SSID Stripping Attack: From Vulnerability to Tangible Threat

Author: Emima Vaipan
Supervisor: Eirini S Anthi
Moderator: Padraig Corcoran

School of Computer Science and Informatics
CM3203 One Semester Individual Project - 40 credits
Cardiff University

May 2022

24421 words

Acknowledgements

Firstly, I want to thank my supervisor, Eirini A., for accepting to supervise me during this project and for her support, constant encouragement, and overall insights.

I would also like to thank the one and only James C. for sticking by me and always keeping faith that I could do this, particularly when I did not. I cannot thank you enough.

Finally, I thank my family, friends, and church family for supporting me during my dissertation.

Contents

Chapter 1

Introduction

1.1 Project Description

SSID Stripping is a recently disclosed [1] and discovered vulnerability compromising the overall security of IEEE 802.11-compliant devices running on platforms such as Android, Windows, Ubuntu, Apple macOS, and iOS. Therefore, this vulnerability can potentially cause a severe impact on a range of devices at scale, making its exploitation particularly dangerous. In addition, the consequences of this can be anything from attackers accessing or running unauthorised functions to carrying out malicious activities on user devices or deceiving them into connecting to rogue Access Points (APs), which mimic legitimate ones.

This vulnerability relates to how the Wi-Fi standard handles the discovery of devices near a networking hardware device, such as an AP. In addition, this relates to how manufacturers implement particular devices to behave in the probing phase of the Wi-Fi communication process. According to the security researchers at AirEye, SSID Stripping enables attackers to maliciously craft the Service Set Identifier (SSID) in the beacon frames sent by an AP specifically to allow unintended behaviour in devices when they process the beacon frames. SSID is commonly known as the displayed name of a Wi-Fi network.

This project aims to research and demonstrate the exploitation and impact of this vulnerability on Android devices through a range of attacks, such as injection and spoofing attacks, using a rogue AP as an initial attack point. It also aimed to provide a learning and research opportunity to understand the Android Wi-Fi stack better and reverse engineering low-level components.

1.2 Project Motivation

Recent years have seen an increase in the number of devices using Wi-Fi connectivity based on IEEE 802.11 standards, making Wi-Fi security paramount. However, with this increase in use and importance, security threats and attacks have constantly evolved to adapt to new technologies and vulnerabilities. As a result of this exponential growth, it is now easier to overlook vulnerabilities and unwanted behaviour stemming from existing functionalities, especially if they are not readily apparent. While considerable effort has been placed into ensuring the communication aspect of Wi-Fi is secure, little research has been done into the un-secured setup that devices undergo to enable Wi-Fi communication. From the limited resources and research on SSID Stripping, it is clear that adversaries can exploit the vulnerability to hack devices running major software such as Android, which demands further investigation.

Considering that the SSID Stripping vulnerability resides within the probing phase, also known as the discovery phase, of the Wi-Fi communication process, a device would not need to be authenticated or associated with a network for a persistent adversary to exploit it. Such exploitation could lead to a device, in this case, an Android smartphone, crashing or potentially leaking sensitive data. Moreover, according to the researchers from AirEye, SSID Stripping brings a new take on spoofing, as it creates more practical and realistic rogue APs to deceive users into connecting to a fake network. The potential of this vulnerability to make devices on any platform vulnerable to a range of attacks motivates me to research it further.

1.3 Primary Research Objectives

This project aims to perform research surrounding SSID Stripping vulnerability and focus on establishing through experimentation the vulnerability profile of various Android devices from popular vendors such as Samsung, OnePlus and Motorola. Basic Proof of Concepts (PoCs) will be developed and demonstrated to validate the findings based on the insights gained. The aim is also to conduct more in-depth reverse engineering against the results to better understand the vulnerability and potentially propose mitigations to protect users.

1.4 Research Contributions

Through our research, we demonstrated that our Android test devices were vulnerable to a range of SSID stripping based attacks which a malicious actor could leverage to augment various social engineering attacks. We also provided an in-depth analysis of proprietary Wi-Fi specific firmware by vendors, which enabled us to examine and show where and how some of these vulnerabilities are introduced.

This area is largely unexplored, with current research focusing primarily on the communication aspect of the Wi-Fi stack rather than the initial unsecured setup. We hope our findings help illustrate to security researchers the potential risks and vulnerabilities located in these components while also providing device vendors with information to help keep their users secure.

1.5 Research Outline and Literature Review

The rest of the paper is organised as follows. Section 2 covers the background on IEEE 802.11 Wi-Fi implementations and explains essential concepts related to the Android system. This also functions as our literature review, considering that there is no other work that we know of related to SSID Stripping apart from what has been discussed in the previous sections. Furthermore, as of April 2022, the research group from AirEye confirmed that this class of vulnerability is still present in major operating systems, like Windows, Android, and iOS [2]. Throughout this project, we compiled and utilised a range of sources, including official Android documentation, well reviewed technical information sources, and official standards and practices issued by various authorities. Section 3 presents two experiments performed; we outline the test approach and cover our findings in detail along with the implications. Section 4 discusses the impact of our research and experiments and guides some of the future work that could be implemented in this area. It also covers some learning reflections and concludes our investigation.

Chapter 2

Background

2.1 Wi-Fi Fundamentals

This section provides an overview of the IEEE 802 family and background on the IEEE 802.11 wireless standards and Wi-Fi technology. In particular, we discuss some of the services, protocols and components involved in enabling wireless communication and how their functionality can contain various vulnerabilities applicable to our research topic.

2.1.1 IEEE 802 Standards Family

IEEE 802 [3] is a family of network standards developed by The Institute of Electrical and Electronics Engineers (IEEE) [4], which defines specifications for the physical components of a network, such as Network Interface Cards (NICs) and network cabling. IEEE standards play a crucial role [5] in guiding the communication and interaction between technology components and ensuring fast product development, interoperability, flexibility, and compatibility between complex services and products. The logical architecture of IEEE 802 standards follows a Reference Model formed of three layers, which map to the bottom two layers of the ISO Open System Interconnection Reference Model (ISO/OSI) [6]. Generally, the ISO/OSI Reference Model conceptually defines seven layers that information must traverse from one endpoint device over a network medium to another endpoint device.

According to [7], the IEEE 802 physical layer (PHY) standards correspond to the ISO/OSI physical layer, whilst the IEEE 802 medium access control (MAC) and logical link control (LLC) layers map to the ISO/OSI data-link layer, as shown in Figure 1 below. In this context, the LLC layer provides an interface to higher layers and performs flow and error control, and the MAC layer is responsible for managing data from the PHY layer. The latter acts as an interface between the devices and the transmission medium. The PHY layer also defines hardware specifications, signalling and encoding techniques to transfer information-bearing signals over a transmission medium effectively. Figure 1 below illustrates the relationships between the IEEE 802 and ISO/OSI Reference Model, along with some of the active standards and working groups associated with each layer. In addition, other groups of standards are available which are currently hibernating or disbanded [8], such as IEEE 802.2 LLC but are continuing to provide value to current network implementations.

Figure 2.1 below illustrates the relationships between the IEEE 802 and ISO/OSI Reference Model, along with some of the active standards and working groups associated with each layer. In addition, other groups of standards are available which are currently hibernating or disbanded [7], such as IEEE 802.2 LLC but are continuing to provide value to current network implementations.

2.1.2 IEEE 802.11 Wi-Fi Overview

Part of the IEEE 802 family is also the IEEE 802.11 set of standards which define guidelines for Wi-Fi communications necessary to support networking in a local area. These standards have continually evolved to provide better bandwidth and overall capabilities to keep up with user traffic demands, availability, and

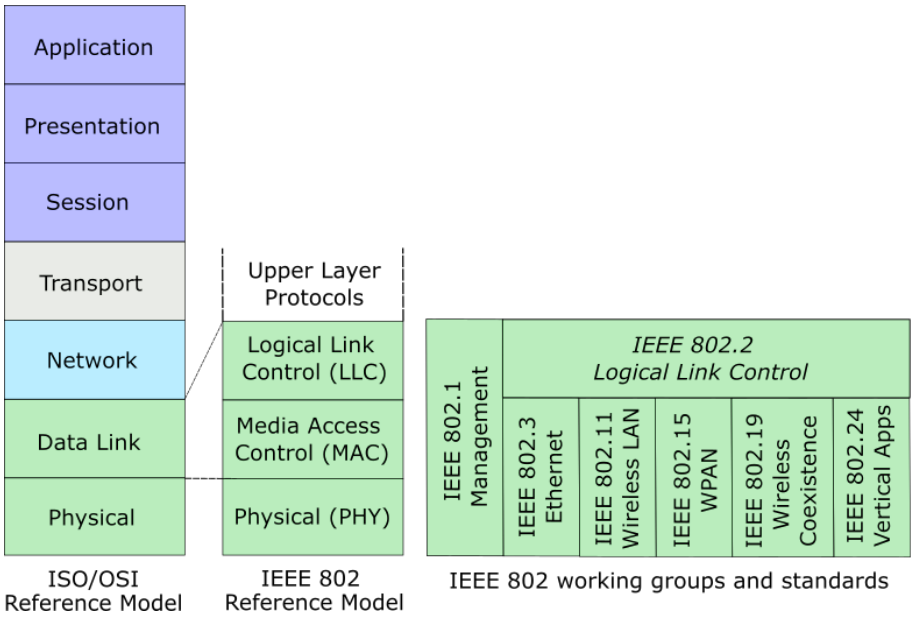


Figure 2.1: IEEE 802 Reference Model relative to ISO/OSI. Examples of groups of standards for each individual layer.

ever-changing technological developments. 802.11ax, also known as "High-Efficiency Wireless" or Wi-Fi 6, is the newest standard as of 2019 that provides the fastest version of Wi-Fi. This replaced 802.11ac (Wi-Fi 5) and 802.11n (Wi-Fi 4); however, Internet Service Providers (ISPs) have yet to upgrade their routers to support Wi-Fi 6, even though they are backwards compatible with older Wi-Fi 5 client devices.

The table below illustrates the evolution [9] [10] of the most popular Wi-Fi standards and how they improved regarding their data rates. The IEEE standard specifications names below are case-sensitive.

Standard	Release Date	Frequency Band	Maximum (Theoretical) Speed
802.11	1997	2.4 GHz	2 Mbps
802.11a	1999	5 GHz	54 Mbps
802.11b	1999	2.4 GHz	11 GHz
802.11g	2003	2.4 GHz	54 Mbps
802.11n / Wi-Fi 4	2009	2.4 & 5 GHz (optional)	600 Mbps
802.11ac / Wi-Fi 5	2013	5 GHz	7 Gbps
802.11ax / Wi-Fi 6 & Wi-Fi 6E	2019	2.4 & 5GHz	14 Gbps
802.11be / Wi-Fi 7	Not yet finalised	2.4 & 5 & 6 GHz	TBD

Table 2.1: Evolution of the Wi-Fi standards.

2.1.2.1 Wi-Fi Radio Frequencies and Channels

Wi-Fi [11] differs from wired networks such as Ethernet [12] in that it uses electromagnetic radio waves technology to communicate with the media instead of transmitting electrical signals over a cable. These radio waves are transmitted on various wireless frequencies and channels defined by IEEE 802.11 standards [13] [14] and regulated by individual countries to prevent harmful interference [15]. For example, in the United Kingdom, 2.4GHz and 5GHz are the commonly used spectrum bands that create the radio waves designated to "carry" the wireless data over the air. The availability of frequency bands depends on several factors, such as individual device capabilities, network configurations, and other environmental conditions (e.g., obstruction, distance, signal strength, interference etc.). Both frequencies are divided into multiple channels to prevent high traffic and interference with other wireless devices. According to Ofcom [16], in the United Kingdom, the 2.4GHz radio spectrum is the most used, and it has 14

channels sharing 100 MHz, with each channel having a bandwidth about 20MHz wide and some of them overlapping. On the other side, there are more and wider channels on the 5GHz spectrum (at least 20MHz channel width without overlap), and they spread over three frequency ranges – Band A, Band B, and Band C, with only the first two available for licence-free use by Wi-Fi devices. Although 5GHz helps with congestion and provides faster and more reliable wireless connections than 2.4Ghz, it has less coverage [17], and some devices do not support this frequency band [18].

From a security perspective, Wi-Fi networks using the 2.4GHz radio spectrum are more prone to attacks because they have a more extended range than 5GHz, making them prone to accidental or intentional interference and packet loss. In addition, the 2.4GHz band is vastly overcrowded because of so many devices and different technologies using it (e.g., Bluetooth, Zigbee, or Thread), which makes unintended attacks, including coexistence attacks more common.

In this case, an adversary does not have to be as close to a target device and could launch attacks that can automatically disrupt the wireless communication of any STAs in its proximity. Examples of network attacks that could benefit from a 2.4GHz signal travelling further are Evil Twin attacks (see Section 2.1.6), ARP Spoofing attacks, or Denial of Service attacks, such as Deauthentication attacks. As part of the experimental setup for this research presented in Section 3, we configure a temporary 2.4GHz routed Access Point using a Raspberry Pi 4.

2.1.2.2 Wi-Fi Architecture and Components

According to [10] and illustrated in Figure 2.1, the IEEE 802.11 standards define specifications only for the PHY and MAC layer, which allow communication across a wireless local area network (WLAN). Therefore, the standards that cover the services, protocols, and components necessary to enable WLAN communication are separated from the LLC layer because of the interdependence between the MAC layer, transmission medium, and network topology. Under this standard, a Wi-Fi network architecture consists of two different types of devices: wireless stations (STAs, referred to as wireless clients) and Access Points (APs).

First, the STA can be any Wi-Fi-enabled device, portable or fixed, such as a laptop or smartphone, equipped with a Network Interface Card (NIC, known as a network adapter) or a Wi-Fi chipset compliant with IEEE 802.11 wireless standards. The NIC or Wi-Fi chipset acts as both a radio transmitter and receiver, tuned to pick up radio frequencies of 2.4 GHz or 5 GHz and connect to any Wi-Fi Access Point in range. Fundamentally, the hardware chip takes the signals from the device, converts them into radio waves and transmits them to other devices simultaneously using the device's internal antennae. When an STA receives wireless signals, it decodes these transmissions back into binary code for the device to read. Second, an AP is an addressable network device that generates Wi-Fi signals to allow multiple STAs to connect wirelessly and access a LAN. Usually, wireless routers have in-built AP functionality, but they also provide routing between wireless clients and the Internet.

The Wi-Fi chipset market has evolved at an impressive rate in the last years, supplying an indispensable technology for delivering low-cost wireless connectivity. Simultaneously, the AP market has also grown, making the price of constructing an AP and hacking Wi-Fi significantly cheaper, considering the range and availability of open-source software, tools and guides online.

As previously mentioned, we are primarily interested in devices such as Android smartphones, given the scope of this project. However, we use other terms interchangeably, such as mobile STAs, wireless clients, or mobile devices, for the remainder of this research unless stated otherwise.

2.1.2.3 Wi-Fi Topology

IEEE 802.11 standards [10] support two main network topologies (also called modes), defining how the physical components mentioned in section 1.2.2 are arranged and interact with each other. These topologies are enabled by hardware chipsets, which can be configured to infrastructure or ad hoc mode. For example, a typical Wi-Fi network with devices in infrastructure mode allows them to communicate through APs connected to a wired LAN. In contrast, networks operating in ad hoc mode [19] require wireless chipsets to be configured accordingly and have the same frequency band, channel, and network name (SSID). In this case, wireless devices must be within each other's signal transmission range, which extends automatically when a new device joins the ad hoc network. This network usually consists of

two or more wireless devices that can communicate directly, peer-to-peer, without needing a hardware AP. Ad hoc mode is also called Independent Basic Service Set (IBSS). Unfortunately, this topology is unavailable in many Wi-Fi-enabled devices [19], including Android smartphones.

Although ad hoc mode is a valuable technology which allows device-to-device connectivity on the go when infrastructure mode is not available or impractical, it is generally more susceptible to attacks. This is because of a lack of centralised physical security and dynamic interconnection between devices, which are difficult to detect or isolate if an adversary compromises them. In addition, broadcasting the Wi-Fi name to allow client devices to discover the network opens the door to attacks such as spoofing, eavesdropping, and denial of service. On the other hand, a Wi-Fi network in infrastructure mode requires an AP as a base station and is more secure when configured correctly. For example, in this case, devices must go through a complete authentication process before associating with a network. Some methods available to ensure secure access and communication to a Wi-Fi network are MAC address filtering, authentication, and data encryption. This research requires setting up an AP as part of the experimental testing, which means that the focus is on infrastructure mode.

Another way of logically grouping wireless devices in a WLAN is through components or building blocks called service sets, which are also defined by the IEEE 802.11 standards. For example, the main basic building block in an infrastructure network is called a Basic Service Set (BSS) and consists of at least one wireless device connected to a single AP, as presented in Figure 2.2.

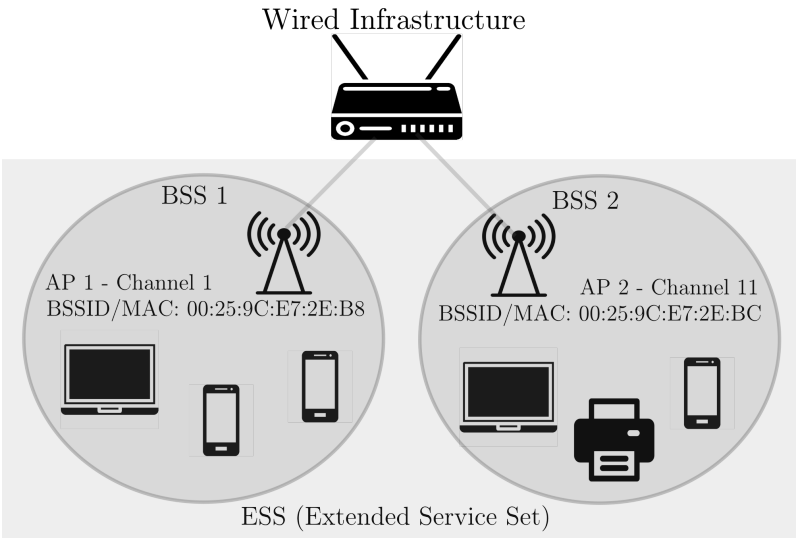


Figure 2.2: 802.11 Infrastructure Service Sets.

Each BSS is uniquely identified by a Basic Service Set Identifier (BSSID), usually the Media Access Control (MAC) physical address of the AP's radio card used by wireless devices to connect to the Internet. However, in some cases, manufacturers enable a physical AP with one or more radio cards to create virtual BSSDs. Still, they reserve a base MAC address for each radio card in the infrastructure device [20]. This identifier is a 48-bit address formed of 12 hexadecimal digits and is included in all wireless packets transmitted on the network. When a WLAN in infrastructure mode has at least two APs, and subsequently more BSSs, it forms an extended network called an Extended Service Set or ESS, but covering this is out of scope.

2.1.3 IEEE 802.11 Wi-Fi Service Discovery

In the context of a standard infrastructure WLAN, a wireless mobile STA and an AP must announce their presence to each other before the mobile STA joins and exchanges wireless packets on the network. Therefore, the IEEE 802.11's MAC layer specifications include a passive and active scanning protocol, as well as various types of management frames to help establish, maintain, and terminate the communication between mobile STAs and an AP. Some of these management frames include beacons, probe requests,

probe responses, disassociation frames, deauthentication frames, association requests, and authentication frames [21]. Since they are broadcasted in plain text and unauthenticated, they add a layer of complexity to maintaining and protecting Wi-Fi networks from potential attackers [22].

Whenever a mobile STA has its Wi-Fi radio interface on, as illustrated in Figure 2.3, it can probe for available APs in its vicinity. In the active discovery protocol , a mobile STA uses probe request frames to broadcast information about itself to APs in range or directly address a specific one.

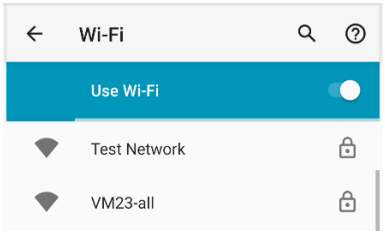


Figure 2.3: Passive Monitoring: Mobile STA - AP Association Process.

Each mobile device has a Preferred Network List (PNL) stored in memory containing APs it previously connected to in order to automatically switch to known networks when they are in range. The device periodically searches for networks saved in the PNL through directed probe requests. Whether broadcast or directed passive scanning is employed, a request frame must receive a positive probe response frame from an AP to continue to the authentication and association phase of the connection process, simplified and presented in Figure 2.4. If this does not happen, the mobile STA assumes the AP is unavailable, and those request frames are discarded.

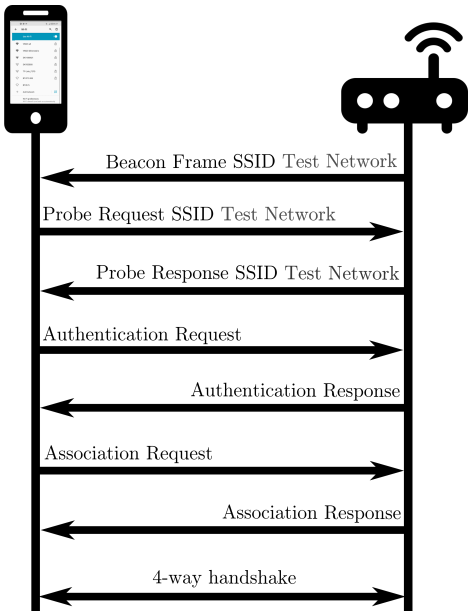


Figure 2.4: Mobile STA - AP Association Process.

Another way for the mobile STAs to receive information about APs in the range is through passive monitoring, which is at the centre of this research. This mechanism is initiated by an AP, which, as mentioned in Section 2.1.2.2, is the component that makes possible the infrastructure WLAN. This network device contains a radio card compliant with IEEE 802.11 standards, particularly with the PHY and MAC layer specifications, allowing it to communicate with other LAN devices. The AP continually scans the radio spectrum and periodically broadcasts beacon frames on either the 2.4 GHz or the 5 GHz band to announce its presence to potential mobile clients containing a radio card. The rate to which these beacon frames are broadcasted by default is every 102.4 milliseconds [23], and they include the MAC address of the infrastructure device and network name (SSID).

In passive discovery mode, mobile STAs iterate through all possible channels, listen to beacons and update their Available Networks List (ANL) upon discovering new APs, as illustrated in Figure 2.5 . If the network is busy, the AP delays the transmission of beacons for a random duration to avoid collisions. In addition, mobile STAs may attempt to authenticate and associate with a specific AP through a set of protocols and services defined by the IEEE 802.11 standards at the MAC layer.

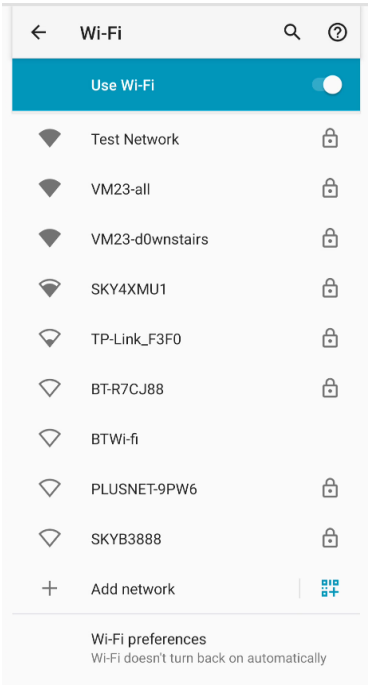


Figure 2.5: Example of a populated ANL.

2.1.4 IEEE 802.11 Wi-Fi Access Network Selection

At the end of the network discovery process, a mobile device publishes the information related to available APs in an Available Networks List (ANL) to facilitate network selection for the end-user. The way this list is ordered is crucial since the end-user usually chooses to connect to a Wi-Fi network that provides the best performance, depending on their requirements and the options available. Therefore, the conventional way of selecting an AP and ordering the ANL is currently based on various factors, such as the signal strength of the received packet, wireless security settings and network quality [24]. The signal strength is one way of measuring the expected quality of the communication link between the mobile STA and AP. It is recorded by the client device as the received signal strength indicator (RSSI) measurement and presented to the end-user through graphic bars, as shown in Figure 2.6. However, there is no standardised way of measuring the signal strength. Even though most Wi-Fi chip vendors use the RSSI as a standard measurement because it is the easiest to read, it can also be done in dBm (decibels relative to milliwatts) and milliwatts (mW) [25]. The latter is the most accurate but the hardest to measure and read, whilst dBm provides mid-accuracy of the signal strength, and it is more comprehensible and common than the milliwatts. Under normal circumstances, when a mobile STA is closer to the AP, the higher the RSSI value, the stronger the Wi-Fi signal.

Given all the above, a mobile phone populates and updates the ASL upon continually discovering the network through active or passive scanning. As a result, the mobile STA will attempt to automatically connect to a saved Wi-Fi network or wait for the end-user to manually pick a network name and proceed to the authentication and association phase. Figure 2.5 can also be used as an example of populating the network list (ANL) through passive scanning.

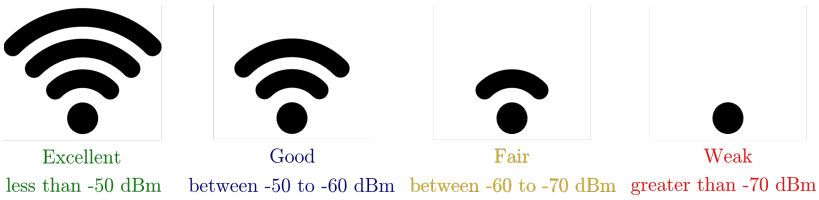


Figure 2.6: Wi-Fi Signal Strength values.

2.1.5 IEEE 802.11 Wi-Fi Beacons

A critical type of management frame in IEEE 802.11-based WLANs involved in passive network discovery is beacon frames. These carry information on an AP's operating channels to announce its presence and help mobile STAs identify compatible networks before connecting. For example, a mobile STA would not know the availability of nearby Wi-Fi devices without beacon frames, whether they were previously connected. Another option is active scanning, where a mobile STA sends broadcast requests to identify the networks in range. Each AP sends out a unicast probe response containing the same information as a beacon frame. In addition to discovery, a beacon can also help with providing information about the operational state of a Wi-Fi network, such as synchronisation for the members of a service set and frequencies, or even allow for power saving features in client devices.

Beacons are broadcasted at regular intervals at the lowest mandatory data rate to ensure every client device in range can hear them. This value depends on the PHY layer specifications and standard version [26]. For example, the minimum mandatory data rate for 802.11n and 802.11ac APs is 6 Mbps to maintain backward compatibility with 802.11a. Together with the size of the beacon frame, these values can help determine the time it takes a beacon to be transmitted from an AP to a mobile STA in common environmental conditions [27]. The size of a beacon packet depends on the amount of transmission done, considering that the frame body section of a beacon frame has a variable size. Mobile STAs also must be close enough to the AP to hear the beacons. The transmission range of beacons depends on standard specifications, frequency, and physical environment, and it is approx. 70 metres indoors.

2.1.5.1 Beacon Frame Structure

A beacon is one of the most information-dense wireless packets and contains the parameters shown in the beacon structure below. Most of these fields are automatically set at the 802.11 MAC layer without end-user intervention. However, the end-user can adjust some of them, depending on the brand and manufacturer of the AP.

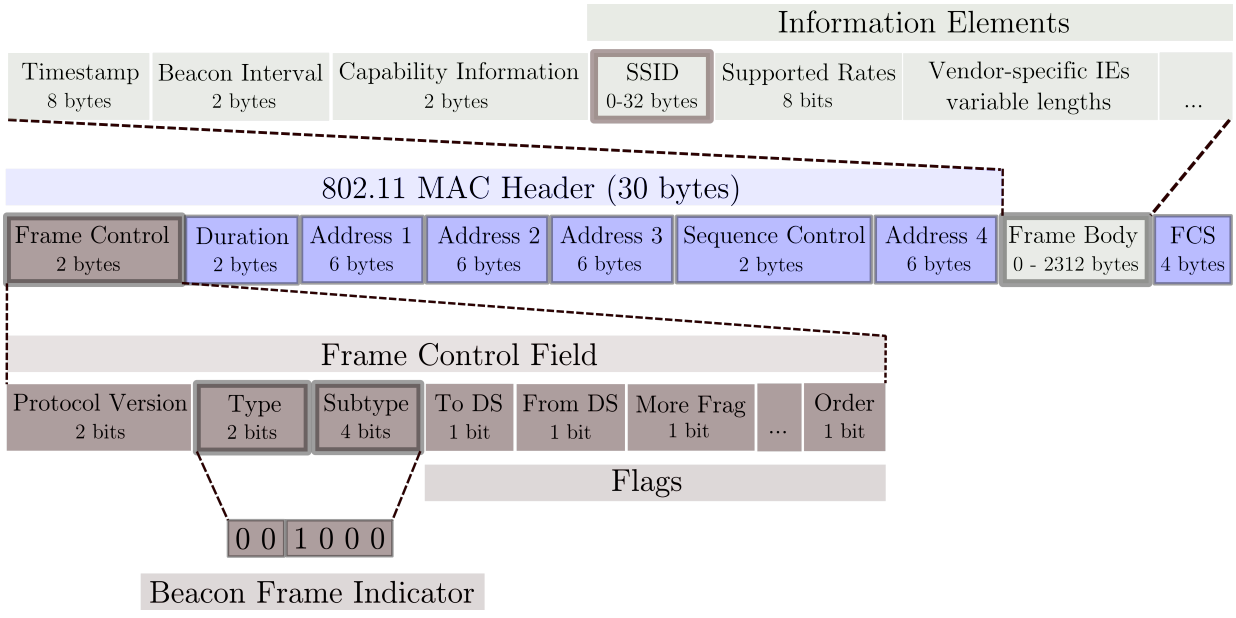


Figure 2.7: Beacon frame format as defined in the IEEE 802.11 standard.

Generally, a management frame comprises a frame header (or MAC header), a frame body, and a Frame Check Sequence (FCS), known as a trailer. The FCS field contains a four-byte value used to check for damaged frames and errors. As seen in Figure 2.7, the MAC header for management frames such as beacons consists of the following mandatory and fixed-length fields:

- Frame Control field to define the protocol version, type and subtype of the MAC frame, and other optional subfields depending on the frame type. Changing the type and subtype in the MAC header format changes the complete frame type. For example, the type and subtype values of 00 and 1000 respectively refer to a management frame of subtype beacon, whilst the type 00 and subtype 0100 refer to a management frame called Probe Request. Other types of frames include data frames (type 10), control frames (type 01), and extension frames (type 11).
- Duration/ID field, which is zero for broadcast frames. This value refers to the amount of airtime reserved on the channel for the pending acknowledgement frame İn this case, it is not necessary.
- The Address fields contain different values depending on the Frame Control field, such as the MAC address of the source/sender (or the BSSID) and destination address as the broadcast FF:FF:FF:FF:FF:FF.
- A Sequence Control Information field indicates the incremental sequence number of a beacon. This field helps to distinguish between retransmissions.

The frame body contains further information specific to the frame type and subtype. Some of the fields in the frame body are fixed-length, whilst other variable-length fields present are called information elements (IEs). These include the supported data rates, Service Set Identifier (SSID, or network name), other network capabilities, and vendor-defined attributes and data, as depicted in Figure 2.7. Both categories contain mandatory and optional fields. In this case, some of the mandatory ones are the timestamp (8 bytes), beacon interval (2 bytes), capability info (2 byte), SSID (variable size), and supported rates (variable size). The IEs have a common format defined by the IEEE 802.11 standards on how they should be interpreted, and they are revised frequently.

Figure 2.8 shows a screenshot of a beacon frame captured by a packet analyser like Wireshark and some information fields related to a specific SSID called “Test Network”.

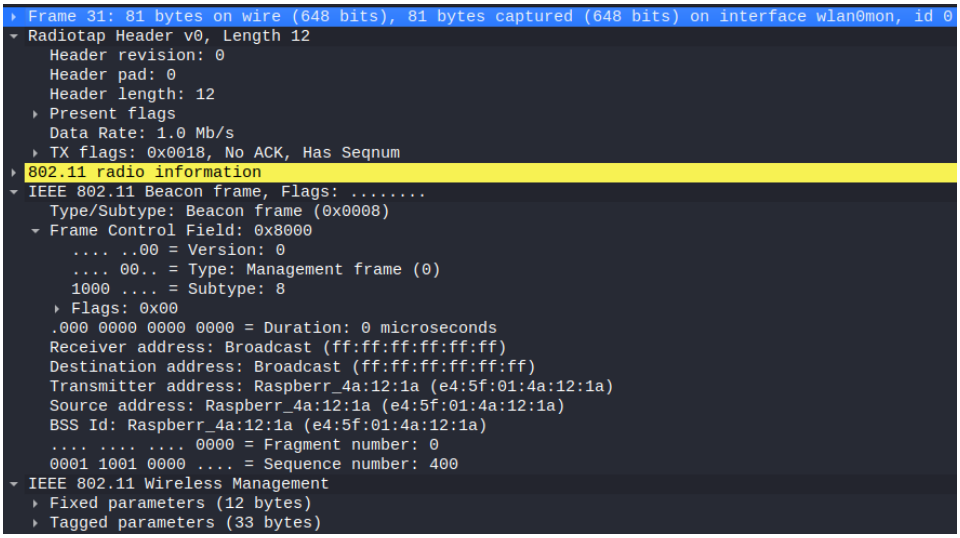


Figure 2.8: Beacon frame capture and visualisation in Wireshark.

Another important management frame similar to the beacon frame is a probe response frame. This frame also includes AP capabilities, authentication information, SSID, supported rates, etc. However, unlike a beacon frame, it is only triggered by a probe request as part of the active scanning of the network. Although we primarily focus on beacon frames, probe responses are also used by Wi-Fi networks to identify themselves to other devices in proximity.

2.1.5.2 Service Set Identifier Overview

As part of this research, we investigate one important IE, more specifically, the Service Set Identifier (SSID) [28]. This is an optional field included in some management frames, such as beacons, probe requests and probe responses. The SSID is a label for a service set (BSS/ESS), which means that all client devices and APs within a WLAN must use the same SSID to communicate wirelessly. Most APs can be configured to provide multiple networks, each identified by an SSID. This identifier is used to logically separate individual networks from one another, keep packets within the correct network boundaries, and apply authentication and security modes and bandwidth limits for a specific network [29].

2.1.5.3 Service Set Identifier Format

Regarding its format, the Wi-Fi standards require an SSID to have between 0 of 32 bytes, and there are no specifications regarding its potential content. This identifier is usually case-sensitive and human-readable, and most of the time, it consists of ASCII characters. However, according to the IEEE Std. 802.11-2012, the SSID is unspecified UTF-8 encoded or unspecified. This means it can contain any Unicode text and be customised by the end-user accordingly. More about encoding is presented in Section 2.1.5.4. Just like the other IEs mentioned in Section 2.1.5.1, the SSID is part of the frame body with ELEMENT ID 0 and has the following structure:

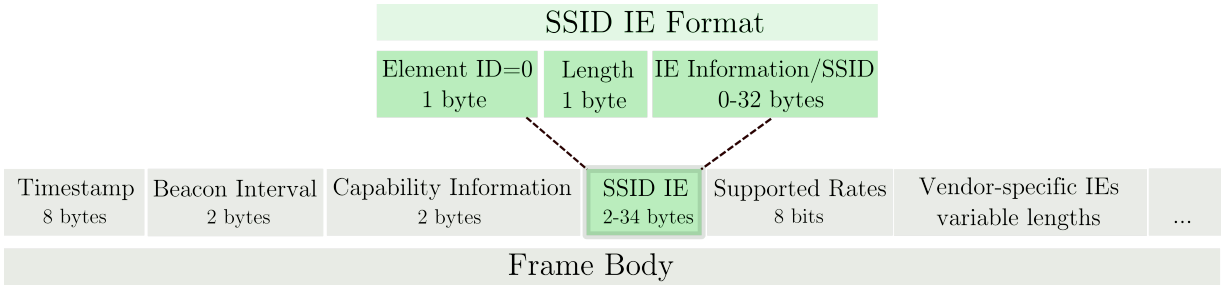


Figure 2.9: SSID IE Structure.

For an AP to work as intended, its radio card must have a configured SSID field. This network device intentionally broadcasts a beacon per SSID at regular intervals as part of the passive discovery of a network. The SSID identifies and announces a WLAN to potential mobile STAs in its vicinity, which passively listens for beacons on each channel. Mobile STAs use this scanning mechanism to update their available network list, which the end-user can view whenever the Wi-Fi interface is on. Figure 2.5 is an example of SSIDs discovered by an Android mobile STA.

A downside to discovering the network by passively scanning each channel is that it is slow and energy-consuming, so mobile STAs also use active scanning to search for compatible networks in their proximity. During active scans, a mobile STA attempts to discover available 802.11 networks by broadcasting probe-any requests containing a wildcard SSID [30]. This wildcard SSID, also called a "null SSID" or "no broadcast SSID", is an SSID that has a length of zero.

In this case, the standard behaviour when receiving these probe requests is for an AP to respond with a probe response advertising the network's SSID, supported data rates, encryption types and other parameters. This active scanning is continually done in the background of a mobile STA, regardless of its association with an AP. This is one of the main reasons why hiding a network by disabling the SSID in beacon frames does not make the network more secure. According to the IEEE standards, the SSID is not intended to be a secret, as it is the only available method of allowing mobile STAs to know what networks they could join. Therefore, stopping it from being broadcasted in beacon frames does not mean a network is hidden. This is because it will be included in the association phase to make the connection possible between a mobile STA and a network device, such as an AP [31].

2.1.5.4 Byte and Character Encoding

At the core of this research is the SSID field and how it is designed to be transmitted as part of the scanning mechanism and processed by the mobile STA. The SSID is an essential parameter, considering that there is no way for a legitimate user to connect to a Wi-Fi network without sending out or receiving this identifier in cleartext. In the context of Android-based devices, parsing and displaying the SSID correctly depends on the format of the bytes received from the AP and on whether the Android operating system allows that specific format.

The sequence of bytes that form the SSID is interpreted as single or multi-byte characters depending on the encoding method [32] used. This means that a system needs to be told of rules that allow mapping between a set of characters and byte representations specified through character encoding schemes. The two most popular ones are ASCII [33] and UTF-based [34] encodings. The ASCII set is the American Standard Code for Information Interchange ordering for a character set. This can use 7 or 8 bits (extended ASCII) to represent 128 or 256 possible characters, including the uppercase and lowercase letters in the English language, numbers 0 through 9, punctuation marks, and special control characters.

However, this set is rather limited by the bits available to represent characters, so UTF was introduced as an encoding method to address its shortcomings. UTF stands for UCS (Unicode) Transformation Format, and unlike ASCII, it uses between 8 and 32-bit to represent every character (or code point) part of the Unicode set [35].

When it comes to Android, this operating system represents Unicode characters using UTF-8 encoding by default [36] in which each character is encoded as one or more sequences of 8-bit bytes. Therefore, the device attempts to decode the SSID embedded in the beacon frame (or frame response) as UTF-8. In addition to this, the wireless network stack of each mobile device should be prepared to handle arbitrary values contained in the SSID field. However, if any invalid code points [37] get detected and cannot be decoded, an exception is triggered to indicate that the whole frame is potentially malformed.

In addition to this, the IEEE Std. 802.11-2020 specifications indicate a separate subfield in the Extended Capabilities parameter of a beacon frame which indicates that the SSID field should be interpreted by UTF-8 when the value is set to 1, or otherwise, the encoding is "unspecified".

ASCII and Unicode also allow us to represent special characters, and we are primarily interested in control characters, also known as non-printable characters. These are code points in character sets that do not represent a written symbol and are strictly used for non-textual usage. The end-user configuring the SSID field can use them as part of the 32 bytes to control the interpretation and display of the string. Still, the way they are digested and acted on depends on the operating system of the receiving end.

According to [38], the first 32 ASCII characters and character 127 (decimal) are control characters used to control input and output devices. Some of these instructions can indicate the end of a line/carriage return (control character - M̂, "̊" - Python, hexadecimal - 0x0D, decimal - 13) , character tabulation (keyboard combination - CTRLÎ, ˆ Python or Java, hexadecimal - 0x09), or backspace (keyboard character -Ĥ, ˙ Python or Java, hexadecimal - 0x08).

UTF-8 inherits all the ASCII characters, including control character sets, C0 and C2 control codes, and escape sequences. These characters have the same byte encoding in UTF-8 as with the ASCII encoding. In addition to this, it includes a variety of control characters exclusive to Unicode as part of the Cf category [43]. An example is the Zero Width Space (hexadecimal - 200B, decimal - 8203), which is UTF-8 encoded as 0xE2 0x80 0x8B.

Configuring the SSID field to display some non-printable characters within a string, such as a comma, backslash or quote, requires escaping the character sequence. In this case, the backslash character ( ) followed by the escaping character allows it to be treated as part of the string and avoid misinterpretation. For example, adding a backslash before an s character will enable it to be displayed on the screen as part of the string. Android also handles the characters that have special usage by allowing them to be escaped through a preceding backslash [44].

2.1.6 IEEE 802.11 Wi-Fi Threat Landscape

Despite the advantages of technology based on IEEE 802.11 standards, it is not easy to protect end-users from the increasing security challenges related to Wi-Fi. In particular, in recent years, researchers have been focusing on security weaknesses in the current Wi-Fi design, which significantly expands the attack surface through the eyes of an adversary and exposes risks for further potential attacks. In this context, the attack surface of Wi-Fi refers to any point of entry into a network in infrastructure mode.

One of the components of Wi-Fi that adversaries can abuse is management frames, which are unencrypted and unauthenticated and have been around since 802.11's original release in 1997. Furthermore, none of the later standards enhances how these frames are constructed and exchanged between an AP and an STA, which does not have to be part of the same BSS or Wi-Fi network to transmit or receive them. Therefore, management frames are attractive candidates for adversaries, and it has been shown that they can be used for location tracking [39], inference of private information, and identity theft. Unfortunately, these are complex security and privacy challenges, still investigated by security researchers [40] and yet to be solved.

As mentioned in Section 2.1.5, exchanging these management frames cannot be avoided as APs must advertise their network capabilities to nearby wireless STAs like mobile devices and help them establish and maintain connections. Mobile devices with their Wi-Fi interface turned on present an excellent opportunity for adversaries to eavesdrop on wireless traffic through passive network discovery, the pre-connection phase discussed in Section 2.1.3. Transmitting and capturing management frames may help them fingerprint both APs and mobile devices and mount attacks such as identity spoofing (e.g., Evil Twin, Rogue Access Point (RAP) attacks), denial of service (Wi-Fi jamming via Deauthentication attacks) and session hijacking attacks [41]. Some information of interest for an adversary contained within a management frame in plaintext includes the AP's MAC, SSID, and RSSI values.

In the context of this research, we are primarily focused on the current threat landscape that the end-user faces because of the exploitation of passive scanning and specific fields within management frames. The most important field in this research is the SSID, which is already known to compromise the privacy of end-users, as reported in [42]. Because the SSID travels unencrypted through the air, adversaries can use wireless sniffing software to find and spoof an AP and mount additional attacks. From a mobile view, devices used Wi-Fi probe requests to periodically scan for certain SSIDs they previously connected to and saved in a PNL on their device, which could be leveraged to observe and infer private information about a target end-user. However, in recent years device vendors switched to a more passive Wi-Fi scanning method for saved networks, which involves waiting for beacon frames instead of directly probing the PNL. They also adopted randomised MAC addresses alongside static ones for devices when exchanging packets, and broadcasting probe requests using a wildcard MAC address (ff:ff:ff:ff:ff:ff) and a wildcard SSID irrespective of an ongoing connection. This is to discover the RSSI or signal strength of the APs in proximity and maintain any existing network connections.

When APs advertise their network through beacon frames, they broadcast the SSID, BSSID/MAC address, and other security features in plaintext, while mobile devices passively listen to these beacons and update their ANL. In this case, an adversary only has to capture and analyse one beacon to compromise a legitimate AP (LAP) and launch an Evil Twin (ET) attack. However, since the ANL is primarily ordered based on the signal strength of APs in proximity, an adversary could use a Beacon Flooding attack to push a legitimate SSID further down an ANL list and convince the victim end-user to select the malicious one. Therefore, the Evil Twin attack introduces various security threats as it can redirect the communication of STAs, cause channel saturation, and degrade the network communication quality. This type of attack usually leads to capturing credentials from WPA/WPA2 pre-shared keys or third-party login pages, and it may also be used as part of an attack on a broader scale. In addition, the ET attack is frequently carried out against public hotspots commonly found in airports, coffee shops etc., to masquerade as a LAP provider and force victim end-users to connect to a rogue network automatically. This attack requires some end-user interaction, and a general user primarily relies on the SSID displayed in their ANL to identify an AP and initiate the connection process. Using management frames to spoof a LAP is particularly dangerous because they are used at the LLC layer of the network stack (see Section 2.1.2). In this case, network-based intrusion can only detect attacks at the IP layer or above, not spoofing attacks, whether pre-or post-association to a RAP. Some of the detections and mitigation introduced in mobile STAs against Wi-Fi spoofing include traffic modification detection, fingerprinting of

802.11 MAC responses from hardware LAPs or RSSI-based device profiling to detect anomalies at lower layers.

In addition to the role SSID plays in spoofing attacks, adversaries may also investigate the semantic information available in the SSID [43] and correlate it to, e.g., business venues, geographical areas, specific etc. This could potentially be used to identify a person (e.g., SSID is called "John's Wi-Fi"), a broadband provider (e.g., SSID begins with "VM" for Virgin Media customers), or the global educational wireless, "eduroam" [44].

2.2 Android Fundamentals

This section presents aspects of the Android OS relevant to this research, such as the Android architecture, security model, different types of Wi-Fi chipsets and network stack. In particular, this section aims to describe the functionality Android has regarding the IEEE 802.11 Wi-Fi protocol and modes and investigate the functions involved in handling the SSID field in the service discovery phase of Wi-Fi provisioning. This section primarily relies on two trusted sources of information: Google's Android cross-reference/source/developer guides [45] [46] [47] and Jonathan Levin's book on Android Internals, called "Android Internals: Power User's View" [48] [49].

2.2.1 Android Overview

Android is an open-source OS purchased by Google in 2005 and developed by the Open Handset Alliance in 2007, which is a group of hardware, software, and telecommunications companies committed to developing open mobile device standards. The global popularity of Android primarily stems from Google releasing its source code under various open-source licences for free use by anybody, including device manufacturers. Google first introduced Android as a mobile OS alternative to Apple's iOS in late 2008 and launched its first Android tablet in 2011. Over the years, Android has evolved significantly and become more complex due to its improved design and wide range of functionality added to each of the 12 versions released so far.

2.2.2 Android Updates

Although Android is built on a modified Linux kernel version, multiple disparate sources contribute to Android's overall source code and functionality. These include the Android Open Source Project (AOSP) code, AOSP external code (third-party, open-source libraries and daemons), platform or Board Support Package (BSP) vendor code, device vendor code (also known as ODM or Original Design Manufacturer), and in some cases, carrier code. Different codebase sources contributing to Android OS allow vendors to customise their design specifications, which is an appealing feature for both developers and end-users. But unfortunately, too many sources can also be problematic and cause fragmentation when providing consistent and reliable software updates and patches, and Google has been actively trying to cope with this in recent years.

Traditionally, the Linux kernel goes through three significant steps before Android is shipped on a device to the end-users:

- Google takes a pre-built and already maintained stable release of Linux kernel , adds additional Android-specific patches, and develops its own kernel branch called the Android Common Kernel .
- The ACK is shipped to a platform vendor, which provides a layer of hardware-specific drivers, also called a Board Support Package, for a particular System-on-Chip (SoC) implementation release. The most prominent SoC vendors include Qualcomm, Samsung Semiconductor, Huawei's HiSilicon, MediaTek, and Nvidia.
- A platform vendor ships its kernel version to a device manufacturer, who can choose to replace any SoC components or program them to support other features.

The ODM also adds other applications and kernel modules to enable different components on the motherboard, and it is responsible for physically designing a smartphone. The code that constitutes these additions is stored in the /odm partition, usually closed-source and specific to a device. Examples of device vendors include Google, Samsung, Huawei, Oppo, Motorola, etc. Each of these vendors customises Android to differentiate from one another, but they have very similar underlying systems. Any of these Android devices can also be purchased from a carrier, which can add their applications to the device or lock it to a particular SIM card. However, since December 2021, UK mobile phone companies have been banned from selling phones locked to their services . Figure 2.10 below summarises the traditional process of getting Android on a mobile phone.

Google has been tackling Android fragmentation in various ways. For example, in 2017, it initiated Project Treble, which re-architectured the Android OS framework by making it modular and separating the platform/chipset code for a specific device from the generic kernel through a universal interfacing layer called a Hardware Abstraction Layer (HAL). Eliminating this dependence required the chipset vendor to

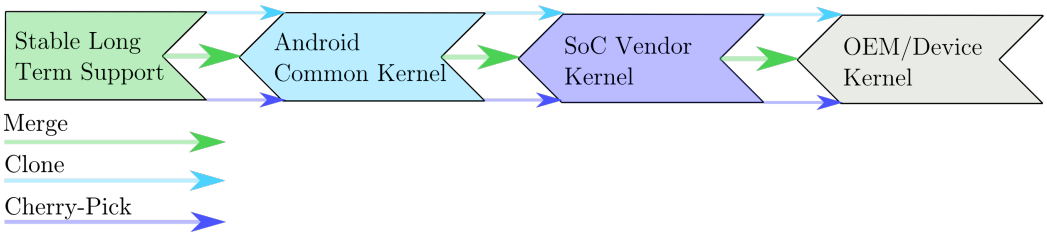


Figure 2.10: Android OS – Stages to Release.

update their BSP code to support Treble. This is beneficial as it allows any device vendor compatible with Treble to update to a new version of Android without waiting for a new code release from the chipset vendor.

Google had continued its work on Treble, and in 2019, it announced Project Mainline with Android 10 as part of Google I/O, an annual conference for developers. Whilst Treble worked on making system updates easier for device vendors, the Project Mainline initiative shifted the control of updating low-level system components to Google without involving the device vendor. The Project Mainline required the ACK to have an updated Linux kernel version of 4.4 or higher for devices to benefit from its upgradable modules. In this case, Google updates core components critical to security in the background, and some of them do not require a reboot. Traditionally, updating these components required a full over-the-air (OTA) update from the device vendor. However, Project Mainline allows updates to be delivered via Google Play Store through a new file type called Android Pony EXpress (APEX). This APEX file is processed by the APEX manager (or apex), which verifies, instals and uninstalls updates as required. Overall, Google converted some of the core components into modules, but only those that do not require further ODM customisation. The figure below shows the difference between Project Treble and Project Mainline and how Google prevents fragmentation by modularising some Android system components and updating them outside an OTA update release.

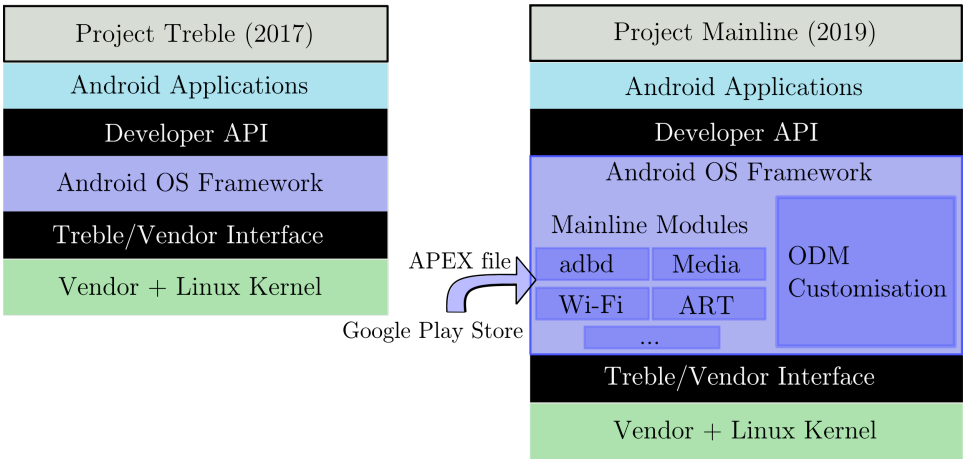


Figure 2.11: Evolution of Android Update System: Project Treble vs Project Mainline.

Three mobile phones are used in this project as part of two experiments presented in Chapter 3, each with a different Android and kernel version, as presented in the table below.

The baseband version relates to the firmware that controls the radio device or modem on the phone and allows it to communicate over the cellular network. Updates for this firmware are critical and, most of the time, included in standard OTA Android updates, which must comply with strict regulations. Without the correct baseband version of the firmware, Android would not know how to communicate with the hardware radio chip; therefore, the phone would not be usable.

2.2.3 Android Architecture

As an open-source software platform, also known as a software stack, Android consists of components at various layers of its architecture. Each layer handles specific operations supporting different functionality of this operating system by receiving calls from its higher layers. Figure 3 details the full-stack architecture

ODM / Brand	Motorola	OnePlus	Samsung
Model	Edge	Nord 2 5G	Galaxy S21 FE 5G (SM-G990B)
Model Number	XT2063-3	DN2103	SM-G990B/DS
Android Version	Android 11	Android 11	Android 12
Kernel Version	4.19.125-perf+	4.14.186+	5.4.86-qgki-22980446-abG990BXXU1BUK5
Baseband Version	M7250_HI20_06.18.02.74.03R RACER_ROW_CUST	M_V3_P10, M_V3_P10	G990BXXU1BUK5
Chipset	Qualcomm Snapdragon 765G	MediaTek MT6893 Dimensity 1200 5G	Samsung Exynos 2100
Platform Vendor / Chipset Manufacturer	Qualcomm	Mediatek	Samsung

Figure 2.12: Basic information on the smartphones used in this project.

for an Android device and highlights the communication between layers, enabling the functionality of the Android platform.

2.2.3.1 Hardware Components

Hardware components are included alongside the Android stack in Figure 3 to visualise better how everything fits together. Each hardware component is assigned to a kernel driver, interacting with the upper layers. Examples of hardware components include the baseband, Bluetooth, or wireless NICs, also called Wi-Fi chipsets, which are particularly important to this research.

Android's ability to adapt to most hardware platforms that its Linux kernel can support contributes to its popularity between vendors and developers. In addition to this, the main processor architecture used for most Android phones is ARM. In this case, chipset vendors like Qualcomm or Mediatek licence existing ARM core reference designs from Arm Holdings and use them to manufacture their SoCs (e.g., Snapdragon, MediaTek etc.).

2.2.3.2 Linux Kernel Layer

The first layer at the bottom of the Android stack is the Linux Kernel layer. Android relies on a modified version of the standard Linux kernel, compiled to ARM mobile architecture. The modified kernel provides core system services for managing the Android OS, such as driver management, which deals with low-level device drivers, such as Wi-Fi driver, Bluetooth driver, etc. Other features Linux provides are related to network management, user management, process management, file management, and security (e.g., SELinux, covered in Section). Android only resembles Linux at this layer, and it does not share any other features at higher layers.

In addition to the functionality provided by the Linux kernel, Android adds some enhanced features called Androidisms to the kernel core. For example, logging is added for different parts of the Android system through four different log buffers, which can be accessed from user space via different standalone applications.

Android provides a native command-line tool called logcat to read these logs from user space, which does not require root privileges to run. The binary of this tool is located in /system/bin in the local file system, and an end-user can use it to dump and view any system-wide log messages. Otherwise, it can also be accessed via adb logcat, another command-line tool used to communicate with a connected Android device. These log messages are helpful when trying to debug errors and display the stack traces associated with them.

Another example is Binder IPC, an inter-process communication (IPC) mechanism built into the kernel

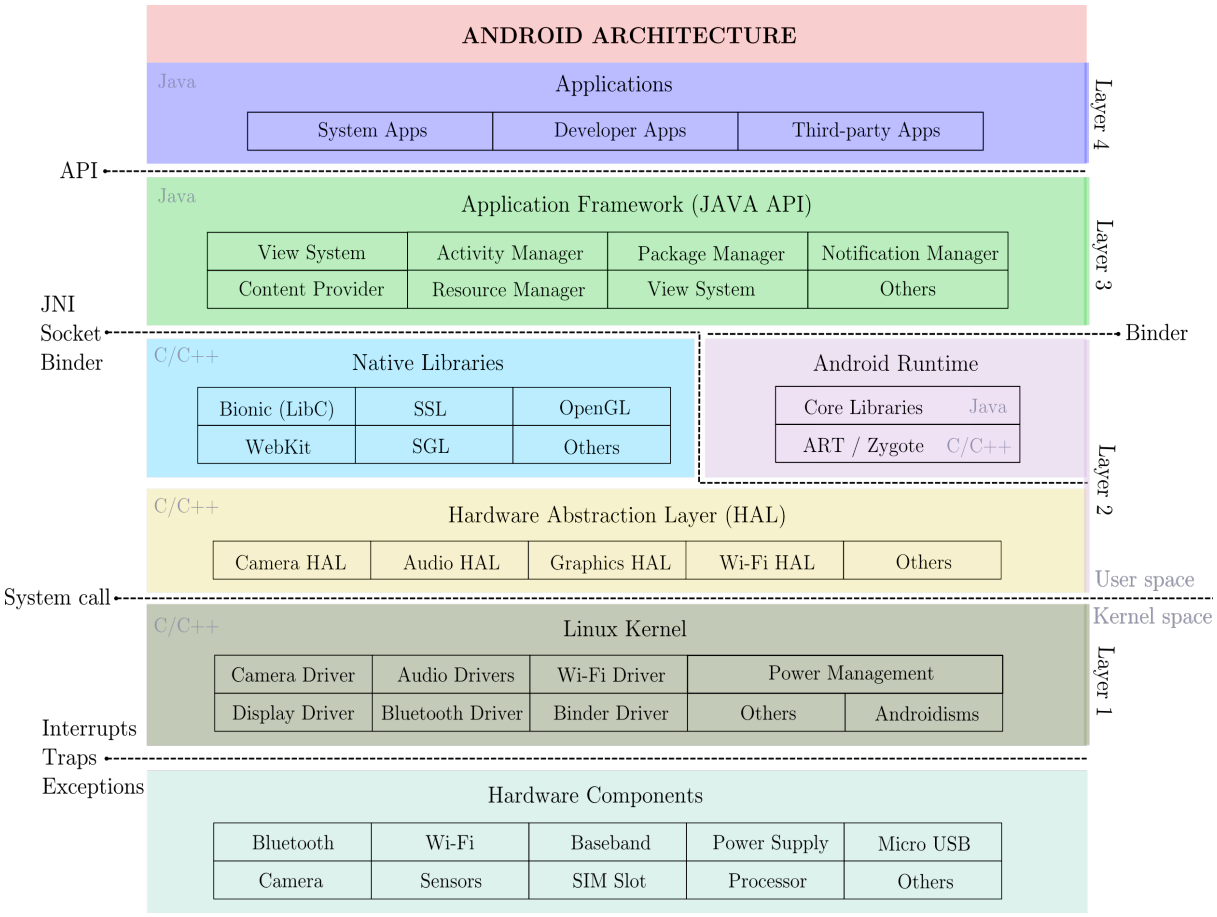


Figure 2.13: Full-stack architecture of an Android-based mobile device

as a driver and used extensively by Android. This allows system components or applications living in an isolated environment to discover and interact with each other.

Various other Androidisms are implemented necessary for mobile embedded platforms. Examples include wakelocks to control power management, and memory management mechanisms, such as Low Memory Killer or the ION Memory Allocation. These are used to better preserve memory by avoiding memory exhaustion and memory fragmentation for hardware with special memory requirements, such as cameras or display controllers.

2.2.3.3 Hardware Abstraction Layer (HAL)

On top of the Linux Kernel Layer is the Hardware Abstraction Layer (HAL), allowing Android to be as hardware agnostic as possible and portable to most hardware devices. As shown in Figure 3, HAL is a user space layer. Any user space code in the Android Framework that needs to interact with a hardware device first calls its HAL interface using Binder IPC, which talks directly to the Linux device driver in kernel space. The main difference between user and kernel space relates to memory and access rights, with the kernel space being strictly reserved for privileged operations and device drivers.

Google provides design specifications using Android Interface Definition Language (AIDL), introduced with Android 11, or Hardware Interface Definition Language (HIDL), which is the main focus in this section. Each device vendor is responsible for following these specifications when implementing a specific hardware device HAL, also called a HAL module, which exposes the hardware functionality to the Android Framework.

Therefore, if a device vendor chooses to support a particular hardware device and provides its hardware driver in user space, it does not have to release its proprietary source code. However, a device vendor can also use existing general HALs implementations provided by Google. These are primarily located in

Android's directory tree under hardware/interfaces in .hal files, which can be compiled, and corresponding interface implementations can be automatically generated. Furthermore, when a software package from a specific directory is compiled, e.g., hardware/interfaces, it is directly mapped to its equivalent namespace, e.g., android.hardware package namespace. This helps to identify a HAL type and version (e.g., vendor, system, external etc.) and where its source code is located. Other HAL implementations can be found in hardware/libhardware/, hardware/libhardware legacy/, or hardware/.

Section 2.2.5 covers the HALs involved in enabling Wi-Fi connections, which are essential for this research.

2.2.3.4 Native Libraries and Android Runtime Layer

This layer is divided into two main parts: native libraries and the Android Runtime environment.

First, an Android system depends on various native libraries compiled and pre-installed by Google, chipset or device vendor for a specific hardware architecture, such as 32-bit and 64-bit. These libraries are non-Java libraries written in a native programming language (C/C++) and provided as part of the open-source AOSP (Android-specific libraries) or as part of AOSP External's codebase (open-source, third-party libraries) to support the Application Framework and system processes. Therefore, native libraries are dynamically linked ELF (Executable and Library Format) shared objects, which are executable libraries available to be used by multiple applications, and loaded on-demand.

Generally, Android applications can be written in Java, native code, or both. The Android Native Development Kit (NDK) provided by Google facilitates native code application development and provides Java applications with the additional functionality to embed native libraries in their code.

One prominent native library is Bionic, a custom C-runtime library written by Google. This is equivalent to the standard C library in Linux called glibc but smaller in size and tuned to mobile computing. Bionic implements a native API containing all the system utilities that most applications need to run on Android. This native API interface consists of system calls made to the kernel on behalf of a user application or process. For an Android application to communicate with the kernel, all its native code must be compiled against Bionic. The utilities provided by this native library are related to string manipulation, mathematical computations, memory management, input/output processing, multi-threading, users and groups, etc. In addition, Bionic includes signal handlers registered to processes whenever a dynamically linked executable starts. These signal handlers capture deadly signals in the event of a crash and contact debuggerd, crash_dump32 or crash_dump64, which are crash-handling daemons which record the information using ptrace and /proc. These daemons also facilitate a crash dump or a tombstone file written in /data/tombstones/ when a process crashes.

Other native libraries include function libraries, such as WebKit, Media Framework or SQLite, or native servers, such as SurfaceFlinger, or AudioFlinger.

Native libraries have a .so extension for each hardware architecture, and they can be accessed/called in an application running on a virtual machine through a dedicated interface called Java Native Interface (JNI). Besides allowing Java code to call native code, JNI also enables native code to call Java code. This interface was initially designed for Dalvik Virtual Machine (DVM), the Android-specific virtual machine used to execute an application compiled to Dalvik bytecode at runtime. The DVM is an interpreter-based virtual machine written in C that uses Just-In-Time (JIT) compilation to translate most Java bytecode to Dalvik bytecode every time an application is launched. In addition to this, some of the Java code is compiled dynamically at runtime, which is beneficial for mobile devices with less storage. An Android-based smartphone can run multiple applications concurrently, and each of them has its own DVM instance. This virtual machine relies on underlying system processes for security, isolation, memory management and threading support.

Although Google uses Java extensively as part of its Android platform, it does not use Java Virtual Machine (JVM) as a runtime environment where Java bytecode can be executed. One of the reasons for this is licensing issues with Oracle , which caused Google to introduce the DVM, which was discontinued in 2014 and replaced with the Android Runtime (ART) virtual machine.

Unlike DVM, ART uses an Ahead-of-Time (AOT) compiler to translate a whole application code into native code only once during its installation. This makes the startup time for each application faster and

improves a mobile device's battery performance and garbage collection by optimising memory usage. The difference between the role DVM and ART play in an application's installation and execution process is highlighted in Figure 2.14.

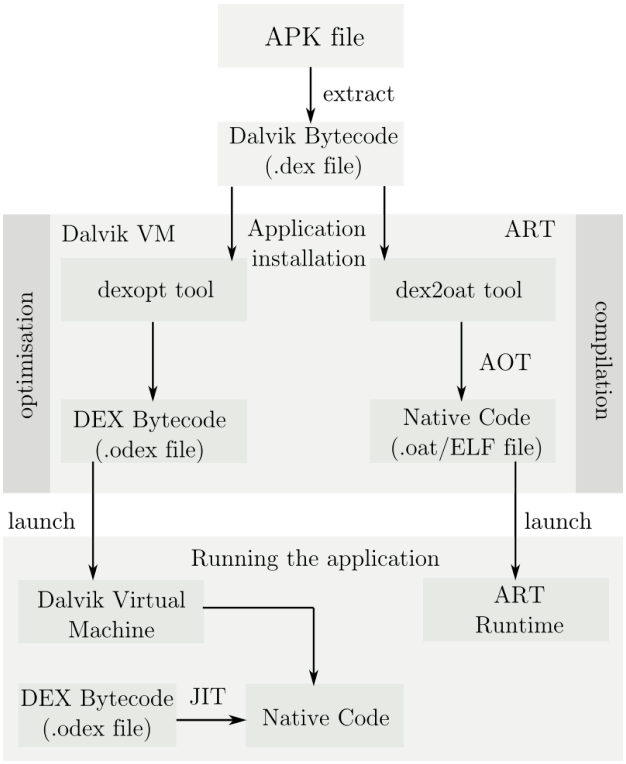


Figure 2.14: DVM and ART architectures compared in the installation/execution process of an Android application

There are various core libraries at this layer that are part of the Android Runtime of an application. These libraries can be core Java or Android-specific libraries, which may contain native code dependencies, and they are extensively used compared to the native ones presented above. For example, core Java libraries are found in java.* and javax.* packages and include concurrency, synchronisation, networking, I/O and other relevant classes etc. In addition to this, the core Android-specific libraries are available in android.* package for managing application development and lifecycle. They cover various functionality related to facilitating access to a user interface and application model, rendering the text on display, graphics drawings, database access etc. These core libraries are accessible from the Applications layer and system services.

This native userspace layer also contains native daemons running in the background, supporting various application framework system services, such as core or main services. These native daemons are primarily located in /system/bin and /system/sbin and include some of the following:

- adbd (Android Debug Bridge) is a critical native daemon present in the Android startup configuration file that provides server functionality of the Android Debug Bridge (ADB) tool, which allows debugging of Android-based devices. adbd can only be started on demand through USB Debugging or other options , and if enabled, it will allow an adb host to communicate via USB or Ethernet with the device. This native daemon can also write trace files in /data/adb on the target. We use ADB tool in this research, and it is essential to understand its architecture, briefly presented in the Figure 2.15.
- servicemanager (Service Manager) is a critical native daemon initialised at Android startup, and it manages all services in the system after they are started.
- installd (Installation daemon) handles the installation of packages for the Package Manager Service.
- app_process is a native daemon responsible for creating an instance of ART, which further initialises the JNI and invokes the zygote process, considered the parent process of all application processes.

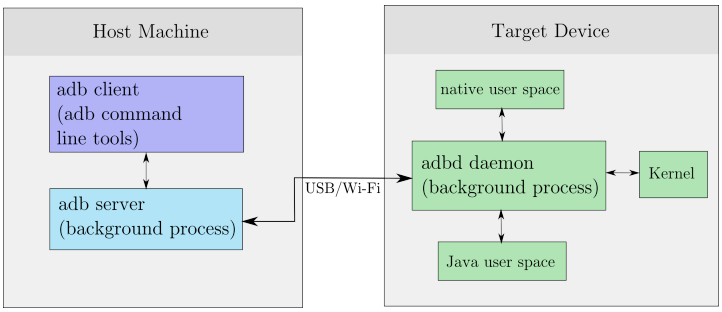


Figure 2.15: Android Debug Bridge (ADB) Architecture

2.2.3.5 Application Frameworks Layer

Above the Native Libraries and Android Runtime layer is the Application Frameworks layer, which provides system services, exposing the functionality of lower-level resources, such as native libraries, to Android applications. These rich open-source frameworks are not part of the Android Runtime. Instead, they include other extensively-documented Java libraries specifically for developing applications and classes, such as Notification Manager, Location Manager, Resource Manager, Package Manager, Connectivity Manager etc. The latter represents the Network Connectivity Service, which allows applications to query or receive information about the state of network connectivity, configure settings and control the network radios. Therefore, this layer provides APIs which handle graphics, audio and hardware access, run background services, etc. It also simplifies an Android application architecture by introducing the following Java classes as basic building blocks or components:

- The Activity class is responsible for creating a window containing the user interface for an application.
- The Service class allows an application to run tasks in the background, such as updating a database, streaming movies, or downloading a file over the internet. Services are launched by various components like activities or broadcast receivers, but they are not tied to a user interface.
- The Broadcast Receiver class enables applications to receive and handle broadcast messages when events occur between various Android components or the Android system and an application. These broadcast messages are known as broadcast intents. A broadcast receiver listens for such messages to react to any system or application events. For example, an application can use a broadcast receiver to receive notifications about the change in Wi-Fi connections. The broadcast receiver must be registered as a receiver in the application's manifest file (AndroidManifest.xml), containing metadata about an application.
- Content Provider class handles access to specific data on the storage device by providing a well-defined set of APIs for applications to use, which allow them to read, insert, update or delete the data. The content provider can store data internally in a local file, local database, or remote server.

The interaction between components is summarised in Figure 2.16.

Interfacing with the Application Framework requires various tools and commands issued within the adb shell, which helps developers log into a device as a non-privileged shell user to access advanced settings and debug capabilities. For example, the Activity Manager (am) binary triggers various system events, such as launching an activity or broadcasting an intent. Using the Package Manager (pm) tool within the ADB shell allows managing system or user application packages installed on the device. Other commands include service list (shows running system services) or the svc binary. This controls and debugs the Power Manager, Wi-Fi Manager, mobile data connectivity, or the USB state of a mobile device.

2.2.3.6 Application Layer

The topmost layer of the Android stack is the Application layer. This is composed of applications the end-user uses, distributed as Android PacKage (APK) files. The Android platform uses files with the .apk extension to distribute and install mobile applications, which can be pre-installed or user-installed

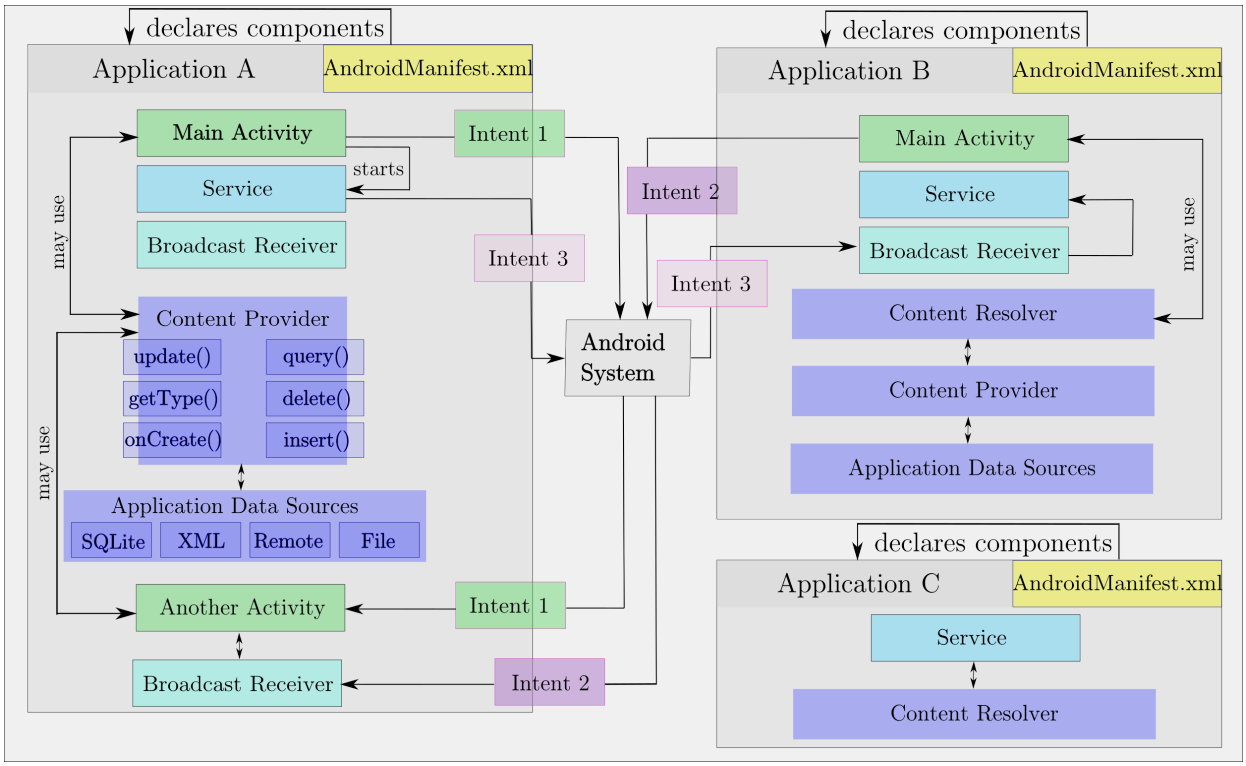


Figure 2.16: Android application inter-components communication

on a mobile device. The system pre-installed applications are shipped with the device. They include the official applications provided by Google, OEMs, and in some cases, mobile carriers. For example, several applications come installed with a device, such as default browser, e-mail, SMS clients, calendar, or contact manager. The user-installed applications are downloaded from third-party application markets, such as Google Play Store, but they can also be non-market applications installed manually.

The .apk format of all Android applications is a modified ZIP format categorised as an executable file. The user-installed applications are signed with the developer's certificate before installation to prevent unauthorised application updates. Similarly, the pre-installed applications are signed with a special platform key, which gives them system user privileges. This compressed file includes the AndroidManifest.xml file placed in the application's filesystem root, application code (.dex files), external libraries and resources. The structure of an Android application binary (.apk) is illustrated in Figure 2.17.

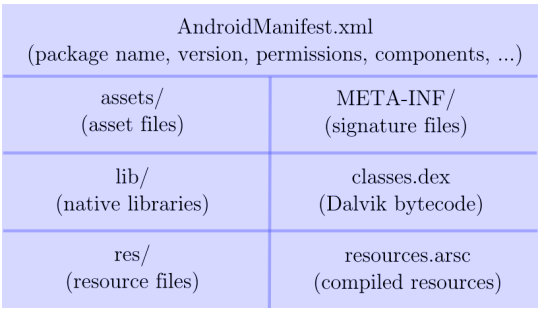


Figure 2.17: Android Application/APK Package Structure

Developers can write their applications in Java, Kotlin (converted to Java bytecode) or native code (C/C++), but each of them runs inside a virtual machine on the mobile device. As mentioned previously, the AndroidManifest.xml declares essential information about the application itself, including its components (activities, services, broadcast receivers, content providers etc.). It also contains a list of permissions, giving the application access to specific device components. However, these permissions are explicitly granted by the end-user at installation time through permission pop-ups. Permissions and

sandboxing are further discussed for this research as part of Android's security model.

2.2.4 Architecture Security Model

This section covers aspects of the Android security model relevant to this research. For example, understanding security mechanisms such as user permissions, sandboxing, or Security-Enhanced Linux (SELinux) are essential to performing vulnerability research. This is because all security mechanisms are closely related to significant vulnerabilities, which an adversary could try to exploit. For example, potential adversaries seek to gain elevated privileges upon a successful injection attack, and this section discusses security measures and controls to minimise unauthorised access and malicious activities.

2.2.4.1 Android Build System

Before looking into security mechanisms further, it is essential to note that, like any other Linux-based distribution, the Android Open Source Project (AOSP) distribution uses a build system to define and control the process of building its codebase before publishing to customers. According to the official documentation, the AOSP uses the Soong build system introduced in Android 7 to replace a Makefile-based system . The Android source tree provided and hosted by Google contains source code organised in different Git repositories, including third-party open-source libraries and other in-house tools (adb, fastboot, etc.) used by the Android system. In this context, Soong bundles the Android codebase into build variants, which have to be compiled into a final file system image, customised and compressed by a device vendor into a distribution or "factory" image installed/flashed on a mobile device. Unpacking the factory image reveals several image files, such as system.img, cache.img, boot.img, recovery.img etc., which correspond to Android's /system, /cache, /boot, and /recovery partitions, respectively.

Having multiple build variants offers device vendors the flexibility to customise and configure the AOSP build for various purposes. For example, device vendors can create builds with different packages and functionality from the same AOSP source tree by using configuration and definition files to specify environment variables and build properties.

Two essential items that must be set for a build are the build variant (user, userdebug, eng) or build type (release, debug). The build variant options customise what modules and debug functionality should be included in the Android system image to be flashed on a mobile device. This customisation primarily revolves around .prop files, which contain all the information about a mobile device (system version, firmware version, device model, etc.) and build settings and properties set by Google and a device vendor. These .prop files are default.prop and build.prop, generated at boot time and usually located under the root and /system directories of the Android filesystem. Since the Android root filesystem is mounted read-only, and system properties are loaded on boot from the build.prop file, editing any system properties requires both elevated privileges and repacking/flashing the modified boot.img.

Some of the main differences between the build variants available are the following:

- The eng build variant instals modules tagged as "eng", "user", or "debug", whilst the userdebug build variant instals the "user" and "debug"-tagged modules. The user build variant only includes the modules labelled as "user". In this context, modules refer to any AOSP component (e.g., binary, library, package etc.) that needs to be built for a target mobile device.
- The eng build variant has adb and root access enabled by default, by setting ro.secure=0 and ro.debuggable=1 in default.prop and ro.kernel.android.checkjni=1 in build.prop. On the other hand, the user build type has the adb and debugging disabled by default, but the end-user can access adb debugging by manually enabling it in settings. Although the userdebug build comes with adb debugging enabled and root shell access, it does require the end-user to enable them manually through settings configuration.

As part of Android's security model, the end-user is limited from removing system applications, controlling system services, modifying read-only partitions, bypassing the Android sandbox, accessing the entire OS etc. These limitations are enforced by flashing only user build variants to mobile devices, with multiple security features inherited from the Linux kernel. These include the user-based permissions model, process isolation through sandboxing, and Security-Enhanced Linux (SELinux), an optional feature available in the standard Linux kernel. Allowing root access by default and enhanced debugging

functionality in release builds violates the security model of Android, which is why the eng and userdebug build variants should only be used in development.

2.2.4.1.1 Android Firmware Structure As mentioned in the previous section, Android simplifies the overall process of mobile device configuration and testing, including compiling and building its firmware images, by introducing the Soong build system. Soong helps compile a final system image, which generates various firmware images necessary to create a working Android system.

There are various ways of installing these images, and they depend on the bootloader used by the device vendor. For example, most device vendors use Qualcomm's Little Kernel Embedded Operating System (LK) , a mini bootable image flashed to the bootloader partition in the eMMC. The bootloader is responsible for powering the mobile device, loading the kernel into memory and passing it the control to initialise the rest of the Android OS. Device vendors provide protocols at the pre-bootloader or bootloader level, but the default LK's protocol is Fastboot, used for flashing firmware images, which is included in the Android SDK. The only device used in this research not using this protocol is Samsung, which uses an alternative proprietary mode called Odin. More about the role bootloaders play in the boot process is presented in the next section.

The firmware image files mentioned play different roles in the boot process, and to reduce their size, Android compresses them in its own type of installation format called Android Sparse Image Format (ASIF) . The compressed sparse images contain the code for the entire Android system, but they need to be converted to mountable raw disk images (.img.raw) before flashing/installing them to the device file system directly. This is done using simg2img , a command-line tool included in the AOSP project, which also provides the tools required to flash the boot.img file, responsible for booting the Android OS to the /boot partition on eMMC . Android divides its persistent data across eMMC (non-removable, non-volatile NAND flash) and external storage. eMMC is the memory on the motherboard reserved for crucial system files mounted as read-only. The external storage, in this case, can be a portable microSD card or a built-in chip for internal storage.

Some of the important firmware images flashed to their corresponding partitions on a device running at least Android 10 include:

- system.img - This is the partition image mounted as a read-only /system directory in the system's root file system and stores the entire Android OS, including permission information (e.g. permissions.xml, SELinux policy files etc.). A device can only boot in recovery or bootloader mode if this partition is wiped. As of Android 10, the root file system, which used to be part of the ramdisk.img (in boot.img) merges with this image at build time, which mounts /system as the root file system (rootfs) . Another recent addition is the introduction of a dynamic partition called super.img, which encapsulates several partitions, such as the system.img, vendor.img, and product.img. The super.img image contains both the root-as-filesystem and the system.img.

- boot.img - This image contains the Linux kernel and ramdisk used for normal boot, and it is flashed to the /boot partition, which is only mounted during the boot process and not when Android OS is running. A device would not boot without this partition.

- recovery.img - This image is independent of other images, and it is used as an alternative to the /boot partition to perform advanced recovery operations (e.g., loading and flashing new kernel and system images etc.) or maintenance on a device. The /recovery partition is also only mounted during the boot process when booting into recovery mode.

- vendor.img - This image is mapped to the /vendor directory in the root's file system, containing proprietary binaries and vendor-specific files.

The boot process in Android requires several other non-mountable partitions presented below, aside the boot.img and recovery.img already mentioned.

- boot partition - primarily covered above.

- recovery partition - primarily covered above.

- vendor_boot - Starting with Android 11, all vendor-specific information is relocated from boot to vendor_boot with the introduction of Generic Kernel Images (GKIs) , a generic ARM64 kernel for all Android devices based on ACKs. Therefore, this partition does not contain a kernel, just the vendor ramdisk used alongside a GKI image. The Android Verified Boot also protects this partition.
- vendor_boot partition - Multiple Device Trees Overlays (DTSs) are in the kernel source to support different hardware configurations. The bootloader uses these data structures at runtime to tell the kernel which hardware devices exist and how they are connected to the system.
- dt[b]o partition - Multiple Device Trees Overlays (DTSs) are in the kernel source to support different hardware configurations. The bootloader uses these data structures at runtime to tell the kernel which hardware devices exist and how they are connected to the system.
- frp partition - It was introduced with Android 5. Factory Reset Protection (FRP) is a security feature which, if activated, prevents the use of a device after a Factory Data Reset by locking it. This special region of persistent state is not wiped unless the device is authenticated with the "correct" Google account. The bootloader loads the partition to check the FRP signature.
- misc partition - Introduced in Android 9. This partition image is used by the recovery partition to communicate information with the bootloader, and controls whether Android boots in Recovery mode.
- vbmeta partition - Implemented after Android 8 as part of Android Verified Boot (AVB), this is an image flashed to the /vbmeta partition containing signature metadata and loaded by the bootloader to verify other images, such as boot.img, system.img, and other partitions/images. More about the AVB is covered in Section 2.2.4.2.1.

2.2.4.2 Android Boot Process

Booting a mobile device refers to the sequential process that starts the Android OS of a mobile device as initiated by the hardware (e.g., button press), and by software. For example, from a hardware perspective, an Android device executes the following three steps:

2.2.4.2.1 Secure Boot and System Integrity In Section 2.2.4, we briefly mentioned some of the functionality Android implements to provide and improve a device's security. The Android Verified Boot (AVB) is a fundamental part of the overall security model, and it is necessary to understand when it comes to unlocking a bootloader or rooting a device. AVB was introduced in Android 4.4 with the dm-verity kernel feature (a device mapping verification) and modified with Android 8.0 to ensure compatibility with Project Treble architecture. AVB helps establish a complete chain of trust starting from the Android Bootloader stage in the boot process and covering all the firmware images and OS partitions. It does that by leveraging cryptographic checks in the aboot of the Android Bootloader to verify the integrity of the firmware images and OS and ensure that they have not been modified from the factory version. If this is the case, AVB and dm-verity prevent the compromised devices from booting into the Android OS and make them permanently inoperable, also known as bricked devices. AVB also uses forward error correction to protect the kernel, partition table and non-volatile system memory from non-malicious data corruption.

Some Android phones have bootloaders which can be unlocked to allow end-users to replace the original Android OS with a custom one. However, this is considered a large security risk and not recommended as it disables the integrity checks, which help detect and defend against unauthorised changes.

2.2.4.2.2 Privilege Escalation Privilege escalation or rooting refers to gaining superuser or root capabilities on an Android system. This is usually seen as an option by end-users looking to overcome limitations related to customising their mobile devices imposed by device vendors. For example, removing bloatware/pre-installed applications requires rooting the device, which none of the vendors recommends as it undermines Android's security model. However, for this research, we root two devices (Motorola, OnePlus) to be able to access all the Android OS, in particular the proprietary vendor code related to Wi-Fi, which is usually closed source. Rooting is also helpful in this case to perform further debugging and access logs when more granular information is needed.

Because Google does not offer Android versions with root permissions, end-users have to find a way to escalate their privileges if they want to customise their devices. However, Android takes security further by using a multi-layered security approach to guard against applications attempting to acquire root privileges. This approach includes some of the following features:

- Discretionary Access Control (DAC) (covered in Section 2.2.4.3) - this is inherited from Linux and implements access control for the system, individual applications and their storage based on User ID/Group IDs and implements read/write/execute object permissions.
- Mandatory Access Control (MAC) (covered in Section 2.2.4.4) - this feature adds additional controls to restrict access to processes and system resources based on predefined labels and policies.
- Linux Capabilities (CAP) – this feature breaks down permissions for both regular and root users to ensure that they only have the capabilities needed to operate.
- seccomp – this security feature inherited from Linux closes down entry points to the Linux kernel by acting as a system calls filter.
- Android Middleware (covered in Section 2.2.4.5) - is related to application-level permissions defined in the AndroidManifest.xml file and enforced against MAC policy configurations at installation time.

Fundamentally, getting root access via exploiting an application or kernel vulnerability requires bypassing more than one of the layers presented above, depending on the exploit target. This makes it challenging for an adversary and limits the legitimate user seeking to take control of their device restricted by a device vendor (e.g., Samsung etc.).

However, one of the popular options available to get root privileges requires unlocking the bootloader, but as previously mentioned, this option is available only on some devices. Removing the bootloader lock enables the end-user to use a custom recovery image specific for a device and containing new features, such as uploading ZIP files to the device. TeamWin Recovery Project (TWRP) [1] is an open-source recovery image for Android, which offers custom recovery images for a range of phones. A custom recovery image can be manually flashed on the device using Fastboot utilities to enable the new functionality necessary in the rooting process. A rooting tool must be used with the TWRP recovery image to get root access, and this can be Magisk or SuperSU. One of the benefits of Magisk is that it uses a "systemless" root that does not change any of the /system files but only the boot partition. Unfortunately, this tricks the Google SafetyNet provided with the Google Play Services to check whether a phone is rooted or not, which can block access to features like Android Play. On the other hand, SuperSU replaces the system partition entirely, which often triggers SafetyNet and causes security sensitive applications to stop working.

It is crucial to understand that rooting a device leads to voiding its warranty in most cases, and it also comes with security risks, as it loses some of the security protections provided out-of-the-box, including security updates to keep potential vulnerabilities.

2.2.4.2.3 Samsung Samsung took SELinux further and used it as a foundation to implement their own security mechanism, called Knox [2]. This is a secure container framework leveraging Samsung's proprietary version of a Trust Execution Environment (TEE) built on top of ARM TrustZone. Knox supports system-wise isolation at both software and hardware levels by building a TEE into the main CPU of a mobile phone through a Knox hardware chip (eFuse). This creates a trusted execution environment for security-sensitive code, and it comes pre-installed and built into Samsung's version of Android. Attempting to remove Knox and/or unlock the bootloader (or boot program) to obtain root access to a mobile device voids the warranty and stops applications using Knox security from working (e.g., Samsung Pay, Secure Folder). Knox uses an electronic fuse with Knox Warranty Bit, which detects if an unsigned kernel is loaded during the boot stage or startup of an Android device. If this happens, the one-time programmable Knox bit trips and turns from 0X0 to 0X1. Consequently, an application used in the TrustZone would detect this change and delete the encryption key necessary to access its encrypted data, which is not recoverable to prevent unauthorised access.

Other security features implemented with Knox in the kernel to mitigate privilege escalation include Kernel Address Space Layout Randomisation (KASLR), Data Flow Integrity (DFI), and SELinux enhancement. Samsung also moved away from the traditional fastboot protocol for communication with a

device in bootloader mode. Instead, as mentioned in Section 2.2.4.1.1, it opted to implement its dedicated pre-boot and communication platform called Odin, which provides several security enhancements and typically restricts functionality more than the standard fastboot mode.

One of the devices received from the university to research as part of this project is a Samsung Galaxy S21 FE 5G. However, because of the dangers of blowing the eFuse when rooting the mobile device and voiding its warranty, it will only be used partially as part of Experiment A. For most devices, including Samsung, the device vendor locks the bootloader to prevent boot and recovery partitions from being written. Unlocking it disables the signature verification protection every time a mobile device boots up and allows it to be rooted. However, it is not always necessary to unlock the bootloader for rooting, as it can also be done through an exploitable kernel vulnerability or third-party exploits from the Android community [4]. This will not be done in this case to avoid any potential technical or legal issues, like avoiding the warranty or hard bricking the phone.

2.2.4.2.4 Motorola One of the mobile devices in this research is a Motorola Edge, which comes with a default bootloader locked by its device vendor, Motorola Mobility LLC. Since we only need root privileges to access the whole Android file system and analyse the behaviour of the Wi-Fi functionality, rooting such a device would be possible without unlocking the bootloader. However, this process can cause serious issues on the device as it uses unstable exploits, in the event the device partitions become corrupted and the bootloader is not unlocked re-flashing becomes an issue and the device may be bricked, for Motorola phones in general, it is recommended to unlock the bootloader first to eliminate further risks related to crashing or corrupting the whole system. The device vendor has developed an official method of unlocking the bootloader by requesting an unlock key on their official website [5], but this requires accepting a legal agreement which voids the device's warranty and stops it from receiving over-the-air (OTA) updates. However, there are no other reliable methods for unlocking Motorola Qualcomm devices' bootloaders developed by the Android Developer community. Therefore, the official method is preferred since the university does not provide this phone.

Rooting this Motorola device requires the following guides [x] [x], and the steps are summarised in Summary 1. This method uses the official way of unlocking the bootloader, Magisk and TWRP Recovery tool. In addition, the steps for removing root from the device depend on the tools used as part of the rooting process, and in this case, they are explained in other tutorials [x] [x]. Unfortunately, locking the bootloader for this Motorola device is complex, and there is no official way of doing it at the time of this research. Instructions for unrooting the device are presented in tutorials, such as [x].

1. Take a complete phone backup, as unlocking the bootloader will wipe off all the data from the device.
2. Go to Settings - About Phone, and tap 5 to 7 times on the Build Number to activate Developer Options.
3. Go back to Settings - System - Advanced - Developer Options, and enable USB debugging and UEM Unlocking.
4. Download the Android SDK Platform Tools (ADB & Fastboot) and Motorola USB Driver and install them on the PC.
5. Connect the device to the PC.
6. Open the CMD prompt/terminal, and navigate to the location of platform tools.
7. List connected devices: adb devices
8. Boot the device into Bootloader/Fastboot mode: adb reboot bootloader
9. Check the device is in Fastboot mode: fastboot devices
10. Generate and save to a separate file the string needed to retrieve the unlock key for the device: fastboot oem get_unlock_data
11. Go to the Motorola Bootloader Unlock website [50], create/login into a Motorola account, and follow the instructions to generate an unlock key. If the device is unlockable, a 20 character alphanumeric key is sent to the email.

12. Back to the CMD/terminal, unlock the bootloader by issuing the command: fastboot oem unlock ¡paste-your-unlock-key-here¿
13. Reboot the Android device into Recovery mode: fastboot reboot
14. Download the latest official TWRP Recovery image, and rename the .img file to twrpracer. (https://eu.dl.twrp.me/ra 3.5.2_10-0-racer.img.html)
15. Enter the following command to install/flash the TWRP image on the device: fastboot flash recovery /path/to/twrpracer.img
16. Reboot from Fastboot mode into Recovery mode by entering the following command: fastboot reboot recovery
17. Immediately press the Volume Up + Power button simultaneously to boot into TWRP Recovery mode instead of the "normal" OS.
18. Download the latest official flashable Magisk file and copy it to the phone's internal storage, connected to the PC. (https://magiskapp.com/zip/)
19. Click install on the TWRP, and swipe right to flash the Magisk ZIP file.
20. Once the flashing process completes, on TWRP main screen, press Reboot ¿ System to reboot to the Android OS.
21. Verify that rooting privileges work by using the ADB utility installed in Step 4. adb shell su
22. Step 21 prompts the user device to give root privileges to the shell user, which means that the Motorola device is successfully rooted.

2.2.4.2.5 OnePlus One of the phones provided by the university for this research is a OnePlus Nord 2 5G. This device also comes with a locked bootloader; however, unlocking it does not void its warranty, unlike Motorola and Samsung devices. According to the OnePlus warranty policy [12], it only covers the hardware components and manufacturing defects and partially covers software, which means that damages caused otherwise are out of warranty. In addition, onePlus offers an official method of unlocking the bootloader using unlock tokens, which is explained in one of their support articles [9]. However, this is only available for devices in the US, and outside this region, bootloaders use official ADB and Fastboot utilities.

1. Full device backup, including internal storage.
2. Download and manually install ADB and Fastboot Tools and OnePlus USB drivers to a PC.
3. Enable USB Debugging mode and OEM unlocking on the device.
4. Unlock the bootloader and root the device by following instructions provided in an XDA Developers' community post [12].

It is possible to unroot the device [6] and relock the bootloader by installing the original firmware and disabling the OEM Unlocking option in Settings, as tutorials such as [7] [8] help achieve this.

2.2.4.3 Application Sandbox

A critical security mechanism implemented by Android at the kernel layer but extended to the Android Framework is sandboxing. Fundamentally, this means that applications are installed and executed in isolated environments to reduce the attack surface if a vulnerability is found and exploited. As previously mentioned, Android inherits from Linux the concept of user-based protection, so it assigns application resources a unique user identifier (User ID/UID, identifies applications) and group identifier (GID, affects permissions to access resources) at install time. In this case, applications include anything from system or user-installed programs to daemons or processes like system_server or zygote. This security mechanism isolates them from one another and from the system, so they cannot access memory outside their bounds. Different predefined ranges of UIDs and GIDs are used for separating both applications, and physical device users, which is why Android takes this further and defines more friendly names for combinations of UIDs and GIDs called Android IDs (AIDs). A device vendor usually generates this mapping (numeric

to non-numeric) at build time. However, it is also defined for privileged and system-critical users in the system/core/include/private/android_filesystem_config.h file in the AOSP source code. Some examples are highlighted below:

```
#define AID_ROOT 0 /* traditional unix root user */
#define AID_DAEMON 1 /* traditional unix daemon owner */
#define AID_BIN 2    /* traditional unix binaries owner */
...
#define AID_INPUT 1004  /* input devices */
#define AID_LOG 1007  /* log devices */
#define AID_WIFI 1010 /* wifi subsystem */
#define AID_ADB 1011  /* android debug bridge (adbd) */
#define AID_SHELL 2000 /* adb and debug shell user */
...
#define AID_APP_START 10000 /* first app user */
```

The UID and GID-based isolation is based on a Discretionary Access Control (DAC) model. This contributes to Android's adherence to the "least privilege" principle by implementing identity-based access control. Android employs DAC to assign access permissions to resources owned by a UID. However, although this mechanism isolates an application's memory space and resources, it also has several inherited weaknesses, which potential adversaries could leverage. For example, processes running as root with the UID of 0 are unsandboxed (e.g., /init), compromising the whole system by performing privilege escalation and continuing with various adverse actions, e.g. using ptrace to change the UID of any other running processes. /init is a management process first invoked by the kernel with PID 1 and executed with a UID of 0. This process has no parent and is primarily responsible for starting and shutting down the system. If /init dies, the Linux kernel panics/halts because there is no user space to run.

Therefore, some of the critical shortcomings of DAC are related (but not limited) to its inability to prevent vulnerabilities caused by vulnerable processes or confine actions of system daemons running with root privileges.

2.2.4.4 Security-Enhanced Linux (SELinux)

The Security-Enhanced Linux (SELinux) strengthens the UID-based DAC sandbox starting with Android 4.3 by introducing the Mandatory Access Control (MAC) framework, a feature inherited from Linux and enforced at the kernel level. "SE for Android" was the project which included, but was not limited to, enabling SELinux in Android to address the security gaps in sandboxing based on UID/GID mechanism.

Android uses MAC to sandbox applications, particularly core system daemons with root privileges, by providing an access control framework to define only the necessary permissions for them to be functional. This framework works on two principles, the principle of least privilege and the principle of default denial. Therefore, it defines and performs policy checks in the kernel, limiting all processes from accessing system resources without a predefined policy. SELinux decides what operations are permitted based on both the security context/label of the subject, which is the process initiating action and the object, which is the type of resource target. These labels are mentioned in policy rules that SELinux enforces and come in the form of "allow subject object:class permissions;". In this format, "class" refers to the kind of object being accessed (e.g., file), while "permissions" refer to different types of access (e.g., read, write, open) for each class. An SELinux label can also offer additional information about an object, such as the SELinux user, their role, their type, and the security level, which is used to control access by process, user, and file.

By default, SELinux is configured when building Android and enforced by the /init process when starting the device. In this case, chipset or device vendors are allowed and encouraged to add additional policy files in their corresponding directories to reflect device-specific requirements. Generally, a policy can be a single file policy or a multiple-file split policy. Split policies require the /init process to spawn the SEPolicy compiler (/system/bin/secilc) and write them to the kernel. During the compilation process, checks are performed against "neverallow" rules to ensure that specific access is never allowed by a vendor policy. These permissions marked as "neverallow" are provided by default with the AOSP policies and implemented across all devices. If conflicting "allow" rules are found, they take precedence and interrupt compilation.

One of the key benefits of SELinux is enforcing permissions to features specific to Android, such as global system properties, briefly discussed in Section 2.2.4.1, and Binder IPC, covered in Section 2.2.3.2. For example, the /init process reads the /system/sepolicy/private/property_contexts context file during startup and labels properties to only reflect what other processes can set them. Similarly, the /system/sepolicy/private/service_contexts context file is read at startup by the /servicemanager process started by the /init parent process. /servicemanager registers itself as the Binder context manager via the /dev/binder interface. It also manages all the services in the system by registering and tracking object references, which can be looked up by other services and used to initiate secure inter-process communication upon permission checks. SELinux supports Binder by adding Linux Security Module (LSM) hooks at various places in the kernel driver, which handle service requests from other system services. These hooks determine whether SELinux permissions authorise specific security-sensitive operations and control Binder interface access and IPC communication. Sample entries in this context file related to Wi-Fi services are presented below:

```
wifiscanner     u:object_r:wifiscanner_service:s0
wifi            u:object_r:wifi_service:s0
wificond        u:object_r:wificond_service:s0
wifiaware       u:object_r:wifiaware_service:s0
```

2.2.4.5 User-based Permissions

A crucial security feature of Android's security model is that no application has permissions by default to perform operations that affect other applications or system resources. This is because sandboxing isolates each application's data and code execution, preventing it from interfering with others. In this case, Android introduces a user-based permission system which allows applications to escape their sandbox and access particular system services and sensitive data. To do so, they must specifically define and/or request permissions to access parts of the system by declaring them in their AndroidManifest.xml file. At installation time, the OS extracts all permission from the AndroidManifest.xml and uses the entries to grant appropriate permissions to the corresponding application process. In the case of runtime permissions that request access to private data or an aspect of device control such as ACCESS_FINE_LOCATION, the user must consent to the request when prompted with a dialogue box.

Android has a set of built-in permissions that an application can use, which are documented on Android's permissions API reference page. However, both system and user-installed applications can define additional permissions, called custom. An example of permissions related to Wi-Fi is the ACCESS_WIFI_STATE, which provides access to the WifiManager, responsible for viewing a list of configured networks and providing results of AP scans.

2.2.5 Android Wi-Fi Network Stack

Like other operating systems, Android supports connections via Wi-Fi, enabling various types of attacks mentioned in Section 2.1.6, such as spoofing, sniffing, man-in-the-middle, eavesdropping, etc. This section highlights a few concepts related to Wi-Fi architecture to enable better understanding and debugging for any experiments as part of this research. The availability of Wi-Fi in Android phones depends on the connection with the wpa_supplicant daemon, which can have different states that can be queried by applications with relevant permissions in their AndroidManifest.xml, e.g., android.permission.ACCESS_WIFI_STATE, android.permission.INTERNET, android.permission.CHANGE_WIFI_STATE etc. If this is the case, the android.net.wifi.SupplicantState can return a status state, such as INACTIVE, when the supplicant is in an inactive state, COMPLETED, when the authentication phase is completed, or ASSOCIATED, when the association phase is completed.

s Following the Android Architecture in Section 2.2.3, we can use the same approach to divide the Wi-Fi Stack into layers to understand the sequence of callings between various parts of the Android system.

Figure 2.18 shows that the Application layer contains various classes available in Android SDK's package called android.net.wifi package, which helps the end-user view or control Wi-Fi operations via Android's Graphical User Interface (GUI). For example, the user can invoke an Activity by sending an Intent, which makes a request which is processed and returned to the user visually on the phone's screen. In this context, several commonly used classes, such as WifiSettings or WifiEnabler, allow the end-user to initialise an interface, which is updated according to their actions, e.g., manually turning on the Wi-Fi

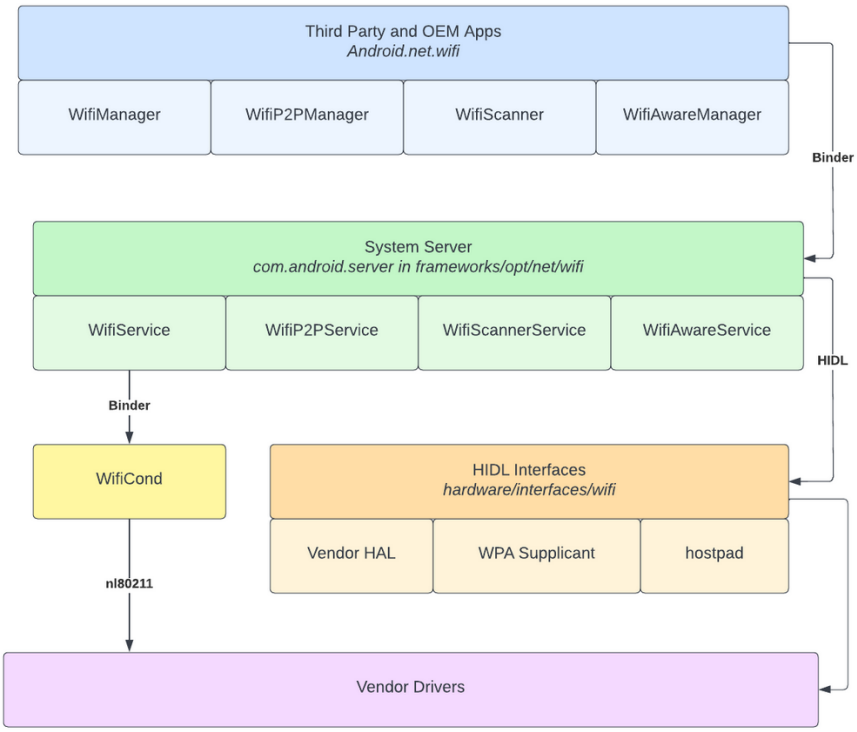


Figure 2.18: Android Wi-Fi Stack.

radio interface.

All classes used at the Application layer use a set of Java APIs, part of the lower layer called Application/Java Framework. This set agnostically exposes system components and services by introducing WifiManager.java, WifiService.java, WifiStateMachine.java, WifiMonitor.java, WifiNative etc. For example, WifiManager provides the functionality to applications to receive the results of AP scans, containing enough information about Wi-Fi networks available.Other functionalities include monitoring a current connection with a Wi-Fi network, which other Android applications with relevant permissions can query.

The request for initialising Wi-Fi is received and processed by the WifiStateMachine API, which uses one of its methods called processMessage to handle the request. This method uses WifiNative to call a native function that can load the Wi-Fi driver and start the Wi-Fi HAL. The WI-FI HAL function dealing with loading the driver is wifi.c. The Java Framework communicates with the wpa_supplicant using this function containing a WIFI_DRIVER_MODULE_PATH for the Wi-Fi driver required to communicate with the Wi-Fi chip. wpa_supplicant and netd are two demons responsible for initialising interfaces and network management, respectively. They usually use wireless extensions or nl80211 interface to control Wi-Fi drivers at the kernel layer.

As part of the Project Treble introduced with Android 8, Android re-architectured and simplified the Wi-Fi HAL to allow for better hardware abstraction and upgrades to its higher layers. Android did this by introducing three HIHL software packages that the Android OS can use to talk to Wi-Fi hardware available, which are:

- Vendor HAL - optional HIDL interface providing Android-specific commands.
- Client HAL - providing a HIDL interface to the wpa_supplicant daemon.
- Hostapd HAL - providing a HIDL interface for hostapd daemon.

Android uses the nl80211 interface between the user space (iw, wpa_supplicant, etc.) and the 802.11 drivers, which are provided as kernel modules (e.g., cfg80211 and mac80211) or specific drivers from the device vendor.

Overall, Android supports various Wi-Fi protocols, such as Wi-Fi Infrastructure, Wi-Fi Hotspot, Wi-Fi Awareness etc. However, in this research, we primarily focus on the Wi-Fi Infrastructure, which helps end-users connect to Wi-Fi networks.

Chapter 3

Research Methodology and Testing

3.1 Research Methodology

Our research goal is to test a range of Android devices and perform research against their handling of management probes, specifically how the SSID field can be manipulated and test how devices handle unexpected inputs in this field. The research methodology followed for this work is primarily derived from reverse engineering techniques, this largely consists of a combination of static and dynamic analysis.

Statically we disassembled firmware and performed analysis on the various components that make up the Android Wi-Fi stack, this involved disassembly and referencing various pieces of documentation. Much of a devices Wi-Fi stack is proprietary, in these instances research was conducted into the wider open source vulnerability research community to see if any existing work had been conducted that may help guide our efforts.

Dynamically we created our own test rogue access point that enabled the injection of test data into the SSID field in management probe, we then performed dynamic analysis on the devices through both visual inspection, to emulate what a user who is being subjected to an attack such as Evil Twin may see on their device, logging to see what the device itself records when processing a crafted SSID along with debugging applications involved in the Wi-Fi stack to gain a better understanding of their operation.

This approach can be summarised as trial and error focused, using a high volume of test data until something noteworthy happens, then backtracking using our research methodology to establish why the noteworthy event occurred and how to replicate it along with any wider implications.

3.2 Experimental Overview

In Section 1.3, we presented the scope and objectives of this project. It included research goals that can only be answered through experimentation and testing on 802.11-compliant devices. This chapter covers two experiments, one for the case where SSID can be used to spoof a target network, and the other involves exploring its potential use for further attacks. Further information is provided on the hardware and software used to run each experiment. The table below extends Table 3.1 in Section 3.3.1, specifying what mobile phones we use as part of each experiment.

3.3 Experiment A

3.3.1 Goals

Spoofing of APs is a common technique for adversaries to trick victim users into connecting to Rogue Access Points (RAP) that attempt to steal information such as login credentials. These attacks are commonly referred to as Evil Twin attacks, in reference to cloning the AP parameters such as the SSID. Using SSID spoofing, an adversary can trick and persuade an end-user using an Android-based device in the proximity of their AP to join a malicious network. In this scenario, the BSSID of the malicious

Brand	Motorola	OnePlus	Samsung
Model	Edge	Nord 2 5G	Galaxy S21 FE 5G (SM-G990B)
OS	Android 11	Android 11	Android 12
Chipset	Qualcomm Snapdragon 765G	MediaTek MT6893 Dimensity 1200 5G	Samsung Exynos 2100
Chipset Manufacturer	Qualcomm	Mediatek	Samsung +
Experiment	A, B	A, B	A

Figure 3.1: Basic information on the smartphones used in this project.

AP does not have to be the same as the legitimate one, as we focus on creating an AP which only looks identical to the end-user and not to the Android OS.

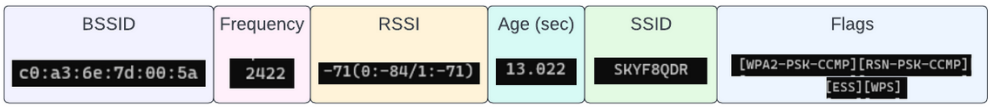


Figure 3.2: Example of what an Android system captures about an AP in its proximity.

As a common attack vector, Google and vendors that supply Android devices have implemented several protections to alert or otherwise prevent Evil Twin (ET) attacks from occurring. Our experiment aims to exploit SSID stripping to bypass these protections and enable attackers to perform ET attacks that look benign to the underlying operating system.

Within the scope of this experiment, we primarily look to identify the following classifications of vulnerabilities:

- Character Omission Vulnerability: This fundamentally results when particular characters are omitted from the display to a user. For instance, a new line character (
n, with the hex value 0x0a) injected at the end of an SSID may not be shown, resulting in visual sameness between two SSIDs despite one having an additional character.
- Display Overflow Vulnerability: This type of vulnerability focuses on exploiting how mobile phones show text to the end-user, considering that the physical size of their displays is limited. In this case, characters injected into an SSID push data outside a visible field of the end-user. This results in two SSIDs appearing similar despite one having additional information outside of the user's view space.
- Prefix Vulnerability: This occurs when certain characters prevent any subsequent ones from being displayed; for instance, some software and programming languages treat strings (in this case, SSIDs) as null-terminated, resulting in an SSID such as "Hello¡NULL¿World" being displayed as just "Hello".

Our outcome is to assess any discovered vulnerabilities and their applicability to existing ET or more generic RAP attacks, focusing on how exploitation of these vulnerabilities could lead to attackers bypassing security mechanisms and otherwise making their attacks more covert and more likely to succeed.

3.3.2 Scope

As part of this research, we are interested in how our mobile test devices handle intentionally crafted malicious SSIDs that contain characters placed with the intention of causing unexpected behaviour by

the Android operating system when displaying said SSIDs to users. To accomplish this, we craft SSIDs with special characters that can be broadcast as part of management frames like beacons. Broadcasting of beacon frames should happen on the same frequency band and channel as the target Wi-Fi network. This ensures that the malicious AP provides high output power/signal strength to encourage end-users to select it over the legitimate one. In this case, if an adversary physically moves their malicious AP adjacent or closer to mobile target devices, it will determine the position of the SSID in any available network list in its proximity.

As part of this experiment, we use the control characters of the ASCII character set, some Unicode control characters, as well as a list of naughty strings, also provided in the Github repository. This list contains a list of strings with a higher probability of causing issues when used as user-inputted data.

If any results indicate injection or formatting bugs in lower-level Android components, these will be investigated as part of Experiment B.

3.3.3 Setup and Tools

For both experiments, we use various equipment, such as a Raspberry Pi 4 (RPI4) Model B [1], to set up an AP, which requires the following accessories: a power supply, HDMI cable, a microSD card, keyboard and mouse. An RPI is a compact and powerful single-board computer, and its low cost, small size, portability and features make it ideal to act as an AP for our experiments. RPI4 is equipped with built-in dual-band wireless (Wi-Fi 5/IEEE 802.11ac), and it is powered by a Broadcom BCM2711 system-on-a-chip (SoC), which has all the necessary components to power the RPI. Considering the built-in Wi-Fi support monitor mode, we do not need an external USB Wi-Fi adapter in this project. However, we only consider the 2.4GHz band and 11 channels, as it travels further than a 5GHz signal, and a potential adversary would have more extensive coverage. Furthermore, the wireless chip is integrated into an RD shielded module alongside Bluetooth, as can be seen in Figure X below, alongside other main components:

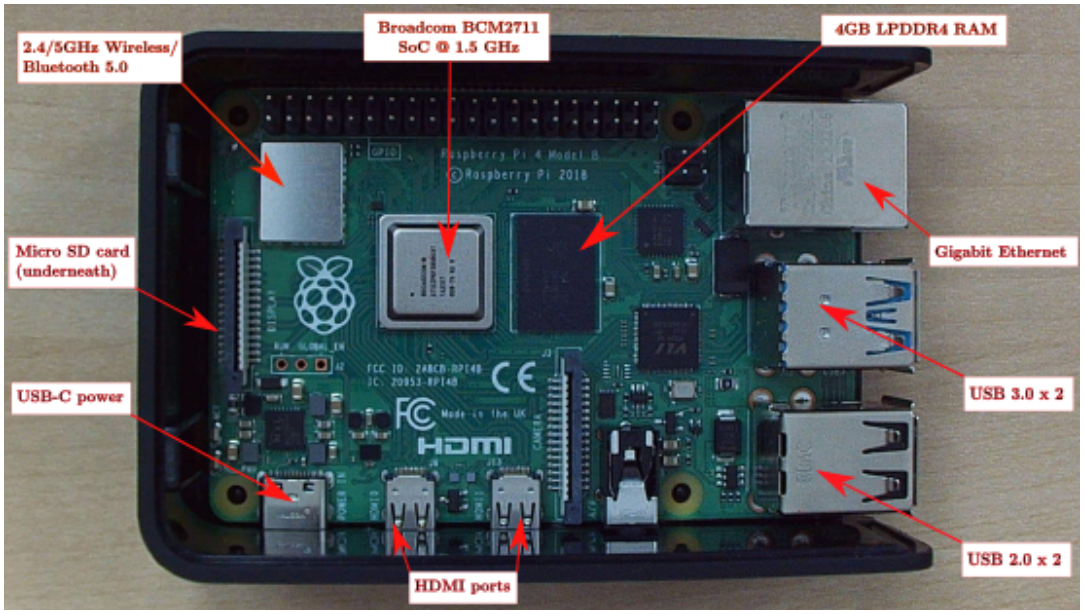


Figure 3.3: Raspberry Pi 4 model components.

RPIs can run an extensive range of operating systems; however, we leverage a Kali Linux version created specifically for devices like this one embedded with an ARM processor. Kali Linux is a Debian-based distribution, and its main advantage is that it comes pre-installed with security tools. This OS is installed on an SD card, which must be inserted into the RPI and hooked up to a monitor and keyboard. Once the RPI device is powered on, it will attempt to boot Kali Linux directly from the microSD card.

In addition to this, we identified the following software tools to create an AP using existing hardware:

- Hostapd (Host Access Point Daemon) [x] is a Linux software tool which is capable of turning a Wi-Fi interface into an AP.

- Dnsmasq [x] is a lightweight tool that helps create a DHCP and DNS server locally.

Other equipment includes three smartphones (Table 3.1) used to observe the behaviour of a specially crafted SSID, which is displayed in the available networks list of a wireless mobile device in proximity. In terms of software tools, we use Wireshark [X] to capture Wi-Fi management frames and analyse relevant fields, particularly the SSID. In addition to this, each experiment below involves different tools, which will be presented as part of their individual sections.

3.3.3.1 Rogue Access Point (RAP) Configuaration

To set up the RAP used for this experiment, we download and install a Kali Linux image on the microSD card inserted into the RPI4, following the [x] tutorial. Logging into the OS requires the default username "kali" and the password "kali", which should be changed once logged in. To change the password for both root and kali users, we type the following commands in the terminal.

```
$ sudo -s
# passwd root
# passwd kali
```

As a general convention for terminal commands, the hash sign before a command warns the user that they are in root mode, whilst the dollar ($) sign indicates they are running as a normal user. This configuration uses three different shell scripts, which are text files containing a series of commands related to installing, configuring and removing a RAP for the shell to execute. These are provided and can be accessed in the public Github repository (https://github.com/emimav/FinalYearProject-SSID) for future referencing.

The scripts files need to be executable to work, and we can set the executable flag and run the scripts through the following commands:

```
$ chmod +x install.sh
$ chmod +x configure-ap.sh
$ chmod +x remove.sh
```

The installation script install.sh needs to be executed as a root user and the RPI4 must have Internet connectivity. This is because it updates and upgrades Kali Linux packages and installs other required software, such as hostapd and dnsmasq. Other commands in the script refer to stopping these services while configured through config.sh.

```
$ sudo -s
# passwd root
# passwd kali
```

The configuration script configures a static IP for the Wi-Fi network interface wlan0 to act as an AP. wlan0 is the interface of the wireless card, and it uses the nl80211 driver to specify to hostapd how to communicate with the Wi-Fi kernel modules. wlan0 is also the monitor interface used by many tools, such as Wireshark, to capture all traffic intercepted by the NIC. This script configures a new hostapd file with various parameters, such as the interface and driver used, SSID, frequency operation mode, channel, password and other parameters, as seen below. In addition, DNSmasq is also configured to assign IP addresses to devices within a configured range but does not share Internet access as this is outside the scope of our project. Finally, as part of the DHCP configuration, nohook wpa_supplicant disables the wpa_supplicant for the wlan0 interface. In this case, wpa_supplicant [x] is an IEEE 802.11 WPA component found in client devices responsible for connecting the device to a Wi-Fi Internet. Once all wireless interface and environment settings, authentication and encryption configuration are all set, the script unmasks the hostapd, starts and enables it to auto-start at boot. When this completes, the RPI4 reboots and any device user in its proximity can use it to connect to it like any other Wi-Fi AP.

Optionally, the remove.sh script gives the end-user the option to remove the RAP, all the configuration files, and packages used to create it in the first place. This could be beneficial when testing various packages, settings or if something goes wrong at any point during the RAP configuration process.

3.3.4 Implementation

Considering that our research involves testing the effectiveness of many encoded characters as part of an SSID field, we automate changing the SSID by creating another executable shell script, config-ssid.sh, also available on Github. When this script executes, it changes the SSID in the hostapd.config file with a new one and restarts the hostapd after each trial.

The initial configuration should look similarly to Figure 3.4 below.

```
62  echo "country_code=GB
63
64  # Interface Settings
65  ####################
66  interface=wlan0
67  # Use the nl80211 driver with the brcmfmac driver
68  driver=nl80211
69
70  # Wifi AP Settings
71  ##################
72  # This is the name of the network
73  ssid=MyLittlePonyy
74  # Use the 2.4GHz band
75  hw_mode=g
76  # Use channel 7
77  channel=7
78  # The network passphrase
79  wpa_passphrase=Something
80
81  # Enable WMM (QoS)
82  wmm_enabled=0
83
84  # Accept all MAC addresses
85  macaddr_acl=0
86
87  # Encryption Settings
88  ####################
89  # Use WPA authentication
90  auth_algs=1
91  # Use WPA2
92  wpa=2
93  wpa_key_mgmt=WPA-PSK
94  # Use AES, instead of TKIP
95  wpa_pairwise=TKIP
96  rsn_pairwise=CCMP
97
98  # Require clients to know the network name
99  ignore_broadcast_ssid=0" >> /etc/hostapd/hostapd.conf
100 sleep 5s
```

Figure 3.4: Example of setting up hostapd.config for our research.

In the first part of our experiment, we take all ASCII control characters part of the C0 set [x], and test them on all three phones - Motorola, Samsung, and OnePlus. Generally, control characters are intended to structure data and control a particular action, such as line spacing or starting a new line, and they generally do not show up when displayed to the end-user. Although many of them are obsolete, not in use or blocked from being entered into user-inputted fields, there are still a few common characters found, and one of them is NULL. These control characters are not forbidden under the IEEE 802.11 specifications, but there are also no recommendations for device vendors from Android when it comes to processing or rendering them.

Experiment A first involves embedding these control characters into two SSIDs of different formats. The first SSID is "Hello(CHAR)World", whilst the second only uses three control characters inputted as "(CHAR)(CHAR)(CHAR)". Placing our target control characters in the middle of a valid SSID will enable us to test for any prefix or display overflow vulnerabilities. This could lead to further security issues when processed by the Android device or if it goes unnoticed by an end-user as part of a spoofing attack.

The structure of these SSIDs is illustrated in Figures 3.5 and Figure 3.6. The malformed strings in the config-ssid.sh are inputted into the SSID parameter directly as a hex-encoded string to match the way hostapd processes individual bytes at a lower level. For this experiment, we use the tcpdump [x] utility to capture the malformed beacon frames and observe the hex values of the ASCII characters. The aim is to follow how hostapd maps ASCII control characters in single-byte characters, as well as if any changes are made to the malformed string before sending it wirelessly as part of the beacon frame.

Whilst testing the ASCII set in this experiment, we report on the encoding bugs that have visual

Figure 3.5: Representation of the first SSID.

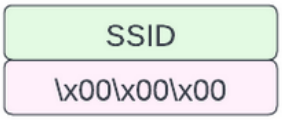


Figure 3.6: Representation of the second SSID.

and unexpected modifications in how they are rendered by individual Android phones in the ANL. In addition, we observe changes to the encoding of the SSID, which results in a very subtle modification to the SSID field when displayed to the end-user, which could also go unnoticed when looking to connect to a legitimate AP. Finally, once we inject the complete series of ASCII characters into the two forms of SSID and report our findings, we observe:

- The control characters that can be beneficial from an adversary's perspective to hide information.
- The effect of deletion control characters, such as backspaces, when they are injected into the SSID string.
- The control characters that could be used by an adversary to launch an ET attack.
- The control characters that are not rendered in the same way by all three devices, which could be used as a basis for investigation during Experiment B.

Aside from testing the two SSIDs with a series of ASCII control characters SSIDs above, we also inject various Unicode control characters encoded in a UTF-8 format to observe whether there is any unexpected or even expected behaviour which could be used negatively by an adversary. Other testing includes utilising a list of naughty strings linked on our Github repository to observe any anomalies in rendering them in ANL on Android devices

3.3.5 Outcomes

As part of our experiment we first injected ASCII control characters in two different SSIDs across three different Android-based devices. A complete list of this testing is provided in Appendix B. When exploring anomalies and bugs in malformed SSIDs caused by ASCII encoding, we found that the Samsung phone to be more prone to character omission vulnerabilities, with 30/32 possible control character injections causing it. Figure 3.7 illustrates this findings:

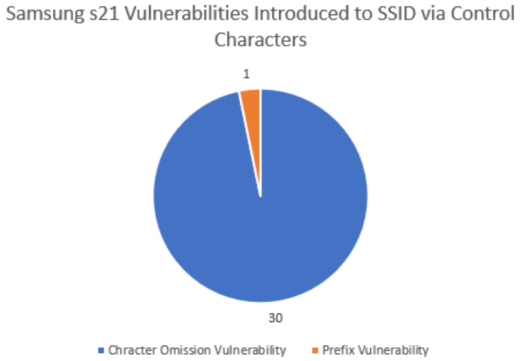


Figure 3.7: Samsung S21 Findings.

Both Motorola and OnePlus omitted 14 out of 32 possible control characters, which indicates a similar way of rendering or sanitization mechanism by their implementation of Android system. Both findings are illustrated below:

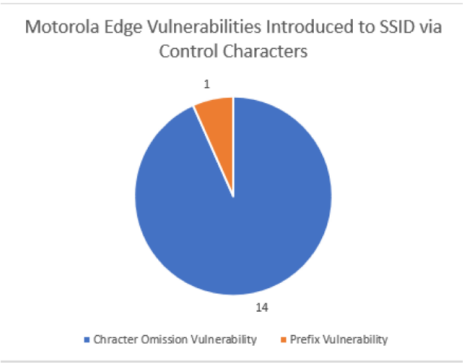


Figure 3.8: Motorola Edge Findings.

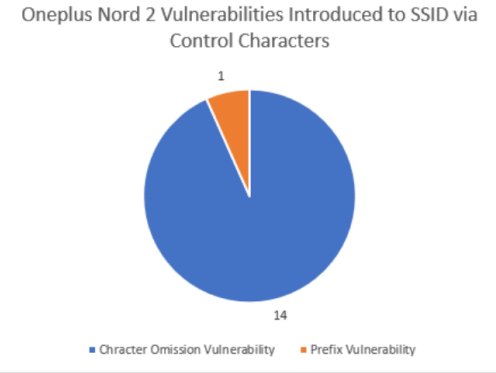


Figure 3.9: OnePlus Nord 2 Findings.

3.3.5.1 SSID Truncation and Spoofing

When passing a NULL character into the SSID for all devices, we can observe a prefix vulnerability. For example, the SSID = "Hello¡NULL¿World" produces the result from Figure 3.10:

Figure 3.10: SSID embedded with NULL control character.

This could potentially allow attackers to hide malicious code inside an SSID which may not be adequately sanitised for the NULL byte by certain lower-level components, for instance, consider the following:

Although two SSIDs may appear the same, they contain different bytes, which are not noticeable to the end-user. This enables social engineering attacks such as ET or RAP, which, combined with the prefix vulnerability, could inject malicious code into a vulnerable function. While the display components may strip anything after the NULL control character, the injectable string may hit internal elements that do not before any sanitization occurs. Considering that we could use malformed SSIDs to launch spoofing attacks, even though they visibly match a legitimate network name from an end-user perspective, the Android devices do not seem to have any measures to detect an attempt to perform similar attacks, as illustrated in Figure 3.12 and Figure 3.13.

Legitimate SSID	Malicious SSID
Guest_Network	Guest_Network \00 " or ""="
	

Figure 3.11: Example of malicious SSIDs to launch an ET attack.

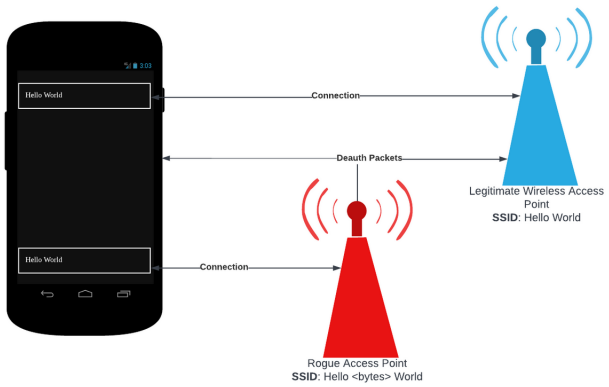


Figure 3.12: Example of an Evil Twin attack.

```
racer:/ # cmd wifi list-scan-results
    BSSID              Frequency     RSSI           Age(sec)     SSID                                 Flags
  e4:5f:01:4a:12:1a       2442    -36(0:-36/1:-46)     8.261    VM23-all                             [WPA2-PSK-CCMP][RSN-PSK-CCMP][ESS]
  60:6c:63:af:6f:a8       2437    -46(0:-54/1:-47)     8.383    VM23-all                             [WPA2-PSK-CCMP][RSN-PSK-CCMP][ESS]
  62:6c:63:af:6f:a0       5180    -50(0:-52/1:-55)     7.342    VM23-d0wnstairs                      [WPA2-PSK-CCMP][RSN-PSK-CCMP][ESS]
  50:70:43:55:a2:dd       5180    -64(0:-66/1:-67)     7.344    SKY4XMU1                             [WPA2-PSK-CCMP][RSN-PSK-CCMP][ESS][WPS]
  50:0f:f5:6d:f9:c9       2412    -67(0:-69/1:-71)     8.771    SKY4XMU1                             [WPA2-PSK-CCMP][RSN-PSK-CCMP][WPA-PSK-CCMP][ESS]
  50:70:43:55:a2:da       2412    -68(0:-71/1:-71)     8.821    SKY4XMU1                             [WPA2-PSK-CCMP][RSN-PSK-CCMP][ESS][WPS]
  38:a6:ce:23:c7:7e       2412    -74(0:-78/1:-75)     8.798    SKY4XMU1                             [WPA2-PSK-CCMP][RSN-PSK-CCMP][ESS][WPS]
  d8:47:32:d1:f3:f0       2412    -76(0:-79/1:-80)     8.763    TP-Link_F3F0                         [WPA-PSK-CCMP][WPA2-PSK-CCMP][RSN-PSK-CCMP][ESS]
  08:40:f3:6e:c5:48       2412    -77(0:-79/1:-81)     8.807    SKY4XMU1                             [WPA-PSK-CCMP][WPA2-PSK-CCMP][RSN-PSK-CCMP][ESS]
  38:a6:ce:23:c7:7b       5180    -83(0:-88/1:-85)     7.341    SKY4XMU1                             [WPA2-PSK-CCMP][RSN-PSK-CCMP][ESS][WPS]
  72:97:41:ac:da:93       2462    -90(0:-93/1:-96)     7.956    BTWi-fi                              [ESS]
  50:0f:f5:6d:c5:39       2412    -92(0:-94/1:-97)     8.802    SKYB3888                             [WPA2-PSK-CCMP][RSN-PSK-CCMP][WPA-PSK-CCMP][ESS]
  70:97:41:ac:da:92       2462    -92(0:-93/1:-99)     7.967    BT-R7CJ88                            [WPA2-PSK-CCMP][RSN-PSK-CCMP][ESS][WPS]
racer:/ #
```

Figure 3.13: Example of how the SSIDs are displayed using the wpa_cli utility which enables interaction with the wpa_supplicant.

3.3.5.2 SSID as an Injection Surface

The SSID itself proved to be a difficult attack surface; even with a vulnerability that allowed for injection, all implementations are expected to have a hard limit of 32 bytes, as per IEEE 802.11 specifications. Unfortunately (or fortunately), this is not enough space for SQL or other higher-level language injection attacks.

While we have witnessed questionable handling of SSIDs, especially regarding data sanitization, which is typically a sign that injection may be possible, more work would have to be conducted into conducting valid injection attacks within the size constraints of the IEEE 802.11 standard.

3.3.5.3 SSID and Unicode Encoding

When performing further testing with Unicode encoding and naughty strings, we made the following observations: Some of the ASCII characters are rendered so that, although they are different, they look the same to the end-user. For example, the double-quote 0x22 is identical to two of the single quotes (0x27 0x27). The same problem was encountered for two double quotes (0x22 0x22) and a combination

of (single quote, double quotes, single quote), which has the ASCII hex value of (0x27 0x22 0x27). An example is showcased in Figure 3.14

ID	115	116
Naughty String	" (Double quotes - 0x22)	'' (2 x Single quote – 0x27 0x27)
Motorola		
OnePlus		
Samsung		

Figure 3.14: ASCII rendering issues when processing and rendering similar characters

When processing SSIDs containing hidden Unicode characters with special properties, such as formatting (e.g., right-to-left override), it caused devices to render them very differently. All three phones seem to struggle to maintain character order with mixed character sets, which could introduce further vulnerabilities related to character ordering. Two examples of malformed SSIDs containing naughty strings containing Unicode characters are presented in Figure 3.15

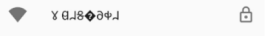
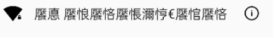
ID	134	174
Naughty String	ɣ ɐ˩ɹ8ɘ∂ɸ˩	test (0xe2 0x80 0xaa 0xe2 0x80 0xaa 0x74 0x65 0x73 0x74 0xe2 0x80 0xaa)
Motorola		
OnePlus		
Samsung		
Official List Position	200	305
Observations	String containing two-byte letters. We can notice that Samsung and Motorola support most or all UTF-16 encoded characters.	String containing hidden Unicode characters with special properties (format characters – e.g., right-to-left override).

Figure 3.15: Examples of SSIDs crafted with special Unicode characters.

3.4 Experiment B

In this experiment, we take our findings from Experiment A and perform reverse engineering on firmware extracted from the devices. Our primary focus is to understand the SSID "code flow" and look for any vulnerabilities we may be able to leverage into an attack. We are also generally interested in coding errors and researching/understanding an area of Android that is rarely reversed and is poorly understood outside of a dedicated group of developers.

3.4.1 Goals

For this experiment, we take observations made in Experiment A and approach them from a reverse engineering standpoint. We aim to extract firmware from devices and then reverse engineer the various Wi-Fi components to understand our findings better and potentially identify any vulnerabilities or coding errors that an attacker might exploit.

Obtaining firmware can be challenging on modern devices, and many vendors have begun restricting the free distribution of firmware. In addition to this, they no longer provide a simple download method. Therefore, static firmware analysis is vital for this experiment, which is why obtaining and analysing earlier versions of firmware on a computer is better than relying on our two test devices. In general, there are three ways of obtaining firmware for each device:

1. Perform a Man-in-the-Middle Attack to obtain a system update (OTA) for the device. However, considering that we already have rooted our devices and OTAs are delivered over HTTPS, this would hinder our chances of doing this successfully. Some vendors, however, do not ship complete firmware for updates. One example is Samsung, which provides "patches" rather than complete re-distributions.

2. Extract the firmware from the recovery partition as an artefact left over due to the Android A/B update scheme.

3. Find the firmware from a third party. Many devices have a large modding community, so this is often possible via Android communities like XDA.

We opted for the third option for our two devices (OnePlus and Motorola), primarily because the firmware was available in any other way, and it was simpler and more efficient than the other two. In this case, all metadata for firmware was verified, and certificates were checked to match the vendors to ensure the firmware was genuine and comparable to what would be running on one of the devices.

3.4.2 Scope

Android firmware and components related to Wi-Fi is a large area; we want to focus our research on vendor-specific additions and changes to the components as we believe this will be the most fruitful area of vulnerabilities and code errors. We will generally focus on vendor-specific services and HIDL interfaces along with vendor driver-versing engineering. We avoid the application level as part of this experiment, considering that typically, it will not deal with initial management probes or SSID other than potentially passing them into a database or to lower-level components.

3.4.3 Tools

To reverse engineer firmware, we made use of several existing tools:

1. Binwalk: A general-purpose tool for analyzing firmware images, primarily allowing us to inspect proprietary headers and extract interesting artefacts such as filesystems or other files.

2. Imjtool: A powerful utility for extracting android firmware that can handle most common formats.

3. Ghidra: A suite of reverse engineering tools that allow for static analysis of binaries and other data.

4. simg2img: As previously mentioned, this is a valuable utility for reconstructing sparse android image files.

3.4.4 Implementation

This section outlines our research into reversing vendor firmware to enable access to the underlying Wi-Fi and radio firmware allowing us to understand better issues and possible vulnerabilities discovered in experiment A. Although we chose to focus on the Motorola firmware for documentation as it proved the most challenging to extract, the processes used here can also be applied to the other two devices.

3.4.4.1 Initial Analysis

Motorola firmware is distributed using a standard compression format and is easy to unpack; it contains various tools the Motorola ship to assist with flashing and other diagnostic processes. Looking into the firmware, we can see the following layout.

```
total 4.5G
-rwxrwx—— 1 root vboxsf 1.8M May 22 14:58 adb.exe
-rwxrwx—— 1 root vboxsf  96K May 22 14:58 AdbWinApi.dll
-rwxrwx—— 1 root vboxsf  62K May 22 14:58 AdbWinUsbApi.dll
-rwxrwx—— 1 root vboxsf  96M May 22 14:58 boot.img
-rwxrwx—— 1 root vboxsf  13M May 22 14:58 bootloader.img
-rwxrwx—— 1 root vboxsf 185K May 22 14:58 BTFM.bin
-rwxrwx—— 1 root vboxsf  30K May 22 14:58 cmd-here.exe
-rwxrwx—— 1 root vboxsf  64M May 22 14:58 dspso.bin
-rwxrwx—— 1 root vboxsf 2.0M May 22 14:58 dtbo.img
-rwxrwx—— 1 root vboxsf 834K May 22 14:58 fastboot.exe
-rwxrwx—— 1 root vboxsf 1.2K May 22 14:58 flashfile.bat
-rwxrwx—— 1 root vboxsf 3.0K May 22 14:58 flashfile.xml
-rwxrwx—— 1 root vboxsf 207K May 22 14:58 gpt.bin
-rwxrwx—— 1 root vboxsf  12M May 22 14:58 logo.bin
-rwxrwx—— 1 root vboxsf  125 May 22 14:58 motostockrom.com.url
-rwxrwx—— 1 root vboxsf 1.1K May 22 14:58 RACER_RETAIL_QPD30.70-16-1_subsidy-DEFAULT_regulatory-DEFAULT_CFC.info.txt
-rwxrwx—— 1 root vboxsf 303M May 22 14:58 radio.img
-rwxrwx—— 1 root vboxsf 100M May 22 14:58 recovery.img
-rwxrwx—— 1 root vboxsf    0 May 22 14:58 regulatory_info_default.png
-rwxrwx—— 1 root vboxsf 2.7K May 22 14:58 servicefile.xml
-rwxrwx—— 1 root vboxsf    0 May 22 14:58 slcf_rev_d_default_v1.0.nvm
-rwxrwx—— 1 root vboxsf 512M May 22 14:58 super.img_sparsechunk.0
-rwxrwx—— 1 root vboxsf 490M May 22 14:58 super.img_sparsechunk.1
-rwxrwx—— 1 root vboxsf 512M May 22 14:58 super.img_sparsechunk.2
-rwxrwx—— 1 root vboxsf 463M May 22 14:57 super.img_sparsechunk.3
-rwxrwx—— 1 root vboxsf 511M May 22 14:58 super.img_sparsechunk.4
-rwxrwx—— 1 root vboxsf 511M May 22 14:58 super.img_sparsechunk.5
-rwxrwx—— 1 root vboxsf 512M May 22 14:58 super.img_sparsechunk.6
-rwxrwx—— 1 root vboxsf 413M May 22 14:58 super.img_sparsechunk.7
-rwxrwx—— 1 root vboxsf 4.0K May 22 14:58 vbmeta.img
```

For our research, we are primarily interested in the functionality contained within the radio and super partition. A helpful trick for understanding how raw image files are mapped to partitions is to examine the flashing configuration file that will be present in all Android firmware. This file describes the vendor's internal tooling as well as the recovery functionality of the device and how to map the raw data to the appropriate partition.

```
<steps interface="AP">
  <step operation="getvar" var="max-sparse-size"/>
  <step operation="oem" var="fb_mode_set"/>
  <step MD5="8f17b022e95c7baaa91042d1ac7589e2" filename="gpt.bin" operation="flash" partition="partition"/>
  <step MD5="9b72197c8eedb77e3bf0792a10076de5" filename="bootloader.img" operation="flash" partition="bootloader"/>
  <step MD5="15bd1a50a514e356463d8d75a28bf7ff" filename="vbmeta.img" operation="flash" partition="vbmeta"/>
  <step MD5="021dd96c485bfbd2af1bd4adbbe14482" filename="radio.img" operation="flash" partition="radio"/>
  <step MD5="b2eee7f0cd833ff9523570ef56207b18" filename="BTFM.bin" operation="flash" partition="bluetooth"/>
  <step MD5="0b205208076e640e1593f8b41ee5902f" filename="dspso.bin" operation="flash" partition="dsp"/>
  <step MD5="3fd7f6ad422b15507de68a16cb91c2a0" filename="logo.bin" operation="flash" partition="logo"/>
  <step MD5="3875fcae2783412ca6f3305d051f7c98" filename="boot.img" operation="flash" partition="boot"/>
  <step MD5="060ca2ecff932c576809e051843f1bcd" filename="dtbo.img" operation="flash" partition="dtbo"/>
  <step MD5="e1e47551320c5bdea3c9e3eee1d6cd7e" filename="recovery.img" operation="flash" partition="recovery"/>
  <step MD5="a458e7bbc3b3ff228491378ead520de0" filename="super.img_sparsechunk.0" operation="flash" partition="super"/>
  <step MD5="3558475ed0c855fe192a501361fc539f" filename="super.img_sparsechunk.1" operation="flash" partition="super"/>
  <step MD5="8de474d8caf4bfddca9527a5081b6104" filename="super.img_sparsechunk.2" operation="flash" partition="super"/>
  <step MD5="499448cddeeafab8f8820f343b4e67ce" filename="super.img_sparsechunk.3" operation="flash" partition="super"/>
  <step MD5="ee37e2672647eb8ecf3cd6916d9471de" filename="super.img_sparsechunk.4" operation="flash" partition="super"/>
  <step MD5="5523b9f343409ece70e80f5eb0d6add1" filename="super.img_sparsechunk.5" operation="flash" partition="super"/>
  <step MD5="e554c67483310bc28205c960b0faf1b1" filename="super.img_sparsechunk.6" operation="flash" partition="super"/>
  <step MD5="ebd26e8078ba5aee4a3bc67162d070b3" filename="super.img_sparsechunk.7" operation="flash" partition="super"/>
  <step operation="erase" partition="carrier"/>
  <step operation="erase" partition="ddr"/>
  <step operation="oem" var="fb_mode_clear"/>
</steps>
```

We can see the process for the radio partition seems straightforward and examining the radio.img file, we can see that its header doesn't match any obvious filesystem.

```
└─$ hexdump -n 50 radio.img
0000000 4953 474e 454c 4e5f 4c5f 4e4f 4c45 0059
0000010 0000 0000 0000 0000 0000 0000 0000 0000
*
0000030 0000
0000032
```

This is not uncommon considering that firmware like this is often dynamically packed and can contain

various files and filesystems. Using a data analyzer like binwalk, we can better understand what is included within the radio.img file.

```
└─$ binwalk -t -P radio.img

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
512           0x200           XML document, version: "1.0"
4864          0x1300          XML document, version: "1.0"
9216          0x2400          XML document, version: "1.0"
13608         0x3528          Linux EXT filesystem, blocks count: 77568, image size: 79429632, rev 1.0, ext2 filesystem data, UUID
83217370      0x4F5CBDA       Intel x86 or x64 microcode, sig 0x151c1e1c, pf_mask 0x1a1a211a, 190F-12-18, rev 0x-1000000, size 465
83501734      0x4FA22A6       VxWorks symbol table, big endian, first entry: [type: uninitialized data, code address: 0x600, symbo
83542344      0x4FAC148       PEM certificate
83545216      0x4FACC80       Base64 standard index table
85878209      0x51E65C1       LZMA compressed data, properties: 0x90, dictionary size: 0 bytes, uncompressed size: 256 bytes
96536101      0x5C10625       Intel x86 or x64 microcode, sig 0x3b861f92, pf_mask 0x901fffd2, 2000-13-28, size 206703236
115300603     0x6DF58FB       Certificate in DER format (x509 v3), header length: 4, sequence length: 20787
117945664     0x7078540       Zlib compressed data, default compression
119125528     0x7198618       Unix path: /usr/lib/libc.so.1
119207664     0x71AF6F0       Copyright string: "Copyright (c) 1992-2006 by P.J. Plauger, licensed by Dinkumware, Ltd. ALL RIGHTS
119215424     0x71B1540       ELF, 32-bit LSB executable, version 1 (SYSV)
119218620     0x71B21BC       Certificate in DER format (x509 v3), header length: 4, sequence length: 1233
119219857     0x71B2691       Certificate in DER format (x509 v3), header length: 4, sequence length: 979
119220840     0x71B2A68       Certificate in DER format (x509 v3), header length: 4, sequence length: 961
```

We can see a range of data, including at least one ext filesystem. Extracting dynamically packed firmware can be quite a time-consuming task. We assume that this is an Android sparse image, which is a well-defined standard and, as such, can be reconstructed quite easily. However, the initial hex dump of the file did not provide the correct magic header. So, exploring to see if the Android Sparse Image magic header was present, we can across:

```
└─$ LANG=C grep -obUaP "\x3a\xff\x26\xed" radio.img
13568:
278738432:
```

Examining further, we deduced that while this is an Android Sparse Image, it has had some modifications applied, most notably some custom header and footer presumably by Motorola. The exact purpose of this modification is unclear. We know that binwalk assumed there was an ext file system of 13608 bytes from the start and our sparse image header is just before that at 13568. While binwalk registers this as an ext2, this is unlikely to be the case as this is a modern device. It is more likely an ext4 filesystem, and it is known that binwalk often confuses the two. Therefore, we proceeded under the assumption that we were dealing with an EXT4.

3.4.4.2 Unpacking the radio.img data

At this point in our research, we know the following:

- The raw image file is likely an Android sparse image, a well-defined specification that on its own is unpackable using a range of existing tooling and techniques.
- There appears to be vendor modification to the image itself, at least through adding a header roughly 130Kb in size and possibly elsewhere in the image file.
- Binwalk returned some indicators for known data types contained within the raw image, most notably an ext based filesystem.

To establish the validity of the suspected EXT4 filesystem, we can look for indicators as the standard is well established. From this, we can establish that the ext4 disk layout starts with 1024 bytes of padding followed by a superblock.

The s_magic field provides us with a reliable way of identifying that what we have is, in fact, an EXT4 filesystem. Binwalk estimates the filesystem starts at 13608 bytes offset, +1024 bytes as per the specification means that the superblock should be visible at the 14632 offset.

From here, we know that at a further 0x38 offset, we should have a 16-byte magic header of 0xEF53 in a little-endian format. 0x38 = 56, so our offset becomes 14688 byte.

Where we can see the magic header confirming that this is an EXT4 superblock, manual analysis of the rest of the block confirmed this. Given what we know, we manually removed the suspected vendor header and used, a convenient utility for unpacking sparse images on android. The results were very promising.

We also noticed some Easter eggs left in by the firmware creators:

This is the exact format we would expect to see radio firmware in. The XML files again provide information about how the firmware is built.

Offset	Size	Name	Description
0x0	__le32	s_inodes_count	Total inode count.
0x4	__le32	s_blocks_count_lo	Total block count.
0x8	__le32	s_r_blocks_count_lo	This number of blocks can only be allocated by the super-user.
0xC	__le32	s_free_blocks_count_lo	Free block count.
0x10	__le32	s_free_inodes_count	Free inode count.
0x14	__le32	s_first_data_block	First data block. This must be at least 1 for 1k-block filesystems and is typically 0 for all other block sizes.
0x18	__le32	s_log_block_size	Block size is 2 ^ (10 + s_log_block_size).
0x1C	__le32	s_log_cluster_size	Cluster size is (2 ^ s_log_cluster_size) blocks if bigalloc is enabled. Otherwise s_log_cluster_size must equal s_log_block_size.
0x20	__le32	s_blocks_per_group	Blocks per group.
0x24	__le32	s_clusters_per_group	Clusters per group, if bigalloc is enabled. Otherwise s_clusters_per_group must equal s_blocks_per_group.
0x28	__le32	s_inodes_per_group	Inodes per group.
0x2C	__le32	s_mtime	Mount time, in seconds since the epoch.
0x30	__le32	s_wtime	Write time, in seconds since the epoch.
0x34	__le16	s_mnt_count	Number of mounts since the last fsck.
0x36	__le16	s_max_mnt_count	Number of mounts beyond which a fsck is needed.
0x38	__le16	s_magic	Magic signature, 0xEF53
0x3A	__le16	s_state	File system state. Valid values are: 0x0001 Cleanly Unmounted 0x0002 Errors Detected 0x0004 Orphans Being Recovered

Figure 3.16: Table outlining the beginning structure of the superblock, __le means little endian in reference to size.

Both the fsg.mbn and NON-HLOS.bin are Android sparse images, but they are default constructions with no additional vendor modification, making further work much more manageable. In this case, the radio firmware is now in a state where we can begin work on reverse engineering its functionality.

```
└─$ hexdump -s 14632 -n 100 -C radio.img
00003928  00 2f 01 00 00 2f 01 00  07 03 00 00 a6 1b 00 00  |./...../........|
00003938  f0 2e 01 00 00 00 00 00  02 00 00 00 02 00 00 00  |................|
00003948  00 80 00 00 00 80 00 00  00 65 00 00 00 00 00 00  |.........e......|
00003958  e3 38 97 5e 00 00 ff ff  53 ef 01 00 01 00 00 00  |.8.^....S.......|
00003968  e2 38 97 5e 00 00 00 00  00 00 00 00 01 00 00 00  |.8.^............|
00003978  00 00 00 00 0b 00 00 00  80 00 00 00 38 00 00 00  |............8...|
00003988  42 00 00 00                                       |B...|
0000398c
```

```
└─$ hexdump -s 14688 -n 100 -C radio.img
00003960  53 ef 01 00 01 00 00 00  e2 38 97 5e 00 00 00 00  |S........8.^....|
00003970  00 00 00 00 01 00 00 00  00 00 00 00 0b 00 00 00  |................|
00003980  80 00 00 00 38 00 00 00  42 00 00 00 7b 00 00 00  |....8...B...{...|
00003990  7d 7c f3 e7 35 c6 40 6e  93 4d 09 9c 42 3b e2 6d  |}|..5.@n.M..B;.m|
000039a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000039b0  2f 6d 6f 64 65 6d 00 00  00 00 00 00 00 00 00 00  |/modem..........|
000039c0  00 00 00 00                                       |....|
000039c4
```

```
total 303M
-rwxrwx--- 1 root vboxsf  38M May 22 15:28 fsg.mbn
-rwxrwx--- 1 root vboxsf  140 May 22 15:28 index.xml
-rwxrwx--- 1 root vboxsf   53 May 22 15:28 LONELY_N_SINGLE
-rwxrwx--- 1 root vboxsf 266M May 22 15:28 NON-HLOS.bin
-rwxrwx--- 1 root vboxsf  214 May 22 15:28 radio.default.xml
-rwxrwx--- 1 root vboxsf   86 May 22 15:28 radio.pkg.xml
```

```
└─$ cat LONELY_N_SINGLE
All the lonely people, where do they all come from?
```

```
└─$ cat radio.default.xml
<?xml version="1.0" ?>
<recipe>
        <flash partition="modem"      filename="NON-HLOS.bin"/>
        <flash partition="fsg"        filename="fsg.mbn"/>
        <erase partition="modemst1"/>
        <erase partition="modemst2"/>
</recipe>
```

3.4.4.3 Unpacking the radio.img data

The super partition is split into seven different pieces. These are all Android sparse image formats. To effectively unpack the super.img image containing firmware such as system/ and vendor/, we need to do a few extra steps compared to the radio image we unpacked.

```
super.img_sparsechunk.0
super.img_sparsechunk.1
super.img_sparsechunk.2
super.img_sparsechunk.3
super.img_sparsechunk.4
super.img_sparsechunk.5
super.img_sparsechunk.6
super.img_sparsechunk.7
```

First, we can use the UNIX command-line utility simg2img to patch them together. This is specifically designed to deal with sparse image files. By doing so, we will get a singular super.img. When we begin

```
└─$ simg2img super.img_sparsechunk.0 super.img_sparsechunk.1 super.img_sparse
chunk.2 super.img_sparsechunk.3 super.img_sparsechunk.4 super.img_sparsechunk
.5 super.img_sparsechunk.6 super.img
```

to analyse the super.img, we can see features similar to those encountered during the radio firmware reversing. We can assume that the file format will also not be recognised, and some unknown data will be in the header section of the image. After preprocessing the image to remove the header, we can again use imjtool to remove the filesystems it contains.

3.4.5 Outcomes

Analysis of the underlying Android firmware was a beneficial learning activity enabling a better understanding of the overall Wi-Fi stack. A few interesting handling observations were also made as a result of this, along with a general increase in knowledge of the results from experiment A.

```
┌──(root㉿kali)-[/media/…/moto/Motorola_Edge_XT2063-3_
└─# file super.img
super.img: data

┌──(root㉿kali)-[/media/…/moto/Motorola_Edge_XT2063-3_
└─# hexdump -n 10000 super.img
0000000 0000 0000 0000 0000 0000 0000 0000 0000
*
0001000 4467 616c 0034 0000 ff12 f055 a7ab 06b5
0001010 5cf2 dab5 ca5d 3409 3442 dfe8 9c1d ae93
0001020 a482 d999 1980 7e46 0000 0001 0003 0000
0001030 1000 0000 0000 0000 0000 0000 0000 0000
0001040 0000 0000 0000 0000 0000 0000 0000 0000
*
0002000 4467 616c 0034 0000 ff12 f055 a7ab 06b5
0002010 5cf2 dab5 ca5d 3409 3442 dfe8 9c1d ae93
0002020 a482 d999 1980 7e46 0000 0001 0003 0000
0002030 1000 0000 0000 0000 0000 0000 0000 0000
0002040 0000 0000 0000 0000 0000 0000 0000 0000
*
0002710
```

```
liblp dynamic partition (super.img) - Blocksize 0×1000, 3 slots
LP MD Header @0×3000, version 10.0, with 6 logical partitions on block device of 10239 GB, at partition super, first sector: 0×800
Partitions @0×3080 in 3 groups:
        Group 0: default
        Group 1: mot_dp_group_a
                Name: system_a (read-only, Linux Ext2/3/4/? Filesystem Image, @0×100000 spanning 1 extents of 883 MB) - extracted
                Name: vendor_a (read-only, Linux Ext2/3/4/? Filesystem Image, @0×39e00000 spanning 1 extents of 674 MB) - extracted
                Name: product_a (read-only, Linux Ext2/3/4/? Filesystem Image, @0×64100000 spanning 1 extents of 2 GB) - extracted
        Group 2: mot_dp_group_b
                Name: system_b (read-only, Linux Ext2/3/4/? Filesystem Image, @0×37500000 spanning 1 extents of 40 MB) - extracted
                Name: vendor_b (read-only,  empty) - extracted
                Name: product_b (read-only,  empty) - extracted
```

3.4.5.1 Ambiguous Specification Leads to Potential Overflow

We began looking at reversing the wifi-service.jar on the Motorola device near the top of the Wi-Fi stack. We observed interesting behaviour in this device during experiment A where it correctly rendered non-UTF8 standard characters, whereas the other test devices failed. This implies that alterations may have been made to the processing of the SSID. We can see some Wi-Fi services of interest for our research inside the firmware. For example, at offset 0x500c97fc, we can find the function responsible for

```
./apex/com.android.wifi
./apex/com.android.wifi/javalib/service-wifi.jar
./apex/com.android.wifi/javalib/framework-wifi.jar
./apex/com.android.wifi@301700100
./apex/com.android.wifi@301700100/javalib/service-wifi.jar
./apex/com.android.wifi@301700100/javalib/framework-wifi.jar
./apex/com.android.vndk.v30/lib64/android.hardware.wifi.hostapd@1.0.so
./apex/com.android.vndk.v30/lib64/android.hardware.wifi.hostapd@1.1.so
./apex/com.android.vndk.v30/lib64/android.hardware.wifi.hostapd@1.2.so
./apex/com.android.vndk.v30/lib64/android.hardware.wifi.offload@1.0.so
./apex/com.android.vndk.v30/lib64/android.hardware.wifi.supplicant@1.0.so
```

validating an SSID. The function initially does some basic checks, for instance, if SSID is null or is an empty string. Then, one of two code paths is diverted depending on if the SSID is in byte string or char encoded format. Here we can observe several potential issues. The above shows the handling of the SSID, specifically the minimum and maximum size of the SSID when passed as a byte string. In this case, 0x40 corresponds to 64 bytes. The maximum length of an SSID is generally considered to be 32 bytes, as the device encodes data in UTF-8, where 2 bytes can equal one character; as being correct. The issue lies in ambiguity surrounding the 802.11 specifications and whether the 32 size limit applies to bytes, fixed size, or characters, which vary depending on the encoding scheme.

This type of code can introduce several issues. For example, suppose a lower-level component such as a driver interprets the 802.11 standards as having a maximum size of 32 bytes. In that case, this could potentially lead to an overflow if the input is not properly sanitised and proper bounds checking is not in place. It could also lead to many other undefined behaviours as the SSID may be truncated. Another similar issue exists in the encoded character code path for this function. In this variant, the maximum size of the SSID is 33, not 32 bytes. We assume this accounts for an additional null byte appended to the SSID when passed down to drivers (which will be in C that has null-terminated strings, unlike Java). However, there are no checks that this final character is a null byte and no obvious logic for appending the last byte to be NULL. This would lead to a breach in the 802.11 specifications and could allow an SSID of 33 bytes to be processed. This again could cause undefined behaviour in lower-level components

```
boolean validateSsid(String p0,boolean p1)

{
  boolean bVar1;
  undefined uVar2;
  byte[] pbVar3;
  int iVar4;
  String pSVar5;
  StringBuilder pSVar6;

  pSVar5 = "WifiConfigurationUtil";
  if (p1 == false) {
    if (p0 == null) {
      return true;
    }
  }
  else if (p0 == null) {
    Log.e(pSVar5,"validateSsid : null string");
    return false;
  }
  bVar1 = p0.isEmpty();
  if (bVar1 != false) {
    Log.e(pSVar5,"validateSsid failed: empty string");
    return false;
  }
```

```
  bVar1 = p0.startsWith("\"");
  if (bVar1 == false) {
    iVar4 = p0.length();
    if (iVar4 < 2) {
      pSVar6 = new(StringBuilder);
      uVar2 = pSVar6.<init>();
      return (boolean)uVar2;
    }
    iVar4 = p0.length();
    if (0x40 < iVar4) {
      pSVar6 = new(StringBuilder);
      uVar2 = pSVar6.<init>();
      return (boolean)uVar2;
    }
  }
```

```
else {
  pbVar3 = p0.getBytes(StandardCharsets.UTF_8);
  if (pbVar3.length < 3) {
    pSVar6 = new(StringBuilder);
    uVar2 = pSVar6.<init>();
    return (boolean)uVar2;
  }
  if (0x22 < pbVar3.length) {
    pSVar6 = new(StringBuilder);
    uVar2 = pSVar6.<init>();
    return (boolean)uVar2;
  }
}
```

expecting a buffer with a fixed size of 32 bytes.

Chapter 4

Discussion and Conclusion

4.1 Discussion

In experiment A, we conducted a manual test of how three Android devices handle a variation of SSIDs, crafted to mimic malicious strings that may be encountered in the wild. We witnessed a good deal of variation between how the three devices handled certain permutations. We showed that each device was vulnerable to at least one of the potential vulnerabilities:

- Prefix Vulnerability: For example, when a NULL byte was placed as part of a network, any data after the NULL byte was removed by the device. This could allow an attacker to craft a malicious SSID that, while appearing legitimate to the user, contains some malicious or injectable data after the NULL byte that other components on the device may process.
- Character Omission Vulnerability: This was widespread, particularly on the Samsung device, where certain characters did not display to the user. During an Evil Twin style attack, this would allow an adversary to craft an SSID that is technically different but appears identical to a legitimate access point. This would enable them to potentially bypass security mechanisms designed to flag if the properties of a known access point have changed.
- Display Overflow Vulnerability: The three devices were also vulnerable to attacks where format characters such as newline or tab were used to push a network name suffix outside the user's view. This has similar implications to the character omission vulnerability, enabling attackers to craft visually identical SSIDs to trick users in Evil Twin style attack scenarios.

Our findings show a reasonably severe issue with how Wi-Fi communications are implemented within the Android system. An attacker could leverage the conclusions we have made to enhance a range of attacks, such as Evil Twin social engineering, to create a malicious AP to appear more legitimate. Upon further investigation within experiment B, we uncovered implementation vulnerabilities stemming from what we suspect as ambiguity inside Wi-Fi standards that could lead to potential overflows or other unexpected behaviour. For example, in this case, an attacker could attempt to inject certain strings into the SSID or other areas of the management probe. During experiment B, we also appreciated just how complex and poorly understood the implementation of Wi-Fi standards is on Android devices. While there seem to have been improvements to secure communication in recent years, very little attention is paid to the initial pre-association stages required to set up a Wi-Fi connection. We believe this to be a potential hotbed of issues and vulnerabilities that are likely to surface in the coming years as more researchers turn their focus toward it.

4.2 Learning Reflection

Along with performing research into Android vulnerabilities relating to SSID stripping and other SSID focused vulnerabilities, we hoped to gain a large amount of practical reverse engineering and code review experience. On reflection, several primary skills were acquired during this research, ranging from learning to properly access and understand more in-depth technical information to gaining skills in reverse engineering complex components, which are not adequately documented. As discussed in experiment B, the internals of Wi-Fi communications built into Android devices is highly complex and broadly not

understood. In this case, while the AOSP components are well documented, the vendor-specific additions are often proprietary and require considerable effort to understand.

In experiment B, we demonstrated the process of reversing proprietary firmware that contained undocumented structures that we needed to remove or otherwise process to re-construct valid file systems required for our reverse engineering efforts. This gave us an appreciation of how much work and the complexities required in doing so, and we gained an understanding of firmware reverse engineering in general and how to use tools to aid in our efforts. On reflection, a considerable amount of knowledge was achieved related to Android-specific firmware reverse engineering along with more general EL64 and APK reverse engineering techniques.

4.3 Future Work

As stated, we believe that the un-secured aspect of Wi-Fi communication is poorly understood and is likely to be of significant interest to security researchers. Further exploitation into these areas on Android would likely yield some valuable outputs. During experiment B, a large portion of our time was spent learning primary skills, but with more time building on these skills, more of the firmware could be reversed to look for vulnerabilities.

As mentioned in Experiment B, we believe that general ambiguity surrounding 802.11 standards regarding the length of the SSID led to an implementation of a validation function that could lead to unexpected results later down the Wi-Fi stack. This is likely not just going to apply to SSID, and a more broad range of research against the un-secured Wi-Fi communications stages could be conducted to look for other injections or possible sources of vulnerability.

During experiment A the vulnerabilities that an attacker could use were highlighted, thought and developed could be done into the creation of a system designed to detect the type of SSID manipulation we demonstrated. This could make it harder for an attacker to use these vulnerabilities in a real-world scenario, like an Evil Twin social engineering attack.

4.4 Conclusion

Considering the latest IEEE 802.11 specifications related to management frames and Wi-Fi scanning and how these are implemented in Android's Wi-Fi stack, we have shown that the SSID field could be used to embed malicious strings, which could potentially be attack vectors for malicious actors. We further demonstrated that by performing reverse engineering on elements of a device's Wi-Fi stack, we could identify areas of code that poorly sanitise the SSID, potentially leading to an attacker being able to cause undefined behaviour.

Bibliography

[1] Amichai Shulman. *The SSID Stripping Vulnerability: When You Don't See What You Get*. Sept. 2021. URL: https://aireye.tech/2021/09/13/the-ssid-stripping-vulnerability-when-you-dont-see-what-you-get/.

[2] AMICHAI SHULMAN. *Wi-Fi Spoofing: Employing RLO to SSID Stripping*. Sept. 2021. URL: https://aireye.tech/2022/04/04/wi-fi-spoofing-employing-rlo-to-ssid-stripping/.

[3] IEE802. *IEEE 802 Home Page*. URL: https://www.ieee802.org.

[4] IEEE 802. *IEEE 802 Main Site*. URL: https://standards.ieee.org/featured/ieee-802.

[5] IEEE Standards Association (IEEE SA). *What are Standards? Why are They Important?* 2011. URL: Https://beyondstandards.ieee.org/what-are-standards-why-are-they-important.

[6] iso.org. *Open systems interconnection (OSI)*. URL: https://www.iso.org/ics/35.100/x/.

[7] IEEE Standard for Local, Metropolitan Area Networks: Overview, and Architecture. *802.1 WG - Higher Layer LAN Protocols Working Group*. 2014. URL: https://standards.ieee.org/ieee/802/4158/.

[8] IEEE 802 Working groups and Executive Committee Study Groups. *IEEE 802 LAN/MAN Standards Committee*. 2019. URL: https://www.ieee802.org/NADots.shtml.

[9] Black Box Corporation. *Wi-Fi Standards Explained*. 2018. URL: https://www.bboxservices.com/resources/blog/bbns/2018/04/30/802.11-wireless-standards-explained.

[10] LAN/MAN Standards Committee / IEEE Computer Society. *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*. 2021. URL: https://ieeexplore.ieee.org/document/9363693/.

[11] Wikipedia Contributors. *Wi-Fi*. 2019. URL: https://en.wikipedia.org/wiki/Wi-Fi.

[12] Wikipedia Contributors. *Ethernet*. 2019. URL: https://en.wikipedia.org/wiki/Ethernet.

[13] electronics-notes.com. *Wi-Fi Channels, Frequency Bands & Bandwidth*. 2019. URL: https://www.electronics-notes.com/articles/connectivity/wifi-ieee-802-11/channels-frequencies-bands-bandwidth.php.

[14] IEEE. *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 Amendment 1: Enhancements for High-Efficiency WLAN*. 2019. URL: https://ieeexplore.ieee.org/document/9442429.

[15] Wikipedia Contributor. *List of WLAN channels*. 2019. URL: https://en.wikipedia.org/wiki/List_of_WLAN_channels.

[16] Ofcom. *Statement on improving spectrum access for consumers in the 5 GHz band and Notice of proposal to make Wireless Telegraphy Exemption Regulations 2017 Consultation on Regulations and proposed technical parameters Statement and Consultation*. 2017. URL: https://www.ofcom.org.uk/__data/assets/pdf_file/0032/98159/5p8-Regs.pdf.

[17] Netgear Support. *What is the difference between 2.4 GHz, 5 GHz, and 6 GHz wireless frequencies?* 2021. URL: https://kb.netgear.com/29396/What-is-the-difference-between-2-4-GHz-5-GHz-and-6-GHz-wireless-frequencies.

[18] U-M Information and Technology Services - University of Michigan. *Devices Incompatible With 5GHz WiFi Frequency*. 2018. URL: https://its.umich.edu/enterprise/wifi-networks/wifi/5hgz-incompatible-devices.

[19] C. Hoffman. *What's the Difference Between Ad-Hoc and Infrastructure Mode Wi-Fi?* 2016. URL: https://www.howtogeek.com/180649/htg-explains-whats-the-difference-between-ad-hoc-and-infrastructure-mode.

[20] Juniper Networks. *Juniper Networks*. 2018. URL: https://www.juniper.net/documentation/en_US/junos-space-apps/network-director4.0/topics/concept/wireless-ssid-bssid-essid.html.

[21] M. Gast. *802.11 Wireless Networks: The Definitive Guide: The Definitive Guide. [online] Google Books, 'O'Reilly Media, Inc.', pp.82–86*. 2005. URL: https://www.google.co.uk/books/edition/802_11_Wireless_Networks_The_Definitive/lX3WatnVUe4C.

[22] K. Cronin and M. Ham. *Open Source Capture and Analysis of 802.11 Management Frames. 17th International Conference on Information Technology–New Generations (ITNG 2020)*. 2020. URL: https://link-springer-com.abc.cardiff.ac.uk/chapter/10.1007/978-3-030-43020-7_81%20[Accessed%2011%20Apr.%202022.

[23] T. Carpenter. *Open Source Capture and Analysis of 802.11 Management Frames. 17th International Conference on Information Technology–New Generations (ITNG 2020)*. 2014. URL: https://link-springer-com.abc.cardiff.ac.uk/chapter/10.1007/978-3-030-43020-7_81%20[Accessed%2011%20Apr.%202022.

[24] ACCL Make Connections. *What Affects Wifi Speed: 17 Factors That Can Impact Your WiFi Connection Speed & Quality*. 2019. URL: https://network-data-cabling.co.uk/blog/wi-fi-17-factors.

[25] M. Pierce. *WiFi Signal Strength: A No-Nonsense Guide*. 2021. URL: https://www.securedgenetworks.com/blog/wifi-signal-strength.

[26] MCS Index. *MCS Index Table, Modulation and Coding Scheme Index 11n, 11ac, and 11ax*. 2022. URL: https://mcsindex.com.

[27] A Granados. *Honey, I shrunk the beacon interval!* 2017. URL: https://www.intuitibits.com/2017/08/28/honey-shrunk-beacon-interva.

[28] M. O'Leary. *The care and feeding of SSIDs*. 2005. URL: https://community.jisc.ac.uk/library/advisory-services/care-and-feeding-ssids.

[29] 7signal. *Wi-Fi Beacons & Beacon Interval: Everything You Need To Know*. 2016. URL: https://www.7signal.com/news/blog/controlling-beacons-boosts-wi-fi-performance.

[30] E. Hozan. *Let's Discuss: WPA2 Packet Captures*. 2019. URL: https://www.secplicity.org/2019/07/23/lets-discuss-wpa2-packet-captures/.

[31] M. Gast. *802.11 Framing in Detail - 802.11 Wireless Networks: The Definitive Guide, 2nd Edition*. 2005. URL: https://learning.oreilly.com/library/view/802-11-wireless-networks/0596100523/ch04.html#wireless802dot112-CHP-4-SECT-3.3.

[32] Internet Assigned Numbers Authority. *Character Sets*. 2022. URL: https://www.iana.org/assignments/character-sets/character-sets.xhtml.

[33] Wikipedia Contributors. *ASCII*. 2019. URL: https://en.wikipedia.org/wiki/ASCII.

[34] Network Working Group. *RFC 3629 - UTF-8, a transformation format of ISO 10646*. 2003. URL: https://datatracker.ietf.org/doc/html/rfc3629.

[35] Network Working Group. *Unicode Format for Network Interchange*. 2008. URL: https://datatracker.ietf.org/doc/html/rfc5198.

[36] Android Developers. *Charset*. 2021. URL: https://developer.android.com/reference/java/nio/charset/Charset.html#defaultCharset().

[37] J. Wilson. *Are there any invalid code points in Unicode?* 2020. URL: https://it-qa.com/are-there-any-invalid-code-points-in-unicode/.

[38] Wikipedia Contributors. *Appendix:Control characters*. 2021. URL: https://developer.android.com/guide/topics/resources/string-resource#escaping_quotes.

[39] A. Dagelić. *Location Privacy and Changes in WiFi Probe Request Based Connection Protocols Usage Through Years*. 2019. URL: https://ieeexplore.ieee.org/document/8783167.

[40] G. Chatzisofroniou. *Association Attacks in IEEE 802.11: Exploiting WiFi Usability Features*. 2021. URL: https://link.springer.com/chapter/10.1007/978-3-030-55958-8_6.

[41] A. Clark. *Passive techniques for detecting session hijacking attacks in IEEE 802.11 wireless networks*. 2005. URL: https://ieeexplore.ieee.org/document/8783167.

[42] P Madani. *RSSI-Based MAC-Layer Spoofing Detection: Deep Learning Approach*. 2021. URL: https://www.mdpi.com/2624-800X/1/3/23/htm.

[43] F. Jiang. *SSIDs in the wild: Extracting semantic information from WiFi SSIDs*. 2015. URL: https://ieeexplore.ieee.org/document/7366361.

[44] A. Amici. *"Careful with that Roam, Edu": experimental analysis of Eduroam credential stealing attacks*. 2022. URL: https://dl.ifip.org/db/conf/wons/wons2022/wons22-final65.pdf.

[45] *Android Developer*. URL: https://developer.android.com/.

[46] *Android Cross References*. URL: https://developers.google.com/style/cross-references.

[47] *Android Source Code*. URL: https://source.android.com.

[48] Jonathan Levin. *Android Internals: Power User's View Volume II*. 2022. URL: http://newandroidbook.com/.

[49] Jonathan Levin. *Android Internals: Power User's View Volume I*. 2005. URL: http://newandroidbook.com/.

[50] *Unlock your Bootloader*. URL: https://motorola-global-portal-en-ca.custhelp.com/app/standalone/bootloader/unlock-your-device-b.

Appendix A

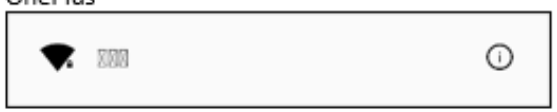
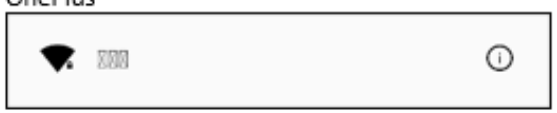
Experiment A

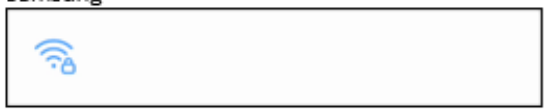
OCT	DEC	CHAR	HEX	CTRL KEY	COMMENTS (^D means to helod the CTRL key and hit d)	Observations Hello(CHAR)World	Observations (CHAR)(CHAR)(CHAR)
\000	0	NUL	\x00	^@ \0	(Null byte)	All 3 phones 	Error: invalid SSID “”
\001	1	SOH	\x01	^A	(Start of heading)	All 3 phones 	All 3 phones 
\002	2	STX	\x02	^B	(Start of text)	All 3 phones 	All 3 phones 
\003	3	ETX	\x03	^C	(End of text) (see: UNIX keyboard CTRL)	All 3 phones 	All 3 phones 
\004	4	EOT	\x04	^D	(End of transmission) (see: UNIX keyboard CTRL)	All 3 phones 	All 3 phones 
\005	5	ENQ	\x05	^E	(Enquiry)	All 3 phones 	All 3 phones 
\006	6	ACK	\x06	^F	(Acknowledge)	All 3 phones 	All 3 phones 
\007	7	BEL	\x07	^G	(Ring terminal bell)	All 3 phones	All 3 phones

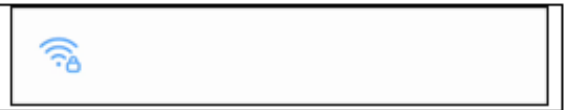
							
\010	8	BS	\x08	^H \b	(Backspace) (\b matches backspace inside [] only) (see: UNIX keyboard CTRL)	All 3 phones 	All 3 phones 
\011	9	HT	\x09	^I \t	(Horizontal tab)	OnePlus, Motorola  Samsung  (no space in between)	All 3 phones 
\012	10	LF	\x0A	^J \n	(Line feed) (Default UNIX NL) (see End of Line below)	Error: Line 13: invalid line 'World'	Line 12: invalid SSID "" /etc/hostapd/hostapd.conf ... ssid=Hello World
\013	11	VT	\x0B	^K	(Vertical tab)	All 3 phones 	All 3 phones  /etc/hostapd/hostapd.conf ssid=^K^K^K
\014	12	FF	\x0C	^L \f	(Form feed)	All 3 phones 	All 3 phones 

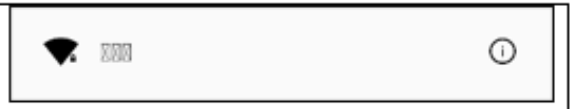
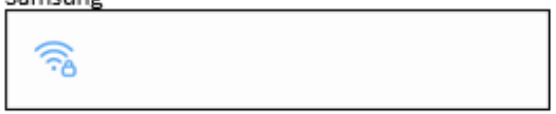
						/etc/hostapd/hostapd.conf ssid=Hello^LWorld	/etc/hostapd/hostapd.conf ssid=^L^L^L
\015	13	CR	\x0D	^M \r	(Carriage return)	Motorola  OnePlus  Samsung 	All 3 phones  /etc/hostapd/hostapd.conf ssid=^M^M^M
\016	14	SO	\x0E	^N	(Shift out)	All 3 phones 	All 3 phones  /etc/hostapd/hostapd.conf ssid=^N^N^N
\017	15	SI	\x0F	^O	(Shift in)	All 3 phones 	All 3 phones  /etc/hostapd/hostapd.conf ssid=^O^O^O
\020	16	DLE	\x10	^P	(Data link escape)	Motorola	Motorola

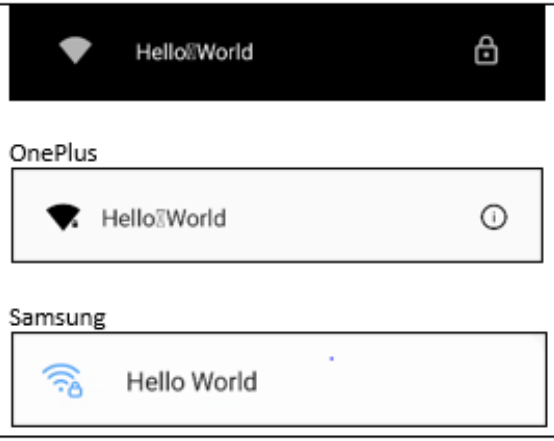
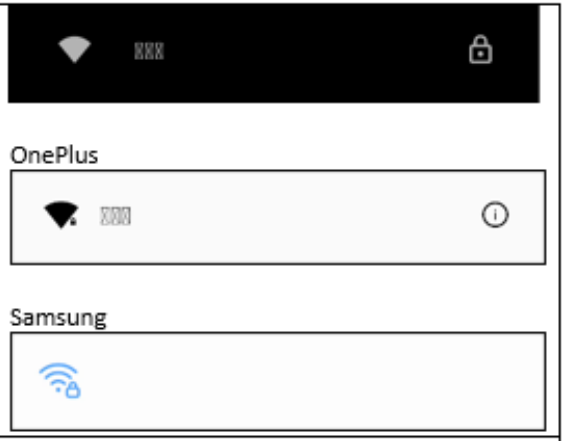
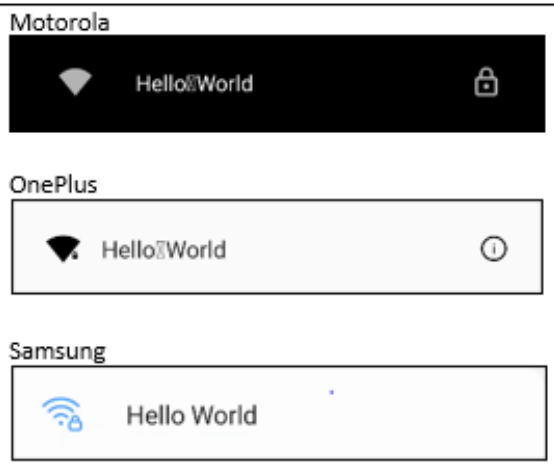
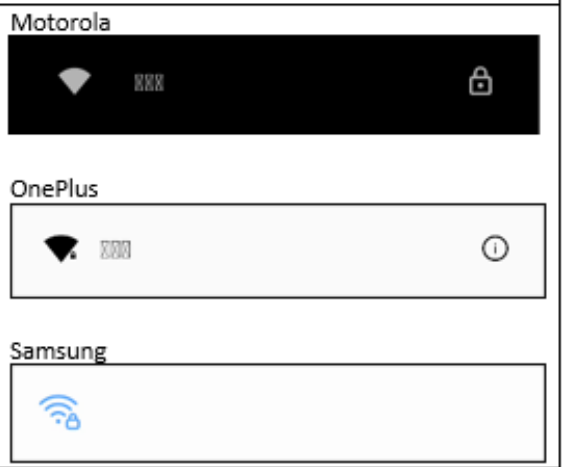
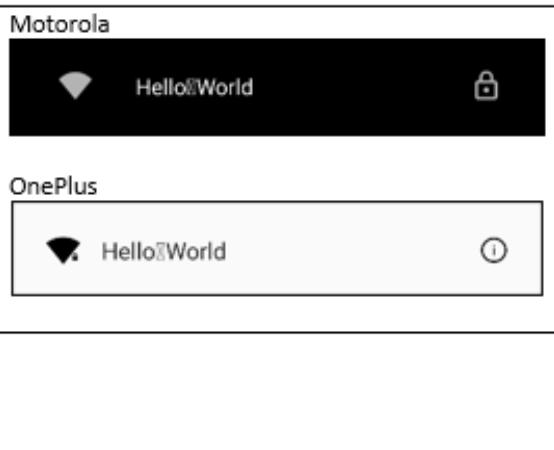
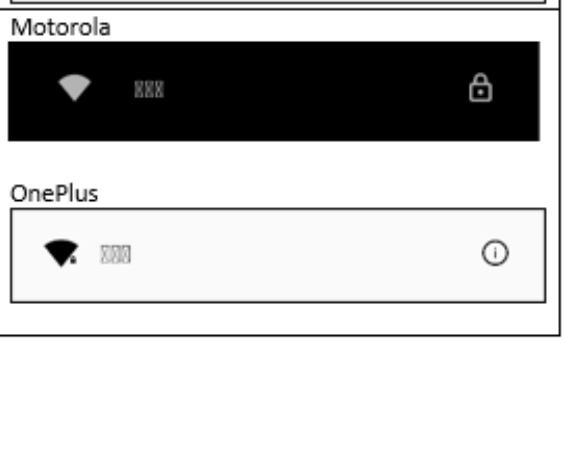
						 OnePlus  Samsung 	 OnePlus  Samsung 
\021	17	DC1	\x11	^Q	(Device control 1) (XON) (Default UNIX START char.)	Motorola  OnePlus  Samsung 	Motorola  OnePlus  Samsung 
\022	18	DC2	\x12	^R	(Device control 2)	Motorola  OnePlus 	Motorola  OnePlus 

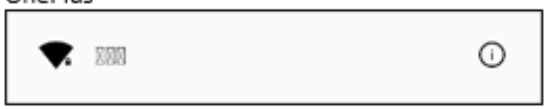
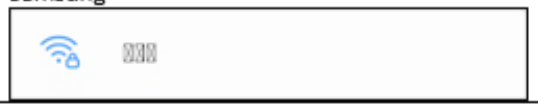
						Samsung 	Samsung 
\023	19	DC3	\x13	^S	(Device control 3) (XOFF) (Default UNIX STOP char.) (Device control 4)	Motorola  OnePlus  Samsung 	Motorola  OnePlus  Samsung 
\024	20	DC4	\x14	^T	(Device control 4)	Motorola  OnePlus  Samsung 	Motorola  OnePlus  Samsung 
\025	21	NAK	\x15	^U	(Negative acknowledge) (see: UNIX keyboard CTRL)	Motorola 	Motorola 

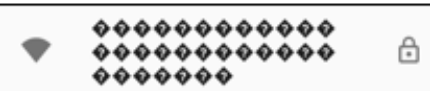
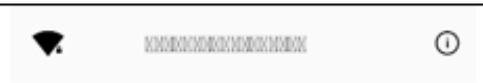
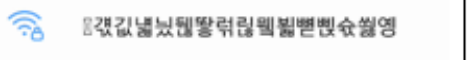
						OnePlus  Samsung  	OnePlus  Samsung 
\026	22	SYN	\x16	^V	(Synchronous idle)	Motorola  OnePlus  Samsung  	Motorola  OnePlus  Samsung 
\027	23	ETB	\x17	^W	(End of transmission block)	Motorola  OnePlus  Samsung	Motorola  OnePlus  Samsung

							
\030	24	CAN	\x18	^X	(Cancel)	Motorola  OnePlus  Samsung 	Motorola  OnePlus  Samsung 
\031	25	EM	\x19	^Y	(End of medium)	Motorola  OnePlus  Samsung 	Motorola  OnePlus  Samsung 
\032	26	SUB	\x1A	^Z	(Substitute character)	Motorola  OnePlus	Motorola  OnePlus

						 Samsung 	 Samsung 
\033	27	ESC	\x1B	^[	(Escape)	Motorola  OnePlus  Samsung 	Motorola  OnePlus  Samsung 
\034	28	FS	\x1C	^\	(File separator, Information separator four)	Motorola  OnePlus  Samsung 	Motorola  OnePlus  Samsung 
\035	29	GS	\x1D	^]	(Group separator,	Motorola	Motorola

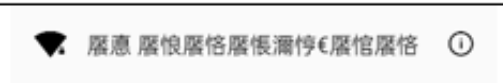
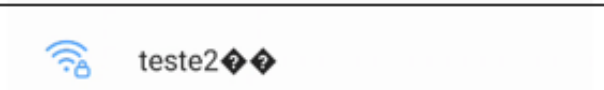
					Information separator three)		
\036	30	RS	\x1E	^^	(Record separator, Information separator two)		
\037	31	US	\x1F	^_	(Unit separator, Information separator one)		

						Samsung 	Samsung 
\177	127	DEL	\x7F	^?	(Delete) (see: UNIX keyboard CTRL)	Motorola  OnePlus  Samsung 	Motorola  OnePlus  Samsung 

ID	94	95	107
Naughty String	Non-white C0 controls: U+0001 through U+0008, U+000E through U+001F, and U+007F (DEL). Appears blank in browsers and text-based formats (e.g., XML) and in theory they should never appear in input.	Non-white C2 controls : U+0080 through U+0084 and U+0086 through U+009F. Commonly misinterpreted as additional graphic characters. It appears blank.	␉␊␋␌ String containing common Unicode symbols.
Motorola			
OnePlus			
Samsung			

Official list position	Position 116	Position 121	Position 155
Observations			

ID	115	116	117
Naughty String	" (Double quotes - 0x22)	' ' (2 x Single quote – 0x27 0x27)	"" (2 x Double quotes – 0x22 0x22)
Motorola			
OnePlus			
Samsung			
Official List Position	Position 173	Position 174	Position 175
Observations	Both ID 115 and 116 look the same to the end-user.		Looks visibly the same as the string at Position 175 in the official list, which is a combination single quote, double quotes, single quote (0x27 0x22 0x27)

ID	134	174	
Naughty String	ɣ ʁ˩˥8∂ɸ˩	‮test (0xe2 0x80 0xaa 0xe2 0x80 0xaa 0x74 0x65 0x73 0x74 0xe2 0x80 0xaa)	
Motorola			
OnePlus			
Samsung			
Official List Position	200	305	
Observations	String containing two-byte letters. We can notice that Samsung and Motorola support most or all UTF-16 encoded characters.	String containing hidden Unicode characters with special properties (format characters – e.g., right-to-left override).	