TECHNISCHE UNIVERSITÄT DARMSTADT

INTERNALBLUE – A BLUETOOTH EXPERIMENTATION FRAMEWORK BASED ON MOBILE DEVICE REVERSE ENGINEERING

DENNIS MANTZ

Master Thesis

July 19, 2018

Secure Mobile Networking Lab
Department of Computer Science

SECURE MOBILE NETWORKING

InternalBlue –
A Bluetooth Experimentation Framework Based on Mobile Device Reverse Engineering
Master Thesis
SEEMOO-MSC-0127

Submitted by Dennis Mantz
Date of submission: July 19, 2018

Advisor: Prof. Dr.-Ing. Matthias Hollick
Supervisor: Jiska Classen and Matthias Schulz

Technische Universität Darmstadt
Department of Computer Science
Secure Mobile Networking Lab

ABSTRACT

Bluetooth has long been one of the most established technologies for digital wireless data transmission. With the advent of smart *Wearables* and the *Internet of Things (IoT)*, Bluetooth has once more vitally gained in importance. To live up to this condition, it is fundamental to advance security research in this area. This implies that there exist openly available tools and experimental platforms, which allow research on hardware-near protocol levels.

For this reason, a new open source platform for the development of research tools is implemented in this thesis. This is done by modifying the firmware inside the *Broadcom* Bluetooth Controller which is embedded in the *Nexus 5* and make it accessible through an interactive Python framework. The framework named *InternalBlue* is an openly available and flexible tool which supports security researchers in analyzing and experimenting with protocol layers that have been difficult to access before. In addition, this work offers deep insight into the internal architecture of a widespread commercial family of Bluetooth controllers, which is used in smartphones and *IoT* platforms.

ZUSAMMENFASSUNG

Bluetooth zählt seit langem zu den etabliertesten Technologien für digitale drahtlose Datenübertragung. Mit dem Aufkommen von intelligenten *Wearables* und dem *Internet der Dinge (IoT)* hat Bluetooth nochmals entscheidend an Bedeutung gewonnen. Um diesem Zustand gerecht zu werden, ist das Vorantreiben der Sicherheitsforschung in diesem Bereich unabdingbar. Dafür werden frei verfügbare Werkzeuge und Experimentierplattformen benötigt, welche eine einfache Untersuchung der hardwarenahen Protokollebenen ermöglichen.

Aus diesem Grund wird in dieser Thesis eine neue Open-Source-Plattform für die Entwicklung von Forschungswerkzeugen implementiert. Hierzu wird die Firmware des im *Nexus 5* verbauten *Broadcom* Bluetoothchips modifiziert und durch ein interaktives Python-Framework zugänglich gemacht. Das auf den Namen *InternalBlue* getaufte Framework ist ein frei verfügbares und flexibles Werkzeug, welches Sicherheitsforscher dabei unterstützt, bisher schwer zugängliche Protokollebenen zu analysieren und mit diesen zu experimentieren. Zusätzlich bietet diese Arbeit einen tiefen Einblick in die interne Architektur einer kommerziellen Familie von Bluetoothcontrollern, welche durch den Einsatz in Smartphones und *IoT*-Plattformen weite Verbreitung gefunden hat.

Without exception,
every single thing I did in my career that I was proud of,
was something that I accomplished with the help of open source software.

— [36]Michael Ossmann

ACKNOWLEDGMENTS

I would like to express my deepest gratitude
to my family and my girlfriend Lisa for
supporting me in all the years of my studies
and also while writing this thesis.

Special thanks for giving helpful advice
while writing this thesis goes to
Prof. Dr.-Ing. Matthias Hollick,
Jiska Classen and Matthias Schulz.

Furthermore, I especially thank
Kevin Schaller, Niki Vonderwell and
Daniel Hoff for proofreading my thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ACL	Asynchronous Connection-Less
ADB	Android Debug Bridge
AES	Advanced Encryption Standard
AFH	Adaptive Frequency Hopping
AGC	Automatic Gain Control
AHB	Advanced High-performance Bus
AOSP	Android Open Source Project
API	Application Programming Interface
AWS	Amazon Web Services
BLE	Bluetooth Low Energy
BR	Basic Rate
CCM	Counter mode with CBC-MAC
CLI	Command Line Interface
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
ECDH	Elliptic Curve Diffie-Hellman
EDR	Enhanced Data Rate
FIFO	First-In-First-Out
FPB	Flash Patch and Breakpoint
HCI	Host Controller Interface
HEC	Header-Error-Check
IO	Input-Output
IoT	Internet of Things
IP	Internet Protocol
ISM	Industrial, Scientific and Medical
IVT	Interrupt Vector Table

JW	Just-Works
L2CAP	Logical Link Control and Adaption Protocol
LAP	Lower Address Part
LE	Low Energy
LLID	Logical Link ID
LM	Link Manager
LMP	Link Manager Protocol
LNA	Low Noise Amplifier
LSB	Least Significant Bit
MIC	Message Integrity Check
MITM	Man-In-The-Middle
NC	Numeric-Comparison
NIST	National Institute for Standards and Technology
NMI	Non-maskable Interrupt
NVIC	Nested Vectored Interrupt Controller
OOB	Out-Of-Band
PAN	Personal Area Network
PCM	Pulse-code Modulation
PDU	Program Data Unit
QoS	Quality of Service
RAM	Random Access Memory
RF	Radio Frequency
RFCOMM	Radio Frequency Communications
RNG	Random Number Generator
ROM	Read Only Memory
RTOS	Real-Time Operating System
SCO	Synchronous Connection-Orientated
SDIO	Secure Digital Input Output
SDP	Service Discovery Protocol
SDR	Software-Defined Radio

SIG	Special Interest Group
SP	Stack Pointer
SSP	Secure Simple Pairing
TCP	Transmission Control Protocol
TDD	Time-Division Duplex
TID	Transaction ID
TLS	Transport Layer Security
TLV	Type-Length-Value
TRM	Technical Reference Manual
UAP	Upper Address Part
UART	Universal Asynchronous Receiver-Transmitter
UI	User Interface
USB	Universal Serial Bus
WEP	Wired Equivalent Privacy

1 INTRODUCTION

The first chapter of this thesis motivates the topic, gives a brief overview of its contributions and finally outlines the following chapters.

1.1 MOTIVATION

Bluetooth, *the* standard for wireless short range communication, has been around for almost 25 years since it was invented in 1994 by Ericsson. In the early days its main application were wireless headphones, hands-free speakerphones and the replacement for infra-red data links between devices. Today, Bluetooth experiences a comeback with the emerge of Wearables, Smart Devices and the Internet of Things (IoT). The latest version of the specification, Bluetooth 5, introduces new interesting features such as mesh networking [16] and indicates that Bluetooth will play an important role in the future of wireless communications.

Compared to its big brother WiFi (IEEE 802.11), it has some special characteristics, such as battery friendliness and simple device-to-device connection setup. Both standards complement each other. However, while WiFi was subject of intense security research in the past decade, Bluetooth was often not in focus.

Many security weaknesses in WiFi could be found and fixed thanks to openly available research tools.

To some extent this is due to the availability of powerful, open-source tools which allow easy experiments on raw WiFi frames with cheap, off-the-shelf hardware. When the first patches for WiFi drivers enabled the so called *monitor mode* and frame injection capabilities it did not take long for researchers to implement practical attacks on low-level parts of the WiFi stack and the now deprecated Wired Equivalent Privacy (WEP) standard [10, 26]. Only recently [4, 7], the firmware running on WiFi cards has shown to contain severe vulnerabilities which can lead to over-the-air compromisation of devices. *Blueborne* [50], a collection of weaknesses that have been uncovered in many of the major Bluetooth stacks, has raised the awareness for Bluetooth security. However, *Blueborne* targets host-side Bluetooth drivers in opposition to the lower layers of the protocol which are handled in firmware and are still difficult to audit.

The Bluetooth physical layer is complex to implement and therefore available research tools are either expensive or do not work well in every situation.

Unfortunately, compared to WiFi there is no similarly easy way to monitor and inject raw Bluetooth management frames. On the one hand, professional equipment targeted at hardware developers exists, but is very expensive. An example is the *Bluetooth Explorer BEX400* from *Ellisys*, which costs well above 10.000$ [13]. On the other hand, cheaper platforms such as the *Ubertooth* (120$ [52]) or Software-

Defined Radio (SDR) solutions still struggle with fundamental problems such as the frequency-hopping characteristics of the Bluetooth physical channel.

Firmware modifications for commercial Bluetooth controllers can be the basis for new research tools.

Another way of building a capable research platform is to directly instrument the firmware of proprietary wireless chipsets. The Nexmon project [46] did exactly this for the *Broadcom* WiFi chipset in the *Nexus 5* smartphone. Apart from creating a powerful and mobile research platform for modern WiFi communication, the project also enables in-depth analysis of the chip internals and helps security researchers to uncover vulnerabilities in the firmware.

This thesis lays the foundation for a similar platform focused on Bluetooth security. It targets the firmware of the *Broadcom* Bluetooth chipset named *BCM4339* which resides on the very same chip inside the *Nexus 5*.

1.2 CONTRIBUTIONS

The main focus of this work lies on reverse engineering the essential parts of the *BCM4339* firmware. The results provide the groundwork for building a research platform on top of the Bluetooth chip. The presented firmware internals include but are not restricted to:

- the controller side of the Host Controller Interface (HCI),
- the Link Manager Protocol (LMP) handler,
- the firmware update and patching mechanism, and
- the internal memory, resource and scheduling management of the firmware.

Furthermore, the thesis includes an in-depth analysis of various security critical modules such as the Random Number Generator (RNG) which is, amongst others, used for the derivation of encryption keys. It also contributes by extending the *Nexmon* [46] framework so that it is able to produce patches for the *BCM4339*.

Eventually, the final outcome on top of the reverse engineering results, is the research and analysis framework *InternalBlue* which enables direct interaction with *BCM4339* firmware internals at runtime. The framework comes in the form of a flexible Python library which acts as an interface to the firmware over the Android Debug Bridge (ADB). It is supplemented by an interactive front-end with an extendable set of useful commands which support the analysis of the firmware and low-level Bluetooth activities. A non-exhaustive list of features follows:

InternalBlue *is a Python framework which enables direct interaction with the Bluetooth controller firmware.*

- read arbitrary memory regions (ROM & RAM),
- write to RAM,

- apply ephemeral patches to the main ROM,
- run arbitrary code in the context of the running firmware,
- send arbitrary HCI commands to the chip,
- enable monitor mode for link layer Bluetooth management packets (LMP), and
- enable injection of arbitrary LMP packets in an active Bluetooth connection.

The last two points on this list stand out as they turn an off-the-shelf and low-cost smartphone into an LMP monitor and injection device. To the knowledge of the author there exists no openly available solution to monitor and craft LMP messages in the context of Bluetooth connections until now. Such a tool is especially useful for researching and testing other Bluetooth devices on the Link Manager (LM) layer.

InternalBlue *can monitor and inject LMP packets.*

1.3 OUTLINE

The thesis organizes the accomplished work in a contextually coherent structure rather than reflecting the actual timeline of the work as it has been done. First, Chapter 2 introduces the Bluetooth standard with a special focus on the lower protocol layers which are implemented in firmware and not observable from a host machine. After that, Chapter 3 gives an overview of related work about Bluetooth security research and firmware reverse engineering.

All information about the internal workings of the *BCM4339* that have been gathered during the reverse engineering process are collected in Chapter 4. The information is then used in Chapter 5 to design and implement a Bluetooth analysis and debugging framework based on the *Nexus 5* smartphone. Chapter 6 will evaluate the framework and the applications which have been implemented on top of it.

Finally, Chapter 7 discusses the results, gives an outlook about other possible applications of the framework and Chapter 8 concludes the thesis.

2 BACKGROUND

The Bluetooth standard is developed by the Bluetooth Special Interest Group (SIG) which is a non-profit organization whose members are companies that incorporate Bluetooth into their products. At the time this thesis was written, the current standard is Bluetooth Core 5. It is available free of charge and can be downloaded from the Bluetooth SIG website [15].

Since version 4.0, Bluetooth systems come in two different variants:

- Bluetooth Basic Rate (BR) & Enhanced Data Rate (EDR) (also referred to as classic Bluetooth).
- Bluetooth Low Energy (LE) (newly introduced in version 4.0).

Although both variants share many commonalities, their protocol stack differs significantly. Depending on its use cases a device can implement either of the two variants. It is also possible to have both stacks implemented together in a combined Bluetooth controller. For the sake of simplicity the focus of this thesis will be on the classic Bluetooth protocol stack. The peculiarities of Bluetooth LE are therefore omitted in the following sections.

Bluetooth LE differs from classical Bluetooth and is not covered in this thesis.

2.1 OVERVIEW

A Bluetooth network is called a piconet. It always consists of one master and up to seven slave devices. It is possible for a device to only support the operation as a slave node. Communication is only possible between the master and a slave but not between the slaves themselves. A device may be part of multiple piconets at the same time but it can only be the master in one of them. All devices that are part of the same piconet share a physical channel which is defined by a common, synchronized clock and a frequency hopping pattern.

In a Bluetooth piconet one master device communicates with multiple slave devices. They use frequency hopping to avoid interference.

Bluetooth operates in the 2.4 GHz Industrial, Scientific and Medical (ISM) band which it has to share with other technologies such as WiFi or ZigBee. The pseudo-random frequency hopping pattern spreads the signal over all 79 channels, each 1 MHz wide. Certain channels may also be excluded from the hopping sequence on a per-slave basis to cope with interference from other devices and technologies. This is called Adaptive Frequency Hopping (AFH). As all devices in the piconet share the same physical channel, it is divided in a Time-Division Duplex (TDD) scheme in which the smallest entity is called a slot.

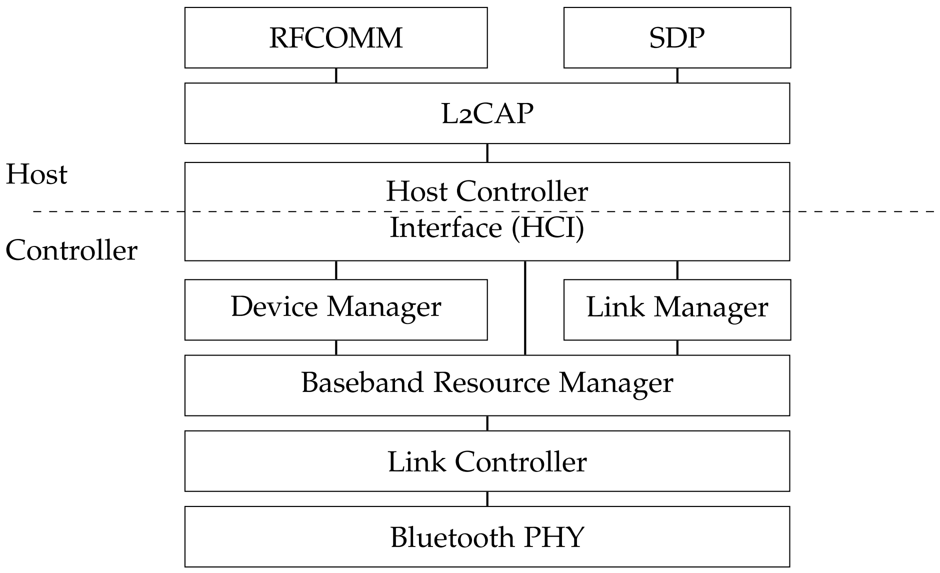


Figure 1: Architecture of the Bluetooth Protocol Stack.

2.2 BLUETOOTH ARCHITECTURE

The Bluetooth stack is split between the host driver and the Bluetooth controller. HCI is the interface between both parts.

The Bluetooth architecture is defined in the architecture specification (Volume 1, Part A) [15]. It separates duties onto two entities, the host and the Bluetooth controller. They communicate via the Host Controller Interface (HCI) on top of an underlying transport channel which can be realized with different technologies such as Universal Asynchronous Receiver-Transmitter (UART) or Universal Serial Bus (USB). Figure 1 visualizes the layered protocol stack of the Bluetooth architecture.

On the host, the Logical Link Control and Adaption Protocol (L2-CAP) is responsible for providing multiplexed, logical channels to the upper protocol layers. It implements segmentation, flow control and retransmission of Program Data Units (PDUs). These services are used by higher level protocols such as the Service Discovery Protocol (SDP) and the Radio Frequency Communications (RFCOMM) protocol. The former is responsible for the enumeration of capabilities of the remote device. The latter emulates classical RS-232 compatible serial data streams and is the basis for many protocols which need reliable data transmission like with the Transmission Control Protocol (TCP).

On the controller side, HCI directly interfaces with three manager entities:

DEVICE MANAGER This entity is responsible for general behavior of the device. Many local device related HCI commands are handled by the Device Manager, including management of encryption keys, device name and device inquiry.

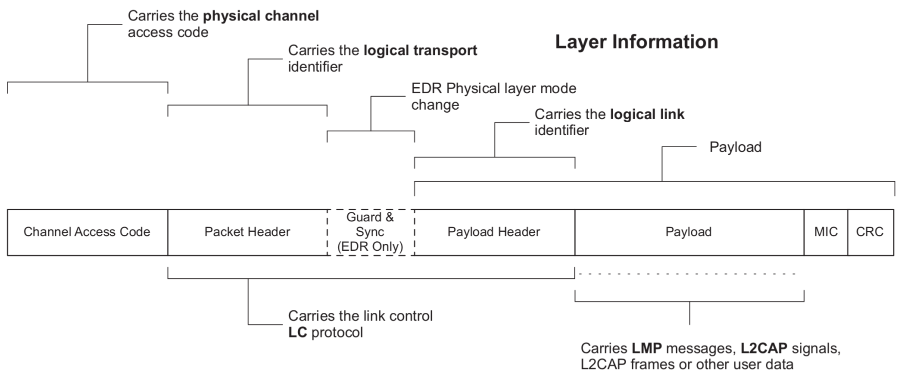

Carries the **physical channel** access code
Carries the **logical transport** identifier
Layer Information
EDR Physical layer mode change
Carries the **logical link** identifier
Payload
Channel Access Code
Packet Header
Guard & Sync (EDR Only)
Payload Header
Payload
MIC
CRC
Carries the link control **LC** protocol
Carries **LMP** messages, **L2CAP** signals, L2CAP frames or other user data

Figure 2: General Structure of Bluetooth BR and EDR Packets. [15]

LINK MANAGER The Link Manager communicates with its remote counterpart through the Link Manager Protocol (LMP) and controls the logical links between devices. It is therefore responsible for tasks including device paging, pairing, remote name resolution and clock synchronisation.

BASEBAND RESOURCE MANAGER This entity is a scheduler for the slots in the physical channel of the piconet. It handles requests from the Device and Link Managers as well as from the host system and commits to deliver the negotiated Quality of Service (QoS).

Below those layers is the Link Controller which is responsible for de- and encoding packets from the physical channel data stream. It is tightly coupled with the Baseband Resource Manager and also manages flow control and link layer retransmissions.

2.3 BLUETOOTH PACKET FORMAT

Figure 2 shows the encapsulation of the previously described protocol layers. The header fields are defined in the core system specification (Volume 2, Part B: Baseband Specification) [15].

The access code is always at the beginning of a packet and defined by the physical channel. Depending on the type of the physical channel the code is constructed differently. For normal communication in an established piconet, it is derived from the Bluetooth address of the master. For paging and other special purpose channels the access code may also be derived from the address of the paged device or from predefined constants. Beside the channel classification, the access code is also used at the receiver side to detect the beginning of a packet.

The Link Control header, as shown in Figure 3, contains a 3-bit logical transport address (LT_ADDR) which identifies the destination device inside the piconet. Due to the size of the LT_ADDR, a master

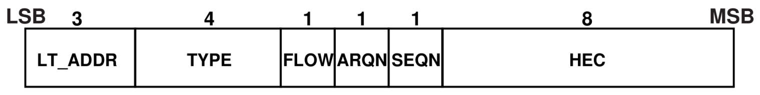


Figure 3: Structure of the Link Control Header. [15]

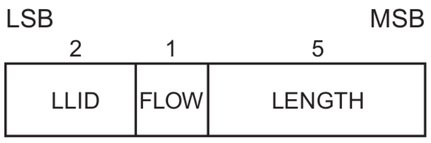


Figure 4: Structure of the Payload Header. [15]

may only be connected with up to seven slaves at a time. The second field in the Link Control header is the type field which defines the usage of the packet as well as the number of time slots it occupies. Finally the header also contains flags for flow control and a Header-Error-Check (HEC) field.

The payload field depends on the packet type. In case of an asynchronous data packet it contains a short payload header that consists of a 2-bit Logical Link ID (LLID), a flow control flag and a length field (Figure 4). The LLID indicates whether the payload is a L2CAP or a LMP packet. For most packet types the payload is followed by a Cyclic Redundancy Check (CRC) and a Message Integrity Check (MIC) if encryption with Advanced Encryption Standard (AES) in Counter mode with CBC-MAC (CCM) is enabled.

2.4 INQUIRY, PAGING AND PAIRING

Inquiry scans discover devices in proximity. Paging a device is necessary for a connection establishment.

In order to find surrounding devices, a Bluetooth controller can do a so-called inquiry scan. This is done by sending multiple inquiry requests on certain specified channels. Other devices which are currently set to be discoverable have to respond to the request and thereby provide the inquirer with its address information. A device that has the necessary address information can start a connection by paging the other device. Connectable devices scan for incoming paging requests from other devices and answer them. A device which is set to be undiscoverable but connectable does not answer to inquiry scan packets but still answers paging requests and accepts connections as long as the connecting device knows the required parts of its address.

Pairing establishes a trust relationship between two devices and exchanges key material.

At the first time two devices connect with each other they go through one of the specified pairing procedures to exchange key material for authentication and encryption:

- Legacy Pairing, or
- Secure Simple Pairing (SSP) (since Bluetooth 2.1).

The legacy pairing mechanism is deprecated and should not be used anymore. It was replaced by the SSP protocol which consists of five phases:

1. public key exchange,
2. authentication stage 1,
3. authentication stage 2,
4. link key calculation, and
5. LMP authentication and encryption.

In the first phase both devices send their public Elliptic Curve Diffie-Hellman (ECDH) key (either P-192 or P-256) which gets authenticated in the second and third stages. Depending on the Input-Output (IO) capabilities of the devices this can be done using three different authentication protocols:

- the Out-Of-Band (OOB) protocol,
- the Numeric-Comparison (NC) protocol, or
- the Just-Works (JW) protocol.

Eventually, both devices are able to derive the link key from the exchanged information and start an authenticated and encrypted connection. For the encryption AES-CCM is used.

2.5 FIRMWARE ANALYSIS

Recovering information on the internal workings of a device can be done through reverse engineering its firmware which is the machine code running on the processor. A disassembler such as *IDA Pro* [30] or *radare2* [59] is indispensable to accomplish this task. At the beginning of the reverse engineering process certain techniques help to quickly discover interesting parts of the firmware.

FREQUENTLY USED FUNCTIONS Advanced disassemblers are able to find cross references inside the firmware which means that for a specific function or data object it is aware of all places where it gets referenced. Therefore it is possible to derive a ranking of all functions based on how frequently they are being used. Starting at the top of this list often reveals basic functions such as `memcpy`, `malloc` and `free`.

MAGIC VALUES Many modules, especially cryptographic functions use very unique constant values in their algorithms. Often times these values stick out during the reverse engineering process and with the help of an online search engine they can be identified. When it is known beforehand that a firmware uses certain features, the technique can also be used the other way around: Searching the firmware for the byte sequence of the P-256 elliptic curve can reveal the code responsible for cryptographic functions.

RECOVER DATA STRUCTURES It is important to find and reverse engineer important data structures used by the firmware early in the reverse engineering process. Not only do they help with tracing data flows between functions but without knowing the data structures in use, most functions are simply accessing memory at various offsets with no semantic meaning. For reverse engineering data structures it is very helpful to have access to a Random Access Memory (RAM) dump which contains different instances of the structures.

3 RELATED WORK

Research on embedded firmware has been very popular over the past decade and some outstanding examples are referenced in this chapter. It will also give a brief overview over today's landscape of Bluetooth hacking and analysis platforms.

3.1 BROADCOM WIFI AND BLUETOOTH CHIP FAMILY

Especially noteworthy is the research surrounding the *Broadcom* embedded WiFi and Bluetooth chip family, including the *BCM20734*, *BCM4329*, *BCM4339* and *BCM4359*.

3.1.1 *WiFi Firmware Modifications*

In 2012 Blanco et al. [11] published their work about reverse engineering and patching the *BCM4329* firmware. This chip was used by many flag-ship mobile devices at this time and the authors showed that it is possible to enable new features such as WiFi monitor mode by firmware modification. Three years later, Schulz et al. [45] released an open source firmware modification and patching framework called *Nexmon* [46]. The framework was initially targeting the *BCM4339* chip inside the *Nexus 5* smartphone (hence the name) but has since been extended to work with many other *Broadcom* chips and also non-WiFi related firmware [44].

Nexmon *is a firmware patching framework mainly targeted at* Broadcom *WiFi chips.*

3.1.2 *WiFi Firmware Vulnerabilities*

The aforementioned work paved the way for security researchers to analyze the firmware for vulnerabilities. Gal Beniamini from *Google Project Zero* found a heap-based buffer overflow bug in the *BCM4359* firmware and wrote a detailed blog post describing the steps to leverage it to an over-the-air remote code execution vulnerability (*CVE-2017-0561*) [7]. About at the same time Nitay Artenstein from *Exodus Intelligence* found a similar vulnerability (*CVE-2017-9417* a.k.a. "*Broadpwn*") in the *Broadcom* firmware which also leads to remote code execution [4].

Vulnerabilities inside the firmware are a serious threat and can also lead to a complete compromise of the host system.

In a second blog post [8] Beniamini went one step further and showed that it is also possible to take control over the kernel running on the main application processor in the smartphone. He presented two exploitation paths using different communication channels between the host and the WiFi chip. The first path exploited a vulnerability in the *Broadcom* WiFi driver running inside the kernel and processing events coming from the WiFi chip. The second path abused the WiFi chip's Direct Memory Access (DMA) ability to write directly to the Random Access Memory (RAM) of the main application processor.

3.1.3 *Bluetooth Firmware Research*

The latest handheld game console from *Nintendo*, the *Nintendo Switch*, has two detachable controllers called *Joy-Cons* which are connected via Bluetooth to the main console. Each *Joy-Con* contains a *BCM-20734* Bluetooth controller which also manages the hardware buttons. Therefore, the *BCM20734* has drawn interest by the game console hacking community and many useful resources about the chip and its firmware can be found online in forums and on *Github*. One example is a *git* repository [47] containing scripts which can load *Broadcom* firmware patch files (HCD files, which are described in Section 4.6) and apply them onto an *IDA Pro* database. Another repository [1] has a discussion thread about dumping the chip's firmware and contains, amongst others, address offsets of memory sections in the firmware.

3.2 FIRMWARE ANALYSIS AND REVERSE ENGINEERING

Compared to the analysis of software running in the userspace on top of an operating system, firmware analysis has to struggle with many additional obstacles and challenges. This section contains references to research projects which focus on these challenges and propose novel techniques to assist the reverse engineering task.

3.2.1 *Firmware Emulation*

Firmware emulation can give deep insights into the firmware at runtime and compensate for missing debugging capabilities.

Debugging firmware running on an embedded chip is often not possible without a hardware debugger and physical access to the debug pins. The lack of a debugger makes reverse engineering and analysis much harder as the researcher is not able to trace and comprehend complex code paths. A project called *LuaQEMU* [33] tries to overcome these issues by approximating a full-system emulation of the firmware. The tool is based on the popular *QEMU* emulator [41] and exposes several internal Application Programming Interface (API) functions to a *LUA* interpreter. In a blog post [34] the authors of *LuaQEMU* explain how the tool is used in practice. As example scenario the *CVE-2017-0561* vulnerability in *Broadcoms* WiFi firmware for the *BCM4358* is reproduced inside the emulator. Along the way they show that with some reverse engineering and patching effort it is possible to emulate the firmware including the initialization routine and reach a state were the firmware is ready to receive actual WiFi packets. This setup can be extended to do further security analysis of the firmware and allows very fine-grained instrumentalization. However, the blog post makes clear that the approach relies on the ability to dump the RAM during runtime in order to setup a working state for the emulator. For the *Broadcom* WiFi chip this can be done with the `dhdutil` [56] tool provided by *Broadcom*. Unfortunately, no such tool exists for the Bluetooth firmware. The *InternalBlue* framework will fill this gap and therefore might enable similar emulation setups.

Emulation often requires a memory dump of the RAM as an initial state.

3.2.2 *The Code-Cut Problem*

At the *REcon* 2018, a security researcher by the handle *evm* presented about automating static analysis of embedded firmware images [57]. The talk emphasizes firmware reverse engineering challenges, especially the problem of having a single, large address space with no clear distinction between application, library and operating system code. This is the result of the linker which produces the firmware by concatenating all object files and therefore discarding the module boundaries of the original source code. The author formalizes the so-called *Code-Cut* problem as the challenge of recovering the boundaries of the original object files when only the call graph of the firmware is given. He then continues to present a solution strategy which applies the concept of local function affinity.

The linker produces one single large binary chunk and discards module boundaries. This raises the difficulty for reverse engineers.

3.2.3 *Real-Time Operating Systems*

A year earlier, at the *REcon* 2017, Vitor Ventura and Vladan Nikolic from *IBM* presented their work on reverse engineering firmware which is based on Real-Time Operating Systems (RTOS), in this case *FreeRTOS* [53]. As it turns out, the Bluetooth controller firmware running on the *BCM4339* features many components which are also part of a RTOS. This includes for example a scheduler for preemptive and cooperative multitasking, blocking queues, semaphores and timers (see Section 4.7). In their talk Ventura and Nikolic give an overview of real-time operating system internals and list the initial steps of the reverse engineering process such as finding the initial entry point by parsing the Interrupt Vector Table (IVT). They continue to present a plugin for *IDA Pro* which can filter functions that access internal and external peripherals.

A RTOS usually includes data structures for multitasking, such as queues or semaphores.

3.3 BLUETOOTH HACKING

This section presents work related to Bluetooth Security and Analysis.

3.3.1 *The Frequency-Hopping Challenge*

When attempting security research on Bluetooth over the air, the first major challenge is to be able to reliably capture traffic between two devices. Dominic Spill summarizes the reasons for this problem in his blog post *"Sniffing Bluetooth is Hard"* [49]. Apart from having a complex and feature-rich protocol stack, Bluetooth uses frequency hopping which makes it significantly harder to be captured than other standards in the 2.4 GHz band such as WiFi or ZigBee. The blog post explains that in order for an attacker to find the correct pseudo-random hopping sequence, he would have to learn the Upper Address Part (UAP) and partial clock state of the target device first. Both can not be easily obtained from sporadically captured packets.

The hopping sequence of a piconet depends on parts of the masters Bluetooth address and internal clock.

At the *DEFCON* conference in 2009, Ossmann et al. presented a Bluetooth sniffing technique based on intentional aliasing [38]. They use a Software-Defined Radio (SDR), in this particular case a *USRP2*, which is able to capture 25 Bluetooth channels simultaneously. Through bypassing the anti-aliasing filters inside the *USRP2* they create intentional aliasing. As a result the 79 Bluetooth channels overlap each other within the bandwidth that can be captured by the *USRP2*. The downsides of this approach are that it only works in low-noise environments with no other Bluetooth piconets present. Additionally, it is still not easy to demodulate all 25 captured channels in real time without a sufficiently powerful computer.

Capturing all 79 channels simultaneously is possible but comes with certain penalties.

3.3.2 *The Ubertooth*

Two years later in 2011, Michael Ossmann started the *Ubertooth* [37] project which is an open-source development platform for Bluetooth experimentation. The project consists of an Universal Serial Bus (USB) dongle featuring a microcontroller and a Radio Frequency (RF) interface capable of receiving and transmitting Bluetooth Basic Rate (BR) and Low Energy (LE) packets. The host software which is also part of the project operates the dongle and provides features such as passively scanning for undiscoverable but actively transmitting Bluetooth devices in proximity. In a blog article [35] Ossmann explains the implemented techniques for passively recovering the UAP and partial clock state of the Bluetooth piconet from the captured information. Together with the Lower Address Part (LAP), which can be easily obtained from every captured packet, these values can be used to recover the frequency hopping sequence and hop along with the target devices.

The Ubertooth *can calculate the masters clock and UAP from sporadically captured packets.*

Due to its open-source and low-cost nature, the *Ubertooth* project is a major advance in making the lower layers of the Bluetooth stack accessible to researchers. However, there are of course some shortcomings which are important to mention:

- the *Ubertooth* is not capable to process Enhanced Data Rate (EDR) packets, and
- the dongle is limited to a bandwidth of 1 MHz. It is only possible to capture the entire traffic of one single connection if the frequency hopping sequence is known or can be determined ahead of time.

This might become a problem for researchers that need to reliably capture the full sequence of packets transmitted over a Bluetooth connection. In conclusion, the *Ubertooth* is a powerful platform for implementing practical techniques for analyzing and attacking Bluetooth. Due to its openness and also affordability it was adopted and extended by many researchers. However, there is still room for improvement, and this thesis aims to add a new research platform with complementary capabilities.

The Ubertooth *is not suitable for experiments that require reliable packet capturing.*

3.3.3 *Bluetooth Weaknesses*

A paper by Chai et al. [21] gives a top-level survey of known Bluetooth vulnerabilities and attack primitives. The authors point out that the weakest point of the protocol is during the initial stages of a connection setup which includes device inquiry, paging and pairing. Especially the legacy pairing procedure has been shown to have major weaknesses [48].

The initial pairing procedure is a weak point in Bluetooth security.

In 2010 Haataja et al. [28] presented a Man-In-The-Middle (MITM) attack approach against the Secure Simple Pairing (SSP) procedure. SSP has been introduced with Bluetooth 2.1 and replaces the legacy pairing mechanism. In their attack the authors manipulate the initial exchange of Input-Output (IO) capabilities between the victim devices. This way they can make both devices belief that they pair with a device that has no sufficient IO capabilities. Hence, the devices choose the insecure Just-Works (JW) association model instead of doing a secure Out-Of-Band (OOB) check. Downgrading attacks like this are a popular attack vector against handshake protocols and have also been used against other protocols such as Transport Layer Security (TLS) [12].

Several severe vulnerabilities in popular Bluetooth stacks were disclosed in 2017 under the name *Blueborne* [50]. Although not the Bluetooth protocol itself was targeted, this release showed that implementation weaknesses can be found on any major platform, including *Android*, *Linux*, *iOS* and *Windows*. Many of the vulnerabilities that have been uncovered can be abused to execute arbitrary code on the

Vulnerabilities can be found in any major Bluetooth stack implementation.

remote device through memory corruptions. Another attack vector which was presented by the authors is to abuse weaknesses in the Personal Area Network (PAN) profile implementation of the remote Bluetooth stack in order to create a malicious network interface. This allows an attacker to gain a MITM position for all of the Internet Protocol (IP) traffic of the victim.

4 FIRMWARE REVERSE ENGINEERING

This chapter contains a collection of information about the internal structure and functioning of the *Broadcom BCM4339*. The information was obtained through various different methods and channels such as:

- vendor provided datasheets,
- public patents,
- reverse engineering of the extracted firmware,
- analysis of Random Access Memory (RAM) dumps of the running firmware, and
- online investigation in various forums, blogs and mailing lists.

For better readability, the presented information and results of the reverse engineering process are not in chronological order of their finding. They are instead grouped into coherent topics and presented in an order of increasing technical depth and complexity. Additionally, the process of obtaining the firmware and its reverse engineering is not described in this chapter but in the subsequent chapter which covers the implementation and usage of various reverse engineering and analysis tools, including the *InternalBlue* framework.

4.1 OVERVIEW

The *Broadcom BCM4339* combines an IEEE 802.11ac (WiFi) and Bluetooth 4.1 transceiver on a single, highly integrated chip. It is part of *Broadcoms* Wireless Internet of Things (IoT) product line which was acquired by the *Cypress Semiconductor Corporation* in 2016 [23]. Since then it got rebranded and is now also known under the part number *CYW4339*.

The BCM4339 *combines a WiFi and Bluetooth interface on a single chip.*

The chip is used as WiFi and Bluetooth chip inside smartphones like the *Google Nexus 5*. It features a shared receiver path for the Bluetooth and WiFi signals which also includes a single, shared Low Noise Amplifier (LNA) and Automatic Gain Control (AGC) algorithm. Signaling is done between the WiFi and the Bluetooth core on the link layer which is advertised under the term *Enhanced Coexistence*.

Inside the chip the WiFi and Bluetooth units are separated and each unit operates on its own processor core. Figure 5 shows a block diagram from the *CYW4339 Cypress* datasheet [6]. The Bluetooth processor is an *ARM Cortex-M3* that has 196 KB RAM and 608 KB Read Only

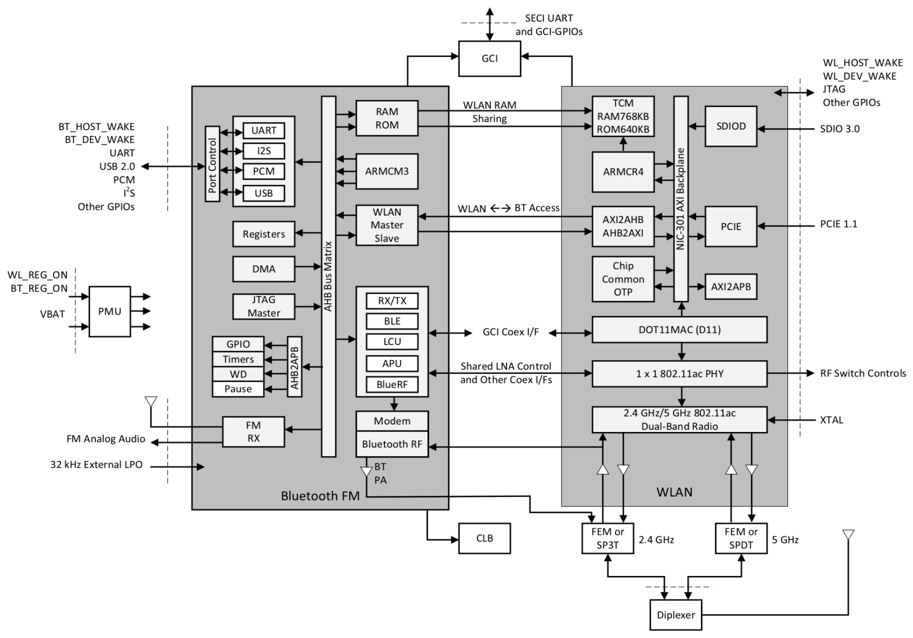


Figure 5: Block Diagram of the *BCM4339* Chip. [6]

The physical layer of the Bluetooth protocol is managed by a separate core.

Memory (ROM). In addition the chip contains a Bluetooth modem which is controlled by a component referred to as `Bluetooth Baseband Core`. This component handles the lower layer of the Bluetooth protocol stack while the higher levels such as the Link and Device Manger are implemented in the Cortex-M3.

The communication with the host system is done via the Host Controller Interface (HCI) protocol over a Universal Asynchronous Receiver-Transmitter (UART) interface or Universal Serial Bus (USB). The default baud rate for the UART interface is 115000 baud but the chip features baud rates up to 4 Mbps and has two 1040 byte buffers for receiving and transmitting data over the interface. It also contains a Pulse-code Modulation (PCM) connector for transferring audio data. The components on the chip use the *ARM* Advanced High-performance Bus (AHB) for communication between each other.

4.2 PROCESSOR ARCHITECTURE AND MEMORY LAYOUT

To analyze a firmware image it is important to first understand the underlying processor architecture, its instruction set and the memory layout of the address space. Most of this information can be learned from the *Cypress* datasheet [6] and the *ARM Cortex-M3* Technical Reference Manual (TRM) [22]. The *ARM* Cortex-M3 implements the *ARMv7-M* architecture and therefore uses the Thumb-2 instruction set. Thumb-2 instructions are either 16 or 32 bits wide. They must always be word-aligned. The *ARM* architecture uses the Least Significant Bit (LSB) of the address to distinguish between the *ARM* and the Thumb instruction set. When operating in Thumb mode, the

Start	End	Size	Type	Description
`0x000000`	`0x090000`	576 kB	ROM	Most of the firmware
`0x0D0000`	`0x0D8000`	32 kB	RAM	*Patchram*
`0x200000`	`0x228000`	160 kB	RAM	Stack and heap
`0x260000`	`0x268000`	32 kB	ROM	Data and jumptables
`0x300000`	`0x300400`	1 kB	IO	Memory Mapped IO
`0x310000`	`0x310400`	1 kB	IO	*Patchram* Control
`0x314000`	`0x314400`	1 kB	IO	Memory Mapped IO
`0x318000`	`0x321800`	38 kB	IO	Memory Mapped IO
`0x324000`	`0x325000`	4 kB	IO	Memory Mapped IO
`0x326000`	`0x32D000`	28 kB	IO	Memory Mapped IO
`0x350000`	`0x364000`	80 kB	IO	Memory Mapped IO
`0x600000`	`0x600800`	2 kB	IO	Memory Mapped IO
`0x640000`	`0x640800`	2 kB	IO	Memory Mapped IO
`0x650000`	`0x650800`	2 kB	IO	Memory Mapped IO
`0x20000000`	`0x20090000`	576 kB	ROM	Copy of main ROM
`0x200D0000`	`0x200D8000`	32 kB	RAM	Copy of *Patchram*
`0xE0000000`	`0xE0100000`	1 MB	IO	Private Peripheral Bus

Table 1: Memory Map of the Firmware's Address Space.

program counter has the LSB always set to 1. Because the Cortex-M3 does only support the Thumb-2 instruction set, all code references and offsets in the firmware do always point to an odd address.

Unfortunately, the datasheet does not contain information about the mapping of the different memory regions into the address space of the chip. This had to be uncovered during the initial reverse engineering attempts (see Section 5.4.2). Both, the RAM and the ROM memory are split up into two sections respectively. Additionally, Input-Output (IO) registers are mapped into the address space as well. Table 1 lists each discovered memory section, their start and end address, as well as their type and purpose.

The RAM and ROM are each divided into two sections.

The main ROM section starts at address `0x0` but is also mapped at address `0x20000000`. It has a size of 576 kB and contains mostly code and the Interrupt Vector Table (IVT). The other ROM section which starts at address `0x260000` does not contain code but holds constant data such as function tables, cryptographic constants and test data.

The ROM sections contain the code and constants.

The RAM section at address `0xD0000` is called the *Patchram* and is also mapped at address `0x200D0000`. It is used for runtime patches to the firmware and holds the patch code as well as a *Patchram* table. The *Patchram* mechanism is explained in Section 4.6.

The RAM sections contain code patches and runtime data.

At address `0x200000` starts the main RAM section which is also referred to as *Scratchpad*. It contains the application stack, global

Offset	Interrupt Name	Value
`0x00`	Initial Stack Pointer	`0x200400`
`0x04`	Reset	`reset_handler_200`
`0x08`	NMI	`NMI_handler_6759E`
`0x0C`	Hard Fault	`fault_handler_67566`
`0x10`	Memory Management	`fault_handler_67566`
`0x14`	Bus Fault	`fault_handler_67566`
`0x18`	Usage Fault	`fault_handler_67566`
`0x2C`	Supervisor Call	`sv_call_handler_688C4`
`0x30`	Debug	`debug_handler_675A0`
`0x38`	Pending SV Call	`sv_call_handler_688C4`
`0x3C`	System Tick Timer	`systick_handler_67594`
`0x40-0x110`	Vendor Specific Interrupts	

Table 2: The Interrupt Vector Table of the *BCM4339*.

data structures and the heap. The initial stack on boot time starts at address `0x200400` and grows towards lower addresses. However, later in the boot process, the initialization routine starts multiple threads and each thread has its own, dedicated stack inside the main RAM. The initialization of the chip will be explained in Section 4.8.

From address `0x300000` and above follow multiple regions of what seems to be memory mapped IO or system control registers. For many of them their purpose could not yet be reverse engineered from the firmware because of time constraints during this thesis. Finally, at address `0xE0000000` lies the *ARM Private Peripheral* bus, which holds system control registers.

4.3 INTERRUPT CONTROLLER

The *Cortex-M3* features a Nested Vectored Interrupt Controller (NVIC) which handles exceptions and external interrupts. The associated IVT starts at address `0x0` and is shown in Table 2.

4.3.1 *Interrupt Masking*

Except from Reset, Non-maskable Interrupt (NMI) and Hard Fault, the priority of all exceptions is configurable. The priority mask register (`PRIMASK`) can be set to 1 in order to temporarily disable all exceptions with configurable priority. This register can either be read or written directly with the `MRS` and `MSR` instructions or it can be cleared or set by the special `CPSIE I` and `CPSID I` instructions. The firmware contains a set of functions to manage interrupts and disable them

Listing 1: Interrupt Management Functions.

```
get_primask_register_43A00:
    MRS.W   R0, PRIMASK
    BX      LR

set_primask_register_43A0C:
    MSR.W   PRIMASK, R0
    BX      LR

disable_interrupts_43A06:
    NOP
    CPSID   I
    BX      LR
```

on critical code paths. They belong to the group of most used functions in the firmware and are shown in Listing 1. To protect a critical code path from being interrupted one would first use the `get_primask_register` function to save the current state of the `PRIMASK` register. Then interrupts can be disabled by `disable_interrupts`. Once the critical code path is passed the `set_primask_register` function is used to restore the original state of the `PRIMASK` register.

The firmware disables most interrupts in critical code paths which must not be interrupted.

4.3.2 *Fault Handler*

The fault handler is of special interest when it comes to experimenting with patches and modifying the memory of the chip in the next chapter. As seen in Table 2, the fault handler handles the Hard Fault, Memory Management, Bus Fault and Usage Fault exceptions. This means it is invoked in case the processor tries to execute an invalid instruction, reads from or writes to an invalid or protected address or encounters any other non-recoverable fault situation. The fault handler implementation of the *BCM4339* sends multiple vendor specific HCI events which contain, amongst others, the program counter, stack pointer, link register and the values of the registers `r0` to `r6`. Additionally, they contain a complete copy of the *Scratchpad* RAM section.

In case of a fault situation the firmware sends vendor specific HCI events which contain a crash report.

Each of the vendor specific event payloads start with the four-byte fixed constant `0x039200f7`, whose purpose is not clear, other than identifying the packet as message from the fault handler. It is followed by three one-byte fields which could be named `fragment_type`, `checksum` and `payload_type`:

FRAGMENT_TYPE This field is set to `0x2c` in case of unfragmented payloads. If the payload is fragmented over multiple event packets, the `fragment_type` is set to `0xf0` except for the last fragment which has the field set to `0x4c`.

CHECKSUM The checksum is calculated such that summing up all bytes in the packet starting from the checksum field results to `0x00` when trimmed to one byte.

PAYLOAD_TYPE There exist three types of payloads which are indicated by the values `0x01`, `0x02`, and `0x03`. Type `0x01` could not yet be reliably reverse engineered. The second type, `0x02`, contains a one-byte number of contained register values, followed by two null bytes and finally the register values of `pc`, `lr`, `sp` and `r0 - r6` in little endian byte order. Finally, type `0x03` contains a part of the RAM dump. The payload starts with an one-byte length field, followed by two null bytes and then the address of the following dump in little endian byte order.

4.4 FLASH PATCH AND BREAKPOINT UNIT

The FPB unit can be used for ephemeral patches to instructions or data located in the ROM.

The Flash Patch and Breakpoint (FPB) unit is a debugging feature of the *Cortex-M3* (Section 11.4 of the *ARM Cortex-M3* TRM [22]). It allows developers to temporarily patch parts of the program in the ROM. The CPU compares the addresses which are loaded from the instruction or data bus to a programmer defined target address and, in case of a match, returns a different value which is also defined by the programmer. The special purpose registers and their usage is explained in the following subsections.

4.4.1 *Comparator Registers*

The *Cortex-M3* features eight comparator registers (`FP_COMP0` to `FP_COMP7`) which are mapped at address `0xE0002008` to `0xE0002020`. Theses registers can hold addresses which are then remapped to new values. Comparator 0 to 5 are matching on the instruction bus whereas comparator 6 and 7 are so called literal comparators and match on the data bus. Figure 6 shows the structure of a comparator register. Each register has its own `ENABLE` flag at the least significant bit. The two most significant bits are called `REPLACE` and used as mode selector:

Bit 1	Bit 0	Interpretation
0	0	Remap to memory section defined in `FP_REMAP`.
0	1	Set lower halfword to a `BKPT` instruction.
1	0	Set upper halfword to a `BKPT` instruction.
1	1	Set both halfwords to a `BKPT` instruction.

4.4.2 *Control and Remap Registers*

The FPB unit has a global `ENABLE` flag inside the control register (`FP_CTRL`) which is located at address `0xE0002000`. The unit is disabled

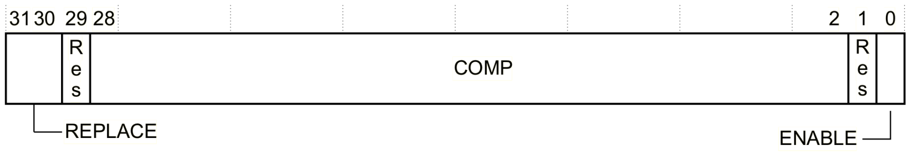


Figure 6: Structure of the FP_COMP Registers. [22]

by default in the *BCM4339* and can be enabled by writing `0x3` to the control register.

The `FP_REMAP` register should be set to a 32-byte aligned address inside the RAM. It is used as base address for looking up the remapped values for comparator 0 to 7. The most significant three bits of this address are always hardwired to `001`. In case of the *BCM4339* the only RAM section which is compliant with this restriction is the *Patchram* section as it is also mapped at address `0x200d0000`.

4.5 host controller interface (hci)

The Host Controller Interface (HCI) provides a uniform method for the host to access controller capabilities and send or receive data from the underlying Bluetooth links. The protocol is based on commands sent by the host to the controller and events which are sent from the controller to the host. The HCI protocol can be transported over different transport layers such as UART, USB or Secure Digital Input Output (SDIO). As the *Nexus 5* uses UART to communicate with the *BCM4339*, only this transport layer will be considered in the following sections.

HCI commands are sent from the host to the controller and HCI events in the other direction.

4.5.1 *HCI Protocol*

Four types of HCI packets are defined on the UART transport layer:

1. HCI command packets,
2. HCI Asynchronous Connection-Less (ACL) data packets,
3. HCI Synchronous Connection-Orientated (SCO) data packet,
4. HCI event packets.

The ACL and SCO data packets are not relevant in the scope of this thesis and therefore only the command and event packets are described in this section. Details about the HCI protocol layer can be found in Volume 2, Part E of the Bluetooth Core specification [15].

The structure of a HCI command packet on the UART transport layer is shown in Figure 7a. It starts with a one-byte type identifier that is always `0x01` for the command packet. After that follows a two-byte `opcode` which can be logically divided into a 6-bit group field

`0x01`	opcode	length	payload
1	2	1	<length>

(a) HCI command packet

`0x04`	event code	length	payload
1	1	1	<length>

(b) HCI event packet

Figure 7: Structure of a HCI Command (a) and Event (b) Packet on UART.

and a 10-bit command field. The length field contains the length of the payload in bytes. All specified opcodes and their corresponding payload structures are listed in Volume 2, Part E-7 of the Bluetooth Core specification. [15]

A HCI event packet (Figure 7b) also starts with the type identifier which in this case is set to `0x04`. The event code uniquely identifies the type of the event and its payload structure and content. The length field again contains the total length of the payload in bytes.

The HCI protocol specifies that the controller has to reply to every HCI command packet within a short timeout by either a `Command Complete` event containing the return parameters or by a `Command Status` event in case of long running background commands.

4.5.2 *HCI Handler*

The BCM4339 *firmware has a function table with one entry per HCI command opcode.*

The *BCM4339* contains a nested function table that holds handler functions for all HCI commands. The first stage is indexed by the HCI command group number. It contains pointers to the respective function table for each command group. The handler function prototype is shown in Listing 2. It is called with two arguments:

1. a pointer to the status or error code variable, and
2. a pointer to the HCI command packet starting at the opcode field.

For each HCI packet type the firmware has a queue to dispatch incoming packets.

To handle incoming HCI commands, the firmware periodically invokes the function `dispatch_hci_cmd_to_handler()`. It queries a queue structure (see Section 4.7) for the number of unhandled HCI command packets. For every command, the dispatcher retrieves the corresponding command handler function by using the `get_handler_for_hci_cmd()` function. Finally, the handler function is invoked and processes the command. It can send HCI events by using the function `send_hci_event()`.

Listing 2: HCI Functions.

```
void (hci_cmd_handler*)(char *error_code, char *hci_cmd_buffer);

hci_cmd_handler* get_handler_for_hci_cmd_99DC(short opcode);

void send_hci_event_79E4(char *hci_event_buffer);
```

4.5.3 *Vendor Specific HCI Commands*

Apart from the standard HCI commands specified in the Bluetooth standard, the *BCM4339* also implements vendor specific commands. These commands can be looked up in a datasheet provided by *Cypress* [18]. Table 3 contains an incomplete list of *Broadcom*'s vendor specific HCI commands. The `Write_RAM`, `Read_RAM`, `Download_Minidriver` and `Launch_RAM` commands are relevant in order to understand the firmware patching process (Section 4.6) and the debugging setup which is described in Section 5.2. Therefore they are explained in greater detail now.

Broadcom *implements special HCI commands which are used in the firmware update process.*

READ_RAM is a command which reads up to 251 bytes of memory from the chips address space and sends it back to the host in the HCI command complete event. Contrary to its name, this command is also able to read the ROM. Any attempt to read unmapped addresses or beyond the end of a memory section causes the chip to crash. The command requires exactly two arguments:

- **Address**: 32-bit (little endian) address from where to read memory.
- **Length**: number of bytes to read (8-bit parameter, max: 251).

WRITE_RAM is the counterpart of the `Read_RAM` command which can be used to write up to 251 bytes to the RAM regions of the chips memory. If attempting to write to a ROM section, the command has no effect. However, trying to write to an unmapped address causes the system to crash. The command takes the following arguments:

- **Address**: 32-bit (little endian) target address for the data.
- **Data**: Up to 251 bytes of data which is written to RAM.

DOWNLOAD_MINIDRIVER will put the device into a special mode in which it is safe to receive patches. In this mode the normal Bluetooth activity is disabled. Only a very reduced subset of HCI commands are interpreted in the download mode, including `Read_RAM`, `Write_RAM` and `Launch_RAM`. The latter is being used

CMD	Name	Description
`0xFC18`	Update Baudrate	Set the Baud rate of the UART interface of the controller
`0xFC45`	Write UART Clock Setting	Change the UART clock (24 MHz or 48 MHz)
`0xFC2E`	Download_Minidriver	Set the chip into a state where it can receive patches
`0xFC4C`	Write_RAM	Write data to the RAM of the chip
`0xFC4D`	Read_RAM	Read data from the RAM of the chip
`0xFC4E`	Launch_RAM	Leave the Download_Minidriver state and reboot to apply the patches

Table 3: Vendor Specific HCI Commands for *Broadcom*.

to to exit the download mode and reboot into the normal Bluetooth firmware. Any downloaded ROM patches are applied at an early stage during this reboot (see Section 4.6). The `Download_Minidriver` command takes no arguments.

LAUNCH_RAM can be used to continue code execution on the chip at a specified address using Thumb mode. One of the intended use cases is to exit the *Download_Minidriver* mode by specifying the pseudo-address `0xFFFFFFFF` which will issue the chip to reboot into Bluetooth mode and apply the downloaded patches. Another use case is to jump to the entry point of a so-called *Minidriver* which was previously loaded into RAM using the `Download_Minidriver` and `Write_RAM` commands. The `Launch_RAM` command takes the target address (32-bit, little endian) as its only argument.

4.6 FIRMWARE UPDATES THROUGH HCD FILES AND PATCHRAM

The *BCM4339* chip allows for firmware updates or rather patches through a mechanism referred to as *Patchram*. Although this mechanism is mentioned in the datasheet of the *BCM4339* [6], it is not publicly documented and therefore has to be reverse engineered. For the firmware update the Bluetooth driver on the host uses the above mentioned, vendor specific HCI commands. The patches are shipped in form of a HCD file which contains:

- Multiple `Write_RAM` commands which can write new functions and data to the RAM sections of the chip. ROM patches are also possible through the *Patchram* mechanism which is described below.

- A terminating `Launch_RAM` command with `0xFFFFFFFF` as argument in order to reboot into Bluetooth mode and apply the patches.

The *Android* Bluetooth driver includes the HCD file in its initialization procedure. By first reading the name of the chip via an HCI command it is able to search for a matching HCD file inside the `/system/vendor/firmware` directory. In case it finds an HCD file it will then go forward and send the HCI commands contained in the file one after another until it reaches the end of the file. From there it will continue with the normal initialization of the chip. Figure 8 summarizes the commands which are sent to the chip.

The Bluetooth driver loads ephemeral firmware patches from a HCI file.

As the `Write_RAM` command can only write data to the RAM, patches to the ROM are handled differently. All patches for ROM sections are collected in a list and written to a special address in the RAM. The list is formatted as a chain of Type-Length-Value (TLV) objects which is shown in Figure 9. Each entry starts with a one-byte type field followed by a two-byte length of the payload which is the last part of an entry. Finally, the `Launch_RAM` command will exit the `Download_Minidriver` state, apply the patches to the ROM and eventually reset the chip in order to boot the patched firmware. Table 4 contains a list of all type values that have been found in the *Patchram* list of the `bcm4335c0.hcd` file. Due to time constraints only few of them could be reverse engineered and their purpose is now explained in greater detail.

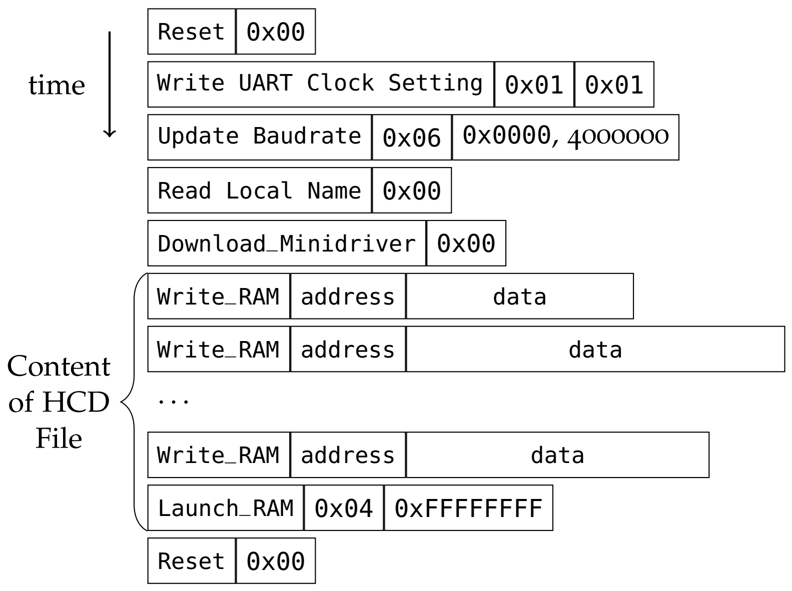


Figure 8: Firmware Patching Procedure.

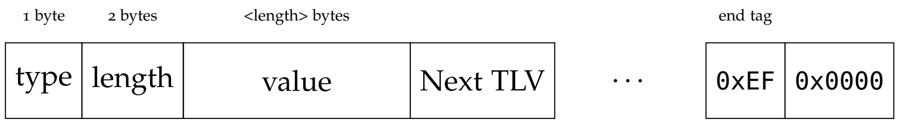


Figure 9: Format of the List of Patches.

Table 4: List of Types for the *Patchram* TLVs.

Type	Length	Description
`0x02`	10	Issue a reboot and continues processing the list after the reset.
`0x08`	15	Patch 32-bit word in ROM.
`0x0a`	var.	Patch arbitray length of bytes in RAM.
`0x40`	6	Set default Bluetooth Device Address.
`0x41`	var.	An ASCII string which is set to be the new local device name.
`0xfe`	0	Marks the end of the TLV list.
`0x03`, `0x0b`, `0x1a`, `0x40`, `0x68`, `0x69`, `0x6f`, `0x70`, `0x82`, `0x86`, `0x90`, `0xb1`, `0xb2`, `0xb3`, `0xc0`, `0xc1`, `0xd8`, `0xfd`		*purpose unknown*

TYPE 0X02 In the analyzed firmware patch (`bcm4335c5.hcd`) this type is used exactly once and relatively early in the list before any of the type `0x08` objects. The value is ten bytes long and consists of a 32-bit little-endian address followed by six zero bytes. When the TLV parser in the `Download_Minidriver` state processes the type `0x02` it initiates a reboot. However, in an early state of the boot process parsing the TLV list is continued at the address specified in the value of the TLV. In case of the `bcm4335c5.hcd` this is actually just the address of the next TLV object in the list.

TYPE 0X08 This type is used for single 32 bit patches to ROM (called *Patchram*). The length of the value is fixed to 15 bytes with the following semantic:

1 Byte	4 Bytes	4 Bytes	2 Bytes	4 Bytes
`#slot`	`target_address`	`new_value`	`0x0000`	*unknown*

The final effect of this *Patchram* object is that four bytes of memory inside the ROM which are located at the `target_address` are temporarily overlain by the `new_value`. Internally this is done by writing the `new_value` into a table at address `0xD0000` (RAM) and the `target_address` into another table at address `0x310000` (hardware register). For both tables the index is given by the `slot` number and the tables are capable to hold up to 128 entries which correspond to slot numbers 0 to 127.

The BCM4339 *can apply at most 128 patches to the ROM.*

TYPE 0X0A This type is used for writing data of arbitrary length into RAM. The first 4 bytes specify the target address and all data that follows gets copied to this address.

4.7 RESOURCE DATA STRUCTURES

This section describes an interesting set of data structures which have been reverse engineered from the firmware and the RAM dump. These structures are related to resource management and are very similar to their equivalents in popular Real-Time Operating Systems (RTOS) such as *FreeRTOS* [43]. Unfortunately, *Broadcom* uses what seems to be a proprietary RTOS which is not documented. The resource structures in this RTOS have a similar form and always contain a four-letter identifier as the first structure member:

Broadcom *uses a proprietary RTOS implementation.*

- "`THRD`" (thread management)
- "`DVDN`" (event management)
- "`BLOC`" (memory management)
- "`QUEU`" (queue data structure)
- "`SEMA`" (semaphore data structure)
- "`TIMA`"
- "`BYTE`"

All resource data structures are organized in double-linked lists and therefore contain two pointers to the previous and the next instance of the data structure in the last two fields. With exception of the `THRD` data structure, all other structures contain a pointer to a thread waitlist. Threads which try to access a resource that is not available at the moment may be suspended and blocked until the resource becomes available again. Details about thread and resource management will be explained in Section 4.8. For resource data structures that could be linked to a semantical purpose the following subsections describe how they work in detail.

If a requested resource is blocked by another thread the RTOS may suspend the current thread.

4.7.1 *THRD*

The firmware manages different threads with a structure called `THRD`. A rough overview of the fields in this structure is shown in Figure 10.

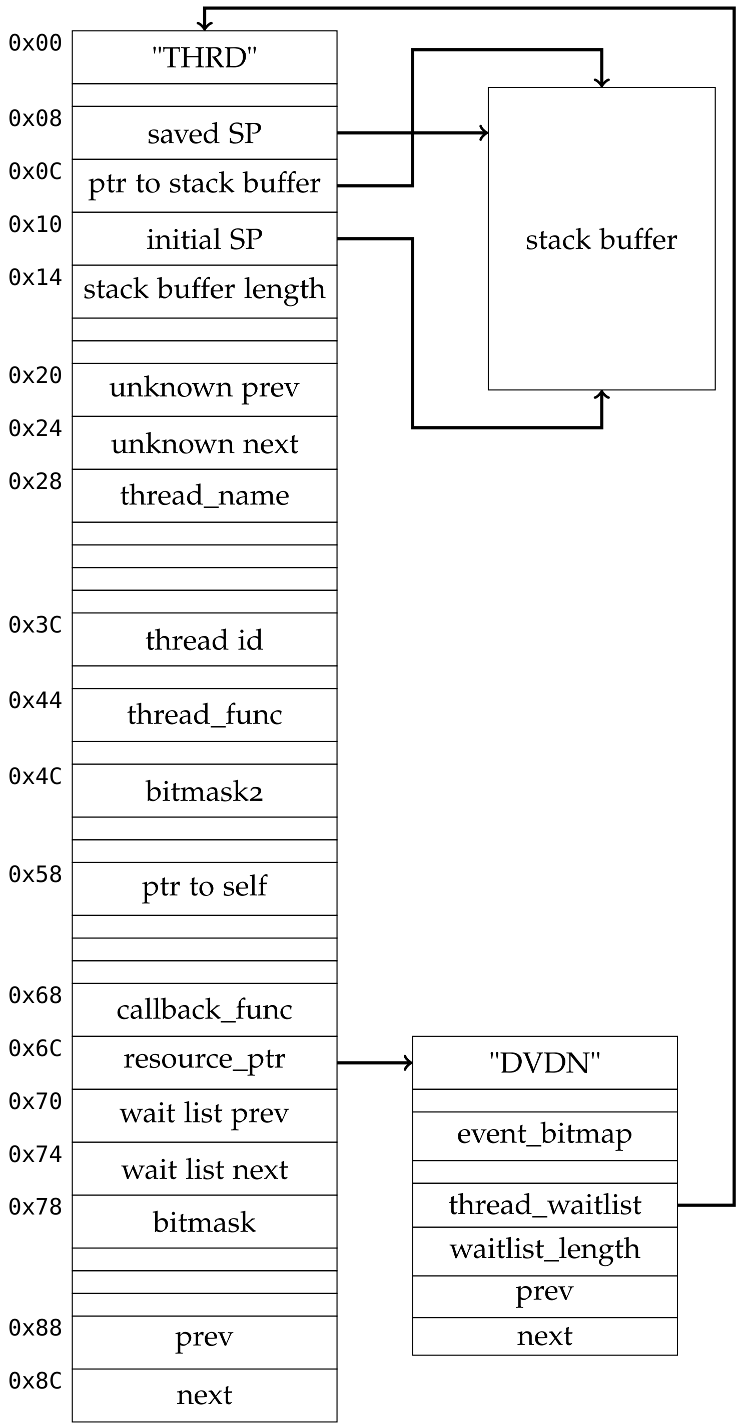


Figure 10: The `THRD` and `DVDN` Data Structure.

The purpose of the fields is explained in this section. For information about how the threads are used in the firmware and how concurrent execution works refer to Section 4.8.

STACK BUFFER Each thread has its own stack which is allocated in a buffer separate from the structure. The buffer is initialized with the byte value `0xEF`. The `THRD` structure contains three pointers to the `stack buffer`:

Each thread has its own stack.

- a pointer to the start of the `stack buffer`,
- a pointer to the end of the `stack buffer` (initial value of the Stack Pointer (SP)), and
- the saved SP.

THREAD_NAME is a pointer to the name of the thread (stored as null-terminated string). Unfortunately, for most of the `THRD` instances in the firmware this pointer is `NULL`.

THREAD_FUNC Each `THRD` structure stores a pointer to the thread function it executes. The function usually consists of an endless loop and will run concurrently with thread functions of other threads.

LIST POINTERS The `THRD` structure has three pairs of next- and previous pointers and is therefore a member of three separate double-linked lists. The pair at the end of the structure belongs to the list which holds all instances of the `THRD` structure. Another pair of list pointers belongs to a double-linked list which acts as a waiting list of threads which wait on the same resource. The purpose of the last pair of linked list pointers could not yet be reverse engineered.

RESOURCE_PTR AND CALLBACK_FUNC In case the thread is waiting on a resource to become available, the `resource_pointer` field contains the address of the corresponding resource structure. All resource structures must contain a 4-character identifier at offset `0x0` and a pointer back to the thread which is at the top of the wait list. The `callback_function` field may contain a pointer to a callback function specific to the type of resource in the `resource_pointer` field. In Figure 10 the `resource_pointer` points to a `DVDN` structure as this is the case which is most likely from observations of the RAM dump. However, the pointer can also point to instances of other resource structures such as `BLOC`, `QUEU` or `SEMA`, if the thread happens to wait on one of these resources.

Threads hold a pointer to the resource they are currently waiting to become available. Most threads wait for events.

4.7.2 *DVDN*

Although the name of the `DVDN` structure could not be recovered from its four-character identifier, its purpose is rather clear. The structure

Listing 3: Functions Operating on the DVDN Structure.

```
int dvdn_set_bits_31D40(struct dvdn *dvdn, int bits, char flags);

int dvdn_get_bits_4B34(struct dvdn *dvdn, int bitmask, int flags,
     int *result_bitmap, int blocking);
```

The DVDN structure is used for signaling between threads.

is deeply involved in the thread scheduling process and it can be reasoned that it is used for event and signal handling between threads. Each thread which handles events has a dedicated DVDN structure assigned which it queries in each iteration of its thread loop. As shown in Figure 10 the structure holds a 32-bit bitmap field that stores which events occurred since the last time the DVDN structure was queried. The bitmap directly affects the control flow of the corresponding thread after it is being scheduled to run.

The functions in Listing 3 are used to access the event bitmap inside a DVDN structure. With dvdn_set_bits(), any component in the firmware is able to signal the occurrence of a certain event to the assigned thread. The dvdn_get_bits() function is periodically invoked by the thread to retrieve this information. The function takes a bitmask as argument which controls the selection of bits from the bitmap. The flags parameter determines if the selected bits should be cleared from the DVDN bitmap (bit 0 of flags) and if the bitmap must contain the complete set of bits from the bitmask (bit 1 of flags). If blocking is set to 1, the function will block the thread until the bitmap does satisfy the conditions of the bitmask.

4.7.3 *BLOC*

The BLOC structure is used for dynamic memory allocations.

The BLOC data structure as seen in Figure 11 manages the temporary allocation of fixed-length buffers which will be referred to as BLOC buffers. Each instance of the structure manages a pool of buffers of the same size. The capacity field stores the number of buffers in the pool and the memory and memory_size fields contain the pointer to the pool memory and its size.

The buffers themself form a single linked list as they contain a pointer to the next buffer in the pool at the start. The buffer_list field points to the first buffer in this list of available buffers. buffer_size corresponds to the size of each buffer but without the additional list pointer at the start. When allocate_bloc_buffer() (see Listing 4) is called, the first buffer is extracted from the buffer_list and the list_length field is decremented by one. The function returns a pointer to the actual buffer, skipping the 4-byte list pointer at the beginning. Additionally, the list pointer is set to point back to the BLOC structure in order to quickly find it inside the free_bloc_buffer() function.

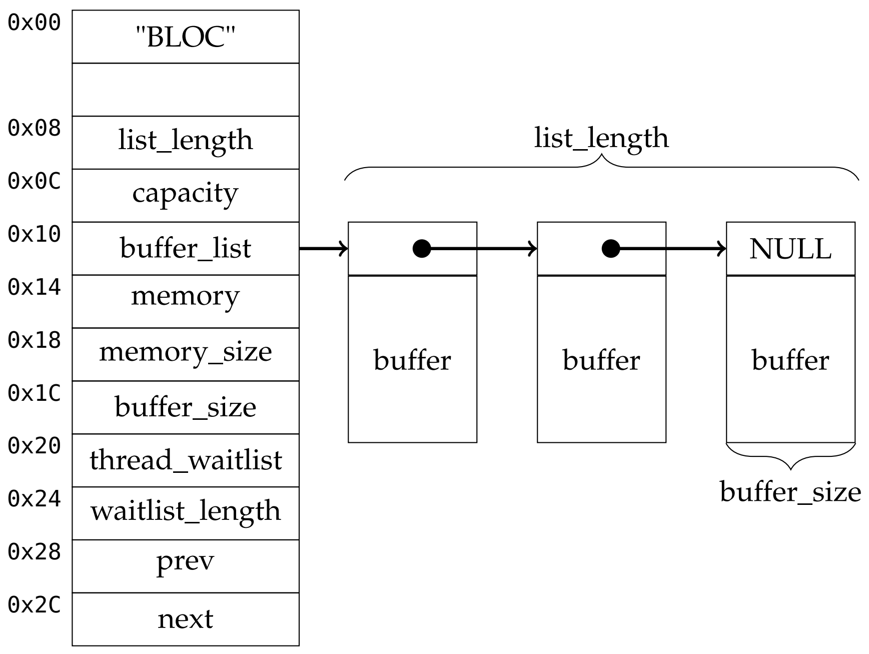


Figure 11: The BLOC Data Structure.

Listing 4: Functions Operating on the BLOC Structure.

```
int malloc_bloc_buffer_4540(struct bloc *bloc, char *out_buffer,
    int blocking);

char* malloc_bloc_buffer_by_size_42994(int size);

int free_bloc_buffer_4728(char *bloc_buffer_ptr);
```

BLOC buffers are used by many other components which need temporary buffers, especially for protocol packets which need to be passed between different protocol layers.

4.7.4 *QUEU*

The QUEU structure implements a queue data structure which is backed by a circular ringbuffer of fixed-sized items. A queue belongs to the First-In-First-Out (FIFO) data structures and is especially useful when passing data chunks between threads. In this case the queue acts as a synchronized data channel. A data buffer containing a protocol payload may be processed by a particular thread, enqueued into a QUEU data structure and later dequeued by another thread which does further processing. For enqueuing and dequeuing packets, the firmware contains two functions, queue_put() and queue_get(), which are shown in Listing 5. The fields of the QUEU structure are explained in the paragraphs below.

The QUEU structure is used for passing ordered data between two threads.

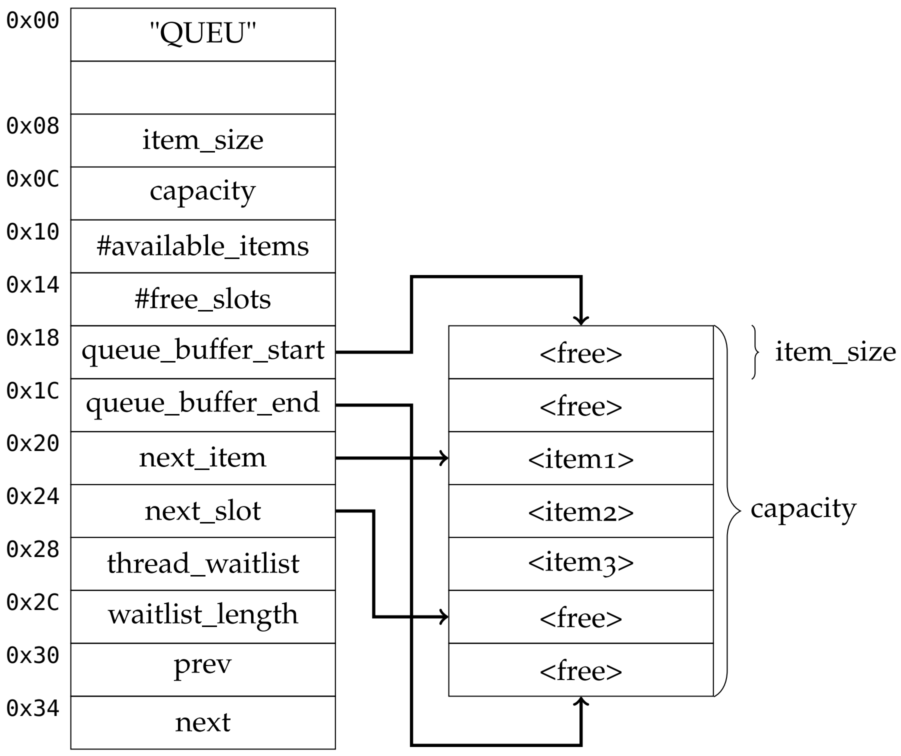


Figure 12: The `QUEU` Data Structure.

ITEM_SIZE This field contains a number which corresponds to the size of a single item in the list. The size is given in double words (4 bytes).

CAPACITY The capacity field contains the maximum number of items in the queue. Therefore, the queue buffer has a size of `item_size` · `capacity` · 4 bytes.

NUMBER OF AVAILABLE ITEMS This is the number of currently enqueued items which can be dequeued by calling `queue_get()`.

NUMBER OF FREE SLOTS The counterpart of the previous field stores the number of currently available free slots in the queue. A new item can be enqueued into a free slot by calling `queue_put()`. Both, the number of available items and available free slots always add up to the capacity of the queue.

QUEUE BUFFER START AND END The `QUEU` structure contains two pointers to the start and end address of the queue buffer.

NEXT_ITEM AND NEXT_SLOT POINTERS These pointers always point to the next retrievable item or free slot respectively. The `queue_get()` and `queue_put()` functions use them to dequeue or enqueue an item. These functions also increase the pointer by the size of an item after each operation, to let the respective pointer point to the next item or slot. If the pointer reaches the

Listing 5: Functions Operating on the QUEU Structure.

```
int create_queue_3F914(char *queu_struct_empty, char*
    queue_buffer, int queue_buffer_len, int item_size);

int queue_put_510C(struct queu *q, char *buffer, int blocking);

int queue_get_4F14(struct queu *q, char *buffer, int blocking);

int queue_get_number_of_available_items_3F99A(struct queu *q)
{
  return q->number_of_available_items;
}
```

value of queue_buffer_end it is set to the value of queue_buffer_start in order to get the behaviour of a ringbuffer.

THREAD_WAITLIST AND WAITLIST_LENGTH As all resource structures, QUEU has a field which may store the pointer to a THRD structure. It is used to implement the behaviour of a blocking queue. This means a thread that tries to read from an empty queue or write to a full queue is suspended until the queue is updated by another thread. The third parameter of the queue_get and queue_put functions is used to switch between blocking and non-blocking invocation. The waitlist_length field stores the number of threads which are waiting on the queue in the waitlist.

4.7.5 *SEMA*

This structure represents a counting semaphore which can control access to a certain resource. As seen in Figure 13 it contains a counter which is initialized with the maximum number of threads that may access the resource at the same time. To acquire access to the resource, a thread has to call the semaphore_acquire() function shown in Listing 6. This function either decrements the counter by one and returns with a success status code or fails if the counter is already zero. By setting the blocking parameter to 1, the calling thread is blocked, until the semaphore is released by another thread with semaphore_release(). The SEMA structure also has a field which holds a string with the name of the protected resource. The only instance of the SEMA structure in the firmware has the name "BTclockUsed".

The SEMA structure is used for synchronizing access to a shared object.

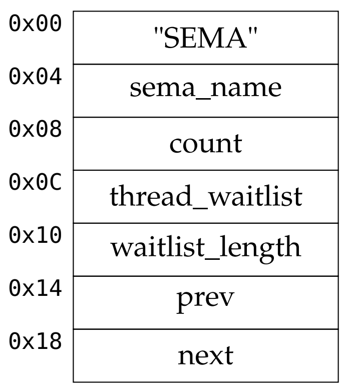


Figure 13: The `SEMA` Data Structure.

Listing 6: Functions Operating on the `SEMA` Structure.

```c
int semaphore_acquire_658C(struct sema *sema, int blocking);

int semaphore_release_666C(struct sema *sema);
```

4.8 CHIP INITIALIZATION AND THREADING

This section describes what happens at boot time right after a `RESET` interrupt occurs. It will explain how and in which order the different threads are started and how they interact with each other.

4.8.1 *Reset Handler*

The entry point of the firmware is the reset interrupt handler at address `0x200`.

The handler for a `RESET` interrupt starts at address `0x200`. At its beginning it will set the `SP` register to its initial value `0x200400` and also zero out the top 256 bytes of the stack as well as registers `r0` to `r10`. It then continues to initialize various different regions and variables in the RAM. If the chip was reset in `Download Minidriver` mode the reset handler processes and installs the downloaded patches as explained in Section 4.6. Eventually, the reset handler initializes everything related to threading:

- two global pointers to the current and next-in-line thread structures,
- list heads of all resource structure types (`BLOC`, `QUEU`, etc.), and
- the system timer of the Cortex-M3.

It will then start the first two of the seven threads in total which are running in the firmware. Unfortunately, only the first thread, which is called `SystemTimerThread`, has a name assigned to it. Apparently, the second thread is an initialization thread, because the bulk of its thread function is executed only once and it initializes many more variables

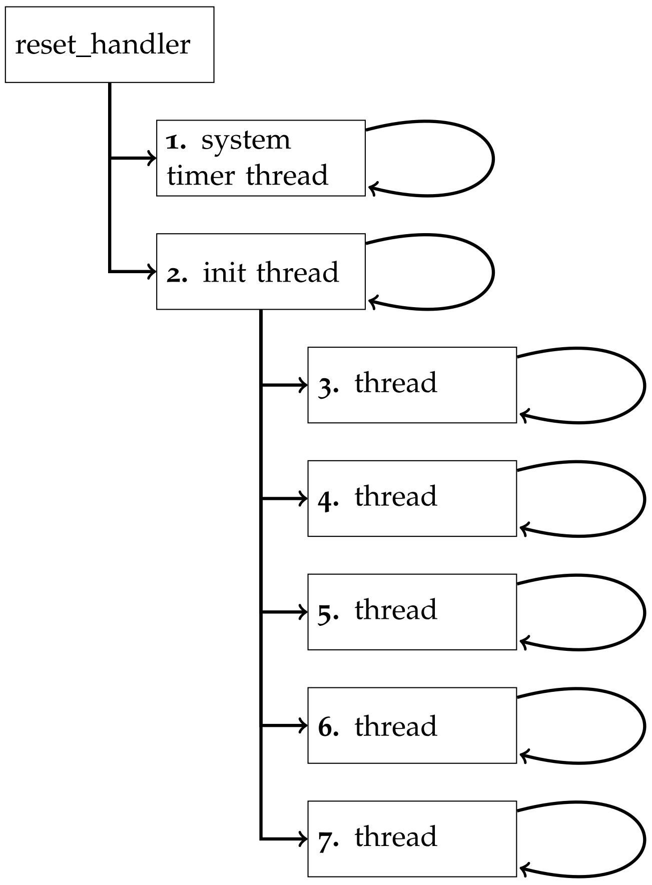


Figure 14: The Thread Initialization Flow.

in RAM and also starts the remaining five threads. Figure 14 shows the hierarchy and order in which the threads are started.

At the end of the reset handler, it uses the `SVC 0` instruction in order to trigger a system service call interrupt. This interrupt is used by the firmware to do a context switch from the current to next thread in line for execution. Therefore the control flow continues in the respective thread function.

4.8.2 *Threads*

The seven threads introduced in the previous section each have a thread function which contains an endless loop. As explained in Section 4.7, many functions which operate on shared resources can be invoked in a blocking manner. If the resource is currently not available, these functions will trigger a system service call interrupt to switch to the next thread which is ready to run.

Most threads have a `DVDN` resource structure assigned to them which is used to dispatch events or signals to the thread. At the beginning of a threads loop it queries the `DVDN` structure for the bitmap which holds all new events. It then invokes handler functions according to the bits set in the event bitmap. The remainder of this section will summarize the purpose of each thread if it could be identified.

The first thread is used for processing timed callback functions.

THREAD 1 The system timer thread does not have a `DVDN` structure assigned to it. Instead it periodically processes a linked list of timer items. These items contain callback functions that are each being called. Other threads can setup timed callbacks by inserting timer items into the list and delete a timer by unlinking the respective item. As many parts of the Bluetooth protocol involves timers, this feature is especially important.

The second thread initializes many components in the firmware.

THREAD 2 This thread does also not have a `DVDN` structure assigned. The main thread function does not contain a periodic loop but is executed only once and linearly. From the context of invoked functions, this thread takes the role of an continuation of the initialization process. The following list only describes a few examples of the actions taken by this thread:

- create most of the queue data structures which are used for packet processing,
- place constants for the National Institute for Standards and Technology (NIST) P-192 and P-256 elliptic curves inside the RAM,
- create `DVDN` structures for threads 3 to 7, and
- start threads 3 to 7.

At its end the main thread function invokes another function which contains a short endless loop whose purpose could not yet be reverse engineered.

The third thread handles incoming HCI and LMP packets.

THREAD 3 This thread has by far the longest thread function of all threads. Only few of its subfunctions could be reverse engineered due to time constraints. The following tasks are, amongst many others, done by this thread:

- processing incoming HCI packets,
- processing incoming Link Manager Protocol (LMP) packets, and
- managing connections and pairing with other devices.

The fourth thread is responsible for dispatching outgoing HCI packets via UART.

THREAD 4 This thread processes four queues which hold HCI packets. Each of the HCI packet types ACL, SCO, HCI event and an unknown type 7 have their own queue. The thread will periodically, query each queue for new packets, retrieve them and send them through the UART interface to the host system.

THREAD 5 AND 6 Unfortunately, the purpose of these threads could not be identified.

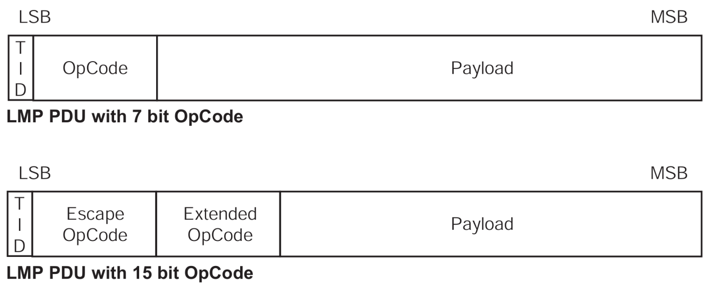


Figure 15: LMP Packet Format. [15]

THREAD 7 This thread mainly operates on IO mapped memory regions and therefore manages some peripheral components.

4.9 LINK MANAGER

The Link Manager (LM) communicates with the LMs of other controllers via the LMP in order to establish and control links between devices. Its tasks include:

The Link Manager controls connections to other devices.

- set-up and control of logical transports and links,
- clock synchronization between Bluetooth master and slaves,
- control frequency hopping sequence, and
- authentication and encryption set-up.

LMP messages are not propagated to higher levels in the protocol stack. Therefore, the protocol can not be observed directly from a host device as all of the LM's functionality is implemented in the Bluetooth controller.

4.9.1 *Link Manager Protocol*

The LMP specifies transactions which are sequences of LMP packets that accomplish a defined task. The packet format used by LMP is shown in Figure 15 and explained below.

Each packet contains a Transaction ID (TID) bit which indicates whether the transaction has been initiated by the master (TID is set to 0) or the slave (TID is set to 1). The TID is needed to detect transaction collisions in which both devices start a particular transaction at the same time. The TID assures that the beginning of a transaction is not misinterpreted as the response to another packet by the remote controller.

LMP transactions can be initiated by the master or the slave.

LMP packets are defined by an opcode that can either be 7 or 15 bits long. The opcode specifies the purpose of the packet and also the length and structure of its payload. All opcodes are defined in

Table 5: Non-exhaustive List of LMP Opcodes.

Opcode	Length	Name
1	2	LMP_name_req
2	17	LMP_name_res
3	2	LMP_accepted
4	3	LMP_not_accepted
5	1	LMP_clkoffset_req
6	3	LMP_clkoffset_res
7	2	LMP_detach

Volume 2, Part C of the Bluetooth Core specification [15]. Table 5 exemplarily lists the first 7 entries of the opcode table. The length refers to the overall packet length including the opcode. Section 6.2.2 lists and explains an actual captured LMP trace of a pairing procedure.

4.9.2 *Connection Related Data Structures*

The connection table inside the firmware can hold information for 12 different connections.

The *BCM4339* stores information about established Bluetooth pairings and connections in a rather large data structure which from hereon will be referred to as `CONNECTION` structure. The structure is 332 bytes long and the firmware can hold up to 12 of them in a global array at address `0x2038E8` which acts as connection table. Most of the fields in the structure have not yet been reverse engineered. Figure 16 illustrates the information that is known about the `CONNECTION` structure and the individual fields are described below.

CONNECTION INDEX The first field in the structure holds an index into another connection related table which will be referred to as `connection_tx` table. The structures stored in that table are used by functions that send LMP packets to the remote controller. Therefore it can be assumed that it manages functionality related to packet transmission.

BITMAP At the bit in position 15 this bitmap stores whether the own controller is the master (bit is set) or the slave (bit is cleared) in this connection.

BLUETOOTH ADDRESS The Bluetooth address of the remote device is stored in this 6-byte field.

BUFFER Each `CONNECTION` structure has an associated buffer to store temporary data. One example where this is used is for `LMP_name_res` packets. As the name of the remote device might exceed the length of a single LMP packet and therefore be distributed

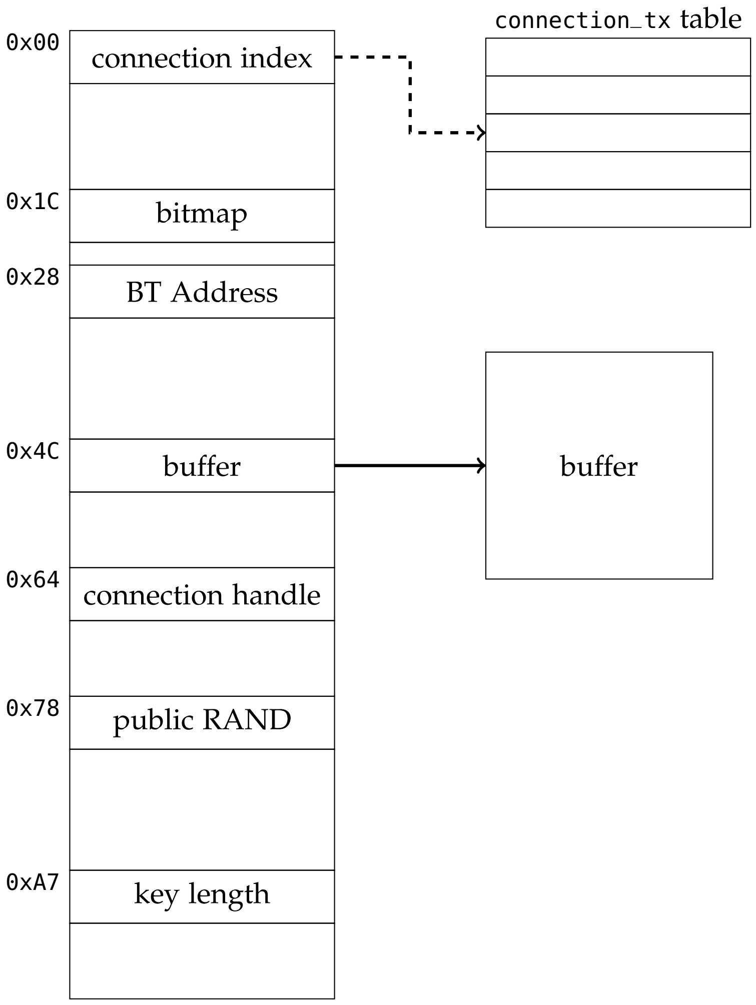


Figure 16: The `CONNECTION` Data Structure.

over several packets, the firmware has to buffer the fragments and reassemble the complete name before it can be processed.

CONNECTION HANDLE The connection handle is used by the HCI layer to uniquely identify a connection. Their usage is specified in Volume 2, Part E of the Bluetooth Core specification [15].

PUBLIC RAND This field holds a 16-byte random number that is used in the pairing and authentication process.

KEY LENGTH This one-byte field holds the effective length of the link key measured in bytes.

4.9.3 *LMP Handler in the BCM4339 Firmware*

At the core of the LM in the *BCM4339* firmware is a handler table for each LMP opcode. The table starts at the address `0x261610` in the second half of the ROM. Each table entry has a length of 8 bytes and contains the following values in the specified order:

- function pointer to the handler function (4 bytes),
- the LMP packet length for this opcode (1 byte), and
- 3 bytes containing additional information.

The firmware handles incoming LMP packets with a function table that contains entries for each opcode.

The third thread periodically checks for incoming LMP packets and calls a dispatcher function if new packets have arrived. At address `0x200478` lies a data structure which holds information about the received packet that is currently processed. The dispatcher function reads the opcode from this global data structure and calls a lookup function to retrieve the corresponding entry from the LMP function table. Finally it calls the handler function from the retrieved entry.

As can be seen in Listing 7 the handler functions take a pointer to the respective `CONNECTION` structure as their only argument and get the data for the received packet from the global data structure. If the received packet requires a response, the handler crafts a new LMP packet and calls `send_LMP_packet()`. This function puts the packet into a linked list inside the `connection_tx` structure for the respective connection. Although the details about this process are not yet fully reverse engineered, it can be observed that this actually causes the LMP packet to be send to the remote device.

Listing 7: LMP Functions.

```
void send_LMP_packet_f81a(struct CONNECTION *c, char* lmp_buf);

void (*LMP_handler)(struct CONNECTION *c);

struct LMP_entry* LMP_lookup_3f2d8(char *lmp_buf);
```

5 INTERNALBLUE IMPLEMENTATION

The collected knowledge from the previous chapters shall be used to create *InternalBlue*, an open and extendable Bluetooth research framework based on the *BCM4339* and similar chips. The aim is to use off-the-shelf consumer devices like the *Nexus 5* and repurpose them as analysis tools for experimenting with the lower layers of Bluetooth connections. By patching the firmware of the Bluetooth controller and writing additional software for the host side it is possible to implement features such as:

- a Link Manager Protocol (LMP) traffic monitor,
- a LMP packet injection tool,
- a LMP fuzzer, and
- a security auditing tool for Bluetooth pairing.

Not all of the above mentioned features could be implemented in the scope of this thesis and the list is far from being complete. Rather than being a full featured tool suite, *InternalBlue* should be the base framework for such tools. However, in order to demonstrate the capabilities of the framework, proof of concept implementations for some of the mentioned features including a LMP monitor and injection tool will be provided in Section 5.4. Section 7.2 will discuss features that might be possible to realize but have not yet been implemented due to time constraints.

InternalBlue *is a framework for creating research tools that need access to the Bluetooth controller firmware.*

5.1 REQUIREMENTS

The *InternalBlue* framework is meant to be used for both, aiding the reverse engineering process and later to provide the base for tools to experiment with the firmware and the lower layers of the Bluetooth stack. Therefore, it shall be easily extendable as new knowledge is gained during the reverse engineering process. To accomplish this the framework should offer an abstraction layer on top of the raw Host Controller Interface (HCI) protocol. This abstraction layer should include basic functionality such as:

- send arbitrary HCI commands and receive the matching response,
- read from Random Access Memory (RAM) and Read Only Memory (ROM),

- write to RAM,
- add and remove patches to the ROM, and
- execute machine code inside the firmware.

For convenience *InternalBlue* should also offer high level functionality that builds on top of the aforementioned abstraction layer:

- read and parse firmware data structures (e.g. the `CONNECTION` structure table),
- receive LMP packets, and
- inject LMP packets.

Interacting with the firmware to accomplish the above mentioned functionality requires an active connection to the *Android* Bluetooth stack as well as root access to the file system of the *Android* device. The framework shall run on a *Linux* host which is connected to the *Android* device via Universal Serial Bus (USB).

The Nexmon *framework should allow to write patches in C and embed them into HCD files.*

In addition to the *InternalBlue* framework which will allow direct interaction with the firmware at runtime it also makes sense to extend the *Nexmon* framework so that it can be used to rewrite HCD files. This will enable researchers to write complex patches in the C language rather than directly in assembly. Embedding the patches into the HCD files makes them more permanent compared to runtime patches done with *InternalBlue* because the HCD file is automatically loaded on every reset of the firmware. The *Nexmon* and the *InternalBlue* frameworks may be used in combination, for example a user can write patches in *Nexmon* and interact with them during runtime with *InternalBlue*. *Nexmon* may also be used to write standalone patches for the firmware which do not require interaction during runtime.

InternalBlue *is a Python module which can also run as standalone CLI.*

For implementing the *InternalBlue* framework the Python language [40] shall be used, because it is easy to read and write. The core of the framework shall be provided in form of a Python module which provides the above mentioned abstraction layer as well as additional helper functions. The framework should also include an interactive Command Line Interface (CLI) which allows to quickly access the functionality of the core module as well as helpful debug commands during the reverse engineering process. The CLI should come with the following commands:

- present memory in various different representations (classical hexdump, disassembly, write to binary file),
- write data to memory in different representations (ASCII/hexadecimal string, integer, assembly, read from file),
- search for data in the memory of the firmware,

- execute code snippets in the context of the firmware,
- activate monitor mode for HCI and LMP and forward the received packets to Wireshark [54], and
- inject arbitrary HCI and LMP packets.

5.2 ARCHITECTURE AND DESIGN

The framework described in the previous section can be realized in many different approaches. This section describes the system architecture that was adopted by this thesis. It relies on debug features in the *Android* Bluetooth stack and the Android Debug Bridge (ADB) [2].

5.2.1 *The Android Bluetooth Stack*

The *Android* operating system uses the *Bluedroid* Bluetooth stack which is provided by *Broadcom*. Earlier versions of *Android* (previous to *Android* 4.2) still used the *bluez* stack which is the default Bluetooth stack for *Linux* systems [24].

Figure 17 presents an overview of the *Android* Bluetooth architecture [17]. The diagram shows that, at the lowest layer, the Bluetooth stack is actually separated from vendor specific extensions. In case of the *Nexus 5*, the vendor specific code handles the initialization of the

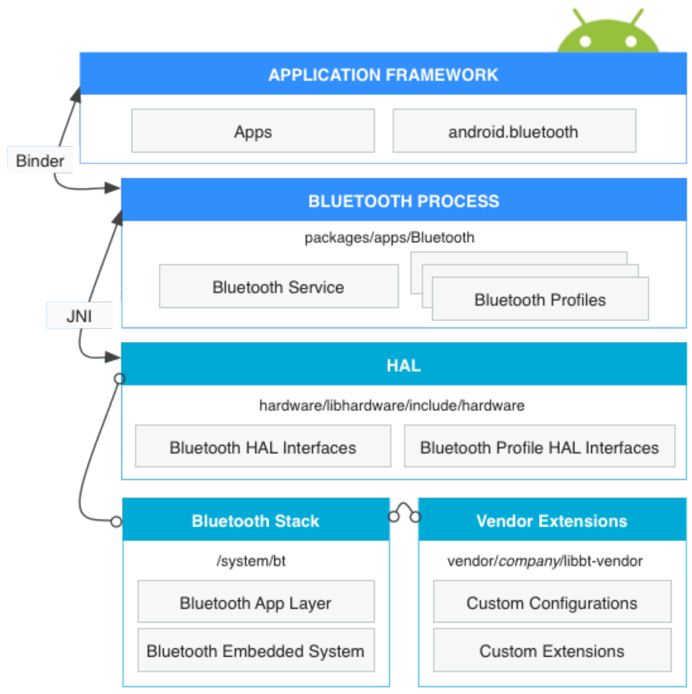


Figure 17: The Bluetooth Architecture of *Android* up to Version 7. [17]

BCM4339 chip including the firmware update process as described in Section 4.6.

Many debug features of the Android Bluetooth stack are not included in release versions.

The Bluetooth stack implements the host side of the HCI, as well as other upper-layer protocols such as Logical Link Control and Adaptation Protocol (L2CAP) and Service Discovery Protocol (SDP). The implementation also provides extensive debugging functionality which can be used to interact directly with the HCI layer. However, most of these debug features are not compiled into the release version of the Bluetooth stack. Section A.1 in the appendix describes the steps to build a custom *Android* Bluetooth stack with enabled debug functionalities.

The Android Bluetooth stack can write a capture file of all HCI transactions with the controller.

A very helpful feature called `HCI snoop log`, writes a capture file of all HCI packets which were transferred between the *Android* host and the Bluetooth controller. Once the feature has been enabled through the *Android* developer settings, a capture file is written to the internal memory of the *Android* device. It uses the *Snoop Packet Capture* format which is specified in RFC 1761 [20]. The file consists of a short file header containing a magic sequence, a version number and the type of the data link which was captured. For the *Nexus 5*, the magic sequence is `btsnoop`, which actually differs from the specification that uses only `snoop`. The version number is set to 1, which is the old version and predecessor to the current version 2 of the file format. The data link type is set to the value 1002. Each HCI packet is encapsulated in a `Snoop Record` which also stores length and time information in a `Record Header`.

A debug build of the Bluetooth stack allows HCI packet monitoring and injection via TCP sockets.

If the *Android* Bluetooth stack is compiled with debugging features enabled, the *Android* system also opens two TCP ports listening on localhost: [14]

- 127.0.0.1:8872 - outputs every HCI packet which is exchanged between the *Android* Bluetooth stack and the Bluetooth controller using the *Snoop* file format, and
- 127.0.0.1:8873 - accepts HCI commands in the format of `Snoop Records` which will be sent to the Bluetooth controller by the *Android* Bluetooth stack.

5.2.2 *System Overview*

This section describes a setup which allows for a close inspection and manipulation of the Bluetooth controller while it is actively used by the *Android* system. Figure 18 shows an overview of the system which consists of an *Android* device with an embedded *Broadcom* Bluetooth controller and a Linux host running the *InternalBlue* framework which is written in Python. The *Android* device is connected to the host via ADB over either USB or a Transmission Control Protocol (TCP) connection.

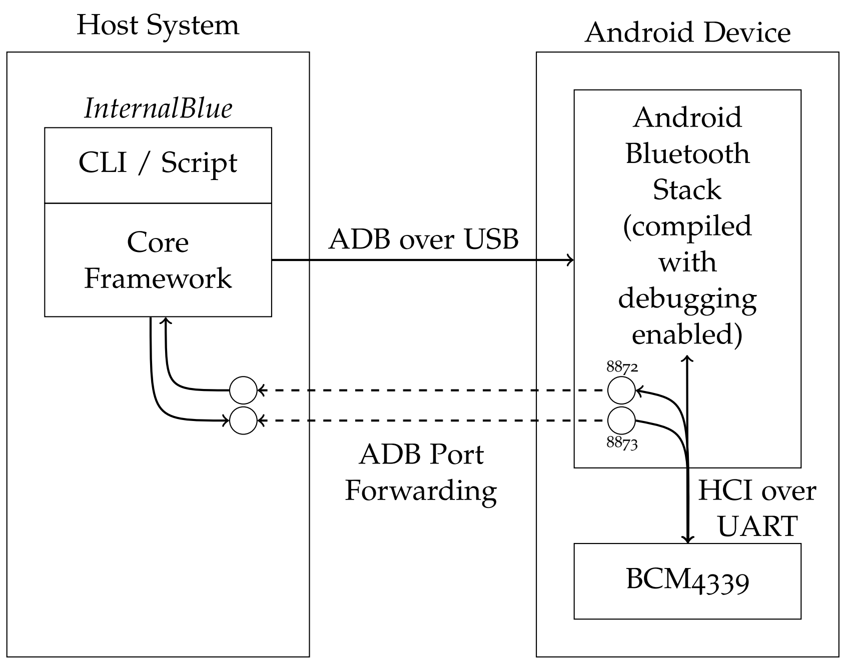


Figure 18: System Overview of *InternalBlue*.

ANDROID DEVICE The *Android* device contains an embedded *Broadcom* Bluetooth controller chip. In case of the *Nexus 5*, this is the *BCM-4339* combined WiFi and Bluetooth controller. The Bluetooth controller connects via an Universal Asynchronous Receiver-Transmitter (UART) interface to the mainboard of the *Android* device. On operating system level the interface is listed as `/dev/ttyHS99`.

The UART device is managed by the *Android* Bluetooth stack which is located at `/system/lib/hw/bluetooth.default.so`. The vendor specific driver is located at `/vendor/lib/libbt-vendor.so`. As explained in Section 4.6, the driver loads the HCD file from `/vendor/firmware/bcm4335c0.hcd` which contains HCI commands that are being send to the controller.

The Bluetooth stack needs to have debug features enabled at compile time in order to offer the HCI debug ports 8872 and 8873. These ports are used by the *InternalBlue* framework to monitor and inject HCI packets exchanged with the controller. Apart from that, Bluetooth is still fully functional on the *Android* device and the user may install and use arbitrary applications to generate a desired usage and traffic pattern.

ANDROID DEBUG BRIDGE The ADB allows developers to debug their apps on the target device and additionally offers many helpful functionalities when working with an *Android* phone. The ADB connection can be established via USB or via a TCP connection. It is possible to forward listening TCP ports from the target device to the

Listing 8: Controlling the *Bluetooth* Subsystem with ADB.

```
# Enable Bluetooth
adb shell 'su -c "service call bluetooth_manager 6"'

# Disable Bluetooth
adb shell 'su -c "service call bluetooth_manager 8"'
```

Listening ports on the Android system can be forwarded to the host with ADB.

host. This feature is used to forward the HCI debug ports so that they can be accessed by the framework on the host. In order to support multiple connected *Android* devices simultaneously each instance of *InternalBlue* will actually forward the ports to random ports on the host. Via ADB it is also possible to invoke shell commands on the device. Listing 8 shows how this can be used to enable or disable the Bluetooth subsystem directly from inside the framework.

The connections to the TCP debug ports are managed by separate threads in the framework.

INTERNALBLUE FRAMEWORK Figure 19 shows the data and control flow of the framework. Solid arrows show the flow of data and dashed arrows indicate the use of functionality or the invocation of functions. The framework starts two separate threads which connect to each of the forwarded TCP ports. Blocking queues are used to pass data between threads in a synchronized manner. The `receive thread` reads captured HCI packets from port 8872 and stores them into the receive queues for the core framework and the `transmit thread`. It also forwards the packets to the `crash detector` class in order to check if the packet belongs to a stack trace message indicating that the chip crashed (see Section 4.3.2). The `transmit thread` waits for HCI commands from the core framework and sends them out via the 8873 injection port. It then reads from the receiver queue to find the matching command response and passes it back to the framework.

The core framework implements functions for reading, writing and patching memory as well as for code execution on the chip. It also contains setup routines for configuring the connected *Android* device and starting the threads. This framework functionality can be used by external Python scripts by simply importing the corresponding module and instantiating the frameworks main class. *InternalBlue* also includes a CLI with an interactive command loop and an extendable set of analysis commands.

5.3 IMPLEMENTATION

InternalBlue is implemented in Python and utilizes the *Pwntools* [39] library. This library contains a lot of helpful functionality including:

- a logging system,

- easy User Interface (UI) development through the *readline* [51] interface,
- an ADB interface,
- an *ARM* assembler and disassembler,
- a *QEMU* interface, and
- many helper functions for binary operations.

The *InternalBlue* framework is organized in multiple files as shown in Figure 20. The purpose and content of each file is explained in the following paragraphs:

CORE.PY This file contains the main class of the framework which includes the thread functions for the receive and send thread. It also implements methods to setup the TCP connection to the *Android* Bluetooth stack via ADB port forwarding as explained in Section 5.2.2.

HCI.PY The `hci.py` file contains classes and functions to parse and craft HCI packets. It reuses code from the *python-btsnoop* project [58] but also adds *Broadcom* specific functionality such as the `crash detector` class.

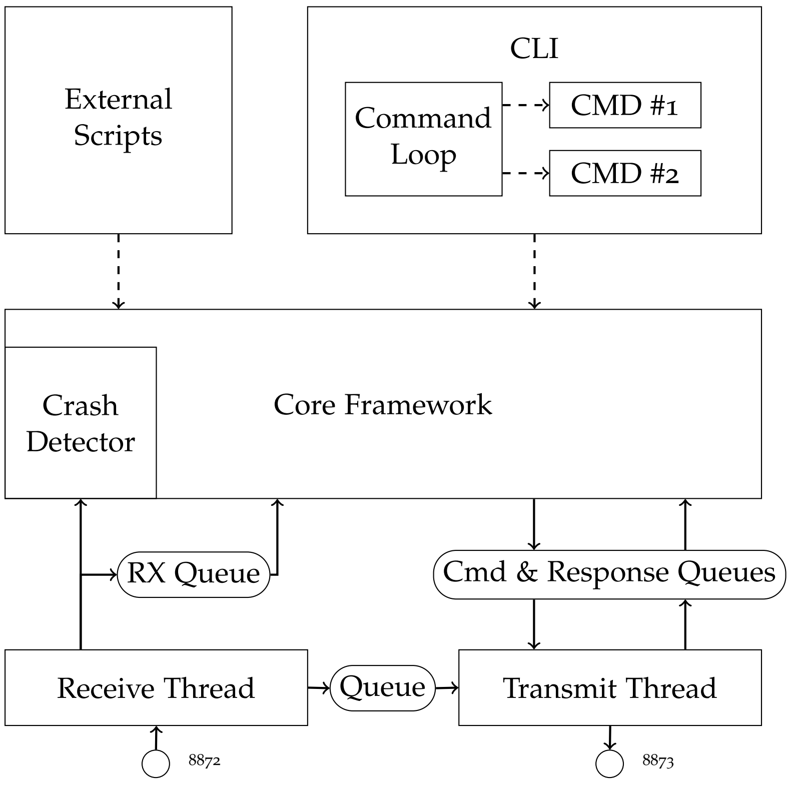


Figure 19: Logical Structure of *InternalBlue*.

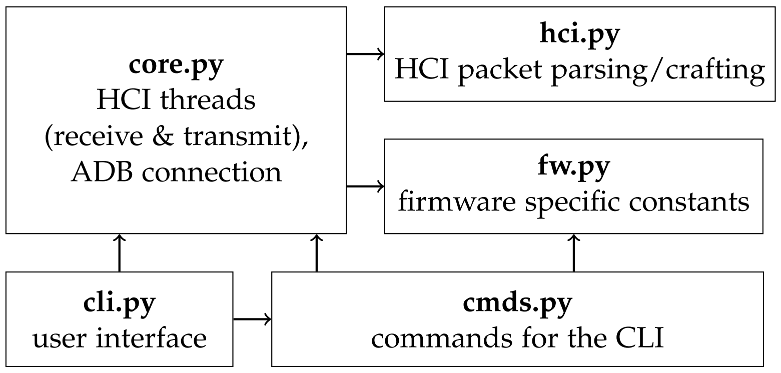


Figure 20: File Structure of *InternalBlue* (arrows indicate dependencies).

FW.PY All firmware specific data such as address offsets are collected in the `fw.py` file. Later versions of the framework will provide multiple copies of this file in order to target different firmware and chip versions.

CLI.PY This file is meant to be executed by the user in order to start an interactive CLI. It creates an instance of the framework and enters a command loop which is implemented with the *readline* interface. Commands entered by the user are matched to the corresponding `Cmd` subclass in the `cmds.py` file and dispatched accordingly.

CMDS.PY All available CLI commands are defined in this file by creating subclasses of the `Cmd` class. The following section explains how the command classes have to be structured and how the CLI can be extended with additional commands.

5.3.1 *Extendable Command Interface*

In InternalBlue *each CLI command is represented by a separate class.*

Listing 9 shows the implementation of the `hexdump` command which is part of *InternalBlue*. This command reads data from the firmware memory and displays it to the user in hexadecimal notation (see Figure 22). The general structure of a command is now explained in detail with the help of the `hexdump` example. Each command is a subclass of `Cmd` and has to have the following class attributes defined:

- `keywords`: A list of keywords which may be used by the user to invoke the command. It must contain at least one keyword and none of the keywords in all subclasses of `Cmd` must be the same.
- `description`: This string will be shown in the command list when the user types `help` or `<cmd-keyword> -h`.
- `parser`: This attribute is optional. If the command takes arguments from the commandline, it may choose to use an instance

Listing 9: Implementation of hexdump (*InternalBlue* CLI).

```
class CmdHexdump(Cmd):
    keywords = ['hexdump', 'hd']
    description = "Display a hexdump of specified memory region."
    parser = argparse.ArgumentParser(
                        prog=keywords[0],
                        description=description,
                        epilog="Aliases: " + ", ".join(keywords))
    parser.add_argument("--length",
                        "-l",
                        type=auto_int,
                        default=256,
                        help="Length of the hexdump.")
    parser.add_argument("address",
                        type=auto_int,
                        help="Start address of the hexdump.")

    def work(self):
        args = self.getArgs()
        if args == None:
            return True

        dump = self.readMem(args.address, args.length)

        if dump == None:
            return False

        log.hexdump(dump, begin=args.address)
        return True
```

of the argparse [55] Python module and store it in the `parser` attribute. This attribute will also be picked up by the `help` command together with the `keywords` and `description` attributes.

The functionality of a command is located in its `work()` method.

Each subclass should also overwrite the method `work` which implements the actual command. It must return either `True` or `False` depending if the command succeeded or failed. To accomplish its work, the command has access to inherited attributes and methods of the `Cmd` class:

- `internalblue`: This is the core framework instance and allows access to the *Broadcom* Bluetooth firmware. It provides methods to send and receive HCI packets as well as many higher level functionality to analyze and manipulate the firmware of the Bluetooth controller. Many of the core framework functions also have a corresponding wrapper method defined in the `Cmd` class. In the example the `hexdump` command uses the `readMem()` wrapper to read memory from the firmware.
- `cmdline`: This is the complete string which was entered by the user to invoke the command.
- `getArgs()` The prefered way of parsing complex commands from the command line is to use the `argparse` module as described above. Calling `getArgs()` will then return the parsed arguments from the commandline.

5.3.2 *Nexmon Integration*

Nexmon *is used to generate patched HCD files with patches written in C.*

As mentioned in Section 5.1, the *Nexmon* framework can introduce a few advantages which supplements the abilities of *InternalBlue*:

- convenient development of large patches as they can be written in C,
- patches to HCD files are permanent as they get reapplied on every reset, and
- patches to the HCD file can also work standalone without the need of a debug Bluetooth stack or the connection to the *InternalBlue* framework.

The internal structure and usage of the *Nexmon* framework has already been widely covered in various publications [44–46] and therefore this section only covers the overall workflow and a few peculiarities with respect to HCD files.

Figure 21 shows the process of creating a patched HCD file from an original `bcm4335c0.hcd` file and patches written in C:

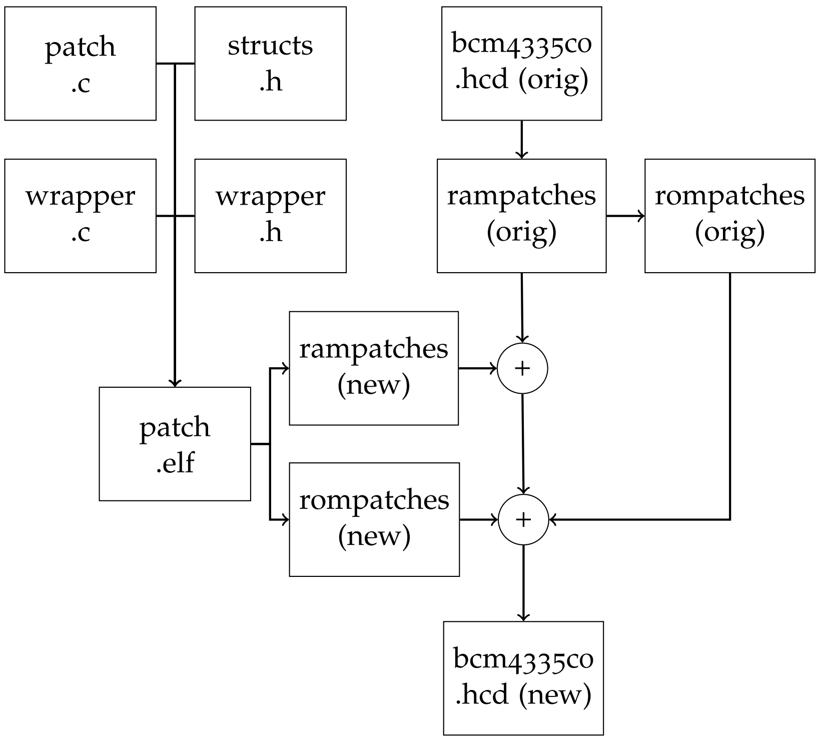


Figure 21: Creation of a Patched HCD File with *Nexmon*.

PATCH.C contains the C code for the patches. The *gcc* plugin which is part of the *Nexmon* framework allows to specify the absolute address in the address space of the firmware for patches and functions.

STRUCTS.H includes structures that were reverse engineered from the firmware (see Section 4.7 and Section 4.9.2).

WRAPPER.C AND WRAPPER.H contain function stubs extracted from the firmware so that they can be called from inside `patch.c`.

PATCH.ELF is the result of compiling the files described above. It contains all patches at their specified addresses.

ORIGINAL HCD FILE is the `bcm4335c0.hcd` file which has been copied from the *Android* system partition. It contains patches and firmware updates from *Broadcom* as described in Section 4.6.

ORIGINAL RAM- AND ROMPATCHES The `Read_RAM` HCI commands contained in the HCD file are extracted to a `rampatches` directory. Furthermore, the special RAM section containing the *Patchram* Type-Length-Value (TLV) list is also extracted and theses ROM patches are stored in the `rompatches` directory.

NEW RAM- AND ROMPATCHES The patches contained in `patch.elf` are now extracted from the binary file and merged with the original patches inside the `rampatches` and `rompatches` directory.

NEW HCD FILE Finally, the patches are combined and merged into a new HCD file which is then pushed onto the *Android* device.

As the *Nexmon* framework is already designed for being extended to new platforms, the major contribution of this thesis is the implementation of a program that extracts and creates HCD files according to Section 4.6.

5.4 APPLICATIONS

This section provides example applications of the *InternalBlue* framework. Due to time constraints only few application ideas could be realized as part of this work. Chapter 7 lists more ideas and brainstorms the potential possibilities of the framework.

5.4.1 *Live Debugging and Analysis*

During the reverse engineering and analysis of the Bluetooth controller firmware it is very valuable to have convenient access to the chip in action. Searching for structures in the memory and observing changes while the chip is actually performing Bluetooth operations can speed up the reverse engineering process vastly.

The *InternalBlue* CLI offers a colorful *readline* UI as can be seen in the screenshot in Figure 22. Executed commands are saved in a history file and it is possible to browse to an earlier command by pressing the arrow-up key as it is known from similar CLIs.

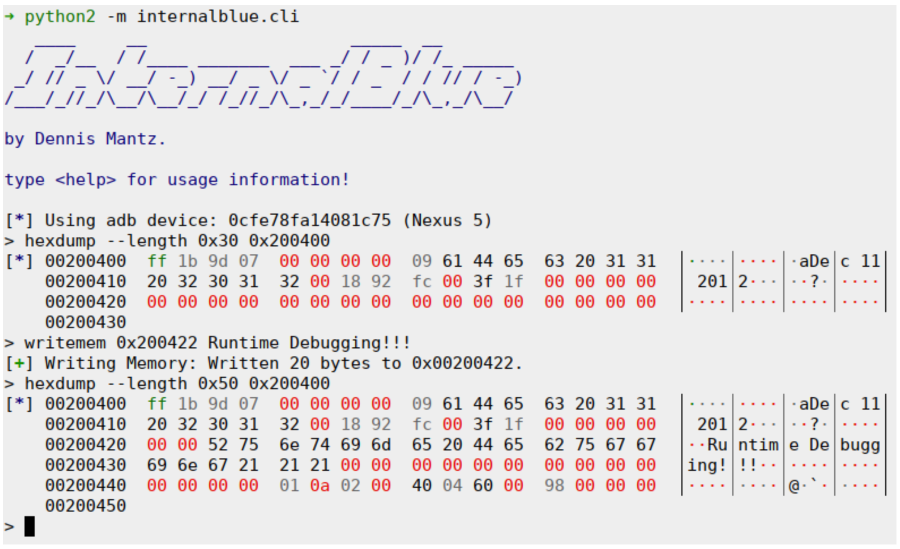
The help command prints all available commands (see Listing 10) and can also be used to get detailed information on specific commands. Alternatively each command can be invoked with the `-h` switch in order to print the help for this particular command.

5.4.2 *Mapping the Firmware's Memory Sections*

One of the first tasks of reverse engineering is to determine the memory layout of the firmware. This can be accomplished through different approaches:

1. iterating over all possible addresses, or
2. recursively extracting addresses from known regions of the firmware starting with the HCD file.

The first approach is rather easy to implement with an early version of *InternalBlue* as it can be done with a brute force Python script which



```
➜ python2 -m internalblue.cli
   ____     __                    _____  __
  /  _/__  / /____ _______  ___ _/ / _ )/ /_ _____
 _/ // _ \/ __/ -_) __/ _ \/ _ `/ / _  / / // / -_)
/___/_//_/\__/\__/_/ /_//_/\_,_/_/____/_/\_,_/\__/

by Dennis Mantz.

type <help> for usage information!

[*] Using adb device: 0cfe78fa14081c75 (Nexus 5)
> hexdump --length 0x30 0x200400
[*] 00200400  ff 1b 9d 07  00 00 00 00  09 61 44 65  63 20 31 31  |····|····|·aDe|c 11|
    00200410  20 32 30 31  32 00 18 92  fc 00 3f 1f  00 00 00 00  | 201|2···|··?·|····|
    00200420  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  |····|····|····|····|
    00200430
> writemem 0x200422 Runtime Debugging!!!
[+] Writing Memory: Written 20 bytes to 0x00200422.
> hexdump --length 0x50 0x200400
[*] 00200400  ff 1b 9d 07  00 00 00 00  09 61 44 65  63 20 31 31  |····|····|·aDe|c 11|
    00200410  20 32 30 31  32 00 18 92  fc 00 3f 1f  00 00 00 00  | 201|2···|··?·|····|
    00200420  00 00 52 75  6e 74 69 6d  65 20 44 65  62 75 67 67  |··Ru|ntim|e De|bugg|
    00200430  69 6e 67 21  21 21 00 00  00 00 00 00  00 00 00 00  |ing!|!!··|····|····|
    00200440  00 00 00 00  01 0a 02 00  40 04 60 00  98 00 00 00  |····|····|@·`·|····|
    00200450
> █
```

Figure 22: Screenshot of the *InternalBlue* CLI.

Listing 10: Output of the `help` Command (*InternalBlue* CLI).

```
> help
disasm          Display a disassembly of a specified region in
                the memory.
dumpmem         Dumps complete memory image into a file.
exec            Writes assembler instructions to RAM and jumps
                there.
exit            Exit the program.
help            Display available commands. Use help <cmd> to
                display command specific help.
hexdump         Display a hexdump of a specified region in
                the memory.
info            Display various types of information parsed
                from live RAM
listen          Dump every received HCI packet on the screen.
log_level       Change the verbosity of log messages.
monitor         Controlling the LMP monitor.
patchrom        Patches 4 byte of data at a specified ROM address.
repeat          Repeat a given command until the user stops it.
searchmem       Search a pattern (string or hex) in the memory
                image.
sendhcicmd      Send an arbitrary hci command to the BT controller
sendlmp         Send LMP packet to another device.
telescope       Display a specified region in the memory and
                follow pointers to valid addresses.
writeasm        Writes assembler instructions to a specified
                memory address.
writemem        Writes data to a specified memory address.
```

Listing 11: Results of the Memory Map Script.

```
> python extract_memory_mapping.py mem_mapping.bin
Found section: 0x0      - 0x90000   size:  0x90000 byte /  576 kB
Found section: 0xd0000  - 0xd8000   size:   0x8000 byte /   32 kB
Found section: 0xe0000  - 0x1f0000  size: 0x110000 byte / 1088 kB
Found section: 0x200000 - 0x228000  size:  0x28000 byte /  160 kB
Found section: 0x260000 - 0x268000  size:   0x8000 byte /   32 kB
Found section: 0x280000 - 0x2a0000  size:  0x20000 byte /  128 kB
Found section: 0x318000 - 0x320000  size:   0x8000 byte /   32 kB
Found section: 0x324000 - 0x360000  size:  0x3c000 byte /  240 kB
Found section: 0x362000 - 0x362100  size:    0x100 byte /    0 kB
Found section: 0x363000 - 0x363100  size:    0x100 byte /    0 kB
Found section: 0x600000 - 0x600800  size:    0x800 byte /    2 kB
Found section: 0x640000 - 0x640800  size:    0x800 byte /    2 kB
Found section: 0x650000 - 0x650800  size:    0x800 byte /    2 kB
Found section: 0x680000 - 0x800000  size: 0x180000 byte / 1536 kB
```

Discovering the memory layout by try and error does work but is rather slow.

iteratively reads memory from all addresses starting at address `0x0`. If the transaction returns a result, the address is valid and mapped. Otherwise, if the firmware does not respond to the command or send a stack dump HCI event (see Section 4.3.2) the address is not mapped and must be skipped. In the latter case the Bluetooth controller must first be reset before the next address can be queried. For this reason the approach is very slow and it is not feasible to apply it to the complete address space of the firmware. However, as it turns out all interesting memory sections are actually mapped at very low addresses which is why the method still works.

Listing 11 shows the results extracted from the output file of the brute force script. A manual inspection of each found section reveals that some of them are false positives. The addresses in those sections can be read without crashing the chip but will always return the value `0x00` and are not actually mapped memory. Section 4.2 contains the final memory mapping table without false positives.

The set of all used addresses in the firmware gives a good estimate of the memory map.

The second approach as stated above is to recursively analyze known parts of the firmware and extract addresses. These addresses can then be read from the chip and analyzed again. All valid addresses which are cross-referenced from the starting point can be found and turned into a memory map by this method. This method can also be used to verify that all interesting memory sections have been found by the brute forcing approach.

5.4.3 *LMP Monitor and Injection Mode*

As the LMP operates exclusively beneath the HCI protocol layer it is usually impossible to monitor or directly manipulate it from the host. However, by using the knowledge gained in Section 4.9 and

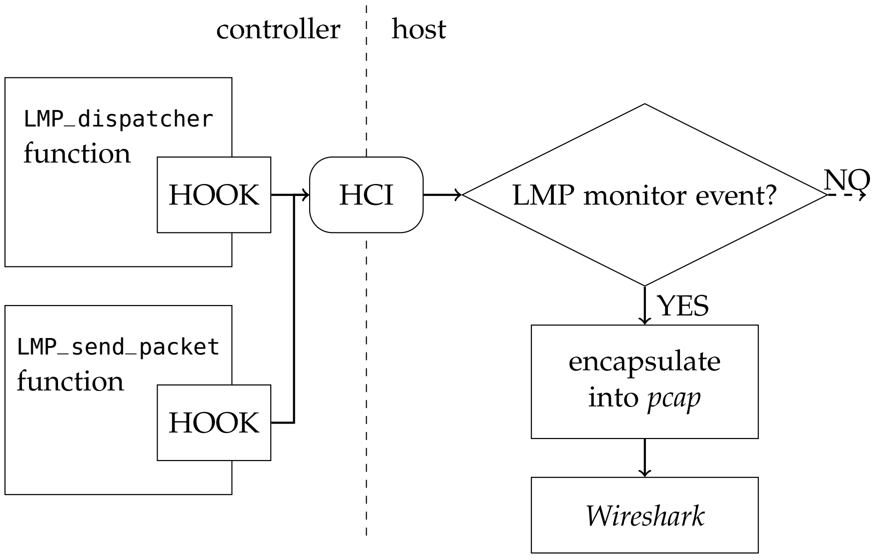


Figure 23: Receive Path of the LMP Monitor Mode.

the patching capabilities of *InternalBlue*, the *BCM4339* firmware can be modified to relay LMP packets to the host and provide an interface for injecting arbitrary LMP packets into an established Bluetooth connection.

The receive path of the monitor mode patch is illustrated in Figure 23. It starts at the dispatcher function which processes incoming LMP packets by invoking the responsible handler function for the specific LMP `opcode`. By hooking the dispatcher function, the received packet can be copied and transferred to the host via a custom HCI event packet. In addition, the function which is responsible for sending LMP packets to a remote device is also hooked in order to have access to those packets on the host as well. At the host side, the monitor feature installs a HCI receiver callback which listens for the custom HCI events, processes them and passes the LMP packets to a *Wireshark* instance. As *Wireshark* is not able to dissect LMP packets by default, it is necessary to install a custom dissector plugin for this protocol. Fortunately, the *Ubertooth* project [37] includes such a plugin which, after minor modifications and bugfixes, can be used for this purpose. Figure 24 shows a screenshot of the captured LMP packets inside *Wireshark*.

LMP packets can be intercepted at the dispatcher function and send to the host via HCI events.

Injection of arbitrary LMP packets into an established Bluetooth connection works by invoking the `send_LMP_packet()` function inside the firmware which in turn will enqueue the packet into the queue for outgoing LMP packets. For this the payload and a small machine code stub is written to the RAM and invoked with the `Launch_RAM` HCI command. The stub copies the payload into a freshly allocated `BLOC` buffer and calls the `send_LMP_packet()` function with a pointer to the corresponding connection structure. It also makes sure that the

LMP injection can be done by invoking the `send_LMP_packet()` function.

Transaction ID (TID) bit in the LMP header is set correctly based on whether the device is master or slave in the connection.

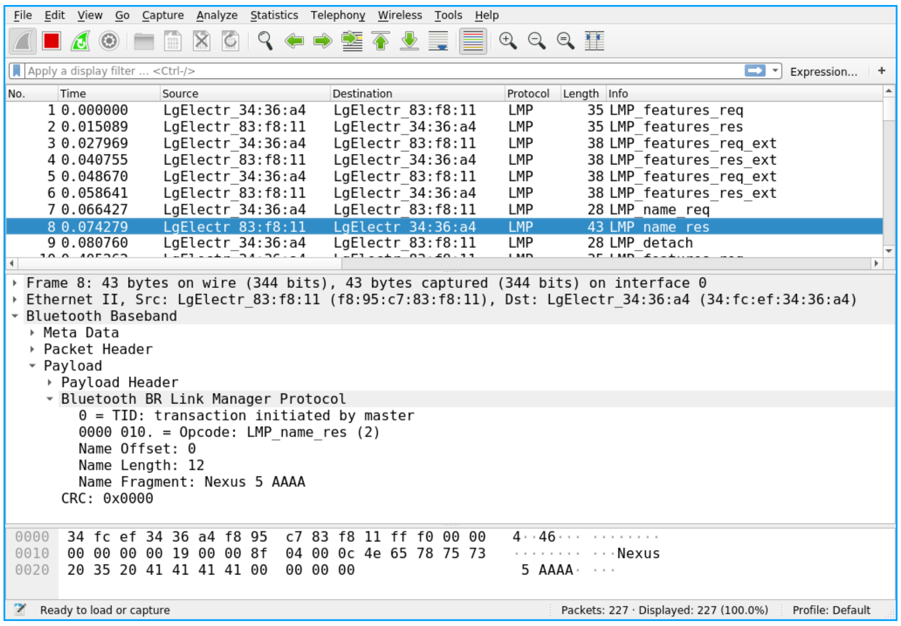
5.4.4 *Random Number Generator*

The *BCM4339* has hardware support for generating random numbers. As true randomness is important for security related functions such as link key generation and encryption, a hardware-backed Random Number Generator (RNG) is important. Table 6 shows the memory-mapped Input-Output (IO) registers which are used for the random number generation. The generation can be triggered by writing a '1' to the least significant bit of the control register. Via the status register it is possible to determine if the new random 4-byte value is already available in the register at `0x31400C`.

To assure the quality of generated random numbers the National Institute for Standards and Technology (NIST) has published a test suite [42]. A small Python script which uses the *InternalBlue* framework can be used to extract 100 MB of random samples from the chip. Figure 25 shows a histogram of the sample bytes from which can be seen that the RNG produces uniformly distributed values. The promising output is confirmed by the results of the NIST test suite which are presented in Listing 18 in the appendix. The results show that the RNG which is built into the *BCM4339* outputs good random numbers which are conform with the NIST requirements.

Table 6: Memory Mapped IO for the RNG.

Address	Description
`0x314000`	Unused
`0x314004`	Control Register
`0x314008`	Status Register
`0x31400C`	Output Register


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...

```
No.  Time        Source                Destination           Protocol  Length Info
   1 0.000000    LgElectr_34:36:a4     LgElectr_83:f8:11     LMP           35 LMP_features_req
   2 0.015089    LgElectr_83:f8:11     LgElectr_34:36:a4     LMP           35 LMP_features_res
   3 0.027969    LgElectr_34:36:a4     LgElectr_83:f8:11     LMP           38 LMP_features_req_ext
   4 0.040755    LgElectr_83:f8:11     LgElectr_34:36:a4     LMP           38 LMP_features_res_ext
   5 0.048670    LgElectr_34:36:a4     LgElectr_83:f8:11     LMP           38 LMP_features_req_ext
   6 0.058641    LgElectr_83:f8:11     LgElectr_34:36:a4     LMP           38 LMP_features_res_ext
   7 0.066427    LgElectr_34:36:a4     LgElectr_83:f8:11     LMP           28 LMP_name_req
   8 0.074279    LgElectr_83:f8:11     LgElectr_34:36:a4     LMP           43 LMP_name_res
   9 0.080760    LgElectr_34:36:a4     LgElectr_83:f8:11     LMP           28 LMP_detach
```

```
› Frame 8: 43 bytes on wire (344 bits), 43 bytes captured (344 bits) on interface 0
› Ethernet II, Src: LgElectr_83:f8:11 (f8:95:c7:83:f8:11), Dst: LgElectr_34:36:a4 (34:fc:ef:34:36:a4)
⌄ Bluetooth Baseband
  › Meta Data
  › Packet Header
  ⌄ Payload
    › Payload Header
    ⌄ Bluetooth BR Link Manager Protocol
        0 = TID: transaction initiated by master
        0000 010. = Opcode: LMP_name_res (2)
        Name Offset: 0
        Name Length: 12
        Name Fragment: Nexus 5 AAAA
      CRC: 0x0000
```

```
0000  34 fc ef 34 36 a4 f8 95  c7 83 f8 11 ff f0 00 00   4··46··· ········
0010  00 00 00 00 19 00 00 8f  04 00 0c 4e 65 78 75 73   ········ ···Nexus
0020  20 35 20 41 41 41 41 00  00 00 00                   5 AAAA· ···
```

```
Ready to load or capture          Packets: 227 · Displayed: 227 (100.0%)     Profile: Default
```

Figure 24: *InternalBlue* LMP Monitor Mode in Wireshark.

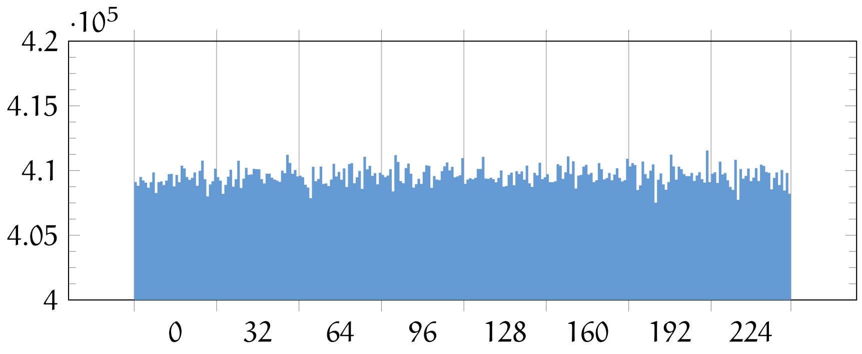


Figure 25: Histogram of the RNG Output (Byte Values). Sample Size: 104857600 Byte (100 MB).

6 EVALUATION

This chapter provides performance measurements of the *InternalBlue* framework and lists several representative use cases which demonstrate its usefulness in a reverse engineering setup.

6.1 PERFORMANCE EVALUATION

As the implemented framework utilizes proprietary, of-the-shelf hardware which was also not meant to be used as development platform, its performance is certainly limited. To estimate the capability of the framework a number of test cases have been implemented and evaluated.

6.1.1 *HCI Communication Speed*

Especially for the memory search functionality it is important how fast the `Read_RAM` operations can be executed. Three test cases shall therefore measure the maximum speed of reading Read Only Memory (ROM) and Random Access Memory (RAM) as well as writing to RAM. The results are summarized in Table 7.

Test case one dumps the main ROM section of the firmware (576 KB) and measures the time. The second test case does the same for the main RAM section (160 KB). As overwriting the complete RAM section is not possible without crashing the chip, the third test case instead does 650 consecutive 251-byte `Write_RAM` operations to the same address. This results in 159 KB written to RAM and ensures that every single operation utilizes the maximum Host Controller Interface (HCI) payload capacity (255 byte minus 4 byte needed for the target address).

Additionally, a fourth test case measures the duration of 650 consecutive one-byte `Write_RAM` operations in order to calculate the latency

Table 7: HCI Performance Evaluation.

Test Case	Test Size	Duration	Speed	Time / Cmd
1. Read ROM	576 KB	135 s	4.25 KB/s	0.058 s
2. Read RAM	160 KB	36 s	4.39 KB/s	0.056 s
3. Write RAM	650 · 251 B	37 s	4.26 KB/s	0.058 s
4. Write RAM	650 · 1 B	37 s	0.02 KB/s	0.058 s

Table 8: Processor Performance Evaluation.

Test Case	Loop Cycles	Duration
5. Single Instruction	1	0.064 s
6. Delay Loop	$134 \cdot 10^6$	28.1 s

The high latency of HCI transactions is the bottleneck for the overall throughput.

overhead of the HCI communication. The last column in Table 7 contains the average time that was needed for a single HCI command to complete. It makes clear that the bottleneck of all test cases is the HCI transport latency.

6.1.2 *Processor Speed*

The *BCM4339* datasheet [6] does not specify the clock speed of the Cortex-M3 processor core. It is also hard to estimate the characteristics of the task scheduler just from static analysis of the firmware. Thus, this section contains a performance test with the aim to determine the maximum processing speed of the *ARM* core.

First the latency of invoking just a single `BX LR` instruction with the `Launch_RAM` HCI command is measured. After that, the main test case executes a delay loop with 134 million (`0x8000000`) cycles and three instructions per iteration. After the loop, the code sends a custom HCI event to the host to signal its completion. The code takes 28.1 seconds to generate this event which results in a processing speed of 14.3 million instructions per second. Table 8 shows all test results. It is noteworthy that in some cases the *Android* system restarted the Bluetooth subsystem during the long-running loop. This is because the loop is actually blocking the thread which handles HCI communication and therefore the *Android* system reasons that the firmware got unresponsive.

Long-running code which is executed through `Launch_RAM` may cause issues.

6.1.3 *Available Memory for Patches*

As mentioned at the beginning of this chapter, the *BCM4339* was not built to be an extendable research platform. Accordingly, the *Patchram* mechanism is rather meant to be used for minor firmware updates and bugfixes instead of large feature upgrades. Therefore it is interesting to know how much memory is available for patches on the chip.

The *Patchram* memory section has a size of 32 KB (see Section 4.2) but the patches from the original `bcm4335c0.hcd` file already occupy 31240 bytes (30.5 KB) of this section. They also use 112 of the 128 available *Patchram* slots which can be used for ROM patches. This means that large additional patches need to replace some or all of the existing patches from the `bcm4335c0.hcd` file.

Most of the available space for patches is already used in the latest firmware version.

Unfortunately, the RAM usage is not easy to determine. One strategy to estimate the amount of free space in the RAM would be to initialize the whole section with a distinctive pattern before it gets used by the firmware. Then the unused space can be calculated by dumping the RAM at runtime and adding up all chunks in which the pattern is still there and was not overwritten by the firmware. However, it is not possible to add patches this early in the boot sequence of the firmware. This means that significant parts of the RAM are already initialized by the firmware once a custom patch can take over the control flow, rendering this method unusable. The reverse engineering efforts have shown that most of RAM is actually used and only small fragments seem to be untouched by the firmware.

To summarize, available memory might become a problem for including sophisticated patches into the firmware. However, there are ways to work around this issue:

- removing the original patches from the `bcm4335c0.hcd` frees the complete 32 KB *Patcham* section and makes all 128 *Patchram* slots available, and
- freeing up RAM can be done through shrinking the space used by the `BLOC` buffers at the cost of possible stability issues.

There exist ways to make space for custom patches.

6.2 REVERSE ENGINEERING USE CASES

The implemented analysis framework, especially the Command Line Interface (CLI), has already proven to be very helpful during the reverse engineering work done in this thesis. This section lists various exemplary use cases.

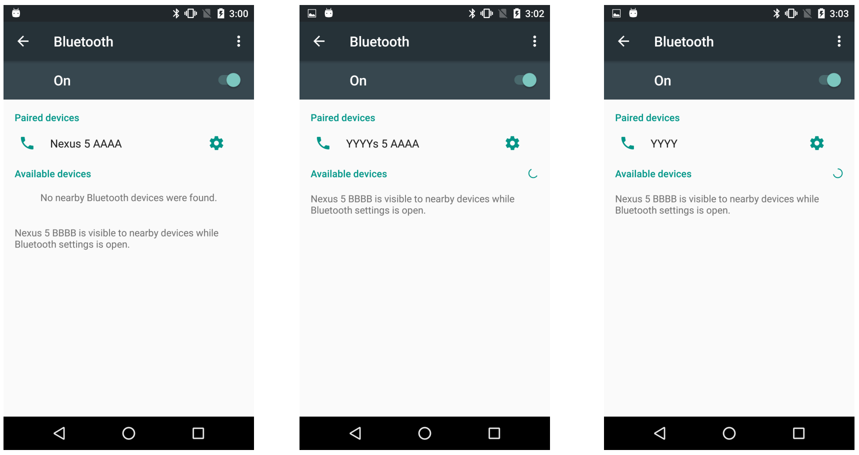
6.2.1 *Bluetooth Device Name*

One interesting target for reverse engineering the firmware is the memory location of the own Bluetooth device name. The device name of a Bluetooth device is sent to a remote device when asked for it via a Link Manager Protocol (LMP) request. It is not hardcoded in the firmware and can be configured dynamically with a HCI command by the host. Therefore, in most phones the user has the option to change the Bluetooth device name in the settings.

The Bluetooth device name can be set through a HCI command.

A simple test setup for gathering initial information about the handling of the device name inside the firmware could include two devices:

1. a *Nexus 5* with attached *InternalBlue* framework, and
2. a second device capable of displaying the device names of surrounding Bluetooth devices. In this case, another *Android* smartphone is used (see Figure 26).


Bluetooth
On
Paired devices
Nexus 5 AAAA
Available devices
No nearby Bluetooth devices were found.
Nexus 5 BBBB is visible to nearby devices while Bluetooth settings is open.

Bluetooth
On
Paired devices
YYYYs 5 AAAA
Available devices
Nexus 5 BBBB is visible to nearby devices while Bluetooth settings is open.

Bluetooth
On
Paired devices
YYYY
Available devices
Nexus 5 BBBB is visible to nearby devices while Bluetooth settings is open.

(a) Initial state (b) After overwriting the device name (c) After overwriting the length field

Figure 26: The Bluetooth Name Reported by the *Nexus 5* is Shown in the *Android* Bluetooth Settings on a Second Device.

Listing 12: Device Name Search (*InternalBlue* CLI).

```
> searchmem Nexus 5 AAAA
[*] Match at 0x0020368c:
[*] 00203680  09 61 53 01  03 00 00 00  00 00 00 00  4e 65 78 75  |.aS.|....|....|Nexu|
    00203690  73 20 35 20  41 41 41 41  00 00 00 00  00 00 00 00  |s 5 |AAAA|....|....|
    002036a0
[*] Match at 0x002178b6:
[*] 002178b0  ef ef ef ef  0d 09 4e 65  78 75 73 20  35 20 41 41  |....|..Ne|xus |5 AA|
    002178c0  41 41 17 03  05 11 0a 11  0c 11 0e 11  12 11 15 11  |AA..|....|....|....|
    002178d0
```

The Bluetooth name of the *Nexus 5* is set to "`Nexus 5 AAAA`" via the *Android* settings and the device is set to be discoverable by other Bluetooth devices. Figure 26a shows a screenshot from the second device which correctly displays the name of the *Nexus 5*.

In the second step, the *InternalBlue* CLI is used to find the string "`Nexus 5 AAAA`" in the memory of the *Nexus 5*. Listing 12 shows that two locations in the RAM match the given string: `0x20368C` and `0x2178B6`. To quickly determine which of these locations is used when reporting the name to a remote device, the CLI can be used to overwrite the names with different patterns (see Listing 13).

Listing 13: Overwrite Device Names (*InternalBlue* CLI).

```
> writemem 0x0020368c XXXX
[+] Writing Memory: Written 4 bytes to 0x0020368c.

> writemem 0x002178b6 YYYY
[+] Writing Memory: Written 4 bytes to 0x002178b6.
```

Listing 14: Overwrite Length Field (*InternalBlue* CLI).

```
> writemem --hex 0x002178b4 05
[+] Writing Memory: Written 1 bytes to 0x002178b4.
```

As can be seen in Figure 26b, the second device shows the name as "`YYYYs 5 AAAA`" which indicates that the second location is actually used when responding to remote name requests. After doing more experiments it can also be found that the length of the device name is stored at address `0x2178B4`, two bytes before the actual name starts. This one-byte field contains the length of the name increased by one. Listing 14 and Figure 26c show that the length of the reported name can be manipulated by writing to this field.

The Bluetooth device name can be manipulated through the InternalBlue *CLI.*

6.2.2 *Pairing Procedure on the LMP Layer*

The pairing procedure of a Bluetooth connection is done on the LMP layer and therefore not visible when monitoring the HCI traffic with the *btsnoop log* developer feature of *Android*. However, by using the LMP monitor mode implemented in *InternalBlue* it is possible to follow along with the pairing protocol. For a demonstration two *Nexus 5* smartphones are being paired with each other:

- device **A**: "Nexus 5 AAAA" [`f8:95:c7:83:f8:11`], and
- device **B**: "Nexus 5 BBBB" [`34:fc:ef:34:36:a4`].

Device **A** will be connected to the *InternalBlue* framework and capturing the LMP traffic of the pairing procedure. The other device will initiate the pairing process from the *Android* Bluetooth settings menu. Figure 24 in Section 5.4.3 already showed a screenshot of the capture process in Wireshark. The full packet trace is listed in Section A.4 in the appendix. The purpose of the so-called LMP transactions is specified in Volume 2, Part C of the Bluetooth Core specification [15]. This section does only a coarse analysis of the captured packet trace. The trace is divided into three parts which is also noticeable by examining the timestamps:

1. packets 1 to 91: Secure Simple Pairing (SSP) procedure,
2. packets 92 to 149: first connection, and
3. packets 150 to 225: second connection.

The first part contains the SSP procedure as explained in Section 2.4. After the devices exchange their respective features and capabilities, both devices request the names of each other (packets 7 and 45).

The LMP monitor mode enables the analysis of pairing procedures based on real-world traffic.

The actual SSP procedure starts with packet 44 (`LMP_IO_Capability_req`). In the following `LMP_encapsulated_header` and `LMP_encapsulated_payload` packets both devices exchange their public key. What follows is a Diffie-Hellman key exchange and the procedure is finished with the `LMP_DHkey_Check` packets (80, 85). Now both devices share a secret key and can authenticate each other (`LMP_au_rand` and `LMP_sres` in packet 87 to 90). The devices are now paired and after about 4 seconds of no interaction the connection is already terminated by Device **A** which sends a `LMP_detach` in packet 91.

The second part shows another connection setup which was again manually triggered by the user in the *Android* settings menu. It can be seen that both devices authenticate each other (packets 126 and 127) and start an encrypted communication channel. Even though the packets are now sent encrypted, the payload is still visible in Wireshark as *InternalBlue* extracts the packets before the encryption and after the decryption happens in the firmware. The connection is then terminated again with 149.

The reason for the connection teardown is presumably that the connection between the two devices does not have a purpose. This can be changed by enabling Bluetooth tethering in device **A**. As can be seen from the timestamps, this causes the second connection (starting at packet 150) to stay alive until it is manually terminated by the user from device **A** in packet 227.

7 DISCUSSION

This chapter discusses the results of Chapter 6 and gives an outlook on open tasks and ideas for future work.

7.1 DISCUSSION

After Section 6.2 listed several use cases in which the implemented *InternalBlue* framework shows its strengths, it is now also necessary to discuss its weaknesses and shortcomings.

STABILITY Although it was shown in Section 4.6 that it is possible to patch arbitrary parts of the firmware there are still restrictions when modifying the internals of a complex and highly interdependent system such as a Bluetooth controller. Being able to observe and modify the firmware while it is part of a fully functional *Android* Bluetooth stack has many advantages. However, any modification of the controller firmware that causes unexpected behavior being observed by the host system will cause the *Android* system to reset the chip.

Firmware modification bear the risk of being unstable.

CAPABILITIES If compared to a platform like the *Ubertooth*, the presented framework falls short in giving access to the lowest protocol layer which handles raw Bluetooth frames on the physical channel. This layer is expected to be handled by a separate real-time processing core as indicated in the block diagram of the chip (Section 4.1, Figure 5). It is possible that this core is not entirely implemented in hardware and might run code which can also be manipulated. Schulz et al. [44] have shown that the *D11* core in the *BCM4339* can be reprogrammed. The *D11* core is the real-time core which processes WiFi frames on the physical layer and therefore has a similar role as the Bluetooth real-time core.

The physical protocol layer is not yet reverse engineered and can therefore not be manipulated with InternalBlue.

SECURITY Eventually, the bottom line question is: Does the framework fill the gap in the arsenal of security related Bluetooth tools? The answer to this is: Not yet! *InternalBlue* adds important and novel features such as Link Manager Protocol (LMP) monitoring and injection but still lacks the potential to monitor traffic on the physical layer and in promiscuous mode. However, the potential of the *BCM4339* chip is certainly not reached yet and future reverse engineering advances and firmware modifications may produce the missing pieces in the tool arsenal.

InternalBlue *has not yet used all of its potential and needs further development.*

7.2 FUTURE WORK

The work presented in this thesis should be considered to be the foundation for future work and follow-up projects. This section lists open tasks and also includes interesting feature and application ideas for the *InternalBlue* framework.

7.2.1 *Missing Functionality and Optimizations*

The framework will most likely work with more Broadcom *controllers once their firmware is reverse engineered and included into* InternalBlue.

A major open task is porting the framework to other members of the *Broadcom* Bluetooth-WiFi chip series. Interesting targets include the *BCM4358* which is embedded in the *Nexus 6P* and the *BCM43430a1* used with the *Raspberry Pi 3*. In theory all *Broadcom* specific Host Controller Interface (HCI) commands should be implemented on every chip and therefore the basic functionality of the framework such as reading, writing and executing memory does not need any chip specific handling. However the addresses of interesting data structures and functions are most likely different between different chips and also between different firmware versions of the same chip. Functionality inside the framework which relies on such addresses and assumptions needs to be aware of the chip version. As explained in Section 5.3, firmware specific code and constants are located in a separate Python file called `fw.py`, thus simplifying the task of making the framework compatible with multiple firmware and chip versions. In addition, there exist powerful reverse engineering tools such as the *IDA Pro BinDiff* [9] plugin which help in such situations. Nevertheless, porting still requires major reverse engineering efforts and thus *InternalBlue* currently only supports the *BCM4339*.

Some components of InternalBlue *can be implemented more efficiently by executing code directly inside the firmware.*

As suggested in Section 6.1.1, searching for a sequence of bytes in the entire Random Access Memory (RAM) is quite slow due to the high latency of the HCI debug interface. The search functionality could instead be rewritten to execute the search directly on the firmware. A search would cause the framework to write a short code snippet to the memory of the chip which compares the search sequence to every position in the RAM and reports matches by sending custom HCI events back to the host. This removes the HCI bottleneck and should increase the search speed by orders of magnitude. Similar performance optimizations might be possible for other features as well.

7.2.2 *Open Reverse Engineering Tasks*

Most of the firmware of the *BCM4339* is still undiscovered as only the basic structure and some specific areas of the firmware have been reverse engineered as part of this thesis.

For the scope of the thesis, all code related to Bluetooth Low Energy (BLE) was not reverse engineered because of time constraints. Although BLE was not in focus of this work it is still a relevant technology, especially for many modern Internet of Things (IoT) devices which rely entirely on BLE communication. Reverse engineering the respective parts in the firmware and extending the framework accordingly can increase its range of applications considerably.

BLE operations should be included into the framework to cover most of the IoT devices.

Debug strings and template messages have been found in several locations of the firmware during the reverse engineering phase of the thesis. Often the code paths referencing these strings were disabled by different variables in RAM which indicates that the firmware can be put into a debug mode in which it generates various debug messages. Reverse engineering these code paths and tracing the guard variables might reveal this debug mode. Additionally, there exist multiple patents by *Broadcom* [31, 32] which indicate that there might be a remote debugging feature implemented in their chips.

The BCM4339 *might feature a (remote) debug mode.*

As already mentioned in the previous section, the lowest protocol layer is probably handled by another processor core with real-time capabilities. In block diagrams, it is either called `BT PHY` [19] or `Bluetooth Baseband Core` [5]. Unfortunately, the firmware running on this core could not be located in the address space accessible by the Cortex-M3. However, with additional time and effort it is not unlikely that the firmware can be found and even modified, just like it was the case with the *D11* core in the WiFi part of the chip [44]. This would open the door for a new category of possible tools such as a promiscuous mode to monitor raw Bluetooth traffic just like the *Ubertooth*.

Access to the Bluetooth Baseband Core would yield many more interesting applications for the framework.

7.2.3 *Firmware Emulation and Fuzzing*

The *LuaQEMU* project [33] has already be mentioned in Section 3.2 as a powerful tool to compensate for the missing debugging capabilities of an embedded firmware. A promising idea is to combine *LuaQEMU* with *InternalBlue* by adding a new feature referred to as tracepoints to the framework. A tracepoint is similar to a breakpoint in the way, that it can be placed on an arbitrary instruction inside the firmware. Internally, the tracepoint is implemented as a hook, which dumps the exact processor and memory state at the current instruction and sends it via HCI events back to the framework. This state can then be fed into *LuaQEMU* to emulate and debug the execution of the code path following the tracepoint.

Firmware emulation could be a feature included with InternalBlue.

With a working emulation solution, not only debugging but also fuzzing of certain code paths can be accomplished. From a security perspective it would be interesting to fuzz various different protocol handlers such as the code which processes incoming LMP packets. However, for efficient fuzzing it is important to detect memory corruptions, even if they do not directly lead to a crash. Section 4.7.3 explained how the firmware manages dynamic buffer allocation through the `BLOC` data structures. By patching the initialization routine of these data structures it is possible to add guard spaces between the buffers which can be used during the emulation to detect linear out-of-bound read or write instructions. As most protocol related buffers use the `BLOC` data structures, this technique might be sufficient to find memory corruption vulnerabilities in the firmware. Thanks to the reverse engineering work done in this thesis it is also feasible to conduct a classic static code analysis of the firmware. A good starting point for this would be the LMP handler functions which can be considered as attack surface as they directly handle untrusted input data.

Emulation, fuzzing and static code analysis might reveal security vulnerabilities inside the firmware.

7.2.4 *Security Analysis Toolkit for Bluetooth*

As mentioned in the motivation section of this thesis, Bluetooth is a rather old technology that has been extended multiple times through the years. Over time features such as pairing and encryption mechanisms have been deprecated due to security weaknesses and replaced by modern alternatives. From the user perspective it is usually not obvious which technology is used when pairing two Bluetooth devices. However, by leveraging the LMP injection capabilities of the *InternalBlue* framework it is possible to build a tool which audits other Bluetooth devices by enumerating proposed pairing and encryption capabilities. It would also be possible to do further analysis such as

InternalBlue *can be used to audit remote Bluetooth devices.*

- measure the entropy of random numbers used in authentication and pairing procedures,
- inspect the behavior of the remote device to unexpected packets and deviations from the specified protocols,
- test how the remote device reacts to downgrade attacks, and
- fingerprint the behavior of the remote device to identify the vendor of its Bluetooth controller.

8 CONCLUSIONS

This thesis documents the development of *InternalBlue*, a novel analysis and research platform for lower Bluetooth protocol layers. The framework is based on the *Broadcom BCM4339* Bluetooth controller embedded in the *Nexus 5*, whose firmware has been reverse engineered and documented in the process.

In Chapter 1 and 3 it was found that there is a shortage of openly available research tools which allow for efficient experimenting on Bluetooth protocol layers beneath the Host Controller Interface (HCI). A prominent example is the Link Manager Protocol (LMP) which has been introduced in Chapter 2. On the one hand, high-level tools based on regular Bluetooth dongles are not able to access this protocol layer directly as it is solely handled inside the firmware of the Bluetooth controller. On the other hand, Software-Defined Radios (SDRs) and dedicated platforms such as the *Ubertooth* have to cope with Bluetooth's frequency hopping characteristics and would need to implement considerable parts of the specification in order to become reliable tools for LMP experiments. *InternalBlue* closes the gap and allows access to the internals of a commercial Bluetooth controller. It benefits from the ability to reuse and modify the existing Bluetooth stack implemented in the firmware of the controller.

Lower protocol levels of Bluetooth are difficult to access. Therefore researchers need better tools to analyze these layers.

Commercial, off-the-shelf hardware can be repurposed through firmware modifications.

To achieve this, the firmware had to be extracted from the controller and go through a thorough reverse engineering process. The results are cataloged and described in Chapter 4. The initial challenge of dumping the firmware has been solved by using vendor specific HCI commands which allow reading and writing memory inside the address space of the controller. In the next step, the memory sections were identified and correlated with the information from the datasheet. After the Interrupt Vector Table (IVT) was located, reverse engineering the `reset` interrupt handler provided initial insight into the initialization routine of the firmware. By progressively exposing basic functions and data structures it was possible to finally understand the complex firmware architecture and internal processes such as memory allocation, task scheduling and synchronization, event handling and connection management.

The internal operations of the firmware have been reverse engineered.

The patching mechanism could be dissected from the firmware update procedure and combined with the knowledge from the reverse engineering process to build the *InternalBlue* framework in Chapter 5. Furthermore, the framework has been used as the basis for a LMP monitoring and injection tool and Chapter 6 has shown more applications in which *InternalBlue* can be applied. Finally, Chapter 7 has

The firmware update process revealed patching capabilities which can be used to manipulate the firmware.

discussed the shortcomings and open tasks of the project and also gave an outlook to more interesting ideas for future tools based on *InternalBlue*.

Overall, the addition of a new tool to the utility belt of security researchers can only be a good thing and advance the state of Bluetooth security. Furthermore, reverse engineering the firmware of a commercial Bluetooth controller paves the way for open reviews by the security community and may reveal vulnerabilities that have been hidden up to this point.

A APPENDIX

The appendix contains sections which are not directly related to the core research topic of the thesis but might still be relevant for readers. It also contains detailed experiment results that have been to large for the main thesis and have been relocated to the appendix for readability.

A.1 BUILDING A CUSTOM ANDROID BLUETOOTH STACK

In order to build a custom Bluetooth stack with enabled debugging features for *Android* it is necessary to setup a build environment for the *Android Open Source Project (AOSP)*. In recent versions of *Android* the Bluetooth stack can also be built as standalone project. However, this guide explains the build process for the *Nexus 5* running *Android* 6.0 which requires the complete *AOSP* build setup.

The build process of an *Android* ROM needs around 100 GB of storage for the source tree and the compiled outputs. Using an *Amazon EC2* [25] instance has a few advantages:

- convenient selection of the correct *Ubuntu* version matching the targeted *Android* version,
- a very fast network connection for downloading the *Android* repository, and
- the option to select fast storage and high-power CPU instances.

This section explains the build process for the Bluetooth stack on *Android* 6.0.1 [27] for a *Nexus 5* (code name: *hammerhead*) smartphone. Most steps are taken from an online tutorial from Sony [29]. For the most recent *Android* version (at the time of the writing of this thesis the most recent *Android* version is 8.1) there exist preconfigured *Amazon Web Services (AWS)* instances [3] which come with the latest *AOSP* repository and all necessary tools installed. However, for a significantly older version, it is easier to setup a fresh instance with an older *Ubuntu* version to get all necessary tools (Java, gcc, etc.) in their correct versions.

The instance should have high network throughput, fast storage (dedicated SSD) and a decent amount of virtual CPU cores for the build process. A reasonable option would be the *i3.large* instance which comes with a dedicated NVME SSD.

After connecting to the instance via SSH the necessary tools have to be installed as shown in Listing 15. The listing also shows how to

Listing 15: Preparing the *AWS* Instance for the *AOSP* Build.

```
# Install all necessary tools for the build
sudo dpkg --add-architecture i386
sudo apt update && sudo apt upgrade
sudo apt install openjdk-7-jdk gcc-multilib g++ bison git zip
sudo apt install g++-multilib gperf libxml2-utils make zlib1g-dev
    :i386
mkdir ~/bin
curl http://commondatastorage.googleapis.com/git-repo-downloads/
    repo > ~/bin/repo
chmod a+x ~/bin/repo
export PATH=~/bin:$PATH

# Prepare the SSD
sudo mkfs.ext4 /dev/nvme0n1
sudo mount /dev/nvme0n1 mnt
chown -R ubuntu:ubuntu mnt/
mkdir mnt/android
cd mnt/android/

# Clone the Android repository
repo init -u https://android.googlesource.com/platform/manifest -
    b android-6.0.1_r81
repo sync
```

Listing 16: Building the *AOSP* Bluetooth Stack.

```
source build/envsetup.sh
lunch aosp_hammerhead-userdebug
cd system/bt/
bdroid_CFLAGS='-DBT_NET_DEBUG=TRUE' mma -j4
```

format and mount the NCME drive that comes with the *AWS* instance. Now the *AOSP* repository of the corresponding *Android* version can be downloaded onto the SSD with the help of the *repo* tool.

Then it is possible to build the Bluetooth stack with all necessary dependencies. The `lunch` script can also be invoked without any arguments to select the build target from an interactive list. The debugging features of the Bluetooth stack are enabled by setting the preprocessor define `BT_NET_DEBUG=TRUE`. The build script is called `mma` and takes an optional argument `-j` to specify the number of CPU cores to use in parallel. It should be chosen according to the selected *AWS* instance. Listing 16 shows the commands for building just the Bluetooth stack of the *AOSP* repository.

After the build process is done, the `bluetooth.default.so` shared library can be found in `/home/ubuntu/mnt/android/out/target/product/hammerhead/system/lib/hw/bluetooth.default.so` and pushed onto the smartphone via Android Debug Bridge (ADB). To overwrite the exist-

Listing 17: Installing the new Bluetooth Stack.

```
adb push bluetooth.default.so /sdcard/bluetooth.default.so
adb shell 'su -c "mount -o remount,rw /system"'
adb shell 'su -c "cp /sdcard/bluetooth.default.so /system/lib/hw/
    bluetooth.default.so"'
adb shell 'su -c "chmod 644 /system/lib/hw/bluetooth.default.so"'
adb shell 'su -c "chown root:root /system/lib/hw/bluetooth.
    default.so"'
```

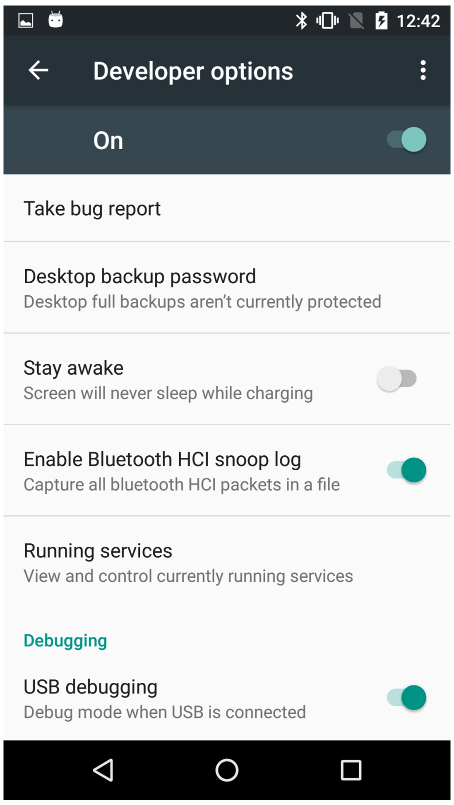

12:42
Developer options
On
Take bug report
Desktop backup password
Desktop full backups aren't currently protected
Stay awake
Screen will never sleep while charging
Enable Bluetooth HCI snoop log
Capture all bluetooth HCI packets in a file
Running services
View and control currently running services
Debugging
USB debugging
Debug mode when USB is connected

Figure 27: Screenshot of the HCI Snoop Log Setting.

ing library on the Android `system` partition it must first be remounted in order to make it writable. It is also important to verify that the new library is actually set to be executable, otherwise Bluetooth will not work on the device. The commands are summarized in Listing 17. Finally, the `HCI snoop log` feature has to be enabled in the developer settings of the Android phone as shown in the screenshot in Figure 27.

A.2 USED SOFTWARE

The following software products in their respective versions were used to accomplish the work of this thesis:

- ADB 1.0.40 [2]
- Android 6.0.1 [27]
- IDA Pro 7.0 [30]
- NIST STS 2.1.2 [42]
- Pwntools 3.12.0 [39]
- Python 2.7.15 [40]
- Radare2 2.6.9 [59]
- Wireshark 2.6.1 [54]

A.3 TEST RESULTS OF THE NIST TEST SUITE FOR THE RNG

This section contains the results (Listing 18) of the National Institute for Standards and Technology (NIST) test suite [42] for the Random Number Generator (RNG) inside the *BCM4339* Bluetooth controller.

Listing 18: Results of the NIST Test Suite for RNGs.

```
------------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------------------
   generator is <dump.bin>
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
 11   6   7  13  10   9   9  13  14   8  0.678686     99/100     Frequency
 10   7  10  12   6  10  10  11  11  13  0.911413     99/100     BlockFrequency
  9  12  10  10  12   4  11   9  14   9  0.699313     99/100     CumulativeSums
  9  12   8  12   8   9  11  10   8  13  0.955835     99/100     CumulativeSums
 11  13   8  10   6   6  12  12   7  15  0.455937    100/100     Runs
 11  12   8   9  11  13   6   9   9  12  0.897763    100/100     LongestRun
 14   4   8   8  10  21   6  14   9   6  0.006196     97/100     Rank
 17  11  11   9  14   6   9   7   9   7  0.319084     98/100     FFT
  4  14  10  20  10   6  11   3  12  10  0.008266    100/100     NonOverlappingTemplate
  7  14  13  12   8  13   7   5  13   8  0.366918     97/100     NonOverlappingTemplate
  8   8   9  12   9  10  11   8  10  15  0.883171     99/100     NonOverlappingTemplate
  9   4   9  13  14   7  13  14   7  10  0.304126     98/100     NonOverlappingTemplate
 13  13   9   9   7  13   8   9  10   9  0.883171     99/100     NonOverlappingTemplate
  7   8  10  10  13  13  12   7   9  11  0.867692    100/100     NonOverlappingTemplate
  7  11  12   8  17   8  10   8   8  11  0.534146     99/100     NonOverlappingTemplate
 12   5  14  13   5  14   7  10   6  14  0.137282     98/100     NonOverlappingTemplate
  9   6   8   8  10  13   9   9  14  14  0.657933     99/100     NonOverlappingTemplate
 12  12  12   8   7   7  12  12   7  11  0.816537     99/100     NonOverlappingTemplate
  4   8  13   7  13   8  20  12   8   7  0.026948    100/100     NonOverlappingTemplate
  9  12  12  13   4  14   7  15   9   5  0.162606     98/100     NonOverlappingTemplate
 10  12   8   8   7  17   8   7  12  11  0.455937    100/100     NonOverlappingTemplate
  6   8  11   8  11  17  16   8   7   8  0.171867     99/100     NonOverlappingTemplate
 10  16   6  14  10  11   5   5  12  11  0.191687     99/100     NonOverlappingTemplate
  8   8   8  10  13   8  16   6   9  14  0.401199     99/100     NonOverlappingTemplate
 14   8   9  13  10   6   7  15   8  10  0.494392    100/100     NonOverlappingTemplate
 11  11  12   3  11  14   7   6  17   8  0.090936     99/100     NonOverlappingTemplate
  4   7  12   8  10  10  15  11  12  11  0.494392     99/100     NonOverlappingTemplate
 13   6  12  12   9  10   9  13   7   9  0.798139     94/100  *  NonOverlappingTemplate
  8   9  11   2   5  23  15   8  11   8  0.000474     98/100     NonOverlappingTemplate
 11   4  14   7  11  11   8  13  13   8  0.437274    100/100     NonOverlappingTemplate
 12  12  10   8   6  10   7  12  11  12  0.867692     99/100     NonOverlappingTemplate
 17   9   5  10   4  15   6  14  11   9  0.048716     99/100     NonOverlappingTemplate
 10   5  11  11  13  16  12   6  10   6  0.289667     99/100     NonOverlappingTemplate
 13   9   9   9  10   9   8   9  13  11  0.971699     98/100     NonOverlappingTemplate
```

6	9	8	11	13	14	8	8	9	14	0.616305	100/100		NonOverlappingTemplate
8	13	10	8	13	10	9	9	16	4	0.350485	100/100		NonOverlappingTemplate
11	13	10	6	13	11	13	8	7	8	0.719747	100/100		NonOverlappingTemplate
9	8	9	9	12	8	14	9	14	8	0.816537	99/100		NonOverlappingTemplate
8	5	12	19	15	5	7	8	11	10	0.037566	100/100		NonOverlappingTemplate
11	9	11	11	13	8	6	10	8	13	0.867692	97/100		NonOverlappingTemplate
13	6	10	9	6	13	8	9	12	14	0.574903	100/100		NonOverlappingTemplate
9	9	4	10	22	15	10	7	8	6	0.004981	99/100		NonOverlappingTemplate
14	7	11	5	12	6	9	11	13	12	0.474986	100/100		NonOverlappingTemplate
11	14	13	9	7	7	8	9	13	9	0.739918	98/100		NonOverlappingTemplate
10	10	13	10	9	6	10	14	11	7	0.816537	98/100		NonOverlappingTemplate
11	14	8	5	12	12	9	13	5	11	0.437274	98/100		NonOverlappingTemplate
2	12	14	11	10	8	10	13	11	9	0.350485	100/100		NonOverlappingTemplate
6	3	11	11	9	17	9	9	12	13	0.153763	100/100		NonOverlappingTemplate
13	9	13	8	13	6	12	6	12	8	0.574903	99/100		NonOverlappingTemplate
11	4	12	11	6	10	11	13	10	12	0.616305	98/100		NonOverlappingTemplate
9	8	6	14	12	13	9	9	10	10	0.816537	100/100		NonOverlappingTemplate
13	11	10	12	11	6	8	16	6	7	0.383827	99/100		NonOverlappingTemplate
11	7	11	6	10	9	10	12	11	13	0.897763	99/100		NonOverlappingTemplate
13	10	10	14	4	11	13	8	12	5	0.319084	100/100		NonOverlappingTemplate
8	8	14	8	12	7	14	9	8	12	0.678686	99/100		NonOverlappingTemplate
6	8	12	10	9	8	11	12	15	9	0.739918	99/100		NonOverlappingTemplate
11	6	12	4	11	6	15	13	14	8	0.171867	99/100		NonOverlappingTemplate
9	12	12	4	13	8	10	5	16	11	0.213309	100/100		NonOverlappingTemplate
8	10	9	14	14	5	13	6	9	12	0.419021	99/100		NonOverlappingTemplate
9	13	11	11	9	13	8	11	8	7	0.911413	100/100		NonOverlappingTemplate
8	11	10	11	8	11	8	3	16	14	0.236810	99/100		NonOverlappingTemplate
6	10	15	10	13	14	8	5	8	11	0.350485	100/100		NonOverlappingTemplate
10	11	6	4	9	14	13	12	17	4	0.051942	98/100		NonOverlappingTemplate
9	11	12	7	10	9	14	8	6	14	0.657933	98/100		NonOverlappingTemplate
13	10	9	12	10	12	11	10	8	5	0.851383	100/100		NonOverlappingTemplate
11	8	9	11	11	8	7	8	10	17	0.595549	98/100		NonOverlappingTemplate
10	12	9	11	11	4	11	17	6	9	0.275709	96/100		NonOverlappingTemplate
12	10	8	14	7	11	9	11	7	11	0.867692	98/100		NonOverlappingTemplate
7	14	9	8	6	10	9	14	12	11	0.657933	100/100		NonOverlappingTemplate
8	10	9	9	10	18	5	11	12	8	0.319084	96/100		NonOverlappingTemplate
8	12	6	8	7	13	13	14	11	8	0.574903	98/100		NonOverlappingTemplate
6	13	12	9	11	5	10	11	12	11	0.719747	100/100		NonOverlappingTemplate
14	8	11	8	9	8	10	6	13	13	0.699313	96/100		NonOverlappingTemplate
8	6	13	8	13	8	13	12	7	12	0.616305	100/100		NonOverlappingTemplate
10	8	8	11	9	18	11	9	7	9	0.474986	98/100		NonOverlappingTemplate
3	6	12	10	6	18	8	12	9	16	0.021999	99/100		NonOverlappingTemplate
10	8	11	14	9	8	9	9	8	14	0.851383	98/100		NonOverlappingTemplate
13	17	11	14	4	8	8	7	7	11	0.129620	98/100		NonOverlappingTemplate
11	11	12	6	10	10	11	9	7	13	0.897763	98/100		NonOverlappingTemplate
7	18	8	15	10	7	10	7	9	9	0.202268	100/100		NonOverlappingTemplate
14	8	6	15	9	9	8	6	10	15	0.289667	94/100	*	NonOverlappingTemplate
8	10	10	20	7	4	8	9	9	15	0.035174	99/100		NonOverlappingTemplate
4	14	10	20	10	6	11	3	12	10	0.008266	100/100		NonOverlappingTemplate
6	5	9	15	11	9	12	14	9	10	0.437274	100/100		NonOverlappingTemplate
4	9	11	14	6	15	9	11	11	10	0.366918	100/100		NonOverlappingTemplate
8	11	11	6	10	9	11	11	14	9	0.897763	99/100		NonOverlappingTemplate
12	16	6	6	18	8	7	6	12	9	0.048716	99/100		NonOverlappingTemplate
9	8	10	11	10	10	13	12	12	5	0.851383	99/100		NonOverlappingTemplate
11	6	10	16	6	11	14	4	9	13	0.153763	100/100		NonOverlappingTemplate
11	9	9	9	10	8	11	11	10	12	0.997823	98/100		NonOverlappingTemplate
5	14	16	10	10	12	6	10	8	9	0.334538	99/100		NonOverlappingTemplate
15	13	4	12	13	5	7	11	12	8	0.181557	100/100		NonOverlappingTemplate
12	8	9	17	10	6	4	7	16	11	0.075719	98/100		NonOverlappingTemplate
9	16	4	9	6	15	10	16	6	9	0.051942	97/100		NonOverlappingTemplate
8	12	9	8	10	13	8	11	13	8	0.911413	100/100		NonOverlappingTemplate
10	13	10	10	8	9	6	14	13	7	0.699313	99/100		NonOverlappingTemplate
10	10	8	10	10	11	7	10	9	15	0.911413	97/100		NonOverlappingTemplate
13	8	11	14	10	10	8	6	12	8	0.759756	99/100		NonOverlappingTemplate
12	9	8	12	13	8	10	12	7	9	0.911413	99/100		NonOverlappingTemplate
7	16	11	9	14	9	4	8	11	11	0.304126	99/100		NonOverlappingTemplate
11	13	11	8	17	7	9	3	13	8	0.137282	99/100		NonOverlappingTemplate
10	7	7	8	9	12	12	17	7	11	0.437274	99/100		NonOverlappingTemplate
11	8	10	10	10	9	12	4	10	16	0.514124	99/100		NonOverlappingTemplate
12	10	8	12	12	11	10	7	11	7	0.935716	100/100		NonOverlappingTemplate
10	8	14	10	11	10	7	10	12	8	0.924076	99/100		NonOverlappingTemplate
15	11	9	11	5	9	11	8	10	11	0.739918	98/100		NonOverlappingTemplate
19	6	6	9	7	18	10	6	7	12	0.010237	100/100		NonOverlappingTemplate
5	8	9	8	17	14	10	14	7	8	0.171867	100/100		NonOverlappingTemplate
10	6	16	7	9	10	8	19	9	6	0.058984	97/100		NonOverlappingTemplate
6	14	10	9	5	11	10	14	11	10	0.574903	99/100		NonOverlappingTemplate
7	9	6	8	10	11	8	12	17	12	0.419021	100/100		NonOverlappingTemplate
12	11	10	8	5	8	10	12	16	8	0.514124	100/100		NonOverlappingTemplate
10	8	10	10	7	11	12	9	11	12	0.983453	100/100		NonOverlappingTemplate
3	4	16	11	12	8	14	15	9	8	0.040108	100/100		NonOverlappingTemplate
10	7	8	11	12	11	10	9	10	12	0.983453	99/100		NonOverlappingTemplate
11	8	6	11	10	15	12	10	6	11	0.657933	99/100		NonOverlappingTemplate
15	9	16	8	10	5	11	9	5	12	0.202268	99/100		NonOverlappingTemplate
9	14	12	18	3	14	8	10	6	6	0.028817	99/100		NonOverlappingTemplate
8	13	7	11	7	14	13	8	6	13	0.474986	99/100		NonOverlappingTemplate
8	7	11	15	10	6	13	10	9	11	0.678686	98/100		NonOverlappingTemplate
20	9	6	9	6	13	13	10	5	9	0.037566	98/100		NonOverlappingTemplate
11	14	7	9	14	9	11	9	10	6	0.719747	98/100		NonOverlappingTemplate
5	12	9	10	9	10	10	12	14	9	0.816537	98/100		NonOverlappingTemplate
14	15	5	12	8	9	10	7	7	13	0.334538	100/100		NonOverlappingTemplate

```
12  8 10  9 12 12 13  8  6 10  0.867692   99/100   NonOverlappingTemplate
 9  8 14  8 12 12  7 11 12  7  0.779188   99/100   NonOverlappingTemplate
 5 10  9 10 11 12  9 15  8 11  0.719747  100/100   NonOverlappingTemplate
 6  6 11 18  6 14  8 12  9 10  0.129620  100/100   NonOverlappingTemplate
 9 10  7 11  9 14  9  8 13 10  0.897763   99/100   NonOverlappingTemplate
 9 12  3 15 16  6 10 11  9  9  0.145326  100/100   NonOverlappingTemplate
12  8 11 10  8 11  7  9 14 10  0.911413   98/100   NonOverlappingTemplate
15 13  7  9  9 15  6  6 11  9  0.319084   99/100   NonOverlappingTemplate
11 10 14  8 13  8  8  8 12  8  0.834308   98/100   NonOverlappingTemplate
 7 11  6  9  8 14 11 13 14  7  0.514124  100/100   NonOverlappingTemplate
13  9 13 11 12 10  5 10  8  9  0.798139   98/100   NonOverlappingTemplate
10  7  6  9  9 17  7 13 12 10  0.366918   98/100   NonOverlappingTemplate
 8 17  7  8 11 10 11 11  5 12  0.366918   99/100   NonOverlappingTemplate
 8  6 10  8 11  8 11 15 11 12  0.739918   99/100   NonOverlappingTemplate
 6 17  9 11 13  4  6 14 12  8  0.085587  100/100   NonOverlappingTemplate
13 12 18  7 10  8 10  9  5  8  0.213309  100/100   NonOverlappingTemplate
16  8  8 13 10  7  6 10 15  7  0.262249  100/100   NonOverlappingTemplate
 9 12  7  9  5 16 14  9  9 10  0.401199   98/100   NonOverlappingTemplate
 6  5 17 10  7 16  9 13  5 12  0.042808  100/100   NonOverlappingTemplate
 9 10  7 14 11 12 13  7  9  8  0.798139   99/100   NonOverlappingTemplate
 6 11 13 12 11 10  9  9  9 10  0.946308  100/100   NonOverlappingTemplate
10  9  8  7 11  8 12  9 14 12  0.883171   99/100   NonOverlappingTemplate
 9  8 11  8  7 10  9 16 11 11  0.759756   99/100   NonOverlappingTemplate
10 15 13  5 11  8  9  8  7 14  0.401199   98/100   NonOverlappingTemplate
 3 11 18  4 13 13 10 12  9  7  0.032923  100/100   NonOverlappingTemplate
 8 13  8 13 10 14  7  8 12  7  0.657933  100/100   NonOverlappingTemplate
 7 11  9 15  6  8 14 13  9  8  0.474986   99/100   NonOverlappingTemplate
 9 13 13  7 10  5 11 10 12 10  0.759756  100/100   NonOverlappingTemplate
 4  4 18 11 11 10 14 10 10  8  0.071177  100/100   NonOverlappingTemplate
10 11 12  7 11 15  5 10 12  7  0.554420   99/100   NonOverlappingTemplate
11 10 11 10 13 14  9 12  4  6  0.494392   96/100   NonOverlappingTemplate
 8 10 10 20  7  4  8  9  9 15  0.035174   99/100   NonOverlappingTemplate
13 10 10 11 14 14  5  8  7  8  0.494392   98/100   OverlappingTemplate
 6 10 10 14 11  9 15  9  9  7  0.637119  100/100   Universal
10  9  6 11 11 11  8 10 15  9  0.834308   98/100   ApproximateEntropy
 9  7 10 16  7 12  5  3  9 10  0.072289   88/88    RandomExcursions
 6 10 12  3  7  9 13 10  7 11  0.258961   87/88    RandomExcursions
10  9  6 11 11  7  9  4 14  7  0.330628   87/88    RandomExcursions
 9 11  6  8 10  8  9  9  8 10  0.964295   87/88    RandomExcursions
 8 16 11  9  8  6  5  8 10  7  0.242986   87/88    RandomExcursions
 9 13  7 10  6 10 10  7  4 12  0.392456   85/88    RandomExcursions
 8  9  8 10  7 12 12  6  9  7  0.788728   87/88    RandomExcursions
 8 11  8  8  6  9  7 15  8  8  0.534146   88/88    RandomExcursions
 7  7 11 11  8  6  6 15  5 12  0.174249   88/88    RandomExcursionsVariant
 6  6 10  9  5 12 11 12  8  9  0.534146   88/88    RandomExcursionsVariant
 6  9  7 10  4 11  8 14  8 11  0.350485   88/88    RandomExcursionsVariant
 5 10 10  7  3 10  8 16  6 13  0.041438   88/88    RandomExcursionsVariant
 3 11  6 11  7 11 19  6  7  7  0.006196   88/88    RandomExcursionsVariant
 5  5 12  8  8 13  9 15  9  4  0.072289   88/88    RandomExcursionsVariant
 5  7  7 10  8 13 15 11  6  6  0.151616   87/88    RandomExcursionsVariant
 5  9 10  9 14  5 12 10  7  7  0.330628   86/88    RandomExcursionsVariant
 5  8  9  8 14  7  7 15  8  7  0.199580   86/88    RandomExcursionsVariant
 6 12  6  7 12  8 13  5  8 11  0.311542   87/88    RandomExcursionsVariant
 7  8 11  9  8  9 11  7  8 10  0.953553   87/88    RandomExcursionsVariant
 4 11 14  6 12 10  8  5  3 15  0.012650   87/88    RandomExcursionsVariant
 8 11  7  8  6  8  9  9  8 14  0.689019   88/88    RandomExcursionsVariant
 6 12  5  8 11  6  7 15  8 10  0.213309   88/88    RandomExcursionsVariant
 6 11  6  7 10 10  8  7  9 14  0.534146   88/88    RandomExcursionsVariant
 4 10  7 12 12  9  7  8 11  8  0.534146   87/88    RandomExcursionsVariant
 4  6  7 14 12  9  8  9 10  9  0.350485   87/88    RandomExcursionsVariant
 3  7 11  6 10  9 13 10 11  8  0.330628   88/88    RandomExcursionsVariant
14 10  3 18  6 11  8 12  8 10  0.071177  100/100   Serial
12  7 12 13 12  8 13  7  8  8  0.739918  100/100   Serial
 8 13 16 13  6 15  6 12  5  6  0.066882   98/100   LinearComplexity


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
The minimum pass rate for each statistical test with the exception of the
random excursion (variant) test is approximately = 96 for a
sample size = 100 binary sequences.

The minimum pass rate for the random excursion (variant) test
is approximately = 84 for a sample size = 88 binary sequences.

For further guidelines construct a probability table using the MAPLE program
provided in the addendum section of the documentation.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

A.4 LMP CAPTURE OF A DEVICE PAIRING PROCEDURE

In Section 6.2.2 the *InternalBlue* framework was used to monitor the pairing procedure of two devices on the Link Manager Protocol (LMP) layer. This section contains the full packet trace generated by Wireshark [54]. The Bluetooth addresses of the devices have been replaced by **A** and **B** for better readability:

- **A**: the monitoring device and
- **B**: the remote device which initiated the pairing process.

No.	Time	Src	Dest	Proto	Len	Info
1	0.000000	B	A	LMP	35	LMP_features_req
2	0.015089	A	B	LMP	35	LMP_features_res
3	0.027969	B	A	LMP	38	LMP_features_req_ext
4	0.040755	A	B	LMP	38	LMP_features_res_ext
5	0.048670	B	A	LMP	38	LMP_features_req_ext
6	0.058641	A	B	LMP	38	LMP_features_res_ext
7	0.066427	B	A	LMP	28	LMP_name_req
8	0.074279	A	B	LMP	43	LMP_name_res
9	0.080760	B	A	LMP	28	LMP_detach
10	0.405262	B	A	LMP	35	LMP_features_req
11	0.424144	A	B	LMP	35	LMP_features_res
12	0.437391	B	A	LMP	32	LMP_version_req
13	0.447994	A	B	LMP	32	LMP_version_res
14	0.455999	B	A	LMP	38	LMP_features_req_ext
15	0.464920	A	B	LMP	38	LMP_features_res_ext
16	0.465427	B	A	LMP	38	LMP_features_req_ext
17	0.466269	A	B	LMP	38	LMP_features_res_ext
18	0.466846	B	A	LMP	27	LMP_host_connection_req
19	0.474867	A	B	LMP	28	LMP_accepted
20	0.475913	A	B	LMP	29	LMP_packet_type_table_req
21	0.477129	A	B	LMP	27	LMP_setup_complete
22	0.479610	B	A	LMP	29	LMP_packet_type_table_req
23	0.480291	A	B	LMP	30	LMP_accepted_ext
24	0.481621	B	A	LMP	31	LMP_not_accepted_ext
25	0.483058	B	A	LMP	42	LMP_set_AFH
26	0.485874	B	A	LMP	33	LMP_channel_classification_req
27	0.486715	A	B	LMP	38	LMP_channel_classification
28	0.487933	B	A	LMP	27	LMP_setup_complete
29	0.489026	A	B	LMP	28	LMP_max_slot
30	0.490103	A	B	LMP	28	LMP_max_slot_req
31	0.491208	A	B	LMP	27	LMP_auto_rate
32	0.493730	A	B	LMP	27	LMP_timing_accuracy_req
33	0.497615	B	A	LMP	28	LMP_max_slot
34	0.500145	B	A	LMP	28	LMP_max_slot_req
35	0.501144	A	B	LMP	28	LMP_accepted
36	0.502149	B	A	LMP	27	LMP_auto_rate
37	0.503329	B	A	LMP	28	LMP_accepted
38	0.504500	B	A	LMP	27	LMP_timing_accuracy_req
39	0.505928	A	B	LMP	29	LMP_timing_accuracy_res
40	0.507215	B	A	LMP	29	LMP_timing_accuracy_res
41	0.511701	B	A	LMP	27	LMP_clkoffset_req
42	0.512730	A	B	LMP	29	LMP_clkoffset_res
43	0.539288	B	A	LMP	29	LMP_supervision_timeout
44	0.541258	B	A	LMP	31	LMP_IO_Capability_req
45	0.550427	A	B	LMP	28	LMP_name_req
46	0.557892	B	A	LMP	43	LMP_name_res
47	0.560394	A	B	LMP	31	LMP_IO_Capability_res

No.	Time	Src	Dest	Proto	Len	Info
48	0.570879	B	A	LMP	30	LMP_encapsulated_header
49	0.571384	A	B	LMP	28	LMP_accepted
50	0.571954	B	A	LMP	43	LMP_encapsulated_payload
51	0.572500	A	B	LMP	28	LMP_accepted
52	0.573947	B	A	LMP	43	LMP_encapsulated_payload
53	0.577894	A	B	LMP	28	LMP_accepted
54	0.578854	B	A	LMP	43	LMP_encapsulated_payload
55	0.579351	A	B	LMP	28	LMP_accepted
56	0.646454	A	B	LMP	30	LMP_encapsulated_header
57	0.668393	B	A	LMP	28	LMP_accepted
58	0.669584	A	B	LMP	43	LMP_encapsulated_payload
59	0.670738	B	A	LMP	28	LMP_accepted
60	0.672099	A	B	LMP	43	LMP_encapsulated_payload
61	0.673288	B	A	LMP	28	LMP_accepted
62	0.674615	A	B	LMP	43	LMP_encapsulated_payload
63	0.676844	B	A	LMP	28	LMP_accepted
64	0.678185	A	B	LMP	43	LMP_Simple_Pairing_Confirm
65	0.732345	B	A	LMP	43	LMP_Simple_Pairing_Number
66	0.734211	A	B	LMP	28	LMP_accepted
67	0.735655	A	B	LMP	43	LMP_Simple_Pairing_Number
68	0.739479	B	A	LMP	28	LMP_accepted
69	0.913226	B	A	LMP	29	LMP_power_control_req
70	0.914722	A	B	LMP	29	LMP_power_control_res
71	0.934822	A	B	LMP	29	LMP_power_control_req
72	0.951976	B	A	LMP	29	LMP_power_control_res
73	1.439153	A	B	LMP	29	LMP_power_control_req
74	1.453204	B	A	LMP	29	LMP_power_control_res
75	1.675060	B	A	LMP	29	LMP_power_control_req
76	1.683550	A	B	LMP	29	LMP_power_control_res
77	3.826482	B	A	LMP	28	LMP_preferred_rate
78	4.082310	A	B	LMP	29	LMP_power_control_req
79	4.104436	B	A	LMP	29	LMP_power_control_res
80	4.184424	B	A	LMP	43	LMP_DHkey_Check
81	4.586623	A	B	LMP	28	LMP_preferred_rate
82	5.091327	B	A	LMP	29	LMP_power_control_req
83	5.101243	A	B	LMP	29	LMP_power_control_res
84	5.476823	A	B	LMP	28	LMP_accepted
85	5.481388	A	B	LMP	43	LMP_DHkey_Check
86	5.488334	B	A	LMP	28	LMP_accepted
87	5.492018	B	A	LMP	43	LMP_au_rand
88	5.493713	A	B	LMP	31	LMP_sres
89	5.494867	A	B	LMP	43	LMP_au_rand
90	5.499472	B	A	LMP	31	LMP_sres
91	9.679624	A	B	LMP	28	LMP_detach
92	18.059380	B	A	LMP	35	LMP_features_req
93	18.072026	A	B	LMP	35	LMP_features_res
94	18.082736	B	A	LMP	32	LMP_version_req
95	18.087847	A	B	LMP	32	LMP_version_res
96	18.101080	B	A	LMP	38	LMP_features_req_ext
97	18.111140	A	B	LMP	38	LMP_features_res_ext
98	18.118891	B	A	LMP	38	LMP_features_req_ext
99	18.125387	A	B	LMP	38	LMP_features_res_ext
100	18.133227	B	A	LMP	27	LMP_host_connection_req
101	18.150592	A	B	LMP	28	LMP_accepted
102	18.151826	A	B	LMP	29	LMP_packet_type_table_req
103	18.153051	A	B	LMP	27	LMP_setup_complete
104	18.154387	B	A	LMP	29	LMP_packet_type_table_req
105	18.155291	A	B	LMP	30	LMP_accepted_ext
106	18.156412	B	A	LMP	31	LMP_not_accepted_ext
107	18.157963	B	A	LMP	42	LMP_set_AFH
108	18.160657	B	A	LMP	33	LMP_channel_classification_req
109	18.161625	A	B	LMP	38	LMP_channel_classification
110	18.162703	B	A	LMP	27	LMP_setup_complete
111	18.164053	A	B	LMP	28	LMP_max_slot
112	18.165315	A	B	LMP	28	LMP_max_slot_req

No.	Time	Src	Dest	Proto	Len	Info
113	18.166600	A	B	LMP	27	LMP_auto_rate
114	18.169218	A	B	LMP	27	LMP_timing_accuracy_req
115	18.177495	B	A	LMP	28	LMP_max_slot
116	18.180351	B	A	LMP	28	LMP_max_slot_req
117	18.181625	A	B	LMP	28	LMP_accepted
118	18.182845	B	A	LMP	27	LMP_auto_rate
119	18.184058	B	A	LMP	28	LMP_accepted
120	18.185101	B	A	LMP	27	LMP_timing_accuracy_req
121	18.186458	A	B	LMP	29	LMP_timing_accuracy_res
122	18.188902	B	A	LMP	29	LMP_timing_accuracy_res
123	18.199201	B	A	LMP	27	LMP_clkoffset_req
124	18.201115	A	B	LMP	29	LMP_clkoffset_res
125	18.216231	B	A	LMP	29	LMP_supervision_timeout
126	18.225382	B	A	LMP	43	LMP_au_rand
127	18.246120	A	B	LMP	31	LMP_sres
128	18.259430	B	A	LMP	28	LMP_encryption_mode_req
129	18.260062	A	B	LMP	28	LMP_accepted
130	18.263170	B	A	LMP	28	LMP_encryption_key_size_req
131	18.263792	A	B	LMP	28	LMP_accepted
132	18.270523	B	A	LMP	43	LMP_start_encryption_req
133	18.273800	A	B	LMP	28	LMP_accepted
134	18.548077	B	A	LMP	29	LMP_power_control_req
135	18.549611	A	B	LMP	29	LMP_power_control_res
136	18.642828	A	B	LMP	29	LMP_power_control_req
137	18.662303	B	A	LMP	29	LMP_power_control_res
138	19.146650	A	B	LMP	29	LMP_power_control_req
139	19.164798	B	A	LMP	29	LMP_power_control_res
140	19.182314	B	A	LMP	29	LMP_power_control_req
141	19.189985	A	B	LMP	29	LMP_power_control_res
142	19.942238	B	A	LMP	28	LMP_preferred_rate
143	20.909652	A	B	LMP	28	LMP_preferred_rate
144	21.967227	B	A	LMP	29	LMP_power_control_req
145	21.979677	A	B	LMP	29	LMP_power_control_res
146	22.043798	A	B	LMP	29	LMP_power_control_req
147	22.055921	B	A	LMP	29	LMP_power_control_res
148	22.094705	B	A	LMP	42	LMP_set_AFH
149	22.392042	B	A	LMP	28	LMP_detach
150	38.336991	B	A	LMP	35	LMP_features_req
151	38.353400	A	B	LMP	35	LMP_features_res
152	38.369173	B	A	LMP	32	LMP_version_req
153	38.379398	A	B	LMP	32	LMP_version_res
154	38.394002	B	A	LMP	38	LMP_features_req_ext
155	38.395169	A	B	LMP	38	LMP_features_res_ext
156	38.396465	B	A	LMP	38	LMP_features_req_ext
157	38.397523	A	B	LMP	38	LMP_features_res_ext
158	38.398885	B	A	LMP	27	LMP_host_connection_req
159	38.406192	A	B	LMP	28	LMP_accepted
160	38.407602	A	B	LMP	29	LMP_packet_type_table_req
161	38.408932	A	B	LMP	27	LMP_setup_complete
162	38.410487	B	A	LMP	29	LMP_packet_type_table_req
163	38.411430	A	B	LMP	30	LMP_accepted_ext
164	38.412512	B	A	LMP	31	LMP_not_accepted_ext
165	38.413501	B	A	LMP	42	LMP_set_AFH
166	38.416723	B	A	LMP	33	LMP_channel_classification_req
167	38.417432	A	B	LMP	38	LMP_channel_classification
168	38.418431	B	A	LMP	27	LMP_setup_complete
169	38.419450	A	B	LMP	28	LMP_max_slot
170	38.420701	A	B	LMP	28	LMP_max_slot_req
171	38.421747	A	B	LMP	27	LMP_auto_rate
172	38.423982	A	B	LMP	27	LMP_timing_accuracy_req
173	38.432215	B	A	LMP	28	LMP_max_slot
174	38.435181	B	A	LMP	28	LMP_max_slot_req
175	38.436349	A	B	LMP	28	LMP_accepted
176	38.437974	B	A	LMP	27	LMP_auto_rate
177	38.438941	B	A	LMP	28	LMP_accepted

No.	Time	Src	Dest	Proto	Len	Info
178	38.440117	B	A	LMP	27	LMP_timing_accuracy_req
179	38.442716	A	B	LMP	29	LMP_timing_accuracy_res
180	38.443914	B	A	LMP	29	LMP_timing_accuracy_res
181	38.444914	B	A	LMP	27	LMP_clkoffset_req
182	38.447010	A	B	LMP	29	LMP_clkoffset_res
183	38.456956	B	A	LMP	29	LMP_supervision_timeout
184	38.464291	B	A	LMP	43	LMP_au_rand
185	38.495776	A	B	LMP	31	LMP_sres
186	38.504249	B	A	LMP	28	LMP_encryption_mode_req
187	38.505543	A	B	LMP	28	LMP_accepted
188	38.507990	B	A	LMP	28	LMP_encryption_key_size_req
189	38.509143	A	B	LMP	28	LMP_accepted
190	38.515321	B	A	LMP	43	LMP_start_encryption_req
191	38.519170	A	B	LMP	28	LMP_accepted
192	38.909171	A	B	LMP	29	LMP_power_control_req
193	38.914150	B	A	LMP	29	LMP_power_control_req
194	38.915809	A	B	LMP	29	LMP_power_control_res
195	38.917960	B	A	LMP	29	LMP_power_control_res
196	39.413838	A	B	LMP	29	LMP_power_control_req
197	39.427893	B	A	LMP	29	LMP_power_control_req
198	39.441579	A	B	LMP	29	LMP_power_control_res
199	39.449709	B	A	LMP	29	LMP_power_control_res
200	41.175125	A	B	LMP	28	LMP_preferred_rate
201	42.215815	B	A	LMP	28	LMP_preferred_rate
202	42.559520	A	B	LMP	29	LMP_power_control_req
203	42.580892	B	A	LMP	29	LMP_power_control_res
204	43.108283	B	A	LMP	29	LMP_power_control_req
205	43.121566	A	B	LMP	29	LMP_power_control_res
206	59.370363	A	B	LMP	36	LMP_sniff_req
207	59.385889	B	A	LMP	36	LMP_sniff_req
208	59.388366	B	A	LMP	29	LMP_not_accepted
209	59.390656	B	A	LMP	36	LMP_sniff_req
210	59.392169	A	B	LMP	28	LMP_accepted
211	59.424681	A	B	LMP	35	LMP_sniff_subrating_req
212	59.886512	B	A	LMP	35	LMP_sniff_subrating_req
213	59.899891	B	A	LMP	31	LMP_not_accepted_ext
214	60.187846	B	A	LMP	35	LMP_sniff_subrating_req
215	60.202005	A	B	LMP	35	LMP_sniff_subrating_res
216	60.440768	A	B	LMP	29	LMP_power_control_req
217	60.882877	B	A	LMP	29	LMP_power_control_req
218	60.892291	A	B	LMP	29	LMP_power_control_res
219	60.915759	B	A	LMP	29	LMP_power_control_res
220	60.957424	A	B	LMP	27	LMP_unsniff_req
221	61.383683	B	A	LMP	29	LMP_power_control_req
222	61.396784	A	B	LMP	29	LMP_power_control_res
223	61.408650	B	A	LMP	28	LMP_accepted
224	64.260217	B	A	LMP	29	LMP_power_control_req
225	64.274172	A	B	LMP	29	LMP_power_control_res
226	70.713803	B	A	LMP	42	LMP_set_AFH
227	71.269490	A	B	LMP	28	LMP_detach

BIBLIOGRAPHY

[1] *A Look at Inner Workings of Joycon and Nintendo Switch.* URL: https://github.com/dekuNukem/Nintendo_Switch_Reverse_Engineering/issues/41 (visited on July 19, 2018).

[2] *Android Debug Bridge (ADB).* URL: https://developer.android.com/studio/command-line/adb (visited on July 19, 2018).

[3] *Android Open Source Project ROM Builder (HVM).* URL: https://aws.amazon.com/marketplace/pp/B01AOKYCZY (visited on July 19, 2018).

[4] Nitay Artenstein. *Broadpwn: Remotely Compromising Android and iOS via a Bug in Broadcom's Wi-Fi Chipsets.* 2017. URL: https://blog.exodusintel.com/2017/07/26/broadpwn/ (visited on July 19, 2018).

[5] *BCM2045 Datasheet.* Broadcom. URL: https://html.alldatasheet.com/html-pdf/175091/BOARDCOM/BCM2045/774/2/BCM2045.html (visited on July 19, 2018).

[6] *BCM4339: Single-Chip 5G WiFi IEEE 802.11ac MAC/Baseband/Radio with Integrated Bluetooth 4.1 and FM Receiver.* 002-14784. Rev. *G. Cypress Semiconductor Corporation. Oct. 2016.

[7] Gal Beniamini. *Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 1).* 2017. URL: https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html (visited on July 19, 2018).

[8] Gal Beniamini. *Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 2).* 2017. URL: https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_11.html (visited on July 19, 2018).

[9] *BinDiff.* zynamics. URL: https://www.zynamics.com/bindiff.html (visited on July 19, 2018).

[10] Andrea Bittau, Mark Handley, and Joshua Lackey. "The Final Nail in WEP's Coffin." In: *Security and Privacy, 2006 IEEE Symposium on.* IEEE. 2006, 15–pp.

[11] Andrés Blanco and Matias Eissler. *One Firmware to Monitor'em All.* 2012.

[12] Hayden Blauzvern. *Man-in-the-Middle TLS Protocol Downgrade Attack.* URL: https://p16.praetorian.com/blog/man-in-the-middle-tls-ssl-protocol-downgrade-attack (visited on July 19, 2018).

[13] *Bluetooth Explorer: All-in-One Bluetooth Analysis System*. Ellisys. URL: `https://www.ellisys.com/products/bex400/` (visited on July 19, 2018).

[14] *Bluetooth Network Ports*. URL: `https://chromium.googlesource.com/aosp/platform/system/bt/+/master/doc/network_ports.md` (visited on July 19, 2018).

[15] Bluetooth SIG. *Bluetooth Core Specification 5*. Tech. rep. `https://www.bluetooth.com/specifications/bluetooth-core-specification`. Bluetooth SIG, Dec. 2016.

[16] Bluetooth SIG. *Mesh Networking Specification 1.0*. Tech. rep. `https://www.bluetooth.com/specifications/mesh-specifications`. Bluetooth SIG, July 2017.

[17] *Bluetooth in the Android Open Source Project*. URL: `https://source.android.com/devices/bluetooth/` (visited on July 19, 2018).

[18] *CYW4329/CYW4330: Cypress Vendor-Specific Bluetooth Commands*. 002-14847. Rev. *A. Cypress Semiconductor Corporation. Oct. 2016.

[19] *CYW43438 Datasheet*. Cypress. URL: `http://www.cypress.com/file/298076/download` (visited on July 19, 2018).

[20] B. Callaghan and R. Gilligan. *Snoop Version 2 Packet Capture File Format*. RFC 1761. 1995.

[21] Elaina Chai, Ben Deardorff, and Cathy Wu. *Hacking Bluetooth*. 2012. URL: `http://css.csail.mit.edu/6.858/2012/projects/echai-bendorff-cathywu.pdf` (visited on July 19, 2018).

[22] *Cortex-M3 Technical Reference Manual*. ARM. 2006.

[23] *Cypress to Acquire Broadcom's Wireless Internet of Things Business*. 2016. URL: `http://www.cypress.com/news/cypress-acquire-broadcom-s-wireless-internet-things-business-0` (visited on July 19, 2018).

[24] Jake Edge. *Returning BlueZ to Android*. 2014. URL: `https://lwn.net/Articles/597293/` (visited on July 19, 2018).

[25] *Elastic Compute Cloud (EC2)*. Amazon. URL: `https://aws.amazon.com/ec2/` (visited on July 19, 2018).

[26] Scott Fluhrer, Itsik Mantin, and Adi Shamir. "Weaknesses in the key scheduling algorithm of RC4." In: *International Workshop on Selected Areas in Cryptography*. Springer. 2001, pp. 1–24.

[27] *Full OTA Images for Nexus and Pixel Devices: "hammerhead" for Nexus 5*. URL: `https://dl.google.com/dl/android/aosp/hammerhead-ota-mmb29x-13c27a19.zip` (visited on July 19, 2018).

[28] Keijo Haataja and Pekka Toivanen. "Two practical man-in-the-middle attacks on bluetooth secure simple pairing and countermeasures." In: *IEEE Transactions on Wireless Communications* 9.1 (2010).

[29] *How to Build AOSP Marshmallow for Unlocked Xperia Devices*. URL: `https://developer.sonymobile.com/open-devices/aosp-build-instructions/how-to-build-aosp-marshmallow-for-unlocked-xperia-devices/` (visited on July 19, 2018).

[30] *IDA Pro: The Interactive Disassembler*. Hex-Rays. URL: `https://www.hex-rays.com/products/ida/index.shtml` (visited on July 19, 2018).

[31] Wenkwei Lou. *Wireless Remote Firmware Debugging for Embedded Wireless Device*. US Patent 7,318,172. 2008.

[32] Wenkwei Lou. *Wireless Remote Firmware Debugging for Embedded Wireless Device*. US Patent 7,913,121. 2011.

[33] *LuaQEMU: QEMU-based Framework Exposing Several of QEMU-internal APIs to a LuaJIT Core Injected into QEMU Itself*. URL: `https://github.com/Comsecuris/luaqemu` (visited on July 19, 2018).

[34] Ralf-Philipp Weinmann Nico Golde. *Emulation and Exploration of BCM WiFi Frame Parsing using LuaQEMU*. URL: `https://comsecuris.com/blog/posts/luaqemu_bcm_wifi/` (visited on July 19, 2018).

[35] Michael Ossmann. *Discovering the Bluetooth UAP*. 2014. URL: `http://ubertooth.blogspot.com/2014/06/discovering-bluetooth-uap.html` (visited on July 19, 2018).

[36] Michael Ossmann. "Keynote." Troopers Conference. 2018. URL: `https://www.youtube.com/watch?v=9L762WQtWcM` (visited on July 19, 2018).

[37] Michael Ossmann and Dominic Spill. *Project Ubertooth: Open Source Wireless Development Platform Suitable for Bluetooth Experimentation*. 2011. URL: `https://github.com/greatscottgadgets/ubertooth` (visited on July 19, 2018).

[38] Michael Ossmann, Dominic Spill, and Mark Steward. "Bluetooth, Smells Like Chicken." DEFCON 17. 2009. URL: `https://www.youtube.com/watch?v=9WAmMwUyzMc`.

[39] *Pwntools: A CTF Framework and Exploit Development Library*. URL: `https://github.com/Gallopsled/pwntools` (visited on July 19, 2018).

[40] *Python 2.7 Release*. URL: `https://www.python.org/download/releases/2.7/` (visited on July 19, 2018).

[41] *QEMU: The Fast Processor Emulator*. URL: `https://www.qemu.org/` (visited on July 19, 2018).

[42] *Random Bit Generation: NIST Statistical Test Suite*. National Institute of Standards and Technology. 2016. URL: `https://csrc.nist.gov/Projects/Random-Bit-Generation/Documentation-and-Software` (visited on July 19, 2018).

[43] *Reference Manual for FreeRTOS Version 10.0.0 Issue 1*. Amazon.com Inc. 2017. URL: `https://freertos.org/Documentation/FreeRTOS_Reference_Manual_V10.0.0.pdf` (visited on July 19, 2018).

[44] Matthias Schulz. "Teaching Your Wireless Card New Tricks: Smartphone Performance and Security Enhancements Through Wi-Fi Firmware Modifications." PhD thesis. Technische Universität, 2018.

[45] Matthias Schulz, Daniel Wegemer, and Matthias Hollick. "NexMon: A Cookbook for Firmware Modifications on Smartphones to Enable Monitor Mode." In: *arXiv preprint arXiv:1601.07077* (2015).

[46] Matthias Schulz, Daniel Wegemer, and Matthias Hollick. *Nexmon: The C-based Firmware Patching Framework*. 2017. URL: `https://nexmon.org` (visited on July 19, 2018).

[47] *Script Collection for Nintendo Switch Joy-Con*. URL: `https://github.com/shuffle2/nxpad` (visited on July 19, 2018).

[48] Yaniv Shaked and Avishai Wool. "Cracking the bluetooth PIN." In: *Proceedings of the 3rd international conference on Mobile systems, applications, and services*. ACM. 2005, pp. 39–50.

[49] Dominic Spill. *Motivating the Problem: Sniffing Bluetooth is Hard*. 2013. URL: `http://ubertooth.blogspot.com/2013/02/motivating-problem.html` (visited on July 19, 2018).

[50] *The Attack Vector 'BlueBorne' Exposes Almost Every Connected Device*. Armis. 2017. URL: `https://www.armis.com/blueborne/` (visited on July 19, 2018).

[51] *The GNU Readline Library*. URL: `https://tiswww.case.edu/php/chet/readline/rltop.html` (visited on July 19, 2018).

[52] *Ubertooth One - Your BLE Hacking Tool*. Great Scott Gadgets. URL: `https://www.attify-store.com/products/ubertooth-one-your-ble-hacking-tool` (visited on July 19, 2018).

[53] Vitor Ventura and Vladan Nikolic. *Reversing FreeRTOS on Embedded Devices*. 2017. URL: `https://recon.cx/2017/brussels/resources/slides/RECON-BRX-2017-FreeRTOS_Embedded_Reversing.pdf` (visited on July 19, 2018).

[54] *Wireshark*. URL: `https://www.wireshark.org/` (visited on July 19, 2018).

[55] *argparse - Parser for Commandline Options, Arguments and Subcommands*. URL: `https://docs.python.org/3/library/argparse.html` (visited on July 19, 2018).

[56] *dhdutil*. URL: `https://android.googlesource.com/platform/hardware/broadcom/wlan/+/master/bcmdhd/dhdutil/` (visited on July 19, 2018).

[57] evm. *A Code Pirate's Cutlass: Recovering Software Architecture from Embedded Binaries*. 2018. URL: `https://recon.cx/2018/montreal/schedule/system/event_attachments/attachments/000/000/042/original/RECON-MTL-2018-CodePiratesCutlass.pdf` (visited on July 19, 2018).

[58] *python-btsnoop: Parsing Module for BtSnoop Packet Capture Files and Encapsulated Bluetooth Packets*. URL: `https://github.com/joekickass/python-btsnoop` (visited on July 19, 2018).

[59] *radare2*. URL: `https://www.radare.org/r/` (visited on July 19, 2018).

THESIS STATEMENT

pursuant to § 22 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Dennis Mantz, have written the submitted Master Thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once. In the submitted thesis the written copies and the electronic version for archiving are identical in content.

Darmstadt, July 19, 2018

Dennis Mantz

ERKLÄRUNG ZUR ABSCHLUSSARBEIT

gemäß § 22 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Dennis Mantz, die vorliegende Master Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden. Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Darmstadt, 19. Juli 2018

Dennis Mantz