Ausarbeitung zur Bachelorarbeit*. https://git.hhu.de/rabia100/ba-mende, 2016. Online, zuletzt geprüft: 19.06.2016

[mic12] MICHAELMOTES: *Aircrack-ng on Galaxy Nexus w/ AWUS036H usb wifi adapter (RTL8187 drivers)*. http://forum.xda-developers.com/galaxy-nexus/general/kernel-aircrack-ng-galaxy-nexus-t1650817, 2012. Online, zuletzt geprüft: 03.06.2016

[Moi13] MOIDU, Nizam: *Aircrack Suite for Galaxy Note 2 with RTL8187 & AR9271*. http://www.maxters.net/2013/02/aircrack-suite-for-galaxy-note-2/, 2013. Online, zuletzt geprüft: 02.06.2016

[mon16] *Network Monitor Operation Mode*. https://msdn.microsoft.com/en-us/library/ff568369.aspx, 2016. Online, zuletzt geprüft: 04.06.2016

[Moo09] MOON, Silver: *Code raw sockets in C on Linux*. http://www.binarytides.com/raw-sockets-c-code-linux/, 2009. Online, zuletzt geprüft: 05.06.2016

[NKAS15] NYMAN, Gabriel; KERSHAW, Mike; ATANASOV, Vencislav; SHI, Jinghao: *GitSource of the Android-PCAP fork*. https://github.com/gnyman/AndroidPCAP, 2015. Online, zuletzt geprüft: 15.03.2016

[NZR08] NACHTIGALL, Jens; ZUBOW, Anatolij; REDLICH, Jens-Peter: The Impact of Adjacent Channel Interference in Multi-Radio Systems using IEEE 802.11 / Humboldt University, Germany. http://edoc.hu-berlin.de/oa/conferences/reSNAVwf8bA/PDF/28TjAjjEtU7ts.pdf, 2008. Forschungsbericht. . Online, zuletzt geprüft: 10.06.2016

[Par08] PARTH: *Difference - Promiscuous vs. Monitor Mode*. http://lazysolutions.blogspot.de/2008/10/difference-promiscuous-vs-monitor-mode.html, 2008. Online, zuletzt geprüft: 04.06.2016

[pca15] *PCAP Description*. http://www.tcpdump.org/manpages/pcap.3pcap.html, 2015. Online, zuletzt geprüft: 04.06.2016

[pur15] PUROHIT, Himanshu: *Reaver for Android (CRACK WPA WITH YOUR ANDROID DEVICE)*. http://cheatglitchesdownloadandroidgames.blogspot.de/2015_01_01_archive.html?m=1, 2015. Online, zuletzt geprüft: 03.06.2016

[Raw12] *RawSocket.java aus libcore*. https://android.googlesource.com/platform/libcore/+/jb-mr2-release/luni/src/main/java/libcore/net/RawSocket.java, 2012. Online, zuletzt geprüft: 12.06.2016

[raw16] *Socket Options in Java*. https://docs.oracle.com/javase/8/docs/technotes/guides/net/socketOpt.html, 2016. Online, zuletzt geprüft: 05.06.2016

[Sch07] SCHOEN, Seth: Detecting packet injection: a guide to observing packet spoofing by ISPs / Electronic Frontier Foundation (EFF). https://www.eff.org/de/wp/detecting-packet-injection : eff.org, 2007. Forschungsbericht. . Online, zuletzt geprüft: 04.06.2016

[Shi14] SHI, Jinghao: *Get Packet Signal Strength of RTL8187 Dongle*. http://jhshi.me/2014/09/21/get-packet-signal-strength-of-rtl8187-dongle/, 2014. Online, zuletzt geprüft: 01.06.2016

[SLF16] *Simple Logging Facade for Java (SLF4J)*. http://www.slf4j.org/, 2016. Online, zuletzt geprüft: 06.06.2016

[tcp15] *tcpdump public repository*. http://www.tcpdump.org/, 2015. Online, zuletzt geprüft: 31.05.2016

[ton16a] TONY19: *logback-android auf github von tony19*. https://github.com/tony19/logback-android, 2016. Online, zuletzt geprüft: 04.05.2016

[ton16b] TONY19: *logback-android Wiki von tony19*. https://github.com/tony19/logback-android/wiki, 2016. Online, zuletzt geprüft: 04.05.2016

[Ull12] ULLENBOOM, Christian: *Java 7 - Mehr als eine Insel, Kapitel 21: Java Native Interface*. http://openbook.rheinwerk-verlag.de/java7/1507_21_001.html#dodtp283ed9de-cbc2-463f-9fad-4f7eef376a8d : Rheinwerk Computing, 2012. ISBN 978-3-8362-1507-7

[Wir16] *Wireshark Learning Page*. https://www.wireshark.org/#learnWS, 2016. Online, zuletzt geprüft: 31.05.2016

[zit13] ZITSTIF: *Steps Toward Weaponizing the Android Platform*. http://zitstif.no-ip.org/?tag=android-packet-injection, 2013. Online, zuletzt geprüft: 03.06.2016

Glossar

Android application package Dateiformat zur Installation von Anwendungen auf Android. 20

Activity Activities stellen die GUI dar und es kann immer nur jeweils eine aktiv sein, mit welcher der Benutzer interagiert. Siehe: `https://developer.android.com/reference/android/app/Activity.html`. 23, 24

Application Application-Klasse startet vor der Activity und dient der Einstellung des Applikationszustandes. Siehe: `https://developer.android.com/reference/android/app/Application.html`. 20

Assembler Maschinennahe Programmiersprache. 7

Beacon-Frame Beacon-Frames werden vom Access-Points gesendet, um die Anwesenheit ihres WLANs bekannt zu machen. 35, 36

BroadcastReceiver Empfängt systemweit gesendete Intents, die mittels Broadcast versendet wurden. Siehe: `https://developer.android.com/reference/android/content/BroadcastReceiver.html`. 23

Bundle Datenformat von Android mit Mappings von Keys zu Datenpaketen. Siehe auch `https://developer.android.com/reference/android/os/Bundle.html`. 20, 60

C Imperative Programmiersprache, häufig für Systemprogrammierung verwendet. 7, 16

Exploit Systematische Möglichkeit zur Ausnutzung von Schwachstellen. 8, 13

Galaxy Note2 Spezifisches Smartphone Modell vom Hersteller Samsung. 6

gradle Build-Management Tool, welches unter anderem in Android Studio verwendet wird. 25

Handler Kommunikationsschnittstelle mit Warteschlange von Threads, zum Senden und Empfangen von Messages. Siehe auch: `https://developer.android.com/reference/android/os/Handler.html`. 20, 23, 28, 30, 60

HTC Hero Spezifisches Smartphone Modell vom Hersteller HTC. 5, 6

Intent Abstrakte Beschreibung an das Betriebssystem mit der Absicht bestimmte Aktionen, wie das Starten anderer Apps oder Services auszuführen. Siehe auch `https://developer.android.com/reference/android/content/Intent.html`. 20, 23

Message Definiert eine androidspezifische Nachricht zum Senden an einen Handler. Kann ein Bundle enthalten. Siehe auch: `https://developer.android.com/reference/android/os/Message.html`. 16, 20, 23, 24, 27–30, 32, 60

Messenger Der Messanger ist eine Referenz auf einen Handler zum Senden von Messages an diesen. Siehe: `https://developer.android.com/reference/android/os/Messenger.html`. 23

Native Development Kit Erlaubt es Teile der entwickelten App mittels JNI und nativem Code zu implementieren. 27

Nexus Android Geräteserie von Google, die insbesondere auf Entwickler abzielt und mit einem Vanilla Android läuft. 8

Preferences Verwaltet die Einstellungen und bietet das passende Benutzerinterface. Siehe: `https://developer.android.com/reference/android/preference/Preference.html`. 20

Python Höhere Programmiersprache mit guter Lesbarkeit. 7

RTL8187 Spezifischer, gerne von Entwicklern verwendeter, WLAN Chip von Realtek. 8, 15, 16, 20, 31, 45

Runnable Wird benutzt, um Code in anderen Threads auszuführen. Siehe: `https://developer.android.com/reference/java/lang/Runnable.html`. 28, 29

Service Services können langlaufende Hintergrundaktivitäten ausführen und stellen keine GUI zur Verfügung. Siehe: `https://developer.android.com/guide/components/services.html`. 19–21, 23, 24, 28–30

SharedPreferences Interface zum Speichern der verfügbaren Einstellungen der APP. Siehe: `https://developer.android.com/reference/android/content/SharedPreferences.html`. 24

Socket Ein vom Betriebssystem bereitgestellter Endpunkt für Kommunikation mit anderen Sockets. 2

Service Set IDentifier Wählbarer Name eines WLANs unter welchem es ansprechbar ist. 12

Strict-Mode Der Strict-Mode ist eine Hilfe für Entwickler, um zu prüfen ob ihre App in der ausgeführten Activity wesentliche Berechnungen oder blockierende Aktionen ausführt. Das soll sicherstellen, dass die GUI nicht auf diese Aktionen warten muss. Sonst kann der Benutzer oder das Betriebssystem glauben, die App wäre abgestürzt.. 32

Transmission Control Protocol Verbindungsorientiertes Netzwerkprotokoll zur zuverlässigen Datenübertragung. 38

USB On-The-Go Ein Standard, der es dem Smartphone erlaubt als USB-Host zu agieren. 15

Wardriving Wardriving bezeichnet das systematische Suchen und Kartografieren von WLANs, meist von einem Fahrzeug aus. 44

Wired Equivalent Privacy Verschlüsselungsstandard für WLAN Netzwerke, sollte aufgrund von Schwachstellen nicht mehr verwendet werden. 6

WiFi Protected Access Verschlüsselungsstandard für WLAN Netzwerke, als kurzfristige Lösung zwischen WEP und WPA2. 7

Extensible Markup Language Sprache zur Darstellung hierarchischer Daten mit guter Lesbarkeit für Mensch und Computer. 31

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 20.Juni 2016 Fabian Mende

Diese DVD enthält:

- eine *pdf*-Version der vorliegenden Bachelorarbeit
- die LaTeX- und Grafik-Quelldateien der vorliegenden Bachelorarbeit samt aller verwendeten Skripte
- die Quelldateien der im Rahmen der Bachelorarbeit erstellten Software WTest, sowie ihre apk-Datei
- den bei Tests erstellten Datensatz an Aufzeichnungen und Auswertungen
- die Websites der verwendeten Internetquellen
- die kompilierbare Version des verwendeten Android PCAP
- ein Readme.txt mit Erläuterungen zu den Inhalten der DVD