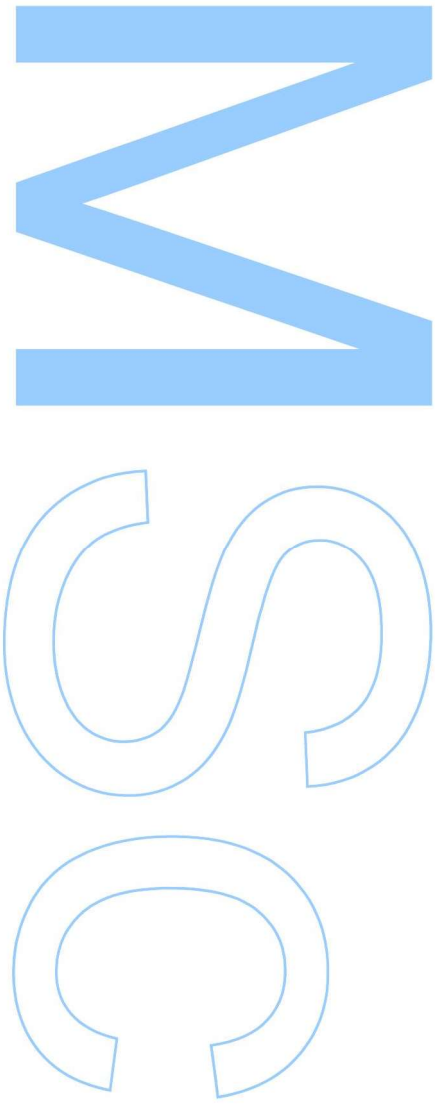
Porting Wireless Mesh Networks to the Android System

Luís Miguel Salgueiro Barroso
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciências de Computadores
2016

Orientador
Rui Prior, Professor Auxiliar, FCUP

Coorientador
Pedro Brandão, FCUP



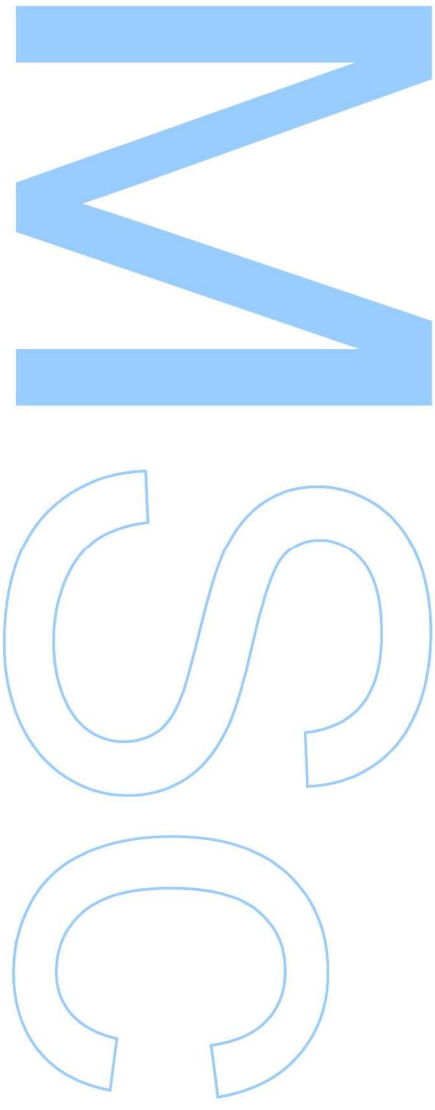
U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____



Abstract

Wireless Mesh Networks (WMN) are now a part of IEEE 802.11 standard. Presented as an non-infrastructured solution to deploy large scale networks, they are now on the focus of academical and commercial interest. Mobile devices are simultaneously growing, implementing and providing the state of the art wireless technologies available, but are still lacking native support for WMNs. As of 2010, Cozybit created the open80211s project, which is now integrated in the Linux Kernel and since then has not been maintained by Cozybit, leaving developers without a source of documentation on how to accomplish WMN support in current devices and architectures. On the course of this dissertation, we present the methodology associated with the whole process of providing support for WMN in Android devices. More than the final result, we will provide every single implementation step, to help towards the development of new solutions and hopefully provide a starting point for future implementations. We will overview the android architecture regarding wireless drivers, and present a step-by-step solution to natively create and integrate WMN capabilities on an open-source CyanogenMod distribution, based on android, providing all the details on the system build as well as presenting the result we obtained.

Resumo

As Wireless Mesh Networks (WMN) são agora parte integrante da norma IEEE 802.11 para as sem fios. Inicialmente apresentadas como uma solução para a criação de redes não infra-estruturadas para criar redes de proporção metropolitana, estão agora no foco da investigação da comunidade académica e empresarial. Os dispositivos móveis têm também crescido de forma exponencial e usufruindo da evolução tecnológica das redes, ao incluirem as novas tecnologias nos produtos apresentados, apesar do suporte para WMN não estar ainda presente nos dispositivos de forma nativa. Em 2010, a CozyBit criou o projecto open80211s, sendo este agora parte integrante do kernel, e desde então não mais mantido pela Coxybit, deixando os futuros *developers* sem uma fonte viável de documentação em como conseguir providenciar suporte para WMN nos dispositivos e arquitecturas actuais. No decorrer desta dissertação, apresentamos a metodologia associada com o processo de provicenciar suporte para WMN em dispositivos Android. Mais do que o resultado final, providenciamos todos os detalhes de implementação de forma auxiliar o desenvolvimento de novas soluções, e providenciar um ponto de partida para futuras implementações. Iremos analisar a arquitectura de rede em android, especificamente os *drivers wireless* e apresentar discritivamente uma solução para criar e integrar o suporte para WMN numa distribuição do sistema aberto CyanogenMod baseado em Android, bem como apresentar todos os resultados que obtivemos.

Dedico o trabalho desenvolvido aos meus pais, Adelino Barroso e Maria José Salgueiro. Sem eles, nunca tería conseguido ser a pessoa que sou hoje. Pelos valores que me foram transmitidos na minha educação, por me ensinarem o valor do trabalho, pelos sacrifícios feitos para que eu aqui chegasse, o meu muito obrigado.

Acknowledgements

I would like to thank my advisors, Rui Prior and Pedro Brandão, for all the guidance throughout the development of this development. It is rare to find people willing to help you no matter what or when. I will always carry the 5 minutes meetings that lasted for two hours in my memory. I would also like to thank Eduardo Soares, for all the input in the research and implementation steps.

I would also like to thank my colleagues, whose friendship and companionship made the long days and nights of work bearable.

Finally, and most importantly, I would like to thank my parents Adelino Barroso and Maria José Salgueiro, my brothers João and Ricardo, my sister Matilde, and my girlfriend Liliana, for all the patience, for being my best friends, for helping me go through with my work, for all the smiles, companionship, confidence votes, and for giving me the strength when I needed it the most. I love you with all my heart.

Contents

List of Tables

List of Figures

Listings

Acronyms

IEEE Institute of Electrical and Electronics Engineers

WMN Wireless Mesh Networks

CM Cyanogenmod

MAN Metropolitan Area Networks

AP Access Point

WDS Wireless Distribution System

DS Distribution System

IBSS Independent Basic Service Set

MANET Mobile Ad-hoc Networks

STA Mesh Station

QoS Quality-of-Service

MAC Media Access Control

ESS Extended Service Set

MP Mesh Point

TTL Time-to-live

HWMP Hybrid Wireless Mesh Protocol

LTS Long Term Support

IDE Integrated Development Environment

SDK System Development Kit

TCP Transmission Control Protocol

ADB Android Debugging Bridge

SELinux Security Enhanced Linux

MAC Mandatory Access Control

DAC Discretionary Access Control

ACL Access Control List

WEXT Wireless Extensions

NDK Native Development Kit

SAE Secure Authentication of Equals

API Application Programming Interface

HAL Hardware Abstraction Layer

MSM Mobile Station Modem

ARM Address Resolution Protocol

GUI Graphical User Interface

ARP Address Resolution Protocol

ADB Android Debug Bridge

BS Base Station

BSS Basic Service Set

IP Internet Protocol

Chapter 1

Introduction

The Wireless Mesh Networks (WMN) concept revolves around a catastrophe scenario. Initially though to serve military purposes [34], WMNs are now the focus of the commercial and academical community.

The main goal is to quickly deploy a self-healing, self-configuring and self-organizing network, without the need of a central infrastructure. WMN are extremely versatile, as any intervening device can employ several functions such as relay data, distribute internet access or interconnect with non-Mesh networks.

Smartphones, and other capable devices are suitable targets to integrate a WMN, as long as the support exists on the operating systems that manages them. Mobile devices users have also been increasing meaning that there is a wide range of candidates that could resort to the WMN technology. Android poses as the leading operating system among mobile users [30], encompassing the advantage of being free, open, and with community driven alternatives freely available.

The IEEE 802.11 added Mesh networking support to the standard in the 802.11s ammendment [41], providing rules in order to standardize the operation modes and functions of WMN.

Some community WMN have been tested and deployed [29][5][26], proving that the concept is viable and properly working on the creation of a wide coverage range community networks.

Although it is an emerging technology, WMN are not natively supported on the majority of the aforementioned smart devices. Some of the reasons as why this happens are related with the lack of native support, either from from the wireless network card driver or from the android API itself.

Cozybit created the open80211s[10] project, which is now integrated in the Linux Kernel and since then has not been maintained by Cozybit, leaving developers without a source of documentation on how to accomplish WMN support in current devices and architectures.

Our goal is to port its features onto Android, with recent technology, providing the specific

documentation assuring that future developers can have a starting point to accomplish similar or new goals. The final product of this implementation is a Cyanogenmod (CM) rom with native WMN support, that relies on a Mesh compliant driver.

Chapter 2

Related Work

In this chapter we will go through some of the current technology regarding Wireless Mesh Networks. We will also overview the functionalities of Mobile Ad-hoc Networks. Finally, we will compare both, highlighting the main differences.

2.1 Wireless Mesh Networks

As previously mentioned, the Wireless Mesh Networks (WMN) concept was originally conceived considering military or catastrophe scenarios. Now, some commercially interesting applications such as community and neighborhood networks, intelligent transportation systems, Metropolitan Area Networks (MAN) and spontaneous deployments are arising and the inherent technology is being a great focus of the academic and commercial community. WMNs have a great echonomical and pratical interest attached, as a WMN node can be either a device, a router, an Access Point (AP) or all the aforementioned. Wireless Mesh Networks operate under 802.11s ready devices, meaning that a Mesh capable wireless interface can be used as a Mesh Station (STA) omitting the need for specially developed sensors in order to deploy a WMN. Moreover, WMNs overcome the imposed limit on the range achieved by WLANs in the 2.4 and 5 GHZ unlicensed bands, by implementing multi-hop communication, allowing for the deployment of wider wireless coverage, as seen, for instance, in office/university campuses ow Community-driven Wireless Networks [3][2][29][26][5]. Taking into account the exponential growth in smartphones and other smart devices users, as well as the increase of manufacturers providing several levels of cost for android enabled devices, a solution which resources to them seems a good approach for the future, and that subject matter is the focus of the work presented.

2.1.1 Overview

The 802.11 Wireless Standard [41] came from the need to replace the existing ethernet cabled network. The concept of WDS was defined by the 802.11 working group in 2003 as a mechanism

for wireless communication using a four address frame format ([41] def 3.170) and was not further improved, even lacking a specification of the frame format for such a mechanism.

802.11 Wireless Networks provide a solution in terms of replacing an ethernet link with a wireless link, but are restrained in terms of coverage and need of specific hardware (multiple AP to provide a good coverage) although efficient when wishing to expand a wireless network to a parking lot or a small public place. But if the need arises to expand to a college campus, or even an metropolitan wide area network, it would imply a significant infrastructure cost. Moreover, 802.11 Wireless Networks always need to rely on a wired backbone, accessible from an AP in order to communicate. In order to efficiently replace the wired backhaul, WMN were conceived to achieve the following goals:

- Flexibility: Hardware restrictions regarding the switch ports available are now discarded as wireless links are established between the intervening Mesh nodes. The routing capabilities enable connection to a Distribution System (DS) via just one AP (through the backhaul) even if located at a great distance;
- Self forming: If the routing protocol is able to determine the possible paths in the WMN, expanding will be a question of adding new Mesh capable devices;
- Self-healing: Given the capability to automatically find a path from one device to the other, the failure of one link is solvable by finding a different path to the target node, if it is still within reach of other nodes;

The IEEE 802.11 standard [41] describes ad-hoc networking in the Independent Basic Service Set (IBSS)) mode, where devices could connect to each other directly, without the need of an infrastructure (AP). IBSS operate in a single-hop manner (we will see that Mobile Ad-hoc Networks (MANET) resort to IBSS to implement multi-hop with additional protocols in section 2.2), meaning that the device could only communicate to devices within its reach. But it lacked the access or connection to the DS [44]. The wireless Mesh concept was developed to overcome such restrictions and combine both the IBSS and infrastructure mode integration in a new type of multihop networks. WMN implement a two-tier network, composed by a backhaul tier (Mesh node to Mesh node) and an access tier (Mesh node to client), enabling operation without an wired backhaul.

For several years, proprietary solutions [40] arose to implement such networks, defining basic intervening parts (Mesh routers, Mesh clients and Mesh gateways) and a routing protocol to allow inter-working among them. But, being proprietary solutions, lacked coherency, and would no work with other solutions.

The need for a standard was urgent, and IEEE 802.11 instantiated the task-force S in order to define a standard for such networks, and in 2012 an amendment was published and the norm 802.11s created.

2.1.2 IEEE 802.11s

The IEEE 802.11 taskforce S was originally assigned in 2004, taking more than 7 years to come to a final amendment for Mesh networking. In 2011 the amendment was approved and published as 802.11s. 802.11s is now officially part of IEEE 802.11. In this section, we will do a brief analysis of the device and network terminology, as well as the features presented by the 802.11s amendment.

Table 2.1: Terminology [41]

Name	Acronym	Description
Mesh Station	MSTA	An 802.11-standard-compliant MAC and physical (PHY) layer implementation device
Mesh Basic Service Set	MBSS	If two Mesh STAs connect to each other, they form the most simple elementary 802.11 network, called a Mesh Basic Service Set (MBSS)
Access Point	AP	A Mesh STA can also provide integration service to other devices, becoming an Access Point (AP)
Distribution System	DS	Provides the services that are necessary to communicate with devices outside the station's own BSS
Extended Service Set	ESS	When the DS allows multiple APs to unite and roam from one BSS to another
Mesh Portal	MP	Provides integration of WLANs with non-802.11 networks
Mesh Gate	GW	Logical architectural component that integrates the MBSS with the DS

In order to understand how a WMN operates, some concepts need to be introduced. According to [23], a "station" is a subset of functions for a device and not a physical device for itself. This is an important definition to cover the versatility of Mesh devices. A Mesh-compliant android cellphone can operate with Mesh station functions and at the same time operate as a AP, enabling other 802.11 devices to associate. A Mesh STA is a versatile device, that can be a source, a sink or a propagator of traffic [23]. Only Mesh STAs participate in Mesh functionalities such as formation of the Mesh Basic Service Set (BSS), path selection, and forwarding, and it does not connect with non-Mesh STA by standard.

Connection with non-Mesh - e.g. wired, other 802.11 devices - is accomplished through the logical Mesh gate and Mesh portal functions. Inside a Mesh BSS, all STAs establish wireless links with neighbour STAs for mutual message exchange, but connection to outside infrastructure networks is done via the Mesh Distribution System (DS). The DS is a logical component that handles address to destination mapping. It handles the frame distribution from interfaces, enabling for instance, that data from a client on the 2.4 GHz radio can be transferred to a the 5 GHz radio or to the wired network. This means that the DS is a logical entity, and not necessarily a separate medium.

In order for a STA to access the DS, an AP is needed. The AP is a station that provides access to the distribution services, via the wireless medium for associated stations. It enables association from other STAs and operates as the central point of transit for its client stations

and responsible for delivering the frames sent by them. The function of an AP that performs this translation between the wireless network and the wired network defined as the portal. Like the STA, a portal is a function, and is defined as the logical point where wireless Mandatory Access Control (MAC) service data unit (MSDU) are translated to and from a non-802.11 network ([41] def 3.39, 3.110). When that translation occurs between a BSS and a non-Mesh 802.11 DS, this function is called Mesh gate. Simply put, a portal is responsible for connection to non-802.11 DS and a gateway is responsible for the frame translation to a non-Mesh 802.11 DS (wireless).

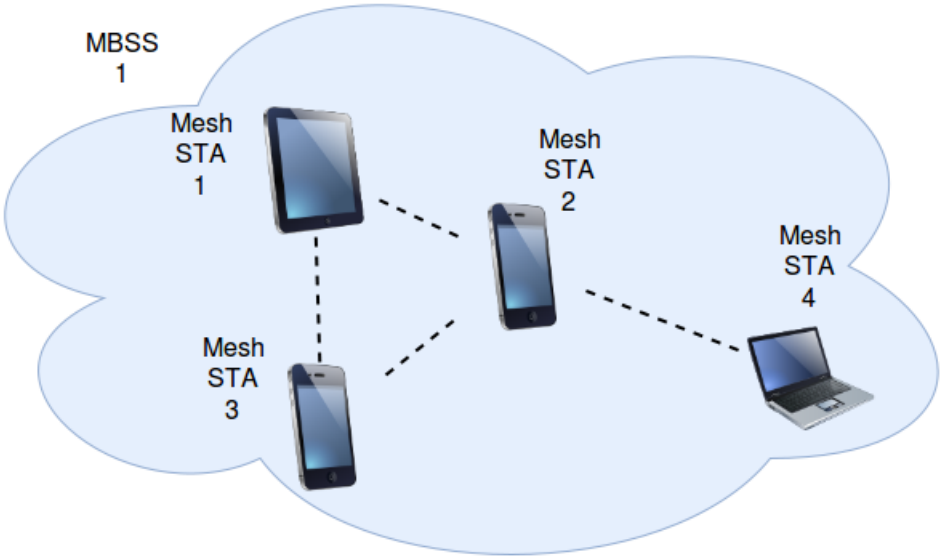


Figure 2.1: Mesh Basic Service Set

In Figure 2.1 we can state a MBSS formed by mobile Mesh capable devices. All the Mesh STAs within the MBSS can communicate directly without the need of an infrastructure. Mesh STA 4 can communicate with Mesh STA 1 and Mesh STA 3 through Mesh STA 2.

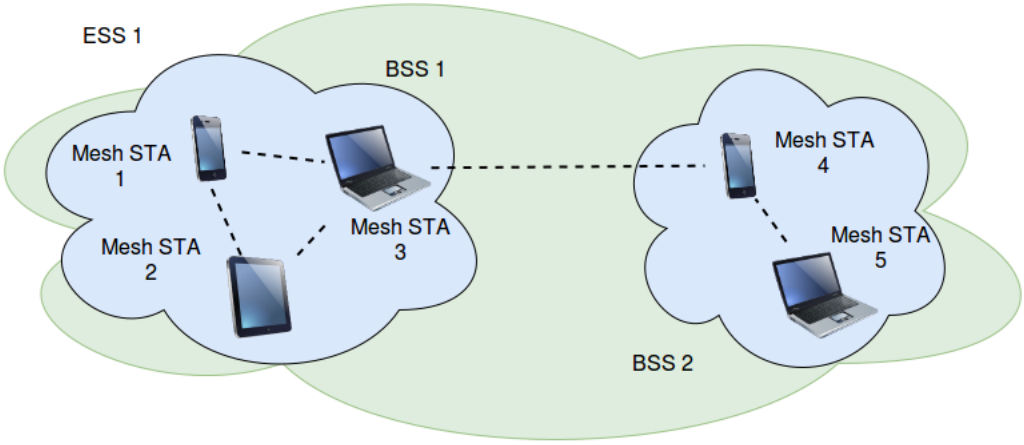


Figure 2.2: Mesh Extended Service Set

If a third Mesh STA, operating as an AP on an external BSS would join BSS 1, and was providing access to devices within its native BSS, it would then expand both BSSs into an Extended Service Set (ESS) as seen in Figure 2.2. That would enable all the intervening Mesh STAs to directly interconnect seamlessly at the Medium Access Control (MAC) Layer. Through the routing capabilities of 802.11s, Mesh STA 5 would be able to communicate with Mesh STA 1 as if they were right next to each other. It is important to notice that both Mesh STAs must be operating as Mesh gateways, as they are both part of 802.11 networks (wireless or not).

The setup portrayed would only enable Mesh STAs to communicate with each other. Expanding the ESS capability to - for instance - provide internet access to all the intervening Mesh STAs would be easily accomplished if a device was available working as a Mesh Portal. If a Mesh-compliant Router R1 was introduced and configured to operate as an AP and Mesh Portal, and also connected to an 802.3 cabled network with internet connection, all the devices connected in ESS 1 would be able to connect and provide connection to the internet. That scenario can is portrayed in Figure 2.3, where R1 provides the logical entry point to the DS (802.3 Lan), by operating as both an Mesh portal and a Mesh AP.

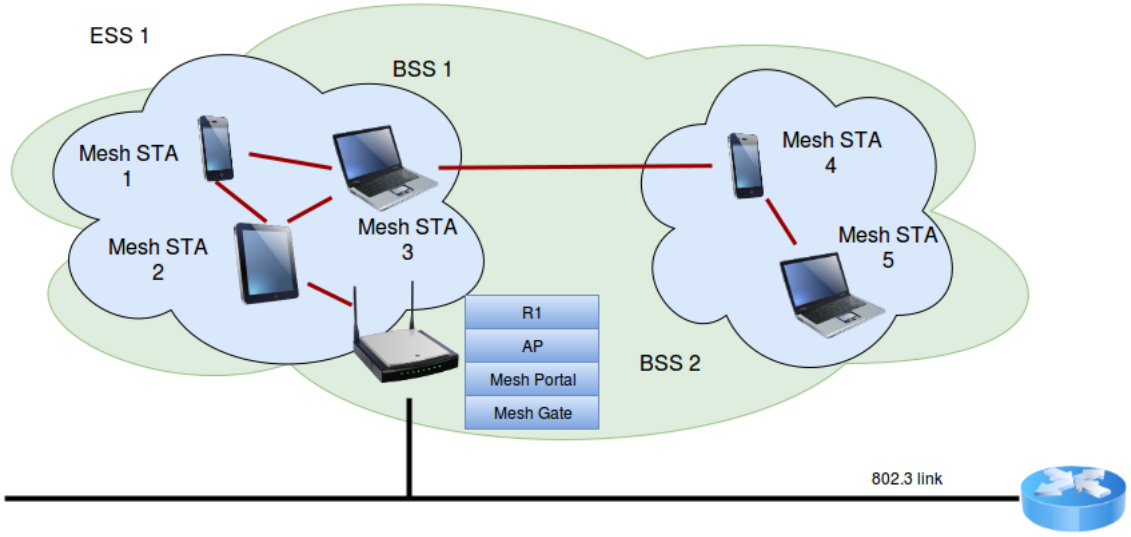


Figure 2.3: Distribution System Integration

In order to communicate Mesh nodes must choose an channel to operate. To accomplish that, the Simple Channel Unification Protocol is used in order for the Mesh Point (MP) to perform either a passive or an active scanning of possible Mesh neighbors [35]. If a neighbor is found, the channel is set to match its neighbor, if not, the MP can establish itself as the initiator of a Mesh network and the channel is calculated based on a channel precedence value (boot time of the MP plus a random value) [35]. If two disjoint MPs, or networks, are discovered, the channel is selected based on highest precedence value. After the channel is selected, the MP needs to find the topology and establish a link with neighbor MPs. For that, a joining MP will perform a neighbor discovery, scanning for beacons that contain at least one profile (Mesh ID, path selection protocol identifier and link metric identifier). If the link is valid, the link can be established through a secure protocol. When the MP is validated and connected to the WMN, data can start to flow.

In terms of MAC enhancements, Mesh networks implement a new Mesh-supported 802.11 frame format, introducing a Mesh Header in the MAC frame structure. This header contains:

- Mesh flags – Bit 0 Address extension, bits 1-7 reserver for future use.
- Mesh Time To Live (Time-to-live (TTL)) - similar to TTL in common wireless frames, the number of hops a Mesh frame can be transmitted, decremented by one at each hop and discarded when zero is reached.
- Mesh Sequence Number – sequence number to discard duplicate received frames
- Mesh Address Extension (present in some configurations) – extension modulo to support four-address, six-address or multi-hop frames.

In the next chapter, we will overview the Mobile Ad-hoc Networks concept. MANET are improvements made to ad-hoc mode. In order to deploy a Mesh-like MANET, all the devices must be configured, routing protocols must be added, and drivers adapted. In the case of 802.11s, almost all drivers that support softmac via mac80211, will be able to natively implement a WMN without further configuration. Currently, some drivers are surfacing, that have native Mesh support. Atheros provides ATH9k and ATH10K that equip several portable devices (laptops, smartphones, tablets), fixed devices (desktops, routers) and are also available in several USB wireless cards.

Also, 802.11s is a standard by IEEE, and therefore all the devices will agree on the same rules and operation mode predicted by the draft. We will briefly present some of the features of WMN.

In order to accomplish data delivery, a routing (or path selection) protocol must be used. 802.11s [41] presents the Hybrid Wireless Mesh Protocol (Hybrid Wireless Mesh Protocol (HWMP)). As WMNs implement a tree structure, with traffic flowing mainly from and to gateway nodes, HWMP performs hierarchical routing to exploit such logical structure. Based on AODV [1], it allows for on-demand and proactive routing. Being the standard, all MPs must have HWMP implemented, although the 802.11s draft also defines and optional Radio Aware-Optimized Link State Routing. It is also important to mention that WMNs provide a Extensible path selection framework ([41] ch. 13.8.2) to ease implementations of flexible path selection protocols and metrics. The link metric used by default is the Airtime Link Metric ([41] ch13.9) but can be overridden by the extensible path selection framework.

Since communication is done through a wireless link, and nodes are mobile, there can be a scenario where a device is operating as a bridge or portal to several devices which need that path. In order to avoid data loss, Mesh networks implement congestion control mechanisms, providing Quality-of-Service (QoS) and assurance regarding its communication ([41] ch 6.3.78).

Security is also defined within the standard. When connection is firstly established, the nodes negotiate, and agree upon, a pairwise ciphersuite and a group cipher suite also establishing a Mesh TKSA and Mesh GTKSA to be used with the pairwise cipher suite and group cipher suite, respectively. ([41] ch 13.7). A WMN is said to be secure when it provides security authentication protocols. When there is no authentication, it is said to be an open Mesh.

The 802.11s draft was first put to action within the open80211s [10] project. Open80211s, is an open-source implementation of the IEEE 802.11s wireless Mesh standard, created to run on any hardware supported by the Linux Kernel. The original idea behind open80211s was to consolidate the multiple non-interoperable Mesh protocols into one, based on the IEEE 802.11s draft.

[1] https://www.ietf.org/rfc/rfc3561.txt

2.2 Mobile Ad-Hoc Networks

Mobile Ad-hoc Networks (MANETs) emerged in the 1990s due to the, at the time, and were popularized by recent technologies such as Bluetooth that enabled devices to directly communicate without resourcing to an infrastructure and the appearance of new standards by IEEE 802.11. Ad-hoc Networking [32] is not a new technology, and some original implementations date back as early as 1972, with the DARPA Packet Radio Network (PRNet) which explored the packet switching technology and store-and-forward routing among devices.This was one of the first approaches regarding routing among devices communicating without an infrastructure. In 1932, DARPA also developed Survivable Radio Networks (SURANs) as an improvement present in PRENet regarding the network scalability, security, energy management and process capability. The main target was to develop network algorithms to support scalable networks up to the tens of thousands of nodes keeping in mind security, cost and power consumption. In the late 1980s, and due to the growth of the internet structure, a high progress was noted. The U.S. Department of Defense initiated the DARPA Global Mobile (GloMo) information system program in 1994 which provided Ethernet-type multimedia connectivity anytime, anywhere, among wireless devices.

The aforementioned technologies, such as Bluetooth, in the 1990s attracted the community and implementations outside of the military domain started to appear. As a result, several applications have surfaced, namely within the academical, home and commercial domain. An extensive historical overview can be found in [32]

MANETs can be interpreted as a subset of Mesh networks [39]. The inherent concept of Mesh networks was originally derived from mobile ad-hoc networks and its capabilities. The basic concept of MANETs was the rapid deployment of a network, that was self configured and self-healing, as the nodes are mobile. A node should also be able to join and relay data among other intervinients in the MANET. But MANETs lack some capabilities present in Mesh networks. If a node, in a MANET, is operating in a given radio frequency, it cannot communicate with other devices running on different radio frequencies. The WMN relies on two-tier node implementation, where the routers maintain an active connection, working on a stationary manner and enabling nodes to connect, hence, introducing an hierarchy in the network architecture [39]. Also, Mesh nodes operate as Mesh routers, routing the information among them, enable multiple functions within a single device. Integration of different IEEE networks is not possible within ad-hoc nodes. The authors in [39] state

> "Clients in ad hoc network operating with a particular radio technology cannot access a client in a different radio technology network (for example a client in WiFi cannot access a client in WiMax or other networks)"

Although we may define the main differences, MANETs have been highly developed by the community. Several upgrades and implementations brought the MANET capacity to almost a full Mesh capacity namely with the development of routing protocols and node behaviour that enable such an abstraction. Nontheless, WMNs are defined by a standard and all the capabilities

are present within it. MANETs can approach a WMN, but pre-configuration is needed among the nodes. In terms of mobile implementations, both MANETs and WMNs are supported among several devices. But in such devices, WMNs take an advantage in power-management. The authors in [37] analize the power consumption and restrains within MANETs, also providing techniques used to improve the default consumptions. Mobile devices are battery-powered meaning that power-consumption must be taken into account seriously in order to keep providing QoS to the final user, and WMNs have power saving options which enable three modes such as Active, Light Sleep and Deep Sleep which can contribute to deployments in smartphones. Taking into account the aforementioned reasons, we think WMNs are a more suitable implementation taking into account that the target devices are smartphones and battery operated devices. Mobile wireless-enabled devices now range from small devices such as smartwatches, up to cars and home-security systems. Due to that range of different devices, we need to assure that devices operating in different radio frequencies can communicate, that security is implemented, and the deployment can be quick and effective. WMNs are a more suitable choice regarding these concerns, as all the aforementioned problems are solved and available natively within the standard. Also, new drivers are surfacing that have native Mesh support. Atheros provides ATH9k and ATH10K that equip several portable devices (laptops, smartphones, tablets), stationary devices devices (desktops, routers) and are also available in several USB wireless cards providing a great versatility in possible participant devices.

Mesh Networks are now available within drivers that equip smartphones. There are some projects that aim to provide Mesh support, such as the WCN36XX project, which provides a Mesh solution for the driver of the Nexus 4 smartphone. This device was available for use and research within the hardware range of the college.

Due to the aforementioned reasons, we chose to go through with adding Mesh support on the Nexus 4.

2.3 802.15.5

802.15.5 [28] is a recommended practise for implementing low-rate Mesh networks resourcing to 802.15.4 [1] enabled devices as well as provide the architectural framework enabling (Wireless Personal Area Networks) WPAN devices to promote interoperable, stable, and scalable wireless Mesh topologies.

The 802.15.4 defines ultra low complexity, ultra low cost, ultra low power and low data rate wireless connectivity among inexpensive devices. This is a common norm applied to Wireless Sensor Networks, on which devices have a precision ranging capability that is accurate to one meter. 802.15.4 operates in a star or peer-to-peer topology, creating a Personal Area Network (PAN). To accomplish so, it nominates a PAN Coordinator, which is the primary controller of the PAN and intervening devices. It is usually a powered device as it requires more processing overhead, but generally, PAN devices are commonly small battery powered devices.

The peer-to-peer topology also has a PAN coordinator, however, intervening devices can communicate with each other, as long as they are in the range of one another. This is the primary topology chosen for the Mesh network topology. Each device is capable of communicating with any other device within its radio communications range. The P2P PAN allows multi-hopping, as data hops from intermediary devices to reach its target, but on a higher layer. The 802.15.4 norm does not describe multi-hopping among 802.15.4 devices. To accomplish a simple form of Mesh topology, the 802.15.4 resources to the P2P topology in order to form a cluster tree.

The simplest form of a cluster tree network is a single cluster network, but larger networks are possible by forming a Mesh of multiple neighboring clusters. [2]

This kind of formation enables the creation of a Mesh-like topology, where devices are interconnected and some multi-hopping communication can occur. As the multi-hop support is not described in [1], the 802.15.5 recommended practise was submitted in order to enable a full-Mesh operation mode. Namely, the 802.15.5 norm encompasses the following features (on the LR-WPAN):

- unicast, multicast, and reliable broadcast Mesh data forwarding
- synchronous and asynchronous power saving for Mesh devices
- trace route function
- portability of end devices

To do so, it provides a new data frame format, incorporating Mesh functions. Association of a new device works similarly to 802.11s, with devices associating to a participant, and becoming a leaf node. Such association creates a tree, starting on the root device (first one). The tree is not a logical-tree, as devices do not have addresses in an early stage. Devices have the autonomy to determine the amount of devices that can join it (how many branches they can host connected to itself). Then a bottom-up procedure is used to calculate the number of devices along each branch. After the root receives the information from all the branches, it should begin to assign addresses, this time using a top-down approach, where the root checks the number of devices and assigns a block of consecutive addresses to the nodes below. After address assigning, a logical tree is formed and each device has populated a neighbor list for tracking branches below it. In case not enough addresses are available from its parent node, a device can either assign addresses for the new nodes below, or require addresses from a predecessor.

In order to accomplish link state, devices broadcast an hello command frame, which enables neighbour nodes to maintain its neighbour list. Devices can also broadcast neighbour information request frames to refresh neighbour lists, enabling them to maintain a so called connectivity matrix, that is refreshed with one-hop information from the hello command frame. Than enables a recurrent method to map and determine devices, and how many "hops" they have from distance.

[2] [1]

Path selection and data forwarding is then accomplished by consulting the neighbour list. If a give device is one hop away, data is forward, if not, a next-hop algorithm is used to find the next hop to forward data. If several neighbour devices are available for the next hop, a device may randomly select one neighbour for load balancing purposes.

Chapter 3

Technologies Used

Every project relies on tools to accomplish goals. In this chapter we will overview which technology was used on the course of this project, stating the reasons that lead to a specific choice.

3.1 Operating Systems

3.1.1 Ubuntu

Ubuntu is a Debian based Linux Operation system, maintained by Canonical Ltd. It is currently one of the most used distributions worldwide. For the purpose of this Master Thesis, we chose version 14.04 as it is a Long Term Support (LTS) version and it is highly stable with both the hardware and software used in the spectrum of this thesis. We chose Ubuntu for some reasons, mainly:

- It natively supports most of the needed drivers used in the development of this thesis;
- There is a big ammount of documentation related with Debian/Ubuntu based distributions;
- Ubuntu 14.04 is Long Term Support LTS, and by that we could assure that the needed dependencies would still be available at implementation time;
- It is free of charge;

3.1.2 CyanogenMod

Cyanogenmod (CM) is an open source operating system distribution based on the Android mobile operating system. It is developed as a free and open source software based on the official releases of Android by Google, with added original and third-party code. [38] CM is developed mainly by unpaid volunteers that form a strong community. Developers urge to share knowledge and provide

help, either via the CM forums, IRC channels and other communities such as XDA-developers [1]. They also provide specific[4] documentation for some devices, on how to compile, modify, tweak and customize the final result.

The CM source code also comes with a set of development tools, in the form of shell scripts that allow the developer to easily test new implementations as well as to compile source code with the call of just one function. In the scope of this Master Thesis, we used mainly[6, 8, 9]:

- `repo`: Repo is a python script, that works as a repository management tool. It is built based on Git and creates an easier to work environment for android development. In our case, repo is mainly used to fetch the several git repositories needed and structure them in our target folder;
- `mm`, `mma`: mma and mm are scripts that will build a module with dependencies and place the target compiled files in the output directory;
- `breakfast`: breakfast will setup the device tree. If it is not available, it will fetch the proper device and its depedencies from Cyanogenmod Github repository for officially supported devices;
- `brunch`: brunch sets the build environment and starts the compilation process. Its final binary output is placed in the output directory and a flasheable zip file is generated. Such file can then be installed in the device through recovery;

CM provides the source code through the use of the "repo" command, wich gathers the necessary components of the operating system from various sources. By doing so, all the source code is divided into several modular components, then compiled with resource to the brunch command which in terms calls all the Android Makefiles available in the source tree and hence developing the final unnoficial Cyanogenmod flasheable zip file. [8]

3.2 Related Tools

3.2.1 Android Studio

Android Studio is the official Integrated Development Environment (IDE) for Android Development. It is based in Intellij IDEA and provides further improvements and tools to Android developers. The full feature list can be found in [42]. In the scope of this Master Thesis, we mainly resourced to Android Studio as the chosen IDE to develop the Mesh Settings APK.

[1] See: http://forum.xda-developers.com/

3.2.2 Android SDK

A System Development Kit is a set of tools used to develop in a given programming language. Google provides Android developers with the Android-SDK, which contains a wide set of software, services and runnable scripts that a developer can use in order to ease the development process. System Development Kit (SDK) tools are a stand-alone component of the Android SDK and provide a set of development and debugging tools for the Android SDK.

In the scope of this Master Thesis, we used mainly to the Android Debug Bridge (`adb`).

3.2.2.1 Android Debug Bridge

Android Debug Bridge (ADB) is a command line tool that allows shell communication with an emulator or an Android device. It works in a client-server approach, by creating a client, a server and a daemon.

When ADB is started on the development machine, it checks for the adb-server processes, and if it cannot find one, it will start the server process. This server runs as a background process on the development machine and listens for commands sent from ADB clients. All the communication is done via the Transmission Control Protocol (TCP) port 5037, and the server is responsible for managing the communication between the client and the daemon running on the device being debugged. In order to communicate, a device must be ADB enabled. Both Android and CM provide ADB mode via USB connection through the hidden developer options menu. Finally, by resorting to the operating system command line to invoke ADB, the client is created and commands can then be sent through it.

Although ADB comes natively with Android Studio, we needed more control over the functions it has. A detailed list of ADB commands can be found in [33].

For implementation purposes, we mainly used

- devices: to identify the devices connected to the development computer. This also assures the daemon is running correctly on the target device;
- logcat: The logcat command enables for log data to be printed in real-time to the terminal. We used it to track error logs ;
- pull and push: to download and upload files directly to the device, namely the compiled flashable zip files;
- shell: to remotely start a shell in the target device;
- install: Calling adb install would enable us to remotely install APK files in the target device;

ADB was extremely important during the implementation of this project. As the phone was rooted, using ADB we were able to access, and debug most of the issues encountered, and try solutions with root access without the need of recompiling. It allowed us to access logcat in real time, modify, upload and download files on-the-spot, which led to an enormous time-saving as we did not have to manually change, compile and upload every single file we defined.

3.2.3 Backports

Backports drivers is a tool that enables newer drivers to run in older Kernels. As the Linux Kernel is continuously updated, new drivers are added for either new devices, or updates released to current ones. The Backports drivers project was originally called compatwireless and then compat-drivers[25]. Backports is widely used in current desktop Operating systems [18] as a way to maintain stable drivers for older kernel versions.

Backports was a very important tool throughout the work developed as we are working with Cyanogenmod Kernel version 3.4 and the wcn36xx driver is not present in that version. After some investigation, we attempted to compile wcn36xx directly from backports python script, but with no luck. We then used the compiled version provided by the Linux Foundation [19]. This version was chosen as it allowed to be ran from a simple Makefile and using the correct flags we were able to automatically assign the Kernel folder and backports automatically resolved the target version and started the compilation process.

As of version 3.16, wcn36xx is available in the Linux Kernel, and hence, we needed to port that driver from 3.16 to 3.4. The backporting process relies on a hierarchical compatibilization of the drivers from the source version to the current version. A `compat.ko` file is then generated with the appropriate compatibilization headers that allow for the driver to work propperly.

3.2.4 SELinux

Security Enhanced Linux (SELinux) is a mandatory access control Mandatory Access Control (MAC) security mechanism implemented in the kernel. [20] [21] Prior to SELinux, the linux Kernel worked in a Discretionary Access Control (DAC) i.e. the access permissions on a given object are derived from a user or software access rights. By having access rights based on rules specified by users, each file is accompanied by a list containing subjects and their rights to that file, i.e., each file has an associated Access Control List (ACL).

A given user or software can, in fact, modify the permissions of a file they own and DAC has in it a flaw. If a given user A provides B with a piece of software, and B trusts A, B will trust the software, even though none of them is aware that the software may contain malicious intents. That is one of the basic principals of computer virus and why peoples trust is exploited (for instance, virus propagation among social networks). Another case is commonly known as a Trojan Virus. A simple scenario would contemplate a given user S that has in its possession

desirable information on the form of an object O. An malicious user S' could then create a file O', assign write permissions to S on O', assign read permissions for itself on that file and copy the O information to O'. If that piece of software manages to be ran with the proper permissions (perhaps hidden as a videogame that the user S would run), O' would receive the contents of O and S' would be able to collect such information. In some cases, a physical user is not needed in order to violate security, as processes inherit users rights.

To avoid that kind of security issues, the MAC policy was implemented. In MAC, permissions are not assigned to each file, nor the user can modify permissions that easily. Instead, access rights are based on the regulation by a central authority, that is, the security policies belong to an organization rather than individual members.

SELinux falls under the MAC category, by following strict enforcing settings, everything is denied and exception policies are written to give the proper permissions or access to an element of the system (either a user, a program, etc). When a program requires something that it has no need for, i.e., there is no rule defined by the developer in the security policy, it gets denied and a flag is registered in an pre-defined log entry.

SELinux has three modes of operation:

- Enforcing: The default mode where the SELinux security policy is active;
- Permissive: SELinux is active, it only issues warnings and writes to logs regarding policies violations;
- Disabled: SELinux is not running;

As previously stated, SELinux follows a least-privilege model, meaning that by default everything is denied unless an specific policy is associated with an element of the system. Implementing such a strict policy in a big system would lead to a great amount of work by the system administrator, in order to get every single process to be validated in a proper policy. To ease that process, SELinux allows different policies to be written that are interchangeable. Usually, the targeted policy is the default and covers selected system processes such as `httpd`, `dhcpd`, etc. The remaining system processes and remaining userpace programs, run in an unconfined domain, which is not covered by the SELinux protection model. The main goal was to restrict boot-running processes to a confined domain [21].

CM follows the SELinux security policy as of version 12.1.

3.2.5 NDK Tools

The Android Native Development Kit (Native Development Kit (NDK)) is a development kit that allows integration of native C and C++ code within the Android Framework.

Some applications may need to overcome java limitations such as memory management or performance, and hence, Android provides Native Development Kit to support native development in C/C++. NDK contains all the necessary tools to accomplish native code integration within the Android Java code such as compilers, libraries and header files. [22]

Also, NDK can be called from the command line, and provides the means to compile native code to be ran directly under the Android device Shell, by compiling into a binary executable file.

During the implementation stage of this project, NDK was used to port `iw` 3.2.6 with compatibility to the Nexus 4 ARM CPU as well as to compile native C applications present within the CM source code, such as WPA_Supplicant.

3.2.6 iw

`iw` [11] is the replacement for the deprecated `iwconfig`. It is the new nl80211 based Command Line Interface (CLI) configuration utility for wireless devices. As mentioned previously, nl80211 provides the means to create a Netlink communication to interact directly with cfg80211 within the Kernel. `iw` supports all the current drivers supported by the Linux kernel resources to nl80211 instead of the deprecated Wireless Extensions (Wireless Extensions (WEXT)) although it still provides support.

`iw` enables the developer to interact directly with the intended interface and was highly used during the implementation stage of this master thesis. Namely, `iw` was used to:

- Change the network card properties, namely the device type to Mesh Point as well as frequency modes and further setup alongside the `ifconfig` command;
- Create an open Mesh, via the included Mesh controls;
- Join an open Mesh;
- Find which connections are established with the node issuing the command;
- Gather information regarding the wireless network interfaces;
- Create a virtual device associated with the physical device;
- Set power management modes;
- Create an Mesh Portal alongside with `hostapd` [2];

[2] https://wireless.wiki.kernel.org/en/users/documentation/hostapd

3.2.7 WPA_Supplicant

WPA_Supplicant [14] is a program designed to run as a daemon within the system that acts as the backed component controlling the wireless connection. It is the IEEE 802.1X/WPA component used in the client stations, and is responsible for implementing key negotiation and control the authentication/association of the WLAN driver. It is, therefore, responsible for maintaining the active connection, namely secure connections. It contains several cryptographic support libraries to establish connections as needed. The open80211s project requires Secure Authentication of Equals (SAE) and WPA_PSK in order to establish a connection to the desired node, and the WPA_Supplicant version 8 has the needed native support.

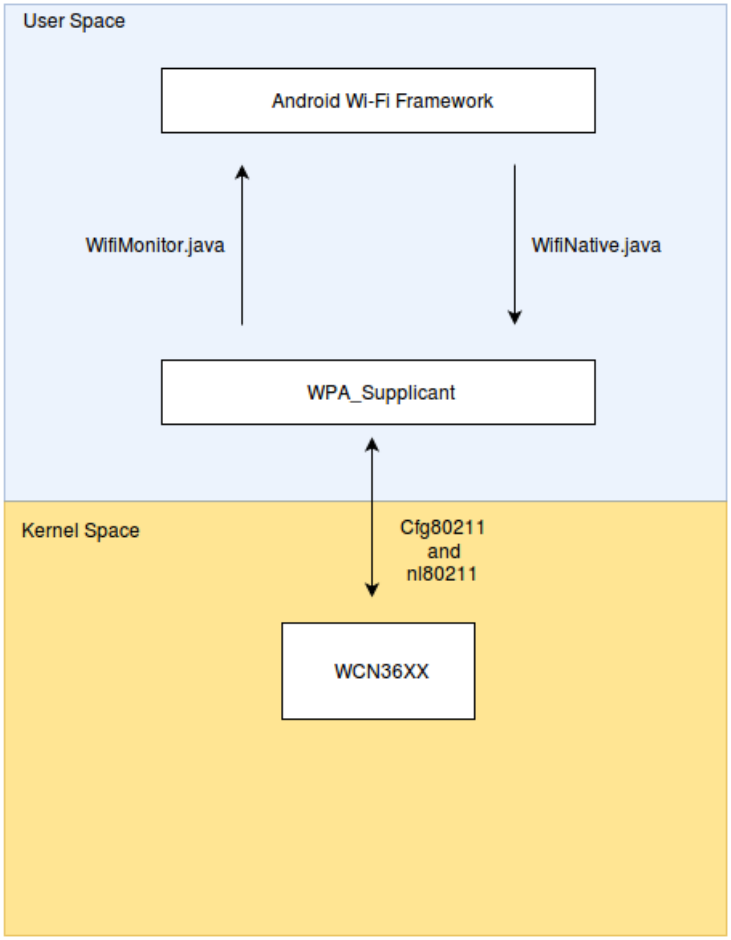


Figure 3.1: WPA_Supplicant operation method

Android and CM use WPA_Supplicant to manage the connections. The android Application Programming Interface (API) provides high level Java functions that enable a piece of software to manage its network connections as long as the permissions are correct such as the android.net.wifi [15]. Ultimately, this API will rely on the WifiNative [16] class to interact to WPA_Supplicant. Furthermore, the Android API will receive communications from the WifiMonitor class.

During the implementation stage, WPA_Supplicant was the tool used to try and create a

secure persistent Mesh network without the need of specific manual configuration by `iw`, since configuration can be loaded from a configuration file and WPA_Supplicant would be able to setup all the parameters needed to instantiate the secure Mesh. Also, by running as a daemon, WPA_Supplicant will run in the background and is not affected by power saving routines coded in Android.

Chapter 4

System Overview

In this chapter, we will analyze how the different components interact and interconnect in the Android operating system, and do a close analysis regarding the Google Nexus 4 target device. Our main focus will rely on the Wireless architecture of Android devices.

4.1 Wireless Driver Architecture

In Android, as in Linux, the user space and kernel space are divided for security. That will prevent user applications to run kernel applications or commands at the kernel level and harm the system itself or gain unauthorized access to hardware. On the other hand, the kernel layer is the only one that can communicate with the hardware layer, once again, as a security measure. But some software needs to interact between layers, and that connection is usually assured by communication sockets. In our case, we must be able to communicate with the wireless card firmware embedded in the chipset from the user space level.

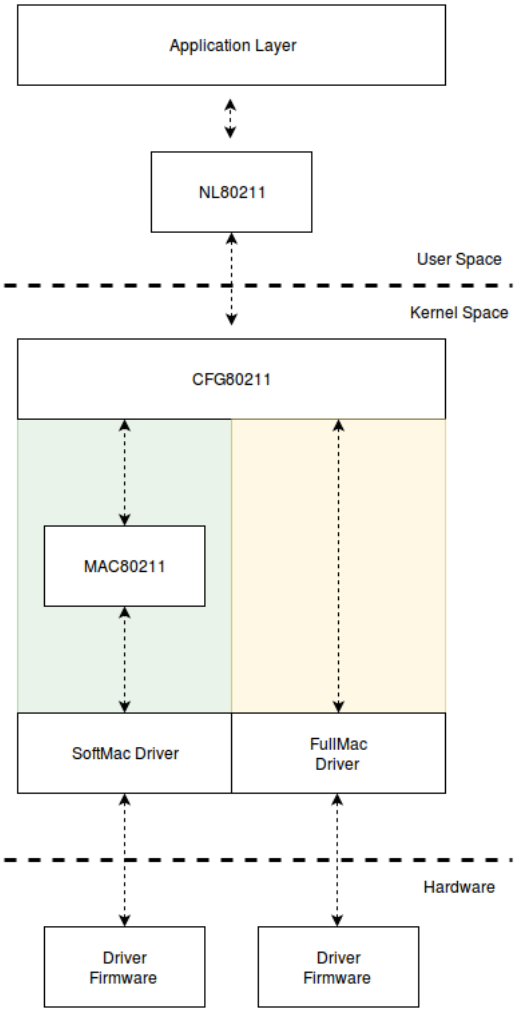


Figure 4.1: Wireless Driver operation method

When there is a need to communicate wirelessly, usually that means that there is an action being requested from the application layer. For instance, if a given user implements an "app" that relies on the Wifi API (Android.net.wifi)[1], that means that an instance of the wifi class is requesting information or issuing commands to the device supplicant.

WPA_Supplicant is present in both Linux and Android-based operating systems like Cyanogenmod (CM), and when there is the need to interact with the network interface, it is done through it. It resides in the Hardware Abstraction Layer (HAL) and when WPA_Supplicant receives a request, it must pass the instructions downward in the hierarchy. To accomplish that, it needs to communicate with the kernel space. To do so, a socket is needed to provide communication between the userspace and the kernel. NL80211 provides a netlink between the user space and the kernel space resourcing to the libnl library (netlink protocol).

But the commands need to be interpreted and passed to the drive and that is accomplished with CFG80211. CFG80211 is the new Linux 802.11 configuration Application Programming

[1]https://developer.Android.com/reference/Android/net/wifi/package-summary.html

Interface (API).CFG80211 is fairly new, and sometimes the legacy Wireless Extensions (Wireless Extensions (WEXT)) is still in use. It enables the userspace to control the driver with help from the nl80211 netlink. As stated in [27]:

> The term SoftMAC refers to a wireless network interface device (WNIC) which does not implement the Mandatory Access Control (MAC) layer in hardware, rather it expects the drivers to implement the MAC layer. [2]

If the driver is a softmac driver, it implements the MAC layer in software. This means that the necessary frame management is done through software rather than hardware. Figure 4.1 states the difference between an SoftMac and a FullMac driver. The full MAC drivers do all the frame management directly on the device chipset. It is also important to acknowledge that SoftMac and FullMac drivers can operate simultaneously.

SoftMac approaches bring some advantages:

- Potencially lower hardware costs - As the mac layer processing is done via software, allowing for the hardware to process less as tasks are delegated;
- If new standards are released, the device can maintain support as only the driver needs to be updated;
- If the MAC layer has any faults, it can be fixed easily again through driver update;

To do so, the MAC80211 module must be present within the system. The MAC80211 is a framework for driver development and represents a subsystem to the linux kernel that implements shared code for softMAC wireless devices. SoftMac drivers communicate to and from the MAC80211 module. Finally, using the headers provided by the MAC80211 framework, communication can be established to the device chipset and data be exchanged. If the used driver is a fullMac driver, MAC80211 wont be used as the MAC layer is directly implemented in the device chipset. The target device for our implementation is the Google Nexus 4 by LG. The wireless chipset is the Qualcomm atheros WCN3660 managed by the Qualcomm Atheros Prima WLAN driver. This is a fullMac driver, that does not have native Mesh support needed. WCN36XX [13] is a MAC80211 driver developed for the wcn3660 and wcn3680 chips which contains, in its features, Mesh support. Also, wcn36xx only compiles with Mobile Station Modem (Mobile Station Modem (MSM)) kernels, and Nexus 4 kernel is MSM based.

[2] http://stackoverflow.com/questions/28343384/do-access-points-use-softmac-or-hardmac

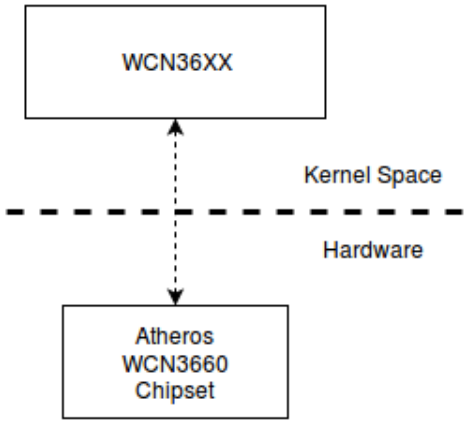


Figure 4.2: WCN36XX Softmac Driver

Taking into account the available hardware and driver, we dcided to chose WCN36XX as the driver to be installed on the Nexus 4 smartphone. We have now identified the SoftMac driver to be used in our implementation. WCN36XX will be implemented as the devices driver, linked to the WCN3660 Chipset.

To further implement, we need to identify the target Kernel for our implementation. The Cyanogenmod version 12.1 contains the Linux 3.4 Kernel Version. Version 3.4 does not contain the WCN36XX driver and, as seen in 4.2, the driver must operate within the Kernel.

WCN36XX is natively available in the Linux Kernel as of version 3.16. To accomplish our expected results, we had to compile the wcn36xx driver in order to allow for it to operate under the Linux 3.4 Kernel present in our CM distribution. To do so, the driver needed to be ported from a newer kernel to our kernel version. The backports project was created to accomplish just that. By compiling the WCN36XX driver according to the Nexus 4 ARM architecture and generating compatibility headers that compatibilize the driver onto an earlier kernel version, we can then use the WCN36XX driver within our target device.

Backports generates the appropriate kernel modules and dependencies needed to include the WCN36XX driver in the CM distribution, as well as depedencies namely mac80211, cfg80211, the compat module, and wcn36xx itself. The driver will also need the wcn36xx_msm module which is responsible for establishing a connection from the chipset firmware to the driver.

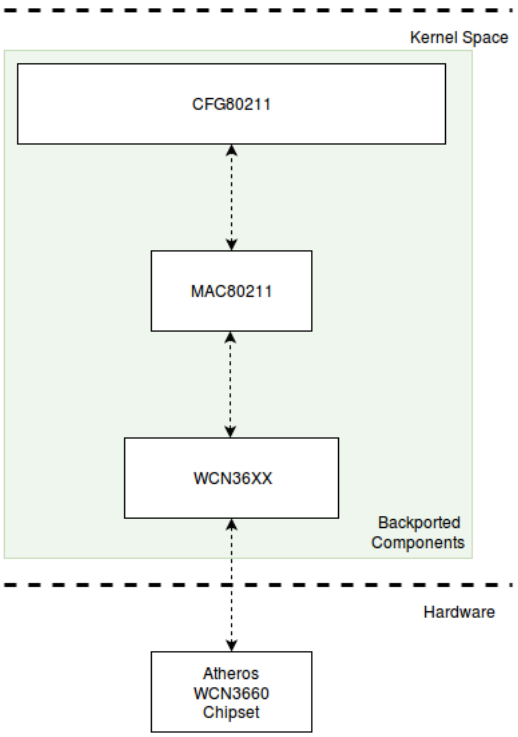


Figure 4.3: Backported WCN36XX structure

The netlink module, nl80211, is available within the 3.4 Kernel Version and will be generated once the Kernel is compiled with the CM Build. The compiled kernel modules must now integrate the CM distribution on the Nexus 4 cellphone.

We now have the grounds to port the new driver onto the target device. In the next chapter we will describe thoroughly how to compile the chosen driver, as well as how to create the needed described components needed for correct operation.

Chapter 5

Procedures

This chapter will describe the whole development process associated with the proposed solution. In order to accomplish the final result expected, we first must define the steps needed to succeed.

Firstly, we will be choosing the hardware and installing the require software to compile and develop. After that, we will define which parts need to be configured and when to apply such changes.

For this implementation, CM was the chosen operating system. As previously mentioned, CM is an open-source, community driven operating system based on Android. The open-source code is provided via repositories, and documentation is widely and freely available.

At the development stage (and in the beginning of our implementation), CM was releasing its 13.0 version. Therefore, we primarily opt by this version. However, because it was a fairly new release, we encountered several issues within the first compilation attempt. The CM 13 compilation process was accomplished successfully, but the final installable file was not running correctly on the target device. Therefore, we chose Cyanogenmod (CM) 12.1 (the immediate previous release) since it was already bug-tested and running stable in the target devices. We will provide all the sources followed, as well as all the new implementation work needed to successfully obtain support in the Nexus 4 device, by altering the source code for CM. All the solutions proposed should work properly within CM 13 as the main changes made in the previous version are compatible with the system architecture for it.

The final product is highly customized. In order to change the device driver, we must take into account other inherent changes that need to be made. The kernel will need to provide support for the new driver, as well as to disable the original one. We will have to integrate the driver in the CM source code for it to be available after compilation, add the means to interact with the driver, test and verify. Security policies within Security Enhanced Linux (SELinux) will have to be compliant and finally, the user must be able to easily deploy the new driver and its capabilities.

Since we need to interact with low level processes, we need to have special permissions. On

Android, that is accomplished by granting root privileges. To do so, we need to root the device.

Before starting, we had to identify which components must be changed and in which order.

1. Root the device
2. Make the first CM build to assure integrity and generate the output directory.
3. Give Mesh and driver support within the kernel, by enabling the necessary modules and disabling the standard driver.
4. Build the driver with backports against our serialized (specific) kernel version, integrating in the build process.
5. Enable the modules load on boot.
6. Modify the security policy to consequently enable external module loading on boot.
7. Port and integrate the necessary binary tools, namely `iw`.
8. Test and configure WPA_Supplicant.
9. Develop an user-friendly graphic tool to easily setup an open Wireless Mesh Networks (WMN).

In the next sections, we will thoroughly describe the implementation steps taken.

5.1 Rooting

Rooting refers to the capability of obtain privileged access and permissions within the operating system. In Linux, the `su` command provides root access to the system. Similarly in Android or Android subsystems, "Rooting" means obtaining Linux like "superuser" rights and permissions. With these elevated user privileges, a wide range of functionalities are made available, such as writing permissions. Rooting is a critical part of installing a custom operating system within an Android device. Without root access, we would not be able to flash the required partitions to install our CM distribution.

To attain root privileges, we resourced to the Nexus Root Toolkit software. This software will automatically provide root access and flash a new recovery partition, with a wider range of options. The recovery partition will then be managed by the TWRP [43] software, which provides an intuitive graphical user interface (Graphical User Interface (GUI)) with the available options.

5.2 Setting the environment

The development environment will be defined in two separate parts, hardware and software. In terms of hardware, a development machine running Ubuntu 14.04 Long Term Support (LTS), and a an Android development device capable of running CM - the Nexus 4 were used. As for software, we needed several software components and dependencies in order to successfully go through the compilation process. The first step, once the development machine is running on the required operating system, is to install the proper dependencies.

```
sudo apt-get install bison build-essential curl flex git gnupg
gperf libesd0-dev liblz4-tool libncurses5-dev libsdl1.2-dev
libwxgtk2.8-dev libxml2 libxml2-utils lzop maven openjdk-7-jdk
openjdk-7-jre pngcrush schedtool squashfs-tools
xsltproc zip zlib1g-dev
```

Next, create the development folder and move there.

```
mkdir Android/system
cd Android/system
```

Cyanogenmod provides its code via the `repo` command. `repo` is a script, based on Git, that collects code from various repositories and gathers it in a correct and orderly manner, necessary for the compilation process to work. CM is an operating system, meaning that it is composed by several independent pieces of software, such as kernel, drivers, applications, etc. The `repo` command will fetch all the necessary components and gather them in the "system" folder, having a manifest file as an input. This manifest file will provide `repo` with all the urls needed to download the several components. After that, `repo` will prompt for git information (name and e-mail). Finally it will be available for use.

```
1.  repo init -u https://github.com/CyanogenMod/Android.git -b cm-12.1
2.  repo sync -j 8
```

In the first command, we are providing `repo` with the necessary repository, as well as defining the branch we wish to download. In our case, we are pointing to CM github repository, where `repo` will download the base code as well as the Manifest file that will provide the source of all the repositories needed by the CM 12 branch. After that, we are instructing `repo` to start the synchronization process. This process consists on downloading and synchronizing information with the source repository. Since the development machine has an I7 Quad-core CPU, we are speeding the process by telling `repo` to launch 8 threads and do the synchronization in a parallel manner. Although 8 threads are operating simultaneously, this process can take several hours.

In our case, it took around 3 hours to finish.

Once finished, we had the CM source code available in the system folder, and ready to start the build process. We start by enabling some of the functions needed. For that purpose, we will rely on build/envsetup.sh script provided. The envsetup.sh script will add several functions to our build environment that can ease the development process [9]. The first command that needs to be issued is the `breakfast` command. `Breakfast` is used to prepare the build environment for a specific device, meaning that it will fetch the device tree and dependencies needed for the target device. In this case, we issue:

```
breakfast mako
```

"Mako" is the development branch name for the Nexus 4 device. By doing so, the device tree and its dependencies are fetched from CM GitHub repository. The Nexus 4 is an officially supported CM device and so the Mako branch is available. After that, there is a need to extract the "proprietary blobs", i.e., proprietary files specific to the nexus 4. There are hardware-specific files that are needed in order to set, for instance, CPU frequencies. That can be done by invoking the "extract-files.sh" script within the device/lge/mako folder.

Now that the environment is specifically built for the target device, we can start the first build. To accomplish that, we resource to other script provided by envsetup.sh.

```
brunch mako
```

As stated in [9], brunch sets up your build environment for a device and commences the build process of a full CM (unofficial) release that can be installed through recovery. After executing brunch, a new folder named `out` was created and the compiled final files were available. The out directory is where the final, uncompressed files are. When the build process is completed, the files residing in the out directory are compressed to form the different partitions. They are compressed yet again to create the flashable unnoficial CM file that will be flashed onto the device through recovery. Once created, the flashable zip file will be available in `/out/target/product/mako` and can be flashed into the device. Once stated that the zip file was working properly, we proceeded with the modifications.

To install our compiled zip file, we booted the device in recovery mode. To do so, we issued:

```
luis@static:~$ adb reboot recovery
```

The phone reboots and boots into recovery mode, showing the TWRP interface. We then had to:

- Wipe data partition
- Install from zip file
- Select the compiled zip file
- Clean Dalvik cache

Now that we assured that our source code is working properly, we are able to start the modifications needed.

5.3 Kernel configuration and compilation

Unlike a Linux kernel targeted for a computer, the Android kernel has its drivers statically linked into the kernel at compile time, in order to save space. That means that it is not possible to compile standalone modules and load them onto an existing Android devices. More specifically, whenever a new driver or module is developed, it must be linked to the specific serialized version of the Android kernel being used in the target device, otherwise, it will not work. In our implementation, we need to modify the network driver to support Mesh and work natively within the CM version compiled.

If we compiled a CM 12 version and used the kernel sources to generate a compatible driver and then we uploaded it to the target device, we would be able to run it after some configurations. But our goal is to provide an ready to use CM rom with native Mesh support. Therefore, we integrated the module compilation directly onto the CM build process, so it would be assured it would match and run with the target kernel. Also, the integration allows the creation of a ready-to-use flashable zip file that has immediate Mesh support through its driver.

In order to accomplish the correct driver compilation and generate the appropriate dependencies, we must configure the target kernel. The Android Kernel, as the Linux Kernel, can be customizable. We can compile a standalone driver within our development environment if we define the appropriate compilation flags, environment variables and assign the compiler for the source code. The CM repository provides the appropriate cross compiler for the kernel to be compiled onto a different architecture. This cross compiler enables the compilation of the needed kernel files with the defined specifications (flags), and customize what is linked with the kernel via a GUI.

The authors in [10] advise the use of environment variables, i.e., a set of defined values present within the system and to which the software can resort. In this case, we will export the device name as well as the source code directory (build directory). Such variables define constant values that are used throughout the compilation process. This will be useful since the defined values will be used throughout the implementation process. The command ran to access the graphical configuration is:

```
export CM_ROOT= /Android/system
export CM_BUILD=mako

make -C kernel/google/msm/
O=$CM_ROOT/out/target/product/$CM_BUILD/obj/KERNEL_OBJ
INSTALL_MOD_PATH=../../system
ARCH=arm
CROSS\_COMPILE=\$CM\_ROOT/prebuilts/gcc/linux-x86/arm/arm-eabi-4.8/bin/
arm-eabi- menuconfig
```

The command is composed by the `make` instruction, which will invoke a Makefile. Some arguments are passed, such as the directory in which the source is present (C), the output directory (O), the module installation path (INSTALL_MOD_PATH), the target architecture (ARCH) and the pointer to the desired cross compiler. Finally, menuconfig is the argument that will instantiate the graphical environment.

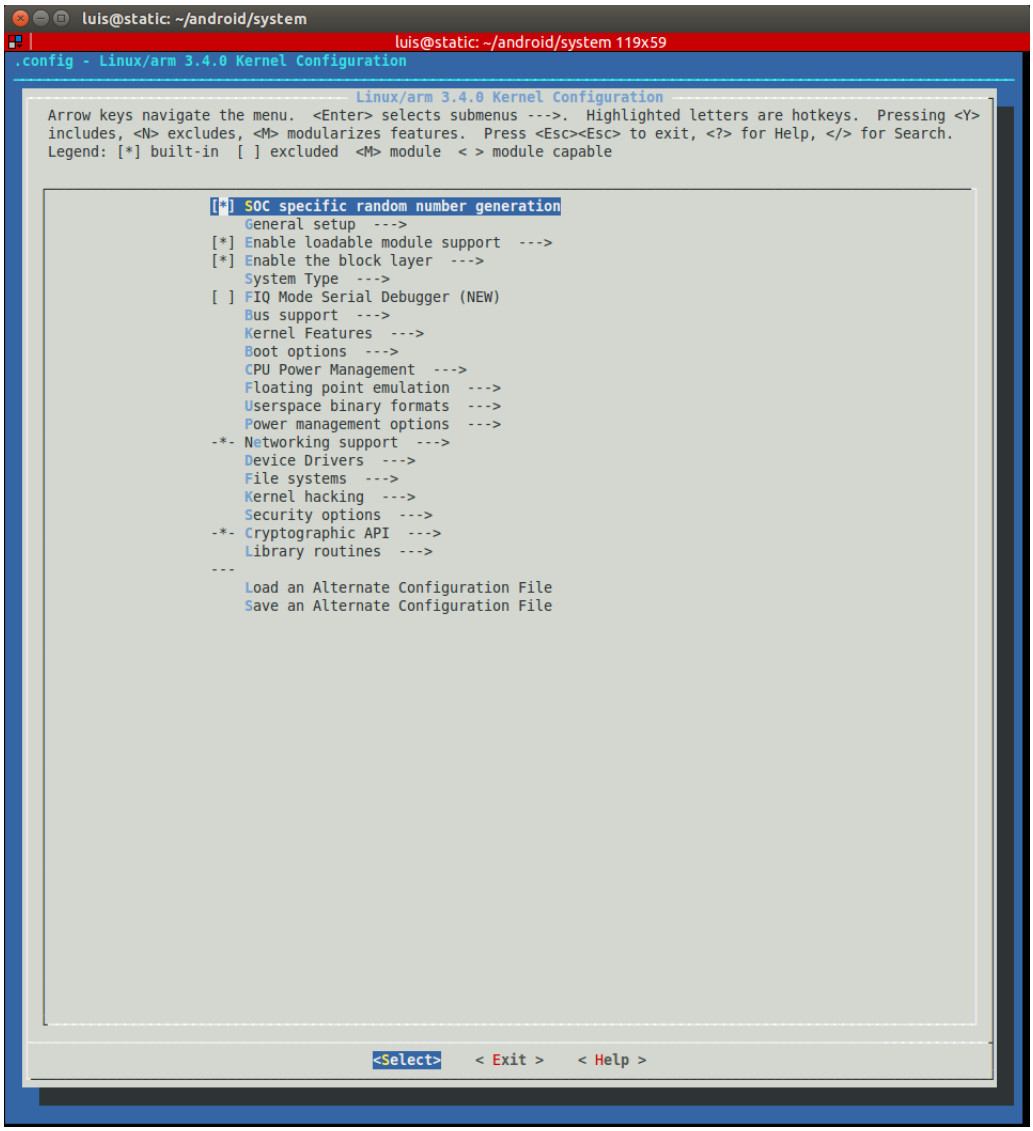


Figure 5.1: Backported WCN36XX structure

We must then make the following changes.

- Enable loadable module support and all sub-menu entries [*];
- Enable CCM support under Cryptographic (AS MODULE [M]);
- Disable cfg80211 - wireless configuration under Networking support /Wireless;
- Disable PRIMA-WLAN under Device drivers / Staging drivers / Qualcomm Atheros Prima WLAN module;

The first option will enable the use of external modules, as is the case of our driver, alongside the kernel and enabling the module linking. Crypto CCM is the cryptographic module used in

WPA-PSK and Secure authentication of equals Secure Authentication of Equals (SAE) protocol, providing the necessary functions to assure secure connection. Hence, it must be loaded.

Cfg80211 will be provided by the driver compilation since it is associated with wcn36xx. Finally, the default driver must be disabled as it will be replaced by wcn36xx.

When the menuconfig finishes, it generates a configuration file where the compilation flags for the kernel are properly defined. This configuration file will be placed in the previously defined output directory ("O" flag). It is possible to compile only the kernel and place it on the target folder, but in order to automatize the process, we must place the configuration file in the correct location in the source directory. That will enable that each time the source is compiled, the default configuration will be the one we defined.

The source documentation did not produce the desired output, as the kernel would remain unchanged. We found that the kernel configuration file was defined within the device/lge/-mako/BoardConfig.mk file and noticed that the default configuration file was set for TARGET_KERNEL_CONFIG to be cyanogen_mako_defconfig as opposed to the source, which stated that we should copy the modified configuration file with the name "mako_defconfig".

After that change was done, the source compiled successfully and a flashable zip file was produced with the desired kernel.

The following step was to compile the wcn36xx driver and link it to the kernel. Backports was the chosen tool to compile the new driver. This receives as an input the folder that contains the zImage file, which is a compressed version of the kernel image that is self-extracting. Backports will then, gather information regarding the specific kernel and automatically matches the driver to it. According to the provided documentation, we will also need a specific wcn36x_msm module. The backports version we used was the official 3.16 version, as it already provides wcn36xx support, but it did now provide the wcn36xx_msm driver. After some research, we found a version of wcn36xx source that produced the proper wcn36xx_msm module from [31].

The used sources [10][13] only provided a solution that would enable the final user to compile the driver and then manually upload it to the target device. As it was previously mentioned, kernel drivers and modules are statically linked with the kernel. That means that a given module or driver must be compiled to match the specific driver compilation made. Consequently, even if we use the same kernel version used for the final device, a driver would not work in it as it must be built for the specific kernel running on that specific device. Our solution integrates the driver compilation process within the CM build process, and automatically places the compiled files in the correct folder in the output directory so they can then be compressed and used in the target device.

As mentioned in 3.2.3, the CM build process relies on a hierarchical tree of Makefiles. When the brunch command is issued, the build/core/main.mk file is called. Main.mk will then call other makefiles and so on.

Our first approach was to create a local folder and a Makefile that would be called by the CM build process. But creating a makefile to be integrated within an complex build process as cyanogenmods is very laborous. Since our final result is a specific CM version with Mesh support, we hard coded the backports build within the compilation process. To do so, we analyzed the output of the build process and we found a “Building Kernel” string. We then searched for files containing such string:

```
grep -r "Building Kernel"
build/core/tasks/kernel.mk:@echo e ${CL_GRN}” Building Kernel ”${CL_RST}
```

Makefiles define several values like temporary environment variables and rules. Since Backports will rely on the zImage file, and we know that the zImage file is produced in the output directory for the kernel, we opted to search for the zImage string, and found :

```
TARGET_PREBUILT_INT_KERNEL_TYPE := zImage
```

After that, we searched for the TARGET_PREBUILT_INT_KERNEL_TYPE variable, and found

```
$(MAKE) $(MAKE_FLAGS) -C $(KERNEL_SRC) O=$(KERNEL_OUT)
ARCH=$(KERNEL_ARCH)
$(KERNEL_CROSS_COMPILE) $(TARGET_PREBUILT_INT_KERNEL_TYPE)
```

Which is the same command we previously found for compiling the kernel and hence, the line that will effectively call the kernel compilation process. So we uploaded the backports and wcn36xx_msm to the "external" folder in a new folder called "wcn36xx". Backports will need to be configured prior to the compilation. Since the build process will automatically call the backports makefile, we can setup the configuration file prior to compilation hence leaving backports ready to be ran. In the "external/wcn36xx/backports" folder run:

```
make wcn36xx-defconfig
make menuconfig
```

Listing 5.1: Driver compile steps

Enable pre-80211s support under network / wireless [*].

```
make KLIB=$CM_ROOT/out/target/product/$CM_BUILD/obj/KERNEL_OBJ
KLIB_BUILD=$CM_ROOT/out/target/product/$CM_BUILD/obj/KERNEL_OBJ
ARCH=arm
CROSS_COMPILE=$CM_ROOT/prebuilts/gcc/linux-x86/arm/arm-eabi-4.8/
```

bin/arm-eabi

The KLIB and KLIB_BUILD parameters point to the compiled kernel source directory, where the zImage relies. When we save the changes made, a ".config" file will be generated in the backports folder. When "make" is called further in the compilation process, it will resource to the .config file defined.

```
          echo "Kernel Modules not enabled" ; \
       fi ;
@echo -e \${CL_GRN}"Building wcn36xx kernel objects matching against
    "${CL_RST}
$(MAKE) -C $(Android_BUILD_TOP)/external/wcn36xx/backports
    KLIB=$(KERNEL_OUT) KLIB_BUILD=$(KERNEL_OUT) ARCH=$(KERNEL_ARCH)
    $(KERNEL_CROSS_COMPILE)

$(MAKE) -C $(Android_BUILD_TOP)/external/wcn36xx/wcn36xx-master/wcn36xx_msm
    KLIB=$(KERNEL_OUT) KLIB_BUILD=$(KERNEL_OUT) ARCH=$(KERNEL_ARCH)
    $(KERNEL_CROSS_COMPILE)

cp $(Android_BUILD_TOP)/external/wcn36xx/wcn36xx-master/
    wcn36xx_msm/wcn36xx_msm.ko $(OUT)/system/lib/ modules/wcn36xx_msm.ko

cp $(Android_BUILD_TOP)/external/wcn36xx/backports/compat/compat.ko
    \textpf{$(OUT)}/system/lib/modules/

cp $(Android_BUILD_TOP)/external/wcn36xx/backports/net/wireless/cfg80211.ko
    $(OUT)/system/lib/modules/

cp $(Android_BUILD_TOP)/external/wcn36xx/backports/net/mac80211/mac80211.ko
    $(OUT)/system/lib/modules/

cp $(Android_BUILD_TOP)/external/wcn36xx/backports/drivers/net/wireless/
    ath/wcn36xx/wcn36xx.ko $(OUT)/system/lib/modules/
```

Listing 5.2: Kernel Compilation Integration

With this we are forcing the kernel to generate the new modules, install them onto the proper location and then commencing the wcn36xx build process. Then, we compile the wcn36xx driver, the wcn36xx_msm module, and then copy the files onto the output directory, in the proper folder to be later loaded.

We implemented a solution that automatically runs backports immediately after the kernel is compiled. By integrating the backports compilation process directly onto the CM build, we can assure that the kernel will be compiled first, and immediately after it, the driver.

5.4 OnBoot Load

Once the driver is correctly compiled and working alongside the kernel, we implemented a solution to enable the driver loading in the boot process of CM itself. As Linux, CM has init files that are ran on boot, where configurations and procedures can be implemented. Usually, we would find the init.rc file in Linux. In our case, CM provides the init.rc and init.rako.rc files.

The init.rc file has its own language, the init language. The init language is used in plaintext files that take the .rc file extension. As stated in [17], init.rc is the primary .rc file and is loaded by the init executable at the beginning of the boot process. It is responsible for the initial set up of the system.

The init language allows for the creation of services or actions.

> "Services are programs which init launches and (optionally) restarts when they exit.[1]"

So we need to define a service, which will load our drivers and modules on boot.

> "All services whose binaries reside on the system, vendor, or odm partitions should have their service entries placed into a corresponding init .rc file, located in the /etc/init/ directory of the partition where they reside.[2]"

A driver or module can be manually loaded in Linux based operating systems via the use of the `insmod` command. So, we defined a service which would call a shell script. That shell script would resource to `insmod` to load all the necessary drivers.

```
service wcn36xx /system/bin/sh /system/etc/init.mako.wcn36xx.sh
    class main
    user root
    oneshot
```

Listing 5.3: Service entry creation

and the init.mako.wcn36xx.sh script

we also added a new line in device.mk for the new shell script to be copied to the proper location on the target system upon build

We flashed the device, and tested the implementation. What we found out was that the modules were not beeing loaded at all. The script was in fact, being ran, but the instructions within it were not working properly. If the script was manually ran, it would succeed, but on

[1] https://Android.googlesource.com/platform/system/core/+/Android-5.0.0$_r$2/*init/readme.txt*
[2] https://Android.googlesource.com/platform/system/core/+/Android-5.0.0$_r$2/*init/readme.txt*

```
!/system/bin/sh

/system/bin/insmod /system/lib/modules/wcn36xx\_msm.ko
/system/bin/insmod /system/lib/modules/compat.ko
/system/bin/insmod /system/lib/modules/cfg80211.ko
/system/bin/insmod /system/lib/modules/mac80211.ko
/system/bin/insmod /system/lib/modules/wcn36xx.ko
```

Listing 5.4: Init script configuration

```
PRODUCT_COPY_FILES += \
        device/lge/mako/init.mako.bt.sh:system/etc/init.mako.bt.sh \
        device/lge/mako/init.mako.wcn36xx.sh:system/etc/init.mako.wcn36xx.sh
```

Listing 5.5: Device.mk configuration file

boot it would fail. After several debug attempts, we found log entries denying the sys_module capability.

After research, we found that there was a security block affecting our script, more properly, the block was being done to the insmod command by SELinux. Basically, when the cellphone boots, it does not operate in the user space. Only after the boot process is completed, the concept switch to user space is done. So the init_shell process was being denied the sys_module capability, and was not being able to insert the kernel modules.

Our script was developed to run on boot, so it would be ran in by a different user, in this case the init_shell process.

We tested two different approaches. One was to disable SELinux, by setting it to work in a permissive manner, where logs are written but security policies are not enforced. That could be done by adding BOARD_KERNEL_CMDLINE += Androidboot.selinux=permissive in device/lge/mako/BoardConfig.mk. But disabling the SELinux secure policy is not a good approach as it disables all the security levels implemented and enables malicious code to operate on boot, if the user gets infected.

The proper way was to define a specific policy to our needs, that would not require more than the operations required to enable wcn36xx to function correctly. In order to help defining the rules needed to be written, we resourced to the audit2allow tool. audit2allow scans the logs and traces SELinux denied log entries. It then produces the appropriate rule to be added to the SELinux policy.

So we ran and it produced

so within the device/lge/mako/sepolicy folder, we created the wcn36xx.se file and added the content provided by audit2allow.

```
adb shell su root dmesg | audit2allow -p out/target/product/mako/root/sepolicy
```

Listing 5.6: Audit2Allow Debug

```
allow init_shell self:capability sys_module;
```

Listing 5.7: Audit2Allow Output

and added the following to file_contexts

```
/system/etc/init\.mako\.wcn36xx\.sh        u:object_r:wcn36xx_exec:s0
```

Listing 5.8: wcn36xx file_contexts entry

The graphical menus in CM are not prepared to work with WMN, hence, it was impossible to test our implementation using the standard GUI. Throughout the process of compiling and uploading the drivers, `iw` was the chosen piece of software to manage interfaces and attempt the creation of a Mesh network. Although present in most of the Linux distributions, it is not present natively in Android or CM. So we needed to port `iw` to the Android system. Binary executable files can be ported to Android with help from `NDK tools`. `iw` runs on the linux command line. Since we are using Android Debug Bridge (ADB), and the local shell within the device, `iw` suits properly within what we needed.

To port `iw` onto Android, we resourced to [24]. We actually found that this source code is not able to compile as the documentation states. When NDK is called, it will read instructions from tool_chain/jni/Application.mk file, namely the target architecture. The target architecture is, by default, set to "all", and is defined in the APP_ABI flag. In order to go through the compilation process without error, we needed to change the APP_ABI flag to

```
APP_ABI := armeabi
```

Listing 5.9: iw Makefile configuration

So, in order to successfully compile `iw`, we had to do some changes to the recipe.

We then finally have an Android-ARM compiled iw binary. Since iw relies in on the netlink protocol library, it also produces the necessary headers in the form of shared objects (.so files). The CM build was also updated with iw, we uploaded the shared objects to system/lib/modules in the output directory, and the `iw` binary executable file to system/bin. That assures that `iw` will be compressed in the target flashable zip file.

After that, the device is now capable of creating and joining Mesh networks.

```
git clone https://github.com/imlinhao/Android-iw-libnl3.git

cd Android-iw-libnl3/
sh .autogen.sh

cd Android_toolchain/

*/apply the previously mentioned changes */
ndk-build
```

Listing 5.10: iw build process

Chapter 6

Results

In this chapter we will describe the results obtained by the aforementioned development and investigation steps. We will start by briefly explaining how to upload and install the custom ROM and will, hence, describe the features enabled by the implementation of the mesh capabilities. We will describe such features in terms of GUI capabilities available for the user as well as embedded capabilities now available within the system itself.

We then describe procedures that will enable access/guide researchers and developers towards the IEEE 802.11s properties by resourcing to low level tools such as iw.

After, we will access the results in terms of the final developed ROM, as well as briefly describe the concept mesh APK developed to enhance the IEEE 802.11s protocol at a higher, GUI, level.

Finally, we will describe some results obtained by implementing test mesh networks and and gathering results, such as distance obtained, relay node to amplify range, and running services resourcing to a IEEE 802.11s active connection.

6.1 Preliminary Results

As a result of all the previously described development steps, we were able to do a preliminary testing of the Mesh capabilities in the Nexus 4 cellphone. To do so, we uploaded the compiled flashable zip file from the output directory to the device using adb push function.

```
adb push $OUT/cm_12_final.zip /sdcard
adb reboot recovery
```

Listing 6.1: Remote file upload

Once recovery booted into TWRP, we formated the data, cache and system partitions, and selected "install from zip file" option. After selecting our zip file, the installation process went

through. Once finished, we wiped the dalvik cache and rebooted the phone.

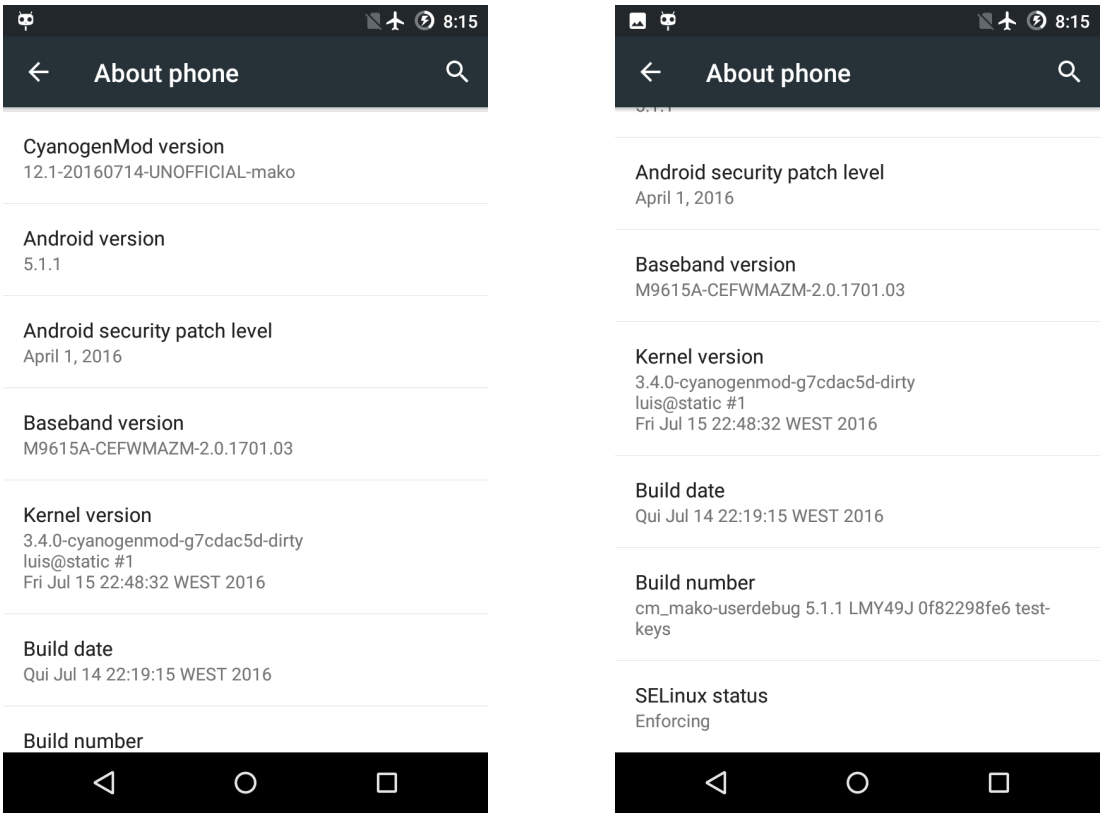


Figure 6.1: About phone menu entry

Figure 6.1 shows the settings menu from the compiled rom, where we can state the kernel version, compiled by us, as well as the Security Enhanced Linux (SELinux) status defined to enforcing. CM booted normally, available wireless Access Point (AP)s and at first we could not connect to secure networks as there was authentication problems associated with the AP association process.

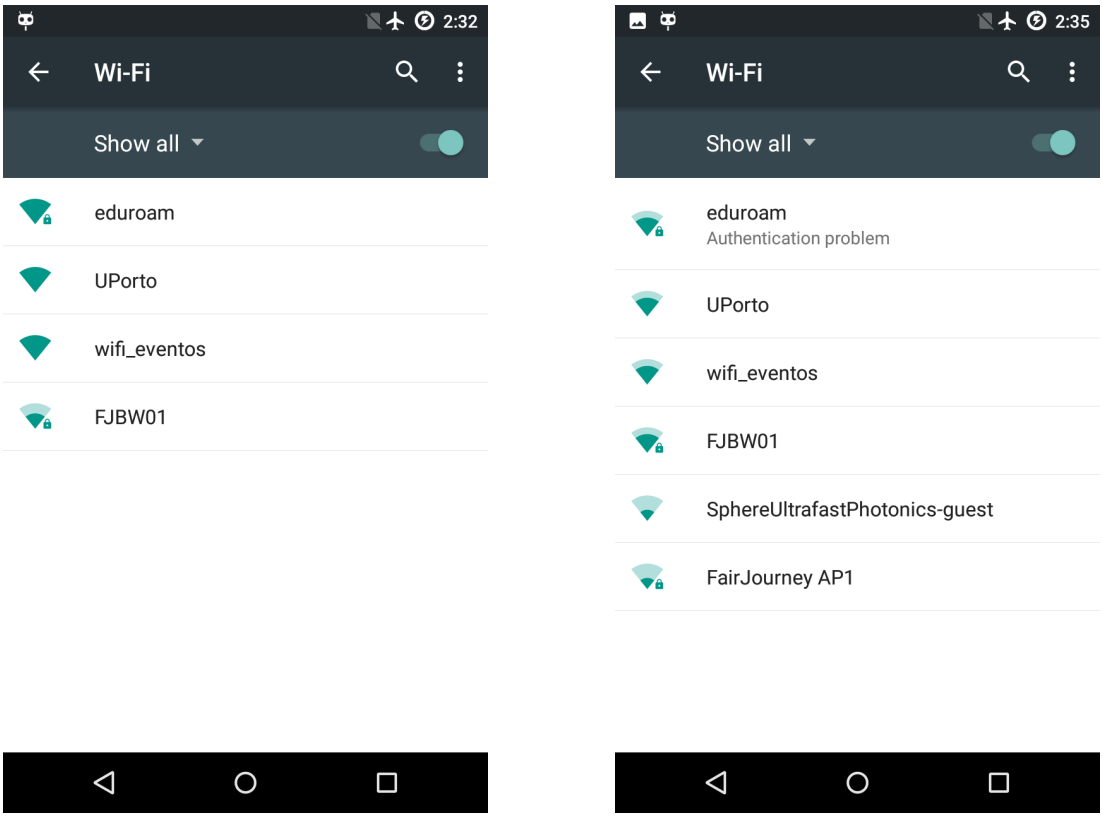


Figure 6.2: Wireless Scanning GUI

The authentication problem ended up being traced to the lack of cryptographic support. The problem was resolved by forcing the kernel to compile the ccm module as an external module and load it on boot as well.

We then proceeded to the Mesh creation, via iw. To do so, we enabled the hidden Developer Options menu by tapping "Build Number" under Settings / About phone, and the Android Debugging under the Developer Options menu. The phone is then available via terminal due to the adb shell.

Once inside the local shell, we needed to do some configuration. Firstly, we need to set the interface type to Mesh Point (MP). By default, the interface will operate as managed meaning that it is beeing managed by a process or daemon, in this case, WPA_Supplicant is managing the interface. At this point, WPA_Supplicant did not have native support for Mesh networks, hence, the interface could not be operating as managed as it would invalidate the Mesh capabilities. In order to avoid WPA_Supplicant managing the interface, we disabled wireless in the phone (Turn Wifi off option under Settings). We manually setup the interface configurations, creating the Mesh point, and enabling the interface to operate in accordance with the Mesh network requirements. To do so, we had to first disable the interface, then manually assign the Mesh type and a corresponding Internet Protocol (IP). For testing purposes, we also manually assigned a Mesh id, creating an open Mesh network.

```
ifconfig wlan0 down
iw dev wlan0 set type mp
iw dev set meshid mesh
ifconfig wlan0 192.168.1.10 up
```

Listing 6.2: Wireless interface configuration

This will configure the interface to operate as a Mesh point, with all the configurations. Also, an open Mesh (unprotected) is created with the Mesh id "mesh". The second Nexus 4 used to perform tests was also configured to operate as a Mesh point, and assigned an IP in the same sub-net. We stated that it was possible to scan the Mesh network within the Graphical User Interface (GUI).

Connection will not be possible via the GUI at this point. We suspect that the main reason is related with the lack of native support for Mesh operations within the Android framework and the provided classes.

Although not possible via GUI, it is possible to manually join the second device with the first one, and it can be done with one of two methods:

1. Setting the meshid parameter to match the open mesh created.
2. Explicitly joining the device via iw dev <devname> mesh join <mesh ID>

In the first method, the mesh_id parameter is matched with the existing Mesh network. If they match, and since the Mesh network is open, it will automatically peer with one device and authenticate in the network. Normally, when a device joins, it should announce its membership. Apparently, this is not case in our implementation, but if we force communication by pinging one device, it will automatically update the Address Resolution Protocol (ARP) table, and the device will show his peer links.

6.2 Mesh Application

Due to the lack of support from higher level classes that manage the network connection, the GUI was unable to setup a Mesh network in an user-friendly manner. The manual process described in the previous chapters does not match the main purpose of ou implementation as it will require special knowledge from the user and deployment will be time-consuming.

As previously mentioned, to operate in Mesh mode, we need to interact directly with the network interface. That requires root privileges, and we are operating at user level. To gain root operation access we resourced to the chainfire SuperSU library for Android. That enabled us to run shell commands within our application an properly configure the network interface. The APK was designed to be easily usable in testing and creating open WMN.

We could then create an open Mesh, with the click of a button.

6.3 Secure Mesh with WPA_Supplicant

The author in [10] propose the creation of Mesh networks resourcing to authsae. Authsae [36] is an application defined to handle the key derivation and cipher negotiation within Mesh devices. Authsae would have to be manually compiled and added to the rom. Moreover, to accomplish secure Mesh, the authors rely on a specific WPA_supplicant development branch created for Mesh networks.

We decided not to take the authors approach. When the open80211s with wcn36xx project was abandoned, the WPA_Supplicant implementation was still in an experimental stage. We discovered that the most recent WPA_Supplicant version available for CM13 within the CM data repository, at the time of this implementation, was already Mesh compliant and that the CM source already had native support for the Secure Authentication of Equals (SAE) protocol.

Our solution was then to integrate WPA_Supplicant latest version available for Cyanogenmod (CM) 13 in our CM 12.1 implementation.

In order to do so, we replaced the WPA_Supplicant_8 folder content in the "external" folder, taking advantage of the fact that the folder name was the same in both versions of CM and that the build process was going to run the makefile in that folder.

The new version of WPA_Supplicant is no longer relying on OpenSSL [12], running on the new BoringSSL [7] cryptographic library. We compiled the new version of WPA_Supplicant in order to resource to OpenSSL instead of BoringSSL. To do so, we had to change the WPA_Supplicant makefile, adding INCLUDES += external/openssl/include and forcing the value of CONFIG_FIPS = y in order to force the makefile to compile with OpenSSL. Also, we added the CONFIG_MESH=y in both wpa_supplicant and hostapd Android.mk

When WPA_Supplicant successfully compiled, we were able to create open Mesh and secure Mesh with a custom WPA_Supplicant file, but only in an temporary manner. Since we adapted the WPA_Supplicant to support OpenSSL, but it was originally conceived to work with BoringSSL, we came accross a scenario, where a Mesh network would be created, would operate normally for a few seconds and then an Segmentation Fault would occur (SIGSEGV 11). We traced the segmentation fault to the cryptographic Application Programming Interface (API) in use.

With the work developed, we produced two major results.

Firstly, we ported a dropped 2010 project and ported its implementation to current technology. We made it all automatizing the most within the rom compilation process itself, assuring that future developers can download our work and improve it. We produced an working CM 12.1 rom, for nexus 4 devices, capable of creating, joining and communicating with other devices in a

more seamlessly and less time consuming manner.

Secondly, we released every implementation step that we did, duly commented, to developers that are working with CM driver integration, CM customization. By releasing the full project in the form of a webpage, as well as all the source code in an data repository, we provide users with a source on which they can rely to either replicate or adapt to other devices, to reproduce the same outcome we managed to obtain with the work done.

6.4 CyanogenMod Rom

The CM rom does not suffer any change in terms of usability. All the functionalities that are available in the official CM 12.1 release are still available to the target user. But now it provides open Mesh support, enabling the user to connect seamlessly into an established WMN if available, or deploy a Mesh network of its own.

The technology integration enabled the GUI to operate in Mesh compliance. The main proof relies on the fact that the Wifi graphical interface can now list available Mesh networks. Although connection is not yet possible through this menu, it shows that the driver is interacting with higher-layer applications and collecting 802.11s packets, partially parsing their content.

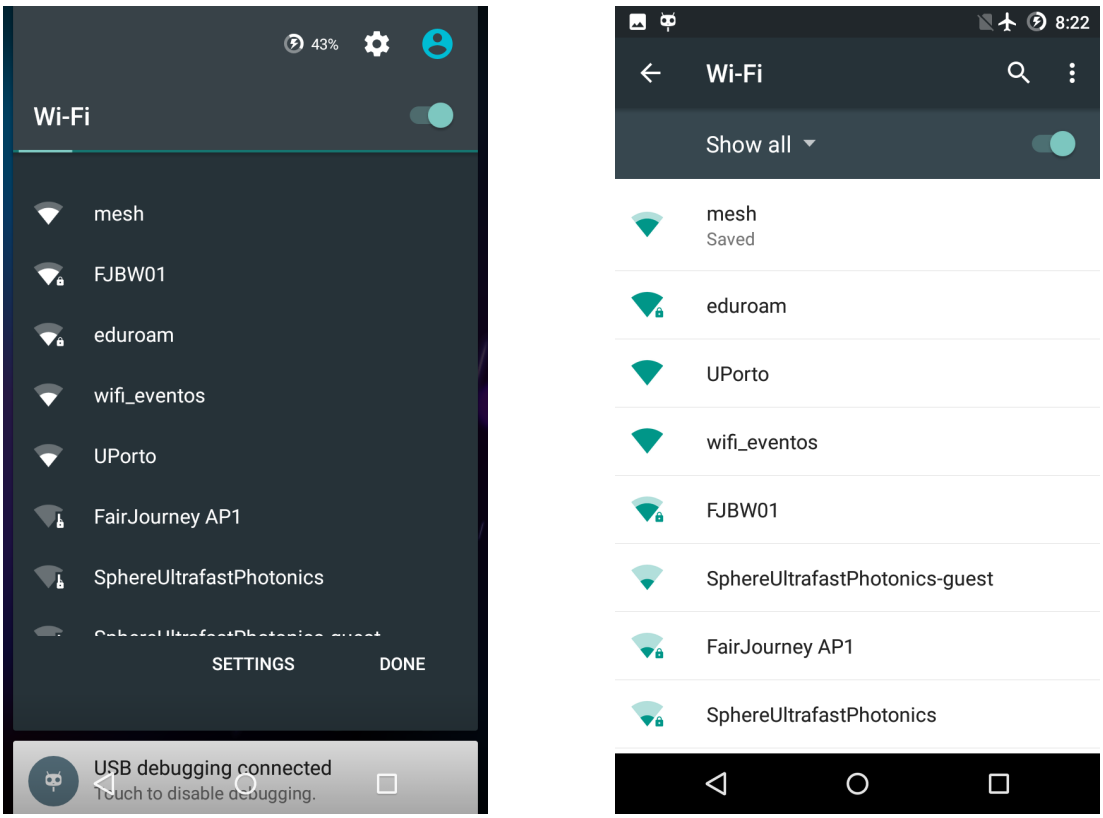


Figure 6.3: About phone menu entry

Although managed by WPA_Supplicant, the interface is also working accordingly. By

explicitly issuing a scan command, the interface is able to pick up the MESH ID field in the packets received and display them to the user.

```
237|root@mako:/ iw scan | grep -i id

   SSID:
   MESH ID: mesh
        * Active Path Selection Protocol ID: 1
        * Active Path Selection Metric ID: 1
        * Congestion Control Mode ID: 0
        * Synchronization Method ID: 1
        * Authentication Protocol ID: 0
}
```

Listing 6.3: iw mesh scan

It is now possible to define the interface type as a Mesh point, and set Mesh parametres such as Mesh id and issue commands such as Mesh join.

```
237|root@mako:/ iw dev wlan0 info
Interface wlan0
   ifindex 22
   wdev 0x1
   addr 00:0a:f5:dd:58:c0
   type mesh point
   wiphy 0
   channel 1 (2412 MHz), width: 20 MHz (no HT), center1: 2412 MHz
```

Listing 6.4: Mesh peering

6.4.1 Setting a MBSS

The devices are able to establish links and communicate with each other. We tested communication between two nexus 4 devices. The first device was assigned the IP 192.168.1.10 and a Mesh network was created with the Mesh id "mesh". We ran tcpdump -i wlan0, waiting for recieving packages. The second device was assigned the ip 192.168.1.20 and the mesh join command was issued. After that, a small communication was established by having once device ping the other.

```
root@mako:/ ping 192.168.1.10
PING 192.168.1.10 (192.168.1.10) 56(84) bytes of data.
64 bytes from 192.168.1.10: icmp\_seq=1 ttl=64 time=4.60 ms
64 bytes from 192.168.1.10: icmp\_seq=2 ttl=64 time=1.92 ms
64 bytes from 192.168.1.10: icmp\_seq=3 ttl=64 time=1.92 ms
64 bytes from 192.168.1.10: icmp\_seq=4 ttl=64 time=1.89 ms
64 bytes from 192.168.1.10: icmp\_seq=5 ttl=64 time=1.92 ms
64 bytes from 192.168.1.10: icmp\_seq=6 ttl=64 time=5.95 ms
64 bytes from 192.168.1.10: icmp\_seq=7 ttl=64 time=1.95 ms
64 bytes from 192.168.1.10: icmp\_seq=8 ttl=64 time=2.28 ms
64 bytes from 192.168.1.10: icmp\_seq=9 ttl=64 time=2.31 ms

--- 192.168.1.10 ping statistics ---
9 packets transmitted, 9 received, 0\% packet loss, time 8011ms
rtt min/avg/max/mdev = 1.892/2.753/5.951/1.395 ms
```

Listing 6.5: Ping output from emitting device

Data received in the first device.

```
23:18:50.917318 IP 192.168.1.20 > 192.168.1.10: ICMP echo request, id 5, seq 1,
    length 64
23:18:50.917714 IP 192.168.1.10 > 192.168.1.20: ICMP echo reply, id 5, seq 1,
    length 64
23:18:51.916127 IP 192.168.1.20 > 192.168.1.10: ICMP echo request, id 5, seq 2,
    length 64
23:18:51.916341 IP 192.168.1.10 > 192.168.1.20: ICMP echo reply, id 5, seq 2,
    length 64
23:18:52.917562 IP 192.168.1.20 > 192.168.1.10: ICMP echo request, id 5, seq 3,
    length 64
23:18:52.917775 IP 192.168.1.10 > 192.168.1.20: ICMP echo reply, id 5, seq 3,
    length 64
23:18:53.918965 IP 192.168.1.20 > 192.168.1.10: ICMP echo request, id 5, seq 4,
    length 64
23:18:53.919179 IP 192.168.1.10 > 192.168.1.20: ICMP echo reply, id 5, seq 4,
    length 64
23:18:54.920369 IP 192.168.1.20 > 192.168.1.10: ICMP echo request, id 5, seq 5,
    length 64
23:18:54.920583 IP 192.168.1.10 > 192.168.1.20: ICMP echo reply, id 5, seq 5,
    length 64
23:18:55.920583 ARP, Request who-has 192.168.1.20 tell 192.168.1.10, length 28
23:18:55.924886 IP 192.168.1.20 > 192.168.1.10: ICMP echo request, id 5, seq 6,
    length 64
}
```

Listing 6.6: Ping output from receiving device

We can also see an Address Resolution Protocol (ARP) request being sent proving that the network protocol is operating.

Once the devices join the same Mesh network, we can verify that the link was created and gather statistics.

6.4.2 Range test

We also conducted a small test to assess the distance capabilities of the WMN created, as well as the interoperability between different devices, using different wireless drivers. In order to accomplish a wider WMN, we resourced to:

- Two Nexus 4 cellphones with the custom rom
- An asus EEE-pc using a TP-Link TL-WN722N usb wireless card with the atk9k driver

The first test consisted of an indoor test, in the computer science department of the faculty of sciences of university of Porto. We set up an WMN, connecting the computer and one of

```
iw dev wlan0 station dump \\
Station 00:0a:f5:0a:39:6f (on wlan0)\\
    inactive time:   520 ms \\
   rx bytes:   15865 \\
   rx packets: 393 \\
   tx bytes:   704
   tx packets: 8
   tx retries: 0
   tx failed:  0
   signal:     -33 dBm
   signal avg: -33 dBm
   Toffset: 194560390 us
   tx bitrate: 1.0 MBit/s
   rx bitrate: 2.0 MBit/s
   mesh llid:  1354
   mesh plid:  1548
   mesh plink: ESTAB
   mesh local PS mode:  ACTIVE
   mesh peer PS mode:   ACTIVE
   mesh non-peer PS mode:  ACTIVE
   authorized: yes
   authenticated: yes
   preamble:   long
   WMM/WME: yes
   MFP:     no
   TDLS peer:  no
```

Listing 6.7: Interface link statistics

the Nexus cellphones. The computer was left running tcp dump and the cellphone was pinging the computer. We then proceeded to move the cellphone trough a hall, which contained several rooms full of computers, wireless APs, people walking, cellphone devices operating and a partially open wooden door. We managed to maintain connection for approximately 35 meters, without data loss. We registered that location and added the second nexus 4 cellphone to the mesh network. That node automatically established a link between the two intervinients and data started to flow, automatically relaying the packets between them. We managed to communicate for more 42 meters (approximately). Between the relay node and the transmitting node, we had another hall, with other networks operating, researchers labs and an anti-fire steel door.

The second test was an outside, straight-line point-of-view test. Again, the Asus EEE-pc was registering received packets and one nexus cellphone was registering. We managed to maintain connection for around 110 meters until we reached an rock wall. We managed to maintain connection even further, but the rock wall was placing an interference, invalidating our test.

6.4.3 Running a service over Mesh

We tested the ability to run a service using our Mesh link and maintain an active connection. In this case, we successfully established a SSH connection to a computer, using one of the intervening Nexus 4.

```
root@mako:/ ssh 192.168.10.3

The authenticity of host 192.168.10.3 (192.168.10.3) cant be established.
ECDSA key fingerprint is 42:7c:e5:21:29:91:13:14:3e:d1:d3:a1:f1:51:90:6c.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.10.3' (ECDSA) to the list of known hosts.
root@192.168.10.3's password:
root@mako:/  ssh 192.168.10.3
root@192.168.10.3's password:

Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 4.2.0-42-generic x86_64)

* Documentation: https://help.ubuntu.com/

58 packages can be updated.
32 updates are security updates.

WARNING: Security updates for your current Hardware Enablement
Stack ended on 2016-08-04:
 * http://wiki.ubuntu.com/1404_HWE_EOL

 There is a graphics stack installed on this system. An upgrade to a
configuration supported for the full lifetime of the LTS will become
available on 2016-07-21 and can be installed by running 'update-manager'
in the Dash.

Last login: Mon Aug 15 21:11:02 2016 from 192.168.10.20
root@pc:~
```

Listing 6.8: SSH connection through Mesh link

6.5 Mesh Settings APK

The Mesh settings APK is a small application developed to ease the creation of a WMN. It provides a simple interface, with text boxes for IP, Mac Address and Mesh Id. This input will then be used in issuing commands within the SuperSU library. Interface management is done using the switch button, that will change the interface type. By using the SuperSU library, we provide input parameters to `iw` to create or join an open Mesh.

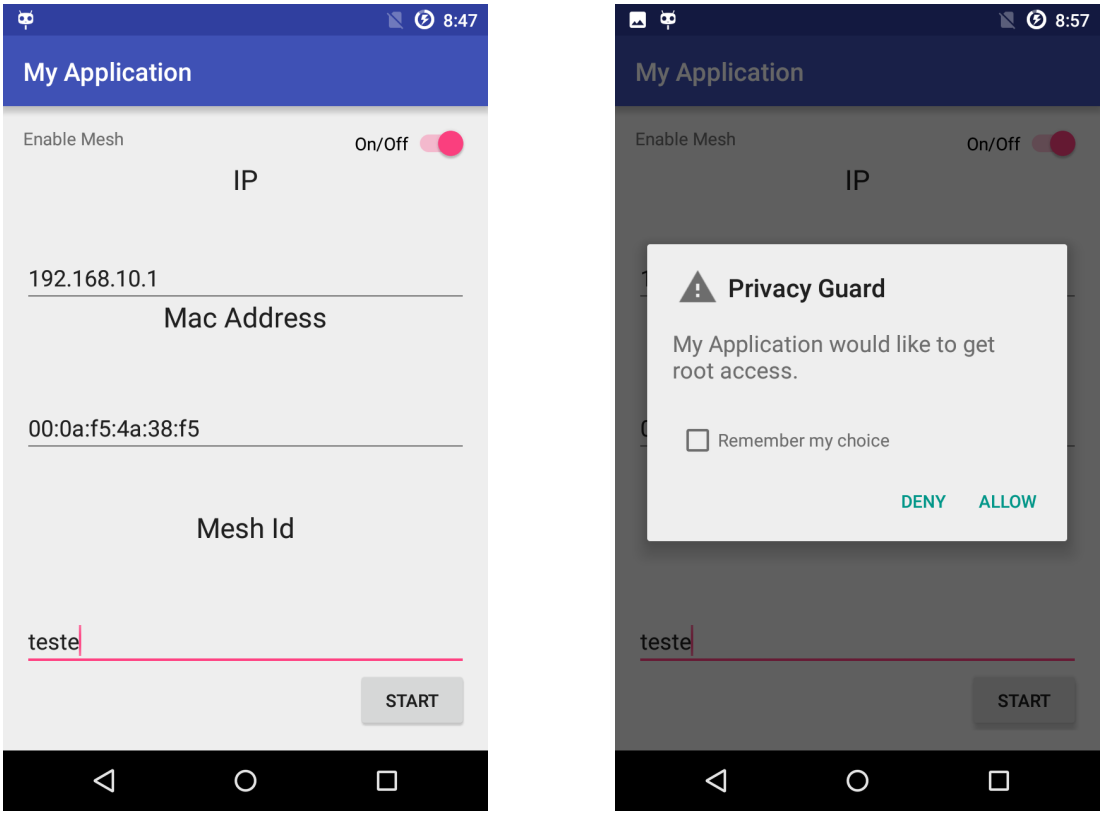


Figure 6.4: Mesh APK

Since SuperSU is being used, the application will need to require root privileges in order to perform the needed instructions.

Chapter 7

Conclusion

The results show that it is possible to implement Wireless Mesh Networks (WMN) in Android devices. The driver overall behaviour was satisfactory in terms of establishing links an relaying data. We were able to observe some behaviour like dynamic path creation and path selection. In addition, we were able to assert reliability in coverage provided. We hope these results and documentation provided will help future developers to continue the work in this field, as potential implementations using mobile devices as intervinients can lead to massive scale networks. We also showed that with some more work, it will be possible to create a finished product that will enable the use of all the features of Cyanogenmod (CM) associated with WMN capabilites.

7.1 Future Work

Regarding the CM source code, we would like to see the Mesh settings APK integrated directly within the settings APK of the CM user interface. We think that it would provide the user with a more familiar experience as well as more intuitive. Also, with some work it would be possible to create or manipulate the existing wifi Application Programming Interface (API) to correctly parse and use Mesh commands directly, instead of resourcing to the `iw` tools or requiring previledge access as root user.

Regarding the driver integration, we think that creating an solution that relies on a native SoftMac driver would be of great value. We have to bare in mind that the WCN36XX project was dropped, and that it does not cover a great deal of current devices since newer chipsets are already available.

7.2 Contribution

All the documentation related with the project, as well as all the source code will be released at www.static.pt/androidmesh and https://github.com/androidmesh. The information will also

be propagated among the several community forums such as the CM community and XDA Developers on the form of tutorial posts. We hope that fresh documentation, with effective results will work as a catapult for future developers to try and create more solutions.

7.3 B.A.T.M.A.N. Routing Protocol Integration

On the course of the Battle of the Wireless Battle of the Mesh V9 conference, we talked to the B.A.T.M.A.N. protocol representative, Simon Wunderlich [1]. Our work was of great interest to them as they are receiving several pull requests for a mobile solution to implement their protocol. Since 802.11s can be used to create a backbone among wireless devices, can provide Layer 2 routing and connection and was previously mentioned that the protocol API is able to customize and run other protocols, we decided that the full source code of this implementation, as well as the documentation created would be provided as a contribution to the B.A.T.M.A.N. routing protocol project.

[1] https://www.open-mesh.org/users/16

Bibliography

[1] IEEE Standard for Information Technology- Telecommunications and Information Exchange Between Systems- Local and Metropolitan Area Networks- Specific Requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs). Technical report, 2006. doi:10.1109/ieeestd.2006.232110.

[2] Sown - the southampton open wireless network. http://redhookwifi.org/, 2007.

[3] Sown - the southampton open wireless network. http://sown.org.uk/, 2007.

[4] How to build cyanogenmod for google nexus 4 ("mako"). https://wiki.cyanogenmod.org/w/Build_for_mako, 2012.

[5] Nycwireless. https://nycwireless.net/, 2013.

[6] Developing | android open source project. https://source.android.com/source/developing.html, 2015.

[7] Boringssl. https://boringssl.googlesource.com/boringssl/, urldate = 2015-11-09, 2015.

[8] Doc: Building basics. https://wiki.cyanogenmod.org/w/Doc:_Building_Basics, 2015.

[9] Envsetup help. https://wiki.cyanogenmod.org/w/Envsetup_help, urldate = 2015-11-09, 2015.

[10] Howto - o11s/open80211s wiki. https://github.com/o11s/open80211s/wiki/HOWTO, 2015.

[11] en:users:documentation:iw [linux wireless]. https://wireless.wiki.kernel.org/en/users/documentation/iw, 2015.

[12] Os. https://www.openssl.org/, urldate = 2015-11-09, 2015.

[13] en:users:drivers:wcn36xx [linux wireless]. https://wireless.wiki.kernel.org/en/users/drivers/wcn36xx, 2015.

[14] W1.fi. https://w1.fi/wpa_supplicant/, 2016.

[15] Developer.android.com. https://developer.android.com/reference/android/net/wifi/package-summary.html, 2016.

[16] Android.googlesource.com. https://android.googlesource.com/platform/frameworks/base/+/618455f/wifi/java/android/net/wifi/WifiNative.java, 2016.

[17] Android.googlesource.com. https://android.googlesource.com/platform/system/core/+/master/init/readme.txt, 2016.

[18] Help.ubuntu.com. https://help.ubuntu.com/community/UbuntuBackports, 2016.

[19] Drvbp1.linux-foundation.org. http://drvbp1.linux-foundation.org/~mcgrof/rel-html/backports/, 2016.

[20] Cs.cornell.edu. http://www.cs.cornell.edu/courses/cs5430/2011sp/NL.accessControl.html, 2016.

[21] Wiki.centos.org. https://wiki.centos.org/HowTos/SELinux, 2016.

[22] Ntu.edu.sg. https://www.ntu.edu.sg/home/ehchua/programming/android/Android_NDK.html, 2016.

[23] Cwnp - certified wireless network professional. https://www.cwnp.com/uploads/802-11s_mesh_networking_v1-0.pdf, 2016.

[24] Porting iw on android. https://github.com/imlinhao/android-iw-libnl3, 2016.

[25] Backports.wiki.kernel.org. https://backports.wiki.kernel.org/index.php/Main_Page, 2016.

[26] Gowasabi.net. http://gowasabi.net/, 2016.

[27] Do access points use softmac or hardmac? http://stackoverflow.com/questions/28343384/do-access-points-use-softmac-or-hardmac, 2016.

[28] 2016. [link].

[29] Meta mesh | wireless networking for all. http://www.metamesh.org/, 2016.

[30] Statista. https://www.statista.com/statistics/232786/forecast-of-andrioid-users-in-the-us/, 2016.

[31] Krasnikoveugene/wcn36xx. https://github.com/KrasnikovEugene/wcn36xx, 2016.

[32] S. Basagni, M. Conti, S. Giordano, and I. Stojmenovic. *Mobile Ad Hoc Networking*. Wiley, 2004. ISBN: 9780471656883.

[33] Android Bridge. Android debug bridge | android studio. https://developer.android.com/studio/command-line/adb.html, 2016.

[34] Raffaele Bruno, Marco Conti, and Enrico Gregori. Mesh networks: commodity multihop ad hoc networks. *IEEE Communications Magazine*, 43(3):123–131, 2005.

[35] Joseph D Camp and Edward W Knightly. The ieee 802.11 s extended service set mesh networking standard. *IEEE Communications Magazine*, 46(8):120–126, 2008.

[36] COZYBIT. Cozybit - authsae. = https://twrp.me/, 2016.

[37] Daniel de O Cunha, Luís Henrique MK Costa, and Otto Carlos MB Duarte. Analyzing the energy consumption of ieee 802.11 ad hoc networks. In *Mobile and Wireless Communication Networks*, pages 473–484. Springer, 2005.

[38] CyanogenMod. About cyanogenmod. = https://wiki.cyanogenmod.org/w/About, 2015.

[39] Sahibzada Ali Mahmud, Shahbaz Khan, Shoaib Khan, and Hamed Al-Raweshidy. A comparison of manets and wmns: commercial feasibility of community wireless networks and manets. In *Proceedings of the 1st international conference on Access networks*, page 18. ACM, 2006.

[40] MIT. Nortel's wireless mesh networks. = https://www.media.mit.edu/sponsorship/getting-value/collaborations/nortel, 2001.

[41] IEEE Org. 802.11-2012 - IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. (IEEE Std 802.11$^{\text{TM}}$-2012).

[42] Meet Studio. Meet android studio | android studio. https://developer.android.com/studio/intro/index.html, 2016.

[43] TWRP. Teamwin - twrp. = https://github.com/cozybit/authsae, 2016.

[44] Xudong Wang and Azman O Lim. Ieee 802.11 s wireless mesh networks: Framework and challenges. *Ad Hoc Networks*, 6(6):970–984, 2008.