NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

ADAPTING ANDROID DEVICES FOR MULTI-HOP COMMUNICATION IN INFRASTRUCTURELESS AD-HOC NETWORKS

by

Claes Nyberg

September 2018

Thesis Advisor: Gurminder Singh
Co-Advisor: John H. Gibson

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 2018	Master's thesis

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
ADAPTING ANDROID DEVICES FOR MULTI-HOP COMMUNICATION IN INFRASTRUCTURELESS AD-HOC NETWORKS	
6. AUTHOR(S) Claes Nyberg	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release. Distribution is unlimited.	A

13. ABSTRACT (maximum 200 words)

Affordable, lightweight, and energy-efficient commercial off-the-shelf (COTS) mobile devices provide a tactical and economical advantage for both military personnel and civilian organizations operating in environments with degraded or no communications infrastructure. Since COTS mobile devices today require an existing and stable infrastructure for communicating reliably "out of the box," this research explores how devices running the Android operating system can be modified and extended to support multi-hop communication in ad-hoc networks lacking infrastructure such as a server and satellite, or cellular or Wi-Fi connectivity.

We show that the previously available wireless ad-hoc mode can be enabled on select Android devices regardless of their version, and provide the design and implementation of the tools necessary to distinguish those devices as well as configure the wireless interface. Furthermore, we provide a list of Android devices already screened by the designed tool and explain how to leverage the enabled ad-hoc mode to design and implement a portable, adaptable, and usable chat protocol with multi-hop capability that does not require a server. Finally, we test and evaluate our implementation, which can be easily installed for communication among multiple mobile Android devices that randomly join or leave the network in environments lacking infrastructure.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Android, IBSS, multi-hop, ad-hoc,wireless, infrastructureless	111
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

ADAPTING ANDROID DEVICES FOR MULTI-HOP COMMUNICATION IN INFRASTRUCTURELESS AD-HOC NETWORKS

Claes Nyberg

Civilian, Department of Defense, Sweden

MS, University of Gothenburg, 2004

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2018

Approved by: Gurminder Singh
Advisor

John H. Gibson
Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Affordable, lightweight, and energy-efficient commercial off-the-shelf (COTS) mobile devices provide a tactical and economical advantage for both military personnel and civilian organizations operating in environments with degraded or no communications infrastructure. Since COTS mobile devices today require an existing and stable infrastructure for communicating reliably “out of the box,” this research explores how devices running the Android operating system can be modified and extended to support multi-hop communication in ad-hoc networks lacking infrastructure such as a server and satellite, or cellular or Wi-Fi connectivity.

We show that the previously available wireless ad-hoc mode can be enabled on select Android devices regardless of their version, and provide the design and implementation of the tools necessary to distinguish those devices as well as configure the wireless interface. Furthermore, we provide a list of Android devices already screened by the designed tool and explain how to leverage the enabled ad-hoc mode to design and implement a portable, adaptable, and usable chat protocol with multi-hop capability that does not require a server. Finally, we test and evaluate our implementation, which can be easily installed for communication among multiple mobile Android devices that randomly join or leave the network in environments lacking infrastructure.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1
A. PROBLEM STATEMENT 1
B. MOTIVATION AND RELEVANCE TO THE U.S. DEPARTMENT OF DEFENSE 1
C. SCOPE AND BOUNDARY 2
D. THESIS ORGANIZATION AND OVERVIEW 2

II. BACKGROUND 5
A. AD-HOC NETWORK OVERVIEW 5
1. Wi-Fi Ad-Hoc Mode 6
2. Wi-Fi Direct 8
3. Wi-Fi Direct on Android Devices 12
B. THE ANDROID OPERATING SYSTEM 12
1. Building Custom Firmware from AOSP 13
2. Building a Custom Kernel 17
3. Gaining Root Privileges on Android Devices 22
C. WIRELESS CHIPSETS 31
1. Overview of Wireless Chipsets 31
2. Broadcom Wireless Chipset 33
D. MANET RELATED PREVIOUS WORK 40
1. AdHocDroid 40
2. Thinktube 41
3. FireChat 41
4. Wi-Fi Direct Modifications 42
5. BLESSED with Opportunistic Beacons 43
6. 802.11s: The WLAN Mesh Standard 43
E. SUMMARY 45

III. DESIGN AND IMPLEMENTATION 47
A. ANDROID FIRMWARE SCANNERS 47
1. Firmware Scanner Design 47
2. OTA Update Package Scanner 50
3. Google Proprietary Drivers Package Scanner 50
4. Google Factory Firmware Package Scanner 51
B. AN EXAMPLE APPLICATION: IBSS CHAT 51
1. Design Goals 51
2. Wireless Interface Configuration 54

3. Serverless Chat with Multi-Hop Capability 56
C. SUMMARY 63

IV. TESTING AND EVALUATION 65
A. SCANNING ANDROID FIRMWARE PACKAGES 65
1. Downloading and Processing Firmware Packages 65
2. Evaluation 65
B. IBSS CHAT APPLICATION 67
1. Testing 67
2. Evaluation 72
C. SUMMARY 74

V. CONCLUSION AND FUTURE WORK 75
A. CONCLUSION 75
B. FUTURE WORK 76
1. How Can IBSS-Enabled Broadcom Firmware Be Ported Across Devices? 76
2. How Can IBSS Functionality Be Added to Existing Broadcom Wireless Chipset Firmware Files? 77
3. What Firmware Files from Other Manufacturers Implement IBSS Functionality That Can Be Used on Android Devices to Enable Ad-hoc Mode? 77
4. Chat Protocol Development 77

APPENDIX A. ANDROID DEVICES WITH IBSS-ENABLED BROADCOM FIRMWARE 79

APPENDIX B. ANDROID DEVICES WITH BROADCOM FIRMWARE 81

APPENDIX C. ANDROID DEVICES WITHOUT BROADCOM FIRMWARE 83

LIST OF REFERENCES 89

INITIAL DISTRIBUTION LIST 95

LIST OF FIGURES

Figure 1. An ad-hoc network without routers or access-points. 5

Figure 2. A Wi-Fi Direct network with the group owner in the center. 9

Figure 3. Flowchart for building and flashing custom Android firmware..... 14

Figure 4. Android software architecture stack. Source: [24]. 18

Figure 5. Configuring a custom Android kernel using menu configuration. 20

Figure 6. The Flattened Image Tree configuration file, FIT.cfg. 22

Figure 7. OEM unlocking and USB debugging enabled. 25

Figure 8. Pixel C waiting for fastboot command. 26

Figure 9. Samsung warning screen and download mode on Android 6.0..... 28

Figure 10. Samsung Odin graphical user interface. Source: [29]. 29

Figure 11. FullMAC and SoftMAC chipset layout..... 32

Figure 12. Broadcom chipset layout. Adapted from [34]. 33

Figure 13. Cypress processor memory layout. 35

Figure 14. Samsung Galaxy S6 sending ping in ad-hoc mode. 39

Figure 15. Firmware scanner pseudo code. 48

Figure 16. Pseudo code of the wireless interface configuration process. 54

Figure 17. Wireless interface configuration structure..... 55

Figure 18. Message header as a data structure. 57

Figure 19. Client joining chat and synchronization with an existing client..... 58

Figure 20. No acknowledgement received when trying to send a message results in failure. 59

Figure 21. Source code for calculating delay between resending of messages when waiting for acknowledgment. 60

Figure 22. Function for creating a BSSID. 68

Figure 23. Devices configured for multi-hop communication test. 71

LIST OF TABLES

Table 1. The AOSP build types. 16

Table 2. Chipset with IBSS firmware. 66

Table 3. Samsung devices used when testing. 68

Table 4. No-hop test with message acknowledgment. 70

Table 5. No-hop test without message acknowledgment. 70

Table 6. Multi-hop test with message acknowledgment. 72

Table 7. Multi-hop test without message acknowledgment. 72

Table 8. Android devices with IBSS-enabled Broadcom firmware..... 79

Table 9. Android devices with Broadcom firmware..... 81

Table 10. Android devices without Broadcom firmware..... 83

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

AOSP	Android Open Source Project
API	Application Programming Interface
ASIF	Android Sparse Image Format
ATIM	Announcement Traffic Indication Map
BLE	Bluetooth Low Energy
BSS	Basic Service Set
BSSID	Basic Service Set Identifier
CBC	Cipher Block Chaining
CDD	Compatibility Definition Document
CM	CyanogenMod
COTS	Commercial Off The Shelf
DHCP	Dynamic Host Configuration Protocol
DoD	Department of Defense
EAP	Extensible Authentication Protocol
FIT	Flattened Image Trees
GO	Group Owner
GUI	Graphical User Interface
IBSS	Independent Basic Service Set
ID	Identifier
IOCTL	Input Output ConTroL
IP	Internet Protocol
IPV4	Internet Protocol version 4
LAN	Local Area Network
MAC	Media Access Control
MANET	Mobile Ad-Hoc Network
MBSS	Mesh Basic Service Set
MLME	MAC subLayer Management Entity
MTU	Maximum Transmission Unit
NDK	Native Development Kit
NVRAM	Non Volatile Random Access Memory
OFDM	Orthogonal Frequency Division Multiplexing
OEM	Original Equipment Manufacturer
OS	Operating System

OTA	Over The Air
PIN	Personal Identification Number
POSIX	Portable Operating System Interface
RAM	Random Access Memory
ROM	Read Only Memory
SAE	Simultaneous Authentication of Equals
SINET	Secure Infrastructureless Network
SSED	Service Set Encoding-based Dissemination
SSID	Service Set Identity
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
UPNP	Universal Plug and Play
VPN	Virtual Private Network
WANET	Wireless Ad-Hoc Network
WEP	Wired Equivalent Privacy
WLAN	Wireless Local Area Network
WME	Wireless Multimedia Extension
WMM	Wi-Fi Multimedia
WPA	Wi-Fi Protected Access
WPS	Wi-Fi Protected Setup

I. INTRODUCTION

An Infrastructureless environment is an area in which infrastructure used for communication (such as communications satellites, cellular or WiFi networks) are either nonexistent, have been degraded, or destroyed intentionally or unintentionally. Both military personnel and civilian organizations operating in such environments face challenges in sharing information. As mobile devices with high-resolution touch screens have become more affordable, lightweight, and energy efficient, they have become popular for both military and civilian purposes. Nevertheless, for communications, these devices rely on existing and stable infrastructure to function “right out of the box.”

Before 2013, mobile devices running the Android operating system supported the Wi-Fi ad-hoc mode and were shipped with tools to configure the wireless interface. This made it easy to build multi-hop solutions by adding routing protocols such as BATMAN [1] or OLSR [2] on top of the wireless physical and link layers. Currently, however, the ad-hoc mode is not available to the end user and Android devices do not include any tools—not even for the root user—to configure the wireless interface.

A. PROBLEM STATEMENT

No reliable, efficient, and easily available solutions exist for commercial off the shelf (COTS) mobile devices running the Android operating system to establish multi-hop communications in infrastructure-less environments.

B. MOTIVATION AND RELEVANCE TO THE U.S. DEPARTMENT OF DEFENSE

This research acts as a platform for the implementation of secure communication on a higher level in the network stack. Through this research, military and civilian organizations may gain insight on how to extend or modify COTS mobile devices running the Android operating system to support reliable and secure multi-hop communication in austere environments. This would enable

leveraging both the economical and tactical advantages of using lightweight and readily available devices on the modern battlefield, as well as in areas where a reliable infrastructure is denied or degraded. By building upon previous research, this study explores solutions to take advantage of known methods to create secure communications as required by the U.S. Department of Defense.

C. SCOPE AND BOUNDARY

The primary focus of our research is to explore how COTS mobile devices running the Android operating system can be modified and extended to support reliable and efficient multi-hop communication using the IEEE 802.11 protocol in infrastructure-less environments. Though important topics on their own, the following are not covered in this research:

1. How to connect a mesh of mobile devices to another, separate infrastructure
2. How to protect data at rest and in use on mobile devices
3. How to implement encryption key distribution for secure communication

D. THESIS ORGANIZATION AND OVERVIEW

In Chapter II, we explain ad-hoc networks together with the challenges of implementing them. This explanation also provides insight into how the Wi-Fi Ad-Hoc and Wi-Fi Direct modes work. After a discussion on the internals of the Android operating system, such as the different types of Wi-Fi chipsets, the chapter presents an overview of how to modify the source code, build the different system firmware files, and how to flash them onto an Android device. Further on, related research on creating mobile ad-hoc networks (MANET) using different approaches for mobile COTS devices is discussed.

In Chapter III we describe the design and implementation of two software packages. The first is a set of programs for scanning different types of Android

firmware files to detect whether ad-hoc mode can be enabled on the device for which the firmware files are intended. The second software is an Android system service, which we call the Independent Basic Service Set (IBSS) Chat service. IBSS Chat service enables ad-hoc mode on Android devices and leverages this mode for implementing a server-less multi-hop capable chat protocol. The design and implementation of the IBSS Chat program is focused toward portability, adaptability, and usability from the aspect of principle of least privilege.

In Chapter IV we start by presenting the outcome of running the firmware scanning implementations on Android firmware files representing over 200 different mobile devices. We divide the results into three different groups according to the triage model and discuss the contents. The main focus of Chapter IV, however, is the testing and evaluation of the IBSS Chat program we have installed on six Android devices. We first run a series of stress tests where multi-hop communication is not required and then configure the devices for a worst-case multi-hop scenario and run the same tests. The chapter concludes with an evaluation of the performed tests.

Chapter V summarizes the research and concludes our findings. The chapter also proposes future work to continue this research.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A. AD-HOC NETWORK OVERVIEW

An ad-hoc network, also called a peer-to-peer [3] or computer-to-computer network, is a decentralized type of wireless network. It allows devices to communicate wirelessly without a pre-existing infrastructure that includes routers or access-points as shown in Figure 1.

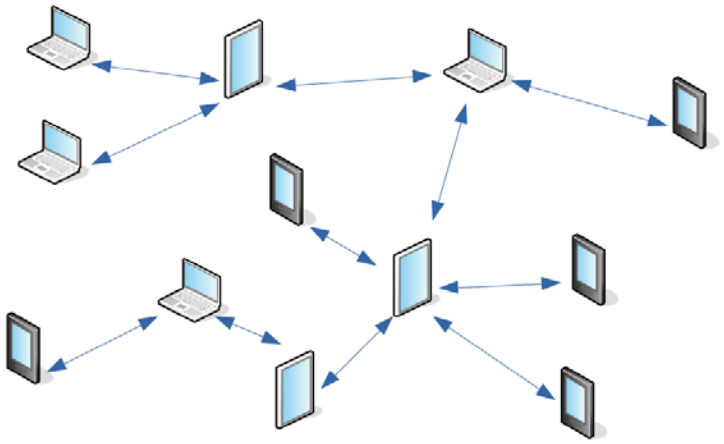

Figure 1. An ad-hoc network without routers or access-points.

The topology of an ad-hoc network is not fixed or pre-determined. Each device in the network can originate or forward traffic. Devices can connect to the network or disconnect at any time. When new devices connect to the network, the range of the wireless signal is automatically extended since the new device may forward traffic for any other device where the best path to other devices is through the new device. When a device disconnects from the network, the network self-heals by calculating new best paths as necessary to reach the remaining devices in the network. If links are degraded, perhaps by temporary obstacles, the network self-heals by finding an alternate path around the degraded link.

In order for an ad-hoc network to function, frames may need to be forwarded over multiple hops, and there are different techniques to accomplish this, which we explore later in this research. Recent research [4] finds that Wi-Fi provides enough range and throughput for the purpose of a reliable multi-hop solution. Independent of which wireless technology is used, there are many challenges with implementing ad-hoc networks [5].

First of all, since the devices can move arbitrarily, the network topology changes may be totally unpredictable and can occur rapidly and independently. This makes routing extremely difficult since devices cannot be assumed to have persistent routing tables. Any device can leave the network at any time for a variety of reasons, such as a battery outage or a simple reboot of its operating system. Furthermore, bandwidth constraints with wireless links compared to using physical wires pose another major challenge. The throughput of wireless links can be highly variable for many reasons, such as multiple device access, temporary obstacles, and interference, while wired links tend to provide stable bandwidth over time with less data loss and, therefore, lower retransmission needs. Implementations where a device uses carrier sensing for sending frames complicates congestion control since devices at the edges have a higher probability of being allowed to send than do devices in the center of the network. This makes it hard to ensure fair access for stable throughput over time in larger ad-hoc networks. Since wireless communication is sent over the air, security-related issues such as denial of service attacks, spoofing, and possible eavesdropping also pose a concern.

1. Wi-Fi Ad-Hoc Mode

The Wi-Fi ad-hoc mode, also called Independent Basic Service Set (IBSS), is available through most 802.11 driver implementations and is described in the IEEE 802.11 standard [6]. The Wi-Fi ad-hoc mode affects only the protocols used but does not affect the different modes of 802.11 in the physical layer [7]. Since the different modes of 802.11 are not affected, encryption protocols such as WPA

and WPA2 can be used to secure the communication, assuming they are up-to-date with respect to security patches.

For the media access layer, all media access control, such as carrier sensing, and the frames used are the same. Since there is no centralized access point, however, the ad-hoc nodes must take on more responsibility with respect to the operation of the media access control. The same channel is used by all devices in the Wi-Fi ad-hoc network.

a. The First Device in an IBSS Network

The first device in an IBSS network starts sending beacon frames on the selected channel, whereas the access point is the only device sending beacon frames in infrastructure-based networks. The beacons are used to maintain synchronization among all devices in an IBSS network and contain, among other information, the name of the network and a time interval between beacons.

b. Beacon Frames

Devices may join the ad-hoc network after receiving a beacon, and once devices have joined the network, they send beacons periodically if no beacon has been received from any other device after a short random time.

A beacon also contains a time stamp. When a device receives a beacon with a time stamp greater than the local clock, the device updates its local clock by setting the time to the time of the received time stamp. This synchronizes the devices so that actions such as power management [8] and beacon transmissions can be performed at the same time by all devices in the network.

c. Power Management

A device can save power by “going to sleep” by setting the power management bit in any frame transmitted by the devices. The other devices connected to the IBSS learn which devices are in power saving mode and buffer their frames for that device locally. The devices holding buffered frames will notify

the sleeping device by sending an Announcement Traffic Indication Message (ATIM) frame [9] telling the device that it has buffered data waiting. Since all devices receive ATIM frames even in power saving mode, any device with frames buffered for it at other devices will stay awake during the next beacon interval period to download its data.

d. Routing Protocols

In order for devices to communicate in an IBSS network, a routing protocol must be used to support multi-hop forwarding of frames between devices. There are many different protocols for this purpose; BATMAN [1] and OSLR [2] are the most frequently used.

2. Wi-Fi Direct

Wi-Fi Direct is a single-hop communication technology that works in a star topology, as shown in Figure 2. It uses a group owner (GO) as a center switch. Due to its single-hop star topology, Wi-Fi Direct cannot replace ad-hoc mode directly. A multi-hop solution is required for all clients to be able to communicate with each other in an environment without infrastructure.

Not only does this make the GO a single point of failure, it also requires increased energy consumption by the GO since the GO can enter power saving mode only when all clients are using the power saving features in the protocol [10], [11]. The Wi-Fi Direct technology is not an IEEE standard but a specification by the Wi-Fi Alliance [10]. Initially named “Wi-Fi peer-to-peer,” this technology is used by many different kinds of devices such as Samsung SMART TVs, PlayStation 4, and EPSON printers.

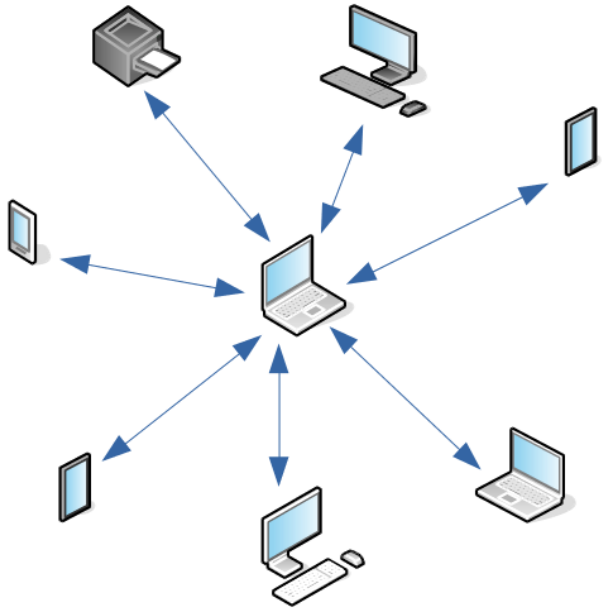

Figure 2. A Wi-Fi Direct network with the group owner in the center.

One of the advantages of Wi-Fi Direct is that only one device, the GO, needs to be compliant with the Wi-Fi Direct specifications for the network to function. The following are some requirements for the connecting clients, which are satisfied by most Wi-Fi enabled devices produced after 2010:

- Wi-Fi Protected Setup [12]
- Minimum of 802.11g
- Orthogonal Frequency Division Multiplexing (OFDM) data rates for management frames
- Wi-Fi Protected Access (WPA2) with AES-CCMP encryption
- Wireless Multimedia Extension

Devices use their media access control (MAC) address as a unique identifier when discovering and negotiating access to the group but generate a temporary MAC address for frames sent within a group.

a. Wi-Fi Direct: Device Discovery

Wi-Fi Direct devices find each other by listening and sending probe requests on channels 1, 6, and 11 in the 2.4 gigahertz (GHz) band. GOs and unconnected devices respond to probe requests with probe response frames. Both probe request and response frames contain a description of the device and details about the group. GOs respond for the devices in their group, but clients connected to a group owner can also choose not to be discoverable. The description of a device includes the following information:

- Device type (e.g., “Printer,” “Smartphone”)
- User configured device name (e.g., “My Camera”)
- Confirmation of whether additional connections are supported
- Confirmation of whether service discovery is supported
- Supported Wi-Fi Protect Setup (WPS) configurations
- Supported channels

The group information includes:

- Whether the GO provides cross-connection (i.e., connection to another network)
- Whether the group is persistent (as explained later in the text)
- Whether the group limit allows more devices to connect
- Whether clients may exchange data

b. Wi-Fi Direct: Service Discovery

When a device is discovered, it can be probed for services before it actually joins a group. It is not mandatory for the client to respond, and various protocols are supported for describing services provided by the client; these include Bonjour,

UPnP, and Web Services Dynamic Discovery. This simplifies automatic connection to services based on names; for example, a user can connect to “My Camera” to capture images or video.

c. Wi-Fi Direct: Group Creation

Any device can take the role of a GO or client but the roles of all connected devices are negotiated [13], [14]. The eagerness of becoming the GO can be expressed with four bits on a scale where 15 indicates the most interest. When a GO is negotiated, a confirmation message is sent to all devices and the selected channel is used [14]. The group owner sends out beacon frames for the group containing the negotiated Service Set Identity (SSID), which is standardized to start with “DIRECT” and followed by a hyphen and a sequence of random characters and numbers [13].

d. Wi-Fi Direct: Wi-Fi Protected Setup

When a client connects to the GO, it uses the WPS protocol to transfer eight Extensible Authentication Protocol (EAP) [15] messages. The user of a device normally has to push a button or enter a PIN to complete the setup. To avoid user acknowledgement through pushing a button or entering a PIN for a specific group, the group can be made what is called “persistent,” in which case the device saves the entered PIN or the key pressed for joining the group automatically without user interaction in the future [13], [14].

When the authentication is finished, the WPA2 handshake takes place to exchange encryption keys. The client then requests an Internet Protocol version 4 (IPv4) from the GO, which must run a Dynamic Host Configuration Protocol (DHCP) server. The DHCP server running on Android defaults to assigning addresses to connecting clients from the private network 192.168.49.0/24 [13]. The network cannot be changed without root access, and it is impossible for connecting clients to be part of multiple groups with identical network addresses.

e. Wi-Fi Direct: Power Management

Wi-Fi Direct added some extensions to the power management scheme of 802.11 [13], [14], where the “Opportunistic Power Save” allows the GO to enter power save mode in periods when all connected clients are running in power save mode [13], [14]. The GO can also include a “Notice of Absence” in the beacon frames telling clients that it will be absent for a while. Clients can send a presence request message to the GO to inform it that they have distinct delay requests or important data to send.

3. Wi-Fi Direct on Android Devices

The Android operating system dropped ad-hoc mode in deference to Wi-FI Direct in version 4.0 API level 14. The critical issue though is Android’s lockdown of the DHCP service configuration. Android statically assigns the GO the address 192.168.49.1 in the Internet Assigned Numbers Authority (IANA) designated private address space. This allows for a theoretical maximum of 253 devices to be connected, but complicates extensions of the software to allow a device to be part of multiple groups [13], [14]. If the DHCP service could be configured dynamically, then it would be possible to utilize Wi-Fi Direct for multi-hop networks by joining multiple groups simultaneously since different groups could have different addresses assigned.

B. THE ANDROID OPERATING SYSTEM

The Android operating system is based on Linux and developed by Google. Most of the source code, with the exception of a small number of proprietary parts, is available for download through the Android Open Source Project (AOSP) [16]. The source code available from AOSP can be used for building firmware images for Google devices, which include both tablet and phones of various sizes.

In order to ensure that applications run efficiently on devices from companies other than Google, such as Samsung, Sony, LG, and HTC, among others, the Android compatibility program defines a set of technical specifications

to be fulfilled by original equipment manufacturers (OEM) [16]. These requirements are listed in the Android Compatibility Definition Document (CDD) [17], and a free commercial grade test suite is available for download to encourage compatibility.

Some manufacturers make their additions and modifications of the AOSP source code available for download. One example of this is the Samsung Open Source Release Center [18], which provides source code for a majority of the Samsung devices running Android.

The source code available through AOSP is not only the foundation for all the manufacturers that wish to use the Android operating system on their devices, but it also functions as a platform for open source projects such as the Unlegacy Android project [19], which aims to port the latest releases of the Android operating system to older devices that no longer receive updates.

The availability of source code makes it possible to customize Android devices with very few limitations. The largest customization project at the time of this writing is the LineageOS [20], which is a continued development of the CyanogenMod (CM) project that had been terminated in 2016 [21].

1. Building Custom Firmware from AOSP

The actions required to build Android firmware from AOSP are documented in detail at the AOSP website [16] and can be summarized in a number of steps as visualized by the flowchart in Figure 3.

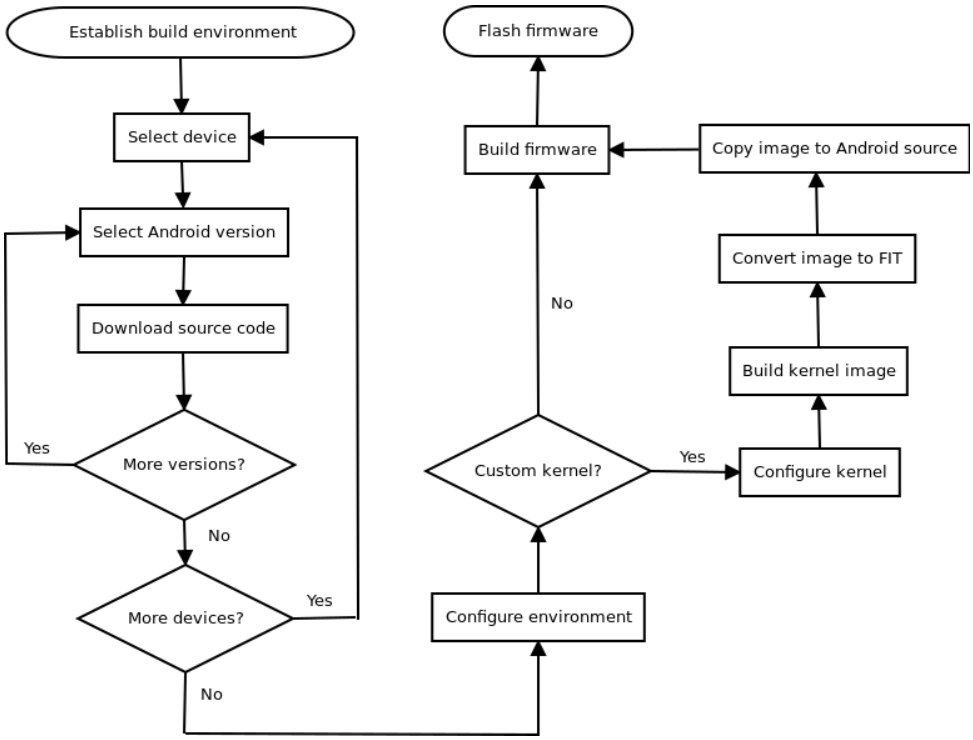

Figure 3. Flowchart for building and flashing custom Android firmware.

The Google Pixel C is used for our examples, but with some modifications it should apply for other Google devices as well.

a. Establishing a Build Environment

The first step is to set up a build environment by selecting the operating system to use. Only Mac OS and Linux are supported for building the Android system from AOSP. For the purpose of this research, Ubuntu Linux 16.04.3 LTS was selected. After obtaining the operating system for the build environment, a small number of required packages must be installed, which differ depending on the operating system used and the Android version to be built.

For Ubuntu Linux 16.04.3 LTS, and post Ubuntu 15.04 in general, the commands for adding the single required Java package, when building for Android 7.0 (Nougat) and Android 8.0 (Oreo) are:

```
sudo apt-get update
```

```
sudo apt-get install openjdk-8-jdk
```

b. Download Source Code

The Android source code is available through a Git repository and has a large number of source code tags for different devices and Android versions. In this step, the Android version to build for a target device is selected and then downloaded using the repo command obtained from Google [16]. For the Pixel C, we selected the Android 8.0 release 13 branch, which was downloaded using the following commands:

```
BRANCH=“android-8.0.0_r13”
URL=“https://android.googlesource.com”
URL=“$URL/platform/manifest”
mkdir $BRANCH
cd $BRANCH
repo init –u $URL -b $BRANCH
repo sync
```

The AOSP source code directory structure contains a configuration for the shell environment that is activated using the source command from the root of the downloaded source code directory:

```
source ./build/envsetup.sh
```

This is an important step as it configures the shell environment for further configuration and building of the firmware images from the current shell.

c. Preparing to Build

For the proprietary parts of the Android operating system where there is no source code available, the proprietary parts are represented in the vendor.img file, which resides in a self-extracting shell script where a license agreement must be accepted before the binaries are copied into the source code directory [16]. When

the proprietary binaries are in place, the build target can be selected for the device. The build target consists of the leading string “aosp,” followed by an underscore character, and then a codename for the selected device followed by a hyphen and one of three different build types shown in Table 1, as given by the following example wherein the `lunch` command, which is the distributed process launcher command for Linux [22], is utilized.

Table 1. The AOSP build types.

Build Type	Use
user	Provide limited access and are best suitable for production build
userdebug	Same as “user” but also with root access and ability to debug
eng	Development build which provides additional debugging tools

The build target for the target device is selected using the lunch command in the root of the AOSP source code directory. The codename for the Google Pixel C is dragon, so selecting a build type of userdebug which provides the most abilities is executed with the following command:

```
lunch aosp_dragon-userdebug
```

After running the build type command in the same shell as the build environment was configured previously, everything is set up for compiling the source code and building the required firmware images.

d. Building Firmware Images

Compiling and building firmware images is achieved by running the make command in the source code directory, preferably with the –j option on a multi-core machine, to provide the number of threads necessary for speeding up the build process. Once the builds have finished, which could take several hours, the firmware images are stored in a sub-directory of the “out” directory, which is located in the root of the AOSP source code directory.

The firmware images are stored in the Android Sparse Image Format (ASIF) [23]. The fastboot protocol [16], which is the default Android protocol for flashing firmware images, uses ASIF. There are numerous files generated by the build; most important are the following three firmware images.

(1) system.img

This is a partition image that is mounted on the system directory located in the system root. It contains binaries required by the Android system, such as applications, fonts, and media codecs. This is the image that changes the most between Android releases and the one in which most Android users are interested when flashing a new firmware image.

(2) boot.img

This image contains the Linux kernel and a ramdisk, which is a read-only file system mounted as the root file system during the boot process. It contains files and binaries required by the core operating system, such as init, which is the first process created during boot-up together with a number of critical daemons.

(3) vendor.img

This partition image contains vendor-specific files and is not present on all devices as it is sometimes included within the system image. This is where the proprietary binaries, extracted earlier into the source code directory, are located on booted Google devices. The additional file system this image represents is mounted at the vendor directory in the root of the Android device.

2. Building a Custom Kernel

The Android operating system is made up of multiple layers with different software modules where the kernel is represented as a separate unit and the steps required for building an AOSP kernel are at the time of this writing not documented in the same detail as when building the Android system. The different software

layers of the Android operating system are shown in Figure 4 where the kernel subsystem is at the bottom layer and marked in pink.

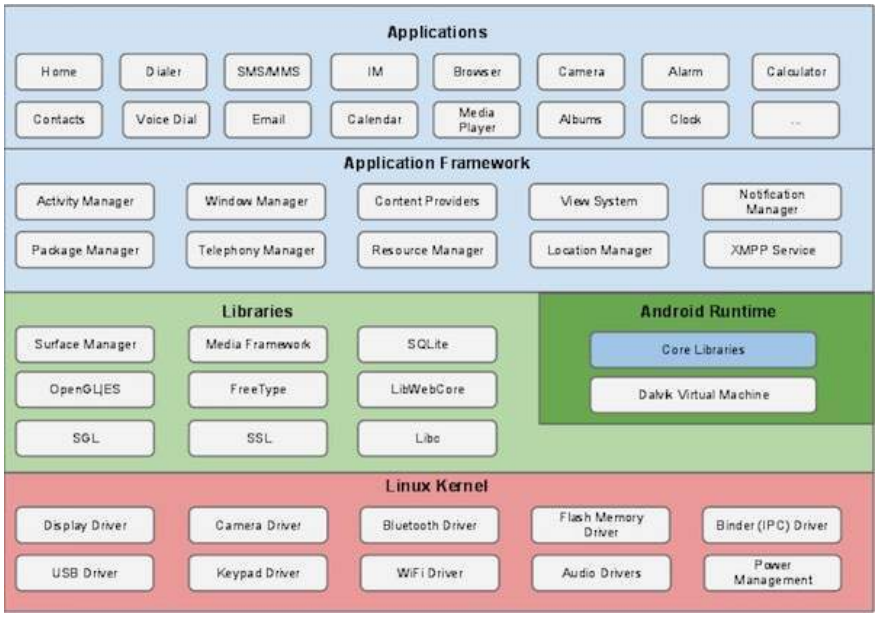

Figure 4. Android software architecture stack. Source: [24].

The boot.img that was generated when building Android includes a default kernel image stored in the Flattened Image Tree (FIT) format [25]. The default kernel image is located inside the AOSP source directory in a subdirectory named after the codename of the target device. For the Pixel C device, which uses the dragon codename, the image is located at device/google/dragon-kernel/Image.fit from the root of the AOSP source code directory.

The kernel source code used to build the kernel image must be downloaded separately as it is not part of the AOSP source code downloaded previously. The FIT image inside the source code directory has to be replaced with an image that includes a custom-built kernel to generate a custom boot.img to be flashed and booted by the Android device [16], [26].

Hence, the first step for building a custom kernel is to download the source code for the kernel that matches the Android source code previously downloaded.

The official source code for the Android kernel tree for all devices can be found online as a Git repository [26]. Once the kernel source that matches the Android source code is located and downloaded, the shell environment has to be set up in the same way as when building the Android image previously.

The kernel source code for the Pixel C that matches the Pixel C source code for the Android version has the codename “tegra” and is downloaded using the following commands:

```
BRANCH=“android-tegra-dragon-3.18-nougat”

URL=“https://android.googlesource.com/kernel/tegra”

mkdir $BRANCH

cd $BRANCH

git clone –b $BRANCH $URL
```

For the Pixel C device, the following commands then have to be executed from the root of the AOSP source code directory to configure the shell environment, if not already done:

```
 source ./build/envsetup.sh

lunch aosp_dragon-userdebug
```

The current directory should then be changed to the root directory of the kernel source and additional environmental variables exported for cross-compiling. For the Pixel C these are:

```
export ARCH=arm64

export CROSS_COMPILE=aarch64-linux-androidkernel-
```

A kernel configuration must be generated if it does not already exist. To generate a default configuration and overwrite any existing configuration, there is a make command with the device codename followed by an underscore and “defconfig.” For the Pixel C device with the dragon codename the command is:

```
make dragon_defconfig
```

At this point a default kernel image can be built using the make command. This is a good way to start, by making sure that the image is built successfully and can be booted before making any modifications to the source code or configuration. Nonetheless, since the Android operating system uses the Linux kernel, it can be easily configured further using the make target “menuconfig” by executing the following command in the top directory of the downloaded kernel source code:

```
make menuconfig
```

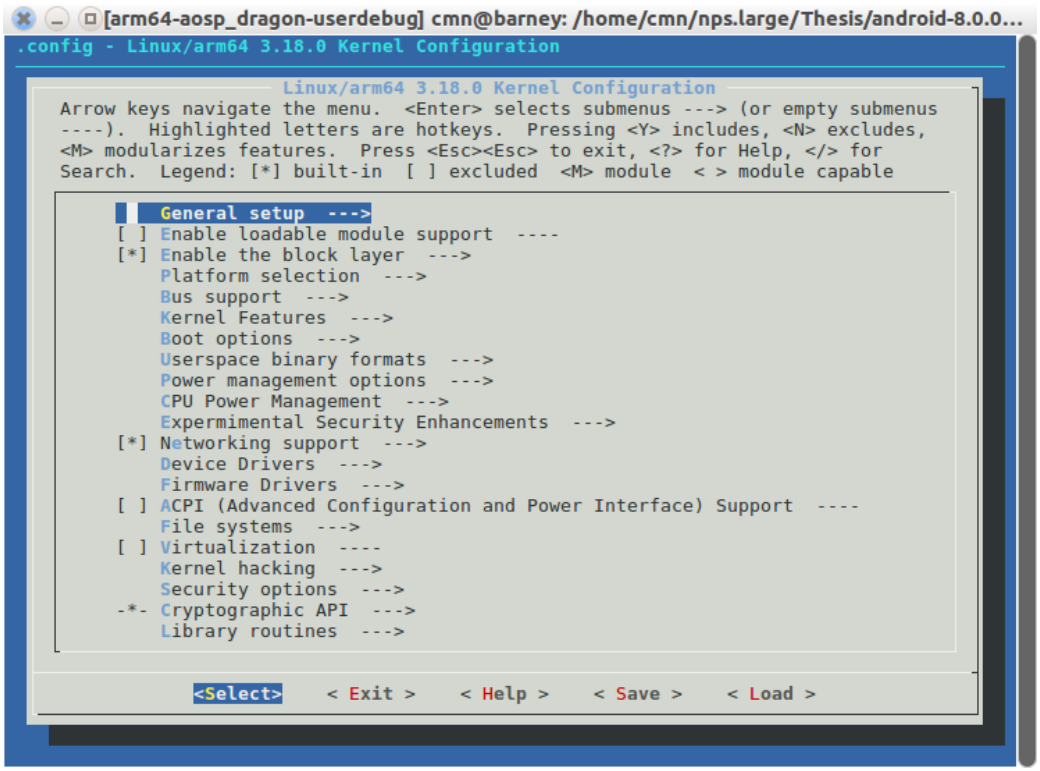

Figure 5. Configuring a custom Android kernel using menu configuration.

Apart from enabling existing features using standard configuration tools, we can modify the kernel code itself and tailor it for specific needs before building a custom image. As we see later, it is also easy to test the new kernel on an Android

device without overwriting the default kernel or making any modifications at all to the existing system.

The build is started by running the make command in the root of the source directory. When the build finishes, an uncompressed kernel image is stored in the subdirectory, “boot,” for the target architecture. For the Pixel C, the image is located at arch/arm64/boot/Image from the root of the kernel source code directory.

The first step toward creating the FIT image is to compress the image using the LZ4 algorithm, which is available as the `lz4c` command under Linux. The compressed image should be stored in the root of the kernel source directory as Image.lz4, since we will use a relative path to reference it later. This can be done from the root of the kernel source code tree by running the following command:

```
lz4c ./arch/arm64/boot/Image ./Image.lz4
```

The next step is to create a configuration file to use as input to the `mkimage` command that finally generates the required FIT image. The FIT image stores the custom kernel image together with a Flattened Device Tree (FDT) that describes the hardware of the device [27]. The FDT file is built together with the kernel image, and for the Pixel C, this file is located under the subdirectory “tegra” at arch/arm64/boot/dts/tegra/tegra210-smaug-p1.dtd from the root of the kernel source directory [26]. The locations of the FDT file and the compressed kernel image are important since they are referenced relative to the FIT configuration file, as shown in Figure 6.

```
/dts-v1/;
/ {
    description = "Simple image with Linux kernel and FDT";
    #address-cells = <1>;

    images {
        kernel@1 {
            description = "Custom Linux Kernel for Pixel C";
            data = /incbin/("./Image.lz4");
            type = "kernel_noload";
            arch = "arm64";
            os = "android";
            compression = "lz4";
        };
        fdt@1 {
            description = "Flattened Device Tree";
            data = /incbin/("./arch/arm64/boot/dts/tegra/tegra210-smaug-p1.dtb");
            type = "flat_dt";
            arch = "arm64";
            compression = "none";
            hash@1 {
                algo = "sha1";
            };
        };
    };
    configurations {
        default = "conf@1";
        conf@1 {
            description = "Boot Linux kernel with FDT";
            kernel = "kernel@1";
            fdt = "fdt@1";
        };
    };
};
```

Figure 6. The Flattened Image Tree configuration file, FIT.cfg.

Using the FIT.cfg file as input, the FIT image is generated with the following command:

```
mkimage –f FIT.cfg Image.fit
```

The Image.fit file is then moved to its location in the AOSP directory and the firmware image, boot.img, can be built with a single command from the root of the AOSP source directory:

```
make bootimage
```

3. Gaining Root Privileges on Android Devices

On a UNIX system, the root user has the highest privileges. Although the user is named root by default, it has a user identity value of zero and all accounts with that value have the highest privileges, which makes them “super users.” On a UNIX system, the super user has no limitations on what he or she can do with the

system since access is granted for everything when it comes to reading, writing, executing, and changing permissions.

The Android system is part of the UNIX family of operating systems, and super user privileges make it possible to enable or disable features and tailor the system to specific needs that differ from the system originally installed on the device. As we see later, this is mandatory to enable ad-hoc mode on certain devices and is our main reason for the execution of this research.

Elevating a user's privilege level on the Android operating system, from that of a regular user to the root user, is in most cases performed by executing the super user (su) binary. The su binary opens a new shell running with user identity of zero to elevate the privileges of the user executing the binary. This is accomplished by enabling the "set user identity bit" on the su binary and by changing the owner to root. The "set user identity bit" makes the process run as the user owning the executed binary, in this case the super user, but it could be any user on the system.

The step of assuring permanent root privileges includes installing the su binary onto the file system with the required permissions. This is most commonly known as "rooting the device." However, this is disabled on most devices since executing set-user-id binaries is prevented by security policies as well as the /system directory being mounted with the "nosuid" option, which also prevents set-user-id binaries to be executed with escalated privileges. In order to achieve root privileges on those devices, a system service is started upon boot as the root user, which local users then interact with to execute commands as root. This is accomplished by adding a command to the /init.rc file in the boot.img firmware file on Android devices.

a. Gaining Root from Flashing Firmware

The most common way of rooting an Android device is to flash a firmware that has the su binary installed on the file system, such as the custom system.img

built with the userdebug build type as described previously. In order to flash firmware onto an Android device, the boot loader has to be unlocked.

(1) The Boot Loader

The boot loader can be thought of as the gatekeeper of the system, as it is the first process that runs when the system is powered up and controls where to redirect execution to continue the boot process. Most boot loaders are locked by default to prevent users modifying the different firmware on the device. Some manufacturers permit unlocking of the boot loader on selected devices; Google permits unlocking on all of their devices to simplify customization through flashing [16].

(2) Unlocking the Boot Loader

Unlocking a boot loader will void the warranty of the device and is considered a great security threat. If the device with an unlocked boot loader is lost or stolen, there is no protection in place to prevent all the user data from being extracted or the device from being overwritten with a new firmware. To protect the privacy of the owner, all user data is erased from the device when the boot loader is unlocked.

For Google devices, the unlocking is performed by executing a few steps on USB-connected devices [14] and requires the fastboot and adb commands, which are available from the AOSP Git repository [16]. But developer mode must first be enabled on the Android device in question.

Developer mode is enabled by tapping “Build number” multiple times. The “Build number” is located inside the “About phone” menu from the settings menu. When developer mode is enabled, the developer settings menu becomes available inside the “About phone” menu. In the developer settings, “OEM unlocking” and “USB debugging” must be enabled to unlock the boot loader.

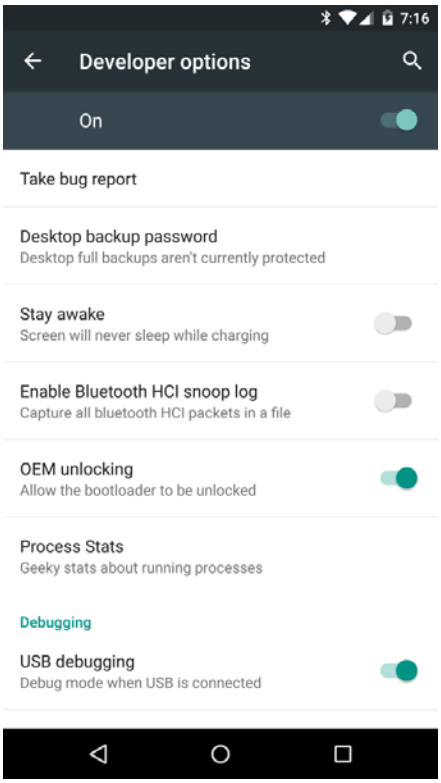

Figure 7. OEM unlocking and USB debugging enabled.

The device is then rebooted into the boot loader, and the boot loader is unlocked after the user has confirmed the unlock using the touch screen. The reboot and unlocking is executed with the following commands on devices released in 2015 and later [16]:

```
adb reboot bootloader
fastboot flashing unlock
```

On devices produced prior to 2015 [16], the following commands are used:

```
adb reboot bootloader
fastboot oem unlock
```

(3) Flashing Firmware

Once the boot loader has been unlocked, the phone can be flashed with custom firmware, but the Android device must first be put in “download mode” to

receive and flash firmware images. Google devices use the standard Android protocol fastboot [16] for flashing images. Manufacturers can have other solutions. Samsung uses its proprietary Odin protocol on a majority of devices [28], and on some of them, both Odin and fastboot are supported. The method to enter download mode differs among devices and depends on the boot loader. Typically, different keys are pressed and held when the device is powered-on to inform the boot loader to continue execution in download mode, instead of redirecting to the Android kernel, and to boot the installed system.

On Google devices, fastboot mode can also be entered using the following adb command for USB-debugging-enabled devices:

```
adb reboot bootloader
```

When the device has rebooted, it is waiting for a fastboot command. On Pixel C the message “Waiting for fastboot command” is displayed on the screen, as shown in Figure 8.

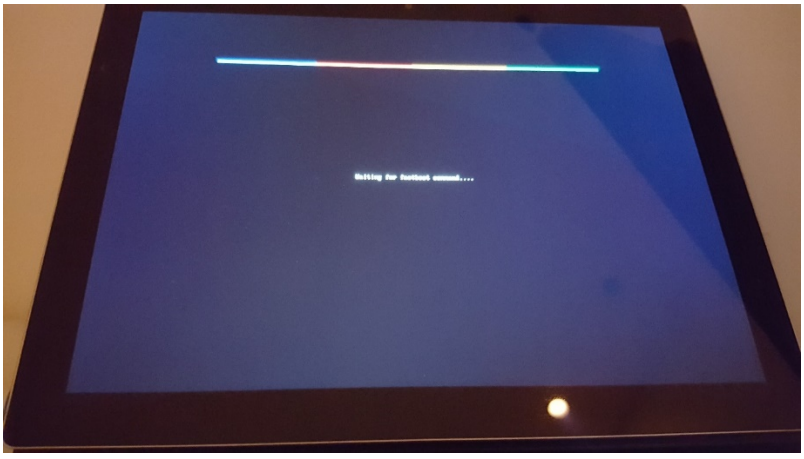

Figure 8. Pixel C waiting for fastboot command.

At this point, the device is ready to receive a command for flashing firmware, erasing data, or simply rebooting the system. To flash all the important firmware images on the Pixel C and then reboot the system, execute the following commands:

```
fastboot –w flash system system.img

fastboot flash boot boot.img

fastboot flash vendor vendor.img

fastboot reboot
```

Notice the `–w` option in the first `fastboot` command; this will ensure that the system is completely erased before flashing the firmware files and requires all firmware files mentioned previously to be flashed afterwards. The erasing is sometimes required when flashing a different version of Android to make the system boot.

Not all firmware files have to be flashed, though. It is possible to flash only a custom boot.img and leave the rest of the system intact. Another convenient feature is to boot a boot.img without flashing it. This allows for testing custom kernels without modifying the system at all. This is performed with the following single command when the device is waiting for a fastboot command:

```
fastboot boot boot.img
```

Samsung devices that use the Odin protocol must first be booted into the Odin download mode by pressing and holding a combination of keys when powering on the device. The boot loader on most Samsung devices first presents a warning about download mode that has to be acknowledged by the user before Odin download mode is entered.

The Odin download screen differs among Samsung devices, but the functionality is the same as it implements the protocol to flash firmware. On Android devices running 6.0 (Marshmallow) and later, the warning and download screen are displayed as shown in Figure 9.

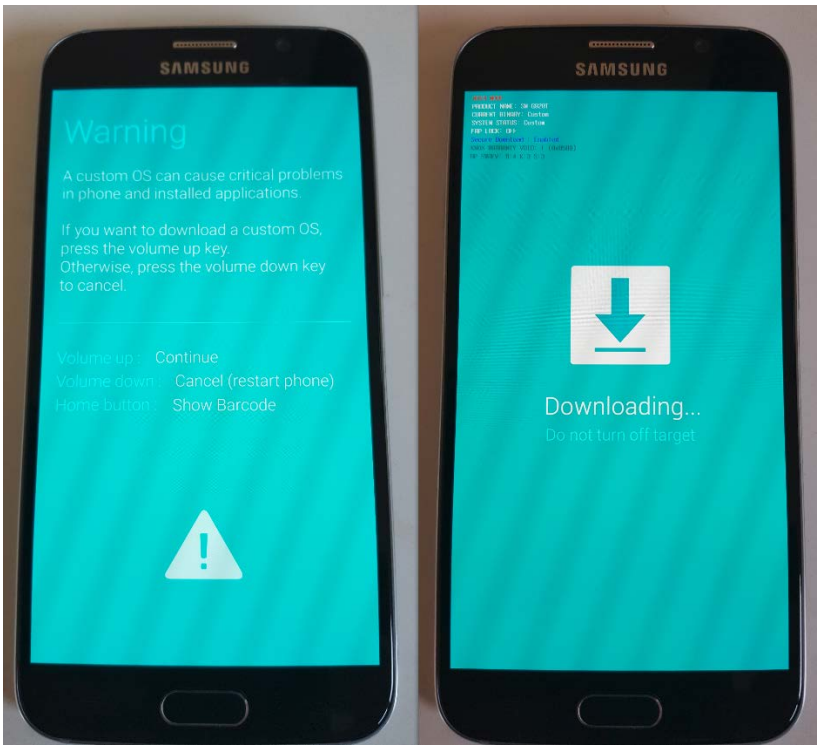

Figure 9. Samsung warning screen and download mode on Android 6.0.

Once the Samsung device has entered the download mode, it is ready to receive and flash firmware. This can be accomplished by using the Samsung Odin software, which is available for Microsoft Windows operating systems [29] with a graphical user interface, shown in Figure 10.

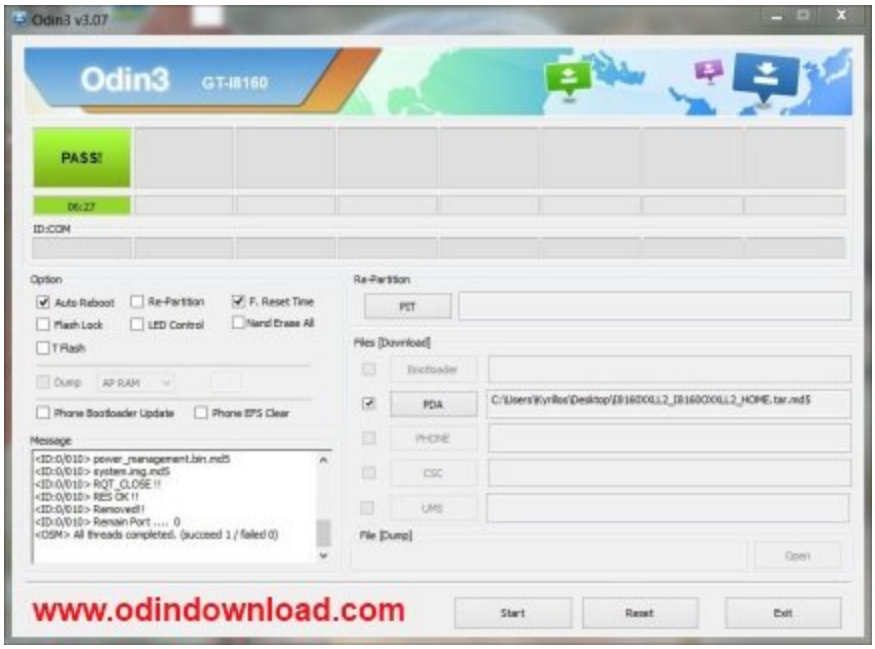

Figure 10. Samsung Odin graphical user interface. Source: [29].

b. Android Firmware Packages

Firmware for Android devices can be downloaded as packages in different formats. The Android Over The Air (OTA) update package is the most commonly used and contains information on how to update an existing Android system [16]. The most important components in the OTA package are the system.new.dat and system.transfer.list files, which together represent the files that should be removed, added, and overwritten on the target system as well as any possible permission changes. The OTA package stores all of its files in a zip archive that is automatically extracted by software such as fastboot on Google devices and Odin for Samsung.

Google provides original images for their devices to simplify restoring of devices to factory mode [24]. The factory firmware images are in the Android Sparse Image format [23] and stored in zip archives. Once the images have been extracted from the archive, they should be flashed in the same way as when flashing custom-built images.

c. Gaining Root from Software Vulnerabilities

One way to obtain super user privileges without unlocking a boot loader and flashing firmware that includes the su binary is to take advantage of security vulnerabilities that can be exploited to elevate privileges. Although this is a possible way to achieve higher privileges on a device, it is not an approach on which to build a reliable solution since vulnerabilities should be patched as soon as they are discovered to preserve the security of devices in the organization where they reside. This way of obtaining super user access is mentioned here for broadening the perspective of obtaining root access and presents an approach that is not always suggested by the manufacturer.

There are numerous publicly available exploits for Android devices that work on different versions [30]. Exploiting a system to get super user privileges first results in a temporary root, sometimes called a soft root, which will be lost when the system is rebooted. The soft root can be made persistent by installing the su binary into the filesystem. This is performed with following four steps:

1. Re-mount /system read and write
2. Copy the su binary into /system/bin
3. Set permissions on the su binary
4. Re-mount /system read only

Depending on the exploit used, the su binary might be installed automatically, but this can be done from a root shell on the Android device with the following commands (assuming that the su binary previously was uploaded to /sdcard/su):

```
mount –o remount,rw /system
cp /sdcard/su /system/xbin/su
chmod 06755 /system/xbin/su
mount –o remount,ro /system
```

Additionally, there are different Android applications that can be installed to grant and manage super user access to installed applications on the system. These work by requesting user confirmation on-screen when the su binary is executed by an application.

C. WIRELESS CHIPSETS

The following sections intend to clarify the different type of wireless chipsets most commonly used on mobile devices.

1. Overview of Wireless Chipsets

The use of Wi-Fi has increased rapidly over the last decades and is today a dominant technology on mobile devices for wireless communication. Correspondingly, the design of wireless chipsets has evolved rapidly over the last decades. In order for Wi-Fi devices to communicate, a protocol for access to the wireless medium is required and the entity responsible for this is called the Media Access Control (MAC) subLayer Management Entity (MLME) [31], [32].

The MLME implements the required functionality for core wireless logic and tasks, such as:

- Beacon
- Associate
- Disassociate
- Re-associate
- Authenticate
- De-authenticate

Many of the early wireless chipsets provided an Application Programming Interface (API) for reading and writing raw IEEE 802.11 frames. Together with

chipset-specific features, these are called SoftMAC chipsets, for which the host operating system is responsible for implementing the MLME [31].

Due to the complexity of the MLME, and the difference in chip-specific APIs, more and more chipsets integrate the MLME functionality and provide an API for IEEE 802.3 (Ethernet) frames to the host instead. Those are known as FullMAC chipsets [31], [32]. Figure 11 shows the layout differences in the communication stack of FullMAC and SoftMAC chipsets.

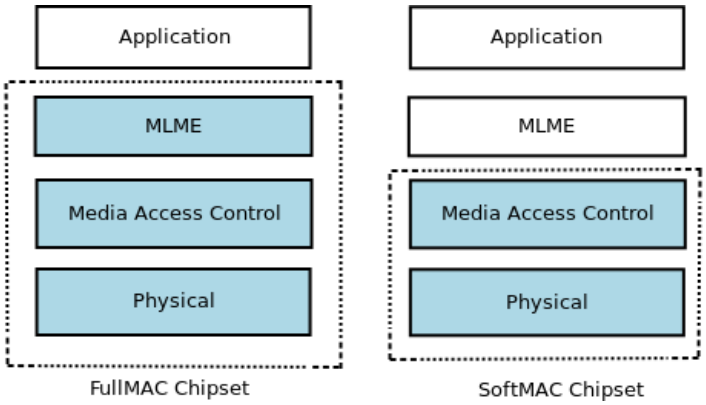

Figure 11. FullMAC and SoftMAC chipset layout.

The FullMAC chipsets reduce complexity for the host operating system, but also improve power consumption [23], which are the two major reasons for the widespread usage of FullMAC chipsets today.

The downside of a FullMAC chipset from a developer's point of view is the limitation of possible modifications since the MLME implementation, together with the rest of the code running on the chip, in most cases, is proprietary and there is no source code available. The FullMAC chipsets implement their functionality in firmware files that are loaded onto the chip by the host operating system using the driver software for the chipset.

2. Broadcom Wireless Chipset

A frequently used set of wireless chipsets on Android devices, especially for Samsung-manufactured devices, is the Broadcom chipset [33] in its different versions, such as the BCM4339 and BCM4358 [31], [32].

The Broadcom family of wireless chipsets is FullMAC that implements MLME in its firmware. According to available datasheets and reverse engineering performed by Beniamini [31], and Schulz, Wegemer, and Hollick [32], the design is similar for the Broadcom chipsets commonly used on Android devices [31], [32].

a. Chip Layout

The Broadcom wireless chipsets used on Android devices use the ARM Cortex R4 processor [31], [32] for the MLME implementation and IEEE 802.11 logic as shown in Figure 12 [34].

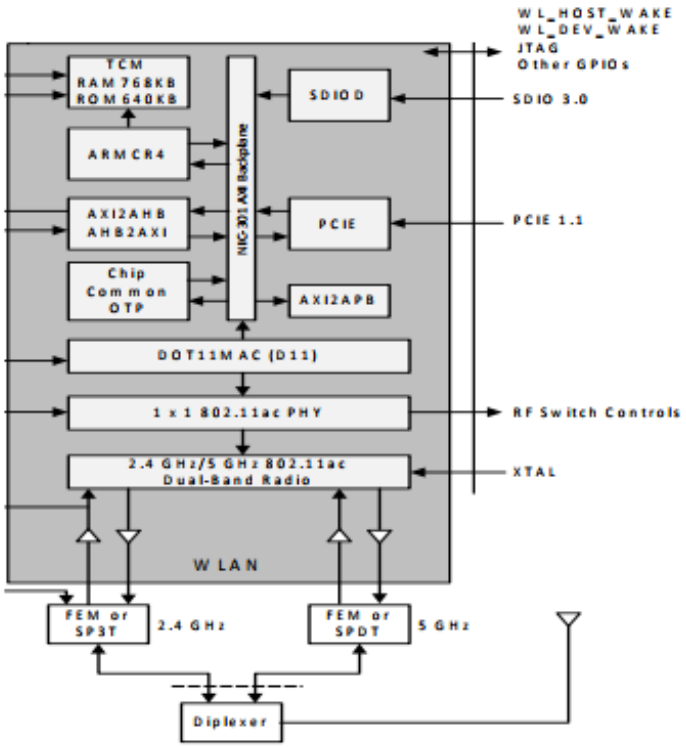

Figure 12. Broadcom chipset layout. Adapted from [34].

The ARM Cortex R4 has 640KB of Read Only Memory (ROM) [34], [35] for storing firmware and 768KB for Random Access Memory (RAM) used for data and stack storage. This limits the size of functionality that can be stored in the firmware. The Cortex R4 can use different instruction sets, including the memory efficient Thumb-2 [34], [35]; which is something that Broadcom leverages to store more code, given the limited size of the chip storage, and improve code density [31].

Apart from packing code into the narrow memory space, different firmware files can be used and loaded on demand for different functionality depending on the configuration of the wireless network card. As we will see later, some Android devices ship with a set of firmware files that can be activated with root privileges from the terminal when the Android Graphical User Interface (GUI) does not provide the functionality.

b. Firmware

The optimization of Broadcom firmware files includes removed symbols and strings [31]. Apart from that, the memory is reused by overwriting functions when they have been called the last time [31], resulting in very compact firmware files.

The firmware file is loaded to the memory locations associated with addresses 0x180000 to 0x240000 [32]. The ROM that contains the base of the code executed by the processor is located at address 0x000000 and spans all the way to 0x0a0000, leaving an unused gap in the middle, which is illustrated by Figure 13 [31].

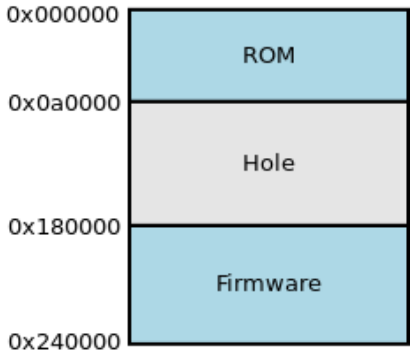

Figure 13. Cypress processor memory layout.

Since multiple firmware files can be used for different functionalities, there must be a way to distinguish between what functionality is supported in a specific firmware; fortunately, each firmware carries a version string that includes this information by merging different tags that individually mark specific functionality [31]. By using the strings command in Linux, the version string for a specific firmware file can be easily displayed:

```
strings bcmdhd_ibss.bin | tail -1
```

```
4358a3-roml/pcie-ag-p2p-pktctx-aibss-relmcast-
proptxstatus-ampduhostreorder-sr-aoe-pktfilter-keepalive-
clm_ss_mimo-sarctrl-sstput-xorcsum-fpl2g            Version:
7.112.17.19 CRC: 1d3581a2 Date: Thu 2016-03-10 15:36:56 KST
Ucode Ver: 986.11001 FWID: 01-aefde609
```

As we can see from the output of the previous command, the firmware supports the IBSS mode since the “aibss” tag is present in the version string [23].

The firmware version number (7.112.17.19) is closely related to the code in the ROM of the chipset, so firmware files cannot in general be moved between systems with different ROM code, even though they use the same Broadcom chipset [36].

c. The bcmdhd Driver and the dhdutil Program

The bcmdhd is the driver for Broadcom chipsets used by the Linux and Android operating systems and is available with source code as part of the Linux and Android kernel [37]. When compiling a kernel, the default path to the firmware file to use can be set with the BCMDHD_FW_PATH variable.

The driver includes functionality for everything that the Broadcom chipsets support, so the limitations of the functionality are left to what is covered by the firmware files delivered with the Android system for the specific device.

Besides making source code available for the driver, Broadcom also released the dhdutil program, which can be used to interact directly with the Broadcom chipset using their driver [31], [32]. The dhdutil program supports a range of capabilities, one of which is the “membytes” command that allows for reading selected bytes from memory [31], [32], [38]. This is important for adding functionality to firmware files.

d. Configuring bcmdhd on Android Devices

Some Android devices ship with multiple Broadcom firmware files that implement different functionality. Configuring this functionality, though, is not consistent between Android devices. Configuring the Pixel C as a hotspot is possible through the Android GUI in the Wi-Fi settings menu and requires the firmware that implements access point functionality to be loaded.

The firmware files themselves are located in either /etc/wifi or /vendor/ firmware and are named with the string “bcmdhd” somewhere in the filename and the suffix .”bin” [16], [37]. The Pixel C stores the firmware files in the directory /vendor/firmware. They can be listed using the ls command and the wildcard character ‘*’ from within a shell:

```
dragon:/ # ls -1 /vendor/firmware/*bcmdhd*.bin
/vendor/firmware/fw_bcmdhd.bin
/vendor/firmware/fw_bcmdhd_apsta.bin
```

As we can see, there are two different firmware files, and the file with the string “apsta” within the name is the firmware that includes the functionality for configuring the wireless card as a hotspot for sharing Internet connectivity [31].

The Android kernel supports configuration of various system modules through interaction with files in the /sys/module directory of the device. The firmware file that the Broadcom driver should use is configured by writing the absolute path to the firmware_path entry in its system module configuration path on the Android device. The path to the Broadcom driver firmware_path entry is one of either two locations:

1. /sys/module/bcmdhd/parameters/firmware_path
2. /sys/module/dhd/parameters/firmware_path

Some Android devices still ship with a firmware that implements the independent basic service set (IBSS) functionality without any configuration possible from the Android GUI or even from the command line. When listing the /etc/wifi directory on a Samsung Galaxy S6 (G920-T) running Android 6.0, the following firmware files are found:

```
shell@zerofltetmo:/ $ ls /etc/wifi/*bcmdhd*.bin
/etc/wifi/bcmdhd_apsta.bin
/etc/wifi/bcmdhd_ibss.bin
/etc/wifi/bcmdhd_mfg.bin
/etc/wifi/bcmdhd_sta.bin
```

Among the list of firmware files, there is a file with the “ibss” string in the filename; this is the firmware that implements the IBSS mode, which also can be verified by running the `strings` command and reading the version string that contains the sub-string “aibss” among the functionality tags [31]:

```
4358a3-roml/pcie-ag-p2p-pktctx-aibss-relmcast-
proptxstatus-ampduhostreorder-sr-aoe-pktfilter-keepalive-
```

```
clm_ss_mimo-sarctrl-sstput-xorcsum-fpl2g Version:
7.112.17.19 CRC: 1d3581a2 Date: Thu 2016–03-10 15:36:56 KST
Ucode Ver: 986.11001 FWID: 01-aefde609
```

In order to configure the wireless interface on the Samsung Galaxy S6 for IBSS mode, we need to perform the following as root:

1. Turn off Wi-Fi in the Android GUI to avoid conflict with the Android network manager.
2. Bring the wireless interface down.
3. Write the path of the firmware file to the system module entry.
4. Configure an IPv4 address and network mask for the wireless interface.
5. Bring the wireless interface up.
6. Set the wireless interface in IBSS mode.
7. Join an IBSS network on a selected frequency.

Configuring an IPv4 address can be performed using the `ifconfig` command. Configuring the IBSS mode and joining a network can be executed with the `iw` command, which communicates with wireless drivers using Input Output ConTroL (IOCTL) commands. Unfortunately, the Samsung Galaxy S6 does not include the `iw` command.

Since the Pixel C and the Samsung Galaxy S6 use the same driver and they both run on 64-bit ARM, the `iw` command can be copied from the Pixel C to the /data/local directory on the Samsung S6. From the /data/local directory, the command can be executed and used for configuring the wireless interface. Once Wi-Fi is turned off in the Android GUI, the following commands are executed as root:

```
FWFILE=“/system/etc/wifi/bcmdhd_ibss.bin”
```

```
FWSETPATH=“/sys/module/dhd/parameters/firmware_path”
ifconfig wlan0 down
echo $FWFILE >> $FWSETPATH
ifconfig wlan0 10.0.0.10 netmask 255.255.255.0 up
/sdcard/local/iw wlan0 set type ibss
/sdcard/local/iw wlan0 join adhocnet 2412
```

By configuring a Linux laptop for the same ad-hoc network using the IPv4 address 10.0.0.20, the Samsung device can send ping requests to the Linux machine, as shown in Figure 14.

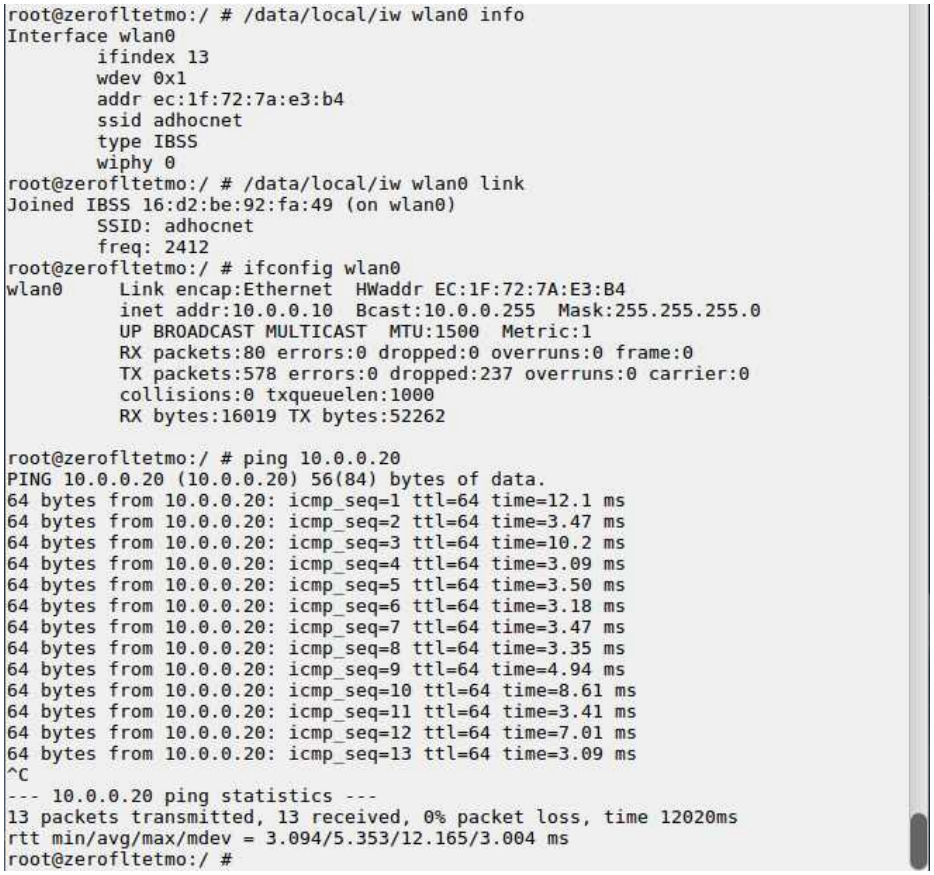

Figure 14. Samsung Galaxy S6 sending ping in ad-hoc mode.

e. Firmware Reverse Engineering and Modification

The NexMon project, by Schulz, Wegemer, and Hollic, provides a C interface for modifying firmware from selected Broadcom chipsets [32], [39]. Schulz, Wegemer, and Hollick started by reverse engineering firmware delivered with the Nexus 5 smartphone running Android 6.0 using the BCM4358 chipset. They used the dhdutil to read the ROM and create an image of the memory together with selected firmware files and then locate functions and critical data structures to get a better understanding of the initial source code that was compiled to generate the firmware [31], [36]. When the memory image was reverse engineered, they developed a framework that can be used to modify selected firmware files.

The NexMon project is active with source code available online [38], and support for different firmware files is constantly growing. The main goal of the project is to extend firmware files with packet monitoring and packet injection, providing both read and write capabilities. Although the number of supported devices and firmware files is somewhat limited at the time of this writing, this is a promising project for the MANET community and wireless development on mobile devices in general. Reading and writing raw IEEE 802.11 frames is the foundation for implementing any wireless protocol.

D. MANET RELATED PREVIOUS WORK

Much research performed on MANETs makes use of the different wireless technologies available on mobile devices. The following sections summarize important efforts in that research.

1. AdHocDroid

The AdHocDroid [40] is an Android application that provides multi-hop communication in an infrastructureless environment on rooted Android phones. These phones run early versions of the Android operating system and still have

the ad-hoc mode available in their wireless drivers. The AdHocDroid developers claim to be the only MANET solution that satisfies their full definition:

- Allows for communication without an Internet connection
- Makes multi-hop communication possible
- Provides support for any network application through a regular socket API without re-write
- Works without needing additional wireless technology

Although the developers claim AdHocDroid is the only true MANET solution, it requires old versions of the Android operating system to run, which quickly makes the application obsolete due to the rapid development of new mobile devices. Surely, there is a requirement of a method to enable ad-hoc mode on future mobile devices to provide a durable MANET solution.

2. Thinktube

The Thinktube project provided patches for some versions of the Android system (4.2.2, 4.3, 4.4, and 5.0) to support ad-hoc mode in the Android GUI with connections to the driver to simplify configuration [41]. The patches were added to early versions of the CyanogenMod project, which was terminated in 2016 [21], and later continued into the LineageOS project in December 2016 [20], which forked with version 14 of CyanogenMod. The patches from Thinktube were submitted to AOSP, but were never accepted [41]. The support for ad-hoc mode in CyanogenMod was removed after version 12 when Android versions after 5.0 were used for the code base [42].

3. FireChat

The FireChat [43], [40] application runs on top of Open Garden [40]. It makes use of Wi-Fi Direct or Bluetooth links to create virtual private network (VPN) connections between devices running the Open Garden application. The FireChat

application was not initially developed with either privacy or security as constraints, which limited the range of possible end-users dramatically. The versions after July 2015 use end-to-end encryption to protect private messages between devices, but it is possible for anyone to join chat rooms in a network and take part in conversations.

4. Wi-Fi Direct Modifications

In order to modify or extend Wi-Fi Direct to support multi-hop communication, root access is required [4], [40]. Funai, Tapparello, and Heinzelman [44] present different modifications of Wi-Fi Direct that require root access to support multi-hop communication. One modification uses a Transmission Control Protocol (TCP)-based time sharing to switch between groups. This causes “flapping” and results in significantly reduced throughput. Another modification makes use of the User Datagram Protocol (UDP) broadcasting between groups. This requires a relay node. To the best of our knowledge, there are no modifications of Wi-Fi Direct that support effective multi-hop communication.

a. Content-Centric Networking

Content-centric networking uses names instead of IP addresses for identifying network hosts. Junh, Ahn, and Ko [42] use Wi-Fi Direct to create content-centric multi-hop communication and connect to multiple groups. The testing was performed only in the NS-3 simulator, which leaves questions about performance using actual physical devices.

Using Wi-Fi Direct with a content-centric solution without root access [45] requires multiple proxies since Android assigns the same IP address to the GO of every group, which clearly shows the ineffectiveness of using Wi-FI Direct for multi-hop communication. Apart from also leaving a specific device as a single point of failure, it suffers from the extreme power consumption of the GO as described previously in this chapter.

5. BLESSED with Opportunistic Beacons

Bluetooth Low Energy profile leveraging IEEE 802.11 Service Set Encoding-based Dissemination (BLESSED) [46] is an alternative to the very limited ad-hoc interface that requires root access on Android devices when it is available [46] and is not natively supported by either iOS or Windows Mobile. BLESSED uses a combination of Bluetooth Low Energy standard connections and Wi-Fi hotspots to create a mesh in an infrastructureless environment [46].

In contrast to FireChat, which also uses a combination of Wi-Fi Direct and BLE links, the authors show that BLESSED is more portable since it uses the Wi-Fi hotspot in contrast to Wi-Fi Direct [46]. The Wi-Fi hotspot is available on Android, iOS, and Windows Mobile devices; Wi-Fi Direct is not available with the latter two operating systems. However, the Wi-Fi hotspot mode is limited to a maximum of ten clients on Android devices and only five on iOS and Windows phones.

Since Bluetooth also limits the number of connections to clients with a maximum of seven on all devices, the connected devices need to alternate between the different connection types to create an ad-hoc network mesh, which increases the power consumption. The biggest constraint related to the BLESSED project, though, is the minimal throughput caused by half-duplex communication in combination with limited data length that limits the implementation to short message applications [46].

6. 802.11s: The WLAN Mesh Standard

The wireless Local Area Network (LAN) mesh standard [47] was developed in 2010. The wireless mesh network, or mesh basic service set (MBSS), can be thought of as an evolution of ad-hoc networks since it specifies how an ad-hoc network can be connected to other types of networks, such as 802.3 (Ethernet) and 802.16 (WiMax). Mesh stations are typically client devices that can also be access points that forward packets to other client devices. A difference between an IBSS and an MBSS is that by using the mesh cloud, the MBSS allows messages to be transferred between devices not in direct communication. The

mesh appears as a single Layer 2 (Ethernet-like) group of devices functioning as interconnected, daisy-chained switches. Thus, clients can connect without making modifications and without knowing that they are connecting to a mesh cloud in which each device may serve as an Ethernet-like switch, only forwarding packets when they are part of the best path between the source host and the destination host.

A mesh network uses the Simultaneous Authentication of Equals (SAE) [48] algorithm for security. This is a simple peer-to-peer protocol for authentication that is resistant to attacks such as eavesdropping, Denning-Sacco, and forward secrecy [48]. The algorithm supports finite field and elliptic curve cryptography [48]. The devices in the mesh dynamically establish a session key. Each link is independently secured [48]. Since there is no end-to-end encryption, a broadcast traffic key must be updated with every new peering [48].

There are numerous challenges with deploying and implementing mesh networks. First of all, the devices use carrier sensing for transmitting frames; therefore, the devices at the edges of the network will have a higher probability of gaining access than the devices in the center of the network, and they also use less power since they will not forward traffic to the same extent as the devices at the center of the network [47]. Furthermore, the need to forward traffic through multiple hops can lead to congestion within the interior of the network. The MBSS network uses a management frame for congestion control. The management frame requests neighbors to slow down, an effect that will ripple back to the source of the congestion [47]. Since signal strengths and quality can vary in wireless networks, link speed changes unpredictably. Apart from that, the MBSS standard does not fully specify conditions that trigger congestion control [47].

Secondly, an MBSS network uses the same wireless channel for all devices, but there are situations when the frequency channel must be changed. An example of this is when a radar is detected in the 5 GHz frequency; the network is then forced to change the channel used. Yet, there is no guarantee that the propagation of the new channel reaches all the connected devices on larger

networks, resulting in the network being partitioned before all devices have received the new channel [47].

A third challenge is that the access points need to be in range of each other since there is no wired backbone to interconnect them [47].

Finally, independent networks make the mesh network suffer since mesh stations do not have any priority over other 802.11 devices in the area [47], which might disturb the radio signals [47] if the same or overlapping frequencies are used by multiple networks. The 802.11s implementation is available on Linux for SoftMAC chipsets through a kernel module. Since the majority of mobile devices today use FullMAC chipsets [31], [32] a firmware that implements 802.11s for the wireless chipset is required to support MBSS networking on those devices.

E. SUMMARY

This chapter provided background information regarding various technologies and research for creating different types of wireless communication networks to be used in an infrastructureless environment. Furthermore, the shortcomings of ad-hoc networks in a selection of research projects were explained. The Android operating system was discussed in detail along with how a custom system can be built from source code and then flashed to a device where root access has been achieved. The Broadcom chipset, which is a common wireless chipset on Android devices, was reviewed for the purpose of using firmware files with different functionality on Android to enable ad-hoc mode as root from the command line.

The key information from this chapter that informs the discussion in subsequent chapters includes the following:

- Some Android devices ship with firmware that implements IBSS mode, which is not configurable using the GUI or the command line.

- If firmware files exist that implement IBSS mode, they can be detected on a device through physical access or by examining the firmware files for that device.
- Ad-hoc mode can be enabled on some Android devices with root access regardless of the Android version.

III. DESIGN AND IMPLEMENTATION

A. ANDROID FIRMWARE SCANNERS

To build a list of Android devices that can be used to create MANET solutions using IBSS mode, we must first distinguish the devices that ship with IBSS-enabled firmware for the Broadcom chipset. By using the various firmware packages for Android available for download on the Internet, we can determine whether a certain device ships with IBSS-enabled firmware for the Broadcom wireless chipset by examining the firmware files. The advantage with this is that physical access to the device itself is not required. To simplify the examination of the firmware files, we developed software that automates this process for different types of packages.

1. Firmware Scanner Design

Although the layout and content of firmware packages might differ, the scanner software uses a similar methodology for all the different types of packages. The functionality is best described with pseudo code as illustrated in Figure 15. The details of the pseudo code are discussed in the following, and we present an outline for the design of the different firmware scanner implementations.

```
function scan_firmware_archive(filename)
{
   # (a) Extract archive
   create directory ./tmp;
   uncompress filename into ./tmp;
   change working directory to ./tmp;

   # (b) Convert firmware to file system
   convert firmware to file system image;
   fsimage = file system image path;

   # (c) Mount file system
   create dir ./mnt;
   mount $fsimage on ./mnt;

   # (d) Search for files
   retval = false;
   pattern = *bcmdhd*;
   files = find files in ./mnt where name=$pattern;

   # (e) Save firmware files
   if count(files) > 0 {

       for (file in files) {
           copy firmware file to storage;

           if firmware file has ibss functionality:
               retval = true;
       }
   }

   # (f) Clean up
   unmount ./mnt;
   change working directory to ..;
   remove directory structure ./tmp;
   return $retval;
}
```

Figure 15. Firmware scanner pseudo code.

a. Extract Archive

The initial task is to extract all files from the package and locate Broadcom firmware files that can be examined for IBSS functionality. The extracted files are stored in a temporary directory, which can be easily deleted afterwards since they contain a great deal of data that is not pertinent to this research.

b. Convert Firmware to File System Image

This step, Step b in the pseudo code in Figure 15, is what will differ most between the scanner implementation of different packages. This is the step where any firmware image inside the package is converted to a file system image that represents a mountable file system. The conversion will vary between packages, but they all result in a mountable file system image. The file system image can then be mounted locally using the Linux mount command for further examination of the content when it is attached to the local file system.

c. Mount File System

The file system image created in the previous step is mounted locally to a temporary directory. This will add a branch of files to the local file system. The files added will appear in the same way as they would when they are mounted on an Android device. Since the branch of files is part of the local file system, we can use common Linux commands and tools to traverse the directory to examine the contents and investigate the files.

d. Search for Files

The common pattern for naming Broadcom firmware files on Android devices, "*bcmdhd*," is used for finding firmware files when traversing the mounted directory. The path for any file with a name that matches the pattern is saved into a list for later processing. Even though the Android operating system in most cases uses the directories /etc/wifi or /vendor/firmware for storing firmware files, the whole directory structure is traversed in case an alternative location is used by customized firmware.

e. Save Firmware Files

If there were files found where the name matched the pattern commonly used when naming Broadcom firmware files, those files are saved in a separate directory. The version string of the potential firmware files is extracted and examined for the tag representing IBSS functionality. If IBSS functionality is found

in the Broadcom firmware, the implementation returns “success” when the scanner implementation finishes, to notify the user that an IBSS enabled firmware was found. Since the files are saved in a separate directory, they can easily be examined further.

f. Clean Up

Since Android firmware packages represent the initial file system on the device that they are intended for, they are often very large. To avoid filling up the disk with unnecessary files after the scanner has finished investigating the contents of a firmware package, all temporary files are deleted. To save even more space, there is also a configurable option in the implementation that removes the initial archive once it has been processed. If there is a need to investigate the firmware package again, it must be downloaded again.

2. OTA Update Package Scanner

As described previously, the Over The Air (OTA) package contains information on how to update an existing Android system rather than completely replace it. This does not imply that all OTA packages contain a mountable image since there might be only a selection of files that should be copied onto a device or, perhaps, a list of files that should be deleted. Most OTA packages, however, contain information on how new files should be added to the system firmware image and how unused ones should be deleted. The `sdat2img` command provides functionality for converting the OTA package update information into an image with a file system that can be mounted locally [49]. The `sdat2img` command is used as the base functionality for Step b in the pseudo code when implementing the OTA scanner.

3. Google Proprietary Drivers Package Scanner

The proprietary binaries used in the Android operating system can be downloaded from Google and extracted into the directory of the downloaded source code when building custom firmware images. These files are located inside

the vendor.img firmware file, which is mounted onto the /vendor directory branch when the Android system is booted. To automate the extraction of vendor.img from the interactive shell script, the offset can be read from the script using the command `grep 'tail -n +'` since the script contains the offset as a shell command to execute extraction from within itself. The vendor.img is in the Android Sparse Image format [23] and needs to be converted to a format recognized by Linux so that it can be mounted locally. The `simg2img` command converts Android Sparse Image format files into a file system image [50] for this purpose and is the base of Step b in the pseudo code when implementing the scanner for Google Proprietary Drivers Packages.

4. Google Factory Firmware Package Scanner

The Google Factory Firmware Package is a zip archive that contains all firmware files required to restore a device to factory mode. The image of interest here is the system.img, which represents the Android system and is stored in the Android Sparse Image format [23]. By using the simg2img command, we can convert the system.img to a file system image that can be mounted locally as described in Step b of the pseudo code.

B. AN EXAMPLE APPLICATION: IBSS CHAT

The following sections describe the design and implementation of the IBSS Chat application.

1. Design Goals

To verify and test the IBSS functionality on devices that have been shipped with the required firmware, herein we design and implement an example application with multi-hop capability for a chat protocol that does not require a server. Although the Android operating system can easily be extended with a reliable multi-hop routing protocol such as OLSR [2], which is an implementation that does not require modification of the kernel since it can be run as a process on the system, we also provide a basic functionality of multi-hop capability to provide

a standalone and adaptable solution independent of the underlying layers. We identify the main features of the application as well as the configuration of the wireless interface and the message exchanging protocol, which will be used to implement the chat functionality and describe the overall design goals for the application in the following sections.

a. Adaptable

The application must be adaptable to various Android devices. This implies an ease of both installation and un-installation on devices where the source code might not be available for building a custom firmware image. Since Android devices do not ship with tools for configuring the wireless interface in IBSS mode, we need to interact directly with the kernel and wireless driver through Input and Output ConTroL (IOCTL) commands when configuring the interface. The interaction with the kernel and wireless driver requires root privileges, which forces us to implement a system service application that must be running as the root user and should start when the system is booted. Since manufacturers make their own modification to the AOSP source code, there are differences among Android devices. These differences impact how system-level software can be added and integrated. Hence, we implement the application as a single binary, which will run as a daemon process when started. This limits the modifications of the system to an added command-line instruction in a startup script, such as the /init.rc on Android from AOSP, or any other script or command that is executed when the system is booted.

b. Portable

A portable application simplifies maintenance of the source code and adaption to other systems in the future. When writing software to UNIX-derived systems such as Android, the C programming language provides full integration with the system and access to standards such as Portable Operating System Interface (POSIX) threads and Berkeley sockets, which are available across modern, commonly-used UNIX-derived operating systems. While implementing

the application, Android specific code can be isolated for substitution of the equivalent code on other systems. An example of this is the Android logging routines that can be made available through the preprocessor definition `__ANDROID__` only when compiling the software for the Android system.

c. Usable

A standalone binary that is started upon boot and run as the root user provides complete access to the system and allows for any kind of modification, including configuration of the wireless interface. Providing full access to the system for users to configure the wireless interface and send chat messages does not comply with the principle of least privilege and would render the Android device unreliable and unsecure, as it would allow for either intentional or unintentional modification of the system. We provide an API for interaction with the service that limits the features to those required and at the same time makes them available for unprivileged users through standard Android applications, which can be developed by third party developers using the API.

d. Testable

A fully functional terminal-based chat client is implemented for interaction with the provided APIs to configure the wireless interface and participate in a chat conversation as a regular user. The source code for the client is written in C, and key parts, such as the protocol implementations, can easily be re-used in a native library when using Android Studio for developing a GUI that provides additional portability. The client also provides sending of randomized messages to simplify testing. Although the client implementation can be available as a separate application, it is included in the service binary for simplicity and activated through command-line options when the binary is executed. This provides a single binary that can easily be installed on systems for testing purposes without the overhead of graphical features, which can be added later through third party applications. As the implementation allows for multiple clients accessing the APIs in parallel, many implementations can co-exist on a single system.

2. Wireless Interface Configuration

The wireless interface configuration is the core functionality of the service and provides an API over TCP on the local host address for interaction with local users through any network-enabled software. The pseudo code of the process is illustrated in Figure 16 and discussed in the following sections.

```
# The configuration service process
function config_daemon()
{
   socket = create_server_socket();

   # Accept connecting client s
   while (client = accep_client($socket))
       spawn_thread(handle_client, $client)
}

# Handle configuration client
function handle_client($client)
{
   # Read client request
   req = read_client_request();

   # Set wireless interface configuration
   if ($req == SET_CONFIGURATION) {
       response = set_wireless_config($req);
       if ($response == OK)
           chat_process_restart()
   }

   # Get wireless interface settings
   if ($req == GET_CONFIGURATION)
       response = get_wireless_values();

   # Send response to client
   send_client($response);

   close_socket($client);
   exit_thread();
}
```

Figure 16. Pseudo code of the wireless interface configuration process.

Our API protocol for configuring the wireless interface is simple and allows future extensions. The client connects and sends a single request to which the server responds, and then the connection is closed. Although the implementation provides access for multiple clients in parallel through threads that are spawned to

handle each connecting client, care has been taken to use appropriate locks of memory regions and execution of specified configurations to avoid overlapping of configuration values between clients.

The client can send two different requests:

1. Request configuration of the wireless interface
2. Request status of the wireless interface

The client initially sends a one-byte request code that is followed by a data structure for that specific request, and the server responds in the same way (this leaves 254 command codes for future additions). When the client requests configuration of the wireless interface, the data structure following the command code contains all the settings for the wireless interface, as shown in Figure 17.

```
/* Wireless interface information */
struct wiface {
   /* The name of the wireless interface */
   char iface[IFNAMSIZ];
   /* Interface IPv4 address */
   uint32_t ipv4;
   /* Interface network mask */
   uint32_t mask;
   /* Wireless mode (IW_MODE_ADHOC) */
   uint8_t mode;
   /* Wireless channel */
   uint8_t channel;
   /* The hardware address of the access point */
   uint8_t mac_bssid[6];
   /* The name of the ad-hoc network */
   char essid[IW_ESSID_MAX_SIZE];
   /* The encryption key for the chat protocol */
   uint8_t key[CRYPTO_KEY_MAXLEN+1];
} __attribute__((packed));
```

Figure 17. Wireless interface configuration structure.

If the configuration of the wireless interface is successful, the server responds with the single-byte response code representing OK without any following data. When a status of the wireless interface is requested, the client sends only a single-byte with the command code for the status request and the

server responds with the command code for the wireless structure in Figure 17, which contains the current configuration of the interface. The server can also respond with an error. The response code for an error is followed by a string that describes the error further. When the wireless interface is successfully configured for IBSS mode, the configuration process starts a child process for handling messages for the chat protocol, which is described in the next section. If the wireless card is reconfigured, the chat process is started again to reflect the configuration changes.

3. Serverless Chat with Multi-Hop Capability

Although our example application implements a serverless chat, the underlying message exchanging protocol can be used for arbitrary data, as we see in the following sections. This provides a solution that can be extended and customized further for specific needs. We use Internet Protocol (IP) Multicast broadcasting of User Datagram Protocol (UDP) packets for transmitting messages among the participants. The use of broadcast messages implements a best-effort solution that does not guarantee delivery of messages; we do, however, improve the probability for delivery of messages by both acknowledgment and resending as part of the multi-hop capability. As we show when testing the implementation in the next chapter, this improves delivery of messages considerably.

a. Message Format

The total size of chat messages is fixed at 100 bytes, but that can easily be changed at compile time, if necessary. Nonetheless, it should be limited to the Maximum Transmission Unit (MTU) of the network to avoid fragmentation of packets. Although the fixed size creates an overhead for smaller messages, it is beneficial for privacy as all encrypted messages are of the same size and do not reveal the length of a specific message. Each message starts with a byte containing the message code that identifies the message type in the data that follows the header. The message code is then followed by the IPv4 address of the original sender, which is used to provide multi-hop capability as we will see later.

Each message has a 64-bit identifier (ID) that together with the IPv4 address of the sender gives a unique ID for a specific message. The message header is displayed as a data structure in Figure 18.

```
struct msghdr {
   uint8_t code; /* Message code */
   uint32_t ip;  /* Originator IPv4 address */

   /* Message identifier */
   struct {
       uint32_t sec; /* Seconds */
       uint16_t us;  /* Lower 16 bits of micro seconds */
       uint16_t sum; /* Message data checksum */
   } id;
} __attribute__((packed));
```

Figure 18. Message header as a data structure.

For the purpose of our example implementation, we only use two message codes to identify discovery and chat messages. This leaves 254 additional message codes that can be used in future extensions to describe customized layouts of the data following the header. An example of this would be adding a code for a private message where the data contains an additional header with the destination IPv4 address and a message encrypted with the receiver's public key.

The unique message ID is used to count how many times a message is seen by a device and is required for retransmitting and multi-hop capability. Although the message ID could be any random number generated by a device, we want to avoid the possibility of reusing previous numbers and generate the ID based on time and a checksum of the data to limit the risk of reproducing a previous ID number. The first 32 bits contain the number of seconds since 1 January 1970, also called Epoch time, when the message is generated. The following 16 bits are masked out as the lower 16 bits of the number of microseconds when the message was generated, and the final 16 bits contain the two's-complement checksum of the message data. This creates a unique identifier that limits the risk of reusing

any previously used ID number if the device is restarted or the IPv4 address is reused by another device.

b. Joining the Chat

A client that wants to join the chat creates and sends a discovery message. The created discovery message is used for the duration of the chat process and the identifier of the discovery message informs other devices if it has been seen before. When a client receives a discovery message, it responds with the discovery message that was generated at startup to inform the joining client about currently active clients on the network and any other clients that might have missed a previous discovery message of its presence. The joining client then waits for discovery messages from other clients for two seconds before it attempts to synchronize with the client that responded the fastest.

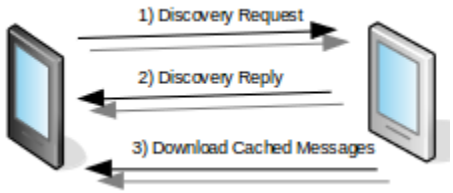

Figure 19. Client joining chat and synchronization with an existing client.

The synchronization is implemented as a TCP connection for reliable transfer of all the previous messages buffered by a client. The buffer size of a client can be changed at compile time, but is set to 1,000 messages in our example application. When the message buffer is full, the oldest message is deleted to create space for a new message.

c. Sending a Chat Message

Figure 20. No acknowledgement received when trying to send a message results in failure.

Chat messages are sent by sending a User Datagram Protocol (UDP) packet to the IP multicast address and port used by the IBSS Chat application. For a message to be considered as sent, it must be acknowledged by at least one other device. The acknowledgment is identified when the message sent is received back with a source address of the UDP packet that is different from the IPv4 address in the message itself, indicating a retransmission from another device. A message will be resent ten times before a client gives up and returns an error.

During retransmissions, the delay is increased differently, depending on the number of previous attempts, as shown in the source code in Figure 21. The purpose is to give other clients time to reply if the network suffers from congestion. There are other approaches for calculating the delay such as increasing with a small random number instead of multiplying with two. The tests performed in the next chapter show that doubling of the delay works very well for our test environment, but this might not be the case for other networks with more or different kinds of devices.

```
/* Wait for response and increase time
 * based on the number of re-sends */
usec = retry * 100000; /* 100 milliseconds */
if (retry > 3)
    usec  *= 2;
```

Figure 21. Source code for calculating delay between resending of messages when waiting for acknowledgment.

d. Multi-Hop Capability

Multi-hop capability is implemented through retransmission of received messages. When a client receives a message that has the same IPv4 address as the IPv4 address of the sender in the UDP packet, it is always retransmitted since it indicates a transmission from the original source that requires an acknowledgement. When the IPv4 addresses do not match, the probability of retransmission is based on the number of times that message has been received. A message can only be retransmitted the first ten times it is received to avoid an infinite retransmission of messages, and the probability is calculated with p = 1 / r where p is the probability of a message to be retransmitted and r is the number of times the message has been received, for r less than 11.

e. Privacy

The chat protocol is independent of the underlying communication layers, which could be using strong cryptography such as VPN tunnels for securing the

communication. Regardless of the underlying communication, the chat protocol provides privacy through symmetric encryption of all messages transmitted on the ad-hoc network. This allows for a chat group to exist on a larger network where all network clients may not be part of the chatroom. As shown earlier, the key used for encryption is supplied when configuring the wireless interface using the command

```
ibsschat --conf
```

f. The Chat Process

The chat process started by the service is multi-threaded and is responsible for both sending and receiving messages. The chat process consists of three major threads, as follows.

(1) Accept Connecting Clients that Want to Receive Messages

This thread listens for connecting clients on a TCP port. When a client has connected, a new thread is spawned to handle the client to allow for multiple clients in parallel. The new thread then sends all messages in the buffer to the connected client for synchronization. If the client is connected locally, that is, if the network portion of the client IPv4 address matches the network portion of the IPv4 address of the listening TCP port (e.g., the leading 24 bits of the IPv4 address), the socket is registered as a receiver of future messages in the message buffer. This allows for local clients to “block on read” from the socket to receive new messages when they arrive without constantly polling for new messages and consuming unnecessary power.

(2) Accept Connecting Clients that Want to Send Messages

Similar to the first thread, this thread listens on another TCP port and allows for multiple clients to be handled in parallel by spawning a new thread for each connected client. The thread that handles a client reads a single message structure from a client that should be sent to the chat group. The thread then responds with a return value to the client reporting whether the message was sent successfully before closing the connection.

(3) Multicast Reader

The last thread in the chat process is the thread responsible for receiving and resending messages using IP multicast. It uses a message buffer implementation that allows for counting how many times a specific message has been received and registers socket descriptors as recipients for new messages when they arrive. This thread performs the main work of the chat process and implements any resending of messages required for multi-hop capability.

g. Starting the Service

The service can be started for a specified wireless interface from the command line as root with two different options to the `ibsschat` command:

```
–-daemon <iface>

--daemon-nofork <iface>
```

The first option creates a daemon process, which is a process that runs as a service without a controlling terminal. This is accomplished by executing the `fork()` system call twice and exiting the first process created. The second option also starts the service, but does not create a daemon process, which leaves the process attached to the terminal from which it was started; this is convenient for debugging since error messages are written to the standard error output of the terminal where `ibsschat` was started.

h. Integrated Client

We integrated a client implementation in the `ibsschat` command and made it available through a set of command line options:

```
–-conf <iface> <ipv4> <mask> <network-name> <chan> <key>

--status <interface>

--chat-send <iface> <message>

--chat-send-rand <iface> <count> <delay-sec>
```

`--chat-prompt <iface>`

The `--conf` option configures the wireless interface into IBSS mode with the supplied values and the `--status` option displays the current settings of the interface. The `--chat-send` option sends a single chat message.

The `--chat-send-rand` option is required only for testing the protocol as it randomizes test messages up to 40 characters long. The number of messages to generate is specified with the `count` argument and the delay in seconds between messages is specified with the `delay-sec` argument. A delay of zero makes the client send the messages as quickly as possible for stress testing the application.

The last option is the `--chat-prompt` command line option; this enables a multi-threaded chat client that reads messages to send from standard input and write received messages on standard output, creating a fully functional chat client. The chat prompt client can display the number of received messages and the IPv4 addresses of other chat clients on the network if '`.stat`' is entered. This is useful when testing the protocol, as we will see in the next chapter.

C. SUMMARY

In this chapter, we presented the design and implementation of two different types of software applications. The first is a suite of programs for examining different Android firmware packages and searching for IBSS-enabled Broadcom firmware. For the second, we explained the design and implementation of an application that runs as a system service and provides an API for configuring the wireless interface as well as an API for a serverless, multi-hop-capable chat for ad-hoc networks. We also presented a client that makes use of the APIs to configure the interface and participate in the chat. In the client implementation, we integrated functionality for randomizing messages to simplify testing. The next chapter provides the test methodology and results for these software packages.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. TESTING AND EVALUATION

A. SCANNING ANDROID FIRMWARE PACKAGES

The following sections discuss the execution of the firmware scanner applications implemented for different types of firmware packages.

1. Downloading and Processing Firmware Packages

Firmware packages for the Android operating system were downloaded from Google as proprietary drivers [16] and factory images [51], and were examined by the two different firmware scanner programs developed for that purpose. The LineageOS project provides OTA packages for installation on a large number of devices [20]. All available packages were downloaded from the LineageOS project [20] and examined with the OTA firmware scanner implementation. In total, 217 firmware packages representing 203 different devices were downloaded and processed. When a firmware package was processed for a specific device, that device was put into one of three groups according to the triage model:

- Device with IBSS-enabled Broadcom firmware
- Device with Broadcom firmware
- No Broadcom firmware found

2. Evaluation

Scanning firmware files is a reliable way to search for IBSS-enabled firmware. However, absence of a firmware with the IBSS functionality does not preclude that such a firmware exists for that device since there might be some other firmware package containing a Broadcom firmware that implements the IBSS functionality. The devices in the three different groups are listed in Appendices A, B, and C, respectively. The number of devices found with a Broadcom firmware was 70, which is 34 percent of the total number of different devices investigated in

this research. Out of those 70, there were 28 devices shipped with a Broadcom firmware that implemented the IBSS functionality. By examining the devices in the first group, we can see that there are five different chipsets used by the 28 different devices that ship with IBSS functionality, which is illustrated in Table 2.

Table 2. Chipset with IBSS firmware.

Broadcom chipset	Number of different devices
bcm4354	11
bcm4339	6
bcm4358	6
bcm4335	3
bcm4359	2

As we can see from Table 2, the Broadcom chipset bcm4354 is the most common chipset shipped with a firmware that implements the IBSS functionality. By comparing the tables in Appendix A and Appendix B, we also see that all of the chipsets with IBSS firmware, except for bcm4359, are represented among the devices that do not ship with firmware implementing IBSS functionality. This is interesting and promising for continued research that leverages the NexMon project [32] for adding IBSS functionality to firmware files.

The differences found by reading and comparing the ROM of the devices in the two different groups can aid in understanding what is required to modify and move IBSS-enabled firmware files across devices using the same Broadcom wireless chipset. A good selection of devices for this purpose would be the Google Nexus 6P and Google Nexus 5, which use chipsets bcm4339 and bcm4358, respectively, and are part of the NexMon project. Hence, the ROMs for those chipsets have been reverse-engineered already and provide an initial understanding of the layout of source code that was used to compile the firmware, and that can be leveraged to research how to add IBSS functionality.

Furthermore, we can see from the tables that 96 percent of the devices with any Broadcom firmware are supported by the LineageOS project [20] version 14.1, which is based on Android 7.1.2 (Nougat) from AOSP. The next release of LineageOS is under development and will be based on Android 8.0 (Oreo) [20]. Clearly, LineageOS provides an optimal platform for further customization of a majority of Android devices since source code is provided that allows for building the system similar to when using the AOSP source code, as discussed in Chapter II.

B. IBSS CHAT APPLICATION

The following sections discuss the execution of the IBSS Chat application for the purpose of testing the implementation.

1. Testing

During the development of the IBSS Chat application, the software was compiled on 64-bit Linux (Ubuntu 16.04.3 LTS) and unit-tested to remove major bugs as well as resource and memory leaks. This resulted in source code that compiles under Linux but has not been fully tested as an application. The final Android application was cross-compiled using the Android Native Development Kit (NDK) version r11c, on the same Linux machine.

One of the most time-consuming problems encountered during development was connecting Android devices to the same ad-hoc network when configuring the wireless interface. The first device in an ad-hoc network uses a random hardware address, called the Basic Service Set Identifier (BSSID), for the IBSS network. Other devices that join the network with the same name should adapt and use the same address. This was not the case for us, as devices sometimes created a different BSSID, which resulted in multiple networks with the same name where the devices in the different networks could not communicate with each other. We solved this by manually setting the BSSID for the IBSS network when configuring the wireless interface. We base the BSSID on a (randomly generated) template and use the exclusive-or operation with the network name to create a BSSID, as shown in the source code in Figure 22.

```
/*
 * Create a BSSID based on the string by XOR
 * BSSID must be 6 bytes.
 */
static void
mkbssid(uint8_t *bssid, char *str)
{
   size_t len;
   size_t i;

   /* Set default value */
   memcpy(bssid, "\x8a\x64\x2d\x92\x93\x69", 6);

   /* XOR the string (network name) */
   len = strlen(str);
   for (i=0; i < len; i++) {
       bssid[i%6] = (uint8_t)bssid[i%6] ^ (uint8_t)str[i];
   }

   /* Use a fixed value for first byte */
   bssid[0] = 0x8a;
}
```

Figure 22. Function for creating a BSSID.

a. Environment

The testing of the Android application was performed using the six Samsung devices listed in Table 3.

Table 3. Samsung devices used when testing.

ID	Device Name	OS	Chipset
1	Galaxy S6	Android 5.1.1 (stock)	bcm4358
2	Galaxy S5	LineageOS 14.1 (Android 7.1.2)	bcm4354
3	Galaxy Note 4	Android 6.0.1 (stock)	bcm4358
4	Galaxy S6	Android 6.0.1 (stock)	bcm4358
5	Galaxy Note 4	Android 6.0.1 (stock)	bcm4358
6	Galaxy Note 4	Android 6.0.1 (stock)	bcm4358

The `ibsschat` binary was copied to the /system/bin/ folder on all of the devices and the service was started as root by executing the command:

```
/system/bin/ibsschat –-daemon
```

Each device was then configured using the Terminal Emulator application by executing the following command as a regular user:

```
ibsschat --conf wlan0 10.0.0.X 255.255.2550 myadhoc 11 cryptokey
```

where x was substituted for the ID of the device. When all devices were configured, they were placed on a table about an inch apart in the order of the ID number from left to right.

b. Sending Test Messages

For the purpose of testing the protocol, we used two different versions of the IBSS Chat implementation. One of them was the initial release of the original software implemented as designed in Chapter III. The other was a modified version that defined `DONT_WAIT_FOR_ACK` when compiling the binary to disable the requirement of acknowledgment when sending messages. By removing the required acknowledgment in one binary, we can compare the results and measure how much more reliable delivery of messages is when requiring an acknowledgment from at least one device. This allowed us to test the protocol itself rather than the wireless medium where the devices are used, since the medium can be affected by numerous conditions as discussed in Chapter II.

c. No-Hop Stress Test

The no-hop test was performed when all the devices were in range of each other so there was no need for multi-hop communication to reach another device. This is the default condition when devices are placed on a table as described previously. Since all devices can talk directly to all other devices, we measure the increased reliability of delivery when adding acknowledgment to messages under close to optimal conditions.

The test was divided into three phases. In each phase all devices sent 150 random messages at the same time and we measured how many messages were successfully received on each device. The three different phases for the test used a

delay of zero, one, and two seconds between messages, respectively. We first ran each phase with the original version of the IBSS Chat software and then with the modified version. Since the message buffer in the IBSS Chat software is set to 1,000 messages, each device was expected to receive a total of 906 messages after the test. The extra six messages were for the discovery messages when the devices joined the chat. The results for the test with the original version of IBSS Chat are available in Table 4, and the results for the modified version are listed in Table 5.

Table 4. No-hop test with message acknowledgment.

Device	**0 sec**	**1 sec**	**2 sec**
1	906	906	906
2	906	906	906
3	906	906	906
4	906	906	906
5	906	906	906
6	906	906	906
Average	906	906	906
Average %	**100%**	**100%**	**100%**

Table 5. No-hop test without message acknowledgment.

Device	**0 sec**	**1 sec**	**2 sec**
1	300	814	885
2	319	795	884
3	329	818	891
4	304	801	888
5	331	811	884
6	365	818	891
Average	324.7	809.5	887.2
Average %	**35.84%**	**89.35%**	**97.93%**

d. Multi-Hop Stress Test

To test the multi-hop capability of the protocol, we configured the devices using the `iptables` firewall in Android to receive and send network traffic only to adjacent devices according to the ID of the device. Device 1 could communicate only with Device 2, and Device 2 could communicate only with Device 1 and Device 3, etc. This configuration simulated a worst-case scenario where the edge devices are required to transmit a message through all other devices to reach the device on the other end. This configuration is illustrated in Figure 23.

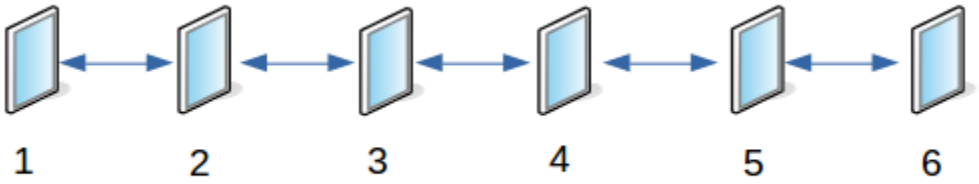

Figure 23. Devices configured for multi-hop communication test.

When the devices were configured for multi-hop communication, we started the IBSS Chat service and configured the devices as noted. We then ran the same tests as in the no-hop test. The results are shown in Table 6 and Table 7.

Table 6. Multi-hop test with message acknowledgment.

Device	0 sec	1 sec	2 sec
1	856	866	887
2	866	880	896
3	877	882	903
4	854	873	899
5	851	874	894
6	840	874	888
Average	857.3	874.8	894.5
Average %	**94.62%**	**96.56%**	**98.74%**

Table 7. Multi-hop test without message acknowledgment.

Device	0 sec	1 sec	2 sec
1	332	801	836
2	321	800	840
3	279	832	847
4	260	834	843
5	268	828	842
6	353	827	845
Average	302.2	820.3	842.2
Average %	**33.36%**	**90.54%**	**92.96%**

2. Evaluation

For the no-hop test, we can see in Table 4 that all messages were delivered successfully on all devices when using acknowledgment compared to only 35.84 percent when it was disabled. The data in Table 5 show that the average number of messages successfully delivered increases with the number of seconds of delay between messages. This is expected as a delay allows for less congestion when

multiple clients attempt to broadcast messages. Although messages are not required to be acknowledged by another device in the modified version of the IBSS Chat software, they are still retransmitted by other devices, which increases the probability of delivery of a single message and explains why the number of successfully transmitted messages increases dramatically when adding just a single second of delay. We also note that the number of successfully delivered messages is somewhat equal across all devices, which also is expected since they share the same medium and can send messages directly to each other without the requirement of multi-hopping.

The multi-hop simulation test caused packet loss even when acknowledgment was required. Since messages must be resent by other devices to reach all the edges of the network, the probability of successfully sending a message from one edge device to another decreases with the number of hops as each device forwarding a message does not require acknowledgment of a message when retransmitting.

Comparing the results of the two tests, we can see how much the acknowledgment of messages actually increases the probability of delivery. When sending messages as fast as we can (zero seconds delay), only 33.36 percent of the messages were successfully transmitted when we did not require an acknowledgment, in comparison to 94.62 percent when we did require acknowledgment. That is an increase of 61.26 percent, clearly showing dramatic improvement in delivery. However, these tests were only performed using six devices and more tests with a larger number of devices, and a less “linear” dispersion of devices, should be performed for a better understanding of the limitations of the protocol. One could suspect that the probability used for deciding a retransmission in the current implementation is suitable for six devices but might be a limitation for a larger number of devices since it is fixed. Also, integrating the devices using the chat protocol into a larger ad-hoc network with other traffic might reveal how the broadcasted messages can co-exist with other solutions.

a. Device Maintenance

Since root privileges are required to install any system service or perform customization of the Android operating system, the device will be unlocked for future modifications by someone with both physical access and user-level access to the device. To preserve the integrity of the system, the current state after modification should be saved to allow tracking of unwanted changes. This can be performed by computing cryptographic hashes of the flashed images as well as on all of the system files. With the hashes stored offline, they can be used to compare the state of the device reliably when it has been physically returned for maintenance since altering of the hashes themselves is not possible. It is desirable, however, to detect unwanted system modifications as early as possible. Although anything could be altered on a device, including added security features, additional software, such as Tripwire [52], can be installed (and perhaps even customized further since source code is available) to help identify changes at an early stage and alert the organization about unwanted changes using a secure channel over the network.

C. SUMMARY

In this chapter, we described test and evaluation of the software that we designed and implemented in Chapter III. First, we scanned firmware files to produce a list of 28 devices that ship with a Broadcom firmware version supporting IBSS mode as a selection list for our IBSS Chat application. We also produced a list of devices with Broadcom chipsets that may be used in future research on how to move firmware files across devices or add IBSS functionality to devices that do not have it organically. We then stress tested our IBSS Chat implementation that was installed on six Samsung devices. The results of the stress tests showed that our requirement for message acknowledgment from at least one device when sending a message improves delivery significantly.

V. CONCLUSION AND FUTURE WORK

A. CONCLUSION

The purpose of this research was to extend the Android operating system to provide a reliable way to support multi-hop communication in ad-hoc networks with degraded or no network infrastructure. We found that although the Android operating system no longer includes tools for configuring ad-hoc mode on the wireless interface, there are Android devices, regardless of the version of the operating system, that ship with wireless firmware for the Broadcom chipset that implement ad-hoc functionality. The Android devices that ship with ad-hoc-network-enabled firmware can be found by investigating firmware files without requiring access to a physical device. We designed and implemented software for scanning through different types of firmware files and produced a list of devices with various versions of the Android operating system that can be used for enabling ad-hoc mode on the wireless interface.

Even those Android devices which do ship with the ad-hoc mode firmware, do not include the tools required to configure the wireless interface in the ad-hoc mode. We overcame this limitation by implementing a system service that communicates directly with the wireless driver to change the firmware file used by the wireless Broadcom chipset and configured the interface manually. The system service is implemented as a standalone binary that can be easily installed, and uninstalled, on an Android device. This step of installing the service requires root access on the device. Once installed, the service provides an API for regular users to configure the interface without requiring root access. We also designed and implemented a server-less, multi-hop-capable, chat protocol that is integrated into the system service. This protocol provides a functionality that leverages our findings and can be used in an infrastructure-less environment without the requirement for any server, which could be a single point of failure. As our system service is designed and implemented to support portability, it uses standards such as POSIX threads and isolates Android-specific parts to simplify future

development and usage on later Android devices, as well as on operating systems other than Android. The key contributions of our research can be summarized as follows:

1. A list of Android devices that can be used in wireless ad-hoc mode
2. A portable and adaptable Android system service that provides an API for users to enable configuration of the wireless interface without requiring root access
3. The design and implementation of a serverless, multi-hop-capable, chat protocol

B. FUTURE WORK

The following sections discuss especially thoughtful, detailed suggestions for future research.

1. How Can IBSS-Enabled Broadcom Firmware Be Ported Across Devices?

The list of Android devices shipped with firmware for the Broadcom chipset that implements IBSS functionality is currently limited to Samsung. As the list of devices that use the Broadcom chipset is quite large and includes devices from many different manufacturers, it raises the question as to how IBSS-enabled firmware files can be ported across disparate devices. Initially, this inquiry would be focused on devices that use the same chipset, such as the Google Pixel C and Samsung Galaxy S5 phones, both of which use the bcm4354 chipset, but only the S5 ships with an IBSS-enabled firmware. Yet, as discussed in Chapter II, there are many similarities across the different Broadcom chipsets, which suggests that it might also be worth researching how to move IBSS-enabled firmware across different chipsets.

2. How Can IBSS Functionality Be Added to Existing Broadcom Wireless Chipset Firmware Files?

The NexMon project [38] provides a framework in C for modifying Broadcom firmware files and provides modifications of existing firmware files for a selection of devices that implement both sending and receiving of raw 802.11 frames. This can be leveraged to implement any protocol, including ad-hoc mode, as sending and receiving raw 802.11 frames is the foundation for implementing any 802.11 protocol. Although implementing and porting ad-hoc mode to a NexMon-modified firmware would be challenging, it also raises the question as to how the NexMon framework can be used to move IBSS-enabled firmware across devices by simply detecting and moving selected parts of the code inside firmware files that already implement the IBSS functionality.

3. What Firmware Files from Other Manufacturers Implement IBSS Functionality That Can Be Used on Android Devices to Enable Ad-hoc Mode?

This research focuses on Broadcom chipsets and raises the question as to what other wireless chipsets that ship IBSS-enabled firmware can be used to enable ad-hoc mode on Android devices. The firmware scanning software produced a list of devices on which a Broadcom firmware was not found; this list could be the initial list of devices to research firmware files that are available for the wireless chipset used on those devices.

4. Chat Protocol Development

We used a fixed delay when sending randomized messages for testing our protocol. Because of random delays that are introduced during retransmission in the physical layer and time between system calls as well as different performance of the devices used, we avoided sending messages at the same time from multiple devices. Apart from that, the tests were started manually on the devices with about a second delay in between, which prevented synchronized sending of messages that could cause periodic packet drops.

The protocol needs to be tested further on larger networks with perhaps other types of traffic as well to better understand how the protocol would function in real-world scenarios and how it can be tuned automatically for maximum performance. The testing would include randomized delays between sending test messages and injecting errors such as mal-formatted messages into the network.

A possible addition to a future release of the chat protocol would be geographical awareness for improved routing. Since the current implementation includes knowledge about the adjacent network neighbors through messages with the same source address as the original source address, this could perhaps be leveraged to create a topology graph that can be used to improve routing of messages.

APPENDIX A. ANDROID DEVICES WITH IBSS-ENABLED BROADCOM FIRMWARE

Table 8. Android devices with IBSS-enabled Broadcom firmware.

Device	Wi-Fi Chip	LineageOS
Samsung Galaxy Tab S 10.5 (chagallwifi)	bcm4354	Yes (14.1)
Samsung Galaxy Tab S2 9.7 (gts210ltexx)	bcm4358	Yes (14.1)
Samsung Galaxy Tab S2 9.7 (gts210wifi)	bcm4358	Yes (14.1)
Samsung Galaxy Note 3 (ha3g)	bcm4339	Yes (14.1)
Samsung Galaxy S7 Edge (hero2lte)	bcm4359	Yes (14.1)
Samsung Galaxy S7 (herolte)	bcm4359	Yes (14.1)
Samsung Galaxy Note 3 LTE (hlte)	bcm4339	Yes (14.1)
Samsung Galaxy Note 3 LTE (hltetmo)	bcm4339	Yes (14.1)
Samsung Galaxy S4 (jfltevzw)	bcm4335	Yes (14.1)
Samsung Galaxy S4 (jfltexx)	bcm4335	Yes (14.1)
Samsung Galaxy S5 (k3gxx)	bcm4354	Yes (14.1)
Samsung Galaxy Tab S 8.4 (klimtwifi)	bcm4354	Yes (14.1)
Samsung Galaxy S5 LTE (klte)	bcm4354	Yes (14.1)
Samsung Galaxy S5 LTE (kltechn)	bcm4354	Yes (14.1)
Samsung Galaxy S5 LTE Duos (kltechnduo)	bcm4354	Yes (14.1)
Samsung Galaxy S5 LTE Duos (klteduos)	bcm4354	Yes (14.1)
Samsung Galaxy S5 LTE (kltedv)	bcm4354	Yes (14.1)
Samsung Galaxy S5 LTE (kltekdi)	bcm4354	Yes (14.1)
Samsung Galaxy S5 LTE (kltekor)	bcm4354	Yes (14.1)
Samsung Galaxy S4 LTE-A (ks01lte)	bcm4335	Yes (14.1)
Samsung Galaxy Note 10.1 2014 (lt03lte)	bcm4339	Yes (14.1)
Samsung Galaxy Note 10.1 2014 (n1awifi)	bcm4339	Yes (14.1)
Samsung Galaxy Note Pro 12.1 (v1awifi)	bcm4339	Yes (14.1)

Device	Wi-Fi Chip	LineageOS
Samsung Galaxy S6 (zeroflte)	bcm4358	No
Samsung Galaxy S6 Edge (zeroflte)	bcm4358	No
Samsung Galaxy Note 4 (trelte)	bcm4358	No
Samsung Galaxy S5 (k3g)	bcm4354	Yes (14.1)
Samsung Galaxy S5 (klteatt)	bcm4354	Yes (14.1)

APPENDIX B. ANDROID DEVICES WITH BROADCOM FIRMWARE

Table 9. Android devices with Broadcom firmware.

Device	Wi-Fi Chip	LineageOS
Asus Zenfone 2 (Z008)	bcm4339	Yes (14.1)
Asus Zonfone 2 (Z00A)	bcm4339	Yes (14.1)
Asus Zenfone 2 (Z00D)	bcm43430	Yes (14.1)
Google Nexus 6P (angler)	bcm4358	Yes (14.1)
Google Pixel C (dragon)	bcm4354	Yes (14.1)
Google Nexus 10 (manta)	bcm4324	Yes (14.1)
Google Nexus 6 (shamu)	bcm4356	Yes (14.1)
Google Galaxy Nexus (VZW)	bcm4330	Yes (14.1)
Google Galaxy Nexus (Sprint)	bcm4330	Yes (14.1)
LG G2 (d800)	bcm4335	Yes (14.1)
LG G2 (d801)	bcm4335	Yes (14.1)
LG G2 (d802)	bcm4335	Yes (14.1)
LG G2 (d803)	bcm4335	Yes (14.1)
LG G3 (d851)	bcm4339	Yes (14.1)
LG G3 (f400)	bcm4339	Yes (14.1)
LG G4 (h811)	bcm4339	Yes (14.1)
LG G4 (h815)	bcm4339	Yes (14.1)
LG G5 (h830)	bcm4358	Yes (14.1)
LG G5 (h850)	bcm43455	Yes (14.1)
LG G6 (h870)	bcm43455	Yes (14.1)
LG V20 (h910)	bcm4358	Yes (14.1)
LG V20 (h918)	bcm4358	Yes (14.1)
LG V20 (ls997)	bcm4358	Yes (14.1)
LG G3 S (jag3gds)	bcm4334	Yes (14.1)
LG G3 Beat (jagnm)	bcm4334	Yes (14.1)

Device	Wi-Fi Chip	LineageOS
LG G3 (ls990)	bcm4339	Yes (14.1)
Samsung Galaxy S III (d2att)	bcm4334	Yes (14.1)
Samsung Galaxy S III (d2spr)	bcm4334	Yes (14.1)
Samsung Galaxy S III (d2tmo)	bcm4334	Yes (14.1)
Samsung Galaxy S III (d2vzw)	bcm4334	Yes (14.1)
Samsung Galaxy Tab 2 7.0 / 10.1 (espresso3g)	bcm4330	Yes (14.1)
Samsung Galaxy Tab 2 7.0 / 10.1 (espressowifi)	bcm4330	Yes (14.1)
Samsung Galaxy S III (i9300)	bcm4334	Yes (14.1)
Samsung Galaxy S III (i9305)	bcm4334	Yes (14.1)
Samsung Galaxy Note 8 (n5100)	bcm4334	Yes (14.1)
Samsung Galaxy Note 8 (n5110)	bcm4334	Yes (14.1)
Samsung Galaxy Note 8 (n5120)	bcm4334	Yes (14.1)
Samsung Galaxy Note II (t0lte)	bcm4334	Yes (14.1)
Samsung Galaxy Note II (t0ltekor)	bcm4334	Yes (14.1)
Sony Xperia Z3+ (ivy)	bcm4356	Yes (14.1)
Sony Xperia Z4 Tablet LTE (karin)	bcm4356	Yes (14.1)
Sony Xperia Z4 Tablet WiFi (karin_windy)	bcm4356	Yes (14.1)
Sony Xperia Z5 Compact (suzuran)	bcm43455	Yes (14.1)

APPENDIX C. ANDROID DEVICES WITHOUT BROADCOM FIRMWARE

Table 10. Android devices without Broadcom firmware.

Device Name
Asus Zenfone 2 (1080p)
Asus Zenfone 2 (720p)
Asus Zenfone 2 Laser (720p)
Asus Zenfone 2 Laser/Selfie (1080p)
Asus Zenfone 2 (Z00D)
Asus ZenPad 8.0 (Z380KL)
BQ Aquaris E5 4G (vegetalte)
BQ Aquaris M5 (piccolo)
BQ Aquaris U Plus (tenshi)
BQ Aquaris X5 (paella)
BQ Aquaris X5 Plus (gohan)
Fairphone 2 (FP2)
Google Android One 2nd gen (seed)
Google Droid 4 (maserati)
Google Droid Bionic (targa)
Google Droid Razr (spyder)
Google Moto E (2015) (otus)
Google Moto E (condor)
Google Moto E LTE (2015) (surnia)
Google Moto G 2014 (titan)
Google Moto G (2015) (osprey)
Google Moto G3 Turbo (merlin)
Google Moto G 4G 2014 (thea)
Google Moto G4/G4 Plus (athene)
Google Moto G 4G (peregrine)
Google Moto G4 Play (harpia)

Device Name
Google Moto G (falcon)
Google Moto X 2013 (ghost)
Google Moto X 2014 (victara)
Google Moto X Play (lux)
Google Moto X Pure (2015) (clark)
Google Moto Z (griffin)
Google Moto Z Play (addison)
Google Nexus 4 (mako)
Google Nexus 5 (hammerhead)
Google Nexus 5X (bullhead)
Google Nexus 7 2013 (4G) (deb)
Google Nexus 7 2013 (Wi-Fi) (flo)
Google Nexus 9 (LTE) (flounder_lte)
Google Nexus 9 (Wi-Fi) (flounder)
Google Nubia Z9 Max (nx512j)
Google Photon Q (xt897)
Google Pixel 2 (walleye)
Google Pixel 2 XL (taimen)
Google Pixel (sailfish)
Google Pixel XL (marlin)
Google Razr (umts_spyder)
Google Robin (ether)
Google Shield Android TV (foster)
Google Shield Portable (roth)
Google Shield Tablet (shieldtablet)
Google Vibe K5 / K5 Plus (A6020)
Google Vibe Z2 Pro (kingdom)
HTC 10 (pme)
HTC One 2014 Dual SIM (m8d)
HTC One 2014 (m8)

Device Name
HTC One A9 (GSM International) (hiaeuhl)
HTC One A9 (GSM US) (hiaeul)
HTC One M9 (GSM) (himaul)
HTC One M9 (VZW) (himawl)
Huawei Honor 4/4x (cherry)
Huawei Honor 4x (China Telecom) (che10)
Huawei Honor 5X (kiwi)
LeEco Le 2 (s2)
LeEco Le Max 2 (x2)
LeEco Le Pro 3 (zl1)
LG G2 Mini (g2m)
LG G3 (AT&T) (d850)
LG G3 (Canada) (d852)
LG G3 (Unlocked) (d855)
LG G6 (US) (us997)
LG G Pad 7 LTE (v410)
LG G Pad 7 (v400)
LG G Pad 8.3 (v500)
LG G Pad 8 (v480)
LG G Pad X 8.0 (TMO) (v521)
LG K10 (m216)
LG Optimus L70 (w5)
LG Optimus L90 (w7)
LG V20 (GSM Unlocked) (us996)
LG V20 (VZW) (vs995)
OnePlus 2 (oneplus2)
OnePlus 3 / 3T (oneplus3)
OnePlus 5 (cheeseburger)
OnePlus One (bacon)
OnePlus X (onyx)

Device Name
OPPO Find 7a (find7)
OPPO Find 7s (find7s)
OPPO N3 (n3)
OPPO R5/R5s (Intl) (r5)
OPPO R7 Plus (r7plus)
Samsung Galaxy S4 Mini (3G) (serrano3gxx)
Samsung Galaxy S4 Mini (Dual SIM) (serranodsdd)
Samsung Galaxy S4 Mini (LTE) (serranoltexx)
Samsung Galaxy S5 LTE-A (lentislte)
Samsung Galaxy S5 LTE (G900AZ/F/M/R4/R7/T/V/W8,S902L) (klte)
Samsung Galaxy S5 Plus (kccat6)
Samsung Galaxy Tab 3 7.0 (Sprint) (lt02ltespr)
Samsung Galaxy Tab E 8.0 (LTE) (gtesqltespr)
Samsung Galaxy Tab E 9.6 (Wi-Fi) (gtelwifiue)
Samsung Galaxy Tab Pro 10.1 (Wi-Fi) (n2awifi)
Samsung Galaxy Tab Pro 8.4 (mondrianwifi
Samsung Galaxy Tab S2 8.0 2016 (Wi-Fi) (gts28vewifi)
Samsung Galaxy Tab S2 9.7 2016 (Wi-Fi) (gts210vewifi)
Sony Xperia L (taoshan)
Sony Xperia SP (huashan)
Sony Xperia Tablet Z LTE (pollux)
Sony Xperia Tablet Z Wi-Fi (pollux_windy)
Sony Xperia T (mint)
Sony Xperia TX (hayabusa)
Sony Xperia V (tsubasa)
Sony Xperia ZL (odin)
Sony Xperia ZR (dogo)
Sony Xperia Z (yuga)
Wileyfox Storm (kipper)
Wileyfox Swift (crackling)

Device Name
Wingtech Redmi 2 (wt88047)
Xiaomi Mi 3w and Mi 4 (cancro)
Xiaomi Mi 4c (libra)
Xiaomi Mi 5 (gemini)
Xiaomi Mi 5s (capricorn)
Xiaomi Mi 5s Plus (natrium)
Xiaomi Mi Max (hydrogen)
Xiaomi Mi MIX (lithium)
Xiaomi Redmi 1S (armani)
Xiaomi Redmi 3/Prime (ido)
Xiaomi Redmi Note 3 (kenzo)
Xiaomi Redmi Note 4 (mido)
YU Yunique (jalebi)
YU Yuphoria (lettuce)
YU Yureka (tomato)
ZTE Axon 7 (axon7)
ZTE (Z831) (chapel)
Zuk ZUK Z1 (ham)

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] L. Sicard, M. Markovics, and G. Manthios, “An Ad-hoc Network of Android Phones Using B.A.T.M.A.N,” IT-Univ., Copenhagen, Project Report SPVC-E2010, 2010. [Online]. Available: http://docplayer.net/7526964-An-ad-hoc-network-of-android-phones-using-b-a-t-m-a-n.html

[2] T. Clausen and P. Jacquet, (2003, Oct.). “Optimized Link State Routing Protocol (OSLR),” Project Hipercom, Internet Engineering Task Force, RFC 3626, 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3626.txt

[3] J. Cope, “What’s peer-to-peer (P2P) networking?,” *Computerworld*, Apr. 8, 2002. [Online]. Available: https://www.computerworld.com/article/258828888887/networking/networking-peer-to-peer-network.html

[4] A. Micah, “Secure Infrastructure-less network (SINET),” M.S. thesis, Dept. of Comp. Sci., NPS, Monterey, CA, USA, 2017. [Online]. Available: https://calhoun.nps.edu/handle/10945/55583

[5] I. Kaur, N. Kaur, T. Tanisha, G. Gurmeen, and D. Chaudhary, “Challenges and Issues in Adhoc Network,” Chandigarh Eng. Coll., Punjab, India, 2016. [Online]. Available: http://www.ijcst.com/vol74/1/14-iqbaldeep-kaur.pdf

[6] *IEEE Standard for Information Technology, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, IEEE Standard 802.11-2016, 2016. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7786995

[7] G. Anastasi, M. Conti, and E. Gregori, “IEEE 802.11 Ad Hoc Networks: Protocols, performance and open issues,” *Mobile Ad Hoc Networking*, pp. 69–116, IEEE Press & Wiley-Interscience, 2004. [Online]. Available: https://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=5237135

[8] P. Swain, S. Chakraborty, S. Nandi, and P. Bhanduri, “Performance modeling and analysis of IEEE 802.11 IBSS PSM in different traffic conditions,” in *IEEE Trans. Mobile Comput.* vol.14, no. 8, pp. 1644–1658, 2015. [Online]. Available: https://www.researchgate.net/publication/279633862_Performance_modeling_and_analysis_of_IEEE_80211_IBSS_PSM_in_different_traffic_conditions

[9] Mayarasi, 802.11 management frame types. Sep. 29, 2014. [Blog]. [Online]. Available: https://mrncciew.com/2014/09/29/cwap-802-11-mgmt-frame-types/

[10] Wi-Fi Alliance Technical Committee P2P TG, “Wi-Fi peer-to-peer (P2P) technical specification,” Wi-Fi Alliance, Tech. Rep. Ver. 1.7, 2016. [Online]. Available: https://www.wi-fi.org/file/wi-fi-peer-to-peer-p2p-technical-specification-v17

[11] D. Camps-Mur, A. Garcia-Saavedra, and P. Serrano, “Device to device communications with WiFi Direct: Overview and experimentation,” *IEEE Wireless Communications, vol. 20,* 2013. [Online]. Available: http://www.it.uc3m.es/pablo/papers/pdf/2012_camps_commag_wifidirect.pdf

[12] Wi-Fi Alliance Technical Committee P2P TG, “Wi-Fi protected setup specification,” Wi-Fi Alliance, Tech. Rep. Ver. 1.0h, 2006. [Online]. Available: http://cfile28.uf.tistory.com/attach/16132E3C50FCFFCB3EC74E

[13] “Forum for questions and answers,” Android Enthusiasts. Accessed Dec.19, 2017. [Online]. Available: https://android.stackexchange.com/questions/173875/what-is-the-functionality-concept-behind-wi-fi-sharing-aka-wi-fi-profile-share

[14] “Why Wi-Fi Direct cannot replace ad-hoc mode,” Thinktube. Accessed Jan. 3, 2018. [Online]. Available: http://www.thinktube.com/index.php/tech-en/android/wifi-direct

[15] B. Aboba, L. Blunk, J. Vollbrecht, J. Carlson, and H. Levkowetz, “Extensible Authentication Protocol (EAP),” Microsoft, RFC 3748, 2004. [Online]. Available: http://tools.ietf.org/rfc/rfc3748.txt

[16] “The Android source code,” Android Open Source Project. Accessed Dec. 15, 2017. [Online]. Available: https://source.android.com/

[17] “Android compatibility,” Android Open Source Project. Accessed Dec. 15, 2017. [Online]. Available: https://source.android.com/compatibility/

[18] “Open source release center,” Samsung Opensource Release Center. Accessed Dec. 16, 2017. [Online]. Available: http://opensource.samsung.com/reception.do

[19] “Overview – Unlegacy Android,” Unlegacy Android. Accessed Dec. 18, 2017. [Online]. Available: https://www.unlegacy-android.org/projects/unlegacy-android

[20] “LineageOS – LineageOS Android distribution,” LineageOS Android Distribution. Accessed Dec. 26, 2017. [Online]. Available: https://lineageos.org/

[21] “CyanogenMod,” *Wikipedia*. Dec. 12, 2017. [Online]. Available: https://en.wikipedia.org/wiki/CyanogenMod

[22] “lunch - Distributed process launcher,” Ubuntu Manuals. Accessed May 2, 2018. [Online]. Available: http://manpages.ubuntu.com/manpages/bionic/man1/lunch.1.html

[23] “Android sparse image format,” 2net Training for Embedded Linux and Android, Feb. 9, 2014. [Online]. Available: http://www.2net.co.uk/tutorial/android-sparse-image-format

[24] “Android architecture,” Tutorialspoint. Accessed Jan. 3, 2018. [Online]. Available: https://www.tutorialspoint.com/android/android_architecture.htm

[25] J. Fernandes, “Flattened Image Trees: A powerful kernel image format,” Texas Instruments presentation, Feb. 21, 2013. [Online]. Available: https://elinux.org/images/f/f4/Elc2013_Fernandes.pdf

[26] “Android kernel source code,” Google Git. Accessed Dec. 27, [Online]. 2017. Available: https://android.googlesource.com/kernel/

[27] devicetree.org. (Dec. 2017). The Device Tree specification, rel. v0.2. [Online]. Available:https://github.com/devicetree-org/devicetree-specification/releases/download/v0.2/devicetree-specification-v0.2.pdf

[28] San Jose B2B Team, “KNOX glossary of terms and acronyms,” Samsung Enterprise Mobility Solutions, Santa Clara, CA, 2013. [Online]. Available: https://slidex.tips/download/samsung-telecommunications-america-samsung-knox-knox-glossary-of-terms-and-acron

[29] “Download Odin with Android ROM flashing tool,” Samsung Odin. Accessed Jan. 4, 2018. [Online]. Available: https://odindownload.com/

[30] “Offensive Security’s Exploit Database archive,” Exploit Database by Offensive Security. Accessed Jan. 4, 2018. [Online]. Available: https://www.exploit-db.com/

[31] G. Beniamini, “Project Zero, Over The Air: Exploiting Broadcom’s Wi-Fi stack (Part 1),” Apr. 2017. [Blog]. [Online]. Available: https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html

[32] M. Schulz, D. Wegemer, and M. Hollick, “NexMon: A cookbook for firmware modifications on smartphones to enable monitor mode,” TU Darmstadt, Germany, 2015. [Online]. Available: https://www.researchgate.net/publication/301874463_NexMon_A_Cookbook_for_Firmware_Modifications_on_Smartphones_to_Enable_Monitor_Mode

[33] “Devices using the Broadcom wireless chipset,” WikiDevi. Accessed Jan. 16, 2018. [Online]. Available: https://wikidevi.com/wiki/Broadcom

[34] Cypress, *BCM4339 datasheet*, Cypress Semiconductor Corporation, San Jose, CA, 2017.

[35] ARM Limited, *Cortex-R4 and Cortex-R4F Technical Reference Manual*, rev. r1p4, Apr. 3, 2011. [Online]. Available: infocenter.arm.com/help/topic/comarmdoc.ddi0363G_cortex_r4_r1p4_trm.pdf

[36] G. Beniamini, “Project Zero, Over The Air: Exploiting Broadcom’s Wi-Fi stack (Part 2),” Apr. 2017. [Blog]. [Online]. Available: https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_11.html

[37] “Broadcom wireless driver for Android,” Google Git. Accessed Jan. 3, 2018. [Online]. Available: https://android.googlesource.com/kernel/common.git/+/bcmdhd-3.10

[38] “Nexmon,” Github. Accessed Jan. 3, 2018. [Online]. Available: https://github.com/seemoo-lab/nexmon

[39] M. Schulz, D. Wegemer, and M. Hollick, “DEMO: Using NexMon, the C-based WIFI firmware modification framework,” in *WiSec ‘16, Proc. 9th Annu. Conf. Security & Privacy in Wireless and Mobile Networks*, Darmstadt, Germany, July 18–22, 2016. [Online]. Available: DOI:10.1145/2939918.2942419

[40] E. Soares, P. Brandao, R. Prior, and A. Aguiar, “Experimentation with MANETs of smartphones,” Univ. of Portugal, Portugal, 2017. [Online]. Available: http://arxiv.org/pdf/1702.04249.pdf

[41] “Ad-Hoc (IBSS) mode support for Android 4.2.2 / 4.3 / 4.4 / 5.0,” Thinktube. Accessed Jan. 3, 2018. [Online]. Available: http://www.thinktube.com/android-tech/46-android-wifi-ibss

[42] W. Jung, H. Ahn, and Y. Ko, “Designing content-centric multi-hop networking over Wi-Fi Direct on smartphones,” Ajou Univ., Korea, 2014. [Online]. Available: https://www.researchgate.net/publication/287192258_Designing_content-centric_multi-hop_networking_over_Wi-Fi_Direct_on_smartphones

[43] “FireChat,” Open Garden. [Accessed Jul. 23, 2017. Online]. Available: http://www.opengarden.com/firechat.html

[44] C. Funai, C. Tapparello, and W. Heinzelman, “Supporting multi-hop device-to-device networks through WiFi Direct multi-group networking,” Univ. of Rochester, NY, USA, 2015. [Online]. Available: https://arxiv.org/pdf/1601.00028.pdf

[45] C. Casetti, C-F.Chiasserini, L.Curto Pelle, C. Del Valle, Y. Duan, and P. Giaccone, “Content-centric routing in Wi-Fi Direct multi-group networks,” Polytechnic Univ., Turin, Italy, 2014. [Online]. Available: https://arxiv.org/abs/1412.0880

[46] O. Turkes, H. Scholten, and P. Havinga, “BLESSED with opportunistic beacons: A lightweight data dissemination model for smart mobile ad-hoc networks,” in *CHANTS ‘15*, *Proc. 10th ACM MobiCom Workshop on Challenged Networks*, Paris, France, Sep. 11–15, 2015, pp. 25–30.

[47] G. Hierts, T. Denteneer, S. Max, and B. Walke, “IEEE 802.11s: The WLAN mesh standard,” Aachen Univ., Aachen, Germany, 2010. [Online]. Available: https://www.researchgate.net/publication/224116334_IEEE_80211s_the_WLAN_mesh_standard

[48] D. Harkins. “Simultaneous authentication of equals: A secure, password-based key exchange for mesh networks,” in *Proc. 2008 Second Intl. Conf. Sensor Technol. & Applications (SENSORCOMM), IEEE Comput. Soc.,* Cap Esterel, France, Aug. 25–31, 2008, pp. 839–844.

[49] “sdat2img: Convert sparse Android data image to filesystem ext4 image,” Github. Accessed Jan.10, 2018. [Online]. Available: https://github.com/xpirt/sdat2img

[50] “simg2img: Convert Android sparse images to raw images,” Github. Accessed Jan.10, 2018. [Online]. Available: https://github.com/anestisb/android-simg2img

[51] “Android factory firmware images,” Google APIs for Android. Accessed Jan. 26, 2018. [Online]. Available: https://developers.google.com/android/image

[52] “Cybersecurity solutions for the modern enterprise,” Tripwire Inc. Accessed Mar. 17, 2018. [Online]. Available: https://www.tripwire.com/

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California