# Reverse Engineering a Bluetooth Application: Discovering the Secrets of a Makeup Printing Device

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree BS in the
Computer Science at The College of Wooster

by
Natalie Pargas

The College of Wooster
2025

**Advised by:**

Your advisor (Department)

# Abstract

Include a short summary of your thesis, including any pertinent results. This section is *not* optional, and the reader should be able to learn the meat of your thesis by reading this (short) section.

This work is dedicated to the future generations of Wooster students.

# Acknowledgments

I would like to acknowledge Prof. Lowell Boone in the Physics Department for his suggestions and code.

# Vita

Publications

Fields of Study Major field: Major

Minor field: Minor

Specialization: Area of IS research

# Contents

# List of Figures

# List of Tables

# List of Listings

CHAPTER 1

# Introduction

## 1.1 What is Reverse Engineering?

Reverse Engineering is the process of disassembling something in order to understand how it works. It has been done long before it was used for technology. For example, the dissections that were a common high school experience. They were done to break down something as mystifying as life into clear components and functionalities. One of the best ways to fully understand how something works is to open it up.

Reverse engineering software can be done in many ways. The methods of analysis can be divided into two categories, static and dynamic analysis. Static analysis involves combing over a snapshot of the code in a single state. Dynamic analysis is done when the program is being executed and changing states. To start reverse engineering, usually a program's executable is passed to a disassembler which will break it down into the assembly level code. A programmer can use system monitoring tools to display information that was gathered by the operating system on the program and how it interacts with its environment. A debugger shows what happens in the CPU with the disassembled code one line at a time. Using these different tools, the programmer must use their intuition to guide the reversing process.

There are many reasons why someone might want to reverse engineer software. The only time that the workings of software is available to anyone is when that software is open source. Open source software is freely available source code that anyone can view, share, and modify. However, software developers sometimes opt out of publicly distributing the source code. There are a number of potential causes for this, including licensing and ownership issues, the need to preserve proprietary data, or other personal reasons.

Some of the most common reasons a person would choose to reverse engineer software is so that they can perform malware analysis, improve programming skills, recover lost source code, and implement interoperability between programs. My first introduction to reverse engineering was through a video where the creator was reverse engineering Apple's Facetime for Mac so he could reinstate closed captioning for his hearing impaired mother. The list of uses is as extensive as programs out there.

Most software developers know reverse engineering in the context of malware analysis. A classic example is a Trojan virus. Malicious developers can hide malware in programs that are designed to look innocuous. Breaking apart these programs will unveil the malware hidden inside. Developers also often call on external libraries. With large projects, keeping track of dependencies can be hard. Hackers can hide entry points in these libraries so they can access unauthorized information. Reverse engineering is the only foolproof way to reveal exactly what is happening at each step of a program's execution.

Learning how something is being done can be great to improve skills. Reverse engineering requires extensive knowledge of both assembly code and how high level code structuring. Reverse engineering someone else's code is no simple feat and will cause the reverser to need to learn in depth the possible ways a program could be constructed. This project will be an endeavor in learning both forward engineering and reverse engineering.

People may reverse engineer their own program to recover lost source code. While this is no easy task, it is possible to use reverse engineering to recover source code. Many software developers know the pain of losing source code and, with an executable, reverse engineering offers a potential solution.

Interoperability is one of the reasons a person might choose to reverse engineer something. This purpose is also only able to be accomplished through reverse engineering. When working with external libraries or operating systems that have documentation on usage but no source code, oftentimes documentation doesn't cover all use cases. While a programmer could keep throwing things at the wall or try to contact the vendor, reverse engineering provides a surefire way of figuring things out.

## 1.2 Java RE

### 1.2.1 JVM

### 1.2.2 Reversing Bytecode

### 1.2.3 Java Name Obfuscation

#### 1.2.3.1 Obfuscating names for .class file

## 1.3 Mobile RE

### 1.3.1 Hybrid Approach

### 1.3.2 Symbolic Execution

### 1.3.3 Generating UML Diagram

### 1.3.4 Android

#### 1.3.4.1 Tools

##### 1.3.4.1.1 Dalvik VM Android

##### 1.3.4.1.2 APK Tools

### 1.3.5 Apple?

#### 1.3.5.1 Obj C tools

CHAPTER 2

# Bluetooth

## 2.1 Bluetooth

Bluetooth is a widely adopted wireless technology that has seen rapid growth over the past decade. Originally developed to eliminate the need for short-range wired connections, Bluetooth is now used in countless applications—from music and video streaming to medical devices requiring continuous data updates. While it operates similarly to WiFi in some respects, the key difference is its scope: WiFi connects multiple devices to the internet, whereas Bluetooth focuses on direct device-to-device communication and offers more diverse use cases [**intelBLEguide**].

Another distinction is how Bluetooth is optimized for low-latency applications that transmit small bursts of data quickly and efficiently. This makes it ideal for scenarios where power conservation and responsiveness are more important than high throughput.

### 2.1.1 Bluetooth Types

There are two main types of Bluetooth technology: *Bluetooth Basic Rate (BR)*, also referred to as *Bluetooth Classic*, and *Bluetooth Low Energy (BLE)*. Of the two, BLE has gained far more traction in modern applications.

Bluetooth Basic Rate is the older of the two technologies, supported in versions 1.0 through 3.0. It is primarily used for wireless audio streaming in devices like headphones, speakers, and in-car entertainment systems, and it relies on point-to-point communication.

Bluetooth Low Energy, introduced in version 4.0 and further enhanced in version 5.0, offers a broader range of capabilities. In addition to supporting audio and data transfer, BLE enables device networking and location services. It supports multiple communication topologies including point-to-point, mesh, and broadcast, making it far more versatile than its predecessor. One of BLE's standout features is its utility in high-accuracy location services. By leveraging nearby devices, BLE can determine relative positioning and provide precise location tracking.

## 2.2 Pairing-Based Network

The core of Bluetooth is creating a network where two (or slightly more) devices pair with each other, since Bluetooth was designed to replace corded connections. The devices being paired can be broken down into the *Central* and *Peripheral(s)*. These terms are specific to newer Bluetooth standards and can be used interchangeably with the older terms *Master* and *Slave*.

The Central device coordinates the data being sent between it and any Slaves. This includes handling time synchronization, sleep scheduling, and configuring the channels used through frequency hopping [**nextgenBLE**]. Peripheral devices can communicate bi-directionally with the Master device, sending or receiving data depending on the request.

Figure **[insertnumbah from nextgenble]** shows an example of Central and Peripheral devices. Classic Bluetooth includes two network topology configurations: **Piconet** and **Scatternet**.

**Figure 2.1:** Central and Peripheral device relationship

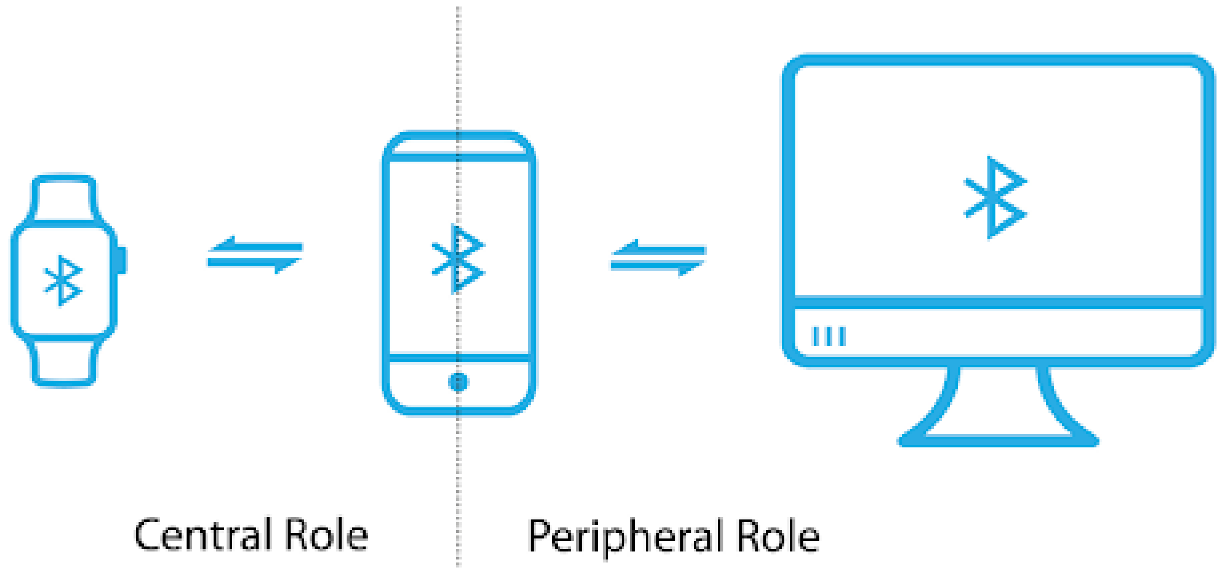


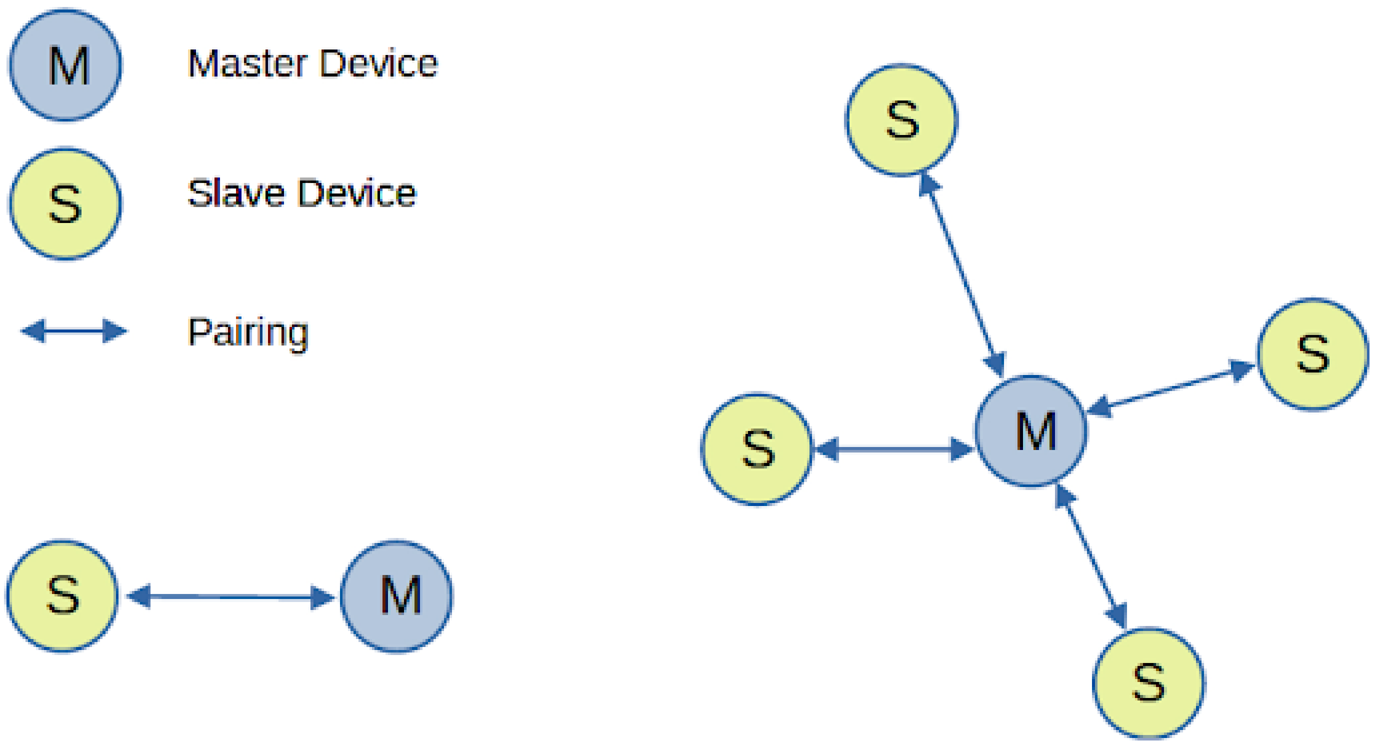
*Figure 7: Multiple roles in BLE*

### 2.2.1 Piconet

A piconet is a type of ad hoc network topology that operates without preestablished infrastructure. Devices in a piconet communicate directly with one another without the need for a centralized device like a router or access point. The network is organized with one central device and one or more peripheral devices. A single central can support up to seven active peripherals, while each peripheral can only connect to one central [**nextgenBLE**]. The central manages the timing and synchronization of all devices in the piconet. Peripheral devices cannot communicate directly with each other; the central acts as a relay, forwarding data between them.

Figure **[insertfigurehere]** shows the structure of a piconet.

**Figure 2.2:** Piconet Topology



All connections made using a BR/EDR controller occur within a piconet. BR/EDR devices communicate over the same physical channel by synchronizing with a shared clock and hopping sequence [**bluetoothcorespec6**]. The piconet clock follows the central device's clock configuration, and the hopping sequence is also derived from the central's clock and Bluetooth device address.

Multiple independent piconets can exist in the same area. Each operates on a separate channel and is organized around a different central device. Bluetooth devices can also participate in multiple piconets simultaneously. While a Bluetooth device can only act as the central in one piconet at a time, it can serve as a peripheral in several.

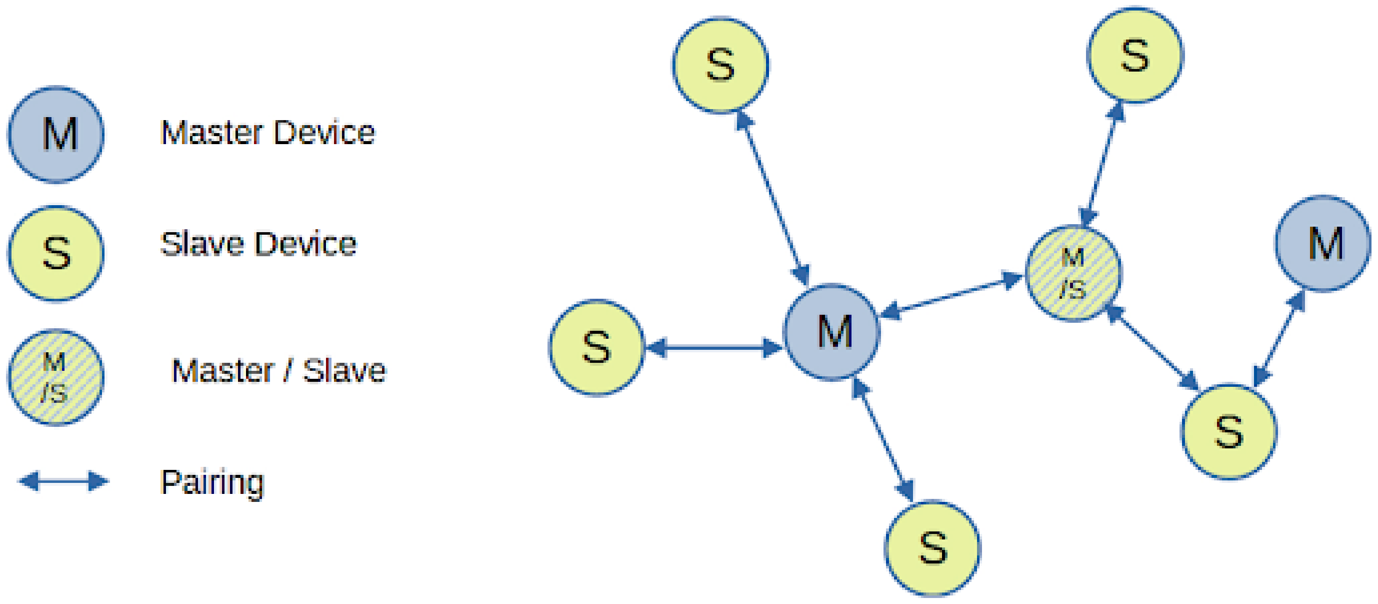
### 2.2.2 Scatternet

A scatternet is formed by connecting multiple piconets together. Unlike a piconet, a peripheral in a scatternet can connect to multiple central devices. Central devices can also connect to each other, sometimes serving dual roles as both central and peripheral.

For example, a scatternet might include a central laptop connected to a Bluetooth speaker, while also acting as a peripheral to a smartphone and printer. Figure **[insertnumber]** shows a diagram of a typical scatternet. Despite the increased complexity, peripherals still do not communicate directly with each other.

Piconets and scatternets are associated with Classic Bluetooth but are still supported in BLE and later Bluetooth versions [**nextgenBLE**]. Although BLE—with its advertising mode—has seen growing popularity, traditional Bluetooth methods still offer advantages such as added security during pairing. BLE services enhance the piconet and scatternet model, providing better security, more efficient pairing, and broader support for services.

**Figure 2.3:** Scatteret Topology



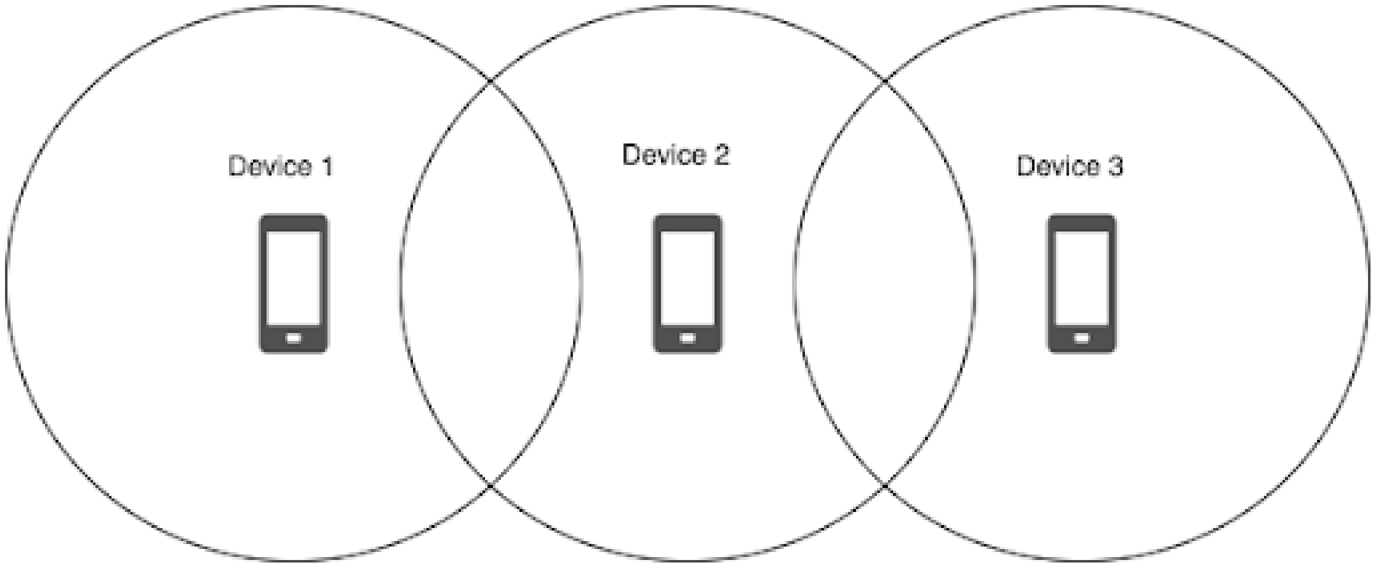
## 2.3 BLE Mesh Networks

Mesh networking, introduced with BLE, marks a significant evolution in Bluetooth topologies. In a mesh network, nodes are arranged in a self-configuring mesh, where each node relays traffic between peers when the destination is not directly connected [**buildingBLEsystems**].

This structure ensures a reliable path for data packets. If one path is blocked or unavailable, the network reroutes traffic dynamically. The self-organizing nature of mesh networks means they adapt in real-time to changes, such as nodes joining, leaving, or moving within the network.

For example, Figure **[insertnumbah]** shows Device 1 out of range of Device 3. Device 2, within range of both, acts as a router forwarding messages between them.

**Figure 2.4:** Mesh topology



## 2.4 Broadcast Network from LE

BLE introduces a new feature called *advertising mode*, which operates differently from the traditional pairing between a central and peripheral. In advertising mode, a device broadcasts data openly to any BLE devices within range that are tuned to the same channel. These listening devices can receive the data and use it as needed—without any pairing, connection, or formal association.

This topology loosely resembles a multi-peripheral piconet, but with a key difference: advertising mode removes the need for a dedicated central or established links. There's no synchronization or connection—just open broadcasting.

Advertising mode has been a major factor in expanding the possibilities of Bluetooth. By offloading the overhead of pairing, BLE made room for a wider variety of lightweight, low-latency applications. This performance shift is part of why so many modern devices—from smart beacons to virtual reality headsets—are feasible today. Devices like VR headsets, which require fast, efficient data sharing without constant handshaking, would be nearly impossible to build the same way without Bluetooth.

## 2.5 Connections and Mixed Topology

A connection becomes necessary when data needs to flow in both directions or when there's more data than can fit into the two available advertising payloads. Connections in BLE are more persistent—they allow devices to remember each other and avoid reestablishing the connection every time. Once established, connections support periodic data exchanges between two devices.

This connection-based pairing is what underpins piconets and scatternets. These connections are private by design: only the two participating devices exchange data (barring external sniffing). As covered earlier, connections involve one central device and one or more peripheral devices.

BLE networks can also incorporate mixed topology configurations depending on the use case. A device that supports both BR/EDR and LE can serve as a bridge between traditional Bluetooth and BLE communication, enabling broader interoperability across protocols. The possibilities for combining these configurations are only constrained by the capabilities of the individual devices' radios and protocol stacks.

Figure **[insertnumbah]** provides an example of a mixed topology Bluetooth network.

## 2.6 SCO and ACL Logical Transports

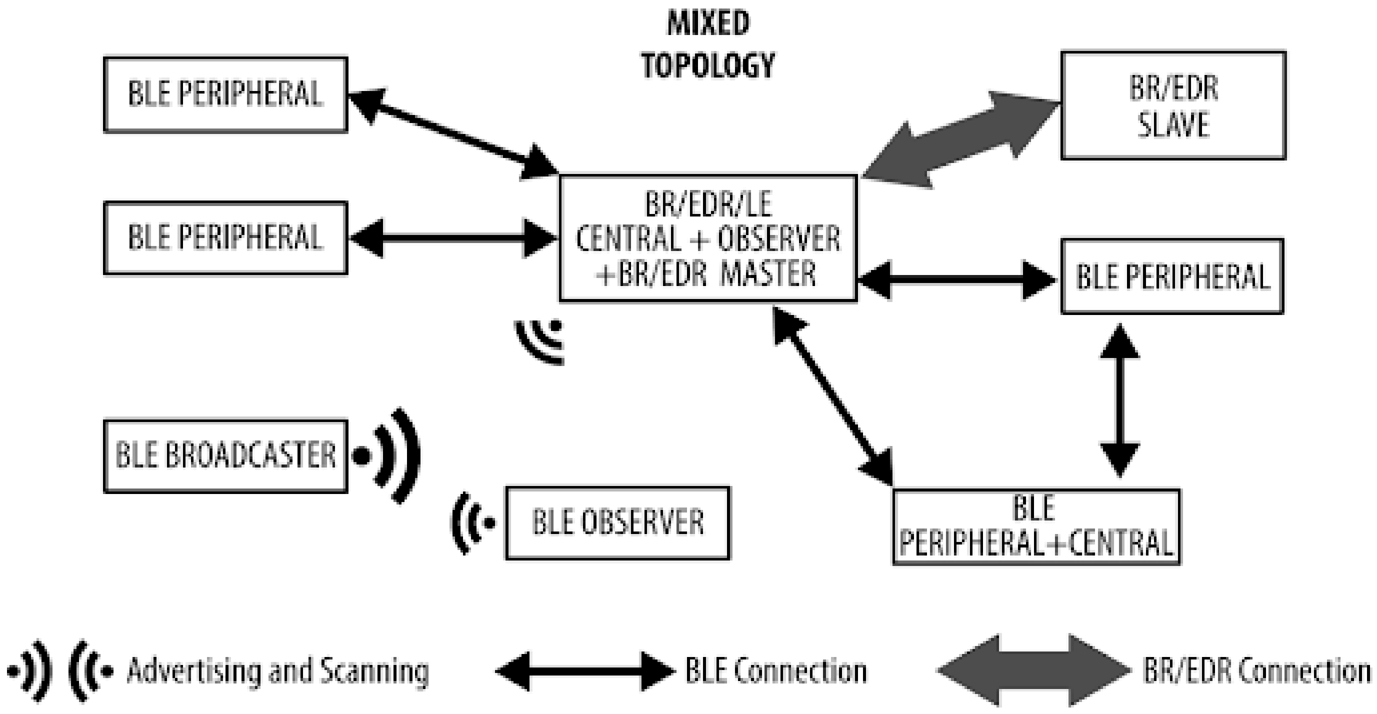
### 2.6.1 BR/EDR Asynchronous Connection-Oriented (ACL)

ACL links are the primary way Bluetooth devices using BR/EDR exchange general data. These links handle both control signals (like LMP and L2CAP) and typical user data. ACL is *asynchronous*, meaning data is sent when available rather than on a fixed schedule.

Every Peripheral in a Bluetooth piconet gets one default ACL link to the Central. This link is established when the device first connects and is identified by a Logical Transport Address (LT_ADDR) assigned by the Central. This LT_ADDR is also used when identifying the physical connection between devices.

However, because multiple types of logical transports (like SCO links, which are used for voice) might

**Figure 2.5:** Mixed Topology



use the same LT_ADDR, it's not enough to identify the ACL connection by LT_ADDR alone. Devices also rely on the packet type to tell them which kind of transport is being used.

ACL links can also carry isochronous data—data that needs to arrive on time, like audio streams—by setting them to automatically drop old packets. At the same time, asynchronous traffic (like file transfers) can still be sent if it's marked not to auto-flush. This means both types of traffic can share the same ACL link if configured correctly.

If the ACL link is removed or if the device loses sync with the piconet, all other connections between the Central and that Peripheral (like SCO) are also dropped immediately [**bluetoothcorespec6**].

### 2.6.2 BR/EDR Synchronous Connection-Oriented (SCO)

SCO links are designed specifically for real-time data like voice. They create a fixed, circuit-like connection between a Central and a Peripheral by reserving slots on the Bluetooth physical channel. These connections carry 64 kbps of data, usually for audio, and are synchronized with the piconet's clock.

There are several SCO configurations that balance audio quality, latency, and bandwidth usage. Every SCO connection uses the same LT_ADDR as the ACL link, so to identify an SCO packet, a device must look at the slot number, packet type, and LT_ADDR together.

Even though SCO reserves bandwidth, the system can override these slots if higher-priority data (like control messages over ACL) needs to get through. This helps maintain overall system reliability and meet Quality of Service (QoS) requirements.

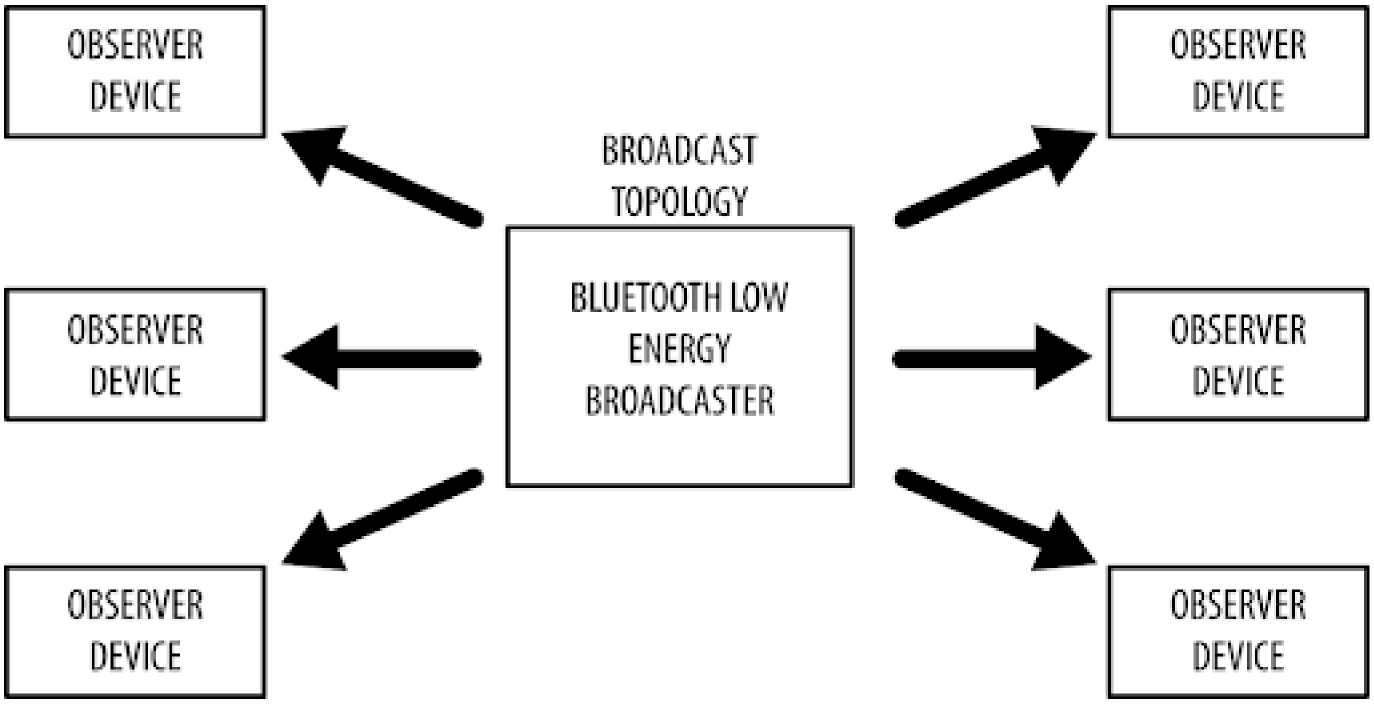
Unlike ACL links, SCO links don't carry complex protocols or layered data structures. They just send a steady stream of audio—either in a standard format or as raw data for the application to decode [**bluetoothcorespec6**].

## 2.7 Bluetooth Low Energy Broadcasting and Observing

Bluetooth Low Energy devices communicate with the outside world in two primary ways: *broadcasting* and *connecting*. Both methods are governed by the Generic Access Profile (GAP), which outlines how devices discover and interact with each other [**gettingstartedwble**].

Broadcasting enables BLE devices to send data one-way to any scanner or receiver within range. It's a form of open communication where any nearby device capable of listening can receive the broadcasted data. Figure **[insertnumbah]** illustrates this communication model. In the diagram, two roles are shown: the *Broadcaster*, which periodically sends non-connectable advertising packets, and the *Observer*, which continually scans for those packets [**gettingstartedwble**].

**Figure 2.6:** Broadcasting and Observing



Broadcasting is a key concept in BLE because it's the only mechanism that allows a device to communicate with multiple other devices simultaneously. This functionality is made possible by BLE's advertising feature.

A standard advertising packet contains a 31-byte payload, which includes information about the broadcasting device and what it offers. This data can be customized to suit different applications, providing useful context to observing devices. If 31 bytes isn't enough, BLE allows for a secondary packet called a *Scan Response*, which provides an additional 31 bytes of payload. Observers can request this secondary data if they need more information [**gettingstartedwble**].

Broadcasting is quick, efficient, and ideal for sending small amounts of data on a regular schedule to multiple receivers. However, it does come with limitations—particularly around security. Since any device within range can intercept a broadcast, there's virtually no privacy. Because of this, broadcasting isn't well-suited for transmitting sensitive or private data.

## 2.8 Bluetooth Protocols and Profiles

Bluetooth has always had a distinction between protocols and profiles. A protocol is a lower layer foundational block that gets used by all devices that have Bluetooth capabilities. They deal with the layers that implement the different packet formats, routing, encoding, decoding, and multiplexing that allows data to be properly sent between peers [**gettingstartedwble**]. Profiles are "vertical slices" of functionality that encompass either fundamental modes of operation common to all devices (such as the Generic Access Profile and Generic Attribute Profile) or support specific use cases (like the Proximity Profile or Glucose Profile). They essentially define how protocols should be applied to accomplish a particular objective, whether general or specialized.

## 2.9 Bluetooth Low Energy Profiles

Profiles specify how BLE protocols are to be used to achieve interoperability and support a wide range of applications. These profiles fall into two primary categories: generic profiles, which are fundamental to all BLE operations, and use-case-specific profiles, which are built on top of the generic layers to support particular functionalities.

### 2.9.1 Generic Profiles

Generic profiles are defined by the Bluetooth Core Specification and provide the foundational mechanisms required for BLE communication. These profiles ensure that devices from different manufacturers can interoperate seamlessly. Two such profiles—Generic Access Profile (GAP) and Generic Attribute Profile (GATT)—are mandatory for all BLE-compliant devices and serve as the cornerstones of BLE communication.

#### 2.9.1.1 Generic Access Profile (GAP)

The Generic Access Profile defines the roles, procedures, and operational modes required for device discovery, broadcasting, connection establishment, connection management, and security negotiation. GAP operates as the top-level control layer within the BLE protocol stack, orchestrating the basic behaviors necessary for device interaction. All BLE devices must implement and conform to the GAP specifications to ensure compatibility and reliable communication.

#### 2.9.1.2 Generic Attribute Profile (GATT)

GATT focuses primarily on how data is exchanged. GATT defines a universal data model and a set of procedures that enable devices to discover, read, write, and notify data values. It functions as the uppermost data layer in BLE and serves as the foundational framework upon which most BLE applications and services are constructed.

Due to their foundational roles, GAP and GATT are commonly exposed through application programming interfaces (APIs), making them the primary entry points for developers interacting with the BLE protocol stack.

### 2.9.2 Use-Case-Specific Profiles

Beyond the generic profiles, the Bluetooth Special Interest Group (SIG) has defined a series of use-case-specific profiles. These profiles are designed to standardize behavior for specific applications and are exclusively built upon the GATT framework. At the time of writing, no BLE profiles exist outside the GATT structure. However, the introduction of connection-oriented L2CAP channels in Bluetooth version 4.1 may pave the way for future GATT-less profiles.

#### 2.9.2.1 SIG-Defined GATT-Based Profiles

The Bluetooth SIG provides an extensive catalog of standardized profiles, each tailored to support a specific application or device type. These profiles fully specify the required procedures and data formats, facilitating rapid development and ensuring interoperability across implementations.

Examples of SIG-defined GATT-based profiles include:

- **Find Me Profile:** Enables physical location of nearby BLE devices (e.g., locating a lost phone or keyring).
- **Proximity Profile:** Detects when a connected device moves beyond a designated range, triggering alerts.

- **HID over GATT Profile:** Facilitates the transmission of Human Interface Device (HID) data over BLE for peripherals such as keyboards, mice, and remote controls.
- **Glucose Profile:** Securely transmits glucose measurement data for health monitoring applications.
- **Health Thermometer Profile:** Enables transfer of body temperature readings from wearable or medical devices.
- **Cycling Speed and Cadence Profile:** Allows cycling sensors to relay speed and cadence information to companion applications or devices.

A comprehensive list of SIG-adopted profiles is available on the Bluetooth SIG Specification Adopted Documents webpage. Developers may also consult the Bluetooth Developer Portal for up-to-date listings of adopted services and characteristics.

#### 2.9.2.2 Vendor-Specific Profiles

While the SIG provides a broad range of standardized profiles, the BLE specification also permits the definition of vendor-specific profiles. These profiles are often developed to support proprietary use cases not covered by SIG standards. Vendor-specific profiles can be implemented privately between two devices (such as with a health accessory and a smartphone) or published to enable broader adoption.

Notable examples of vendor-defined profiles include:

- **Apple iBeacon:** A proprietary proximity-sensing profile used for indoor positioning and location-based services.
- **Apple Notification Center Service (ANCS):** Enables wearable devices or external displays to access and display iOS notifications.

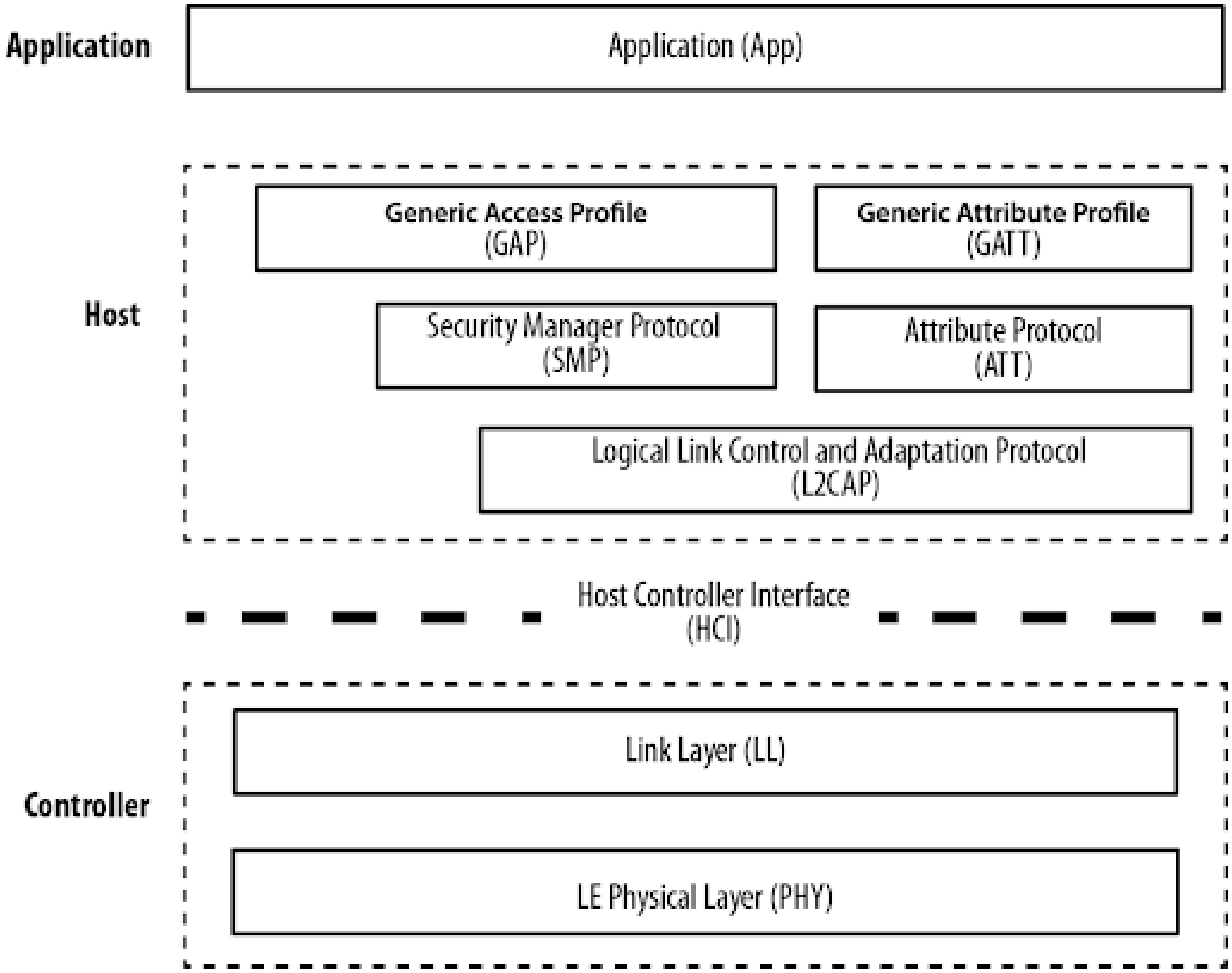
These profiles demonstrate the flexibility of the BLE specification in accommodating both standardized and proprietary communication models.

## 2.10 Bluetooth Low Energy (BLE) Architecture

Although most users will only ever interact with the top layers of the Bluetooth Low Energy (BLE) protocol stack, having a basic understanding of the full stack can provide helpful context. This overview lays the groundwork for understanding how BLE devices work and why each part of the system is important. A typical single-mode BLE device is made up of three main parts: the controller, the host, and the application.

Each of these parts contains multiple layers, each responsible for specific tasks that allow the device to communicate and perform its intended functions. Figure [insertnumah] shows the BLE protocol stack.

**Figure 2.7:** BLE Protocol Stack


Application
Application (App)
Host
Generic Access Profile (GAP)
Generic Attribute Profile (GATT)
Security Manager Protocol (SMP)
Attribute Protocol (ATT)
Logical Link Control and Adaptation Protocol (L2CAP)
Host Controller Interface (HCI)
Controller
Link Layer (LL)
LE Physical Layer (PHY)

## 2.10.1 Application

The application layer sits at the top of the BLE protocol stack. It includes all the logic, data handling, and user interface components that define how a device behaves in a specific use case. This layer is highly customized and varies from one implementation to another [**nextgenBLE**].

### 2.10.2 Host

Below the application is the host, which includes several critical layers that manage communication and device behavior:

- **Generic Access Profile (GAP):** Controls how devices advertise themselves, discover others, establish connections, and manage roles and security [**nextgenBLE**].
- **Generic Attribute Profile (GATT):** Organizes and manages how data is exchanged between connected devices.
- **Logical Link Control and Adaptation Protocol (L2CAP):** Handles multiplexing and segmentation of data packets.
- **Attribute Protocol (ATT):** Supports data operations such as reads and writes via attribute handles.
- **Security Manager (SM):** Manages pairing, authentication, and encryption.
- **Host Controller Interface (HCI) – Host Side:** Facilitates communication between the host and controller [**nextgenBLE**].

### 2.10.3 Controller

At the bottom of the Bluetooth stack, the controller is responsible for radio operations and low-level data transport. It typically includes the following components:

- **Physical Layer (PHY):** Manages the actual transmission and reception of radio signals.
- **Link Layer (LL):** Controls advertising, scanning, and maintaining connections.
- **HCI (Controller Side):** Complements the host side of HCI to enable structured data flow between layers [**nextgenBLE**].

According to the Bluetooth Core Specification, a Bluetooth implementation includes a single controller, which may be configured as one of the following:

- A **BR/EDR controller**, containing the Radio, Baseband, Link Manager, and optionally HCI.
- An **LE controller**, including the LE PHY, Link Layer, and optionally HCI.
- A **dual-mode controller** that combines both BR/EDR and LE controller functions into a single unit.

Together, these layers ensure that BLE devices can connect reliably and communicate efficiently. This layered architecture is often described from the bottom up — starting at the antenna and PHY layer and moving up through the stack to the user-facing application.

## 2.11 BLE Link Layer and Device Addressing

In the Bluetooth Low Energy (BLE) protocol stack, the Link Layer serves as a crucial interface between the Physical Layer—which handles the actual radio transmission—and the upper protocol layers. It acts as the controller for radio operations and is essential for managing the states and timing mechanisms that define BLE communication. One of the key responsibilities of the Link Layer is to abstract the radio hardware, allowing higher-level layers to interact with it through the Host Controller Interface (HCI) [**introtoble**].

In addition to timing and control, the Link Layer manages several hardware-accelerated tasks. These include generating cyclic redundancy checks (CRC) for error detection, producing random numbers for cryptographic operations, and performing encryption to secure data transfer [**introtoble**]. These operations ensure the performance, integrity, and security of BLE communications.

### 2.11.1 Link Layer States

BLE devices operate in a well-defined set of five main states, each with specific behaviors and transitions:

- **Standby:** This is the default state, where the radio is idle and not transmitting or receiving any packets.
- **Advertising:** In this state, the device broadcasts data packets at regular intervals, making itself discoverable to other devices.
- **Scanning:** Devices in this state listen for advertising packets from others, searching for devices they might want to connect with.
- **Initiating:** When a scanning device decides to connect to an advertising device, it enters this state and sends a connection request.
- **Connected:** A persistent communication link is established. Devices in this state can exchange data regularly. The device that initiated the connection becomes the master, while the other becomes the slave [**introtoble**].

This framework enables BLE to support dynamic and responsive device discovery and communication, while keeping energy consumption minimal.

## 2.12 Advertising, Scanning, and Connecting

Advertising and scanning are complementary operations: advertising devices periodically transmit small packets of data, while scanning devices listen for and interpret these packets. If the advertiser allows connections, and a scanning device decides to initiate one, both devices transition into the connected state [**introtoble**]. These processes form the backbone of BLE interactions and are covered in more depth in subsequent chapters of most BLE technical literature.

## 2.13 Bluetooth Device Address

Every Bluetooth device is identified by a 48-bit device address, which serves a function similar to a MAC address in networking. BLE supports two major types of device addresses: public addresses and random addresses, which influence how discoverable and trackable a device is.

### 2.13.1 Public Address

A public address is a globally unique, fixed identifier that is factory-programmed and registered with the IEEE. Because of this registration requirement, it ensures uniqueness across all BLE devices worldwide. However, it can also present privacy concerns, since it does not change and is easily traceable [**introtoble**].

### 2.13.2 Random Address

Random addresses, by contrast, provide more flexibility and are often preferred in consumer devices to protect user privacy. These addresses can either be static or private, and they do not require IEEE registration.

- **Static Address:** Used as a drop-in replacement for public addresses. A static address remains the same between sessions but can be regenerated at boot or kept for the lifetime of the device until a power cycle occurs.
- **Private Addresses:** These are designed to change regularly and enhance privacy. They come in two subtypes:
  - **Non-resolvable private addresses** are temporary, randomly generated, and cannot be traced back to a specific device. They are rarely used due to limited functionality.
  - **Resolvable private addresses** use an Identity Resolving Key (IRK) and a random number to generate temporary addresses. These change over time, making the device hard to track by

unknown parties. However, trusted or bonded devices that have the IRK stored can resolve these addresses, allowing them to re-identify the device securely [**introtoble**].

This system allows BLE to balance two key goals: persistent connectivity with trusted devices and protection against unwanted tracking or eavesdropping by unknown entities.

## 2.14 Host Controller Interface (HCI) Layer

The Host Controller Interface (HCI) serves as the communication bridge between the host and controller layers in a Bluetooth system. These components can be implemented on the same chipset or on separate chipsets. When they are on separate chipsets, HCI is critical for enabling interoperability and communication between them. In such cases, HCI defines both the standardized protocol and the physical transport mechanisms that facilitate message exchange. As shown in Figure [insertnumbah], the host, HCI, and controller are depicted as distinct components. HCI commonly uses transport technologies such as UART, USB, and SDIO to establish physical connections between the host and controller. When the host and controller are integrated on the same chipset, HCI functions as a logical interface rather than a physical one [**introtoble**].

### 2.14.1 Universal Asynchronous Rx/Tx (UART)

UART is a serial communication protocol that uses wired connections to exchange data between devices. It is commonly found in integrated circuits and is especially favored in devices like mobile phones and laptops, where the entire Bluetooth stack is implemented on a single chipset. UART is also well-suited for small, resource-constrained devices powered by microcontrollers, as it is lightweight and efficient in its use of system resources [**nextgenble**].

In the context of the HCI UART protocol, four types of packet transmissions are defined: the Command packet (identified by 0x01), Event packet (0x04), Asynchronous Connection-Less (ACL) packet (0x02), and Synchronous Connection-Oriented (SCO) packet (0x03). Because the receiving HCI layer cannot distinguish these packet types based on content alone, each transmission begins with a leading byte — known as an indicator — which acts as a header to identify the type of packet that follows [**nextgenBLE**].

## 2.15 Connection

To establish a connection in Bluetooth Low Energy (BLE), the central device begins by scanning for advertising peripheral devices that are currently accepting connection requests. Advertising packets can be filtered based on the Bluetooth Address or the contents of the advertising data itself. Once a suitable advertiser is detected, the master sends a connection request packet. If the slave responds, a connection is established [**gettingstartedwble**].

This connection request packet includes several critical parameters:

- **Frequency hop increment:** Defines the hopping sequence used for the duration of the connection.
- Three additional connection parameters:
  - **Connection interval:** The time between the start of two consecutive connection events. This can range from 7.5 ms (for high throughput) to 4 seconds (for minimal power usage).
  - **Slave latency:** The number of connection events the slave is allowed to skip without causing a disconnection.
  - **Connection supervision timeout:** The maximum allowed time between receiving valid packets before the connection is considered lost [**gettingstartedwble**].

Once connected, communication happens through repeated connection events — periodic time slots where the master and slave take turns exchanging packets. A connection event:

- Always begins with a packet from the central.
- Requires the peripheral to respond if it receives a packet.
- Continues until both sides have no more data to send.
- Is repeated at the connection interval until the connection is either closed or lost.
- Can be closed by either the master or the slave. If the central sends a packet and does not receive a response, it waits until the next connection event to resume [**introtoble**].

To manage interference and ensure privacy or security, BLE supports a white list mechanism at the Link Layer. This list defines which Bluetooth device addresses are of interest. Devices not on the list are ignored — their advertising (by scanners) or connection request packets (by advertisers) are simply dropped [**gettingstartedwble**].

## 2.16 Services and Characteristics

### 2.16.1 Attribute Protocol (ATT)

The Attribute Protocol (ATT) defines how a server exposes its data to a client, and how that data is organized and accessed in a Bluetooth Low Energy (BLE) system [**introtoble**].

#### 2.16.1.1 Roles in ATT

There are two main roles in ATT communication:

- **Server:** The server is the device that stores and exposes data, and may allow certain behaviors to be remotely controlled. It receives commands from peer devices and sends responses, notifications, or indications in return. For example, a thermometer acts as a server when it provides access to its current temperature, unit of measurement, battery level, or the interval at which it records readings. Instead of requiring the client to repeatedly poll for changes, the server can proactively notify the client when data updates occur [**introtoble**].
- **Client:** The client is the device that reads data from the server or controls the server's behavior. It sends commands and requests, and accepts incoming notifications and indications. In the thermometer example, a mobile device that connects to the thermometer and reads temperature values is operating as the client [**introtoble**].

#### 2.16.1.2 Data Format: Attributes

The data exposed by the server is organized into attributes. An attribute is a general term for any piece of data available on the server, such as services or characteristics (described later).

Each attribute consists of the following elements:

- **Attribute Type (UUID):** This is a Universally Unique Identifier (UUID) used to distinguish the type of data.
  - A 16-bit UUID is used for Bluetooth SIG-adopted attributes.
  - A 128-bit UUID is used for custom or vendor-specific attributes defined by developers [**introtoble**].
- **Attribute Handle:** This is a 16-bit value that acts like an address assigned by the server to each of its attributes. The handle uniquely identifies an attribute during the lifetime of the connection, allowing the client to reference it directly. Handle values range from 0x0001 to 0xFFFF, while 0x0000 is reserved [**introtoble**].

- **Attribute Permissions:** Permissions control whether an attribute can be read, written, notified, or indicated, and what security requirements are necessary for each operation. These permissions are not specified within ATT itself, but are defined at a higher layer—typically the GATT (Generic Attribute Profile) or application layer [**introtoble**].

## 2.17 Profiles

### 2.17.1 Generic Attribute Profile (GATT)

The Generic Attribute Profile (GATT) defines how BLE devices organize, expose, and exchange data over a connection. It builds directly on top of the Attribute Protocol (ATT), using it as the transport layer for transmitting structured data known as attributes [**introtoble**].

GATT introduces a hierarchical data model centered around three key concepts:

- **Services**
- **Characteristics**
- **Profiles** [**introtoble**]

These elements allow BLE devices to encapsulate user-facing data and device functionality in a standardized, interoperable way. All BLE application-level interactions happen within the framework defined by GATT [**gettingstartedwble**].

### 2.17.2 GATT Roles

Like ATT, GATT supports client and server roles, which are dynamic per transaction rather than fixed per device. This means a single device can simultaneously act as a GATT server for one connection and a GATT client in another [**introtoble**].

- **GATT Server:** Stores and exposes attributes to the client. It responds to requests and may send unsolicited updates via notifications or indications. Every BLE device must at least support a minimal GATT server, even if only to respond with error codes [**gettingstartedwble**].

- **GATT Client:** Initiates communication by performing service discovery, reading/writing attributes, and subscribing to updates. The client has no prior knowledge of the server's structure—it learns through discovery procedures [**gettingstartedwble**].

### 2.17.3 Services

A service is a logical grouping of one or more attributes that collectively provide a certain function on the server. A service always includes at least one characteristic, and often contains supporting attributes like declarations, descriptors, and included services (used to reference other services) [**introtoble**].

- **Primary Services:** Represent core functionality (e.g., Battery Service).
- **Secondary Services:** Offer auxiliary support and must be referenced by primary services (rarely used) [**introtoble**].

The Bluetooth SIG maintains a set of adopted services with published specifications to ensure interoperability across vendors. If a product claims compliance with one of these services, it must strictly follow its specification [**introtoble**].

### 2.17.4 Characteristics

A characteristic is the smallest logical unit of user-accessible data on a BLE server. It always belongs to a service and includes:

- **Value:** The actual data being exposed.
- **Properties:** Operations permitted on the value (e.g., read, write, notify, indicate).
- **Descriptors:** Metadata about the value (e.g., format, units, configuration settings) [**introtoble**].

For example, the battery level characteristic within the Battery Service lets clients read the device's current power level [**introtoble**].

### 2.17.5 Profiles

While services and characteristics define how data is stored and exposed on the server, profiles describe how clients and servers interact, including service usage, connection procedures, and security requirements. Profiles are not discovered over BLE connections like services are; instead, they exist as specification documents, often adopted by the Bluetooth SIG. These define how multiple services should be used together to achieve specific use cases (e.g., Heart Rate Profile, Blood Pressure Profile) [**introtoble**].

Each profile specification includes:

- Required services and characteristics
- Interaction behaviors for both server and client

- Connection and security requirements
- Example usage diagrams and workflows [**introtoble**]

## 2.18 Nordic Semiconductor

### 2.18.1 nRF Connect

nRF Connect for Mobile—formerly known as the nRF Master Control Panel—is a feature-rich application designed for developers working with Bluetooth Low Energy (BLE) devices. Created by Nordic Semiconductor, this free app is available on both Android and iOS platforms and is widely regarded as one of the most powerful tools for BLE development and testing [**buildingBLEsystems**].

The application supports numerous Bluetooth SIG-adopted profiles, including Over-the-Air Device Firmware Updates (OTA DFU), making it especially useful for firmware deployment and debugging [**buildingBLEsystems**].

#### 2.18.1.1 Key Features

[**buildingBLEsystems**]:

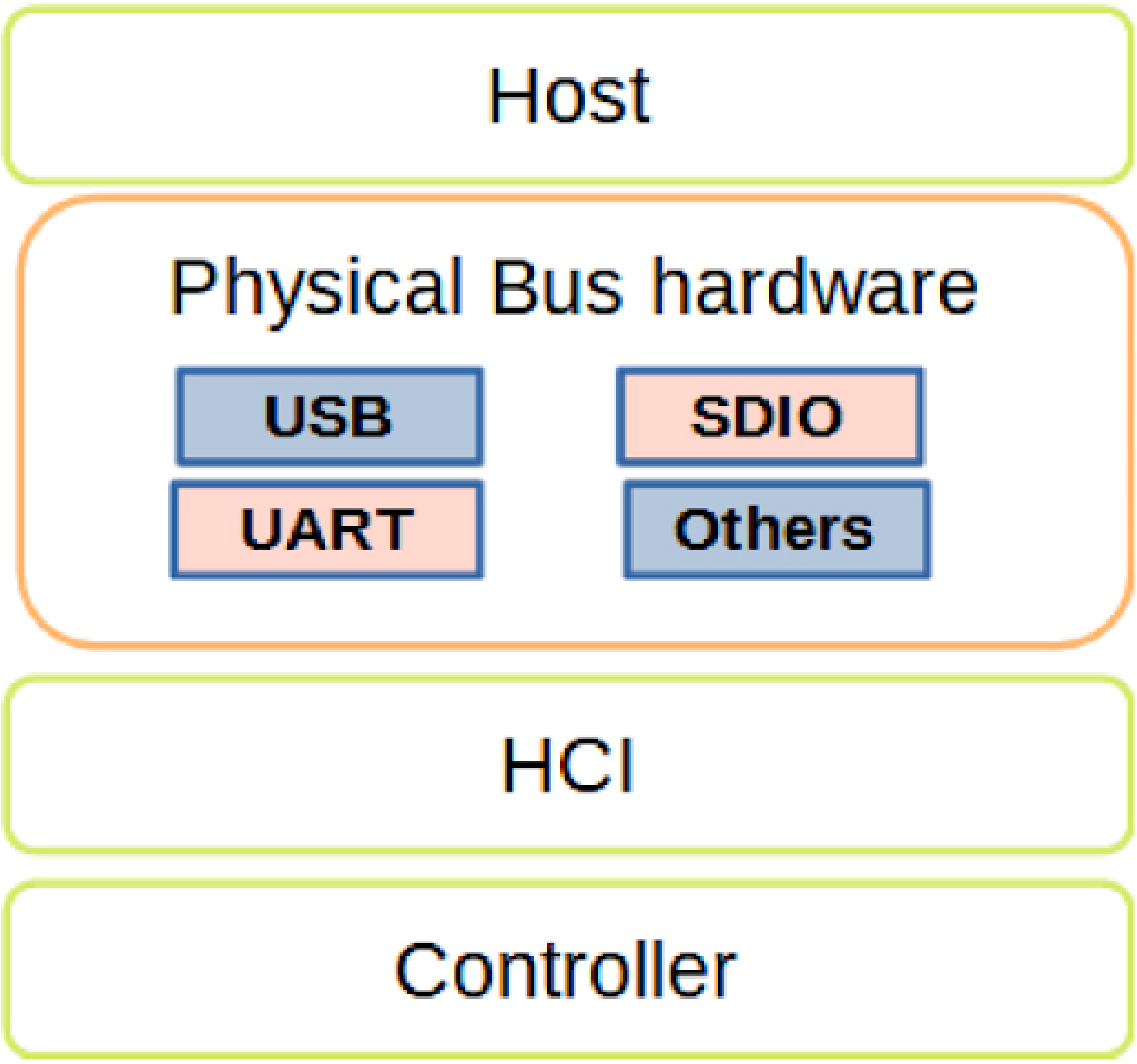
- **BLE Device Scanning with RSSI Graphs:** Visualize signal strength (RSSI) for nearby devices in real-time.
- **Advertisement Data Parsing:** Inspect general advertisements and manufacturer-specific payloads.
- **GATT Client Functionality:** Connect to BLE peripherals and display their services and characteristics.
- **Characteristic Interaction:** Perform reads, writes, and observe values from server-side characteristics.
- **Support for Notifications and Indications:** Enable and receive updates directly from connected devices.
- **GATT Server Emulation:** Simulate a GATT server with built-in configuration presets.
- **OTA Firmware Updates:** Flash firmware over-the-air for supported devices.
- **Automated Testing:** Execute test cases defined in XML scripts for BLE device behavior validation.
- **UART Service Support:** Communicate via UART-over-BLE channels.

#### 2.18.1.2 Compatibility:

- **Android:** Version 4.3 and higher
- **iOS:** Version 8.0 and higher

With its comprehensive feature set, nRF Connect is an essential tool for BLE developers, from prototyping and debugging to final deployment [**buildingBLEsystems**].

**Figure 2.8:** Host, HCI, and Controller components



CHAPTER 3

# Reverse Engineering

## 3.1 Analysis Methods and Tools

### 3.1.1 Static Analysis

Static analysis is a practice of analyzing code without executing it. It can be used to help inform dynamic analysis. Static analysis tools can analyze at either the source code level with code that hasn't been compiled yet or machine language level with code that has. Static analysis tools can be used by reverse engineers to retrieve and comb over assembly level code. This is one of the most basic functions of any static analysis tool, but many of them offer a lot more functionality to help reverse engineers. Most static analysis tools are designed to help give the reverse engineer a starting off point with interpreting assembly level functions.

#### 3.1.1.1 Disassembler

One of the most key static analysis tools for a reverse engineer is a disassembler. Disassemblers are applications that take in a program's executable file and generate a file with the assembly code. This isn't too difficult because assembly is just a text translation of the machine readable binary code. Unfortunately, that makes it a lot less clear than higher level programming languages. Disassembly is also unique to a person's processor, but there are disassemblers that can support more than one CPU architecture. Some examples of disassemblers are IDA, Binary Ninja, Hopper, Ollydbug, Radare2, and Ghidra.

#### 3.1.1.2 Decompiler

Decompilers are similar but more complex than disassemblers. Instead of just producing the assembly language code, decompilers attempt to produce high-level readable code. They attempt to produce something that looks as close to the source code as possible by reversing the compilation process [**Reversing**]. The front end of the decompiler decodes low-level assembly instructions and translates it into an intermediate representation specific to the decompiler. The intermediate representation is then iterated on to remove as many extraneous details as possible while preserving the important details. Lastly the back end takes the polished up intermediate representation and translates it again into a high level language.

While this may sound like a simple way to retrieve the source code, it is highly unlikely the high level language returned will be something that actually matches. Anyone who has ever run text through a translation website multiple times knows that each iteration causes a loss in fidelity and oftentimes the end result will be gibberish. Decompilers go a step further and have no immediate knowledge of things like function or variable names and structures. This does not make them useless, though. For example, a text translator will usually return your original result when given a simple phrase. Similarly a decompiler will most likely be able to pick up on more straightforward functions such as adding a + b. Decompilers offer hints and snippets into what the source code may look like which is valuable information for a reverse engineer. Most of the examples for disassemblers listed also count as decompilers.

#### 3.1.1.3 Dumping Tools

Executable dumping is usually the first step in reverse engineering a program. Executable dumping helps inform a reverse engineer what a program does, how it interacts with external elements, and general knowledge of how a file is structured. Using dumping tools, a reverse engineer will start by figuring out what type of file they're dealing with. They will then start to unpack the format. Extracting strings will immediately offer insight into what language the source code is written in, what library modules it uses, what kinds of data it is dealing with, and what the goals of program functions may be. Other useful information we can retrieve is the layout of the file in memory and what the general logical structure of the file is. Once we know the type of file we also know which tools we can use. For example, some debuggers only work with certain high and low level languages. Some popular executable dumping programs are ones already built into your machine like DUMPBIN for windows or otool for mac.

Otool is identifies a file's mach header. On systems with a Mach kernel like macOS and iOS, the executable binaries they use will have a mach header. All headers have a magic number to identify them. T hin binaries only have that number while fat binaries with fat headers will have locations of the other executable's headers in them

### 3.1.2 Dynamic Analysis

Dynamic Analysis occurs when the program or section being analyzed is run during the analysis process. Because dynamic analysis requires the program to execute with unknown results, it is safest to do it in an enclosed or sandboxed environment to protect the rest of the system. Thus easily manipulated virtual machines are usually set up during the dynamic analysis process. Many tools can be part of the dynamic analysis process. Anything that needs to run and monitor some output is considered dynamic. Tools may run the full program, a portion of the program, or even just a single line. Tools that monitor the external environment while the program runs are also considered dynamic.

#### 3.1.2.1 Debugger

Debuggers are a tool usually used for developers to locate and work through errors in their programs, but they're also vital tools in reverse engineering. Many debuggers can work through assembly language. While assembly language may be difficult for a human to parse through, it is the exact same logic that gets broken down and sent to a computer meaning that the computer can show what each assembly instruction is intended to do. Most debuggers software engineers use were actually designed from the ground up with the purpose of stepping through assembly code. The debugger can show the state of CPU registers along with a memory dump that shows what's in the stack. Debuggers usually contain disassemblers which as discussed is a vital reverse engineering tool. Many debuggers also contain both software and hardware breakpoints. Software breakpoints are instructions added into the code during runtime to pause and hand control over to the debugger. Hardware breakpoints are a CPU feature which allow the processor to pause execution when a certain memory address is reached and hand control over to the debugger. A reverse engineer can use insert breakpoints at data structures of interest and use the debugger to reveal what they are. Debuggers also offer a clear view of registers and memory to aid in a reverse engineer's ability to process low level information.

**3.1.2.1.1 User Mode Debuggers** Most debuggers that a programmer will use are user mode debuggers. They run on a system like any other application then seize control of the target program to debug [**Reversing**]. An advantage to them is that they are easier to set up and navigate than their kernel-mode counterparts. Usually it's fine to stay limited to user mode viewing of an application. It only creates troublesome limitations when the target application has kernel mode components such as device drivers. User mode debuggers are also not always sufficient when trying to debug a program before it reaches the main entry point. These kinds of programs are usually ones that have a lot of statically linked libraries in the executable. The final and most likely to be problematic issue is that user-mode debuggers can only view a single process in a program. This can cause issues when the target application has processes that interact in unknown ways. The user may not know which process to zero in on that has the code of interest.

**3.1.2.1.2** KERNEL MODE DEBUGGERS Kernel mode debuggers are different from user-mode debuggers because they capture the entire system and not just a single process. Instead of running atop the operating system, kernel mode debuggers sit alongside the system kernel and stay ready to capture the entire system's stats. Kernel mode debuggers can be more helpful to reverse engineers because they offer more clues as to what's going on with the system. A key tool for kernel-mode debuggers is that they allow the placement of low level code breakpoints. As a reverse engineer working with assembly code, being able to test different sets of assembly instructions that may be the code section a reverser is looking for is incredibly useful.

For example, picture a scenario where a reverse engineer is looking for the API responsible for handling moving windows? That will need to be managed by a windows manager in the kernel. The complexity is introduced when trying to identify which API moves a particular window. Since there are multiple APIs that can be used to move a window, pinpointing the one you are trying to focus on is a difficult task. This is where kernel mode debuggers come in handy. Low level breakpoints in the operating system responsible for shifting windows around will help you identify which API is getting called when a window is moved [**Reversing**]. From there the reverse engineer will be able to hone in on that API.

Unfortunately, kernel mode debuggers come with their own drawbacks. Setting them up can be challenging because they require access to the full system. They need to suspend the entire system while running so they can go line by line, which means the system can no longer have multiple threads going [**Reversing**]. They will also usually need to be set up inside a Virtual Machine which will be discussed in a later section [**PracticalRE**]. These are reasons why a reverser should exhaust their other options before jumping into using a kernel mode debugger. Still, they are powerful tools when the scenario calls for it.

#### 3.1.2.2 SYSTEM MONITOR

System monitoring can be a crucial part of the reverse engineering process. In some cases, a reverse engineer can figure out what they need through system monitoring tools without ever looking at the decompiled code. System monitoring tools can capture what happens between the code and hardware using the intermediate channels of input/output [**Reversing**]. For local system monitoring, tools monitor things such as file operations (creating, deleting, moving, etc). For programs that communicate over networks, system monitoring might look like recording all TCP/UDP network traffic. System monitoring tools are commonly used in virtual machines.

#### 3.1.2.3 Virtual Machine

Many programmers will encounter virtual machines (VM) at some point during their career. It's not uncommon for an application to interface exclusively with one type of operating system. One reason is because writing applications to work with different operating systems adds time and complexity that not all programmers can afford. Choosing to write in a high level language may make code more portable because there's more verbose built-in libraries or interpreters that deal with the specifics of the systems without the need for any programmer intervention. But high level languages often trade performance for convenience which is not a trade that a developer working with large amounts of data can necessarily afford to take. What happens when a developer has an application that runs on Windows while they have a Mac? This is where a virtual machine enters into the mix. Virtual machines are safe sandboxed environments that work as miniature computers with their own operating systems within a computer. They are constructed with an interpreter and bytecode. During compile time, select code snippets are compiled specific for the VM target architecture and then inserted into the program along with the interpreter. During run-time, the interpreter begins executing the bytecode. Virtual machines are costly to implement because they are so expansive which is why only the necessary code snippets get rendered [**MasteringRE**]. Because virtual machines work in their own isolated environment, they offer powerful protections against unknown or potentially malicious software. This makes them a useful tool for reverse engineers who deal with software that they do not know the contents of. The level of control over virtual machines also makes them a useful tool because there is ultimate knowledge of system conditions.

## 3.2 Compiled Code RE

It is important for a reverser to understand the different goals of high level and low level languages. Understanding these differences will help with interpretation of why code is structured a certain way.

### 3.2.1 High level languages

High level languages exist to take away the complexity of system specific programming. High level languages work so that a programmer can focus on specifying the clean structural logic without needing to dedicate time to figuring out system specific details. While writing in something like assembly can be very performative, it would be nearly impossible for a standalone human to write a modern application in assembly. Because high level languages favor simplicity over the flexibility to do exactly what is the most efficient for achieving a program's goals, many programmers don't even know what is going on at the assembly level. A reverse engineer will be forced to pick apart what potentially roundabout ways a program's goals are being achieved. The main goal of a reverse engineer is to use what they know about what the program was trying to do, potential ways that can be accomplished in high level code, and how to read assembly code to make their best educated guess on what's happening where. For reverse engineers, the most important thing to know about a high level language is to what level does it abstract or conceal the underlying machine code [**Reversing**]. Languages like Python that have a built in interpreter will abstract it a lot. These programs may be full of extraneous machine code. On the other hand, languages like C are written much closer to the target processor and won't have nearly as much separation between the source and machine code.

#### 3.2.1.1 Control Flow

Control flow is what makes code more user friendly. This manifests through statements like conditionals that give general instructions for what a program should do when. ′ A processor has no knowledge what statements like 'if' or 'while' mean. Under the hood, these statements translate into verbose and daunting assembly code. This is because high level conditional statements are often broken down into operation sequences because it would otherwise overwhelm the processor [**Reversing**].

Other typical structural components of programs that aid in control flow are switch blocks and loops. Switch blocks, or $n - way$ conditionals, take in an input and have $n$ number of code blocks that are potentially executed depending on the input value. Each block of code gets assigned at least one value prior to runtime. The compiler also generates code to receive the input value and search for the proper code block to execute. The values for the code blocks are usually stored in a lookup table that has pointers to each corresponding code block. Depending on the input value, the program will go through the process of searching the lookup table then jumping to the proper code block at runtime [**Reversing**]. Loops work to allow a program to repeatedly execute a certain code block multiple times. A loop has a counter to keep track of how many iterations it has performed. There is also a conditional statement that determines when the loop will stop. Loops and conditionals are inherently intertwined. The difference is that loops execute over and over until the condition is no longer met.

To understand control flow sequences, one must understand how low level control flow is implemented. This means that a reverse engineer needs to know the specific rules of each kind of low level architecture because low level control flow is individual to the platform and therefore the language.

### 3.2.2 Low level languages

Earlier we mentioned that complexity is introduced when working with low level system specific details. This is especially true when the goal is to translate high level logic into something that will be understood by your machine. Of course, there must be a way to do this because the CPU does it anytime a modern application is run. But it often involves keeping track of more details than a sole human is capable of. That is why a reverse engineer must develop the skill of parsing through assembly code and being able to create a kind of 'mental image' of how it relates to high level constructs. Data management is one of the things that a person's computer keeps track of that would be difficult for a human to do. To understand why this is, we can look at the data management of a relatively low level language, C, versus the assembly representation. Consider the code:

```
int divide(int a, int b) {
        int result;
        result = a // b;
        return result;
}
```

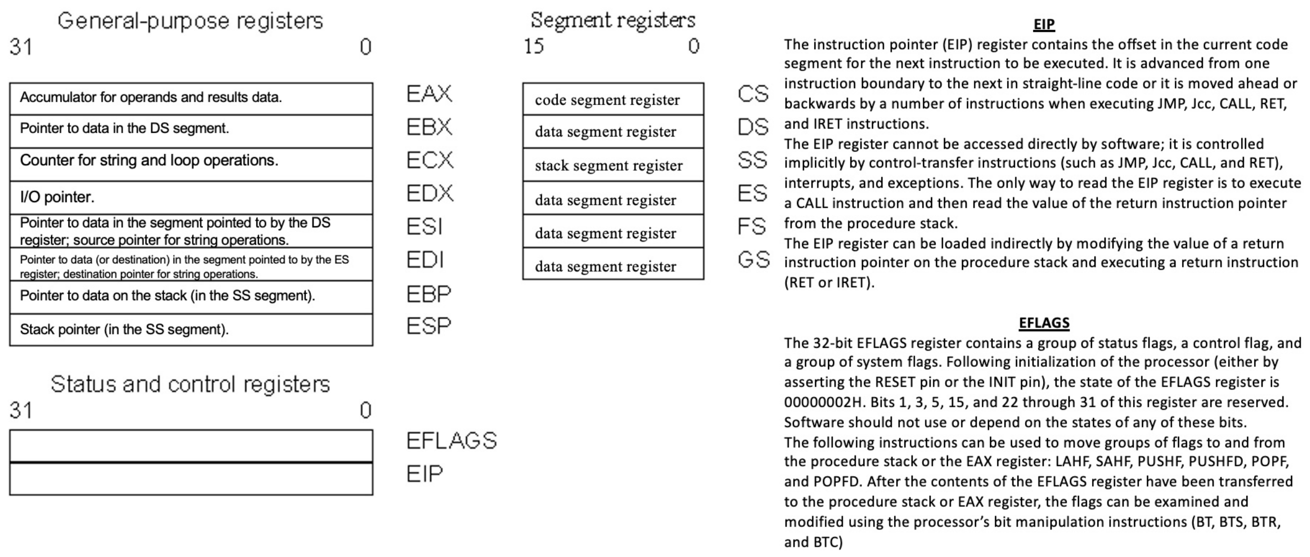
While this function may seem incredibly simple, there is no direct translation into a machine code representation. To execute this in a low level language, it would require first storing the machine state. Then memory would need to get allocated for result. Variables a and b would need to get loaded from memory into a register. Then a would need to be divided by b and the result would need to get stored in the register that got loaded in the beginning. The machine state from before would need to get reloaded. The pointer would need to return to the caller and bring back result [**Reversing**]. One line of high level code could result in any number of assembly instructions. Managing data is one of the biggest challenges of reverse engineering.

#### 3.2.2.1 Registers

To keep from needing to constantly access RAM, microprocessors have a smaller internal memory that can be accessed with barely any performance cost [**Reversing**]. There are multiple types of internal memories that microprocessors keep and registers are one of them. Registers are little pieces of internal memory that live within the processor and are able to be accessed with an imperceptible cost to performance [**PracticalRE**]. The biggest problem with registers is that there are not very many of them. One of the most popular processors, the IA-32, only has eight 32-bit registers that can be used for anything. It does have more registers, but they all have highly specific use cases. Assembly language is written around utilizing registers because they're so performant. They are not good for long term storage, though, that is when RAM becomes the better choice. One of the most important things to take away is that CPUs do not automatically manage this. Data management is outlined in the assembly code which is what reverse engineers will need to get comfortable sorting through.

One thing that reverse engineers will need to do is focus on figuring out what kind of values are getting loaded into a register. For example, it is easy to see when a register is only being used to grant instructions access to a specific value because that register will only show up when transferring value from memory to the instruction or vice versa. Another example is when a register shows up many times in one function. This is a good clue that one of the function's local variables is being stored in that register.

**Figure 3.1:** Figure modified using Intel® 64 and IA-32 Architectures Software Developer's Manual and https://flint.cs.yale.edu/cs422/doc/pc-arch.html#register


General-purpose registers
31
0
Accumulator for operands and results data.
EAX
Pointer to data in the DS segment.
EBX
Counter for string and loop operations.
ECX
I/O pointer.
EDX
Pointer to data in the segment pointed to by the DS register; source pointer for string operations.
ESI
Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations.
EDI
Pointer to data on the stack (in the SS segment).
EBP
Stack pointer (in the SS segment).
ESP
Status and control registers
31
0
EFLAGS
EIP
Segment registers
15
0
code segment register
CS
data segment register
DS
stack segment register
SS
data segment register
ES
data segment register
FS
data segment register
GS

**EIP**

The instruction pointer (EIP) register contains the offset in the current code segment for the next instruction to be executed. It is advanced from one instruction boundary to the next in straight-line code or it is moved ahead or backwards by a number of instructions when executing JMP, Jcc, CALL, RET, and IRET instructions.
The EIP register cannot be accessed directly by software; it is controlled implicitly by control-transfer instructions (such as JMP, Jcc, CALL, and RET), interrupts, and exceptions. The only way to read the EIP register is to execute a CALL instruction and then read the value of the return instruction pointer from the procedure stack.
The EIP register can be loaded indirectly by modifying the value of a return instruction pointer on the procedure stack and executing a return instruction (RET or IRET).

**EFLAGS**

The 32-bit EFLAGS register contains a group of status flags, a control flag, and a group of system flags. Following initialization of the processor (either by asserting the RESET pin or the INIT pin), the state of the EFLAGS register is 00000002H. Bits 1, 3, 5, 15, and 22 through 31 of this register are reserved. Software should not use or depend on the states of any of these bits.
The following instructions can be used to move groups of flags to and from the procedure stack or the EAX register: LAHF, SAHF, PUSHF, PUSHFD, POPF, and POPFD. After the contents of the EFLAGS register have been transferred to the procedure stack or EAX register, the flags can be examined and modified using the processor's bit manipulation instructions (BT, BTS, BTR, and BTC)

### 3.2.2.2 The Stack

The stack is one of two places that a value can be set aside. Registers are not managed by a processor and to use one you just need to load a value in it. Often there will not be registers available or there is a particular reason a variable will need to reside in RAM instead of in a register. That's when you'd put a variable on the stack instead [**Reversing**].

The stack is a short term storage space in memory that gets utilized by both the CPU and the program. It is the spot where short term information gets put when a register can not be used for whatever reason. Registers are for the shortest term data while the stack is for the second shortest. The stack lives in RAM like all other data and is just a carved out section for intermediate term data. Modern operating systems tend to manage multiple stacks at the same time. Each of the concurrently running stacks is a representation of a program or thread [**MasteringRE**].

Stacks use Last In First Out data management. Items are pushed on the top of the stack and popped from the bottom of the stack. Memory in stacks is allocated from the top down where the first in address is allocated and used first while the stack grows backwards towards lower addresses [**Reversing**].

#### 3.2.2.3 Flags

One of the registers you may have noticed from the previous figure is the EFLAGS register. This is a collection of special IA-32 registers that contain system and status flags. The system flags' job is to manage the different modes and states of the processor. The status flags are what a reverse engineer will typically be more interested in and are used by the processor to record its current logical state [**Reversing**]. They are often updated by various logical and integer instructions so they can record the outcome of these actions. There are also instructions whose operation conditions are dependent on the values for the status flags. This is what allows sequences of instructions to run different operations depending on what different input values may be, etc.

Flags in IA-32 are the crux of conditional code. Arithmetic instructions check operand conditions and then set processor flag conditions based on the resulting values. There are also sets of instructions designed to read the flags and do different operations based on the value. An example of a popular instruction set is Jcc or the Conditional Jump. It tests for predefined flag values and then jump depending on whether or not it matches with the specified conditional code [**intelManual**].

#### 3.2.2.4 Functions

Instructions are the actual actions specified in assembly code. They are formatted with operation code (opcode) and one or two operands [**Reversing**] The opcode is what you are asking the computer to do such as MOV or JMP. The operand is the parameters that get passed to the opcode or the data being manipulated. Certain instructions will have no parameters. Data in assembly comes in three basic forms. There are register names, immediate data, and memory addresses. Register names are the names of the general purpose registers to be read from or written to like the EAX, EBX, etc. Immediate data is constant values that are embedded into the code and usually indicates there was a hard coded value in the source code. Memory addresses are the locations of operands stored in RAM. It can be hard coded and tell the processor where to read to and write from. It could also be a register with a value that will be used as a memory address. It is also possible to combine a register with an arithmetic operation and a constant so that there is some base address and then an index offset [**Reversing**].

General purpose instructions are the ones that deal with program flow, logic, arithmetic, string operations, and basic data movement. They deal with data stored in memory at the general purpose registers, EFLAGS register, segment registers, and address information stored in memory [**Reversing**].

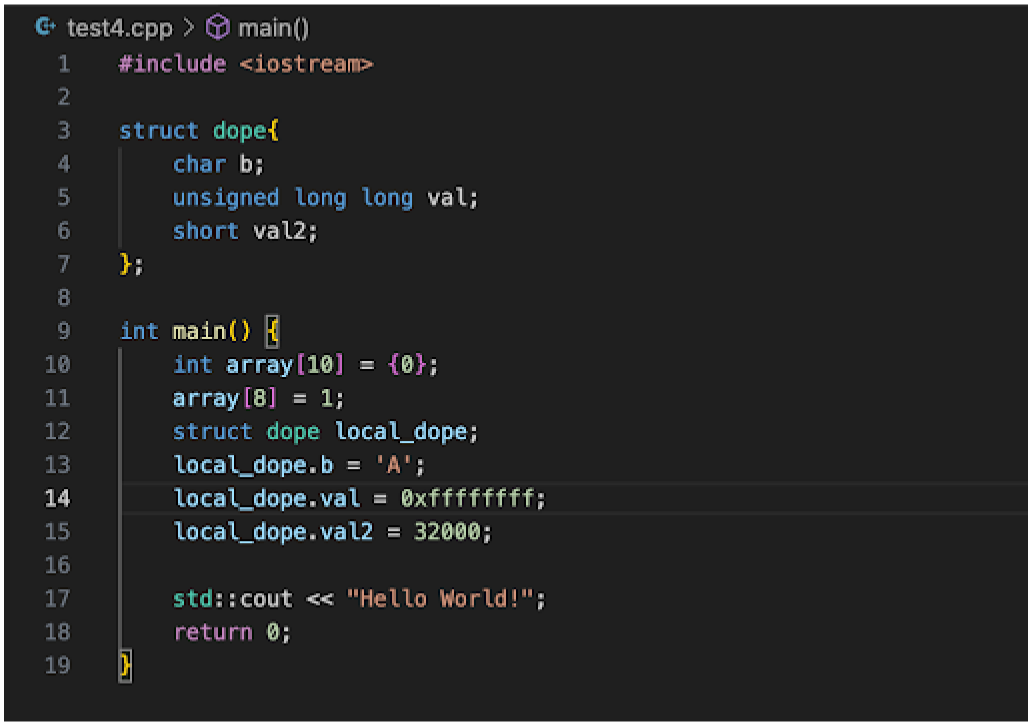
The MOV instruction shows up most frequently in most IA-32 instruction sets. This one deals with basic data movement. It takes in a destination operand and a source operand then moves the data from the source to the destination. The sources can be registers, immediate, or memory addresses. It is important to note that MOV cannot transfer data through memory, it can only take it out or put it in. The destination address can be a memory address (using a register or an immediate) or a register [**Reversing**].

## 3.3 Example Problems

### 3.3.1 Reversing Hello World

Before taking on the larger endeavor of reversing a program written in a declarative language (SwiftUI), it's important to start with the basics. Figure [insert figure here] shows a simple 'Hello World' program in C. To add some complexity, a few random variables and data structures were thrown in as Easter eggs. This is the program whose decompilation will be used as a jumping off point. Figure 3 is the decompilation of the program once it was put through Ghidra. The first thing to note is how with even a relatively low level language like C, the instructions in the code more than doubled. The disassembly being shown in the figure is also only a portion of Ghidra's output. The figure only shows the function section of the program output when imports, exports, labels, classes, etc take up the large majority of the program. One important part of reverse engineering is sorting through all of the information to find out what exactly is important to the reverser. Some reverse engineers refer to this process as finding the shape and edges of the data. With the important section identified, it is time to analyze the assembly code. One thing to keep in mind with the disassembled code is that all the values will be stored in hexadecimal. A good place to start is with the values we know will need to be stored. In the array there are 10 variables which all get set to 0, except for the eighth array member which is changed to equal 1. In our local_dope struct there are three variables equal to 'A', '0xffffffff', and 32000. The highlighted line in the figure shows 'A' being stored in the register w8. Shortly after, #0xffffff shows up without needing to be translated into hexadecimal. #0x7d000 is the hex for 32000. This surface level analysis offers a glimpse into the methodology of more complex reverse engineering. Lastly, the 'Hello_World!' gives us the final piece of information we need to know about what the program is.

**Figure 3.2:** Hello World Program



test4.cpp > main()

```cpp
#include <iostream>

struct dope{
    char b;
    unsigned long long val;
    short val2;
};

int main() {
    int array[10] = {0};
    array[8] = 1;
    struct dope local_dope;
    local_dope.b = 'A';
    local_dope.val = 0xffffffff;
    local_dope.val2 = 32000;

    std::cout << "Hello World!";
    return 0;
}
```

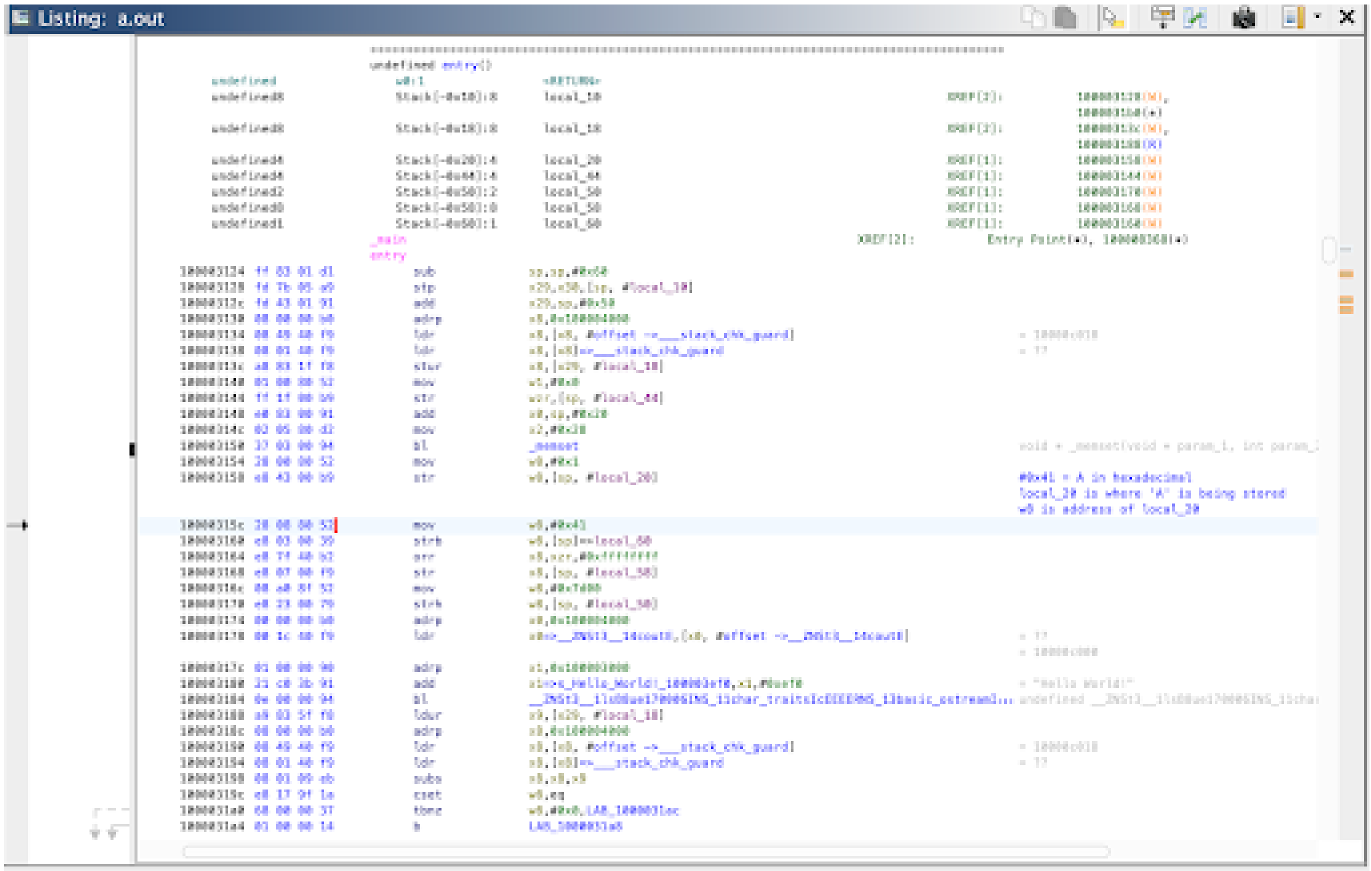
### 3.3.2 Crack Me

The next step in practicing reverse engineering is trying a Crack Me problem. Crack Me problems are toy problems designed by other software developers to help reverser's practice their skills. Usually there's an executable that opens up a little puzzle. The puzzles usually involve finding a password but can involve any number of things like decryption, key generation, etc. This section will cover the process of reverse engineering a simple crack me.

The first step I took in reversing this program was checking the header and libraries as shown in Figure 4 and 5.

Both of these steps were done with an executable dumping tool called 'otool'. The header information shows that the assembly language being used is x86. The library dump shows that the only library being linked is the system library. While these tools didn't offer that much information about this particular program, they are very useful in programs that have more dependencies and more specific assembly languages. These dumping tools ended up being vital later in the project because they offered insight into what libraries were being linked from the project's declarative language as well as what assembly language was being used with Apple's modified version of ARM64. Next, otool was used once again to list the strings. This is a vital part of reverse engineering and is useful to be able to refer back to during the process. Figure 6 shows the result of the string dump.

**Figure 3.3:** Hello World Disassembly



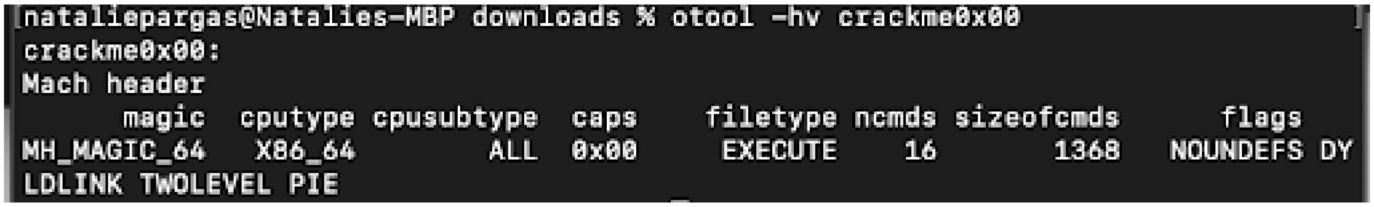
```
                    ****************************************************************
                    undefined entry()
     undefined         w0:1           <RETURN>
     undefined8        Stack[-0x10]:8 local_10                    XREF[2]:     100003128(W),
                                                                               1000031a0(*)
     undefined8        Stack[-0x18]:8 local_18                    XREF[2]:     10000312c(W),
                                                                               100003188(R)
     undefined4        Stack[-0x20]:4 local_20                    XREF[1]:     100003158(W)
     undefined4        Stack[-0x44]:4 local_44                    XREF[1]:     100003144(W)
     undefined2        Stack[-0x50]:2 local_50                    XREF[1]:     100003170(W)
     undefined8        Stack[-0x58]:8 local_58                    XREF[1]:     100003168(W)
     undefined1        Stack[-0x60]:1 local_60                    XREF[1]:     100003160(W)
                    _main                                  XREF[2]:     Entry Point(*), 1000083681(*)
                    entry
 100003124 ff 83 01 d1     sub      sp,sp,#0x60
 100003128 fd 7b 05 a9     stp      x29,x30,[sp, #local_10]
 10000312c fd 43 01 91     add      x29,sp,#0x50
 100003130 08 00 00 b0     adrp     x8,0x100004000
 100003134 08 05 40 f9     ldr      x8,[x8, #offset ->___stack_chk_guard]      = 100004018
 100003138 08 01 40 f9     ldr      x8,[x8]=>___stack_chk_guard                = ??
 10000313c a8 83 1f f8     stur     x8,[x29, #local_18]
 100003140 01 00 80 52     mov      w1,#0x0
 100003144 ff 1f 00 b9     str      wzr,[sp, #local_44]
 100003148 e0 83 00 91     add      x0,sp,#0x20
 10000314c 02 05 80 d2     mov      x2,#0x28
 100003150 37 00 00 94     bl       _memset                                    void * _memset(void * param_1, int param_
 100003154 28 00 80 52     mov      w8,#0x1
 100003158 e8 43 00 b9     str      w8,[sp, #local_20]                         #0x41 = A in hexadecimal
                                                                               local_20 is where 'A' is being stored
                                                                               w0 is address of local_20

 10000315c 28 08 80 52     mov      w8,#0x41
 100003160 e8 03 00 39     strb     w8,[sp]=>local_60
 100003164 e8 7f 40 b2     orr      x8,xzr,#0xffffffff
 100003168 e8 07 00 f9     str      x8,[sp, #local_58]
 10000316c 08 a8 81 52     mov      w8,#0x7d00
 100003170 e8 23 00 79     strh     w8,[sp, #local_50]
 100003174 00 00 00 b0     adrp     x0,0x100004000
 100003178 00 1c 40 f9     ldr      x0=>__ZNSt3__14coutE,[x0, #offset ->__ZNSt3__14coutE]   = ??
                                                                               = 100004008
 10000317c 01 00 00 90     adrp     x1,0x100003000
 100003180 21 c0 3b 91     add      x1=>s_Hello_World!_100003ef0,x1,#0xef0     = "Hello World!"
 100003184 0e 00 00 94     bl       __ZNSt3__1lsB8ue170006INS_11char_traitsIcEEEERNS_13basic_ostreamI... undefined __ZNSt3__1lsB8ue170006INS_11cha
 100003188 a9 83 5f f8     ldur     x9,[x29, #local_18]
 10000318c 08 00 00 b0     adrp     x8,0x100004000
 100003190 08 05 40 f9     ldr      x8,[x8, #offset ->___stack_chk_guard]      = 100004018
 100003194 08 01 40 f9     ldr      x8,[x8]=>___stack_chk_guard                = ??
 100003198 08 01 09 eb     subs     x8,x8,x9
 10000319c e8 17 9f 1a     cset     w8,eq
 1000031a0 68 00 00 37     tbnz     w8,#0x0,LAB_1000031ac
 1000031a4 03 00 00 14     b        LAB_1000031b0
```

Examining this more closely, you can see that there's only a few strings that give a very clear idea of what the program is designed to do. There's a random string, the title of the program, a passphrase prompt, a string indicating the expected input, a congratulations message, and an incorrect guess message. With just this information, you could easily make an educated guess on what the password is. Most likely the passphrase will be the one random string that doesn't get given easily in the executable. But for the sake of gaining understanding of the process, it's good to go over the decompiled code. Figure 7 shows the decompiled code.

The first thing to do is find the entrance into the program. Ghidra already labeled the '_main' at the top but the strings and function calls reaffirm this as the right section to be looking over. Next, thinking back to the string dump, it's likely that the password will be near whatever function takes input. Prior knowledge of programming informs the reverser that after getting the user input, the input string will need to be compared to the right answer. Thankfully Ghidra was able to identify the function 'scanf'.

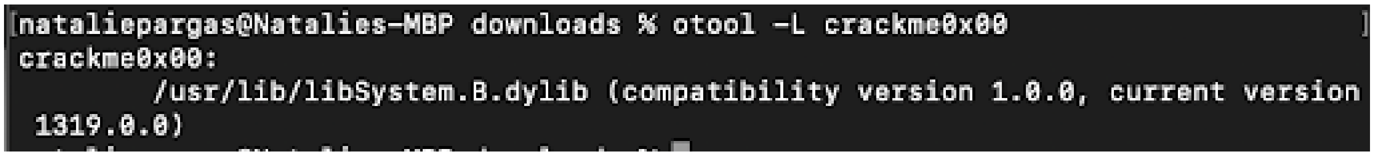
Inspecting the code section at figure 8, it shows that the variable local_88 contains the user input. This is because it loads a %99s into the RDI register to prepare for the input directly above where scanf is called. RSI

**Figure 3.4:** Crack Me Header



```
nataliepargas@Natalies-MBP downloads % otool -hv crackme0x00
crackme0x00:
Mach header
      magic  cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
MH_MAGIC_64   X86_64        ALL  0x00     EXECUTE    16       1368   NOUNDEFS DY
LDLINK TWOLEVEL PIE
```

**Figure 3.5:** Crack Me Libraries

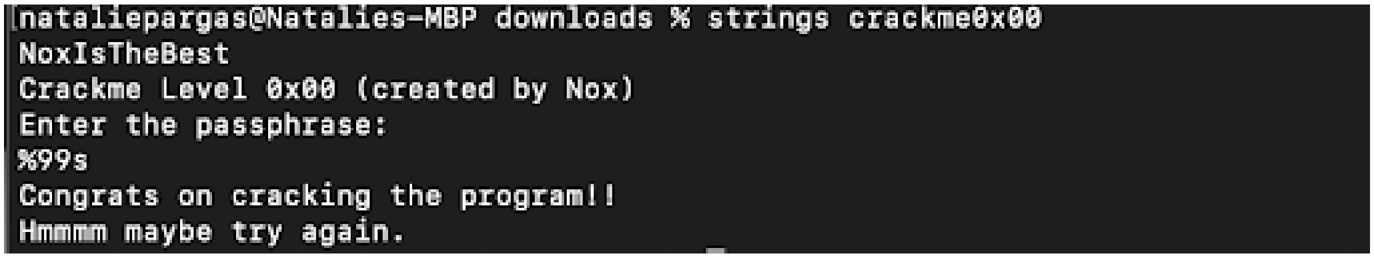


```
nataliepargas@Natalies-MBP downloads % otool -L crackme0x00
crackme0x00:
        /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version
 1319.0.0)
```

has local_1d loaded into it and that variable is immediately 'strcmp' compared to local_88. Using these clues, one could infer that local_1d contains the password. To verify, a reverser should always retrace and check.

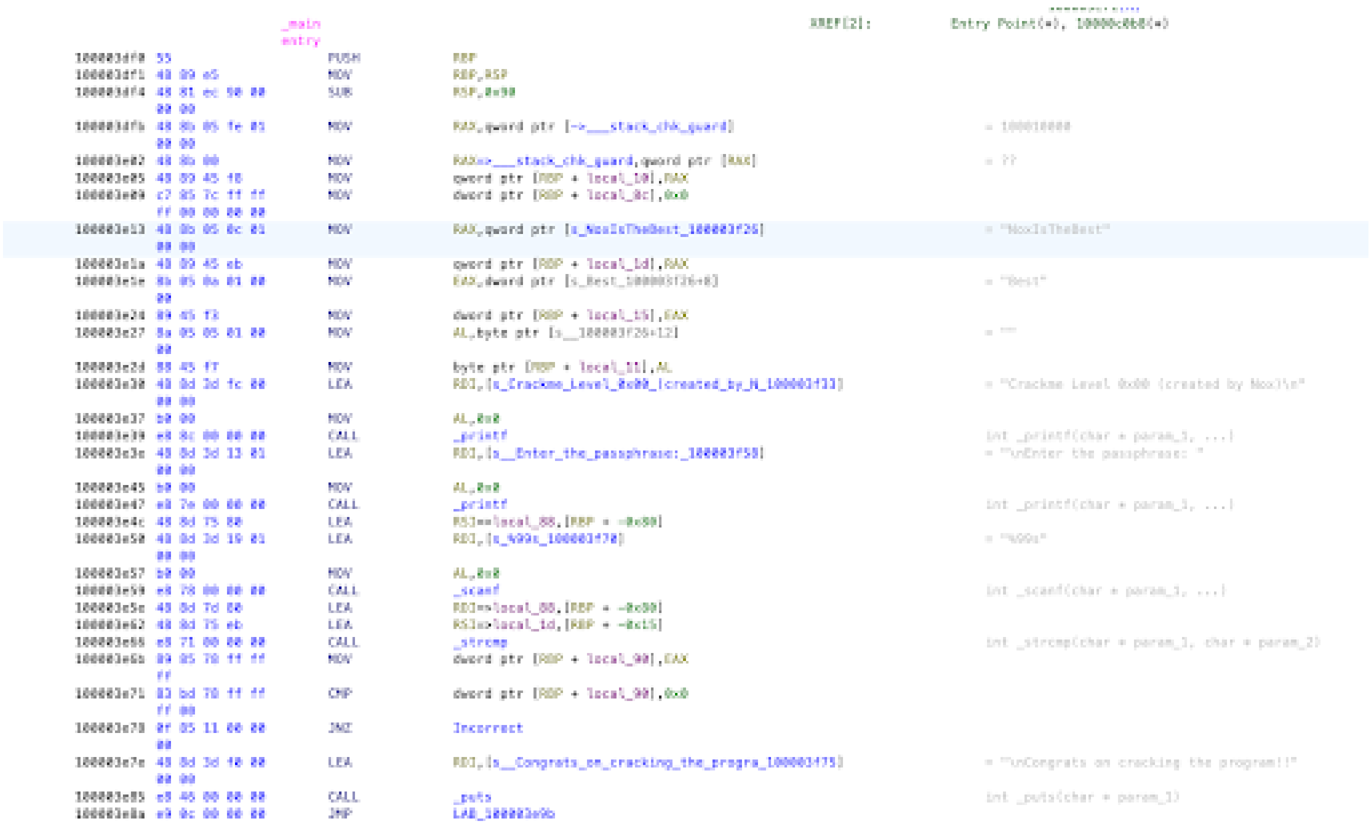
The first instance of local_1d contains the register RAX 9. RAX has the qword ptr ' s_NoxIsTheBest_100003f26' moved into it right above. s_NoxIsTheBest_100003f25 only contains the string "NoxIsTheBest". That string is probably the password, but all that's left is to check. Figure 10 shows the result.

**Figure 3.6:** Crack Me Strings



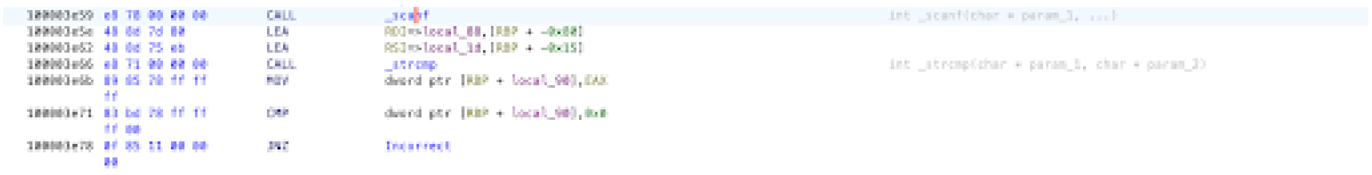
```
nataliepargas@Natalies-MBP downloads % strings crackme0x00
NoxIsTheBest
Crackme Level 0x00 (created by Nox)
Enter the passphrase:
%99s
Congrats on cracking the program!!
Hmmmm maybe try again.
```

**Figure 3.7:** Crack Me Ghidra Disassembly



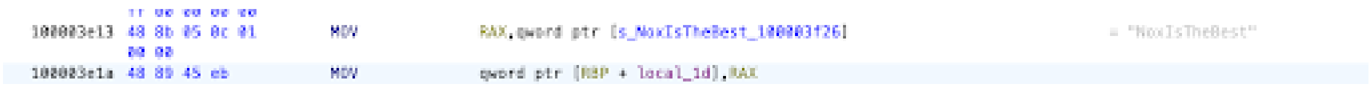
```
                              _main                                          XREF[2]:     Entry Point(*), 10000c0b8(*)
                              entry
100003df0 55                    PUSH       RBP
100003df1 48 89 e5              MOV        RBP,RSP
100003df4 48 81 ec 90 00        SUB        RSP,0x90
          00 00
100003dfb 48 8b 05 fe 01        MOV        RAX,qword ptr [->___stack_chk_guard]                 = 100010000
          00 00
100003e02 48 8b 00              MOV        RAX=>___stack_chk_guard,qword ptr [RAX]               = ??
100003e05 48 89 45 f8           MOV        qword ptr [RBP + local_10],RAX
100003e09 c7 85 7c ff ff        MOV        dword ptr [RBP + local_8c],0x0
          ff 00 00 00 00
100003e13 48 8b 05 0c 01        MOV        RAX,qword ptr [s_NoxIsTheBest_100003f26]             = "NoxIsTheBest"
          00 00
100003e1a 48 89 45 eb           MOV        qword ptr [RBP + local_1d],RAX
100003e1e 8b 05 0a 01 00        MOV        EAX,dword ptr [s_Best_100003f26+8]                   = "Best"
          00
100003e24 89 45 f3              MOV        dword ptr [RBP + local_15],EAX
100003e27 8a 05 05 01 00        MOV        AL,byte ptr [s__100003f26+12]                        = ""
          00
100003e2d 88 45 f7              MOV        byte ptr [RBP + local_11],AL
100003e30 48 8d 3d fc 00        LEA        RDI,[s_Crackme_Level_0x00_(created_by_N_100003f33]   = "Crackme Level 0x00 (created by Nox)\n"
          00 00
100003e37 b0 00                 MOV        AL,0x0
100003e39 e8 8c 00 00 00        CALL       _printf                                              int _printf(char * param_1, ...)
100003e3e 48 8d 3d 13 01        LEA        RDI,[s__Enter_the_passphrase:_100003f58]             = "\nEnter the passphrase: "
          00 00
100003e45 b0 00                 MOV        AL,0x0
100003e47 e8 7e 00 00 00        CALL       _printf                                              int _printf(char * param_1, ...)
100003e4c 48 8d 75 80           LEA        RSI=>local_88,[RBP + -0x80]
100003e50 48 8d 3d 19 01        LEA        RDI,[s_%99s_100003f70]                               = "%99s"
          00 00
100003e57 b0 00                 MOV        AL,0x0
100003e59 e8 78 00 00 00        CALL       _scanf                                               int _scanf(char * param_1, ...)
100003e5e 48 8d 7d 80           LEA        RDI=>local_88,[RBP + -0x80]
100003e62 48 8d 75 eb           LEA        RSI=>local_1d,[RBP + -0x15]
100003e66 e8 71 00 00 00        CALL       _strcmp                                              int _strcmp(char * param_1, char * param_2)
100003e6b 89 85 78 ff ff        MOV        dword ptr [RBP + local_90],EAX
          ff
100003e71 83 bd 78 ff ff        CMP        dword ptr [RBP + local_90],0x0
          ff 00
100003e78 0f 85 11 00 00        JNZ        Incorrect
          00
100003e7e 48 8d 3d 40 00        LEA        RDI,[s__Congrats_on_cracking_the_progra_100003f75]   = "\nCongrats on cracking the program!!"
          00 00
100003e85 e8 46 00 00 00        CALL       _puts                                                int _puts(char * param_1)
100003e8a e9 0c 00 00 00        JMP        LAB_100003e9b
```

**Figure 3.8:** Crack Me Code Section 1



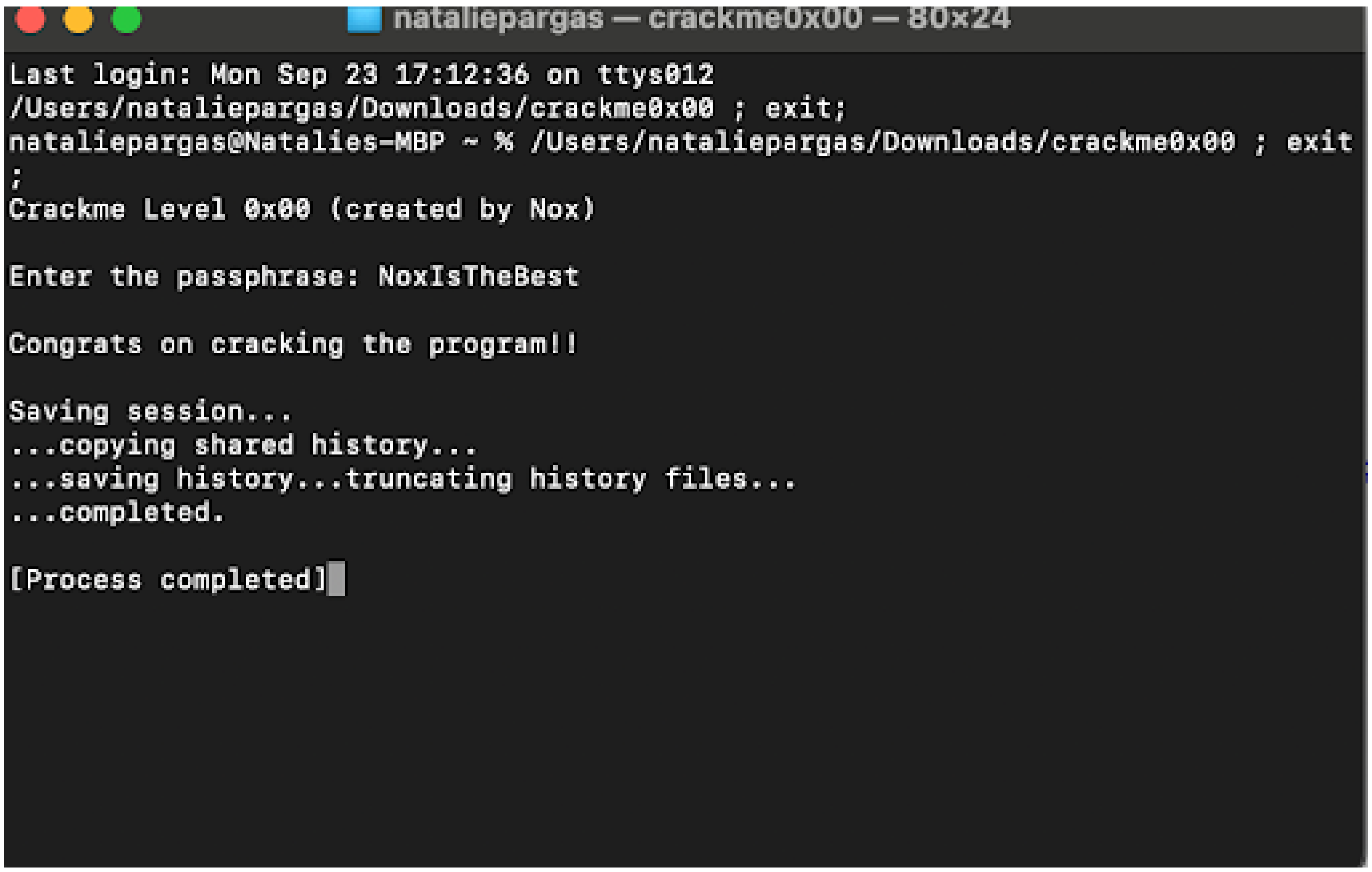
```
100003e59 e8 78 00 00 00        CALL       _scanf                                int _scanf(char * param_1, ...)
100003e5e 48 8d 7d 80           LEA        RDI=>local_88,[RBP + -0x80]
100003e62 48 8d 75 eb           LEA        RSI=>local_1d,[RBP + -0x15]
100003e66 e8 71 00 00 00        CALL       _strcmp                               int _strcmp(char * param_1, char * param_2)
100003e6b 89 85 78 ff ff        MOV        dword ptr [RBP + local_90],EAX
          ff
100003e71 83 bd 78 ff ff        CMP        dword ptr [RBP + local_90],0x0
          ff 00
100003e78 0f 85 11 00 00        JNZ        Incorrect
          00
```

**Figure 3.9:** Crack Me Code Section 2



```
          ff 00 00 00 00
100003e13 48 8b 05 0c 01        MOV        RAX,qword ptr [s_NoxIsTheBest_100003f26]      = "NoxIsTheBest"
          00 00
100003e1a 48 89 45 eb           MOV        qword ptr [RBP + local_1d],RAX
```

**Figure 3.10:** Crack Me Code Executable



```
Last login: Mon Sep 23 17:12:36 on ttys012
/Users/nataliepargas/Downloads/crackme0x00 ; exit;
nataliepargas@Natalies-MBP ~ % /Users/nataliepargas/Downloads/crackme0x00 ; exit
;
Crackme Level 0x00 (created by Nox)

Enter the passphrase: NoxIsTheBest

Congrats on cracking the program!!

Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[Process completed]
```

The attempt was a success and the crack me was successfully reverse engineered!

CHAPTER 4

# Methods

Reverse engineering is a complex process with no universal solution. It often involves extensive trial and error, making it important to adopt a strategy that encourages rapid iteration while minimizing risk. In this case, the most practical approach was to prioritize simplicity and accessibility. The Android version of the app was selected for analysis, as it uses Java and Kotlin—languages that are generally more amenable to reverse engineering. Additionally, a man-in-the-middle (MITM) technique was chosen to intercept and manipulate data exchanged between the app and the target device.

The initial step involved verifying the ability to interact with the app by modifying data in a detectable way. Once this was confirmed, the next objective was to capture the data being transmitted between the app and the makeup printing device, along with identifying the method of transmission. The final step was to replicate that communication in order to send custom data to the device in the expected format.

Java is particularly well-suited for reverse engineering due to its architecture. It is designed to be platform-independent by running on the Java Virtual Machine (JVM), which functions as an abstract computing environment with its own instruction set and memory model. The JVM interprets a binary format known as a class file, which contains bytecode (JVM instructions), a symbol table, and metadata.

The JVM is stack-oriented, with most operations involving the operand stack in the current execution frame. New frames are created whenever methods are invoked, providing a clear and modular structure. This design simplifies the decompilation and analysis process, which is critical when reconstructing application logic and behavior (Oracle JVM Specification, SE8). Figure [insert number] illustrates the flow of data from Java source code through compilation, class loading, bytecode verification, and execution by the JVM.

The use of class files is one of the key factors that makes reverse engineering Java-based applications more straightforward. Unlike many other languages that are compiled directly into low-level assembly, Java is compiled into bytecode, an intermediate representation that retains a significant amount of the original structure of the source code. This structure provides helpful context for decompilation tools. As noted

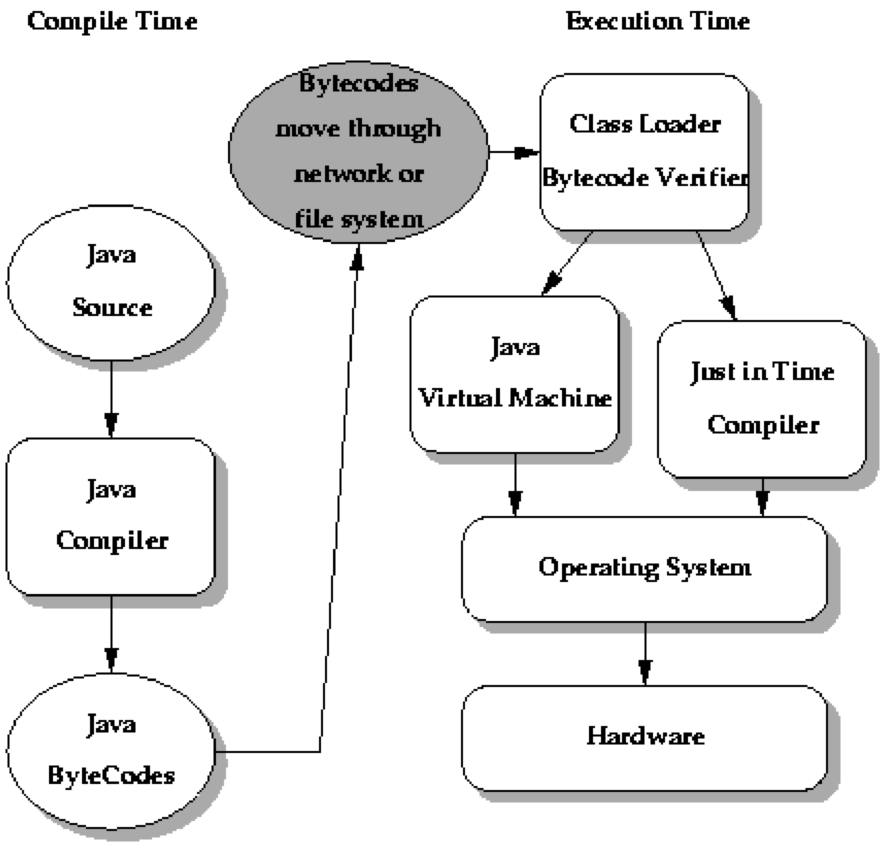

Compile Time
Execution Time
Bytecodes move through network or file system
Class Loader
Bytecode Verifier
Java Source
Java Virtual Machine
Just in Time Compiler
Java Compiler
Operating System
Java ByteCodes
Hardware

Figure 4.1

in Covert Java™: Techniques for Decompiling, Patching, and Reverse Engineering by Alex Kalinovsky, ”bytecode can be interpreted or compiled after loading, which results in a two-step transformation of the high-level programming language into the low-level machine code. It is the intermediate step that makes decompiling Java bytecode nearly flawless” (javaRE). Bytecode preserves nearly all critical information from the source file, with the exception of comments and formatting. Method names, variables, and control flow are typically intact, making it easier to reconstruct the original logic. Additionally, the stack-oriented design of the Java Virtual Machine (JVM) offers a predictable execution model that further aids analysis.

The man-in-the-middle (MITM) approach was selected for its relative simplicity compared to full decompilation or bytecode-level modification. In this context, the MITM method refers specifically to intercepting communication between the client application and the device it controls (e.g., a host device such as a Bluetooth-connected peripheral). This strategy includes techniques such as using Bluetooth sniffers to capture transmitted data and injecting code at runtime to observe or alter behavior without directly modifying the underlying APK or its bytecode.

## 4.1 A Brief Overview of Android Fundamentals

To understand the reverse engineering process of an Android application, it’s helpful to start with some foundational knowledge. At the most basic level, an Android app is made up of activities and layouts. Activities are special classes, typically written in Kotlin, that control how the app behaves and responds to user input. For example, if an app has a button, the activity defines what happens when that button is clicked.

Most Android apps have one or more screens, and the design of these screens is handled through layout files or directly in the activity code. Layout files are usually written in XML and can include elements like text, images, and buttons (Head First Android Development). Figure [insert number] shows a diagram of activities and layouts. Figure [insert numbah] illustrates how they interact during actual device use.

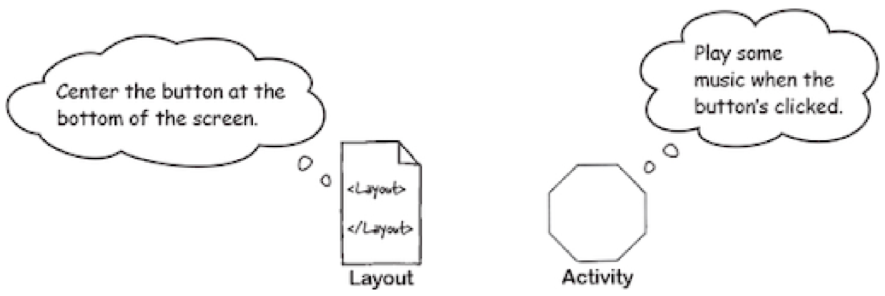

Center the button at the bottom of the screen.
<Layout>
</Layout>
Layout
Play some music when the button's clicked.
Activity

**Figure 4.2**

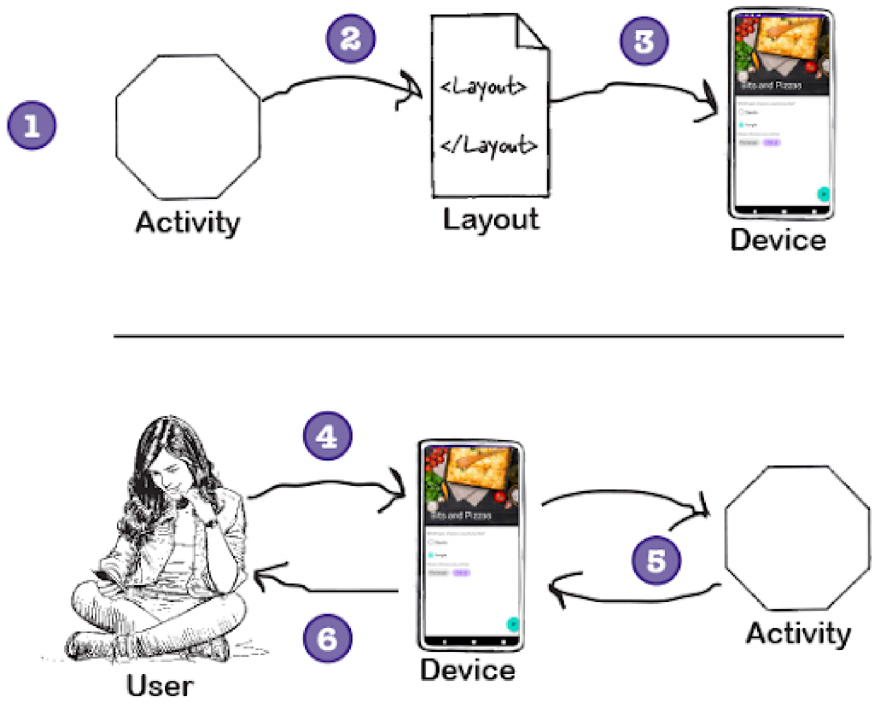


**Figure 4.3**

- Android launches the app's main activity.
- The activity instructs Android to use a specific layout.
- The layout is displayed on the device.
- The user interacts with the layout.
- The activity responds to these interactions and updates the display.
- The user is able to view the updated display seamlessly.

Apps are built using the Android Software Development Kit. The most straightforward place to do this is the Android Studio app. The Android SDK has Android source files and a compiler used to compile code into the Android format.

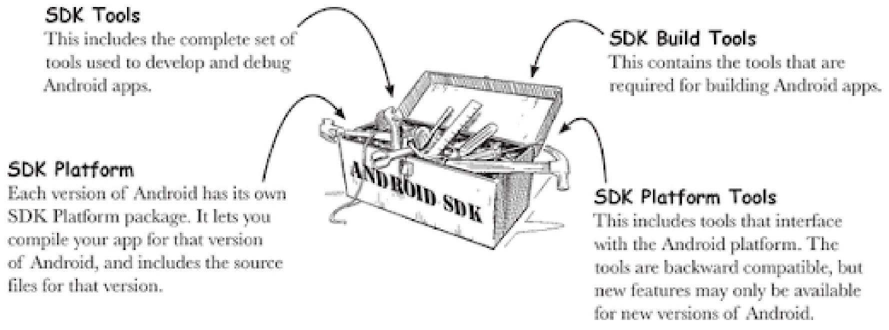

SDK Tools
This includes the complete set of tools used to develop and debug Android apps.
SDK Build Tools
This contains the tools that are required for building Android apps.
SDK Platform
Each version of Android has its own SDK Platform package. It lets you compile your app for that version of Android, and includes the source files for that version.
ANDROID SDK
SDK Platform Tools
This includes tools that interface with the Android platform. The tools are backward compatible, but new features may only be available for new versions of Android.

**Figure 4.4**

If you're familiar with Android, you've probably encountered its fun, food-based versioning system. Each Android version includes a version number and a code name. The version number refers to the specific release (like 8.0), while the code name represents a broader release range (like Oreo). Google stopped using dessert-themed code names after Android 9.

Each version of Android also corresponds to an API level, which is used by apps to ensure compatibility. For example, Android 11 corresponds to API level 30.

Android Studio projects use the Gradle build system to compile and deploy apps. Gradle-based projects follow a standard directory structure, which is shown in Figure [insert number here].

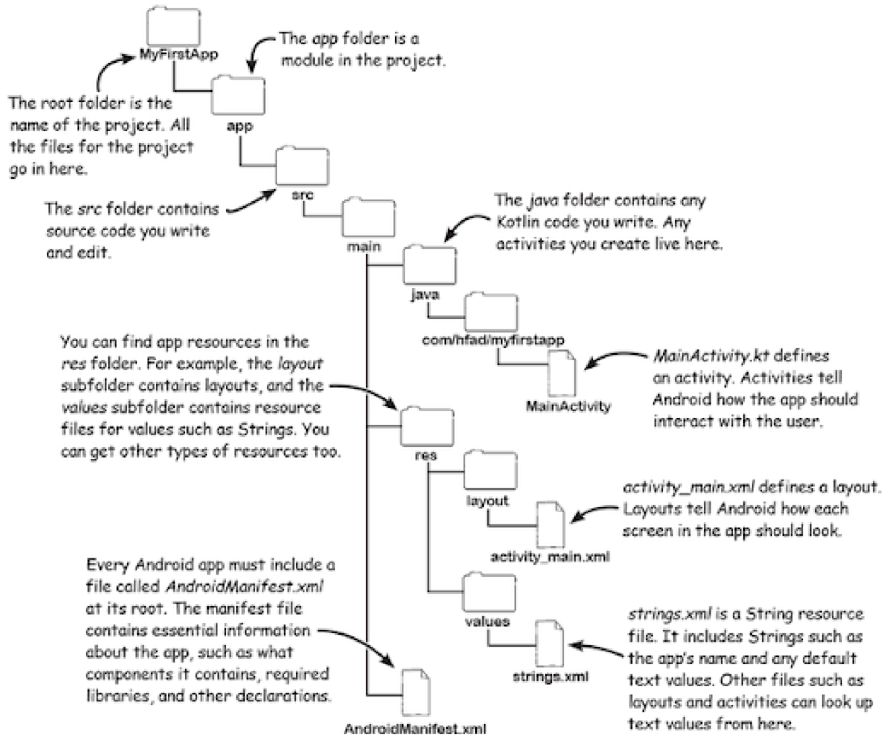

MyFirstApp
The app folder is a module in the project.
The root folder is the name of the project. All the files for the project go in here.
app
src
The src folder contains source code you write and edit.
main
The java folder contains any Kotlin code you write. Any activities you create live here.
java
com/hfad/myfirstapp
MainActivity.kt defines an activity. Activities tell Android how the app should interact with the user.
MainActivity
You can find app resources in the res folder. For example, the layout subfolder contains layouts, and the values subfolder contains resource files for values such as Strings. You can get other types of resources too.
res
layout
activity_main.xml defines a layout. Layouts tell Android how each screen in the app should look.
activity_main.xml
Every Android app must include a file called AndroidManifest.xml at its root. The manifest file contains essential information about the app, such as what components it contains, required libraries, and other declarations.
values
strings.xml
strings.xml is a String resource file. It includes Strings such as the app's name and any default text values. Other files such as layouts and activities can look up text values from here.
AndroidManifest.xml

**Figure 4.5**

A much larger version of this structure is seen in the decompilation of the Persolips app.

An Android app is made up of four main components: activities, services, broadcast receivers, and content providers. Each of these acts as an entry point into the app, either for the user or the system. Some components operate independently, while others rely on or interact with each other.

As mentioned before, Activities are the entry points that users interact with directly. Each activity represents a single screen with a user interface. In a note-taking app, for example, you might have one activity for composing a note, another for creating folders, and another for reading existing notes. While each activity functions independently, they work together to provide a cohesive user experience. Activities can even be started by other apps—if permissions allow. For instance, the Photos app might open a note composition activity so an image can be saved directly into a note.

Activities also manage key interactions between the system and the app. They help Android determine what's important to the user and ensure that relevant processes are kept running. They track backgrounded processes that might be reopened and help manage killed processes so their state can be restored if needed. Activities enable apps to work together and let the system coordinate those interactions. In code, activities are implemented as subclasses of Android's Activity class.

Services are also entry points, but they run in the background and are designed for general-purpose tasks rather than direct user interaction. They're used to keep apps running behind the scenes—for example, to perform long-running operations or interact with remote processes. A service has no user interface. It might do something like keep the flashlight on while a user switches to another app or fetch data from the internet without interrupting the user's experience. Activities and other components can start services and allow them to run independently or bind to them.

There are two types of services: started services and bound services. A started service continues to run even if the user leaves the app—like keeping the flashlight on or syncing data in the background. Services that the user is aware of (like the flashlight) are handled differently than those that operate silently (like data syncing). Bound services, on the other hand, only run when another component is actively connected to them.

Broadcast receivers allow the system to deliver messages or events to apps, even if the app isn't currently running. This makes it possible to respond to system-wide events outside of the usual user flow. For example, a scheduled notification could be delivered through a broadcast receiver without needing the app to stay open. Content providers manage access to data stored outside of the app itself, such as in a local SQLite database, on the device, or in the cloud. They make it possible for apps to query or modify this data securely. For example, a social networking app might use a content provider to access the user's device contacts when adding new friends.

When an Android app is packaged and deployed, it's bundled into a file called an APK (Android Package

Kit). This file contains everything needed to install and run the app on a device. Figure [insert number here] outlines the typical structure of an APK. Kotlin source code is compiled into bytecode, and libraries and resources are gathered. These elements are then assembled into the APK. The contents of an APK typically include:

1. Android launches the app's main activity.
2. The activity instructs Android to use a specific layout.
3. The layout is displayed on the device.
4. The user interacts with the layout.
5. The activity responds to these interactions and updates the display.
6. The user is able to view the updated display seamlessly.

All of these components work together to form the complete, installable app package. When installed, the Android system unpacks the APK and sets up everything the app needs to run on the device.

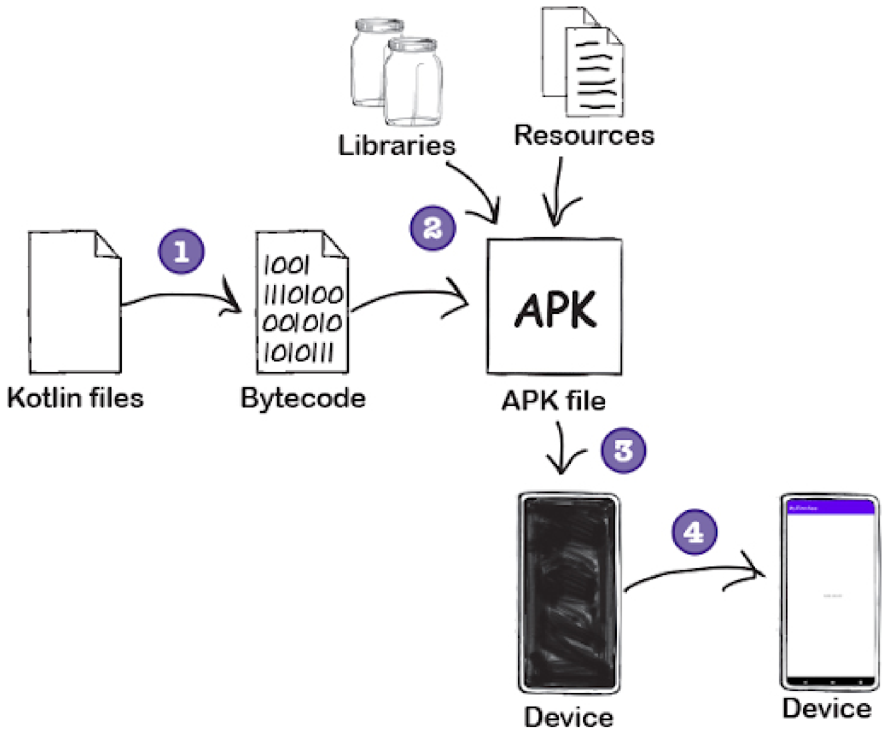


**Figure 4.6**

## 4.2 Test 1: Altering the app

To ensure that I would be able to successfully complete this project, I first needed to confirm that I could alter the app's behavior in some meaningful way. At the time, the only tools I had available were my MacBook with Android Studio installed. I was running the app on an emulated Android device and quickly found out that in order to use Frida—the dynamic instrumentation toolkit I planned to rely on—I needed to root the emulator. While I had decompiled parts of the app using Jadx, modifying the code that way wasn't practical. Decompiled code generally can't be recompiled cleanly, and reconstructing the original logic is both difficult and time-consuming. Given that Frida was going to be the foundation of my approach for real modifications later on, it made the most sense to start testing with it from the beginning. While it's true that .xapk files for this app weren't obfuscated and could have been manually edited and re-signed, that path would have diverged from the strategy I'd ultimately use. Instead, I committed to building out a working Frida workflow right away.

The first step was gathering all necessary tools. This included Frida itself, the frida-server binary compiled for android-arm64 (version 16.7.14), and Objection, a utility built on top of Frida for runtime mobile app exploration. I also installed the Android command line tools directly from Google's site, which are separate from Android Studio's full IDE.

With those in place, I turned to setting up a rooted Android emulator. I used rootAVD, a script that modifies emulators to grant root access. Since I was working on an ARM64 Mac, I selected version 12 (Android S) from rootAVD's compatibility chart. Using Android Studio, I downloaded the system image for API level 28 with an ARM64 architecture. After confirming the image was downloaded, I launched the emulator from the terminal using the emulator command. I first listed the available virtual devices, then ran the appropriate command to launch the desired AVD.

Inside the rootAVD directory, I ran the script by typing ./rootAVD.sh. I listed all the AVDs and selected the correct ramdisk.img from the path system-images/android-28/default/arm64-v8a. If everything was patched successfully, the emulator would shut down and then boot up again normally. A grey or black screen during boot was a sign that something had gone wrong. Once the emulator was running again, I verified root access using ADB. I ran adb shell, then typed su, followed by whoami. If the output returned "root," that meant the emulator was successfully rooted.

At that point, I was ready to run Objection. I launched it in a new terminal window by attaching it to the app's process using the command objection -g com.loreal.ysl.perso.lips explore. If everything was working correctly, Objection would connect to the app, allowing for runtime exploration and testing.

Next, I needed to get frida-server running on the device. I navigated to the folder where the binary was located and made it executable with chmod +x. Then I pushed it to the emulator's file system using ADB. Specifically, I placed it in /data/local/tmp using the command adb push frida-server-16.7.14-android-arm64 /data/local/tmp/frida-server. After entering the device shell and switching to root again with su, I launched the server in the background using `./frida-server &`. A successful start would return a process ID.

With the Frida server running and everything hooked up, I tested a simple proof-of-concept string replacement. I began by choosing the string I wanted to replace: "No activities yet." To work with it at the memory level, I converted the string to hexadecimal using echo -n "No activities yet" | xxd -p, which gave me the raw hex representation of the text. I then used Frida's memory search capabilities to locate that hex sequence in the app's memory.

Next, I decided on the replacement string—"Natalie activities"—and repeated the process to convert it to hex using the same xxd command. Then I retrieved the process ID (PID) of the running app using adb shell pidof com.loreal.ysl.perso.lips. With that PID, I ran the Frida script I had written to replace the original string with the new one. The command was `frida -U -p <PID> -l replace_textview.js`. Occasionally, Frida throws unhelpful syntax errors—like missing semicolons—that don't actually affect runtime behavior. Even if such an error appeared, I would go back to the app, let it load, and check if the text had been replaced. And in this case, once the relevant screen rendered, the string had indeed been swapped out, confirming that the setup was working.

## 4.3 Environment Setup

A variety of tools were used to support both static and dynamic reverse engineering throughout the project. Each tool was selected for its reliability, popularity in the reverse engineering community, and compatibility with Android-based systems.

### 4.3.1 Rooting the phone

Rooting the OnePlus 8T was a necessary step for me in the broader context of reverse engineering a mobile application. I had several reasons for wanting to root the device, all of which stemmed from the technical demands of the project I'm working on.

The first and most pressing reason is that rooting is essential for hooking into the app's internal processes. Through my early experimentation, I discovered that modifying the app's behavior would require a Man-In-The-Middle (MITM) attack, and tools commonly used for this—such as Frida—typically require root access to function properly.

Additionally, rooting a device grants access to lower-level Bluetooth packet data, which is crucial for my analysis. On a rooted phone, I can use Frida scripts to log system-level Bluetooth reads and writes that would otherwise be inaccessible. If progress slows using the current toolset, more advanced offensive techniques, such as those available through Kali NetHunter or Ubertooth, may become necessary. These tools also work more effectively on a rooted device.

Rooting a phone, however, wipes all data stored on it. For this reason, any data I intend to keep must be backed up and restored after rooting is complete. Overall, reverse engineering is an inherently unpredictable process, and having access to a wide range of tools—many of which require root access—means I can pivot more quickly if one approach hits a dead end.

Step 1: Downgrade to Android 11

The first hurdle in rooting the device was downgrading the operating system. Although I found a guide for rooting the phone on Android 14 (the version it shipped with), it required a specific boot image that I couldn't locate. My phone's variant, the OnePlus 8T KB2005 (global version), receives incremental boot image updates, which means a complete boot image for patching and reuploading isn't publicly available.

I attempted to use a boot image I found online, but it didn't match my Android version, which resulted in the phone becoming soft-bricked. To resolve this and simplify the patching process, I used the MSM Download Tool to downgrade the device to Android 11. I followed this XDA Developers guide: XDA MSM Tool Guide.

After connecting the phone to my PC via a trusted USB port, I had to troubleshoot some driver issues. Interestingly, the front USB ports on my PC didn't recognize the phone reliably, while the back ports—connected directly to the motherboard—worked better. When the device appeared as `QHUSB_BULK`, I knew the Qualcomm USB drivers were improperly installed. I downloaded the correct drivers from Microsoft's catalog: Qualcomm USB Drivers.

Next, I confirmed my build version: `kb2005_14.0.0.602(EX01)`, indicating I had the International model, which uses the KB05AA firmware. I downloaded the corresponding MSM tool and followed these steps: Launch MsmDownloadTool V4.0.exe.

At the login screen, choose "Other" and click "Next."

Select the correct target region (O2 for Global).

Press "Start" to initiate the flashing process.

Power off the device and allow it to cool.

Enter EDL (Emergency Download) mode by holding both volume buttons and plugging the phone into the computer.

Wait for the flash to complete ( 300 seconds). Step 2: Root the Phone

With the phone successfully downgraded, I proceeded to root it using another guide from XDA. I began by re-enabling Developer Mode, which involved tapping the build number eight times in the settings. Once Developer Mode was active, I enabled USB Debugging and plugged the device back into my computer. To ensure everything was working properly, I ran adb devices and fastboot devices to confirm the device was being recognized. If drivers weren't functioning correctly, I had to reinstall them.

Next, I needed to identify which slot the device was currently using. I did this with the command fastboot getvar all, which showed that the current slot was a. With that information, I moved forward with extracting the boot image from the active partition. I used a semi-broken TWRP recovery image that I booted into temporarily with fastboot boot recovery.img. This version of TWRP doesn't include a GUI but provides ADB shell access, which was all I needed. Inside the shell, I used the dd command to copy the boot_a partition to the SD card. After exiting the shell, I pulled the boot_a.img file from the device using ADB and then rebooted the phone normally.

Once I had the boot_a.img file on my computer, I transferred it to the phone's internal storage, placing it in an accessible location like the Downloads folder. I then installed the latest version of the Magisk Canary APK on the phone. Opening the app, I selected the install option and chose "Select and Patch a File," pointing Magisk to the boot_a.img I had extracted earlier. This generated a patched image file named magisk _patched_a.img, which I copied back to my computer.

To proceed with rooting, I rebooted the phone into fastboot mode using adb reboot bootloader. Instead of flashing the patched image, I temporarily booted into it using the command fastboot boot magisk _patched_a.img. This gave me temporary root access. With the device booted into this patched environment, I opened Magisk again and selected the install option, this time choosing "Direct Install (Recommended)" to apply root access permanently to the internal boot image. After the installation completed, I rebooted the phone and used a root checker app to verify that root access was working. Everything checked out, and the device was successfully rooted.

### 4.3.2 Virtual Machine

To ensure that the reverse engineering process does not compromise the host system, it is advisable to perform the work within a virtual machine (VM). This not only adds a layer of protection against potentially harmful software, but also provides the flexibility to use tools that may not be compatible with the native operating system. In this case, setting up a virtual machine was necessary for both reasons. Since the focus was on reverse engineering the Android version of the app, many essential tools—such as those for extracting APKs or interfacing with the device—required a Windows environment. Rooting the phone, for example, involved installing legacy drivers from outdated and potentially unsafe sources, increasing the risk of introducing malware. Using a VM offered a controlled environment where such risks could be isolated.

Because the target phone needed to be connected externally via USB, a virtual machine with USB passthrough support was required. Initially, a macOS system was the only available host, so Parallels Desktop was selected. USB passthrough functionality had recently been released in version 20.3.0 (May 7, 2025) (source), just shortly before the virtual machine was set up. However, issues arose when attempting to get the device recognized within the VM. Given that the phone was an older model that also had trouble with driver installation on native Windows machines, the problem was likely not due to the VM itself. It is also worth noting that Parallels requires a paid license, which should be considered when evaluating virtualization options.

Access to a native Windows machine later resolved many of the earlier issues. Windows-based VMs tend to perform more reliably on Windows hosts, and the improvement in stability was noticeable. VMware Workstation was chosen for this setup. It supports USB passthrough effectively and is free to use with a Broadcom account.

#### 4.3.2.1 JADX

The primary decompiler used for this project was JADX, a widely adopted tool specifically designed for reverse engineering Java-based Android applications. JADX takes an APK or XAPK file and converts the included .dex (Dalvik Executable) files—Android's equivalent of Java .class files—into readable Java source code by generating .jar files. While the decompiled Java output is generally highly readable, it is typically not suitable for direct recompilation or execution without additional modification. Nonetheless, it serves as a valuable resource for understanding app structure, logic, and flow.

#### 4.3.2.2 Frida

Frida is a dynamic instrumentation toolkit used to inject JavaScript snippets or custom libraries into native processes. It leverages QuickJS to run scripts inside a target process, providing full memory access, function hooking, and the ability to call native functions at runtime. Frida establishes a bi-directional communication channel between the injected script and the host environment, enabling real-time monitoring and manipulation of application behavior. In this project, Frida was primarily used to extract runtime information from the app and to override or modify methods on the fly.

#### 4.3.2.3 Wireshark

Wireshark is a widely used network protocol analyzer, typically applied to internet and server traffic. In the context of this project, it was used to analyze Bluetooth communication. Although Wireshark does not natively support capturing Bluetooth packets without additional hardware, it becomes highly effective when paired with a Bluetooth sniffing device. Additionally, it can display Bluetooth logs exported from Android devices with Bluetooth HCI snoop logging enabled. This made it an essential tool for inspecting communication between the app and the connected device.

#### 4.3.2.4 JBSE (symbolic Java VM)

#### 4.3.2.5 UML Diagram Creator (Ask Dr.Guarnera)

#### 4.3.2.6 Emulator ?

## 4.4 Scripting

Frida accomadates the use of scripts to perform specific functions in conjunction with their application. It is a very useful tool in the reverse engineering process to be able to target and extract specific information. The first script created was one to enumerate the contents of the BLE send function. This function is how any and all BLE communications get broadcast through the application so it holds a wealth of information in terms of figuring out how the device is expecting to receive data. While it is hypothetically possible to hook into the Android library's Bluetooth send function, it is much more difficult to do with less payoff than hooking into the native send function. The send() method was discovered by reading through the decompiled code in jadx. Its existence was verified by listing the methods in the BLEManager class using Frida.

```
Java.perform(function () {
    var BleManager = Java.use("com.vinsol.loreal.PersoLips.utils.BleManager");
```

Frida has some of its own terminology. `Java.perform()` is one of the most common Frida methods because it alerts Frida where to hook into the application. Here, Frida is being instructed to hook into the `BleManager` class.

```
    var originalSend = BleManager.send.overload('[B', 'int');
```

This line saves the original `send(byte[], int)` function so it can be called as necessary. It will need to be called as usual at the end of enumeration so that there isn't an interruption in the methods.

```
    function enumerateObject(obj) {
```

```
                try {
                        var objClass = obj.getClass();
                        console.log("Object class: " + objClass.getName());

                        var fields = objClass.getDeclaredFields();
                        for (var i = 0; i < fields.length; i++) {
                                var field = fields[i];
                                field.setAccessible(true);
                                try {
                                        var name = field.getName();
                                        var value = field.get(obj);
                                        console.log("  Name: " + name + " = " + va
                                } catch (err) {
                                        console.log(" Could not access field: " +
                                }
                        }
                } catch (err) {
                        console.log("Error during enumeration: " + err);
                }
        }
```

This is the definition for the function to enumerate the contents of each object in the `send()` function. The `try{}`/`catch{}` allows the function to execute safely without crashing. The `getClass` Java method gets the class of the object at runtime. It then loops through each field and sets them as accessible. For each field in the loop, it logs the name and object value.

```
        BleManager.send.overload('[B', 'int').implementation = function (bArr, i)
                console.log("send() called with device ID: " + i);
```

Here is where the `send` function actually gets overridden. It starts by logging that the `send` function was called.

```
                var bytes = Java.array('byte', bArr);
                var hex = Array.from(bytes).map(function (b) {
                        return ('0' + (b & 0xFF).toString(16)).slice(-2);
```

```
}).join(' ');
console.log("Payload (hex): " + hex);
```

This code segment is where the byte array output of `send` gets converted into hex to make it easier to read. It starts by turning the Java byte array `bArr` into a format that can be read by Frida. The bytes are then ANDed with `0xFF` to convert signed bytes to unsigned. It then converts the result into base 16 (hex). Finally, `slice(-2)` ensures each value is represented by two characters (zero-padded if needed).

```
console.log("Enumerating fields of BleManager instance...");
enumerateObject(this);
```

This logs and inspects the current `BleManager` instance — the object calling `send()`. It gives insight into the app's internal state at the moment of the call.

```
return originalSend.call(this, bArr, i);
```

Finally, the original `send()` function is called to allow normal app behavior to continue uninterrupted.

## 4.5 App Interoperability