Security and Privacy in Bluetooth Low Energy

Pallavi Sivakumaran

Information Security Group
Royal Holloway University of London

Thesis submitted for the degree of
Doctor of Philosophy

May 2021

Declaration

These doctoral studies were conducted under the supervision of Dr. Jorge Blasco Alis.

I declare that this thesis has been composed by myself and that the work has not been submitted for any other degree or award in any other university or educational establishment. I confirm that the work submitted is my own, except where work which has formed part of jointly-authored publications has been included. My contribution and those of the other authors to this work have been explicitly indicated in the text. I confirm that appropriate credit has been given within this thesis where reference has been made to the work of others.

Pallavi Sivakumaran
May 2021

Acknowledgements

My sincerest thanks to my supervisor, Jorge Blasco, without whom this thesis would not have been possible. Thank you for your guidance, encouragement and enthusiasm (and patience!) throughout these past four years. Thank you also to Juan Tapiador and Daniel O'Keeffe for a positive viva experience and very useful feedback.

I am deeply grateful to the Engineering and Physical Sciences Research Council for funding my PhD and to the Centre for Doctoral Training at Royal Holloway for giving me this opportunity. Special thanks to Carlos for being so supportive of research-related travel, and to Claire for taking care of so many organisational aspects. Thanks also to CIM and IT, particularly Francesco, for your quick responses to any tech issues. On a similar note, thanks to Jason/Jordy/Feargus for handling the lab infrastructure.

A thank you to the staff and students of the Information Security Group at Royal Holloway, for the research-friendly environment they have created. The PhD seminars, in particular, gave me insights into so many different aspects of security. Thanks also to Mark, James(x2) and the team at F-Secure for my placement. I learned so much in those six months.

Thanks to the folks of the S3Lab for creating such a supportive environment to work in... even when we aren't actually *in* the lab anymore. Also thanks to my fellow PhD students, especially my cohort, for the hikes, chats and fun times. Ashley, our walk-and-talks were some of the highlights of the past year.

Respecting that time-honoured tradition of listing the most important people in a person's life last, I finally thank my family and friends. My family, for being so very loving and supportive and for inspiring in me a love of science, a strong work ethic, and respect for academic integrity. My friends (although, this includes family as well), for the crazy times, the crosswords, the group calls, the laughter. Most of all, for putting up with long and detailed descriptions of my code, results and papers, most of which you probably didn't understand or care about. You can heave a sigh of relief now. Peace and quiet at last. For a little while, anyway.

Abstract

Bluetooth Low Energy (BLE) is a popular wireless technology deployed in billions of devices within the Internet of Things (IoT). Applications for BLE increasingly handle user health and personal data, and perform safety-critical functions. Ensuring the security and privacy of BLE deployments is therefore becoming more and more important.

In this thesis, we seek to advance the existing body of knowledge regarding BLE security and privacy, with the overarching goal of improving the security of real-world BLE systems. We first present a comprehensive overview of existing BLE attacks and vulnerabilities. We define a taxonomy for attacks according to their impact and attack mechanisms, and categorise vulnerabilities to reflect their source in terms of architectural layer and stakeholder(s). This immediately allows for the identification of root causes, mitigation strategies and responsible entities.

Second, we identify and demonstrate an application-level unauthorised data access vulnerability on BLE-enabled multi-application platforms, and develop a custom tool to estimate the number of BLE devices that may be vulnerable to such unauthorised access. Through a pragmatic stakeholder analysis, we propose a backward-compatible specification-level solution to this vulnerability to ensure the security of data on the greatest proportion of BLE devices.

Third, we determine the extent of various BLE vulnerabilities and their associated impacts through a series of measurement studies with purpose-built tools, utilising novel information extraction techniques against multiple information sources. We implement an incremental access algorithm in `ATT-Profiler` to determine minimum access requirements for BLE characteristic data, and utilise this to analyse access restrictions applied to real-world devices. We identify and prioritise vulnerabilities in the absence of physical devices via `BLE-GUUIDE`, a framework for performing security analysis and functionality mapping using BLE Universally Unique Identifiers (UUIDs). We also design and implement `argXtract`, a framework for analysing stripped ARM Cortex-M binaries, to extract multiple security-relevant configurations from BLE firmware files, and identify numerous vulnerabilities therein.

Through these contributions, we look towards a greater understanding of the security considerations surrounding BLE and an increase in security within the BLE ecosystem.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AES	Advanced Encryption Standard
AFH	Adaptive Frequency Hopping
AOSP	Android Open Source Project
API	Application Programming Interface
APK	Android (application) Package
ATT	Attribute Protocol
BLE	Bluetooth Low Energy
BR/EDR	Basic Rate/Enhanced Data Rate
CCCD	Client Characteristic Configuration Descriptor
CFG	Control Flow Graph
CGM	Continuous Glucose Monitoring
CRC	Cyclic Redundancy Check
CSRK	Connection Signature Resolving Key
CVE	Common Vulnerabilities and Exposures
DFU	Device Firmware Upgrade
DH	Diffie-Hellman
DTM	Direct Test Mode
ECDH	Elliptic-Curve Diffie-Hellman
ETSI	European Telecommunications Standards Institute
FPR	False Positive Rate
GAP	Generic Access Profile
GATT	Generic Attribute Profile
HCI	Host Controller Interface
HR	Heart Rate
HID	Human Interface Device
IO	Input-Output
IoT	Internet of Things
IP	Internet Protocol
IRK	Identity Resolving Key
ISM	Industrial Scientific Medical
L2CAP	Logical Link Control and Adaptation Protocol
LE	Low Energy
LESC	Low Energy Secure Connections
LL	Link Layer
LSB	Least Significant Bit
LTK	Long Term Key

MAC	Message Authentication Code
MAC [address]	Media Access Control [address]
MIC	Message Integrity Check
MitM	Man-in-the-Middle
MSB	Most Significant Bit
NLP	Natural Language Processing
NVRAM	Non-Volatile Random Access Memory
OS	Operating System
OTA	Over The Air
PDU	Protocol Data Unit
PHY	Physical Layer
PII	Personally Identifiable Information
RAM	Random Access Memory
RSSI	Received Signal Strength Indicator
SIG	Special Interest Group
SM	Security Manager
SMP	Security Manager Protocol
SoC	System on a Chip
STK	Short Term Key
TCP	Transmission Control Protocol
TK	Temporary Key
TPR	True Positive Rate
UUID	Universally Unique Identifier
VT	Vector Table

Part I
Preliminaries

1 Introduction

In this chapter, we discuss the motivation for our work, our research questions, the contributions we have made, and the structure of the remainder of the thesis. We also provide a list of publications and manuscripts that form the basis of this thesis.

1.1 Motivation

Bluetooth Low Energy (BLE) is a well-known and widely-used wireless communications technology, with billions of BLE-enabled devices extant in the world today [1]. Introduced in 2010 within version 4.0 of the Bluetooth specification,[1] BLE shares many traits with its predecessor, *Bluetooth Classic*,[2] but is intended specifically for resource-constrained devices. The Bluetooth Special Interest Group (SIG) develops and maintains the Bluetooth Classic and BLE standards (both currently in a single specification document, which is at version 5.2).[3]

The focus on low energy usage has made BLE particularly suitable for the Internet of Things (IoT), where many sensor-based devices operate for months, and even years, on coin-cell batteries. Some well-known examples of BLE devices are proximity sensors, fitness trackers and smart door locks. The potential of BLE for vehicular applications [3], home energy monitoring [4] and home automation systems [5] has been analysed in recent years. The technology is also increasingly being used or proposed for use in medical devices, such as heart rate and blood glucose monitoring systems, asthma inhalers, and insulin pumps [6].

As the sensitivity and criticality of BLE applications increase, it is clear that the impact of vulnerabilities will also be greater. In fact, vulnerabilities have been identified in many modern BLE devices that have serious implications for user safety, security and privacy. For example, a number of "smart" locks were found to implement weak authentication schemes or use unprotected passwords [7], which means that an attacker could unlock the device and gain access to the resource that the lock was protecting (which, in the case of smart door locks, could be a house or office building). Research into BLE-enabled hover-boards/eScooters has identified that several devices can be controlled remotely by unauthorised entities due to insufficient protection for control commands sent over the BLE interface. The devices could be forced to accelerate or be brought to a sudden halt by an attacker, both of which could cause injury to the user [8].

[1]We use the terms "Bluetooth specification" and "BLE specification" interchangeably, to refer to the BLE-specific parts of the specification document.

[2]Bluetooth Classic refers to Bluetooth Basic Rate/Enhanced Data Rate (BR/EDR), which was the "original" Bluetooth. Bluetooth Classic and BLE are to be considered distinct and incompatible technologies [2], despite much overlap in their design.

[3]There have been five versions of the standard since the introduction of BLE within v4.0: v4.1, released Dec 2013; v4.2, released Dec 2014; v5.0, released Dec 2016; v5.1, released Jan 2019; v5.2, released Dec 2019.

We observe that the use of BLE with medical intervention devices such as insulin pumps brings forth the possibility of a particularly dangerous attack, where an attacker could control the insulin administered to a user, with potentially life-threatening consequences. On the privacy side, many BLE devices have been found to be vulnerable to tracking [9, 10]. This means that, for devices such as wearables, the user can also be tracked.

The examples above demonstrate that security and privacy aspects of BLE warrant careful consideration and analysis. Examining the volume of research into BLE security and privacy over the years, it becomes apparent that initially such studies were few and far between, and that the number of research papers focusing on the technology has increased significantly in the last five years. Specifically, from its introduction in 2010 until 2015, we were only able to identify seven publications that discussed the security of Bluetooth Low Energy [11–17]. The studies on BLE security and privacy that were published in the year 2016 alone [9, 10, 18–23] exceeded the total number of publications from the preceding five years. Since then, BLE has been the subject of numerous, varied security and privacy analyses, and in 2020 the number of papers published on the subject was twice the number of BLE security publications from the 5-year period 2011-2015. We observe from BLE device shipment data that the technology only started gaining popularity from the year 2015 onward [24], which could account for the corresponding increase in security and privacy studies.

Despite the increase in research into BLE security and privacy over the last few years, there were still several shortcomings at the outset of our research. We discuss these below.

Misconceptions regarding the technology There are several misconceptions regarding BLE security, primarily due to two reasons: (i) misunderstanding the design of BLE and incorrectly assuming that certain operational features are security mechanisms, and (ii) conflating BLE with Bluetooth Classic and assuming that Bluetooth Classic vulnerabilities are applicable to the case of BLE as well. We have observed such erroneous assumptions in earlier works but also in papers published as recently as 2020 [11, 25–27].

Insufficient focus on BLE data security The primary asset on a BLE device is its data, which is tightly-coupled with the BLE stack. Several earlier BLE security and privacy studies focus on the security of the BLE pairing mechanism, which implicitly relates to the security of BLE data. A useful extension to this is an analysis of the protection applied to individual data values on real-world BLE systems, together with an understanding of the *type* of data, which then provides insights into the actual impact of unprotected data. This type of study was notably absent at the outset of our research.

Insufficient focus on the application layer BLE is a full-stack protocol and includes an application layer [28]. However, initial studies on BLE security and privacy did not focus on this layer, except in one case to propose an application-layer solution to link-layer issues [17]. That is, vulnerabilities *within* the application layer were not scrutinised.

Lack of techniques for vulnerability extent analysis In the year prior to our research, a number of BLE security analysis tools were released for interacting with and identifying vulnerabilities on a physical BLE device. However, to estimate the *extent* of a vulnerability within the BLE ecosystem, bulk analysis techniques that do not require physical device interaction are necessary. These again were lacking at the outset of our work.

This thesis seeks to overcome the above shortcomings and further the existing body of knowledge regarding BLE security and privacy by performing critical surveys, clarification of misconceptions, and vulnerability analyses at the application layer, as well as developing tools and techniques for performing bulk measurement studies of the BLE ecosystem.

1.2 Research Objective and Questions

In this section, we define our overarching research objective and describe research questions that stem from it. For each research question, we provide a brief overview in *italics* regarding how this thesis approaches the problem.

The objective of this thesis is to advance the understanding of the existing state of security and privacy within real-world BLE systems, with the overarching goal of improving the security of the BLE ecosystem. As we mentioned in §1.1, the primary asset on a BLE device is its data. Despite this, BLE data security had not undergone sufficient scrutiny at the time that we embarked on our research. In our work, we address this gap by placing particular focus on protections applied to and impacts arising from BLE data.

When considering BLE security and privacy, the BLE specification contains a number of protocols and procedures intended to protect the confidentiality and/or the integrity of BLE data, as well as to preserve device privacy. However, previous studies have shown that real-world devices may not implement such mechanisms. Or they may implement the weakest possible security that is provided by the specification. In addition, protocols defined within the specification may themselves contain weaknesses. This gives rise to the question:

> RQ01: Are existing BLE deployments secure?

This thesis comprehensively surveys existing studies to identify vulnerabilities that affect real-world systems. It also presents purpose-built tools for identifying insufficient security or privacy in real-world BLE deployments via device, firmware and application analyses.

Even though the BLE specification does provide protocols and procedures to protect security and privacy, *application*-layer protection is not fully defined within the specification, despite BLE being a full-stack protocol. This prompts the following research question:

> RQ02: Does the lack of clearly-defined application-layer security mechanisms result in a lack of protection for BLE data?

This thesis examines application layer data security in the context of current BLE usage scenarios, which feature interactions between a BLE device and an app on a mobile phone or computer.

It demonstrates an unauthorised data access vulnerability for multi-application platforms. It also estimates the proportion of real-world devices that do not implement custom protection at the application layer, thereby leaving BLE data vulnerable to unauthorised access.

When a vulnerability has been identified for BLE, the next natural step is to identify its *extent*, i.e., the proportion of real-world systems that may be affected, and its *impact*, i.e., either at a high level in terms of the types of attack that can be conducted, or at a more detailed level in terms of the specific impact (taking into consideration knowledge regarding the functionality of the system). This links to the question:

RQ03: If vulnerabilities exist, what is their extent and impact?

Several measurement studies presented in this thesis automatically enable an estimation of the extent of a vulnerability via bulk analysis. The thesis also demonstrates impact analysis by obtaining markers regarding device functionality from various sources of information.

Vulnerabilities in real-world BLE systems are not always related to weaknesses within the specification. Some are due to coding bugs or end products not incorporating security features. Therefore, when discussing mitigation strategies, the following question is pertinent:

RQ04: How should vulnerabilities be mitigated and who is responsible
for mitigation?

This thesis, within the vulnerability survey, examines the root cause for existing vulnerabilities and identifies possible mitigation options as well as the relevant, responsible stakeholders. For the application-layer vulnerability mentioned previously, it describes a pragmatic stakeholder analysis, which leads to a suitable solution strategy.

1.3 Contributions

We can see that answering the research questions in §1.2 will involve identifying vulnerabilities that are applicable to BLE, measuring their extent and impact, and determining mitigation strategies. Concretely, the contributions within this thesis can be described under those three categories: Identification, Measurement, and Mitigation.

- **Identification:** We present a comprehensive survey of attacks on BLE security and privacy, and identify the underlying vulnerabilities that give rise to the attacks. Within this category, we also include an application-level unauthorised data access vulnerability that we have discovered for BLE-enabled multi-application platforms.
- **Measurement:** We measure the BLE security and privacy landscape by estimating the extent and potential impact of vulnerabilities that are present in current BLE systems. We utilise several sources of information for this purpose: BLE devices, firmware binaries and mobile applications. For each analysis, we present purpose-built tools and frameworks, all of which are available as open-source repositories for the benefit of the research community. These measurement studies not only provide insights as to the current state of BLE security

and privacy, but can also aid in focusing future research by identifying the most widespread or impactful vulnerabilities.

- **Mitigation:** We propose a practical specification-level solution, based on a pragmatic stakeholder analysis, for the application-level unauthorised data access vulnerability that we have identified. Our proposed solution enables security by default, while maintaining backward compatibility with existing BLE systems.

We describe these contributions in greater detail in §1.4.

1.4 Thesis Structure

This thesis is organised into four parts, with each part grouping together related chapters. We describe the structure and content of each part and its integrant chapters below. Table 1.1, at the end of this section, maps the thesis chapters to the research questions presented in §1.2.

Part I: Preliminaries, of which this chapter is a constituent, describes the motivation for our work and presents details regarding BLE that are referenced in the remainder of the thesis. It also presents a comprehensive survey of existing BLE attacks and vulnerabilities.

Chapter 2 provides an overview of BLE, its architectural layers and their operations. It also sets out security and privacy requirements for BLE based on the adversarial model we consider, and describes the features that are available in the BLE specification to fulfil these requirements.

Chapters 3 and 4 collectively present our comprehensive survey on BLE security and privacy, sourcing information from academic studies, industry whitepapers and CVE reports. Chapter 3 presents our taxonomy for classifying BLE attacks, focusing on the impact or outcome of the attacks. It provides details about each attack and the mechanisms for effecting it. In Chapter 4, we present the vulnerabilities that give rise to the attacks. For each vulnerability, we identify applicability and source, and outline proposed solutions and related tools and frameworks. We also briefly survey proposals for increasing the security or privacy within BLE deployments in the presence of one or more vulnerabilities.

Part II: BLE Application Layer Security pertains to an application-level unauthorised data access vulnerability that we have identified for multi-application platforms.

Chapter 5 details our findings regarding unauthorised data access at the application layer. We describe a vulnerability by which a malicious application on a multi-application platform (such as Android or iOS) can read and write data on BLE devices, often without the user's knowledge. We analyse possibilities for mitigating the vulnerability and arrive at the conclusion that, at present, the only available mechanism for preventing application-layer attacks is the implementation of end-to-end security by developers.

Given that developer-defined application-layer security is the only means (for the time being) by which the unauthorised data access vulnerability can be mitigated, a pertinent question is "What proportion of BLE devices actually *do* implement such application-layer security?" Chapter 6 presents a purpose-built open-source taint analysis tool, `BLECryptracer` [29], for analysing BLE-enabled Android applications to identify the presence of cryptographically-protected BLE data at the application layer. It also describes the results from applying this tool to a large dataset of several thousand BLE-enabled Android APKs, and demonstrates that a significant proportion of real-world BLE systems do *not* protect their data at the application layer.

In Chapter 7, we describe a specification-level modification as our proposed solution to the unauthorised data access vulnerability. We arrive at our solution based on defined security and system requirements, and following a pragmatic, multi-faceted stakeholder analysis. We show that our solution ensures protection by default while maintaining backward compatibility with existing BLE devices. We implement a proof-of-concept [30] for the Android-x86 platform and demonstrate the viability and efficacy of our solution via tests against real-world devices and their associated mobile applications.

Part III: Measurement of BLE Security and Privacy describes a set of tools and techniques that we have developed to perform security- and privacy-related measurements of BLE devices and the BLE ecosystem. We perform analyses and measurement studies against physical devices, firmware binaries, and mobile applications that interface with BLE devices.

The measurement study conducted in Chapter 6 revealed that a significant proportion of BLE-enabled APKs do not implement application-layer security, which implies that data on the associated BLE devices will be vulnerable to unauthorised access. However, the presence of a vulnerability in a BLE device does not immediately reveal its impact. For example, unauthorised access to a thermostat's readings cannot be considered to be of the same level of importance as unauthorised access to a glucometer's readings. The impact of a BLE vulnerability therefore depends in large part on the type of device, i.e., the device's functionality. Chapter 8 describes a framework, `BLE-GUUIDE` [31], for BLE vulnerability and impact measurement using a novel source of information: Universally Unique Identifiers (UUIDs), which are used in the structuring of BLE data.

Chapter 9 describes our technique for testing physical BLE devices, to gauge the lowest level of security at which BLE data can be accessed. We accomplish this via a Node.js analysis tool, `ATT-Profiler` [32]. The results from executing this tool against real-world devices show that many devices apply little to no protection for most of their BLE data. We analyse possible reasons for this finding and discuss our results in the light of subsequent findings as well as later research on key downgrade attacks.

In Chapter 10, we discuss measuring BLE security and privacy via the device firmware, which has the potential to divulge rich information regarding a BLE device's security configuration. We detail the challenges inherent to the analysis of stripped binaries (which is the format in which most BLE binaries are available in the wild), and the mechanisms by which we have overcome them in our binary analysis framework, `argXtract` [33]. In fact, `argXtract` is not confined to the analysis of only BLE binaries, but can generally be applied to IoT peripheral binaries that target ARM Cortex-M processors, which is the processor family of choice for many resource-

Table 1.1: Mapping of thesis chapters to research questions.

	Chapter										
Research Question	1*	2*	3	4†	5	6	7	9	10	8	11*
RQ01				🔍		📊		📊	📊	📊	
RQ02					🔍						
RQ03			🔍	🔍		📊		📊	📊	📊	
RQ04				🔍			🔧				

Contribution category: 🔍- Identification, 📊- Measurement, 🔧- Mitigation.

**Chapters 1, 2, 11 are introductory, background and concluding chapters, and do not map to the research questions. †Chapter 4 does describe the applicability (which in some cases denotes extent) and mitigation options for several of the vulnerabilities. However, we obtain this information via a survey, rather than through measurement studies or design. Therefore, the contributions of this chapter are classified only as Identification.*

constrained devices. We describe and analyse the results from the extraction of security-relevant configuration information from stripped real-world Cortex-M BLE binaries.

Part IV Conclusion summarises our contributions with respect to our research questions, and presents recommendations for various BLE stakeholders based on observations we have made throughout this work.

1.5 Publications and Manuscripts

This thesis is based on the following papers. All papers were the result of discussion between my supervisor and myself (also in one instance with other collaborators), and my supervisor provided significant guidance and feedback for each manuscript. The iOS attack validation discussed in PM2 (§5.3.5), the UUID extraction, NLP processing and overall result tabulation described in PM4 (§8.3.1.1, §8.3.2.2-§8.3.2.3, §8.4.2-§8.4.4), as well as the device emulation at the end of §8.5.2, were conducted by co-authors. All other development, testing, analysis and writing tasks were undertaken by myself.

PM1 Pallavi Sivakumaran and Jorge Blasco. “An Attack Taxonomy and Vulnerability Analysis for Bluetooth Low Energy.” ***This work corresponds to Chapters 2 through 4 of this thesis.***

PM2 Pallavi Sivakumaran and Jorge Blasco. “A Study of the Feasibility of Co-located App Attacks against BLE and a Large-Scale Analysis of the Current Application-Layer Security Landscape.” In *Proceedings of the 28th USENIX Security Symposium (USENIX Security ’19)*, pp. 1-18. 2019 [34]. ***This paper is discussed in Chapters 5, 6.***

PM3 Pallavi Sivakumaran and Jorge Blasco. “Who’s Accessing My Data? Application-Level Access Control for Bluetooth Low Energy.” To appear in *Proceedings of the 17th EAI International Conference on Security and Privacy in Communication Networks (SecureComm ’21)*. 2021. ***This work is discussed in Chapter 7.***

PM4 Pallavi Sivakumaran, Jorge Blasco, Chaoshun Zuo, Zhiqiang Lin. "A Large-Scale Analysis of Bluetooth Low Energy Enabled IoT: Measurement, Assessment, and Implications." ***This work is discussed in Chapter 8.***

PM5 Pallavi Sivakumaran and Jorge Blasco. "A Low Energy Profile: Analysing Characteristic Security on BLE Peripherals." In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy (CODASPY '18)*, pp. 152-154. 2018 [35]. ***A modified version of this paper is presented in Chapter 9, with some code fixes and new test results.***

PM6 Pallavi Sivakumaran and Jorge Blasco. "argXtract: Deriving IoT Security Configurations via Automated Static Analysis of Stripped ARM Binaries." To appear in *Proceedings of the 37th Annual Computer Security Applications Conference (ACSAC '21)*. 2021. ***This paper is discussed in Chapter 10.***

2 Background

In this chapter, we present details regarding Bluetooth Low Energy that will be referenced in the remainder of the thesis. We describe the architecture and operation of BLE in §2.1. We describe our adversarial model, set out security and privacy requirements for a BLE system, and describe the available security and privacy features within the BLE specification in §2.2. We present some observations and discuss next steps in §2.3.

2.1 Bluetooth Low Energy: Architecture and Operations

This section provides an overview of the Bluetooth Low Energy stack and of the functions of the various layers and protocols within the stack. It primarily focuses on aspects that will be referenced in subsequent chapters when discussing BLE security and privacy concerns.

Figure 2.1 depicts a typical usage scenario for BLE in the consumer context. A user may have multiple BLE-enabled devices, each with its own companion mobile application (or perhaps a single application communicating with multiple devices). The user may also utilise mobile applications that provide proximity-based services via BLE beacons.[1]

In this section, we will use a well-known application of BLE to illustrate the basic operations of the technology: that of a wearable fitness tracker. The tracker uses multiple sensors to measure parameters such as the user's heart rate, steps taken, quality of sleep, etc., and communicates the readings to a companion application on a mobile phone. The tracker is resource-constrained and is ideally expected to operate for several days on a single charge, whereas the mobile phone will typically have greater memory, power and computing capabilities. For this reason, BLE assigns asymmetric workloads to the two devices, and it is the mobile phone that typically initiates connections, issues commands to the fitness tracker, and performs the bulk of the processing.

The Bluetooth specification uses the roles "Master" and "Slave" for the initiating and responding devices, respectively. In our example, the mobile phone will be the Master device and the fitness tracker will be the Slave. These roles are introduced formally in §2.1.1.3. The Bluetooth specification also defines several other roles, depending on the architectural layer and functionality.

The Bluetooth Low Energy stack is made up of three main subsystems: the Controller, the Host and the Applications. An interfacing layer known as the Host Controller Interface (HCI) also

[1]Beacons are special types of BLE devices that broadcast an identifier over the radio interface. They enable location-based services, such as push notifications for proximity-based advertising or content delivery, and social media "check-in" functionality.

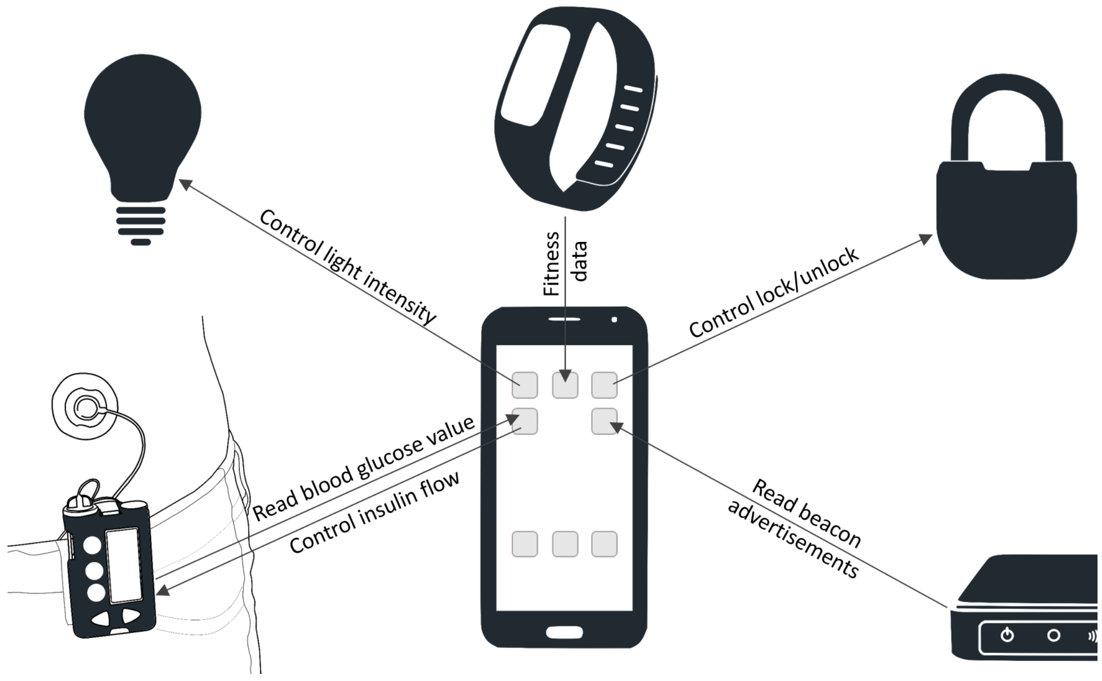


Figure 2.1: Example consumer BLE applications.

exists. Figure 2.2 depicts the BLE stack in terms of these subsystems and the layers within them. The remainder of this section describes the functionality of each component within the stack.

2.1.1 Controller

The Controller contains the radio frequency components to enable a BLE device to transmit and receive data over the 2.4 GHz frequency band. In addition, it may also be capable of performing cryptographic functions. If this is the case, it will have access to physical sources of randomness, and contain an encryption engine to enable the transmission of encrypted or authenticated data [2]. Architecture-wise, the Controller comprises the Physical Layer (sometimes denoted as PHY) and the Link Layer (LL), as well as a test layer known as Direct Test Mode (DTM).

2.1.1.1 Physical Layer

The Physical Layer contains the components that perform signal modulation, frequency hopping, and other functions, all of which collectively are responsible for the actual transmission and reception of data over radio frequencies.

Frequency hopping is often mentioned when discussing Bluetooth security. Bluetooth devices (both Classic and Low Energy) transmit data over the 2.4 GHz Industrial Scientific Medical (ISM) frequency band. This is a “crowded” part of the frequency spectrum, because the same unlicensed band is used by other wireless technologies such as Wi-Fi and Zigbee. In addition, devices such as microwave ovens also operate within this range. For this reason, interference is an issue that must be actively dealt with. Bluetooth has adopted Adaptive Frequency Hopping (AFH) techniques to effectively combat wireless interference. With AFH, the 2.4 GHz band

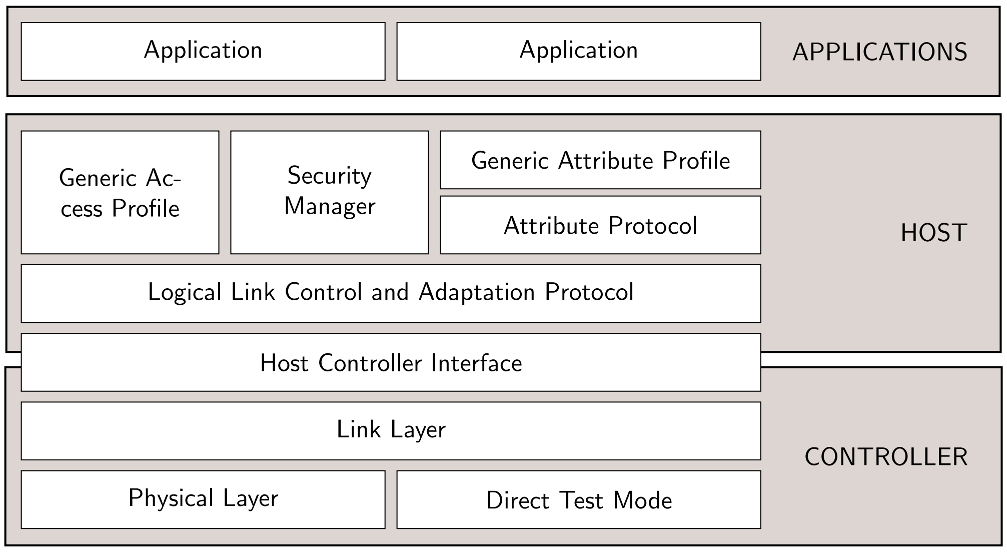


Figure 2.2: The BLE stack.

is subdivided into *channels*. BLE uses 40 channels, each 2 MHz wide. Within a connection between two BLE peers, communications "hop" over channels, following what is known as a hopping pattern. If interference is observed in a particular region within the band, then the hopping pattern will exclude the channels within that region.

2.1.1.2 Direct Test Mode

Direct Test Mode offers a standard method by which to test a BLE device's Physical Layer. It is not an essential component as far as functionality is concerned, but aims to make testing easier even after a device has been packaged into another product [2].

2.1.1.3 Link Layer

The Link Layer performs a number of essential tasks to enable Bluetooth Low Energy functionality. Some of the responsibilities of the Link Layer include defining packet structure, advertising, device discovery, connection initiation and management, data transmission and reception to and from a connected device, and data encryption [2]. The most important functions of the Link Layer, as required for this thesis, have been explained below.

Advertising and scanning Taking the example of the fitness tracker, we have already mentioned that it is a resource-constrained device. It is therefore not desirable for the tracker to remain connected to the phone for extended periods of time, as this can result in quicker draining of the battery. For this reason, the tracker and mobile phone will remain disconnected most of the time, and will connect and exchange data as and when needed.

Such being the case, when they are disconnected and wish to connect, the devices need to know that they are in the vicinity of each other. In the case of Bluetooth Low Energy, this is achieved by *advertising* and *scanning*. That is, the fitness tracker sends out advertisements to let other BLE devices in the area know of its presence. These advertisements are sent on fixed *advertising*

PDU Type (4 bits)	RFU (2 bits)	TxAdd (1 bit)	RxAdd (1 bit)	Length (6 bits)	RFU (2 bits)

(a) Advertising PDU header.

LLID (2 bits)	NESN (1 bit)	SN (1 bit)	MD (1 bit)	CP (1 bit)	RFU (2 bits)	Length (8 bits)

(b) Data PDU header.

Figure 2.3: Breakdown of BLE packet header fields.

channels,[2] and contain various pieces of information, most notably the device's address. The mobile phone scans the advertising channels. The first time the fitness tracker is used, the user will be presented with a list of devices that are advertising in the immediate vicinity, and will be able to choose the tracker from the list. The mobile phone will save the tracker's details and thereafter, when the mobile application wants to connect to the tracker, it will scan the advertising channels and look out for an advertisement matching the tracker's details.

Initiating and connecting When the mobile phone/application has identified the tracker that it wants to connect to, it initiates a connection by sending a `Connection Request` over the link layer. The mobile phone in this case is the *Initiator* and the tracker is the *Advertiser*. The initiating device will take on the role of *Master*, and the advertising device will take on the role of *Slave* within the connection.

Once the tracker and phone have connected, the tracker will stop advertising until they disconnect. Upon connection, data transfer can occur. The protocol for requesting data and for responding to such requests is the Attribute Protocol, which is covered in §2.1.3.3.

Defining packet structure A BLE packet[3] has four main fields:

1. Preamble: A sequence of alternating 1s and 0s, which allows the receiver to identify the frequencies that are used for the two bit values [2].
2. Access Address: For most advertising packets, this is a fixed value of `0x8e89bed6`. With data packets, the value differs and is used for identifying a connection. Note that the access address is *not* the same as the device's address.
3. Protocol Data Unit (PDU): Consists of a Header and Payload. Discussed in detail below.
4. Cyclic Redundancy Check (CRC): Provides a mechanism by which transmission bit errors can be identified.

This generic packet format is used for both advertising and data packets. However, the sub-fields *within* the Header and Payload fields will differ based on whether a packet is an advertising packet or a data packet. The structure of advertising and data PDU Headers is depicted in Figures 2.3a and 2.3b, respectively. A detailed description of PDU fields is provided in Table 2.1

[2]Prior to v5.0 of the BLE specification, advertisements were sent out on channels 37, 38, 39 (and data packets were sent on the remaining channels). In v5.0, Extended Advertising was introduced, in which these three channels are referred to as *primary advertising channels* and the data channels are also able to be used as *secondary advertising channels*.

[3]From v5.0, the specification defines *two* packet formats. We consider the one that is most commonly used in this thesis.

Table 2.1: Format of BLE protocol data units, for advertising and data PDUs.

	Advertising PDU	Data PDU
Header	PDU Type: Specifies the type of packet.	LLID: Logical Link Identifier.
	RFU: Reserved for Future Use in specification versions <5.0 and one bit used as ChSel in v5.0+.	NESN: Next Expected Sequence Number, used to either acknowledge or to request resending of the last PDU sent by peer.
	ChSel: Specifies the Channel Selection Algorithm to use [NDF*].	SN: Sequence Number, used for packet identification.
	TxAdd: Dependant on PDU Type. Normally, 0 indicates public address and 1 denotes random address (see §2.2.2.1).	MD: Indicates that More Data is available.
	RxAdd: As for TxAdd.	CP: CTEInfo Present [NDF].
	Length: Number of octets in payload.	RFU: Reserved for Future Use.
	RFU: Used as part of Length field (to form an 8-bit Length field) in v5.0+.	Length: Number of octets in payload.
Payload	Depends on PDU Type. Prior to specification v5.0, payload could be upto 37 bytes. Version 5.0 introduced Extended Advertising Payloads, where advertising packets were offloaded to traditionally data-only channels.	Can contain control information or data. Payload could be upto 27 bytes in v4.0; increased to 251 bytes in v4.2.
Other		MIC: (Optional) Message Integrity Check.

**NDF = Not discussed further*

for advertising and data PDUs separately. The table also shows that the format for a specific type of packet (i.e., advertising or data) *between different versions of the specification* can also differ slightly.

Encryption If the data that is being sent between the tracker and phone is sensitive, it may be encrypted before being sent over the radio interface. Devices in a connection can transmit packets with encrypted payloads and Message Integrity Checks (MIC). For this, keys must be exchanged, and possibly stored, by the communicating devices. The Security Manager is responsible for key exchange and derivation. However, it is the Link Layer that performs encryption/decryption and MIC computation. The Security Manager is discussed in §2.1.3.2, and further details about authentication and encryption options in BLE are given in §2.2.

It is important to note here that it is not a *requirement* in BLE for the tracker and phone to exchange keys in order to transfer data. It may be that simply forming a connection is sufficient. Key exchange is only required if the data or the device has a security requirement [36].

2.1.2 Host Controller Interface

Not, strictly speaking, a complete subsystem in itself, the Host Controller Interface (HCI) is nevertheless an important part of the BLE stack. It sits between the Host and Controller, and enables standardised communication between the two subsystems.

2.1.3 Host

The Host is responsible for multiplexing, security, and exposing a device's state data. It consists of three protocols: the Logical Link Control and Adaptation Protocol (L2CAP), Attribute Protocol (ATT), and Security Manager Protocol (SMP). It also contains two profiles: the Generic Attribute Profile (GATT) and the Generic Access Profile (GAP).

2.1.3.1 Logical Link Control and Adaptation Protocol

The Logical Link Control and Adaptation Protocol (L2CAP) sits on top of the Host Controller Interface and performs channel multiplexing. An L2CAP channel is a logical link between two endpoints on BLE peer devices. More specifically, it is a sequence of packets between a service on one BLE device and a service on its BLE peer [2], and is represented using a Channel Identifier. Certain channels, referred to as *fixed channels*, are reserved for specific functions. In BLE, there are 3 such channels: one for signalling, one for the Security Manager Protocol (§2.1.3.2), and one for the Attribute Protocol (§2.1.3.3).

2.1.3.2 Security Manager

As mentioned before, BLE devices that wish to access protected data must first exchange keys. To do this, they must perform a process known as *pairing*. The Security Manager describes the processes and algorithms used for pairing, and defines the cryptographic toolbox that is utilised. The pairing process is described in detail in §2.2.5.

2.1.3.3 Attribute Protocol and Generic Attribute Profile

In Bluetooth Low Energy, all data are stored as discrete values called *attributes*, and accessed via the Attribute Protocol (ATT). For example, a fitness tracker measures and stores certain parameters, such as the user's heart rate or step count. A glucose monitor may obtain and store periodic measurements of the user's blood glucose levels. A smart lock may contain a value that controls the lock state (i.e., open or locked). All these values (heart rate, step count, glucose levels, lock control) are stored as attributes on the respective devices. A mobile phone can then use the Attribute Protocol to read or write this information.

The Attribute Protocol operates on a client-server model, where the client accesses attributes from the server. Both the Master and the Slave can act as both client and server. That is, in our example it is not only the phone that is able to read from or write to the tracker. The tracker may also access some information from the phone.

An attribute has four main components: (i) a *handle*, which serves as an address and is unique per device, (ii) a *type*, which is defined by a Universally Unique Identifier (UUID) and which should be unique across devices for a certain type of attribute, (iii) a *value*, which is the actual value held by the attribute and which differs according to the attribute type, and (iv) a set of *permissions*, which control how the attribute can be accessed. Attribute permissions are discussed in detail in §2.2.4.

Table 2.2: Sample attribute database.

Handle	**Type**[*]	**Value**	**Permissions**
0x0001	Primary Service	GAP Service	Read Only, No Authentication, No Authorisation
0x0002	Characteristic	Device Name	Read Only, No Authentication, No Authorisation
0x0003	Device Name	"FitBand101"	Read/Write, No Authentication, No Authorisation
0x0004	Characteristic	Appearance	Read Only, No Authentication, No Authorisation
0x0005	Appearance	"Generic Watch"	Read Only, No Authentication, No Authorisation
0x0006	Primary Service	GATT Service	Read Only, No Authentication, No Authorisation
0x0007	Characteristic	Service Changed	Read Only, No Authentication, No Authorisation
0x0008	Service Changed	<Handles>	None
0x0009	CCCD[†]	0x0000	Read/Write, No Authentication, No Authorisation
0x000A	Primary Service	HR[‡] Service	Read Only, No Authentication, No Authorisation
0x000B	Characteristic	HR Measurement	Read Only, No Authentication, No Authorisation
0x000C	HR Measurement	80bpm	None
0x000D	CCCD	0x0000	Read/Write, Authenticated encryption required, No Authorisation
0x000E	Characteristic	HR Control Point	Read Only, No Authentication, No Authorisation
0x000F	HR Control Point		Write Only, Authenticated encryption required, No Authorisation

[*]*Each entry under the* ***Type*** *column will actually be denoted using a UUID. Textual descriptions have been provided here for clarity.* [†]*CCCD = Client Characteristic Configuration Descriptor.* [‡]*HR = Heart Rate.*

The Generic Attribute Profile builds on top of ATT and defines different types of attributes, as well as procedures for accessing them. The three main types of attributes are services, characteristics and descriptors. Characteristics hold the actual data values of interest (e.g., heart rate or glucose measurements). A characteristic can have zero or more descriptors, which define characteristic properties or format. One or more (usually related) characteristics are grouped into a service. A service may also contain other services.

A device stores its attributes in an *attribute database*. As an example, a fitness tracker's (partial) attribute database may look something similar to that in Table 2.2. In the table, the attribute with the type *Primary Service* is a *service declaration*. All attributes following this and ending at the next service declaration belong to this service [2]. For example, in Table 2.2, the Heart Rate Service declaration is at handle 0x000A, and all attributes up to and including handle 0x000F belong to the Heart Rate Service. Similarly, the attribute with the type *Characteristic* is called a *characteristic declaration*. A characteristic declaration is followed immediately by the characteristic value attribute, which contains the actual value of interest (e.g., the Heart Rate Measurement value at handle 0x000C). If a characteristic has associated descriptors, they will be included in the database after the value attribute. There are different types of descriptors, but we only discuss the Client Characteristic Configuration Descriptor (CCCD) in this thesis.

The CCCD enables data to be obtained from a BLE device in a manner different to the traditional request-response. That is, normally when a client wants to obtain a value from a server, it issues a read request and obtains the value as the response. If we consider a scenario where a mobile phone wants to obtain a heart rate measurement from a fitness tracker, the mobile phone (as the client) will issue a read request to the server on the fitness tracker. However, for frequently changing values such as heart rates, the mobile phone would have to continually poll the tracker to obtain the latest value. BLE instead allows a client to enable *notifications* or *indications* on the server. Notifications and indications allow the server to send the client updated values as and when the value changes, thereby eliminating the need for constant polling. Notifications expect no response, but an indication expects an acknowledgement that the message was received. Both notifications and indications are enabled by writing to the characteristic's CCCD. Note that if a characteristic supports notifications or indications, it will have a *property* of `notify` or `indicate`, respectively. If a characteristic supports traditional reads and writes, it will have `read` and `write` properties. There are additional characteristic properties that are outside the scope of this thesis.

SIG and vendor-defined UUIDs As mentioned previously, an attribute is identified using a UUID. There are some attributes (services, characteristics and descriptors) defined by the Bluetooth SIG, which have specific meanings and functionality. Examples include the Heart Rate Service, Continuous Glucose Monitoring Service, Insulin Delivery Service, and their associated characteristics. Each of these services and characteristics has been assigned a specific UUID, taken from a reserved range of UUIDs. When a BLE device contains one of these SIG-defined UUIDs, the associated data type and behaviour will be known. However, developers are not confined to using only SIG-defined services. They can define custom services and characteristics as well, which will require UUIDs outside the SIG-reserved range.

2.1.3.4 Generic Access Profile

In §2.1.1.3, we saw that the Link Layer performs advertising, device discovery, etc. It is the Generic Access Profile (GAP) that defines *how* devices discover other devices, find out the services that are offered, and reconnect to each other without user intervention after the initial connection [2]. GAP also provides options for security and privacy.

GAP roles GAP defines four roles: *Broadcaster*, which is a device that transmits advertisements; *Observer*, a device that listens for advertisements; *Peripheral*, a device that advertises and becomes a Slave once connected to a BLE peer; and *Central*, a device that scans for and initiates connections to Peripherals, and which is a Master upon connection. In our example, the fitness tracker would be a Peripheral and the mobile phone would be a Central. An example for a Broadcaster would be a BLE beacon, while a device that listens for beacon advertisements would be an Observer.

Discovery and connection GAP defines different modes for device discoverability, and sets out various policies and procedures for advertising and performing device discovery. It also defines different modes and procedures for establishing and terminating connections.

Bonding Bonding is the storage of keys generated or exchanged during pairing, such that they can be used when devices reconnect without the need for repeating the pairing process. In essence, bonding allows for the creation of a trust relationship between two BLE peers.

Security and privacy GAP defines various security *modes* and *levels*, each of which offer different levels of security. A detailed description of the possible security options has been provided in §2.2.3. GAP also provides a mechanism for device privacy, which is discussed further in §2.2.2.1.

2.1.4 Applications

The Application Layer defines characteristic, service and profile specifications, built on top of GATT. Profiles describe possible interactions between two communicating BLE devices, where the two devices implement specific services, which in turn contain specific characteristics. An example of such a profile could be the Continuous Glucose Monitoring (CGM) Profile. This profile specifies two roles: Sensor and Collector (where the Sensor is the device that takes glucose measurements from the user and the Collector is the device that reads data from the Sensor and issues commands to it). The CGM Profile describes the services and characteristics that the two roles must implement or support, and how they interact with each other. For example, the profile specifies that the CGM Sensor must implement the CGM Service, which contains the CGM Measurement Characteristic, CGM Feature Characteristic and many others. The CGM Profile also specifies security requirements for the various characteristics.

2.2 Security and Privacy Requirements and Features in BLE

We describe our adversarial model and the security and privacy requirements we have identified for BLE in §2.2.1. The security and privacy features defined within the BLE specification are described in §2.2.2. Security- and privacy-relevant protocols and configurations that are employed to realise the security features are discussed in §2.2.3 through §2.2.5.

2.2.1 Security and Privacy Requirements for BLE

We outline here the requirements with regard to security and privacy for Bluetooth Low Energy devices. The adversarial model we consider is as follows. A legitimate user owns a BLE-enabled device operating in one of three configurations: (i) disconnected, (ii) involved in BLE communications with a peer device, or (iii) about to be connected or paired to a peer BLE device (by the user). The attacker is in one of two possible configurations: (i) in proximity to the user (within BLE operating range, with or without range extension), or (ii) conducting attacks via an application on the user's device. In the case of the former, the attacker is able to scan BLE channels, perform signal jamming attacks, connect to and interact with BLE devices, perform cryptographic functions (including reasonable brute-forcing), and manipulate data over the wireless interface. In the latter case, the attacker is able to install a malicious application (with suitable permissions) onto the user's mobile phone or computer. The attacker does not have physical access to the user's BLE devices or mobile phone/computer. The attacker also does not have infinite computing capabilities, i.e., computationally hard problems (e.g., in cryptography) cannot be solved by the attacker.

With this adversarial model, the security triad of confidentiality, integrity and availability are applicable to BLE as with other communication technologies. Additionally, taking into consideration the wireless nature of communications and the advertising behaviour of BLE Peripherals and Broadcasters, we include two other essential requirements: device authentication and privacy. We describe these requirements in §2.2.1.1 through §2.2.1.5.

2.2.1.1 Confidentiality

BLE devices exchange attribute data with each other. This data should not be accessible by unauthorised entities, particularly if the data is sensitive (e.g., pertaining to a user's health or lifestyle, or relating to safety or security).

2.2.1.2 Integrity

It should not be possible for BLE data to be tampered with, either in transit or storage, without some evidence being generated regarding the tampering. Note that we do not consider random bit flipping or the effects of a noisy channel here, since those will be detected by the CRC in every packet (see §2.1.1.3). It should also not be possible for BLE vulnerabilities to be exploited in order to attack the integrity of the *device*.

2.2.1.3 Availability

Many BLE devices have health- or security-critical functionality, such as in the case of continuous glucose monitoring devices or smart door locks. Such devices must be available to the user and be able to communicate with peer devices at any time, subject to battery and normal operating considerations.

2.2.1.4 Device Authentication

If an attack device was able to masquerade as a legitimate device, it may be possible for the attacker to read sensitive information from BLE devices or modify the functionality of a BLE device by writing data to it. To prevent this, it should be possible for BLE devices or applications to ascertain that the peer devices that they communicate with are the intended devices, i.e., as expected by the user.

2.2.1.5 Privacy

Many BLE-enabled devices are designed to always be in proximity to the user, e.g., fitness trackers, continuous glucose monitors or contact tracing apps [37]. A BLE device should not leak information that could make the device or its user vulnerable to tracking. In addition, data regarding the type of BLE device (which can reveal information about the user, such as in the case of medical devices) or the user's activities should also not be leaked.

2.2.2 Security and Privacy Features

We describe the security and privacy features that are defined within the BLE specification in §2.2.2.1 through §2.2.2.4.

2.2.2.1 Device Privacy

Device privacy is a feature that was included in the very first version of BLE, i.e., in v4.0 of the Bluetooth specification. As mentioned in §2.1.1.3, BLE Peripherals and Broadcasters periodically transmit advertising messages containing the device address. If a fixed advertising address is used, then the device (and by extension, possibly also the user) will be vulnerable to tracking. For this reason, BLE introduced the concept of *private addresses*, whereby the BLE device has a fixed *identity address* that is only revealed to bonded peers, but advertises using private addresses that should change periodically.

There are two different types of private addresses: resolvable and non-resolvable. Resolvable private addresses should be used when reconnection without re-pairing is required. In this scenario, the device that employs private addresses transmits a 128-bit Identity Resolving Key (IRK) and its identity address to its peer during the bonding process. The device periodically changes the (private) address within its advertisements. However, a bonded peer in possession of the IRK will be able to resolve the private address to obtain the identity address. With non-resolvable addresses, the private addresses cannot be resolved to an identity address.

2.2.2.2 Data Confidentiality

BLE allows for protecting data on the link layer via encryption. It uses AES-CCM cryptography with a key of *up to* 128 bits for this purpose, and has done so since the first version of the technology. Encryption keys are derived from keys generated/exchanged during the pairing process, which is described in §2.2.5. Note that it is not a *requirement* for data to be encrypted on BLE devices.

2.2.2.3 Data Signing

It is possible to optionally sign data on the Link Layer for integrity verification *without* encryption. The signature consists of a Message Authentication Code (MAC) and a replay-protection counter. A 128-bit Connection Signature Resolving Key (CSRK) is used for generating the MAC. The CSRK is transmitted to a peer during the bonding process.

2.2.2.4 Link Layer Filtering

Link Layer filtering is not actually mentioned as a *security* feature within the Bluetooth specification. It is considered a mechanism for reducing the number of devices that the Link Layer needs to respond to, and consists of device whitelisting in combination with various filter policies. However, even if it is not explicitly a security mechanism, Link Layer filtering could be used to reduce the impact of Denial of Service (DoS) attacks. In particular, whitelisting in combination with resolvable private addresses could limit the connections processed by the Link Layer to only those devices that have previously undergone pairing and bonding.

Table 2.3: BLE security modes and levels.

Mode	Level	Description
1	1	No security (no authentication and no encryption)
	2	Unauthenticated pairing with encryption
	3	Authenticated pairing with encryption
	4	Authenticated LE Secure Connections† pairing with encryption using a 128-bit strength encryption key
2	1	Unauthenticated pairing with data signing
	2	Authenticated pairing with data signing
3	1	No security (no authentication and no encryption)
	2	Use of unauthenticated Broadcast_Code
	3	Use of authenticated Broadcast_Code

† *This is the new "generation" of pairing in BLE. See §2.2.5.*

2.2.3 Security Modes and Levels

When two BLE devices connect, one or both devices or some of the data on them may have security requirements. These are specified in terms of a Security *Mode* and a Security *Level*. Table 2.3 presents the Security Modes and Levels available within the BLE specification. The terms "authenticated pairing" and "unauthenticated pairing" in the table refer to the presence or absence of Man-in-the-Middle (MitM) protection. Unauthenticated pairing refers to the pairing process without MitM protection, while authenticated pairing requires MitM protection.

LE Security Mode 3, as described in the table, is a mode introduced in v5.2 of the BLE specification. It is concerned with Isochronous channels used for LE Audio (which is a new feature that is still partially in development) and is out of scope for this thesis.

A "Secure Connections Only" mode has also been defined, and a device in this mode will only use Mode 1, Level 4, unless a service specifies Mode 1, Level 1.

2.2.4 (G)ATT Security

As described in §2.1.3.3, every value used or served by a BLE device is stored as an attribute. This could be personal or protected information, such as the user's heart rate, blood glucose levels, or a value controlling the 'locked' status of a smart lock. It could also be something less sensitive like the device's name or type. For this reason, each attribute may need different levels of security. Restricting access to attributes is facilitated via the attribute permissions mentioned in §2.1.3.3, which are a combination of the following:

- **Access permissions** specify whether an attribute is Readable, Writeable, Readable *and* Writeable, or None (neither Readable nor Writeable).
- **Authentication/Encryption permissions** indicate whether the link between the two devices must be authenticated/encrypted before the attribute can be accessed.
- **Authorisation permissions** specify whether client authorisation is required before the attribute can be accessed.

The permissions are stored in a security database, and a client cannot directly learn of the permissions applied to an attribute. Instead, the client has to first attempt to *access* (i.e., read or write) the attribute. If the access is not allowed or requires more security than is currently available, then the server will typically respond with an error. If the action type (read or write) requested by the client is not permitted, then a `Read/Write Not Permitted` will be returned. If the client is not authorised to access the attribute, an `Insufficient Authorisation` error will be returned. If the server requires a secure connection before it will allow an action to be performed, and the two devices already share keys but the link is unencrypted, then an `Insufficient Encryption` error will be returned; if the devices don't already share keys or if the strength of the pairing process used to generate the keys was insufficient, an `Insufficient Authentication` error will be returned. However, the required *level* of encryption or authentication will not be indicated. In the case of `Insufficient Authentication` errors, the client device will need to pair (or re-pair) with the server device before re-attempting access.

Attribute permissions are most relevant in the case of characteristics (or, more specifically, characteristic *value* attributes) and descriptors. Characteristic *declarations*, on the other hand, are expected to always be freely readable, but not writable. The same applies to service declarations. This is reflected in the attribute database in Table 2.2, where all service and characteristic declarations have the permissions "Read Only, No Authentication, No Authorisation".

2.2.5 Pairing

Pairing is the process by which two devices generate one or more shared secrets. It is handled by the Security Manager and, in BLE, it is always initiated by the Master device. During the pairing process, both devices exchange capabilities and security requirements, and ultimately derive keys that are used to encrypt the connection between them (at the Link Layer).

There are two "generations" of pairing: *LE Legacy* was introduced in the first version of BLE, i.e., v4.0 of the Bluetooth specification. It uses a proprietary key exchange protocol, which was acknowledged within the specification to be vulnerable to passive eavesdropping attacks. *LE Secure Connections* (LESC) was introduced in v4.2 of the specification, and uses Elliptic-Curve Diffie-Hellman (ECDH) for key exchange.

Regardless of the type of pairing (LE Legacy or LESC), the pairing process consists of three phases. These are outlined below and discussed in greater detail in §2.2.5.1 through §2.2.5.3.

1. **Feature Exchange** enables the exchange of supported features and security requirements, and determines which of the two pairing methods (i.e., LE Legacy or LESC) and four association models (discussed in §2.2.5.2) will be used.
2. The **Key Generation** stage differs based on whether LE Legacy or LESC is used, but always results in the generation of a key that is then used to encrypt the transport.
3. **Transport Specific Key Distribution** is an optional phase common to both LE Legacy and LESC, and is when additional keys may be exchanged if required.

Table 2.4: Key features exchanged during Phase 1 of BLE pairing.

Feature	**Description**
Bonding Flag	While pairing results in key generation, it is only if the devices *bond* that keys will be stored for future use, thereby eliminating the need to go through pairing on each connection. The bonding flag is a 2-bit value indicating whether bonding is required.
OOB Data	This is an 8-bit value indicating whether Out Of Band (OOB) authentication data is available. OOB simply refers to the fact that the authentication material is shared between the devices using some method other than Bluetooth, e.g., NFC.
IO Capabilities	This is an indication of the device's input/output capabilities (such as the presence of a keyboard and/or display).
Encryption Key Size	A BLE device may support only up to certain key sizes for encryption, which it will indicate to the pairing device via the "maximum encryption key size" field. The key is actually always 128 bits long. This field really refers to the *entropy*, which must be a minimum of 56 bits and can be up to 128 bits.
Key Distribution	Both devices inform each other of the keys they wish to exchange. The keys can be Long Term Keys (LTK), Identity Resolving Keys (IRK), and Connection Signature Resolving Keys (CSRK).
MitM	Single bit indicating whether MitM protection is required.
LESC	Single bit indicating whether LESC is supported.

2.2.5.1 Phase 1: Feature Exchange

The Feature Exchange stage is initiated when the Master device sends a `Pairing Request` to the Slave. Within this request the Master device sends its input/output capabilities, Out-Of-Band authentication information availability, the requirements it has for authentication and for encryption key size, as well as the keys it would like exchanged. If the features are satisfactory and the Slave wishes to proceed with pairing, it will send a `Pairing Response` back to the Master, with details of its own capabilities and security requirements. Some of the most important features exchanged during the first stage of pairing have been summarised in Table 2.4.

2.2.5.2 Phase 2: Key Generation

Once the two devices have exchanged features, they generate keys to encrypt the link. This is done via one of four possible *association models*, selected based on the features (particularly the IO capabilities) exchanged in Phase 1. The association model determines how the key (that is used to encrypt the transport during pairing) will be generated. There are four possible association models:

1. **Passkey Entry** - This method is probably familiar to most people who have used Bluetooth. During the pairing process, one device displays a passkey (a decimal number which is normally six digits long), and the user keys the passkey into the other device. This process results in an authenticated key. It requires at least one of the two devices to have a display and the other to have a keypad.

2. **Just Works** - *Just Works* is an association model that is used when at least one of the communicating devices does not have suitable input-output capabilities. This method

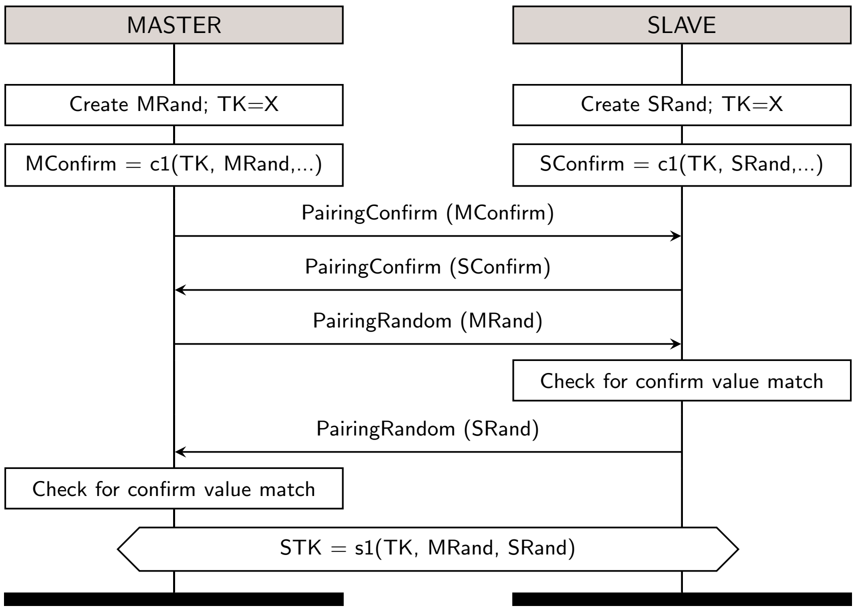

MASTER
SLAVE
Create MRand; TK=X
Create SRand; TK=X
MConfirm = c1(TK, MRand,...)
SConfirm = c1(TK, SRand,...)
PairingConfirm (MConfirm)
PairingConfirm (SConfirm)
PairingRandom (MRand)
Check for confirm value match
PairingRandom (SRand)
Check for confirm value match
STK = s1(TK, MRand, SRand)

Figure 2.4: Legacy BLE pairing: key generation.

results in an unauthenticated key. It affords the least amount of security but is widely used, since many BLE Peripheral devices have neither a display nor a keypad.

3. **Numeric Comparison** - With *Numeric Comparison*, both devices need to only have a display and the ability to indicate Yes/No. They don't need a full keyboard/keypad. The pairing process generates keys and then displays an artefact of the key generation process on both devices' screens. If the displayed values match, then the user can choose 'Yes' to proceed with the remainder of the pairing process. The resultant key is considered to be an authenticated key. *Numeric Comparison* is viewed as being more secure than *Passkey Entry* and *Just Works*, but is only available for LE Secure Connections, i.e., it is not available for LE Legacy pairing.

4. **OOB** - The Out Of Band association model is used when authentication material is shared between the two devices using some transport other than Bluetooth (e.g., NFC). This can be a very secure method provided the OOB mechanism is secure. If the mechanism for transmitting the key is secure enough, then the key is considered to be authenticated.

In the case of LE Legacy pairing, the key that results from the Key Generation phase is a Short Term Key (STK). With LESC, the key is the Long Term Key (LTK).

Figure 2.4 depicts the LE Legacy pairing process. Both devices start off with a Temporary Key (TK). This Temporary Key is derived based on the association model that is used. If *Just Works* is used, then the key is simply 000000. In the case of *Passkey Entry*, the TK will be the 6-digit passkey that is displayed on one device's screen and input by the user on the other device. With OOB, the TK will be the key that is distributed out-of-band. Both devices individually generate a random number. These are referred to as MRand (the random number generated by the Master device) and SRand (generated by the Slave). The devices do not share the random numbers with each other at this point. Instead, they each generate Confirmation Values using a function `c1` with the TK, their own random number, and messages exchanged during Phase 1 as inputs. They then exchange these confirmation values, followed by their random numbers. Each device computes its peer device's confirmation value using the peer's random number, and checks this against the received confirmation value. If the values match, then a Short Term Key is generated using a key generation function `s1`, with TK, MRand and SRand as arguments.

LE Legacy was acknowledged within v4.0 of the Bluetooth specification, i.e., the version of the specification in which BLE was first introduced, as being vulnerable to passive eavesdropping attacks. Version 4.2 of the specification saw the introduction of LESC. With LESC, Elliptic-Curve Diffie-Hellman is used as the key exchange protocol. Each device generates a public-private key pair, and then transmits its public key (which is a point in the format (x, y) on the elliptic curve) to the connected BLE device. Information from the public key, as well as independently generated random numbers are used by the Slave device to compute a confirmation value. The Slave transmits this to the Master device, which verifies the value. Once this has been completed, if *Just Works* or *Numeric Comparison* association models are used, then a User Confirmation value is computed and (in the case of *Numeric Comparison*) displayed to the user for manual confirmation. In the case of *Passkey Entry*, the passkey is converted to a 20-bit value and 20 rounds (one for each bit) of confirmation value generation and validation occurs between the communicating devices.[4] Once this stage is complete, each device computes a MacKey and an LTK using previously exchanged values and the newly derived shared Diffie-Hellman key. Additional confirmation values are exchanged and checked prior to moving on to Phase 3.

2.2.5.3 Phase 3: Transport Specific Key Distribution

This phase is used to transmit any keys that the devices indicated as required during the Phase 1 Feature Exchange. In the case of LESC, this could be a CSRK, IRK, as well as the device identity (i.e., real address) of the communicating devices. In the case of LE Legacy, an LTK, as well as two values EDIV and RAND, which identify the LTK, may also be exchanged.

2.3 Chapter Summary, Observations and Next Steps

In this chapter, we have described the BLE architecture and the operation of each individual architectural layer. We have set out security and privacy requirements for BLE systems and described the security and privacy features that are available within the BLE specification.

At a high level, the security and privacy features within the BLE specification appear to fulfil most of the requirements presented in §2.2.1. That is, the requirements for confidentiality, integrity and privacy are directly addressed, and device availability can be improved via LL filtering, by reducing the number of connections that a device has to process. The device authentication requirement is fulfilled via pairing. However, from the time BLE was introduced, many vulnerabilities affecting these requirements have been identified, both with the specifica-

[4] This reveals one bit of the passkey at a time and necessitates random passkeys.

tion and with implementations. These vulnerabilities can be exploited to perform different types of attacks.

We next present a comprehensive look at BLE attacks and vulnerabilities. Due to the difference in their nature, we consider attacks and vulnerabilities separately. With BLE attacks, we focus on the *outcome* of the attacks and the mechanisms for achieving them. We define a purpose-defined taxonomy (Chapter 3) for analysing such attacks. With vulnerabilities, the goal is to understand the *root cause* such that similar vulnerabilities may be prevented in future. We therefore analyse vulnerabilities differently, taking into consideration the architectural layer within which a vulnerability occurs, as well as the stakeholders who are responsible for it (Chapter 4).

3 BLE Attack Taxonomy

In this chapter, we present our taxonomy for classifying current BLE attacks, and describe in detail each class of attack and the mechanisms for effecting it.

3.1 Introduction

Bluetooth Low Energy has been incorporated into billions of devices, with around 3 billion BLE-capable devices being shipped in 2019 alone [1]. As we have already observed, an increasing number of these BLE devices are handling sensitive user data (such as health measurements or Personally Identifiable Information (PII)) or performing critical functions (such as controlling insulin pumps, eScooters or door locks). As the sensitivity or criticality of BLE applications increases, so too does the scrutiny placed on its security. Over the past few years, a number of security and privacy attacks have been identified for Bluetooth Low Energy. In this chapter, we present a comprehensive survey of such attacks. We describe the methodology employed to survey previous work in the area of attacks on BLE security and privacy (§3.2). We develop a taxonomy for existing BLE attacks, focusing on the impact or outcome of the attack (§3.3). We describe each attack in detail, explaining the mechanisms by which it can be effected (§3.4-§3.8). For completeness, we also provide a brief overview of some well-known Bluetooth Classic attacks and analyse their applicability to BLE (§3.9).

Related work We mention here related work that is applicable to both Chapter 3 and Chapter 4. There have been several attempts over the years to survey BLE attacks and vulnerabilities. For example, Padgette et al. [38] present a succinct account of Bluetooth vulnerabilities and threats (for both Classic and BLE) in a NIST publication, covering every version of the Bluetooth specification up to and including v4.2. However, they do not include a number of attacks on privacy and availability. An overview of key- and cipher-related vulnerabilities in different versions of Bluetooth (mainly applicable to Bluetooth Classic) is presented in work by Cope et al. [39]. Other types of vulnerabilities are not addressed in this work. Lonzetta et al. [40] make references to these vulnerabilities and present a taxonomy for Bluetooth threats (again, predominantly applicable to Bluetooth Classic). Ghori et al. [27] describe BLE attacks in the context of BLE mesh networks, but mistakenly include attacks that are applicable to only Bluetooth Classic. A number of possible attacks against BLE are described in work by Jasek [19], but without addressing all possible vulnerabilities. Attacks specifically targeting BLE *beacons* are explored by Tay et al. [20] and Kolias et al. [41].

As can be seen from the sample above, most Bluetooth security and privacy surveys tend to focus more on Bluetooth Classic. In addition, none have presented a complete overview of *all* security and privacy issues identified in BLE thus far. Further, some studies conflate Bluetooth Classic attacks, such as Bluejacking and Bluesnarfing, with BLE. This chapter and the next together present a comprehensive look at the various security and privacy attacks and vulnerabilities that are specifically applicable to BLE. We mention Bluetooth Classic attacks separately, solely to discuss their potential applicability for the BLE case.

3.2 Methodology

In order to obtain a list of all publications related to BLE security/privacy, we use a keyword-based search against the Digital Bibliographic Library Browser (DBLP) database [42], by generating a combination of each search term within the set {*bluetooth,*[1] *ble, btle*} with each term within the set {*security, privacy, attack, vulnerability, vulnerabilities, track, spoof*}. We cross-check by using each search term within the first set with the names of highly-ranked conferences (i.e., those within the top two tiers in [43]),[2] to ensure that papers presented at such venues are included.[3] Further, we perform a search with each of the terms within the set {*bluetooth low energy, ble, btle*} (without additional terms and manually examining titles for indications of security and privacy relevance) to further build our dataset of BLE security- and privacy-related academic publications.

However, the nature of BLE research is such that oftentimes studies are published as whitepapers or technical papers, rather than as purely academic publications. For this reason, we include such papers when they have made significant contributions to the field. CVEs are another source of information regarding vulnerabilities. We therefore search the MITRE database [44] for BLE-related vulnerabilities. However, we only include them if they apply to a large number of devices, i.e., vulnerabilities in chipsets or platforms rather than small-scale end products.

Since our focus is BLE attacks and vulnerabilities, we filter out works that utilise BLE to build custom security or privacy solutions, and analyses of custom protocols built on top of BLE (e.g., [45]). We also generally do not include studies that merely repeat previously demonstrated attacks or test for known vulnerabilities. Further, we exclude studies pertaining to Bluetooth Mesh, as that is a standard built on top of BLE (using BLE as the transport protocol), containing its own stack, and deserving separate analysis.

[1]We filter out results specific to Bluetooth Classic based on the year of publication (BLE was introduced in 2010. Therefore, any publication prior to that year will not relate to BLE). We also filter out works by examining the title and abstract for terms specific to Bluetooth Classic (e.g., “BR/EDR”, “Classic”, “SDP”, “LMP”).

[2]At the time of writing, this list included the following conferences: IEEE Symposium on Security and Privacy (S&P); ACM Conference on Computer and Communications Security (CCS); USENIX Security Symposium (Security); ISOC Network and Distributed System Security Symposium (NDSS); International Cryptology Conference (Crypto); European Cryptology Conference (Eurocrypt); European Symposium on Research in Computer Security (ESORICS); International Symposium on Recent Advances in Intrusion Detection (RAID); Annual Computer Security Applications Conference (ACSAC); The International Conference on Dependable Systems and Networks (DSN); Internet Measurement Conference (IMC); ACM Symposium on Information, Computer and Communications Security (ASIACCS); Privacy Enhancing Technologies Symposium (PETS); IEEE European Symposium on Security and Privacy (EuroS&P); IEEE Computer Security Foundations Symposium (CSF); International Conference on the Theory and Application of Cryptology and Information Security (Asiacrypt); Theory of Cryptography Conference (TCC); Conference on Cryptographic Hardware and Embedded Systems (CHES).

[3]Where a study has been previously published in preprint form and later at a conference, we use the conference version.

We distil the information from each source within the resultant list and separate out the attack(s) and the underlying vulnerability or vulnerabilities. We taxonomise and discuss the attacks in this chapter, and analyse the vulnerabilities in Chapter §4.

3.3 A Taxonomy for Attacks against BLE Security and Privacy

We construct the first layer of our attack taxonomy hierarchy by mapping the security requirements outlined in §2.2.1 to the following corresponding attacks:

1. *Unauthorised Acquisition of Data*: This is an attack on confidentiality. We consider the unauthorised acquisition of BLE data (i.e., characteristic values) under this category.

2. *Tampering*: This is an attack on integrity. We consider attacks that tamper with data integrity or the device's integrity in this category. However, attacks on a device which render it temporarily or permanently unusable are not included in this category. They are instead included within *Denial of Service*.

3. *Denial of Service*: A detriment to availability, any attack that temporarily or permanently renders the device unusable (such that it cannot be used as expected by a legitimate user) is considered here.

4. *Profiling & Tracking*: This is a privacy issue. Any means by which a device (and possibly the user) can be tracked over time or across locations, or any leakage of information about the user, is considered within this category. We do not include leakage of BLE data values within this category, as that is covered by *Unauthorised Acquisition of Data*.

5. *Spoofing*: This occurs due to a failure of device authentication. Any attack that enables a legitimate device to be spoofed in such a way as to cause a peer device to accept the spoofed device as legitimate is included here.

The second-level categories within the taxonomy expand upon the first-level attacks, denoting the impact or outcome of the attack in greater detail. At the third and fourth levels are the mechanisms by which the attacks are effected, in progressively more detail.

Figure 3.1 presents the resultant taxonomy for classifying BLE-related attacks. The labelling of the nodes corresponds to the associated security requirement. For example, nodes related to unauthorised data access are an attack on **C**onfidentiality and are labelled with "C". Note that the first-level node *(D) Spoofing* and the second-level node *(I.1) Tampering→Tamper with BLE data* do not have subtrees. This is because the subtrees for these nodes are derived from the subtrees of other nodes. For example, tampering with BLE data would require the same conditions as to read the data (except that the data must also be writable for it to be tampered with). Therefore, the node *(I.1)* will have the same subtree as node *(C.1)*. Similarly, to fully spoof a device, its data and address (including the IRK, if relevant) must be known. Therefore, the subtree for *(D)* will be a combination of the subtrees for nodes *(C.1)* and *(P.1)*.

§3.4 through §3.8 describe attack mechanisms within each of these categories in further detail.

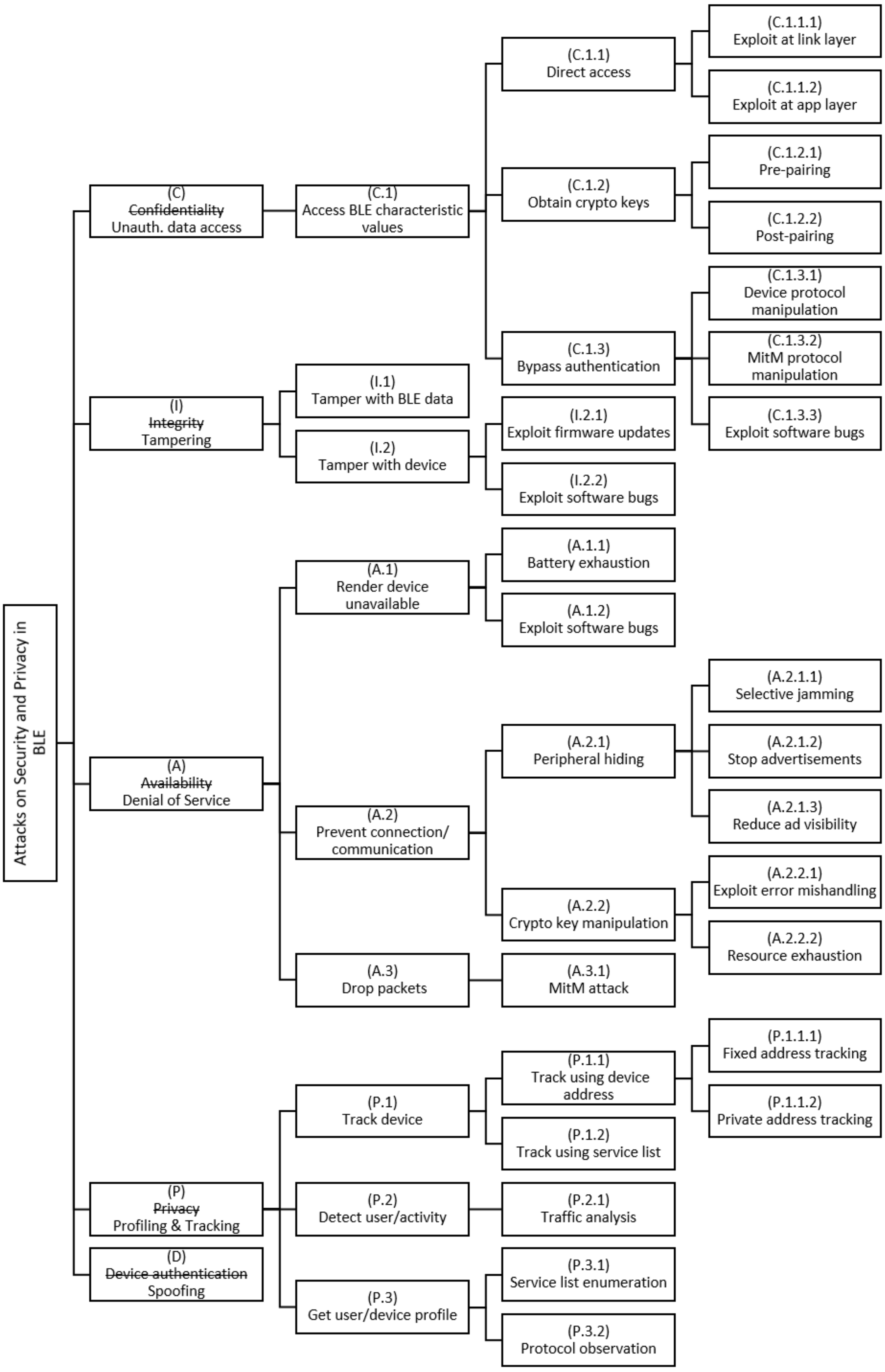

Attacks on Security and Privacy in BLE
(C) ~~Confidentiality~~ Unauth. data access
(C.1) Access BLE characteristic values
(C.1.1) Direct access
(C.1.1.1) Exploit at link layer
(C.1.1.2) Exploit at app layer
(C.1.2) Obtain crypto keys
(C.1.2.1) Pre-pairing
(C.1.2.2) Post-pairing
(C.1.3) Bypass authentication
(C.1.3.1) Device protocol manipulation
(C.1.3.2) MitM protocol manipulation
(C.1.3.3) Exploit software bugs
(I) ~~Integrity~~ Tampering
(I.1) Tamper with BLE data
(I.2) Tamper with device
(I.2.1) Exploit firmware updates
(I.2.2) Exploit software bugs
(A) ~~Availability~~ Denial of Service
(A.1) Render device unavailable
(A.1.1) Battery exhaustion
(A.1.2) Exploit software bugs
(A.2) Prevent connection/ communication
(A.2.1) Peripheral hiding
(A.2.1.1) Selective jamming
(A.2.1.2) Stop advertisements
(A.2.1.3) Reduce ad visibility
(A.2.2) Crypto key manipulation
(A.2.2.1) Exploit error mishandling
(A.2.2.2) Resource exhaustion
(A.3) Drop packets
(A.3.1) MitM attack
(P) ~~Privacy~~ Profiling & Tracking
(P.1) Track device
(P.1.1) Track using device address
(P.1.1.1) Fixed address tracking
(P.1.1.2) Private address tracking
(P.1.2) Track using service list
(P.2) Detect user/activity
(P.2.1) Traffic analysis
(P.3) Get user/device profile
(P.3.1) Service list enumeration
(P.3.2) Protocol observation
(D) ~~Device authentication~~ Spoofing

Figure 3.1: Taxonomy for BLE attacks.

The vulnerabilities corresponding to the attacks described in this chapter are specified in Chapter 4. When an attack is described as exploiting one or more vulnerabilities, it is assumed that the devices do actually contain the vulnerabilities. The true applicability for the vulnerabilities is also provided in Chapter 4.

3.4 Unauthorised Acquisition of Data

This section deals with attacks that result in unauthorised access to BLE characteristic data. Note that the adaptive frequency hopping employed by BLE is incorrectly referred to as a security mechanism and the ability to eavesdrop on (unencrypted) BLE traffic by circumventing AFH is stated as an attack on confidentiality in several studies [11, 25, 26, 39]. However, the Bluetooth specification itself does not refer to AFH as a security mechanism. It states that the "*LE system employs a frequency hopping transceiver to combat interference and fading*" [36]. Therefore, we consider the ability to eavesdrop on raw BLE communications to be akin to eavesdropping on any other wireless connection, i.e., not in itself an attack.

C.1 Access BLE characteristic data

As described in §2.1.3.3, BLE data is stored and accessed as discrete values known as attributes, which can have permissions (§2.2.4) applied to them in order to restrict access. Recall also that applying such protection is not mandatory.

There are three main ways in which an attacker might read data from a victim BLE device:

1. Direct access, if the data has insufficient protection applied to it.
2. If the data is protected but the attacker is able to obtain cryptographic keys that the victim device shares with a legitimate peer.
3. If the data is protected but the attacker is able to bypass authentication.

These mechanisms are discussed in C.1.1 through C.1.3. In all cases, the characteristic data must have the `read`, `indicate` or `notify` properties (see §2.1.3.3) for the attacks to work.

C.1.1 Direct access

This section deals with the acquisition of BLE characteristic values when either no protection is applied or the protection applied to the characteristics is not sufficiently strong.

C.1.1.1 Exploit at link layer If a characteristic does not specify authentication or encryption requirements, i.e., does not require pairing before it can be accessed, then its values can be read by any connected device, including attacker devices.[4] The values can also be obtained via MitM attacks or monitoring of the wireless interface when two legitimate devices are exchanging data.

[4] Note that an attacker will be able to eavesdrop on values in any case, but with pairing, the data will be encrypted and therefore not intelligible to the attacker. Strong cryptographic protections applied at a higher layer can also prevent unauthorised access, and are discussed in C.1.1.2.

Even if a characteristic specifies Mode 1 Level 2 security requirements, this can be achieved via *Just Works* pairing. The nature of this particular pairing model is such that an attacker may be able to pair with the device covertly, i.e., without the user's knowledge. The attacker can also compute the relevant STK as part of a MitM or eavesdropping attack (as described in C.1.2.1, using TK="000000") and thereafter decrypt the traffic between two legitimate devices.

Further, even if two legitimate devices' IO capabilities enable them to conduct authenticated pairing (e.g., *Passkey Entry* or *Numeric Comparison*), an attacker may be able to MitM into the pairing process, downgrade it to *Just Works* by manipulating the feature exchange, and thereafter access poorly protected data. Note that the ability to downgrade pairing is independent of the protection applied to BLE data, and does not automatically mean that BLE data will be accessible to the attacker; the data will only be accessible if little or no protection has been applied to it.

We demonstrate in Chapter 9 and Chapter 10 (and other works [46,47] have also shown) that a significant proportion of BLE Peripherals have either no security requirements or only Mode 1 Level 2. This enables an attacker to read BLE characteristic data from such devices. Apart from being an attack on confidentiality, this also has serious privacy concerns associated with it, particularly if the data is related to a user's health or lifestyle.

C.1.1.2 Exploit at app-layer In Chapter 5 we demonstrate that, on a multi-application Central platform (e.g., Android or iOS), once one application triggers pairing with a BLE Peripheral, any other application on the same Central device will be able to access pairing-protected characteristics from the Peripheral. This is similar to the result identified for Bluetooth Classic by Naveed et al. [48], but with greater attack potential. With this vulnerability, even if a Peripheral device specifies the strongest authentication requirements for its data, if it does *not* implement strong higher-layer security (via authorisation permissions), then a malicious application may be able to covertly read information from the Peripheral.

C.1.2 Obtaining cryptographic keys

If a BLE characteristic requires authenticated pairing before its value can be accessed, then an attacker would need to either (i) pair with the device and bypass authentication, or (ii) obtain the cryptographic keys exchanged by legitimate devices in order to read the characteristic data via a MitM, spoofing or eavesdropping attack. We describe authentication bypass mechanisms in C.1.3 and describe mechanisms by which an attacker can obtain cryptographic keys in this section. We elaborate on two different scenarios in which an attacker can obtain cryptographic keys that are shared by legitimate devices. The first scenario (C.1.2.1) assumes that the attacker is present when the legitimate devices are pairing and exploits protocol weaknesses to obtain the keys. The second scenario (C.1.2.2) assumes that the legitimate devices have already completed pairing (unobserved by the attacker) and describes mechanisms by which the devices can be forced to re-pair such that the attacker can eavesdrop on the protocol and derive the keys using techniques from C.1.2.1.

C.1.2.1 Cryptographic key theft during pairing The *Passkey Entry* association model in LE Legacy has a well-known pairing vulnerability that is fairly straightforward to exploit. Phase 2 in LE Legacy pairing (as depicted in Figure 2.4) shares the Master and Slave random numbers (MRand and SRand) in the clear. These are used, along with the Temporary Key (TK), as inputs to the key generation function `s1` to produce the Short Term key (STK), which is used to encrypt the remaining communications. An attacker who observes the pairing exchange between two BLE devices will therefore be able to obtain two out of three inputs to the key generation function. The third input, the TK, is only 6 digits long. It has a value of "000000" for *Just Works* pairing and also sometimes for *Passkey Entry* (if the device defines a static all-zero passkey). Even if the TK is not all-zero, it only has 20 bits of entropy, which can be brute-forced in less than a second [16]. The attacker can therefore compute the STK and use it to decrypt the remaining encrypted messages and obtain the LTK. This can then be used to decrypt all future communications between the two devices.[5]

While LE Legacy is undoubtedly less secure than LESC, Biham et al. [49] have identified a vulnerability in LESC as well, within the ECDH key authentication mechanism. Specifically, the protocol only authenticates the x-coordinate of the devices' public keys, and older versions of the specification did not specify a requirement for validating the y-coordinate. This meant that a MitM attacker could eavesdrop on the pairing protocol between two legitimate BLE devices, intercept each device's public key, modify the y-coordinate to zero, and forward it on to the intended recipient. This forces the possible key space to a very small set of numbers such that the attacker has a reasonable chance (25% to 50%) of computing the key [49].

Another attack that has the potential to impact LESC, specifically with *Passkey Entry* pairing, exploits the fact that the passkey is revealed bit-by-bit during Phase 2 of the pairing process. If a BLE device uses a fixed passkey (which is disallowed since v5.1 of the Bluetooth specification), an attacker will be able to learn the passkey by eavesdropping on one run of the pairing protocol, and use that knowledge to directly pair with the device or to MitM when the device pairs again [50]. This was originally shown for Bluetooth Classic, but was noted to be applicable for BLE as well [51].

An attack that affects LESC *and* LE Legacy pairing is the provision within the specification for key entropy reduction. That is, while the BLE specification supports encryption key sizes of up to 128 bits, it also has provisions for reducing the entropy down to 7 bytes (56 bits). A MitM attacker could intercept the communications during the pairing feature exchange phase and replace the value of the *maximum supported key size* with its minimum possible value of 7 bytes. If no checks are made by the communicating devices, then the entropy of the resultant key will be 56 bits, making it easier for the attacker to brute-force [52].

[5]Interestingly, this vulnerability was noted within the specification itself. Version 4.0 of the Bluetooth standard states with regard to LE pairing "*None of the pairing methods provide protection against a passive eavesdropper during the pairing process as predictable or easily established values for TK are used.*", and further "*A future version of this specification will include elliptic curve cryptography and Diffie-Hellman public key exchanges that will provide passive eavesdropper protection.*". LE Secure Connections, introduced in Version 4.2 of the Bluetooth specification, uses ECDH for key exchange. However, it is important to note that LE Legacy is still an option within the specification and that it is in widespread use.

Another attack that enables cryptographic key theft exploits a finding by Santos et al. [53] that BLE Peripherals have limited storage for bond information. The authors discovered that some of the mechanisms employed by Peripherals to handle full bond lists can result in the possibility for LTK theft. For example, on some Peripherals, once the bond list is full, subsequent connections will be unable to store bond information and will therefore have to pair each time. If this is the case, the attacker must fill up the bonding list *before* the legitimate Peripheral and Central attempt to pair. This will cause the pairing process to be undertaken every time, rather than LTKs being stored. The attacker would then be able to obtain the keys for each connection that they are able to eavesdrop on, using one of the aforementioned techniques. This assumes unauthenticated pairing (or the attacker would need physical access to the Peripheral).

C.1.2.2 Cryptographic key theft post-pairing This section describes mechanisms by which an attacker can force two legitimate BLE devices that have already paired to re-pair, such that the attacker can eavesdrop or MitM into the pairing exchange to obtain the keys, using the techniques described in C.1.2.1.

As previously mentioned in C.1.2.1, BLE Peripherals often have limited storage for bond information. In such scenarios, one technique employed by Peripherals is that, if the bond list is full, a new bond displaces an existing item on the list. Assuming a legitimate Peripheral and Central have completed bonding (say, with LTK = K_1), an attacker can make multiple pairing requests to the Peripheral using different MAC addresses and fill up the Peripheral's bond list. Once the list is full, the attacker makes an additional request, to displace K_1. This would force the legitimate Peripheral and Central to go through the pairing process again, at which point the attacker can eavesdrop and derive the key [53]. This again assumes unauthenticated pairing.

In addition to the above mechanism, assuming two legitimate devices have previously paired and bonded, and then disconnected, then when the devices later reconnect, a MitM attacker could force a re-pairing by interfering with the re-encryption procedure. That is, when two bonded devices reconnect, they go through a procedure of (plaintext) message exchanges prior to re-encrypting the link. If the attacker injects a message to falsely indicate that one of the legitimate devices has lost the LTK, this will force a key renegotiation that the attacker can eavesdrop on [16].

C.1.3 Bypassing authentication

We discuss within this section three subcategories of attacks by which an attacker might bypass authentication and access protected BLE data. The first subcategory exploits protocol weaknesses and involves the attack device interfacing with a single victim device. The second subcategory also exploits protocol weaknesses, but with the attack device in a MitM position between two legitimate devices. The third subcategory exploits software bugs.

C.1.3.1 Protocol weaknesses: direct device communication Rosa [15] identified that the LE Legacy confirm value generation function `c1`, as used during Phase 2 of the pairing process (depicted in Figure 2.4), did not bind the generated confirmation value to the random numbers used to generate it. Rosa demonstrated that it was possible for a malicious Slave device, with no knowledge of the passkey, to bypass passkey authentication. The malicious Slave device spoofs a legitimate device, such that the Master device connects to it. During the pairing exchange, the spoofed Slave device transmits a *random* confirmation value (SConfirm) and later brute-forces the correct passkey (i.e., TK) using information sent by the legitimate Master device. This is used to compute an appropriate SRand, which is sent to the Master, giving the Master device the impression that the spoofed Slave had knowledge of the passkey all along. In this way, a malicious Slave device (normally a Peripheral) may be able to read protected information from the Master device [15]. This attack assumes that the Master device displays the passkey, which should be entered on the Slave device (as otherwise, the Slave device would have knowledge of the passkey anyway). We observe that it could also be used with the OOB association model (with LE Legacy pairing), when the OOB key has the same entropy as with *Passkey Entry* and the malicious Slave device has no knowledge of the OOB data.

The next attack also involves a Peripheral device reading information from a Central. Wu et al. [54] and Zhang et al. [55] observed that on "typically Central" platforms such as Android, Windows, iOS and Linux, receiving a `0x06 Key Not Found` link layer encryption error caused the communications to be downgraded to plaintext. This is in accordance with the BLE specification. However, Wu et al. [54] additionally observed that the above-mentioned platforms do *not* adhere to the specification requirement that the error should be communicated to higher layers and that, upon receiving notification of such errors, pairing should be restarted after notifying the user. Assuming a legitimate Central and Peripheral have previously completed pairing, a fake Peripheral could connect to the legitimate Central device by spoofing the legitimate Peripheral (details on how spoofing can be achieved are provided in §3.8). When the encryption procedure is initiated, the spoofed Peripheral transmits a message with the `0x06` Error Code. This would cause the remaining communications to occur in plaintext. This could then be used to obtain sensitive data from the Central [54, 55].

C.1.3.2 Protocol weaknesses: MitM manipulation There are two separate BLE attacks, brought to light in 2020/2021, which enable an attacker to place themselves in a MitM position between two legitimate devices that undergo authenticated pairing, i.e., pairing that includes MitM protection. The first attack occurs at the time of pairing, while the second occurs after pairing has completed and the legitimate devices are communicating.

The first attack was disclosed by Tschirschnitz et al. [56], and is best demonstrated using LESC pairing models. The attack exploits the fact that the association model used during pairing is determined independently by the two devices that are undergoing pairing. There is no mutual confirmation of the association model at any point during the pairing procedure. Tschirschnitz et al. [56] demonstrate that it is possible for an attacker to exploit the visible artefacts of the *Passkey Entry* and *Numeric Comparison* pairing models (in particular, the fact that they both display a six-digit value on a device), such that the attacker pairs using *Passkey Entry* with one device and *Numeric Comparison* with the other device to enter a MitM position between the two legitimate devices. The user assumes that the two legitimate devices have paired with each

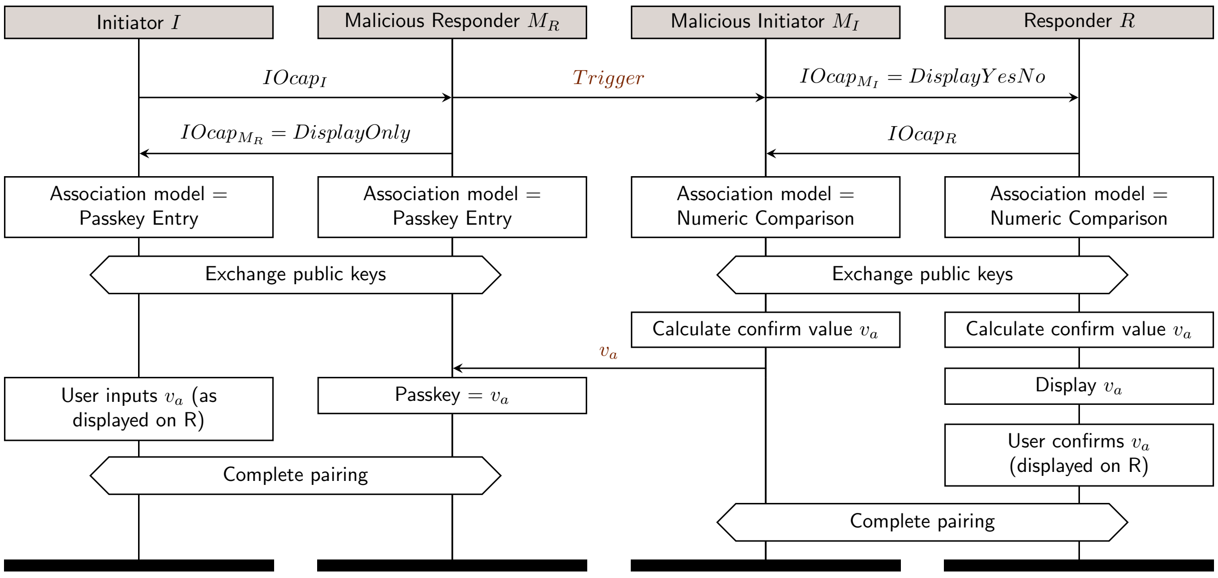

Initiator I
Malicious Responder M_R
Malicious Initiator M_I
Responder R
$IOcap_I$
$Trigger$
$IOcap_{M_I} = DisplayYesNo$
$IOcap_{M_R} = DisplayOnly$
$IOcap_R$
Association model = Passkey Entry
Association model = Passkey Entry
Association model = Numeric Comparison
Association model = Numeric Comparison
Exchange public keys
Exchange public keys
Calculate confirm value v_a
Calculate confirm value v_a
v_a
Display v_a
User inputs v_a (as displayed on R)
Passkey = v_a
User confirms v_a (displayed on R)
Complete pairing
Complete pairing

Figure 3.2: MitM attack by manipulating association model.

other. This attack is depicted in Figure 3.2.

The second attack exploits vulnerabilities in older versions of the Cross Transport Key Derivation (CTKD) mechanism in Bluetooth. CTKD is intended to allow dual-transport devices (i.e., devices that support Bluetooth Classic *and* BLE) to communicate with each other over both transports after pairing over only one. When pairing occurs over one transport, a key can be derived for the other transport using a CTKD algorithm. Antonioli et al. [57] and researchers at Purdue University independently discovered that CTKD-derived keys will overwrite any preexisting key, even if the preexisting key was generated using stronger pairing mechanisms. This gives rise to the possibility of MitM attacks as follows [57]:

- Assume two legitimate dual-transport devices, $DevA$ and $DevB$, have paired over Bluetooth Classic using a pairing mechanism that includes MitM protection (with LTK = $LTK_{AB_{Classic}}$) and are currently communicating with each other.
- An attacker can impersonate $DevA$ and pair with $DevB$ over the BLE transport (specifying no input-output capabilities). This results in LTK generation for BLE and triggers CTKD, which will generate an LTK for the Bluetooth Classic transport, overwriting $LTK_{AB_{Classic}}$. The communication between $DevA$ and $DevB$ is disrupted.
- The attacker impersonates $DevB$ and sends a pairing request to $DevA$ over Bluetooth Classic. $DevA$ completes pairing with the attacker under the assumption that it is pairing with $DevB$. The attacker is now in a MitM position between $DevA$ and $DevB$ and can access BLE characteristic data.

However, we observe that if the BLE characteristic data specifies a requirement for authenticated pairing, then this attack will not be effective, as the attacker will have used unauthenticated pairing. Further, if the BLE characteristic data specifies no security requirements or only unauthenticated pairing (i.e., Mode 1 Level 1 or Mode 1 Level 2), then the attacker will be able to access the data even without exploiting CTKD, using more straightforward methods (see C.1.1.1). The impact of this attack *on BLE* is chiefly the interruption of an existing legitimate connection and enabling the attacker to MitM between two devices that were already communicating with each other.

C.1.3.3 Exploit software bugs Garbelini et al. [58] identified two software bugs in certain Telink chipsets which, when combined, would enable authentication bypass. One was that a zero default LTK was used in LESC pairing. The second bug was that the chipsets accepted an out-of-order encryption request prior to pairing completion. An attacker device can initiate pairing with a vulnerable Peripheral and, once it receives the `Pairing Response`, it can send an `Encryption Request`. The Peripheral transmits an `Encryption Response`. Each of these messages contain half of what is known as a Session Key Diversifier, which together with the LTK are used to derive a session key. The Peripheral then sends a `Start Encryption Request`, which expects a response encrypted with the session key. Because Telink Peripherals default to a zero LTK, the attacker device can easily compute the session key and send a valid encrypted `Start Encryption Response`. This will be accepted by the Peripheral and communications will continue. The attacker can now access data on the Peripheral that has a security requirement of LESC protection, without actually going through LESC pairing [58].

3.5 Tampering

Tampering in this work refers to either tampering with BLE characteristic data or tampering with the device itself.

I.1 Tampering with BLE data

Tampering with BLE characteristic data is accomplished by exploiting the same vulnerabilities as for unauthorised acquisition of data, detailed in C.1, when characteristics are writable. That is, if a device's characteristic data is unprotected, or if the attacker is able to obtain the LTK and MitM into a legitimate connection, or if the attacker can bypass authentication, then any writable characteristics can be modified by the attacker. This could lead to false data being presented to the user, control commands being issued (such as if HID data is modified), or device functionality being manipulated such that user safety is endangered (such as if BLE door locks, eScooters or insulin pumps are tampered with).

I.2 Tampering with the BLE device

There are two mechanisms by which an attacker could tamper with a BLE device: by exploiting a firmware update mechanism, or by exploiting coding bugs.

I.2.1 Exploiting firmware updates

Updates to the firmware of a BLE device can be made over the BLE interface (if supported by the chipset). This is achieved by writing small amounts of firmware code at a time to a special (vendor-specific) characteristic. If the update mechanism is not suitably protected, then it may be possible for an attacker to replace the firmware with one of their choosing, thereby completely compromising the system. We demonstrate this in §6.7.

I.2.2 Exploiting software bugs

Some vulnerabilities identified with specific BLE chipsets or platforms (e.g., [58,59]) cause overflows when exploited. These may also enable an attacker to accomplish code execution on the implementing platform via carefully crafted packets [58–60]. It has been demonstrated in one case that an attacker can gain control of a device by sending multiple apparently-benign messages, each with a small amount of shell code, prior to the actual overflow message [59].

3.6 Denial of Service

We now look at Denial of Service (DoS) attacks against BLE. We consider three specific types of attacks: (i) attacks where the device itself is rendered unavailable (due to bricking, crashing, restarting or battery depletion), (ii) attacks where the device is up but peer devices are prevented from connecting to or communicating with it, and (iii) attacks where packets are dropped.

We note that the simplest form of DoS attack would be to transmit a jamming signal on the 2.4 GHz part of the spectrum. However, this is a fairly crude method, which would affect *all* devices that use the spectrum. We do not explore this attack further in this thesis. However, we do discuss *selective* jamming in A.2.1.1.

A.1 Rendering a device unavailable

Two main types of attacks have been identified which can render a BLE Peripheral unavailable: through battery exhaustion or by exploiting a software bug. We note that the attacks result in the device being unavailable only temporarily, unless they are sustained for long periods.

A.1.1 Battery exhaustion

A BLE Peripheral, when not connected to a Central device, transmits advertising messages periodically but is otherwise inactive. This enables it to conserve battery power. Connecting to a Central device and transmitting or receiving data are actions that use up the battery. For this reason, BLE is intended to be used in situations where data is transferred in short bursts over transient connections, rather than being streamed over long-lived or frequent connections. Issuing a large number of connection requests (which the Peripheral will need to process if it does not implement a whitelist and filtering policies) over a period of time will keep the BLE Peripheral active for a very long time and will drain the Peripheral's battery much faster than with standard operations [18,61]. This could have serious ramifications in certain scenarios (e.g., if the Peripheral performs critical monitoring functions).

A.1.2 Exploit software bugs

A number of software bugs have been identified in BLE chipsets and implementing platforms, mainly to do with insufficient validation of input lengths and improper validation of incoming packets. If exploited, these bugs can lead to device instability, deadlocks, crashes or restarts. For the most part, these vulnerabilities only lead to temporary denial of service, but some may require manual power-cycling in order to be resolved [58–60].

A.2 Preventing peer device connection/communication

An attacker can employ different methods to prevent a legitimate BLE device from connecting to a legitimate BLE peer. One technique is to hide a Peripheral from a legitimate Central, such that the Central cannot connect to it (see A.2.1). Another method is to lock a legitimate peer out by interfering with the cryptographic keys that are used to encrypt data (A.2.2).

A.2.1 Peripheral hiding

A Peripheral can be hidden from a Central device by selectively jamming the Peripheral's advertisements, by stopping the Peripheral from advertising, or by reducing the "visibility" of the Peripheral's advertisements.

A.2.1.1 Selective jamming If a BLE Peripheral utilises a random static or public advertising address (or if it incorrectly implements private addresses, such that the address remains unchanged for a long duration), then it will advertise with a fixed address for extended periods of time. Bräuer et al. [21] have identified that fixed advertising addresses can be used by an attacker to selectively jam the advertisement of only a single Peripheral device. The jammer scans each advertising channel, looking for the preamble and the advertising access address (`0x8e89bed6`). Once those are detected, it continues reading bits until it finds the advertiser's address. If the address matches a programmed address, the jammer transmits a brief noisy signal on the advertising channel. This will prevent the Peripheral's advertisements from being received by a scanning Central device, thereby effectively denying the Central the possibility of connecting with the Peripheral [21].

A.2.1.2 Stop Peripheral advertisements A BLE Peripheral only advertises when it is not in a connection. To stop a Peripheral from advertising, an attacker device can initiate a connection to it. This would prevent a legitimate Central from being able to find and connect to the Peripheral. The attack would need to be conducted in a loop for the effect to be sustained [22].

A.2.1.3 Reducing Peripheral visibility Jasek [19] observed that the "visibility" of a BLE Peripheral was apparently connected with its advertising frequency (although some doubt has been cast upon this finding in subsequent work [62]). Most legitimate Peripherals do not advertise too rapidly, to conserve their battery. Therefore, if an attack device can spoof the legitimate Peripheral but advertise rapidly, it will be more "visible" to the Central, causing the legitimate Central to connect to the spoofed Peripheral rather than to the legitimate Peripheral.

A.2.2 Device lockout via cryptographic key manipulation

We discuss two methods by which an attacker can lock a legitimate BLE peer out by interfering with pairing LTKs. In both cases, the result is likely to be a permanent Denial of Service unless the affected devices are reset or the existing LTKs are cleared.

A.2.2.1 Exploiting encryption error mishandling This attack exploits the `0x06 Key Not Found` encryption error mishandling vulnerability described in C.1.3.1. It assumes that two legitimate devices have previously paired and bonded, but have subsequently disconnected. At this stage both devices share the same LTK (let this be LTK_{good}). An attack device can spoof the legitimate Peripheral, connect to the legitimate Central, and then exploit the encryption error mishandling vulnerability to downgrade the link as described in C.1.3.1. The spoofed Peripheral indicates that its characteristics require encryption, which triggers the Central to start the pairing process. This results in the legitimate Central and spoofed Peripheral sharing an LTK, LTK_{attack}, which the legitimate Central assumes is the LTK that it shares with the legitimate Peripheral. If the spoofed Peripheral now disconnects and allows the legitimate devices to connect, the legitimate devices will find that they are no longer able to communicate because they do not share the same key. The legitimate Peripheral still uses LTK_{good} while the legitimate Central uses LTK_{attack}. Neither device will throw an error because each will believe itself to be in possession of a valid LTK [55].

A.2.2.2 Resource exhaustion This is a variation of the resource exhaustion attack described in C.1.2.2. As mentioned there, Peripheral devices tend to have limited storage for bond information. In some cases, once the bonding list is full, new connections are ignored [53]. In this scenario, the attacker could fill up the bonding list, thereby causing new (legitimate) requests to be rejected.

A.3 Dropping packets

The dropping of legitimate packets within a connection can be considered a form of Denial of Service, and is accomplished by an attacker who is able to MitM into a legitimate connection. While dropping packets at random is always possible, we consider the scenario where an attacker is able to examine and drop *specific* packets.

A.3.1 Dropping packets via MitM attacks

In this section we outline how MitM packet dropping attacks can be accomplished in two scenarios: (i) when there is no security requirement (i.e., pairing) and (ii) when pairing is required.

If no pairing or bonding is required between the legitimate devices, then all the attacker need do is to spoof the legitimate Central and connect to the legitimate Peripheral, then spoof the legitimate Peripheral and wait for a connection from the legitimate Central (see §3.8). In this way, the attack device can find its way into a MitM position. After this point, the attack device can examine and drop any packets it chooses.

If the legitimate devices *do* require pairing and bonding, then additional effort will need to be expended by the attacker. Two MitM attacks are described in C.1.3.2. If these are not possible, the attacker would have to perform one of the attacks described in C.1.2 to retrieve the LTK or session key used by the legitimately bonded BLE peers. During reconnections between the legitimate devices, the attacker would MitM as in the unauthenticated case. To be able to actually read the contents of the packets, however, the attacker would need to know the session key. This is derived from the LTK (which the attacker has obtained) and key diversifying material (which is exchanged in the clear). The attacker will thus be able to compute the session key, such that they can examine the contents of packets and drop any as required.

3.7 Profiling & Tracking

In this section, we discuss the leakage of data from BLE devices that can enable device or user tracking, and expose a user's activity or profile. This differs from unauthorised data access in that the information that is leaked is not BLE data. It is extraneous data that nevertheless reveals information regarding the user or device.

P.1 Device tracking

Being able to track a BLE-enabled device is a privacy concern for the user, because many such devices are worn/carried by the user, i.e., are always on or about the user's person. Therefore, tracking a BLE device may, in these cases, be synonymous with tracking the user.

There are two main techniques that an attacker can employ to track a BLE-enabled device: one is to use the advertising address (P.1.1), and the other is to use data within advertising messages or the device's configuration (P.1.2).

P.1.1 Tracking BLE devices using advertising addresses

As described in §2.1.1.3, BLE advertising messages contain, among other information, an advertising address. The BLE specification allows for periodically-changing private addresses (see §2.2.2.1) in order to enable device privacy. This section describes mechanisms by which a device can be tracked using only its device address (whether private or otherwise).

P.1.1.1 Tracking using fixed addresses Several studies [9, 23, 63] have found that many BLE Peripheral devices do not actually use private addresses, i.e., most used fixed addresses. Our findings from performing firmware analysis against BLE binaries (described in Chapter 10) support these studies. A device that periodically transmits an advertising message with a fixed address opens itself (and its user) to the risk of tracking [9, 23, 64]. Issoufaly et al. [65] state that the tracking could even be performed on a large scale using multiple compromised devices, i.e., a botnet, which would collect and periodically transmit BLE advertisements to an attack server, which could then analyse the data and infer movement patterns for one or more users.

P.1.1.2 Tracking using private addresses

As mentioned in P.1.1.1, most resource-constrained BLE Peripherals do not utilise private addresses. In contrast, BLE-enabled Central platforms such as mobile phones and laptops *do* tend to implement resolvable private addresses. However, as described in C.1.3.1, many of these devices exhibit a vulnerability in which they don't handle encryption errors properly. Zhang et al. [55] describe an attack to exploit this vulnerability in such a way as to enable tracking of the victim Central device: an attack device spoofs a legitimate Peripheral, causing the legitimate Central to connect to it. The spoofed Peripheral issues a `0x06 Key Not Found` link layer encryption error, which downgrades communications to plaintext. The spoofed Peripheral specifies no IO capabilities and Mode 1 Level 2 authentication requirements for its data, which will trigger *Just Works* pairing, and will enable the attacker to steal the Central device's Identity Resolving Key (IRK) and identity address, so that the Central device can thereafter be tracked.

We observe that, if a legitimate BLE Peripheral *does* employ resolvable private addresses but allows for unauthenticated pairing, i.e., *Just Works*, then any malicious device could connect and pair with the device and obtain the IRK in a straightforward manner (since *Just Works* pairing does not require user interaction and may occur without the user's knowledge).

P.1.2 Tracking BLE devices using the service list

Advertising messages often include a list of the BLE services that are available on the advertising device. In the case of beacon-style devices, a single fixed UUID may be used. According to several studies [41, 66, 67], the BLE service UUIDs broadcast in advertising messages may be used to fingerprint device models and manufacturers.

Celosia et al. [46] describe an extension of this attack, exploiting the fact that the service list on a device is freely accessible, as per the BLE specification. An attacker could connect to the device and enumerate all services and characteristics, thereby obtaining a more exact fingerprint for the device. A large-scale study has shown that this is practically feasible, and that the obtained data could be used to track a user [46]. However, this attack requires connecting to every device rather than simply monitoring advertisements.

This type of attack may also be possible with other information within advertising messages. In particular, Becker et al. [68] and Celosia et al. [69] demonstrated that the *Manufacturer Data* and *Device Name* fields included within the advertisements of various devices (including Windows 10 and iOS/MacOS devices) enable identification and tracking of the device.

Note that all of these attacks enable tracking even in the presence of private addresses.

P.2 User/activity detection

In this section, we describe traffic analysis attacks to determine the activity levels of a user and to identify a user from within a group.

P.2.1 Traffic analysis

During a BLE connection, if no data is available, then devices may exchange empty packets. Das et al. [9] observed that the volume of empty packets corresponded to the level of activity of a user in the case of BLE-enabled fitness trackers. This means that simply monitoring BLE traffic may allow an attacker to infer a user's activities. This is true even if the packets are encrypted, as it is the *volume* of traffic that provides an indication of activity.

Further, it was also identified that BLE traffic differed between different users (for the same set of fitness trackers). This means that traffic analysis could also be used to identify a single user from a small group of fixed users, by first training a machine learning model and later monitoring BLE traffic and testing it against the model [9].

P.3 Obtaining user/device profile

This section discusses attacks that divulge information regarding a user (particularly their health, lifestyle or device usage) via their BLE devices.

P.3.1 Exploiting open access to list of services and characteristics

As mentioned previously, the list of services and characteristics on a BLE device are always freely readable. Several BLE services have known meanings (e.g., Heart Rate Service, Glucose Service). A malicious application on a user's Central device could connect to and enumerate all services of all the devices in the user's vicinity. Combining this with signal strength information could provide a reasonable indication of which devices are the user's own. Depending on the types of devices, the attacker might be able to learn about the user's health and general activities (e.g., if the Glucose Service is present, then the user is likely to be diabetic).

P.3.2 Protocol observation

Sun et al. [70] observe that in accordance with the Bluetooth specification, the public keys used by a BLE device for LESC pairing may not change for long periods of time. That is, the device will use the same public key for every new pairing it enters into. An attacker could observe a device's pairing exchange and store the public key, which is exchanged in unencrypted form. If the same public key is observed again in another pairing exchange, the attacker will conclude that the same device is involved. In this manner, an attacker will be able to build a communication profile for a device [70].

3.8 Spoofing

If an attacker is able to spoof a legitimate BLE device to its (legitimate) BLE peer, then the attacker would be able to read data from and inject false data to the peer device, issue commands, and perform MitM attacks. It could do all of this while the BLE peer (and user) assumes that the actions were performed by the legitimate device. In the case of BLE beacons, spoofing attacks can also result in incorrect assumptions regarding user location [20].

Throughout this section, we will use $DevA$ and $DevB$ to denote two legitimate devices in our illustration of spoofing attacks. $DevM$ is an attacker device which spoofs $DevA$ so that it can connect to or otherwise interfere with $DevB$. $DevM$ may also spoof $DevB$ so that it can perform MitM attacks between $DevA$ and $DevB$.

A spoofing attack will require three parts, each of which will require the exploitation of one or more vulnerabilities: (i) cloning the BLE device address (since that is what is typically used to identify a device to its peer) and potentially also the advertising data, (ii) cloning the list of services and characteristics, and (iii) spoofing the data, i.e., characteristic values. Depending on the attack goal, it is possible that only the first one or two parts may suffice.

Cloning the BLE device address and advertising data If $DevA$ does not use resolvable addresses (see §2.2.2.1), then address spoofing is straightforward, as it only requires a freely-available address changing tool [71]. If resolvable addresses *are* used, then an attacker would need to obtain $DevA$'s identity address and IRK in order to be able to spoof $DevA$. This can be accomplished using the techniques described in P.1.1.2. After this point, the attacker can set up $DevM$ with the same address as $DevA$.

If $DevA$ is a Central device, then $DevM$ can now spoof $DevA$ and initiate a connection with $DevB$. If $DevA$ is a Peripheral or Broadcaster, then $DevM$ will spoof $DevA$ and clone the advertising data. In the event that $DevA$ is a beacon, its advertisements will contain a UUID identifying the beacon. If this UUID is static, then the UUID will also be cloned [20]. If $DevA$ is not in the vicinity, then $DevM$ will be able to advertise as $DevA$ and will be assumed to be $DevA$ by $DevB$. If $DevA$ *is* advertising in the vicinity, then $DevM$ will need to ensure that its own advertisements (instead of $DevA$'s) are seen by $DevB$. This can be achieved using the techniques described in A.2.1.

Cloning BLE services and characteristics If $DevM$ is to exchange data with $DevB$, then it will need to implement the same set of services and characteristics as $DevA$. The list of services and characteristics is always freely readable, as per the BLE specification. This enables the attacker to first connect to $DevA$, enumerate the service list, and then set up $DevM$ with the same service list.

Cloning BLE data If $DevM$ is to exchange data with $DevB$, particularly in a covert manner, then $DevM$ will need to know acceptable values for the characteristics that it spoofs. Otherwise, it is possible that $DevB$ will ignore the data sent by $DevM$ or exhibit unstable behaviour. Similarly, if $DevM$ wishes to perform MitM attacks between $DevA$ and $DevB$, then it will need to be able to read and possibly also modify the exchanged data.

If $DevA$ applies no protection for its characteristics, or only specifies Mode 1 Level 2 requirements, then spoofing is straightforward. The attacker can either connect to $DevA$ (silently pairing with *Just Works*, if needed) and access the data, or monitor legitimate traffic between $DevA$ and $DevB$ to identify possible characteristic values and apply them to $DevM$. If, on the other hand, $DevA$ requires *Passkey Entry* or *Numeric Comparison* pairing, then the attacker will need to first obtain cryptographic keys (using one of the techniques detailed in C.1.2, assuming $DevA$ is vulnerable) and then access $DevA$'s data.

3.9 Applicability of Bluetooth Classic Attacks

While Bluetooth Low Energy (BLE) shares many similarities with its predecessor, the two technologies are supposed to be considered as distinct and incompatible [2]. Several aspects of Bluetooth Classic are only available in a simplified form in BLE, and many other Bluetooth Classic features are not available in BLE at all. Therefore, many of the vulnerabilities and associated attacks in Bluetooth Classic will not be applicable to BLE. However, we have observed some studies attributing Bluetooth Classic attacks to BLE. In this section, we outline some well-known Bluetooth Classic attacks and explain why they are not applicable (or are less applicable) to BLE. Note that this is not intended to be a comprehensive overview of Bluetooth Classic vulnerabilities.

3.9.1 Bluejacking, Bluesnarfing, Bluebugging

Bluejacking is an attack on insecure implementations of Bluetooth in which unsolicited messages (or images or sounds) are sent to Bluetooth-enabled devices via the Object Exchange Protocol (OBEX). Bluesnarfing is similar to Bluejacking in that it targets insecure implementations of Bluetooth and uses the OBEX protocol. However, it issues a “get” request to steal files with known names. With Bluebugging, the attack again begins by sending a message via Bluetooth. The transmission is then paused. On devices with faulty Bluetooth implementations, the attacker device is added to the victim device's trust list, enabling the attacker to thereafter connect to the victim device's Bluetooth headset and issue commands to control the victim device [72].

OBEX is supported by Bluetooth Classic but not BLE, and being able to send unsolicited messages in the above-described manner is inapplicable to BLE, which means that these attacks are not applicable in the Low Energy setting.

3.9.2 Bluetooth Impersonation AttackS (BIAS)

BIAS attacks are impersonation attacks that target Bluetooth Legacy and Secure Connections pairing (but not the LE versions). They exploit the fact that in Bluetooth Legacy pairing, authentication is unilateral; they also exploit the possibility for performing Master-Slave role switching in the middle of a connection [73]. In LE pairing (Legacy and Secure Connections), authentication is mutual (albeit not vulnerability-free). Further, role switching is not a feature in BLE. Therefore, these particular attacks are not applicable to BLE.

3.9.3 BlueBorne

BlueBorne [74] was a suite of vulnerabilities identified in different widely-used Bluetooth Classic implementations. Vulnerabilities were identified in various layers of the Bluetooth stack. We discuss each vulnerability in brief below, primarily focusing on why it is not applicable to

BLE. Note that BlueBorne described a Security Manager vulnerability in Bluetooth Classic whereby temporary pairings could take place without the user's knowledge. We observe that this particular issue may be true in the case of BLE as well. If a BLE device has a maximum security requirement of Mode 1 Level 2 applied to it or its characteristics, then it may be possible to covertly pair with the device.

L2CAP Linux's BlueZ stack contained a vulnerability in its implementation of the L2CAP Extended Flow Specification, which enabled a stack overflow. This is not applicable in the case of BLE because BLE only uses three fixed L2CAP channels, and no Extended Flow Specification.

SDP Linux's BlueZ stack and Android each contained an information leak vulnerability in their implementation of the Service Discovery Protocol (SDP). SDP is only used in Bluetooth Classic, and is not available in BLE. This vulnerability is therefore not applicable to BLE.

BNEP Android was found to contain two remote code execution vulnerabilities in its implementation of the Bluetooth Network Encapsulation Protocol (BNEP). However, since BNEP is not used in BLE, these vulnerabilities do not apply.

3.9.4 Pairing Vulnerabilities

Numerous vulnerabilities have been identified for the different types of Bluetooth pairing. We observe that Bluetooth Classic Legacy pairing vulnerabilities are not applicable to BLE, due to the difference in the pairing protocol. However, Bluetooth Classic's Secure Simple Pairing (SSP) is very similar to LESC. Therefore, vulnerabilities identified for SSP need to be analysed carefully to determine their impact on LESC. For example, the fact that the passkey is revealed bit-by-bit during the pairing protocol of SSP is equally applicable to LESC (see C.1.2.1).

In some cases, vulnerabilities exist in Bluetooth Classic and BLE, but with different impacts. The key entropy downgrade vulnerability is a good example. Both Bluetooth Classic and BLE allow for the entropy of the encryption key to be reduced, and the maximum supported key size is exchanged as a feature during the pairing protocol. However, while in BLE the minimum possible key entropy is 56 bits, in Bluetooth Classic the key can be downgraded to a single byte (8 bits) of entropy. Therefore, the key can be brute-forced far more easily for Bluetooth Classic.

Pairing downgrade attacks (distinct from key entropy downgrade attacks) may be conducted against both Bluetooth Classic and BLE. This is accomplished by a MitM attacker manipulating the IO capabilities exchanged by the two devices to force them to use the *Just Works* association model. However, in the case of BLE, this does not automatically give the attacker access to BLE data. If the BLE characteristics specify a security requirement of "authenticated encryption", i.e., Mode 1 Level 3, then even if the link has been downgraded, the characteristic values will not be accessible.

3.10 Chapter Summary and Next Steps

In this chapter, we introduced our BLE attack taxonomy and described the various attacks against BLE security and privacy, focusing on the attack impact or outcome. In Chapter 4, we discuss the vulnerabilities that give rise to these attacks. With vulnerabilities, we focus on understanding the root cause, such that similar vulnerabilities can be prevented in future. We therefore perform a per-layer, per-stakeholder classification.

4 Vulnerability Analysis

In this chapter, we discuss the vulnerabilities that give rise to the attacks described in Chapter 3. We map each vulnerability to one or more attacks and present an analysis of the vulnerabilities according to their source. We also briefly survey proposals for enhancing the security or privacy of BLE deployments in the presence of one or more vulnerabilities.

4.1 Vulnerabilities in BLE

In our categorisation of BLE attacks (§3.3), we used a taxonomy that prioritised understanding the *impact* of attacks. When classifying vulnerabilities, we instead want to identify the *root cause*. There are two considerations that we take into account: (i) identifying the root cause in terms of *vulnerability location*, and (ii) identifying the root cause in terms of *responsible stakeholders*.[1]

For the first, i.e., to analyse vulnerability location, we organise the vulnerabilities according to the architectural layer they are present in, to be able to group related vulnerabilities, better understand related concerns and identify gaps in current research trends.

To determine responsible stakeholders, we analyse whether vulnerabilities occur within the BLE specification itself, as opposed to within implementations. We clearly state the source of vulnerability wherever possible as one of the following three options.

1. **Specification Issue** [**S**]: The problem lies within the specification itself. Any standard-compliant device may be vulnerable.

2. **Product Design Issue** [**D**]: The problem lies with end product implementations, usually addressing non-utilisation of security features. It will affect products from a subset of vendors.

3. **Coding Issue** [**C**]: The problem lies with end product implementations, specifically coding bugs, and will affect products from a subset of vendors.

In one instance (V_{GAP5}), the source of vulnerability is unclear, because it is not yet fully understood how advertising frequency and the "visibility" of the Peripheral are linked. This particular vulnerability is therefore categorised as an **Unclassified** [**U**] vulnerability source.

[1]Note that the dataset we use here is the same as that for the attack taxonomy. With each work (i.e., publication, white paper or CVE), we examine the data to look for explicit mentions of attacks and vulnerabilities or, in cases where only one is explicitly mentioned, infer one from the other. We also cross-check attacks and vulnerabilities from different sources to determine whether one vulnerability can give rise to multiple attacks or vice versa. We discussed the attacks in Chapter 3 and analyse the vulnerabilities in this chapter.

Note that with Product Design and Coding Issues, we only include vulnerabilities that affect a wide range of products (as would be the case if a chipset or widely-used operating system was vulnerable), rather than where the issue affects only a limited number of end products (e.g., [75–77]).

Tables 4.1 through 4.8 summarise the BLE vulnerabilities that have been identified thus far, organised by BLE architectural layer. Each vulnerability has an identifier (denoted by V_{ID}), which is used to cross-reference the vulnerability. For each vulnerability, we also present the vulnerability source (Specification vs. Design vs. Coding, denoted by V_{Src}), and the scope or applicability for the vulnerability (i.e., how many devices might be expected to be vulnerable). For the sake of completeness, we include associated CVEs and "code names" (denoted by V_{CN}), potential solutions, and related tools or frameworks. The tools are either for testing the presence of a vulnerability, for exploiting it or for mitigating it. Tools marked with a circle [$\circ$] are not publicly available. Tools marked with a star [$\star$] were developed by us.

Table 4.9 maps these vulnerabilities to the attacks described in §3.4 through §3.8. It shows that a single vulnerability can give rise to multiple attacks, particularly if the vulnerability enables the theft of cryptographic keys (which would then enable reading and tampering of data, packet dropping, and spoofing). Table 4.10 summarises the number of vulnerabilities in each layer that can be assigned to the different sources (i.e., Specification, Design, Coding and Unknown).

4.2 Architectural Analysis and Research Gaps

Examining the data in Tables 4.1 through 4.8, we note that the source stakeholder differs for vulnerabilities identified in different layers. For example, vulnerabilities identified with the Security Manager layer are more focused on specification issues, with 8 out of 11 SMP vulnerabilities being at least partly rooted in the specification; in contrast, 80% of vulnerabilities identified for GAP were implementation-related.

We could also approach this differently and hypothesise that the design of GAP has not been subjected to sufficient scrutiny, and in the same way that there has been insufficient testing of Security Manager implementations. By approaching the data in this manner, we can identify even more gaps in research, which could be the subject of future work.

We believe that the nature of research conducted thus far, particularly in terms of disparity of coverage for different layers, may be related to the existing expertise within the security community, as well as to the "ease of access" to each layer. For example, analysing Security Manager protocols is similar to most other network security or cryptographic analyses, which have been conducted for several decades [78]. In fact, 80% of all specification-related vulnerabilities were identified within SMP, and around half of these studies were protocol analyses. Further, there is easy access to certain components within a BLE system, such as LL/GAP advertisements and GATT messages (which can be obtained by sniffing traffic over the air interface, or even by using a mobile developer API), whereas "inner" components such as HCI and L2CAP cannot be easily interfaced with at the moment.

Table 4.1: Security and privacy vulnerabilities within the Physical Layer.

$\mathbf{V_{ID}}$	**Vulnerability**	$\mathbf{V_{Src}}$	**Applicability**	**(Potential) Solution**	**Notes**
V_{PHY1}	The volume of empty data packets exchanged during a connection can be correlated with user activity [9].	D	Fitness trackers.	Transmit dummy packets during low-intensity activities [9].	***Tools:*** `Ubertooth` [79], `nRF sniffer` [80], `btlejack` [81] can sniff BLE communications over the wireless interface. Code for inferring the user/activity has not been made available.

Table 4.2: Security and privacy vulnerabilities within the Link Layer.

$\mathbf{V_{ID}}$	**Vulnerability**	$\mathbf{V_{Src}}$	**Applicability**	**(Potential) Solution**	**Notes**
V_{LL1}	Not implementing LL whitelisting and filtering, which is intended to limit the number of connections that a BLE device need process [18].	D	Device specific.	Implement whitelisting and filtering, along with resolvable private addresses and strong pairing requirements.	Implementing only whitelisting is insufficient as an attacker may spoof the whitelisted address [19].
V_{LL2}	Improper handling of invalid length field in advertising packets (see Figure 2.3a) [59, 82].	C	Various Texas Instruments BLE stacks.	Code fix.	$\boldsymbol{V_{CN}}$***:*** BleedingBit ***CVEs:*** CVE-2018-16986, CVE-2019-15948
V_{LL3}	Improper handling of invalid length field in data packets (see Figure 2.3b) [58].	C	Various Cypress, NXP and Dialog Semiconductor BLE SDKs.	Code fix.	$\boldsymbol{V_{CN}}$***:*** SweynTooth ***CVEs:*** CVE-2019-16336, CVE-2019-17519, CVE-2019-17517 ***Tools:*** `Sweyntooth` [83]
V_{LL4}	No input validation to ensure that data packets with empty LLID fields are not accepted (see Figure 2.3b) [58].	C	Various Cypress and NXP BLE SDKs.	Code fix.	$\boldsymbol{V_{CN}}$***:*** SweynTooth ***CVEs:*** CVE-2019-17061, CVE-2019-17060 ***Tools:*** `Sweyntooth` [83]

$\mathbf{V_{ID}}$	**Vulnerability**	$\mathbf{V_{Src}}$	**Applicability**	**(Potential) Solution**	**Notes**
V_{LL5}	Peripheral accepts an out-of-order LL Encryption request and prompts the encryption of packets *before* successfully completing the pairing process [58].	C	Telink BLE SDK v3.4.0 for TLSR8258.	Code fix.	***V_{CN}:*** SweynTooth
V_{LL6}	Insufficient buffer allocated to copy incoming BLE packets [60].	C	Various Cypress chipsets.	Code fix.	***CVEs:*** CVE-2019-13916

Table 4.3: Security and privacy vulnerabilities within HCI.

$\mathbf{V_{ID}}$	**Vulnerability**	$\mathbf{V_{Src}}$	**Applicability**	**(Potential) Solution**	**Notes**
V_{HCI1}	The length of BLE advertising and scan response data (as set by higher layers) not checked to ensure it is below the allowable maximum [84].	C	Android v8.0, 8.1 and 9.0.	Code fix.	***CVEs:*** CVE-2019-2032

Table 4.4: Security and privacy vulnerabilities within L2CAP.

$\mathbf{V_{ID}}$	**Vulnerability**	$\mathbf{V_{Src}}$	**Applicability**	**(Potential) Solution**	**Notes**
V_{L2CAP1}	Unexpectedly short L2CAP packets (outside the range of 4-31 bytes) are not rejected [58].	C	Microchip ATSAMB11 BluSDK Smart up to v6.2.	Code fix.	***V_{CN}:*** SweynTooth ***CVEs:*** CVE-2019-19195 ***Tools:*** `Sweyntooth` [83]

Table 4.5: Security and privacy vulnerabilities within SMP.

$\mathbf{V_{ID}}$	**Vulnerability**	$\mathbf{V_{Src}}$	**Applicability**	**(Potential) Solution**	**Notes**
V_{SMP1}	Pairing key components are transmitted in the clear in LE Legacy [16, 85].	S	BLE devices using LE Legacy pairing with *Just Works* or *Passkey Entry*.	Use LESC, which utilises ECDH for authentication and key generation.	***Tools:*** `crackle` [86] analyses and decrypts LE Legacy-protected BLE data.
V_{SMP2}	Confirm value generation function `c1` used in LE Legacy does not bind the generated confirmation value to the random number [15].	S	BLE devices using LE Legacy pairing with *Passkey Entry*.	Use LESC, which utilises different confirm value generation functions.	
V_{SMP3}	No mutual confirmation of association model during pairing [56].	S	All BLE devices. Most impact for LESC.	Modify pairing protocol to mutually agree on association model [56].	At present, the only option is user vigilance to ascertain the association model used during pairing.
V_{SMP4}	Keys derived for a transport using CTKD overwrite any existing key, even if new key is less secure [57].	S+D	Dual-stack devices that support CTKD and specification <v5.2.	Do not overwrite existing keys if new keys are of lower strength.	
V_{SMP5}	The BLE specification does not require public keys used for ECDH in LESC to change frequently, and platforms may not change the keys either [70].	S+D	BLE devices supporting LESC.	Change public key after some n pairing attempts [70].	
V_{SMP6}	LESC requires only authentication of the x-coordinate of the elliptic curve point. Older versions of the specification do not even require *validation* of the y-coordinate. Most implementations do not proactively validate the y-coordinate [49].	S+D	BLE devices supporting LESC and specification <v5.1.	Validate both x- and y-coordinates of received elliptic curve point.	

V$_{\text{ID}}$	**Vulnerability**	**V$_{\text{Src}}$**	**Applicability**	**(Potential) Solution**	**Notes**
V_{SMP7}	LESC with *Passkey Entry* reveals passkey bit-by-bit during pairing [50, 51]. Older versions of specification do not mandate non-static passkeys. Implementations may use fixed passkeys.	S+D	BLE devices supporting LESC with *Passkey Entry* and specification <v5.1.	Use random passkeys (a new passkey for each new pairing attempt) or *Numeric Comparison*.	***CVEs:*** CVE-2018-5383
V_{SMP8}	Key entropy reduction allowed. No integrity protection in place for *maximum supported key size* during the feature exchange phase, nor is there a check made by implementing platforms regarding acceptable key sizes [52].	S+D	All BLE devices	Fix the key size to a secure value, implement integrity protection for it, and/or check value at higher layers [52].	***Tools:*** `Sweyntooth` [83] includes a script to test for this vulnerability.
V_{SMP9}	Acceptance of large key sizes (i.e., greater than the max allowed size of 16 bytes) during pairing feature exchange [58].	C	Various Telink chipsets	Code fix.	***V$_{CN}$:*** SweynTooth, ***CVEs:*** CVE-2019-19196 ***Tools:*** `Sweyntooth` [83]
V_{SMP10}	The LTK is set to zero if an out-of-order encryption request is received (see V_{LL5}) [58].	C	Various Telink chipsets	Code fix.	***V$_{CN}$:*** SweynTooth, ***CVEs:*** CVE-2019-19194 ***Tools:*** `Sweyntooth` [83]
V_{SMP11}	If a public key is transmitted during LE *Legacy* pairing, the Peripheral accepts the key and tries to copy it to a null address [58].	C	Texas Instruments SDK up to v3.30.00.20 for CC2640R2	Code fix.	***V$_{CN}$:*** SweynTooth, ***CVEs:*** CVE-2019-17520 ***Tools:*** `Sweyntooth` [83]

Table 4.6: Security and privacy vulnerabilities within ATT/GATT.

$\mathbf{V}_{\mathbf{ID}}$	**Vulnerability**	$\mathbf{V}_{\mathbf{Src}}$	**Applicability**	**(Potential) Solution**	**Notes**
V_{GATT1}	According to the Bluetooth specification, the list of services and associated characteristics are freely readable by any connected device, and cannot be protected by authentication [36].	S	All BLE devices.	Require authentication before allowing enumeration of services and characteristics [46]. (This could break backward compatibility with existing devices.)	
V_{GATT2}	In practice, many BLE devices apply minimal or no protection to their characteristics [46, 87–90]. We show this in Chapters 9 and 10.	D	Device specific.	Suitably strong permissions should be applied to characteristics.	***Tools:*** `ATT-Profiler`⋆ [32] identifies minimum security required for accessing each BLE characteristic on a Peripheral. `argXtract`⋆ [33] and `FirmXray` [91] can identify configured characteristic protection levels via firmware analysis. `BALSA`° [17] is an app-layer security add-on (see §4.4.2).
V_{GATT3}	Sequential ATT requests arriving too close to one another are improperly handled [58].	C	STMicroelectronics BLE Stack up to 1.3.1 for STM32WB5x.	Code fix.	$\boldsymbol{V_{CN}}$**:** SweynTooth ***CVEs:*** CVE-2019-19192 [58] ***Tools:*** `Sweyntooth` [83]

Table 4.7: Security and privacy vulnerabilities within GAP.

$\mathbf{V_{ID}}$	**Vulnerability**	$\mathbf{V_{Src}}$	**Applicability**	**(Potential) Solution**	**Notes**
V_{GAP1}	In practice, many BLE devices do not implement private addresses [9, 23, 68, 92], and those that do, may not be doing so in a secure manner [10]. We demonstrate the lack of private address usage in Chapter 10.	D	Device specific.	Use private addresses as outlined in the Bluetooth specification. Change the addresses periodically.	***Tools:*** `argXtract`⋆ [33] and `FirmXray` [91] can identify the presence (or absence) of private addresses. `Valkyrie` [93] verifies correctness of address randomisation implementations [94]. `BLE-Guardian`° [10] is a management tool to enable privacy in existing BLE deployments (see §4.4.1).
V_{GAP2}	A GAP advertisement contains the Peripheral's advertising address, and also usually information such as Service UUIDs, manufacturer data, etc., some of which may be very specific to a certain manufacturer or type of device [46].	D	Device specific.	BLE Peripherals could be configured to limit the information they broadcast in advertisements. However, this could make a Peripheral less easy to identify by a legitimate Central.	***Tools:*** `Valkyrie` [93] tests for identifiers within advertisements. `Venom`° implements a tracking system using such identifiers [95]. `BLEScope`° extracts BLE UUIDs from Android APKs, to test against UUIDs in advertisements [66].
V_{GAP3}	When an LL encryption error is sent to the Host, the BLE specification states that pairing should be restarted after user confirmation [54]. However, many platforms do not do so, but instead allow communications to continue in plaintext [54, 55].	D	Android, iOS and similar operating systems.	On receiving an LL encryption error from its peer, the device should notify the Host, which in turn should re-bond after confirming with the user.	$\boldsymbol{V_{CN}}$***:*** BLESA [54]

$\mathbf{V_{ID}}$	**Vulnerability**	$\mathbf{V_{Src}}$	**Applicability**	**(Potential) Solution**	**Notes**
V_{GAP4}	BLE Peripherals have limited storage for bond information, causing them to reject new bonds or replace existing ones [53].	D	Unknown (likely all BLE Peripherals).	Authenticated pairing or LL whitelisting (with resolvable private addresses) could reduce attack potential.	
V_{GAP5}	The visibility of BLE Peripherals is dependant on advertising frequency [19].	U	Unknown (likely all BLE Peripherals).	Filtering advertisements using thresholds for advertising interval, combined with received signal strength information, etc. could reduce the impact [96,97].	***Tools:*** `GATTacker` [98] performs MitM attacks by exploiting Peripheral visibility due to advertising frequency. `MARC`° is a MitM detection framework for eHealth devices [96] and `BlueShield [99]` is a generic spoofing detection framework; both consider advertising frequency as a parameter (see §4.4.3).

Table 4.8: Security and privacy vulnerabilities within the Application Layer.

$\mathbf{V_{ID}}$	**Vulnerability**	$\mathbf{V_{Src}}$	**Applicability**	**(Potential) Solution**	**Notes**
V_{APP1}	No fully-defined mechanism within the BLE specification to restrict data access at application layer, particularly with SIG-defined Profiles (see Chapters 5, 6). Implementing platforms also do not sufficiently restrict application access to BLE data.	S+D	Android, iOS and similar multi-application platforms.	Specification change, platform-level change or application-layer security. We propose a specification-level change in Chapter 7.	***Tools:*** `BLECryptracer`* [29] determines whether app-layer security is present for BLE data by performing taint analysis against companion Android APKs. `BLESS`° [100] is also a taint analysis tool which tests for app-layer security in APKs; it includes tests for cryptographic correctness.

Table 4.9: Vulnerability-to-attack mapping.

Layer	**Vuln.**	**Src**	**Attacks (Attack Mechanisms)**	**C***	**I***	**A***	**P***	**D***
PHY	V_{PHY1}	D	Activity Detection (P.2)				●	
			User Detection (P.2)				●	
LL	V_{LL1}	D	Battery Exhaustion (A.1.1)			●		
	V_{LL2}	C	Heap Overflow (A.1.2)			●		
			Shellcode Execution (I.2.2)		●			
	V_{LL3}	C	Buffer Overflow→Crash (A.1.2)			●		
	V_{LL4}	C	Faulty State (Power Cycle Required) (A.1.2)			●		
	V_{LL5}	C	Combined with V_{SMP9} and V_{SMP10}	◖	◖	◖		
	V_{LL6}	C	Heap Corruption (A.1.2)			●		
HCI	V_{HCI1}	C	Local Privilege Escalation (I.2.2)		●			
L2CAP	V_{L2CAP1}	C	Remote Device Restart (A.1.2)			●		
SMP	V_{SMP1}	S	Obtaining LTK (C.1.2.1, I.1, A.3, D)	●	●	●		●
	V_{SMP2}	S	Authentication Bypass (C.1.3.1, I.1, D)	●	●			●
	V_{SMP3}	S	MitM Attack (C.1.3.2, I.1, A.3, D)	●	●	●		●
	V_{SMP4}	S+D	MitM Attack (C.1.3.2, I.1, A.3, D)	●	●	●		●
	V_{SMP5}	S+D	Obtain Communication Profile (P.3.2)				●	
	V_{SMP6}	S+D	Secret Key Derivation (C.1.2.1, I.1, A.3, D)	❍	❍	❍		❍
	V_{SMP7}	S+D	Static Passkey Leak (C.1.2.1, I.1, A.3, D)	●	●	●		●
	V_{SMP8}	S+D	Low Entropy Key Negotiation (C.1.2.1, I.1, A.3, D)	❍	❍	❍		❍
	V_{SMP9}	C	Overflow→Crash (A.1.2)			◖		
	V_{SMP10}	C	Security Bypass (C.1.3.3, I.1)	◖	◖			
	V_{SMP11}	C	Hard Fault (A.1.2)			●		
GATT	V_{GATT1}	S	Device Fingerprinting (P.1.2)				●	
	V_{GATT2}	D	Unauthorised Data Access (C.1.1.1, I.1)	●	●			
			Spoofing and MitM (C.1.1.1, I.1, A.3, D)	●	●	●		●
	V_{GATT3}	C	Deadlock (A.1.2)			●		
GAP	V_{GAP1}	D	User Tracking (P.1.1.1)				●	
			Selective Jamming (A.2.1.1)			●		
	V_{GAP2}	D	Device Fingerprinting (P.1.2)				●	
			Beacon Spoofing (D)					●
			User Profiling (P.3.1)				●	
	V_{GAP3}	D	Data Access (C.1.3.1, I.1)	●	●			
			IRK Theft (P.1.1.2)				●	
			Prevent Legitimate Communication (A.2.2.1)			●		
	V_{GAP4}	D	LTK Theft (C.1.2.2, I.1, A.3, D)	●	●	●		●
			Prevent New Bonds (A.2.2.2)			●		
	V_{GAP5}	U	Hide Peripheral Advertisements (A.2.1.3)			●		
			Spoofing (D)					●
App	V_{APP1}	S+D	Unauthorised Data Access (C.1.1.2, I.1)	●	●			

$*$(Attacks on) C-Confidentiality; I-Integrity; A-Availability; P-Privacy; D-Device authentication.
● - Possibility of attack; ◖ - Requires exploitation of another vulnerability; ❍ - Contribution to attack (requires further effort, such as significant brute-forcing), or only *probability* of attack.

Table 4.10: Number of vulnerabilities per-layer, per-source.

Source	**PHY**	**LL**	**HCI**	**L2CAP**	**SMP**	**GATT**	**GAP**	**App**
Specification	0	0	0	0	8	1	0	1
Design	1	1	0	0	5	1	4	1
Coding	0	5	1	1	3	1	0	0
Unknown	0	0	0	0	0	0	1	0

Fuzzing some of these inner layers can be achieved by crafting custom BLE packets using Scapy or similar [58], but this requires knowledge of raw packet formats. We foresee that recent developments such as *InternalBlue* [101], which allows for instrumenting Bluetooth stacks, but which is currently focused on Bluetooth Classic, will help researchers bridge this gap.

4.3 Analysis of Vulnerabilities by Source

Table 4.9 shows that more than 50% of identified vulnerabilities are actually implementation issues, and around half of these are coding bugs. This demonstrates that even if a technology is designed to be completely secure, there is still a high probability of vulnerabilities being introduced during implementation. We present an analysis of vulnerabilities by source below.

Bluetooth specification Table 4.10 shows that most vulnerabilities that are attributable to the Bluetooth specification are those that are concerned with the Security Manager protocol. This is perhaps not surprising, as this is the layer where most security protocols and algorithms are formally defined. Most Security Manager vulnerabilities are to do with the use of proprietary algorithms that may not have undergone security testing and formal verification. However, as we noted in C.1.2.1, the most well-known LE Legacy vulnerability, V_{SMP1}, was known when it was included in the specification. Another vulnerability that was included in the specification despite obvious security connotations was V_{SMP8}, which allows for key entropy reduction. It is possible that this feature was included in case resource-constrained devices were unable to handle larger keys. However, we note that the other keys (CSRK, IRK) are 128 bits long with no option for entropy reduction. Other specification-related Security Manager vulnerabilities appear to be unintended consequences of security features or edge cases. These clearly indicate the need for formal security verification and/or testing of all security protocols.

Product design Most of the vulnerabilities that have been denoted as Design issues are related to non-implementation of available security or privacy features. It is possible that a lack of time, money, technical/security understanding or incentive (or a combination thereof) is the cause for such vulnerabilities. End product vendors tend to focus on time-to-market and getting ahead of their competition. Up to now, there has been limited push from either consumers or lawmakers to focus on security. As long as consumers want products with many features and low costs, and as long as there are no laws surrounding proper protection of data handled by such products, it is likely that the situation will continue.[2]

[2]In 2020, the European Telecommunications Standards Institute (ETSI) adopted a new standard, EN 303 645 V2.1.1 [102], for IoT product security. The government of the United Kingdom is currently considering legislation to enforce some of the standards [103].

However, some design issues, such as V_{PHY1}, are likely to have been unforeseen. With some other vulnerabilities, such as V_{GAP2} or those that are denoted Specification+Design, it is possible that end product vendors implemented the specification in good faith, without realising the potential for attack. This demonstrates the need for threat modelling for end products.

Coding bugs Most vulnerabilities that are denoted as Coding Bugs are to do with poor input validation, often leading to overflows. This type of vulnerability has been present in computer programs for several decades now, and is still featuring in new products despite being one of the best-known vulnerabilities. With BLE, the fact that processing external inputs (from a BLE peer) is necessary for standard system operation dictates that every field within a received packet should be carefully checked by the receiver for length, format and value constraints.

4.4 A Brief Survey of Proposals for Security/Privacy Enhancement

Along with studies that have identified vulnerabilities and attacks against BLE devices, there have also been a number of works proposing enhancements, either to the specification itself or to existing deployments, in order to improve the security or privacy stance of BLE.

4.4.1 Privacy in the Absence of Private Addresses

Real-world device testing has shown that many BLE Peripherals do not employ private addresses (V_{GAP1}), which leaves the devices vulnerable to tracking. Fawaz et al. [10] presented a Central management tool named `BLE-Guardian`, which hides the presence of BLE devices, except from authorised devices, by reactive jamming of wireless frequencies. Further, it also offers access control capabilities, such that only specific devices can issue connection requests to a BLE device. Interestingly, in this scenario, a privacy concern due to choices related to the GAP layer (i.e., advertising) is solved partially using PHY/LL means (with selective jamming).

4.4.2 Application-Layer Security Add-on

Ortiz-Yepes [17] proposed a framework termed *BALSA*: ***B**luetooth Low Energy **A**pplication **L**ayer **S**ecurity **A**dd-on*, to achieve higher-layer security in a standardised manner. The author cites weak LE Legacy pairing mechanisms (V_{SMP1}) as the reason for development of the system. However, as we show in Chapter 5, the strongest pairing mechanisms may still leave data vulnerable and necessitate higher-layer protection. BALSA is intended to work on top of existing BLE Peripherals (termed *Sensors*) and existing mobile devices without modifications to the platforms. However, it requires the use of a *Backend*, with which the BLE Peripheral must share symmetric keys, and a Kerberos-like protocol, which makes the solution unsuitable for most consumer usage scenarios.

4.4.3 Identification of Spoofed BLE Devices

It has been observed that a BLE Peripheral may be more visible to a Central if it advertises rapidly, which gives rise to the potential for spoofing attacks (V_{GAP5}). Yaseen et al. [96] proposed a MitM detection framework termed *MARC*, for the specific use case of BLE eHealth devices that have no input-output capabilities. The framework operates on four LL metrics: the Received Signal Strength Indicator (RSSI), advertising interval, and the Central and Peripheral MAC addresses. The framework establishes threshold values for the RSSI and advertising interval, and a received packet that falls short of either threshold is considered to be from a possible clone. In addition, MAC address whitelisting is used (with resolvable private addresses, if supported) to further determine whether a device is legitimate or cloned. Note that, in the absence of resolvable private addresses, whitelisting will only work when an attack device cannot spoof MAC addresses.

Wu et al. [97] presented a similar spoofing detection framework, implemented as *BlueShield*, for stationary BLE devices in indoor environments. It monitors a larger number of parameters, including advertising pattern and operation state, and implements two phases: a profiling phase, during which normal operational characteristics of a BLE device are determined, and a monitoring phase, during which spoofing attacks are detected based on deviations from normal parameters.

4.4.4 Cryptography Enhancements

A conceptual study by Perrey et al. [12] suggested the use of Merkle's Puzzles (MP) as a key distribution mechanism in the absence of ECDH, possibly as a new association model. The proposed design involves the Central device broadcasting a very large number of puzzles, each containing a puzzle identifier and a strong key, and each encrypted using a weak key. The Peripheral selects a puzzle at random and solves it to obtain the puzzle identifier and strong key. It unicasts its puzzle identifier back to the Central after the Central has broadcast all puzzles. An attacker, however, would need to solve far more puzzles (on average, $\frac{n}{2}$ puzzles) in order to come up with the same key (as the attacker has no way of knowing in advance which puzzle will be solved by the Peripheral). The authors suggest a connection-less approach, using BLE advertisements to broadcast the puzzles, with the Central device advertising and the Peripheral performing the scanning. We note that this conflicts with the design of BLE. They also propose the addition of RC5, to be used for encryption in this mode instead of AES-128. We observe that the key exchanged in this manner can only be used to encrypt data that is useful for a short amount of time, as the attacker may be able to obtain the key eventually.

4.5 Chapter Summary and Next Steps

In this chapter, we analysed vulnerabilities within the BLE specification and implementations, identified their root cause and outlined proposed solutions. We also mentioned tools and frameworks for testing, exploiting and mitigating the vulnerabilities.

This chapter concludes Part I. The next part of this thesis relates to an application-layer unauthorised data access vulnerability that we have identified for BLE (which we have already mentioned briefly in the previous chapter, under attack category C.1.1.2, and in this chapter, as V_{APP1}). Part II describes the vulnerability, analyses possible mitigation options and proposes a backward-compatible solution.

Part II
BLE Application Layer Security

5 Unauthorised Data Access on Multi-Application BLE Platforms

In this chapter, we describe an application-level unauthorised data access vulnerability that we have identified for multi-application BLE platforms. While a similar result was published for Bluetooth Classic three years prior to our work, our results were obtained independently and demonstrate additional attack techniques. Our tests were conducted on Android, and the discussion on permissions is directly applicable to Android. However, the overall results regarding unauthorised data access are applicable, with platform-specific restrictions, to other multi-application platforms as well.

5.1 Introduction

As described in Chapter 2 and depicted in Figure 2.1, a typical consumer usage scenario for BLE involves the user interfacing with one or more BLE Peripherals via applications on a multi-application platform, typically a mobile phone or tablet. In most cases, a BLE Peripheral manufacturer will provide their own specific mobile application for interacting with their Peripheral device(s). These applications will be downloaded by the user from an application marketplace and will reside on the multi-application platform among other downloaded apps. These apps normally originate from a number of different developers and not all will necessarily be trustworthy. The BLE Peripherals, for their part, may access user health information (as with glucose monitors) or perform security-critical functions (as with smart door locks). It is therefore imperative that an application should not be able to access sensitive information from a BLE Peripheral without the user's knowledge.

In many cases, the BLE Peripherals will specify authentication permissions (see §2.2.4) as the mechanism for restricting access to BLE data; in fact, authentication permissions are specified as the *only* security requirements within the vast majority of official SIG-defined BLE services and profiles that have any security requirement at all. For example, the SIG-specified Glucose Service only requires authentication for a single characteristic and the Glucose Profile, as depicted in Figure 5.1, mandates only Mode 1 Level 2 or Mode 1 Level 3 protection for the characteristics within the Glucose Service on the Glucose Sensor device (i.e., a glucometer). No other security requirements are specified. In particular, *authorisation* permissions are specified for no SIG-defined service specification except the Insulin Delivery Service.

If a device (such as a mobile phone) wishes to read/write data on a glucometer that conforms to the SIG-defined Glucose Profile, it must undergo either unauthenticated or authenticated pairing (depending on whether the glucometer specifies Mode 1 Level 2 or Mode 1 Level 3) with the

Glucose Sensor Security Considerations

- *All supported characteristics specified by the Glucose Service shall be set to Security Mode 1 and either Security Level 2 or 3.*
- *The Glucose Sensor shall bond with the Collector.*
- *The Glucose Sensor shall use the SM Slave Security Request procedure to inform the Collector of its security requirements.*
- *All characteristics specified by the Device Information Service that are relevant to this profile should be set to the same security mode and level as the characteristics in the Glucose Service.*

Collector Security Considerations

- *The Collector shall bond with the Glucose Sensor.*
- *The Collector shall accept any request by the Glucose Sensor for LE Security Mode 1 and either Security Level 2 or 3.*

Figure 5.1: Security requirements within Glucose Profile.

glucometer first. The pairing is typically triggered by the official glucose monitoring application on the mobile phone, either programmatically or when characteristic access is attempted.

In this chapter, we describe a vulnerability that we have identified for multi-application platforms, which could enable a malicious application to covertly access data from a BLE Peripheral, even if a different application has previously triggered pairing with the Peripheral. We use the Android platform to demonstrate this attack, using purpose-built BLE Peripheral devices and BLE-enabled Android applications (§5.2). In §5.3, we discuss the implications of the attack and its applicability to platforms other than Android. We also compare our results with results obtained by Naveed et al. [48] for Bluetooth Classic on the Android platform. Finally, we present a high-level analysis of current mitigation options.

Related work Several studies [15,16,52,56,57] have (sometimes implicitly) explored the potential for unauthorised access of BLE data due to vulnerabilities at the SMP/link layer. However, *application*-layer security and privacy concerns have not been widely explored for BLE. Korolova et al. [64] describe an application-level privacy vulnerability in which smartphone application developers can derive fairly unique fingerprints for their users based on the BLE devices in the vicinity. The work that is most closely related to ours is research on Bluetooth *Classic* by Naveed et al. [48], which explores the implications of shared communication channels on Android devices. In their paper, the authors discuss the issue of Bluetooth Classic channels being shared by multiple applications on the same device, and demonstrate unauthorised data access attacks against (Classic) Bluetooth-enabled medical devices. We independently identified this vulnerability for BLE, and discovered greater attack potential in the BLE case.

5.2 Attack Demonstration

In this section, we demonstrate an application-level unauthorised data access vulnerability in BLE, using Android as the test platform. We show how *any* application on an Android device (subject to permissions) can access pairing-protected attributes from a BLE Peripheral, even when the pairing process was initiated by a different application.

We describe two attacks: the first shows that pairing-protected data can be accessed by unauthorised applications, while the second refines the attack and reduces the number of permissions required by the unauthorised application. While these attacks were tailored towards Android's permissions model, the overarching vulnerability is applicable to other multi-application platforms, as we discuss in §5.3.5.

We mimic a normal usage scenario in our tests. We emulate a BLE Peripheral (let us say a glucometer, "GlucoMeter") using the Nordic nRF51 Development Kit, and implement a characteristic that specifies a security requirement of Mode 1 Level 3.[1] We use two purpose-built Android applications to describe the attacks:

1. One app serves as the glucose monitoring app issued by the Peripheral developer ("OfficialApp"). It is expected to be able to connect to the BLE device and access its data.
2. The second application is a malicious app masquerading as a game ("EvilGameApp"). It should *not* be able to access pairing-protected data from the GlucoMeter.

We conducted our experiments on an Alcatel Pixi 4 mobile phone, running Android 6.0, and on a Google Pixel XL, running Android 8.1.0. Version 6.0 was the most widely-deployed release and v8.1.0 was the latest stable release, when these experiments were conducted.[2]

Attack 1: Exploiting system-wide pairing credentials This attack demonstrates that the BLE pairing credentials that are stored on an Android device are utilised by the OS for *all* applications on the device, rather than just the application that originally triggered the pairing.

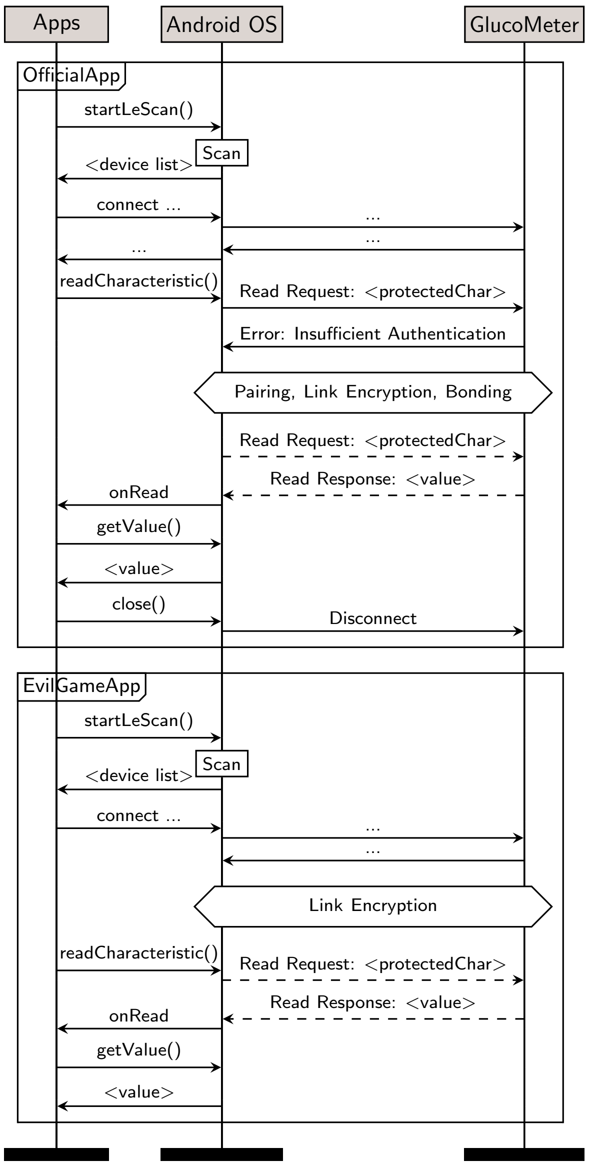
We perform the following steps in order:

- Launch OfficialApp and scan for BLE devices.
- Connect to the GATT server on the "GlucoMeter" and read a characteristic. This will trigger pairing and read a dummy value of `0x12345678`.
- Disconnect from GlucoMeter and close OfficialApp.
- Launch EvilGameApp. This covertly scans for and connects to the GlucoMeter, and reads the same characteristic.

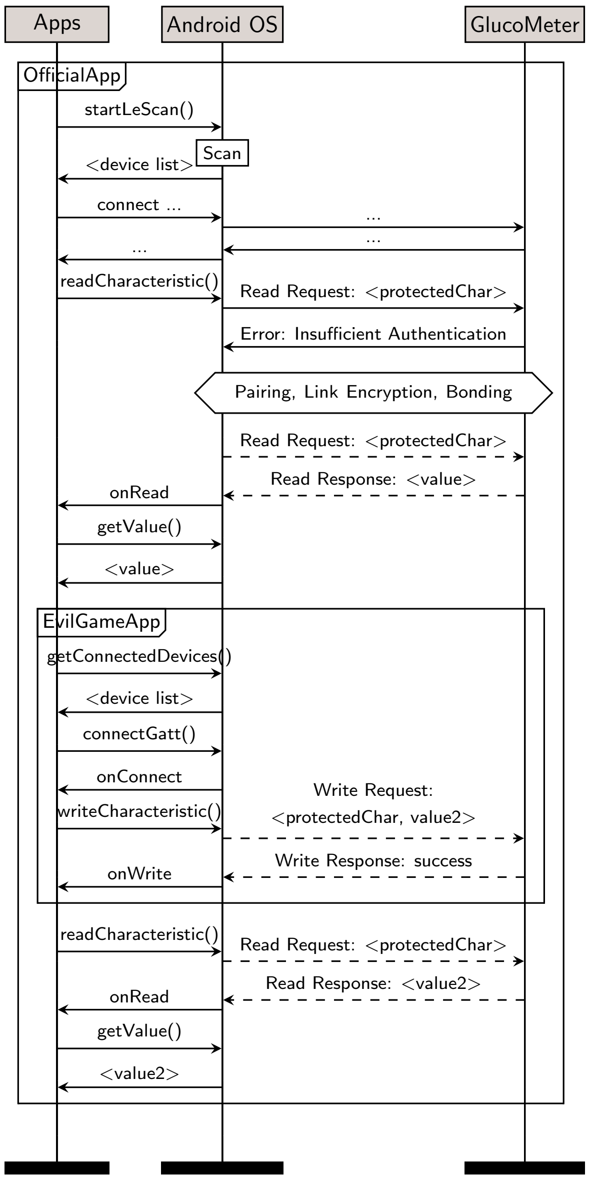
When the OfficialApp connects to the GlucoMeter and attempts to access a pairing-protected characteristic, the resulting exchange will trigger the Android OS into initiating the pairing and bonding process (as depicted in the upper block in Figure 5.2a). The resultant keys are associated with the link between the GlucoMeter and the Android device, rather than between the GlucoMeter and the OfficialApp (which actually triggered the pairing). Therefore, once bonding completes, if the *EvilGameApp* later scans for and connects to the GlucoMeter, the Android OS will complete the connection process and automatically initiate link encryption with the keys that were generated during the previous bonding process (lower block in Figure 5.2a). This

[1] We also repeat the tests with Mode 1 Level 4 protection, i.e., LESC pairing. The same results were observed.

[2] Note that the Android architecture (with respect to Bluetooth) changed with v8.0 [104], but the vulnerability is present even in the newer version, i.e., it is version-independent.



(a) Attack 1 - Unauthorised access of pairing-protected data by creating new connection.



(b) Attack 2 - Unauthorised access of pairing-protected data by reusing an existing connection.

Figure 5.2: Illustrative message exchanges depicting application-level unauthorised data access on multi-application platforms. *Note:* Dashed lines indicate encrypted traffic.

enables the EvilGameApp to have the same level of access to the pairing-protected GlucoMeter data as the OfficialApp.

A key point to note here is that, not only is the EvilGameApp able to access potentially sensitive information from the BLE device, but also the user is likely to be unaware of the fact that this data access is taking place, as there is no indication during link re-encryption and subsequent attribute access.

Attack 2: Reusing existing connection Our second attack exploits the fact that, on Android, a BLE Peripheral can be accessed concurrently by multiple applications [105]. In this attack, the EvilGameApp does not scan for BLE devices. It instead searches for connected BLE devices using the `BluetoothManager.getConnectedDevices()` API call, with `BluetoothProfile.GATT` as the argument. If the OfficialApp happens to be in communication with the GlucoMeter at the same time, this call will return a list with a reference to the connected GlucoMeter. The EvilGameApp is then able to directly connect to the GATT server on the GlucoMeter and read and write to the (readable/writable) characteristics on it, including those that are pairing-protected, without the need for creating a new connection to the GlucoMeter. This again is done surreptitiously, without the user being aware of the data access. An illustrative message flow where the EvilGameApp writes to a protected characteristic on the GlucoMeter (which the OfficialApp subsequently reads) has been depicted in Figure 5.2b.

An interesting observation from this attack is a subtle but relevant impact it has on user awareness, due to the different Android permissions that need to be requested by the two applications. Since both applications access data from a GATT server, they both require `BLUETOOTH` permissions. In this attack scenario, because the OfficialApp scans for the BLE device before connecting to it, it also needs to request the `BLUETOOTH_ADMIN` permission. Both `BLUETOOTH` and `BLUETOOTH_ADMIN` are "normal" permissions that are granted automatically by the Android operating system after installation, without any need for user interaction. However, due to restrictions imposed from Android version 6.0 onward, the OfficialApp also needs to request `LOCATION` permissions to invoke the BLE scanner without a filter (i.e., to scan for all nearby devices instead of a particular device). These permissions are classed as "dangerous" and will prompt the system to display a confirmation dialog box the first time they are required. Because the EvilGameApp merely has to query the Android OS for a list of already connected devices, it does not require these additional permissions. This makes the EvilGameApp appear to be less invasive in the eyes of a user, since it does not request any permission that involves user privacy. This could play a part in determining the volume of downloads for a malicious application. For example, a malicious application that masquerades as a gaming application, and which does not request any dangerous permissions, may be more likely to be downloaded by end users than one that requests location permissions.

Attack limitations The main limitation for the EvilGameApp in the case of the first attack is that it requires the `BLUETOOTH` and `BLUETOOTH_ADMIN` permissions in its manifest, and needs to explicitly request `LOCATION` permissions at first runtime in order to be able to invoke the BLE scanner. This enables the EvilGameApp to connect to the BLE device regardless of whether or not another application is also connected, but increases the risk of raising a user's suspicions.

In the second attack scenario, the obvious limitation for the EvilGameApp that requests only the `BLUETOOTH` permission is that the application will only be able to access data from the BLE Peripheral when the Peripheral is already in a connection with (another application on) the Android device. That is, data access will have to be opportunistic. This can be achieved, for example, by periodically polling for a list of connected devices.

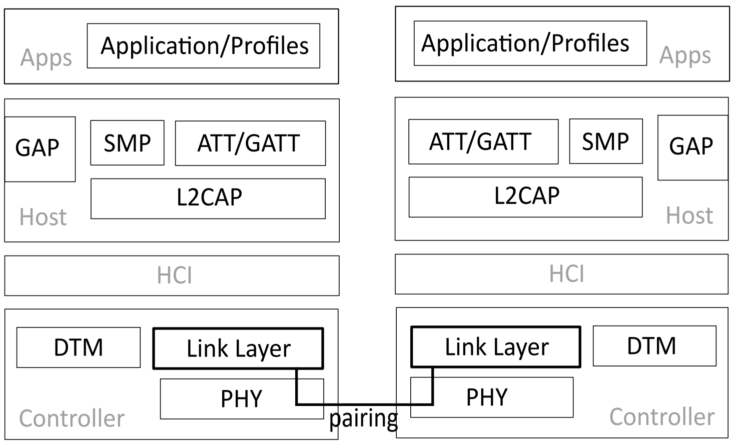


Figure 5.3: Scope of BLE pairing.

5.3 Discussion

In this section we discuss responsible disclosure, examine factors that contribute to the attacks that we have described in §5.2, and outline the impact of our findings. We further compare these findings with the Bluetooth Classic case, discuss applicability to other platforms, and present current options for mitigation.

5.3.1 Responsible Disclosure

We reported our findings to the Android Security Team, focusing on the need of clear documentation so that developers are aware of the need for implementing additional protection measures if they are handling sensitive BLE data. The issue was assigned a severity level of Moderate and the Security Team informed us that remedial action was being considered. Android has since updated its developer guidelines to state: "*When a user pairs their device with another device using BLE, the data that's communicated between the two devices is accessible to all applications on the user's device. For this reason, if your application captures sensitive data, you should implement app-layer security to protect the privacy of that data.*" [106].

5.3.2 Contributing Factors

Scope of pairing and absence of explicit higher-layer protection mechanisms As per the BLE specification, the scope of BLE pairing only extends up to the Link Layer, as depicted in Figure 5.3. That is, it does not apply to the application layer. Because of this, the pairing credentials are implicitly used for all applications that request access to the data on a specific BLE Peripheral. This is in any case justifiable from a user experience perspective, as a user may not desire going through the entire pairing process every time they choose to install a new application.

The BLE specification does mention authorisation permissions, which are an access restriction mechanism applied at the application layer. However, it does not define expected behaviour or protocols to handle such permissions, instead leaving the implementation to the developer. It does not fully define any other means for restricting access at the application layer, despite being considered a full-stack protocol (including an Application Layer). Platforms such as Android also do not implement any restrictions beyond the requirement for certain user permissions (such as the `BLUETOOTH` permission).

Blanket application of Android `BLUETOOTH` permissions The attacks we have described in §5.2 bring to light a concern with regard to how Android permissions are applied for external device accesses. The Android `BLUETOOTH` (and `BLUETOOTH_ADMIN`) permissions, which are required for Bluetooth operations, are applied on a per-application basis, but not on a per-Peripheral basis. This means that if an application is downloaded to be used with one particular Bluetooth Classic or BLE device (which means it will be granted the required permissions by the user), that application is thereafter in a position to covertly access data from other BLE (or Bluetooth Classic) devices. For example, if an application is installed for the purpose of controlling a BLE light bulb, that application could then access data from any other BLE devices, such as fitness trackers or glucose monitors, in the proximity of the user (as long as the devices don't implement application-layer protection).

Connection reuse among applications It is possible that concurrent connections from different applications are allowed by design to enable multiple applications to communicate with a BLE Peripheral without competition. Unfortunately, the possibility of reusing an existing connection to access BLE data (as described in Attack 2) is one that gives rise to greater attack potential for a malicious application, since the application will need to request fewer permissions and may therefore appear to be innocuous.

5.3.3 Implications of Attack

In both of our experiments, the EvilGameApp was able to read and write pairing-protected data from the BLE device. The simplest form of attack would then be for a malicious application to perform unauthorised reads of personal user data and relay this to a remote server. For example, a malicious application could target sensitive health information such as ECG, glucose or blood pressure measurements from vulnerable BLE devices, to build up a profile on a user's health. Further, Smart Home devices and BLE-enabled vehicles may hold information on a user's habits and lifestyle (e.g., time at home, alcohol consumption, driving speed), and could be exploited.

If the BLE Peripheral has writable characteristics, then a malicious application could overwrite values on the Peripheral, such that the written data is read back by the legitimate application, thereby giving the user an incorrect view of the data on the Peripheral. It may also be possible to control device functionality in such a way as to cause unexpected behaviour or even endanger lives (as discussed in §3.5). Further, it may be possible to install malicious Peripheral firmware via GATT writes. We demonstrate the possibility of this against a real-world device in §6.7.

We also found that, in some scenarios, the malicious app is able to circumvent certain (non-cryptographic) protections that have been put into place at the application layer. This was found to be the case for the Mi Band 2 fitness tracker. This device implements the Bluetooth Heart Rate Service and, according to the service specification, characteristics within this service are only supposed to be protected by pairing [107]. However, we observed that access to the Heart Rate Measurement characteristic was artificially "locked" and had to be "unlocked" by first writing to certain other characteristics on the tracker. Despite this, we found that by deploying our second attack, our EvilGameApp was able to obtain Heart Rate Measurement readings without the need for performing any "unlocking". This is because the EvilGameApp connects to the GATT server by reusing an existing connection that was initiated by the official Mi Band 2 application. The unlocking procedure would therefore already have been performed for that connection by the official application. This result shows that artificially restricting access to data using non-cryptographic means will not be effective. We notified the device developer of this issue, but have not received a response.

5.3.4 Comparison with Bluetooth Classic

Naveed et al. [48] analysed unauthorised data access for the Bluetooth Classic case and demonstrated that it was possible for a malicious app to covertly access data from a Bluetooth Classic device that had previously been paired. However, their experiments found that an unauthorised Android application would *not* be able to obtain data from a Bluetooth Classic device if the authorised application had already established a socket connection with the device, as only one application can be in communication with the device at one time. Therefore, a malicious application would either require some side-channel information in order to determine the correct moment for data access, or would need to interfere with the existing connection, thereby potentially alerting the user. This limits the attack window for the malicious application. Our experiments show that this is not the case with BLE communication channels. With BLE, there are no socket connections and if the official application has established a connection with the BLE Peripheral, then this connection can be utilised by any application that is running on the Android device. That is, a malicious application does not have to wait for the authorised application to disconnect before it can access data.

5.3.5 Applicability to Other Platforms

While our tests focused mainly on Android, preliminary tests on iOS showed that similar data access was possible, but with slightly more restrictions. For example, in order to scan for BLE Peripherals when an iOS application is running in the background, it must provide a list of services to scan for (i.e., a list of services that must be present on the Peripheral). However, scanning while in the foreground allows for specifying a null list of services [108]. Connecting to an already-connected Peripheral is possible using `connectPeripheral:options:`, although retrieving a list of Peripherals requires knowledge of either a Peripheral identifier (assigned by iOS) or of one or more services on the Peripheral. In addition, from iOS13 onward, an application that requires Bluetooth must obtain explicit user permission [109]. However, as with Android, this permission is on a per-application basis, not on a per-Peripheral basis, which means that once an application is allowed access to one BLE device, it will be able to access other BLE devices without further user confirmation.

5.3.6 Mitigation Strategies

Allowing all applications on a multi-application platform to share BLE communication channels may work in some situations, for example on a platform where all applications originate from the same trusted source. However, most modern multi-app platforms host applications from various, potentially untrusted sources. In this scenario, providing all applications with access to a common BLE transport opens up possibilities for attack, as we have demonstrated.

Ideally, a fully-defined solution should be available within the BLE specification itself (we propose such a solution in Chapter 7). Otherwise, the various multi-application platforms should incorporate some form of policies or restrictions to prevent unauthorised applications from accessing data on BLE peer devices *on a per-peer basis*. At present, however, such mechanisms are not available and the responsibility of securing BLE data lies in the hands of BLE application/device developers. That is, rather than relying solely on the pairing provided by the underlying operating system, developers can implement end-to-end security from the BLE Peripheral firmware to the companion application. This can be achieved via BLE authorisation permissions. Even though authorisation permissions are, strictly speaking, intended to specify a requirement for end user authorisation, the behaviour of BLE devices when encountering authorisation requirements is implementation-specific. Most modern BLE chipsets implement authorisation capabilities by intercepting read/write requests to the protected characteristics, and allowing for developer-specified validation. One advantage of this method is that it gives the developer complete control over the strength of protection that is applied to BLE device data. This may also be a disadvantage in some cases, if protection is not applied or is insufficient. Another disadvantage is reduced flexibility for the user, in terms of choice of applications.

5.4 Chapter Summary and Next Steps

We have demonstrated an application-level unauthorised data access vulnerability for multi-application platforms, where a malicious application is able to access pairing-protected BLE data in a covert manner. Our tests were conducted on the Android platform and we have responsibly disclosed the attack to the Android Security Team. We have discussed attack implications and applicability to other platforms, and provided a comparison with the Bluetooth Classic case.

At present, the only option for mitigating the vulnerability is the implementation of end-to-end security by developers. However, due to the lack of clear guidelines (at the time of conducting our experiments), it is also possible that developers implement no security at all, due to an assumption that protection will be handled by pairing. In the next chapter, we test this assertion of a lack of developer awareness by exploring the state of application-layer security deployments via a large-scale analysis of BLE-enabled Android applications.

6 Measuring the Prevalence of Application Layer Security

In this chapter, we measure application-layer security implementations within the BLE ecosystem using companion mobile applications. We describe a purpose-built tool, `BLECryptracer`, which performs taint analysis of Android APKs to identify evidence of cryptographically-protected BLE data. We discuss the results obtained from applying `BLECryptracer` to a dataset of 18,900+ APKs.

6.1 Introduction

As evidenced by our experiments (§5.2), it is fairly straightforward for any application on a multi-application platform to connect to a BLE device and read or write pairing-protected data. As discussed in §5.3.6, the only strategy available at present is for developers to implement application-layer security, typically in the form of cryptographic protection, between the application and the BLE Peripheral. However, developers may not implement such protections due to lack of awareness or for other reasons. It is therefore pertinent to examine real-world BLE systems to identify those that do *not* implement application-layer security for their data, to gauge the number of devices that are potentially vulnerable to unauthorised data access.

There are different sources that can be exploited to obtain this information. In Chapter 10, we identify the presence of application-layer security via firmware analysis. In this chapter, we measure the prevalence of higher-layer protections using BLE-enabled mobile applications. That is, we test for the presence of cryptographically-protected BLE data on Peripheral devices by analysing companion mobile applications.

Most BLE Peripherals that interface with mobile applications support more than one mobile platform, typically Android and iOS at the very minimum. If the Peripheral implements end-to-end security that needs to be handled by the companion application, *all* such applications will have evidence of the higher-layer protection mechanisms. We target Android applications for our analysis, due to the availability of large APK datasets and the availability of tools that allow for APK decompilation and analysis. While taint-analysis tools such as `Flowdroid` and `Amandroid` exist for analysing APKs, we found through initial experiments that such tools were too computationally expensive for bulk analysis. For this reason, we develop a custom tool, `BLECryptracer`, for the specific purpose of analysing BLE data access methods within APKs.

We obtain a substantial dataset of BLE-enabled Android APKs (§6.2), determine BLE method calls and cryptography libraries of interest (§6.3), and define a taint-analysis mechanism to determine whether BLE reads and writes make use of cryptographically-processed data (§6.4). We evaluate (§6.5) and then apply this mechanism to our dataset, and analyse the results (§6.6).

Related work In their work with Bluetooth Classic, Naveed et al. [48] performed an analysis of 68 Bluetooth-enabled applications that handled private user data. They used `grep` to identify the locations of potential authentication secrets and, excluding 48 apps where such secrets were present in libraries, manually inspected the remaining 20 apps to conclude that the majority of them offered no protection against unauthorised application-level data access.

Subsequent to our work, Zhang et al. [100] described BLESS, a framework for performing taint-analysis over BLE-enabled Android applications to determine the presence (or absence) of encryption/authentication at the application layer. Rather than identifying the presence of standard cryptographic method calls, they instead look for the presence of keys and nonces, where user input is also assumed to be a key, and further take into consideration proprietary cryptographic implementations. We observe that proprietary implementations are explicitly discouraged by Android and may contain many vulnerabilities (as we demonstrate in Chapter 8). In addition, user input that is written to a BLE device may not necessarily be indicative of an authentication sequence or user authorisation. Many BLE devices allow the user to customise the device name, for example, which would also be accomplished via a GATT write. Applying BLESS to 1073 BLE-enabled Android APKs, the authors found that 76% of such APKs did not implement authentication protocols.

6.2 APK Dataset

We obtained our APK dataset from the AndroZoo project [110]. We focus on only those APKs that were sourced from the official Google Play store, which nevertheless resulted in a sizeable dataset of over 4.6 million APKs. The dataset includes multiple versions for each application, as well as applications that are no longer available on the marketplace. We perform our analysis over the entire dataset, rather than only those APKs that are currently available on Google Play. This is in part because older applications/versions may still be residing on users' devices, and in part to be able to identify trends in application-layer security deployments over time.

As we are only interested in applications that perform BLE attribute access, and because such access always requires communicating with the GATT server on the BLE Peripheral, we filter the APKs by the `BLUETOOTH` permission declaration and by calls to the `connectGatt` Android method,[1] which is called prior to performing BLE data reads or writes. This is achieved using `Androguard` [111], an open-source reverse-engineering tool that decompiles an Android APK and enables analysis of its components.[2] 18,929 APKs, comprising 11,067 unique packages,[3] from the original set of 4,600,000+ APKs satisfied this criteria, and these form our final dataset.

[1]`connectGatt` is a method within the `android.bluetooth.BluetoothDevice` class, within the Android SDK.

[2]We note here that `Androguard` does not support analysis of Native code, which may have resulted in the exclusion of some valid APKs.

[3]An Android application may have many versions, each of which will be a separate APK file (with a unique SHA256 hash), but all of which will have the same package name. We use the terms "unique applications" or "unique packages" to denote the set of APKs that contains only the *latest* version of each application.

Table 6.1: BLE data access methods.

Access	Method Signature*	#APKs	% of Total Methods†
Read	`byte[] getValue ()`	17896	61.58%
	`Integer getIntValue (int, int)`	8051	27.70%
	`String getStringValue (int)`	2313	7.96%
	`Float getFloatValue (int, int)`	800	2.75%
Write	`boolean setValue (byte[] )`	16198	70.49%
	`boolean setValue (int, int, int)`	5542	24.11%
	`boolean setValue (String)`	627	2.73%
	`boolean setValue (int, int, int, int)`	611	2.66%

* *All methods are from the class* `android.bluetooth.BluetoothGattCharacteristic`*.* † *"% of Total Methods" refers to the percentage of occurrences of a particular method for a particular data access type (i.e., read or write), with respect to all methods that enable the same type of data access.*

6.3 Identification of BLE Methods and Crypto-Libraries

We perform our analysis against specific BLE methods and crypto-libraries. When considering BLE methods, we focus on those methods that involve data writes and reads. Such methods have been listed in Table 6.1, and function as the starting point for our analysis. For data writes, the `BluetoothGattCharacteristic` class within the `android.bluetooth` package has `setValue` methods that set the locally-stored value of a characteristic. This is then written out to the BLE Peripheral. For data reads, the same class has `getValue` methods, which return data that is read from the BLE device (this includes data obtained via notifications). In a few APKs that we analysed, BLE data access methods were also called from within other, vendor-specific libraries. However, we do not include these in our analysis as they are now obsolete.

For cryptography, Android builds on the Java Cryptography Architecture [112] and provides a number of APIs, contained within the `java.security` and `javax.crypto` packages, for integrating security into applications. While it is possible for developers to implement their own algorithms, Android recommends against this [113]. We therefore consider only calls to these two packages as an indication of application-layer security.

6.4 `BLECryptracer`

Identification of cryptographically-processed BLE data is in essence a taint-analysis problem. For instance, a call to an encryption method will taint the output variable that may later be written to a BLE device. Therefore, when analysing data that is written to a BLE Peripheral, we consider the cryptography API calls as sources and the `setValue` methods as sinks. Similarly, for data that is read from the BLE device, we consider the `getValue` variants in Table 6.1 as sources and the cryptography API calls as sinks.

Figure 6.1 depicts an anonymised excerpt of code from a real-world APK, which shows the path taken from the output of `getValue` (i.e., BLE data that has been read from a connected device)

```
 1 .class Lcom/example/sdk/g/c$1;
 2 .method public onCharacteristicRead(Landroid/bluetooth/BluetoothGatt;Landroid/
      bluetooth/BluetoothGattCharacteristic;I)V
 3     ...
 4     invoke-virtual {p2}, Landroid/bluetooth/BluetoothGattCharacteristic;->
           getValue()[B
 5     move-result-object v0
 6
 7     iget-object v3, p0, Lcom/example/sdk/g/c$1;->a:Lcom/example/sdk/g/c;
 8     invoke-static {v3}, Lcom/example/sdk/g/c;->b(Lcom/example/sdk/g/c;)[B
 9     move-result-object v3
10
11     invoke-static {v3, v0}, Lcom/example/sdk/i/g;->a([B[B)[B
12     ...
13
14 .class public final Lcom/example/sdk/i/g;
15 .method public static a([B[B)[B
16     ...
17     new-instance v0, Ljavax/crypto/spec/SecretKeySpec;
18     const-string v1, "AES/ECB/NoPadding"
19     invoke-direct {v0, p0, v1}, Ljavax/crypto/spec/SecretKeySpec;-><init>([
           BLjava/lang/String;)V
20     const-string v1, "AES/ECB/NoPadding"
21     invoke-static {v1}, Ljavax/crypto/Cipher;->getInstance(Ljava/lang/String;)
           Ljavax/crypto/Cipher;
22     move-result-object v1
23
24     const/4 v2, 0x2
25     invoke-virtual {v1, v2, v0}, Ljavax/crypto/Cipher;->init(ILjava/security/Key
           ;)V
26     invoke-virtual {v1, p1}, Ljavax/crypto/Cipher;->doFinal([B)[B
```

Figure 6.1: Example smali code for cryptographically-processed BLE data.

to a cryptographic function.[4] In the figure, the obtained BLE data is stored in register `v0` (at line 5). This is used as the second argument to the function `Lcom/example/sdk/i/g;->a([B[B)[B` (line 11). Analysing the smali code of that function, we find that the second input argument is used as an input to a cryptographic function call (line 26). This provides clear evidence of cryptographically-processed BLE data.

The manual "tracing" we have described here is similar to what is performed by taint-analysis tools. There are a number of such tools available, e.g., `Flowdroid` [114] and `Amandroid` [115]. However, running a subset of our dataset of APKs through `Amandroid` (selected because of advantages over `Flowdroid` and other taint-analysis tools [116]), we found that analysis of a single APK sometimes utilised over 10GB of RAM and took several hours to complete. This computational complexity precludes bulk analysis of our dataset of several thousand APKs. We also found through manual analysis that many instances of cryptographically-processed data were not identified by `Amandroid`, especially when the BLE functions were called from third-party libraries. We therefore present a custom Python analysis tool called `BLECryptracer` [29], built on top of `Androguard`, to analyse *all* calls to BLE `setValue`/`getValue` methods within an APK. We note, however, that `BLECryptracer` does not handle certain data transfer mechanisms

[4]The code is in smali format. Android applications are typically written in Java and converted into Dalvik bytecode. The smali format is a human-readable representation of the bytecode.

(discussed in §6.8), which more complex tools such as `Amandroid` and `Flowdroid` do. This could have resulted in missed instances of cryptographically-protected BLE data.

`BLECryptracer` traces values to/from BLE data access functions and determines whether the data has been cryptographically processed. To achieve this, it employs the technique for tracing register values that we have just described (when explaining Figure 6.1). This technique is sometimes referred to as "slicing" and has been utilised in several static code analyses [117–119]. It also traces fields, as well as messages passed via Intents[5] and certain threading functions, e.g., `AsyncTask`. It returns TRUE at the first instance of cryptography that it encounters and FALSE if it is unable to identify any application-layer security with BLE data.

Our tool analyses BLE reads and writes separately, as the direction of tracing is different in the two cases. The two trace mechanisms are described in greater detail in §6.4.1. Tracing is performed at three levels of granularity, in the following order:

1. Direct trace - Attempt to identify link between BLE and cryptography functions via direct register value transfers and as immediate results of method invocations.
2. Associated entity trace - If the direct trace does not identify a link between source and sink, analyse abstract/instance methods and other registers used in previously analysed function calls.
3. "Lenient" trace - If the above methods fail to return a positive result, perform a search through all previously encountered methods (which would have originated from the BLE data access method), to determine if cryptography is used *anywhere* within them.

The results produced by the first trace method will be the best indication of cryptographically-processed BLE data, as the coarse-grained analyses performed in the subsequent methods add increasing amounts of uncertainty. For this reason, `BLECryptracer` assigns *Confidence Levels* of High, Medium and Low to its output, which correspond to the three trace methods above, to indicate how certain it is of the result. We evaluate these confidence levels against a modified version of the DroidBench benchmarking suite in §6.5.1. Note that `BLECryptracer` only looks for application-layer security in benign applications, and these confidence levels apply only when deliberate manipulations are not employed to hide the data flow between source and sink.

6.4.1 Trace Mechanisms

As mentioned previously, `BLECryptracer` analyses BLE reads and writes separately, using two different trace directions. We describe the two directions of tracing in greater detail below.

Backtracing BLE writes BLE writes on Android use one of the `setValue` methods in Table 6.1 to first set the value that is to be written, before calling the method for performing the actual write. `BLECryptracer` identifies all calls to these methods, and then traces the origins of the data held in the registers that are passed as input to the methods.

[5]By matching the `Extra` identifier within the calling method.

```
1  .method private a(Landroid/bluetooth/BluetoothGatt;[B...)V
2    ...
3    invoke-virtual {v0, v3}, Landroid/bluetooth/BluetoothGattService;->
         getCharacteristic(Ljava/util/UUID;) Landroid/bluetooth/
         BluetoothGattCharacteristic;
4    move-result-object v3
5    ...
6    invoke-virtual {v3, p2}, Landroid/bluetooth/BluetoothGattCharacteristic; ->
         setValue([B)Z
7    invoke-virtual {v1, v3}, Landroid/bluetooth/BluetoothGatt; ->
         writeCharacteristic(Landroid/bluetooth/BluetoothGattCharacteristic;)Z
```

Figure 6.2: Sample smali code for BLE attribute write.

Considering the smali code in Figure 6.2 as an example, `setValue` is invoked at line 13 and is passed two registers as input. As `setValue` is an instance method, the first input, local register `v3`, holds the `BluetoothGattCharacteristic` object that the method is invoked on. The second input, parameter register `p2`, holds the data that is to be written to the BLE device, and is the second argument that is passed to the method `a` (line 1 in Figure 6.2). `BLECryptracer` identifies `p2` as the register that holds the data of interest, and traces backward to determine if this data is the result of some cryptographic processing. To achieve this, the method(s) within the APK that invoke method `a` are identified, and the second input to each such method is traced. If the BLE data had come from a local register, rather than a parameter register, `BLECryptracer` would have traced back *within* method `a`'s instructions, to determine the origin of the data. This backtracing is performed until either a crypto-library is referenced, or a `const-<>` or `new-array` declaration is encountered (which would indicate that no cryptography is used). Note that calls to any method within the crypto-libraries mentioned in §6.3 are accepted as evidence of the use of cryptography with BLE data. The tool stops processing an APK at the first instance where such a method call is identified.

During execution, `BLECryptracer` maintains a list of registers (set within the context of a method) to be traced, for every `setValue` method call within the application code. This initially contains the second input register for each `setValue` method call. A new register is added to the list if it appears to have tainted the value of any of the registers already in the list. This could be due to simple operations such as `aget`, `aput` or `move-<>` (apart from `move-result` variants), or it could be as a result of a comparison, arithmetic or logic operation (in which case, the register holding the operand on which the operation is performed is added to the trace list). Similarly, if a register obtains a value from an instance field (via `sget` or `iget`), then all instances where that field is assigned a value are analysed. However, the script does not analyse the *order* in which the field is assigned values, as this would require activity life-cycle awareness.

Where a register is assigned a value that is output from a method invocation via `move-result`, if the method is not an external method, then the instructions within that method are analysed, beginning with the return value and tracing backwards. In some instances, the actual source of a register's value is obfuscated due to the use of intermediate formatting functions. In an attempt to overcome this, `BLECryptracer` traces the inputs to called methods as well. Further,

```
 1  .method public onCharacteristicread(Landroid/bluetooth/ BluetoothGatt;Landroid/
       bluetooth/ BluetoothGattCharacteristic;I)V
 2    ...
 3    invoke-virtual {p2}, Landroid/bluetooth/ BluetoothGattCharacteristic;->
         getValue()[B
 4    move-result-object v0
 5    new-instance v2, Ljava/lang/StringBuilder;
 6    invoke-direct {v2}, Ljava/lang/StringBuilder;-><init>()V
 7    const-string v3, "read value: "
 8    invoke-virtual {v2, v3}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)
          Ljava/lang/StringBuilder;
 9    move-result-object v2
10    invoke-static {v0}, Ljava/util/Arrays;->toString([B)Ljava/lang/String;
11    move-result-object v3
12    invoke-virtual {v2, v3}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)
          Ljava/lang/StringBuilder;
13    move-result-object v2
```

Figure 6.3: Sample smali code for BLE attribute read.

if a register is used as input to a method, then all other registers that are inputs to the method are also added to the trace list. While this captures some indirect value assignments, it runs the risk of false positives and is one scenario where a confidence level of Medium would be applied.

Forward-tracing BLE reads With BLE reads, a `getValue` variant is invoked and the output, i.e., the value that is read, is moved to a register. `BLECryptracer` identifies all calls to `getValue` variants, then traces the output registers and all registers they taint. With forward-tracing, the register holding the BLE data is considered to taint another if, for example, the source register is used in a method invocation, or comparison/arithmetic/logic operation, whose result is assigned to the destination register. The destination register is then added to the trace list. When a register is used as input to a method, then along with the output of that method, the use of the register *within* the method is also analysed.

This method of analysis tends to result in a "tree" of traces. As an example, considering the smali code in Figure 6.3, the byte array output from the `getValue` call in line 3 is stored in register `v0` (line 4). This taints register `v3` via a format conversion function (lines 10 and 11), which in turn taints `v2` via a `java.lang.StringBuilder` function (lines 12 and 13). At this point, all three registers (`v0`, `v2`, `v3`) are tainted and will be traced until either a crypto-library is referenced or the register value changes. Such value changes can occur due to `new-array`, `new-instance` and `const` declarations, as well as by being assigned the output of various operations (such as method invocations or arithmetic/logic operations).

6.4.2 Handling Obfuscation

APKs sometimes employ obfuscation techniques to protect against reverse-engineering, and the question then arises as to whether these techniques may impact the results of our analysis. We therefore briefly discuss common obfuscation techniques (as described in [120]) and their impact on our tool.

One of the most common techniques is *identifier renaming*, where identifiers within the code are replaced with short, meaningless names. However, within smali code, even such short names can be uniquely identified and traced, enabling `BLECryptracer` to overcome the challenges posed by this obfuscation technique. *String encryption* is another obfuscation mechanism, but it again does not affect the output of our tool as `BLECryptracer` does not search for hard-coded strings. Further, our tool was verified successfully against three out of four benchmarking applications that utilised *reflection* (however, two such identifications were at confidence level Low). `BLECryptracer` does not handle *packing* and *runtime-based obfuscation*, due to the complexity of analysing such techniques.

6.5 Evaluation

We evaluate `BLECryptracer` in terms of both accuracy and execution times. For comparison purposes, we include test results from `Amandroid` as well.

6.5.1 Accuracy Measures

At present, there is no dataset of real-world APKs with known use of cryptographically-processed BLE data, i.e., ground truth. Therefore, in order to test our tool against different data transfer mechanisms, we re-factored the DroidBench benchmarking suite [121] for the BLE case.

DroidBench is a test suite for evaluating the effectiveness of APK taint-analysis tools. It contains a number of Android applications, demonstrating different data transfer mechanisms. We cloned each DroidBench test app twice and modified the data flow between the sources and sinks such that in one app, the data would travel from `getValue` to a cryptography method invocation, and in the other app, from the cryptography method invocation to `setValue`. This emulates cryptographically-processed reads and writes, respectively. Some DroidBench test cases were excluded as they were found to be irrelevant due to differences in the objectives of DroidBench and our test set, e.g., applications that employ emulator detection or which leak contextual information in exceptions. Further, applications where BLE data is written to or read from files, or which contain data leaks in inactive code segments were not included. In total, we created 184 APKs: 92 for reads and 92 for writes.

We executed `BLECryptracer` against our benchmarking test set, analysed the results and obtained performance metrics in terms of the three different confidence levels. The statistics differ based on the type of access that is analysed (i.e., reads vs. writes) due to differences in the tracing mechanisms. The same test set was also used against `Amandroid` for comparison. Table 6.2 presents the performance metrics for both tools.

In the case of `BLECryptracer` results, the metrics are with respect to the total analysed APKs at each confidence level. That is, because lower confidence levels analyse only those APKs that do not get detected at higher levels, accurate metrics can only be derived by considering the set of APKs that were actually analysed at each level. For example, when considering the analysis of BLE reads, while the entire dataset of 92 APKs is relevant for confidence level High, only the

Table 6.2: Accuracy statistics.

Access	Tool	Conf.	#APKs*	Id.†	TP	FP	TN	FN	Precision	Recall	F-measure
Read	Amandroid	N/A	92	49	44	5	10	33	90%	57%	70%
	BLECryptracer	High	92	62	58	4	11	19	94%	75%	83%
		Med	30	11	7	4	7	12	64%	37%	47%
		Low	19	12	8	4	3	4	67%	67%	67%
Write	Amandroid	N/A	92	56	49	7	8	28	88%	64%	74%
	BLECryptracer	High	92	50	46	4	11	31	92%	60%	72%
		Med	42	22	19	3	8	12	86%	61%	72%
		Low	20	10	5	5	3	7	50%	42%	45%

**Number of APKs tested. Note that, for confidence levels Medium and Low, we don't consider the APKs detected at higher confidence levels. †The number of APKs that were identified as having cryptographically protected BLE data.*

30 APKs that do *not* result in a TRUE outcome at level High will be analysed for confidence level Medium. This also means that, when obtaining performance metrics for confidence level High, all TRUE results obtained at levels Medium and Low are taken to be FALSE.

The DroidBench test set, and hence our benchmarking suite, is an imbalanced dataset, containing far more samples *with* leaks (77) than *without* (15). For this reason, metrics such as accuracy are not suitable for analysing the performance of our tool when executed against this test set, as they are more susceptible to skew [122,123]. For our analysis, we instead compare the combined True Positive Rate (TPR) and False Positive Rate (FPR), and the combined precision-recall, in-line with other taint-analysis evaluations [124].

Table 6.2 presents the precision and recall (i.e., TPR) for both `BLECryptracer` and `Amandroid`. We further derive FPRs for both tools. With `BLECryptracer`, when analysing reads, False Positive Rates steadily increase as the confidence level reduces, as expected, with values of 27% for confidence level High, 36% for Medium and 57% for Low. When analysing writes, the FPR values are 27%, 27% and 63%, respectively. Regardless of the data access mechanism being tested, `BLECryptracer` (considering only the results at High confidence, for a fairer comparison) performs better than `Amandroid` in terms of FPR, with 27% vs. 33% for reads and 27% vs. 47% for writes. Precision values are also better in the case of `BLECryptracer` for both reads and writes.

In terms of the True Positive Rate, `BLECryptracer` performs better than `Amandroid` for reads at 75% vs. 57%, and slightly worse for writes at 60% vs. 64%. These results show that, overall, `BLECryptracer` performs better than `Amandroid` for analysing the presence of cryptographically-protected BLE data.

It should be noted that three of the four False Positives obtained by `BLECryptracer` at the High confidence level were due to the order in which variables are assigned values (i.e., life-cycle events), which is not tested for by `BLECryptracer`. Other data transfer mechanisms not tested for are `Looper` and `Messenger` functions, which generate False Negatives. The remaining False Positive was due to the presence of method aliasing and was also identified as a False Positive

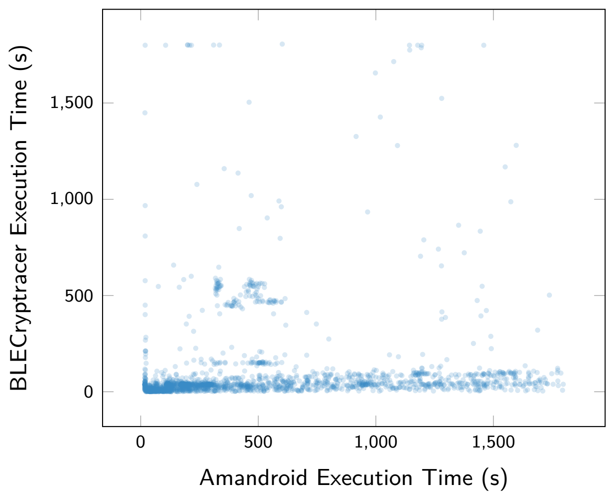


Figure 6.4: Comparison of time taken to execute `BLECryptracer` vs. `Amandroid`, when analysing BLE writes. Each point represents an APK.

by `Amandroid`. In addition, the unexpectedly low TPR (i.e., recall) at level Medium for reads is due to the relatively few cases analysed at that level when compared to High.

6.5.2 Execution Times

We also compare `BLECryptracer` and `Amandroid` in terms of speed of execution. For this, we ran the two tools against a random subset of 2,000 APKs (from our dataset of 18,000+ APKs) and compared time-to-completion in both cases. We imposed a maximum run-time of 30 minutes per APK for both tools, and only compared execution times for those cases where `Amandroid` did not time out. Approximately 40% of the tested APKs timed out when analysed by `Amandroid`. In comparison, fewer than 2% of APKs timed out when analysed by `BLECryptracer`.

Figure 6.4 plots the time taken to analyse BLE writes using `BLECryptracer` vs. `Amandroid`. The figure shows that analysis times with `BLECryptracer` were, for the most part, around 3-4 minutes per application. We observed no obvious correlation between the size of the application's dex file and the execution time, for either tool. APKs that took longer to process with `BLECryptracer` were predominantly of confidence level "Medium", which indicates that the longer analysis times may simply have been because of having to first go through the most stringent analysis (at confidence level High). For `Amandroid`, the execution times vary to a greater extent than with `BLECryptracer`, due to the difference in the mechanisms employed for performing the analysis.

6.6 Results from Large-Scale APK Analysis

We executed `BLECryptracer` against our dataset of 18,929 APKs. 192 APKs timed out when analysing reads and 220 APKs timed out when analysing writes, when a maximum runtime of 30 minutes was imposed. These APKs were re-tested with an increased runtime of 60 minutes. However, even with the longer analysis time, 44 and 76 APKs timed out for reads and writes, respectively, and had to be excluded from further analysis. In addition, approximately 90 APKs could not be processed via `Androguard`'s `AnalyzeAPK` method and were excluded.

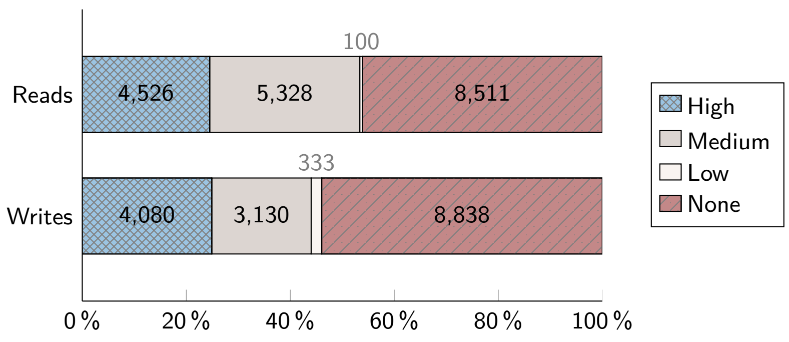


Figure 6.5: Analysis results depicting the presence of cryptographically-processed data with BLE reads and writes, with breakdown according to confidence level.

Due to the differences in performance metrics obtained for the three confidence levels during testing (as mentioned in §6.5), we focus on only those results that either identify cryptographically-protected BLE data at confidence level High or those where no protection (for BLE data) was identified at all.

6.6.1 Presence of App-Layer Security with BLE Data

Our results show that approximately 95% of BLE-enabled APKs call the `javax.crypto` and `java.security` cryptography libraries *somewhere* within their code. While this is a large proportion of APKs, the results also indicate that a much smaller percentage of APKs use cryptographically-processed data *with BLE reads and writes* (approximately 25% for both, identified with High confidence). In fact, about 46% of APKs that perform BLE reads and 54% of those that perform BLE writes (corresponding to 2,379 million and 2,075 million cumulative downloads, respectively) do *not* implement security for the BLE data. Interestingly, of the 16,131 APKs that call both BLE read and write functions, about 36% (i.e., more than 5,700 APKs), with a cumulative download count of 1,005 million, do not implement application-layer security for either type of data access. Figure 6.5 summarises the proportion of APKs that were identified as containing cryptographically-protected BLE data at the three different confidence levels.

6.6.2 Libraries vs. App-Specific Implementations

We found that many BLE-enabled APKs actually use third-party libraries for incorporating BLE functionality. To get an idea of exactly how many APKs relied on libraries, we analyse all methods within an APK that called BLE data access functions. To do this in an automated way, we compare the method class name with the application package name. If the first two components (e.g., `com.packagename`) of each match, then we take it to be a method implemented within the application. If the components do not match, we take it to be a library method. If the package name uses country-code second-level domains (e.g., `uk.ac.packagename`), then we compare the third components as well. We note that this technique is fairly rudimentary and will not identify, for example, instances where libraries are repackaged into an app using the application's own package name, e.g., if a library `com.lib` is packaged as `com.packagename.lib` into an app that has package name `com.packagename`, the library will not be identified.

Table 6.3: Top ten third-party BLE libraries.

Library	Function	#APKs [unique]	Crypto
Estimote	Beacon	3980 [2804]	Yes
Nordicsemi*	DFU	1238 [847]	No
Kontakt	Beacon	1108 [690]	No
Chromium	Web BLE	402 [269]	No
Randdusing	Cordova Plugin	268 [188]	No
Megster	Cordova Plugin	317 [187]	No
Flic	BLE Accessory	173 [164]	Yes
Polidea	BLE Wrapper	138 [114]	No
Evothings	Cordova Plugin	142 [84]	No
Jaalee	Beacon	102 [79]	No

* *Significant overlap between Estimote and Nordic due to repackaging of Nordic SDK into Estimote.*

Of the APKs that call the `setValue` method, 63% use BLE functionality solely through libraries, 32% use application-specific methods only, while 4% use both. Fewer than 1% of the APKs could not be analysed due to very short method names. Within the APKs that use both application-specific methods and libraries, around 34% use an external library to provide Device Firmware Update (DFU) capabilities.[6] Of the APKs that utilise only application-specific methods to incorporate BLE functionality, 67% do *not* implement application-layer security with the BLE data. This proportion was lower at 48% for applications that rely on libraries.

In the case of the APKs that call `getValue` variants, 37% use only application-specific methods, 58% use only libraries, and 5% use both. As with the `setValue` case, a higher proportion of APKs that use only app-specific BLE implementations were found to not use cryptography (60%), when compared with those that use only libraries (39%).

Table 6.3 presents the ten most common BLE libraries within our dataset, their functionality, the number of APKs that use them, and the presence of cryptographically-processed BLE data within the library. The table shows that the most prevalent libraries were those that enable communication with BLE beacons. In fact, a single such library (Estimote) made up more than 90% of all instances of cryptographically-processed BLE writes and 85% of cryptographically-processed BLE reads (identified with High confidence). An analysis of this library suggested that cryptography is being used to authenticate requests when modifying settings on the beacon.

Apart from beacon libraries, we identified five libraries that function as wrappers for the Android BLE API. For example, Polidea wraps the API so that it adheres to the reactive programming paradigm. The libraries Randdusing, Megster and Evothings enable the use of BLE via JavaScript in Cordova-based applications. Similarly, Chromium enables websites to access BLE devices via JavaScript calls. None of these libraries handle cryptographically-processed BLE data. It is expected that developers using these libraries will implement their own application-layer security (using either JavaScript or reactive Java as appropriate).

[6] DFU enables Peripheral firmware to be updated via the mobile application over the BLE interface.

Table 6.4: Number of packages with cryptographic misuse.

Misuse Type*	#Packages†	Misuse Type*	#Packages†
Bad key used with Cipher	11	ECB (or other bad mode)	10
Non-random IV	10	Bad IV used with Cipher	7
Non-random key	6	Incomplete operation (dead code)	4

**Description of misuse based on [117, 126].* † *Unique packages.*

Of the two remaining libraries, Flic, which uses cryptographically-processed data, is a library offered by a BLE device manufacturer. This library allows third-party developers to integrate their services into the Flic ecosystem, to allow them to automate certain tasks.

Finally, Nordicsemi (i.e., Nordic [Semiconductor]) is a key player in the BLE Peripheral SoC market [125]. It provides a library to enable DFU over the BLE interface. With the newest version of the DFU mechanism, the BLE Peripheral verifies that the firmware has been properly signed. Devices using the legacy DFU mechanism will not verify the firmware. However, the mobile application (and by extension, the library) does not need to handle cryptographically-processed data in either case.

6.6.3 Cryptographic Correctness

`BLECryptracer` identified 3,228 unique packages with cryptographically-protected BLE data (with either reads or writes), with High confidence. However, this in itself does not indicate a secure system. We therefore further analysed this subset of APKs to identify whether cryptography had been used correctly in them. The tool `CogniCrypt` [127] was utilised for this purpose. Although this tool does not formally verify the cryptographic protocol between the application and the BLE Peripheral, it identifies various misuses of the Java crypto/security libraries.

Among the 3,228 unique packages, we found that there was significant overlap between APKs in terms of the BLE libraries or functions used.[7] Removing such duplicates resulted in a set of 194 APKs. Of these, 68 were identified by CogniCrypt as having issues. However, because CogniCrypt identifies cryptography misuse within the entire APK, the results were filtered for BLE-specific functions. 24 APKs were found to have issues within or associated with the methods that processed BLE data (as identified by `BLECryptracer`) and often, a single APK exhibited multiple issues. Table 6.4 shows the different types of misuse encountered and the number of unique packages that were identified as having such misuse. Note that because this analysis was performed over unique packages, the number of *APKs* that contain security misconfigurations will be much higher.

[7]There are instances where two applications may have unique package names, but which actually incorporate much the same functionality. This is often the case when the same developer produces branded variants of an application for different clients in a single industry. For example, two applications could have unique package names `com.myapp1` and `com.myapp2`, but their functionality may be derived from a common codebase `com.myapp`. Further, two different applications (with distinct package names) that implement BLE functionality via the same library will have the same security stance in terms of protection for BLE data. Therefore, we would only need to analyse cryptographic correctness within the BLE library, rather than the applications.

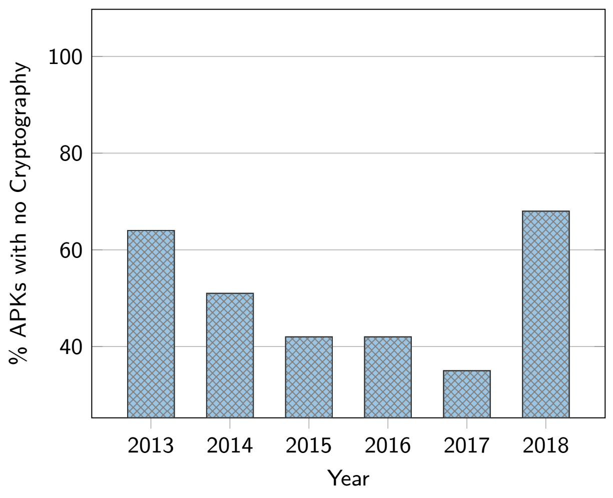


Figure 6.6: Application-layer security trends over time. Graph depicts APKs that perform BLE reads or writes, with no cryptographic protection for either.

We manually analysed the 24 packages that were flagged by CogniCrypt as having BLE-relevant issues, and examined the identified instances of bad cipher modes and hardcoded keys/Initialisation Vectors (IVs). With regard to insecure block cipher modes, we found that explicit use of ECB is prevalent (9 out of 10 cases), but `Cipher.getInstance("AES")` is used in one case without the mode being specified, which may default to ECB depending on the cryptographic provider. When analysing keys, we observed that several apps contain hardcoded keys as byte arrays or strings. Three applications retrieve keys from JSON files. In two cases, keys are generated from the `ANDROID_ID`, which is a system setting that is readable by all applications. We also observed one instance where a key is obtained from a server via HTTP (not HTTPS).

This analysis shows that several real-world applications contain basic mistakes in their use of crypto-libraries and handling of sensitive data, which means that the BLE data will not be secure despite the use of cryptography.

6.6.4 Trends over Time

Figure 6.6 shows the trend of application-layer security over time for applications that incorporate calls to BLE reads or writes. The graph depicts the percentage of applications that do *not* have cryptographic protection for either type of access. APKs with invalid dates [128] or older than 2012 (when native BLE support was introduced for Android) have not been included.

The overall downward trend suggests that there has been some improvement in application-layer security between the years 2014 and 2017 (we refrain from making observations about APKs from 2013 as they were very few in number, and about APKs from 2018 as this was the year in which our analysis was performed and the dataset was not yet fully populated at that time). However, it should be noted that, even in 2017, which had the smallest percentage of APKs without cryptography, these APKs corresponded to 128 million downloads, which is a significant number.

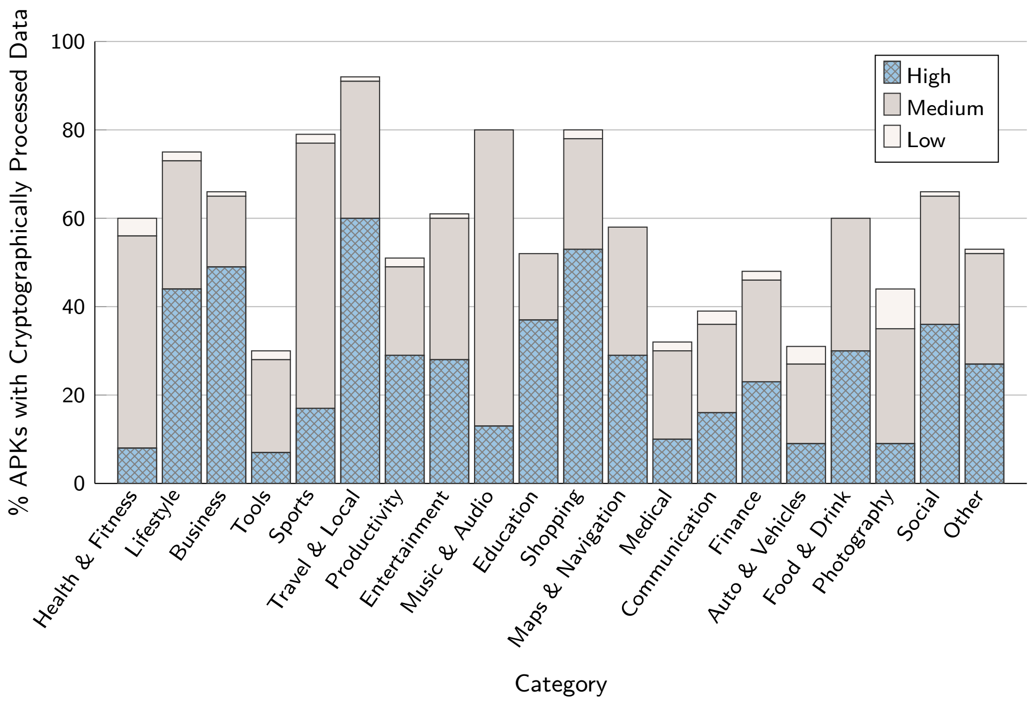


Figure 6.7: Presence of application-layer security in different categories of applications, averaged over BLE reads and writes, and broken down by confidence level. Only unique packages have been taken into consideration. APKs that do not currently have a presence on Google Play have been excluded, as their category cannot be identified.

6.6.5 Impact Analysis

While 18,929 BLE-enabled applications may seem like a relatively small number of applications when compared with the initial dataset of 4.6 million+, a single application may correspond to multiple BLE devices, sometimes even millions of devices as is the case with fitness trackers [129]. For example, even if we consider the slightly restrictive case of unique applications that do not use cryptography with either reads or writes, the cumulative install count is still in excess of 1,005 million. This shows that the attack surface is much larger than may be indicated by the number of APKs.

It is of course a possibility that the data that is read from a BLE Peripheral has no impact on user security or privacy (e.g., device battery levels). To understand the actual impact of the absence of application-layer security, we first need to understand the type of data that is handled by the BLE device. One possible way of obtaining this information is through the application categories on Google Play. That is, we would assume that an application that interfaces with a BLE glucometer would be categorised under “Medical” on Google Play. If a lack of end-to-end protection was identified in such an app, then the ramifications would be clear (i.e., loss of confidentiality for user health information).

On this basis, we graph the percentage of applications that use cryptographically-processed BLE data from each major application category in Figure 6.7. The results are in some cases unexpected. For example, it would be reasonable to expect that most “Medical” applications would implement some level of application-layer security for their BLE data. However, the results show that fewer than 30% of applications under this category actually have such protection mechanisms. It is possible that this is because the devices implement the services/profiles defined by the Bluetooth SIG (which do not mandate any security apart from pairing, as mentioned in §5.1). In fact, of the APKs categorised under "Medical" and with no cryptographic protection for either reads or writes, we found that three of the top ten (in terms of installations) contained identifiers for the standard Bluetooth Glucose Service.

Another surprising result was with APKs that are categorised under "Business", "Shopping" and "Travel & Local". The results indicate that such apps are the most likely to incorporate application-layer security, with around 50% of all such applications being identified as having cryptographically-processed BLE data with High confidence. However, in over 85% of such occurrences, this was found to be due to the Estimote beacon library.

Looking at some of the categories (such as "Tools" or "Productivity") in Figure 6.7, it becomes clear that Google Play categories are too coarse-grained to enable identification of the functionality of BLE devices that the mobile apps interact with. Further, it is possible that the categories don't always reflect *BLE* functionality, but rather other application functionality. This prompted a more fine-grained BLE-specific functionality mapping and impact analysis, which we present in Chapter 8.

6.7 Case Study: Firmware Update over BLE

When analysing our results, we found that one of the APKs that was identified as not having application-layer security was designed for use with the ID107 HR, a low-cost fitness tracker that, based on the install count on Google Play (1,000,000+), appears to be widely used. An analysis of the APK suggested that the device used the Nordic BLE chipset, which could be put into Legacy DFU mode (where the firmware does not need to be signed). To exploit this, we acquired the device and developed an APK that, in accordance with the attacks described in §5.2, connects to the device, sends commands to place it in DFU mode, and then writes a new modified firmware to the device without user intervention. The updated firmware in this case was a simple, innocuous modification of the original firmware. However, given that the device can be configured to receive notifications from other applications, malicious firmware could be developed in such a way that, for example, all notifications (including second-factor authentication SMS messages) are routed to the malicious application that installed the firmware.

This attack was possible because the BLE Peripheral did not verify the firmware (e.g., via digital signatures) nor the source application (via application-layer security). We have informed the application developer of the issue, but have not received a response.

While our attack was crafted for a specific device, it does demonstrate that attacks against these types of devices are relatively easy. An attacker could easily embed several firmware images within a single mobile application, to target a range of vulnerable devices.

6.8 Limitations and Future Work

In this section, we outline some limitations of `BLECryptracer` that may have impacted our results and discuss potential for future work.

Unhandled data transfer mechanisms and analyses As mentioned in §6.5, `BLECryptracer` does not analyse data that is written out to file (including shared preferences), or communicated out to a different application, because it is not straightforward (and many times, not possible) to determine how data will be handled once it has been transferred out of the application under analysis. It is also possible that an application obtains the data to be written to a BLE device from, or forwards the data read from a BLE device to, another entity such as a remote server. That is, the Android application could merely act as a "shuttle" for the data, which means that an analysis of the APK would not show evidence of usage of cryptography libraries. However, the transfer of data to/from a remote server does not in itself indicate cryptographically-processed data, as plain-text values can also be transmitted in the same manner. We therefore do not analyse instances of data transfers to external entities.

`BLECryptracer` does not handle activity life-cycle events. It also does not analyse data transfers between a source and sink when only one of them is processed within a `Looper` function or when the data is transmitted via messages. `BLECryptracer` additionally does not handle transfers when only one of source or sink is present within a callback.

Further, `BLECryptracer` is built on top of `Androguard`, which provides access to code components as smali. However, Native code is not supported, which means that there may be BLE data accesses that are not analysed by `BLECryptracer`.

Conditional statements with backtracing When backtracing a register, `BLECryptracer` stops when it encounters a constant value assignment. However, it is possible that this value assignment occurs within one branch of a conditional jump, which means that another possible value could be contained within another branch further up the instruction list. To identify this, the script would have to first trace forward within the instruction list, identify all possible conditional jumps, and then trace back from the register of interest for all branches. This would need to be performed for every method that is analysed and could result in a much longer processing time per APK file, as well as potentially unnecessary overheads.

Trace termination on first encounter of cryptography At present, `BLECryptracer` terminates analysis of an APK at the very first instance when it identifies cryptographically-processed BLE data. An ideal extension would be to broaden the scope of the tool to analyse the availability of higher-layer protection for each characteristic defined within the APK.

6.9 Chapter Summary and Next Steps

In this chapter, we have analysed the prevalence of end-to-end security in BLE deployments and have identified that a significant proportion of BLE devices are likely to be vulnerable to unauthorised data access at the application layer. We have also observed in some cases where cryptographic protection *is* present, that the crypto-libraries are misconfigured, which can lead to reduced security. This demonstrates that leaving security in the hands of developers can result in vulnerable BLE data. In the next chapter, we design and present a pragmatic solution to the application-level unauthorised data access vulnerability, taking into consideration the various stakeholders and restrictions within the BLE ecosystem.

7 A Solution for the Unauthorised Data Access Vulnerability

In this chapter, we present our proposed solution to the application-level unauthorised data access vulnerability in BLE. We set out security and system requirements and perform a multi-faceted stakeholder analysis, to arrive at an asymmetric specification-level modification. Our proposed solution is fully backward compatible with existing BLE Peripherals and applications. We also present an open-source proof-of-concept of our proposed solution, implemented on the Android-x86 platform, and test it against real-world devices and applications.

7.1 Introduction

We demonstrated in Chapter 5 that an unauthorised data access vulnerability exists on BLE-enabled multi-application platforms. The data on a BLE device[1] may be vulnerable if it does not incorporate end-to-end security (which is typically achieved via authorisation permissions). However, as we have shown in Chapter 6, a significant proportion of BLE devices apparently do not apply such application-layer security, and some that do have done so incorrectly. In any case, we note that authorisation permissions cannot be applied to SIG-defined characteristics when there is a need to maintain compatibility with SIG-defined services and profiles (most of which do not specify a requirement for authorisation permissions). Further, a user may desire the use of a secondary application to interface with their BLE device, which may not be possible if end-to-end protection is applied. Existing platform-imposed restrictions, mostly in the form of user-granted permissions for applications, do not restrict access on a per-peer basis.

Any solution that is proposed to mitigate this vulnerability must take into consideration not only technical limitations but also practical issues. That is, a highly secure solution that is not likely to be implemented in the vast majority of BLE deployments is not particularly useful.

We approach this problem from a practical standpoint. We define security and system requirements, based on typical BLE configurations, usage mechanisms and user involvement (§7.2). We also conduct a pragmatic stakeholder analysis, considering stakeholder numbers, likelihood of participation and technical capabilities (§7.3). Based on our evaluation, we propose a specification-level solution (§7.4) and show that it meets the requirements that we have set out (§7.5). To further demonstrate the viability of our solution, we implement a POC on the Android-x86 platform (§7.7). Specific benefits realised by our solution include no changes to

[1]In actuality, the multi-application platform is also a BLE-enabled device and can operate as a Peripheral. However, we use the term "BLE device" or "Peripheral" to denote the "typically Peripheral" devices that the multi-application platform connects to.

existing layers within the BLE stack, no changes to existing apps, equal protection for SIG- and vendor-defined services, and most changes being applied to mature platforms with stable update capabilities (i.e., existing node devices need not be modified).

Related work A number of studies have suggested different techniques to add on security at the application layer, but each have their own shortcomings. For example, Naveed et al. [48] propose a re-architected Android framework which automatically creates a bonding policy between a Bluetooth device and the first application that attempts to pair with it. The solution would not come into play if a Bluetooth device did *not* require pairing. Our solution protects BLE data from unauthorised access, regardless of whether or not the data/device specifies a security requirement. It also explicitly informs the user of any application that makes a GATT request to a connected BLE device. This ensures that the user is aware of which applications have attempted to access data on their BLE devices and can make decisions regarding whether or not to allow access. The solution by Naveed et al. [48] also assumes that the user will only want to use a single application with their device. However, in the BLE case, we have observed that users often utilise a secondary app with additional features. This is supported by our solution.

Ortiz-Yepes [17] proposes BALSA, an application-layer security add-on comprising an additional authentication back-end and a Kerberos-like protocol. As described in §4.4, BALSA is intended to work on top of existing BLE Peripherals (termed *Sensors*) and existing mobile devices without modifications to the platforms. However, because this requires additional infrastructure and setup, it would not be a suitable option for the vast majority of end users.

Zhang et al. [100] propose a public key cryptography-based solution, where the public key is made available as a QR code on a tamper-evident label on the BLE Peripheral and the Peripheral firmware implements a custom application-level protocol. However, this solution cannot be retrofitted to the billions of BLE Peripherals extant in the world today.

7.2 Environment

In this section, we outline our threat model (§7.2.1) and define a set of security and system requirements (§7.2.2, §7.2.3) that should apply to any solution that addresses the problem of unauthorised data access on multi-application BLE platforms, based on our threat model.

7.2.1 Threat Model

We consider the common use case of a BLE device interfacing with one (or more) applications on a multi-application platform. We make the assumption that any app on the multi-application platform issues BLE requests via a BLE stack implemented by the platform, i.e., the application cannot circumvent the stack that is implemented by the platform. We also assume that the application cannot directly access the components of the platform-implemented BLE stack or influence its operations by any means other than via robust platform APIs. In addition, we assume an honest and uncompromised platform and BLE device. However, *applications* are from multiple third-party developers and are assumed to be untrusted. These applications may abuse the unauthorised data access vulnerability to obtain and manipulate data being stored on a BLE device without the user's knowledge and consent. This is the kind of behaviour our solution aims to protect against.

7.2.2 Security Requirements

To protect against unauthorised data access attacks at the application-layer, we define three main security requirements. These are based on typical multi-application platform configurations and usage, and the shortcomings that we have identified with existing restriction mechanisms.

SecRQ1: Prevention of unauthorised access to BLE data An application should not be able to access the data from a BLE peer device without the user's knowledge and explicit authorisation. Note that the term "authorised" in this context should not be confused with the "authorisation permissions" already defined within the Bluetooth specification.

SecRQ2: Per-device access control User authorisation should be granted to an application for each peer device *individually*, i.e., if an application is granted permission to access one BLE device, it should not automatically be possible for the application to access other BLE devices.

SecRQ3: Access revocation It must be possible for the user to revoke access that has previously been granted to an application for a BLE peer device. This limits the exposure of data in the event of late identification of malicious application behaviour.

7.2.3 System Requirements

There are billions of BLE-enabled devices in use today and most are in consumer applications, where the end users are not necessarily highly technical. This results in a need for security solutions that do not require high levels of user involvement. In addition, most BLE Peripherals are resource-constrained by design and will not be able to handle large amounts of processing. This makes complex cryptographic protocols less desirable. While these factors are not directly related to the security of a system, they need to be taken into consideration when proposing a security solution. We therefore define three key *system* requirements, bearing in mind user involvement, the number of BLE devices extant in the world today and the asymmetric nature of resources on communicating BLE devices.

SysRQ1: Protection by default All devices that implement (the modified version of) BLE should incorporate protection by default. Any specification-compliant BLE system should automatically protect against and be protected from unauthorised data access at the application layer, without the need for additional user intervention (beyond the explicit granting of permissions or authorisation).

SysRQ2: Backward compatibility Devices that incorporate the solution should function with existing devices. Given that billions of BLE-enabled devices exist today, a solution that obsoletes such a vast number of devices would be unacceptable.

SysRQ3: Minimal overhead for resource-constrained devices The solution should not incur a significant processing overhead for the more resource-constrained device, as this would lead to greater power requirements and quicker battery drain, thereby defeating the purpose of BLE.

7.3 Devising a Solution Strategy

The requirements we have described in §7.2.2 and §7.2.3 are necessary for a secure and utilitarian solution to the unauthorised data access problem. However, the most secure solution is of no value if it will not be applied to a large proportion of the BLE ecosystem due to lack of technical capability or stakeholder involvement. In this section, we discuss the primary stakeholders within the BLE ecosystem and describe practical considerations that should be taken into account when proposing a solution. From this, we determine the most suitable solution strategy to ensure maximum coverage.

7.3.1 Stakeholders within BLE

There are five primary stakeholders within the BLE ecosystem:

1. **The Bluetooth SIG:** This is the entity that defines and maintains the Bluetooth specification, as well as BLE services and profiles (such as the Glucose Profile described in §5.1).
2. **Chipset vendors:** These are entities that produce BLE-enabled chipsets, which are then used in platforms and Peripherals. Chipset vendors may also provide BLE stacks for their products, to enable application developers to create BLE end products quickly and easily. Examples include Qualcomm, NXP, Nordic Semiconductor, Texas Instruments, STMicroelectronics and several others.
3. **Platform vendors:** These are entities that develop and maintain BLE-enabled platforms, typically supporting multiple applications. Prominent examples are Android, iOS/Mac OS, Windows and Linux.
4. **Developers:** These entities produce BLE-enabled end products (e.g., fitness trackers, eScooters, smart locks). They normally also develop companion applications (which typically run on multi-application platforms) to interface with their products.
5. **Users:** The ultimate consumers for BLE-enabled products and services.

Users are only considered in terms of the impact of the vulnerability and the ease of applying a solution. Users are at most expected to update their devices' operating system or firmware and provide explicit authorisation to applications. We do not expect users to *implement* any part of a solution. We therefore confine the discussion on implementation to the first four entities.

7.3.2 Practical Considerations

When a BLE security solution is proposed, the likelihood of it being implemented depends on a number of factors. In this section, we analyse those factors in terms of the involved stakeholders.

7.3.2.1 Number of Entities

The likelihood of a solution being implemented depends in part on the number of entities that are required to implement it. The smaller the number, the easier it is to communicate the solution to them and the greater the reach of the solution. When considering BLE stakeholders in terms of numbers, the SIG is a single entity (albeit made up of a large number of members). This makes it a single point of communication, from which the solution will trickle down to implementing entities (platform vendors, chipset vendors and developers). There are a limited number of platform vendors, and the four most prominent platforms (Android, Windows, iOS, and Mac OS) account for over 95% of the worldwide OS market share [130]. BLE chipset vendors are more numerous than platform vendors, but not by a large margin (15-20 vendors [131]). Developers, on the other hand, are multitudinous (several hundreds [132]); it would therefore be very difficult to communicate a solution to all possible developers.

7.3.2.2 Stakeholder Participation

Not all stakeholders respond satisfactorily when they are made aware of a vulnerability. The security behaviour of a stakeholder tends to be associated with the prominence of the stakeholder (in terms of brand value, which may act as an incentive to adopt strong security practices), as well as the availability of organisational support, knowledge and resources [133–135]. The SIG and most platform/chipset vendors have clear, mature processes in place for vulnerability reporting, assessment and mitigation [136–140], whereas many developers may not even respond when informed of issues [141]. We have found this to be the case when reporting vulnerabilities.

Further, in inter-dependent ecosystems such as BLE, where platforms and chipsets implement the specification, and developers create end-products on top of the platforms and chipsets, there is a certain degree of "responsibility relaying". That is, each stakeholder presumes that the responsibility for implementing the solution belongs to another stakeholder. This phenomenon of "passing the buck" is prevalent within IT security, with responsibility being transferred down the supply chain or to other stakeholders [142, 143]. In the case of BLE, we postulate that only a specification-level change will induce most of the remaining stakeholders within the BLE ecosystem to incorporate a solution; the solution would have to be implemented in order to claim conformance with the specification.

7.3.2.3 Availability of Update Mechanism

Stakeholder participation, as described in preceding sections, is key to solution implementation, but equally so is the availability of an actual mechanism for performing the implementation.

In the case of the Bluetooth specification, all updates are to a document, which can be updated in a straightforward manner. The solution must thereafter be implemented by the remaining stakeholders. Platform devices, such as mobile phones and computers, run operating systems that have fairly robust update mechanisms. Therefore, a solution implementation can be easily rolled out on these devices. Most modern BLE chipsets support over-the-air (OTA) firmware updates, enabling updates to applications and sometimes also to the BLE stack [144–147].

However, many IoT devices do not incorporate such update mechanisms [148] (we verify this in Chapter 8); this means that a large proportion of existing BLE *Peripheral* devices cannot be modified.

7.3.3 Discussion

Based on the large number of BLE end-product developers, the lower likelihood of developer participation, and the lack of firmware update mechanisms in many BLE Peripherals, we reach the conclusion that a security solution that does *not* require involvement from end-product developers is more likely to actually be implemented. We also observe that, because a single platform device normally communicates with multiple Peripherals, an asymmetric solution involving changes to only the platforms (which are far fewer in number) will be an effort-efficient way to resolving the unauthorised data access vulnerability for a larger proportion of the BLE ecosystem. Further, according to §7.3.2.2, a specification-level change is more likely to prompt changes from implementing entities than individual communications with each entity. In addition, a specification-level change ensures security by default even if new BLE-enabled multi-application platforms are introduced in the future (without the need for communicating the solution to each new platform vendor individually).

7.4 Solution Design

In accordance with our analysis in §7.3, our proposed solution involves changes to the BLE stack and primarily involves modifications to multi-application platforms. Our solution also requires minor changes to the Peripheral. However, as we discuss in §7.6, the changes to the Peripheral can be avoided while still retaining the expected outcome.

Our solution introduces three new BLE components/properties:

1. **ATT Access Database (AAD):** A database for storing application access permissions.
2. **ATT Access Manager (AAM):** A layer within the BLE stack, responsible for performing the main access control functions.
3. **Device/Platform Mode:** A property for a BLE system, which controls the behaviour of a BLE device with respect to the new functionality.

§7.4.1 to §7.4.3 describe the purpose and, if relevant, the functionality of each of these elements in detail. §7.4.4 discusses concerns relating to user authorisation, while §7.4.5 describes access revocation.

7.4.1 The ATT Access Database (AAD)

The AAD stores per-device (i.e., BLE peer), per-application *access records*, as authorised by the user, for all BLE peer devices connected to the platform, for all applications that have made a GATT request for a BLE peer.

An access record has three components:

1. **AppID:** A unique identifier for the application that has made a GATT request to the platform. This must be assigned by the platform. It should not be possible for the application to manipulate its AppID.
2. **DeviceID:** A unique identifier for the BLE peer, e.g., the device's hardware address.
3. **Permission:** A value indicating whether access for an application (as identified by AppID) to a BLE device (identified by DeviceID) is *Allowed* or *DenyListed*.

By default, records do not exist for an application until the application makes a GATT request. The first time a record is added to the AAD for an application-device pair, the associated permission will be as selected by the user. This is described in detail in §7.4.2.

Positioning of the AAD Similar to the Security Database used within the existing design of BLE, the AAD does not feature within our modified BLE *stack*, but must be implemented by the platform in order for the BLE system to be operational. The AAD only communicates with the AAM. Therefore, the functionality of the AAD can also be subsumed into the AAM.

Access by Applications The AAD must not be accessible to higher layer applications. It should not be possible for an application to query its AAD permissions or to add itself to the AAD, as that would defeat the access control mechanisms that are in place.

7.4.2 The ATT Access Manager (AAM)

We introduce the AAM as a new layer within the BLE Host subsystem. It serves as an access control mechanism for GATT requests. For this reason, it is logically positioned between GATT and the application layer. This position enables it to intercept and arbitrate all GATT requests while also not unduly interfering with applications or lower stack layers.

Basic Workflow Figure 7.1 depicts the overall workflow when a GATT request is received from an application. When an application makes a GATT request, the platform passes the AAM a 3-tuple, consisting of the GATT request, the DeviceID corresponding to the BLE peer, and the AppID representing the requesting application. The DeviceID and AppID are assigned by the underlying platform (as described in §7.4.1). The AAM separates out the three elements and queries the AAD for the DeviceID/AppID combination. The following outcomes are possible:

- **An entry exists** within the AAD for the AppID against the given DeviceID: The AAD forwards the corresponding permission value to the AAM.
  - **The stored permission is *Allowed*:** The AAM forwards the GATT request to ATT/GATT, receives the response, and forwards it to the platform.
  - **The stored permission is *DenyListed*:** This indicates that the user has expressly denied the application from accessing data on the specific BLE peer. The AAM performs no further processing, but indicates the deny-listed status to the platform, which should notify the requesting application that the request has failed and cannot succeed even after multiple tries.

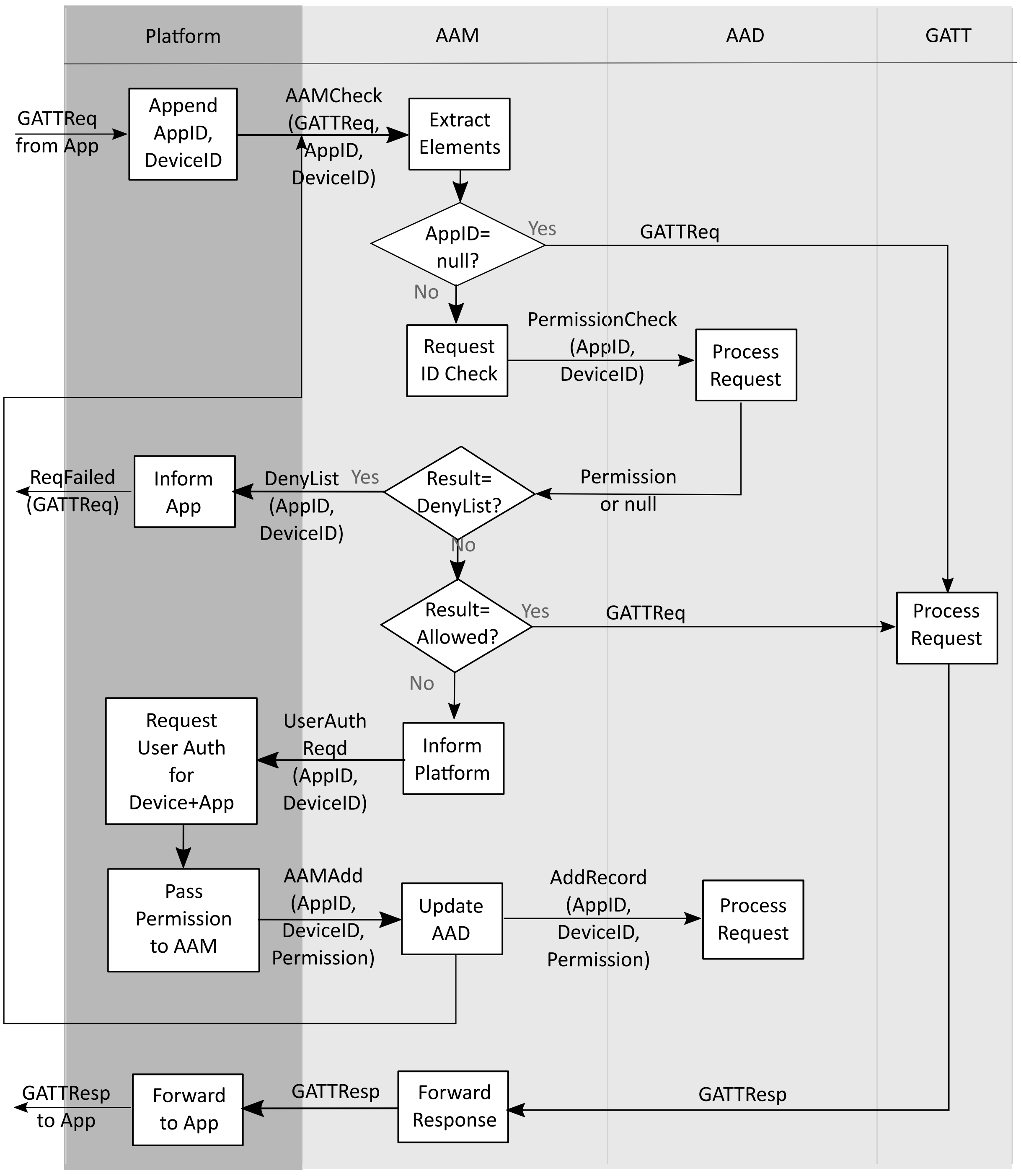


Figure 7.1: Proposed workflow for GATT requests. Light grey shaded areas are part of the BLE stack. Dark grey areas are platform/application components external to the stack.

- **No record exists** with the AAD for an AppID-DeviceID pair: This is signalled to the AAM, which in turn notifies the underlying platform that user authorisation is required. The user should be presented with options to allow or deny the application to access data from the BLE peer.

In this manner, only applications that are explicitly authorised by the user will be able to access data from the BLE peer, and because permissions are defined per-peer and per-application, the user has complete control over exactly which BLE devices each application can access.

Note also that, in this design, the functionality of GATT and other existing BLE stack components do not change. Therefore, changes to the stack are minimal. Further, the AAM only processes GATT *requests*; it forwards GATT responses as received to higher layers.

Null AppIDs Upon receiving a `null` AppID, the AAM will forward the GATT request to ATT/GATT without any further checks. The AAM can only be sent `null` AppIDs if the underlying platform hosts a single application. See §7.4.3 for more details.

7.4.3 Device Mode

We define two modes, "Single-App" and "Multi-App", based on the application hosting configuration of the platform:

A Single-App device is a BLE-compatible device that hosts only one application. A Multi-App device is a BLE-compatible device that may host more than one application.

Examples of Single-App devices are fitness trackers, glucose monitors, door locks or insulin pumps. These devices are resource-constrained and their firmware generally contains only one set of application code. Devices such as mobile phones and personal computers, on which many applications run, would fall under the Multi-App category. The Device Mode is assigned to a device at the time of manufacture and cannot be changed during operation.

Note that the terms "Single-App device" and "Multi-App device" do not refer to how many applications a BLE-enabled device can *interface with*, but rather how many applications a BLE-enabled device *hosts*.

A Single-App device functions almost exactly as BLE does today. The AppID is set to `null`; the AAM simply passes through requests it receives from higher layers to GATT, without performing any further checks; the AAD is dormant. It is important to note that the AppID can only be `null` for Single-App devices.

A Multi-App device incorporates the complete functionality of the AAM, has a functional AAD, and behaves as described in the preceding sections. Because Multi-App devices tend to have greater storage and processing capabilities, we foresee that these changes will not be overly burdensome.

By defining the Device Mode in this manner, the new stack enables resource-constrained devices (that typically host a single application and would therefore be defined as Single-App devices) to operate almost exactly as they always have, thereby not incurring any significant overheads due to additional processing.

7.4.4 Obtaining User Authorisation

The mechanism of obtaining user authorisation will depend on the implementing platform. However, the requirement is that the mechanism must be explicitly visible to the user. We do not foresee that this will present a problem, as authorisation is only required on Multi-App platforms (such as mobile phones and laptops), which typically have fully-fledged input-output capabilities, unlike some resource-constrained Single-App platforms.

7.4.5 Access Revocation

It should always be possible for a user to revoke the access they have granted to an app, on a per-device basis. Similarly, it should be possible for a user to remove the *DenyListed* state for an application-device pair. This could be achieved in a similar manner to privacy controls on modern mobile and computer operating systems, where access to system resources are controlled on a per-application basis. Upon access revocation or state change, a command must be sent to the AAM, to notify the AAD to update the relevant record.

7.5 Requirements Analysis

In this section, we evaluate our proposed solution against the requirements we outlined in §7.2.

SecRQ1: Prevention of unauthorised access to BLE data The AAM intercepts and processes all GATT requests from all applications on the platform. As long as the assumptions stated in §7.2.1 hold, no application will be able to circumvent the AAM checks and covertly access data from BLE peer devices.

SecRQ2: Per-device access control AAM checks are performed per-application and per-device. An application that has been authorised to access data from one BLE device will fail AAM checks if it has not been granted access to a different BLE device.

SecRQ3: Access revocation Explicit mechanisms exist within the AAM (as discussed in §7.4.5) to revoke access for any application that has previously been granted GATT access to a BLE device.

SysRQ1: Protection by default Because our solution involves the modification of the BLE specification itself (rather than of a single device, platform or application), every platform that is qualified against this design will incorporate the GATT access control mechanism, ensuring protection by default across all (qualified) platforms.

SysRQ2: Backward compatibility All new functionality within our design occurs locally, within a single device. The functionality of GATT and lower layers of the BLE stack operate as they have previously, and interface with BLE peers with no changes. Therefore, a device that implements this solution will be backward compatible with all existing BLE systems. We demonstrate this with our POC in §7.7, where a modified Android stack operates with an unmodified BLE Peripheral. Note also that the changes described here do not affect the existing BLE services or profiles in any way.

SysRQ3: Minimal overhead for resource-constrained devices The processing described in §7.4.2 applies to Multi-App devices such as mobile phones, which are expected to have reasonably powerful operating systems and fewer restrictions in terms of battery usage. Most BLE Peripherals have small amounts of storage space, and usually do not support hosting multiple applications. Therefore, such devices will be defined as Single-App devices, and will be spared most of the processing overhead, as described in §7.4.3.

7.6 Additional Benefits

In this section, we describe advantages of our proposed solution, in addition to the fulfilment of the requirements.

No changes to existing BLE stack layers In our solution, a single new layer is added to the stack, and it is within this layer that the bulk of the access control behaviour is implemented. In particular, no modifications are required for any of the existing BLE stack layers, including the ATT/GATT layer, which is the only core layer that interfaces with the AAM. That is, the requests received by the GATT layer with the proposed new stack will be exactly the same as they are at present. This makes it easier for the Multi-App platform to implement the required changes in a modular fashion (assuming the existing stack has also been implemented in such a manner).

No changes to applications Our proposed solution requires no direct interaction between an application and the AAM. GATT requests issued by an application are forwarded *by the platform* to the AAM. This means that applications will not require any changes, apart from possibly to handle a new error status. This is a significant advantage, as there are several thousand mobile applications with BLE capabilities in existence today [66], and making changes to all of them would be extremely challenging, as it would require cooperation from a large number of developers.

Equal protection for all services A BLE device may implement services defined by the Bluetooth SIG (possibly via standard GATT profiles), but may also implement its own custom services. As with the example provided in Figure 5.1, most services and profiles defined by the SIG, including those that read user health data such as heart rate or glucose measurements, do not specify higher-layer protection as a security requirement. With our solution, protection is applied to both types of services, even if SIG-defined services do not specify authorisation permissions.

Protection even in the absence of pairing Many BLE Peripherals tend not to have sufficient input-output capabilities, and therefore either implement weak pairing or no pairing at all (as we show in Chapters 9 and 10). Our solution is separate from and at a higher layer to link layer protection mechanisms such as pairing. Any GATT request from an application has to first pass through the AAM before it can be forwarded via the link layer to the BLE peer, which means that protection is applied at a much earlier step. This means that the proposed new stack will protect data on a BLE device from access by an unauthorised application on a Multi-App platform even if the BLE device does not specify a requirement for pairing. Of course, if the BLE device does not require pairing, then its data can be eavesdropped over the wireless interface when it is in a connection; it can also be accessed by a different unauthorised device. However, that is outside the scope of this solution. Our solution focuses on protecting data from unauthorised access *at the application layer*.

Most changes are to mature platforms While a specification change would typically require a change to all devices that implement the BLE stack, the way in which the proposed change has been designed allows for the system to function even without any changes to existing Peripherals. The only entities that will *require* changes are Multi-App platforms such as mobile or personal computer operating systems. These platforms tend to have a robust update mechanism in place already, which is familiar to users. This ensures greater likelihood of the changes actually being installed on end user devices.

7.7 Proof of Concept

In order to demonstrate the viability of our proposed solution, we have implemented a Proof-of-Concept (POC) on the Android-x86 platform [30]. In this section, we discuss the implementation details, describe the test setup, and evaluate the POC.

7.7.1 Implementation Details

The Android platform was selected for our POC due to its open-source nature, large installation base and potential familiarity to readers. The Android-x86[2] code base was used, to be able to implement and test our solution on a virtual machine,[3] without the need for expensive device installations. The Nougat-r4 release of Android-x86 was the base upon which we built our POC, as it was found to have a stable implementation of Bluetooth. The modified Android-x86 was built on a VM running Ubuntu 18.04.3 LTS with 128GB RAM (with 8GB allocated for heap) and 8 cores.

Figure 7.2 depicts the components within the Android-x86 framework that were modified or added, in order to implement our proposed solution. Specifically, as mentioned in §6.2, an application GATT request on Android (such as those for reading or writing characteristics) must be preceded by the `connectGatt` call. Because of this construct, and due to the nature of the Android architecture, we select `connectGatt` as the entry point for the AAM checks. We use the device's hardware address as the DeviceID and extract the Android app's application ID (which uniquely identifies an application on the Android platform [149]) to use as the AppID.

The actual AAM functionality is implemented within the BT stack. In keeping with Android's workflow for other BLE functionality, we implemented the AAM functionality along a path from

[2]Android-x86 is a port of Android for x86 platforms. It is based on the Android Open Source Project (AOSP) with some modifications.

[3]Official Android emulators do not have Bluetooth capabilities, whereas Android-x86 does.

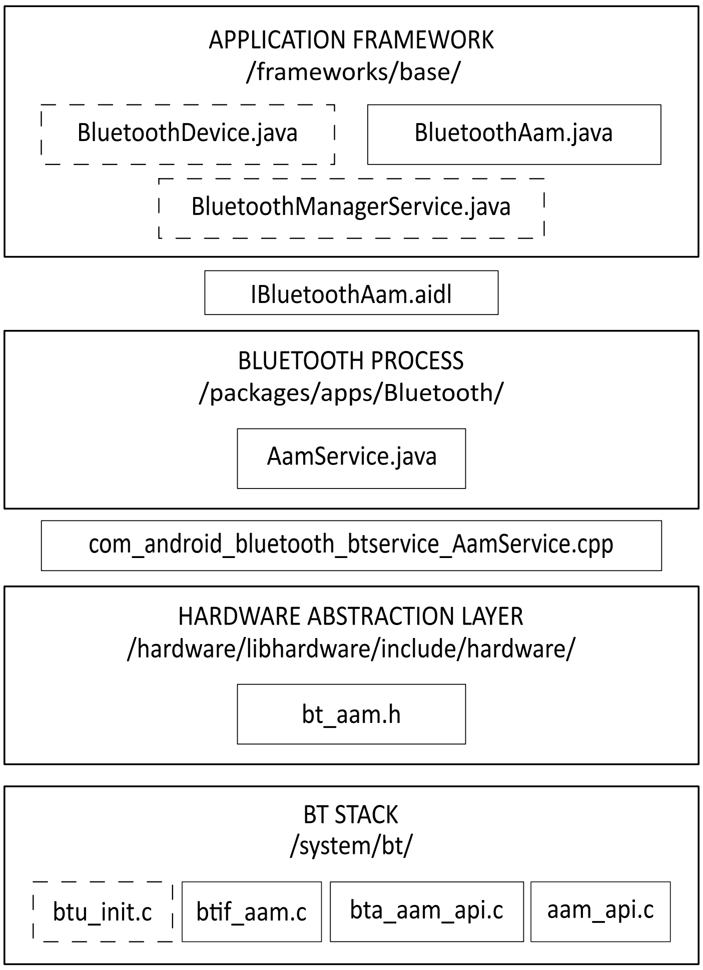
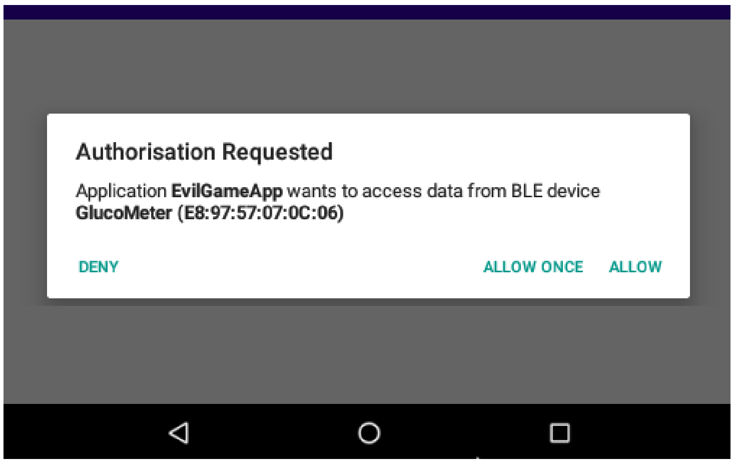


Figure 7.2: Modified Android-x86 platform, incorporating our solution. Utility functions are not included. Solid lines indicate new components, while dashed lines indicate modified components.

the application framework to a custom AAM "layer" within the stack, as shown in Figure 7.2. Within the AAM layer, the AAD is implemented as a linked-list of records, following the same structure that is used natively by Android for storing pairing credentials. Persistent storage to NVRAM is achieved by extending the existing BT interface storage component.

User authorisation is requested via standard Android dialog boxes. To make the contents of the dialog box clear and easily understood by the user, the application name is displayed instead of the AppID. For the device, both the device name (if available) and the hardware address are displayed, to avoid ambiguity in situations where more than one device with the same name are advertising in the vicinity. A sample is depicted in Figure 7.3. The `Allow` and `Deny` options within the dialog box map to the *Allowed* and *DenyListed* permissions described in §7.4.1, respectively. We have also implemented a temporary access option *AuthReqd*, which is displayed as `Allow Once` in the dialog box. This is similar to the *Maybe* partial-consent option in [150].



Authorisation Requested

Application **EvilGameApp** wants to access data from BLE device **GlucoMeter (E8:97:57:07:0C:06)**

Figure 7.3: User authorisation dialog with explicit reference to application and BLE device.

7.7.2 POC Tests

To test our POC, we replicate Attack 2, as described in §5.2, i.e., covert access of data from a connected BLE device by an unauthorised app. The attack involves four main components:

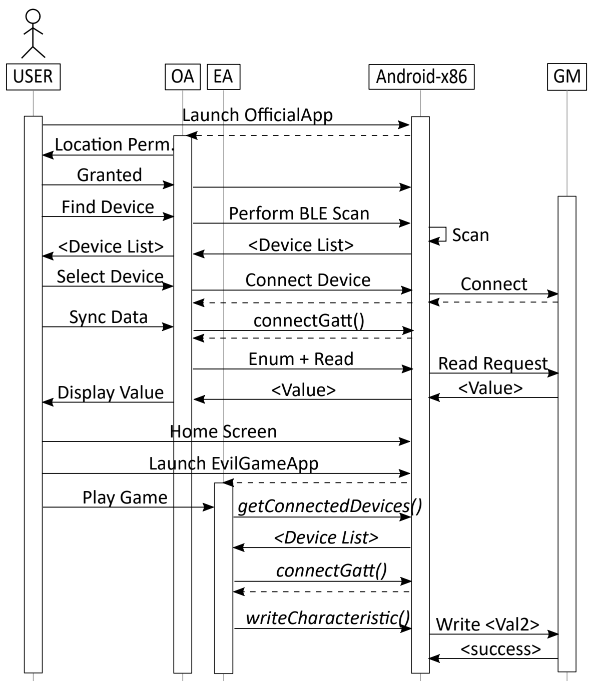
1. BLE-capable VM player, for running the original and modified Android-x86 builds. We used VMWare Workstation 14 Player on a Windows 10 laptop, with a CSR adapter.
2. Nordic nRF51 development kit, in the role of a glucose meter ("GlucoMeter"). No pairing-protected characteristics.
3. Android application, in the role of a glucose monitoring application ("OfficialApp").
4. Android application, in the role of a malicious application masquerading as a legitimate app, e.g., a game, which accesses BLE data covertly ("EvilGameApp").

We deploy a VM with the original Android-x86 build and perform the following:

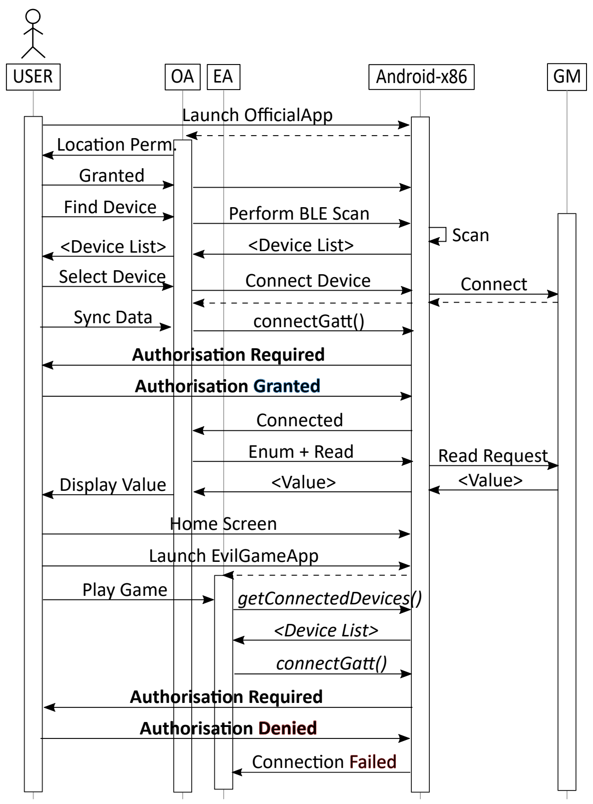
- Launch OfficialApp.
- Scan for BLE devices.
- Connect to the GATT server on the "GlucoMeter" and read a characteristic. This will read a dummy value of `0x12345678`.
- Launch EvilGameApp (which covertly identifies the existing connection to the GlucoMeter, calls `connectGatt` to it and writes the same characteristic).

Figure 7.4a depicts the interactions between five main entities (the user, OfficialApp, EvilGameApp, the Multi-App platform (i.e., Android-x86), and the BLE device) when going through the above test steps in the absence of any protection mechanism.

We then repeat the tests using the modified Android-x86 build. Figure 7.4b illustrates the interaction between the five entities when our controls have been implemented. The two figures demonstrate that unauthorised data access is prevented with our solution because the covert data access attempt is brought to the attention of the user and defeated. That is, data on the BLE peer is prevented from being accessed by unauthorised applications due to explicit user awareness and denial of authorisation.



(a) Current scenario.



(b) With our solution.

Figure 7.4: Interaction between User, OfficialApp (OA), EvilGameApp (EA), Multi-App platform (Android-x86) and BLE GlucoMeter (GM). Items in *italics* are interactions between EA and Android-x86 that occur without user awareness. Items in **bold** are new user interaction elements.

Testing with pairing-protected data We additionally modified the “GlucoMeter” to require pairing prior to data access. Re-running the tests again, we found that our controls worked in that scenario as well, as expected.

Testing with real-world devices and applications We verified the functionality of the POC implementation on real-world devices and applications by testing two popular fitness trackers, the Mi Band 2 and ID107 HR, against the modified Android-x86. For this, we installed the corresponding applications on the POC platform and connected to the devices. The solution worked without the need for any modifications to the fitness trackers or to their applications.

7.7.3 Evaluation

In this section, we present an evaluation of our POC in terms of development effort, performance overheads and user experience.

Development effort The entire set of modifications in our POC, including substantial debugging information, required fewer than 1500 new lines of code. This demonstrates that the solution is not only viable but also that it would not require significant effort to implement.

Performance overheads Analysing Android debug logs, we identified that performing an AAM access check took at most 25 milliseconds. This is well within the 100-millisecond instantaneous reaction perception limit [151,152]. We found while interacting with the system that this amount of time is indiscernible (from our point of view as a user).

User experience and comprehension The user is shown a dialog when an application is first launched and attempts to access data from a BLE device. Once that dialog has been responded to, subsequent access attempts don't require user interaction. Therefore, it is the impact of the first dialog that needs to be analysed.

Due to the prevailing COVID-19 situation, we were unable to conduct in-person tests. We therefore present here a theoretical analysis of the impact on user experience and comprehension.

If a malicious application professes to be benign but covertly accesses BLE data, then it may limit the number of permissions that it requests in order to trick the user into believing that it is harmless. In such a scenario, the presentation of a system dialog could serve to call the user's attention to the fact that covert data access is being attempted. Previous studies on user authorisation mechanisms, such as the ones used in the Android permission system, suggest that using a system dialog the first time a resource is accessed provides the optimal point for user decision making [153]. Our proposal effectively achieves this by raising the authorisation dialog the first time an application issues a `connectGatt` request for a BLE Peripheral. If access to the Peripheral is not in keeping with the purported functionality of the application, the user is likely to get suspicious and deny authorisation. Of course, there exists the possibility that the malicious application portrays itself as a BLE accessory application, possibly with numerous features encompassing a wide variety of Peripherals; this would make it more likely for the user to allow the application to access the Peripheral(s). Identifying malicious behaviour *after* access to the BLE Peripheral has been granted (e.g., leakage of BLE data once it has been read from a device) is outside the scope of this thesis.

7.8 Discussion

In this section, we discuss some limitations of our proposed solution and potential barriers for adoption, and also outline possible extensions to our work.

7.8.1 Limitations

Use of external sources of information The proposed solution requires that the implementing platform supply unique application identifiers to a component within the BLE stack. This removes the self-contained aspect of the stack by introducing an element external to the stack.

Reliance on honesty of platform While the proposed solution enables the user to grant permissions on a per-device, per-application basis, the fact that the access control checks are entirely performed by the Multi-App platform indicates that there is implicit reliance on the integrity and honesty of the platform. That is, there is an underlying assumption that the Multi-App platform will apply access control checks to all applications in an unbiased manner. However, it may be the case that a Multi-App platform that ships with its own set of applications automatically authorises those applications to access any BLE peer, and only applies access control checks to third-party applications. This would then remove some of the visibility and control from the user. Circumventing such an issue would require a complex protocol between the BLE device and companion application, and could be the subject of future work. For our work, we make the assumption of an honest and fair platform.

Complexities in desktop/laptop environments The solution we have proposed is straightforward to implement on mobile operating systems, where the level of user customisation, particularly regarding the BLE stack, is minimal. However, with operating systems such as Windows and Linux, there is a possibility that an application might use a BLE stack that is not provided by the OS. For example, we utilise a CSR adapter with the WinUSB drivers (installed using Zadig) on a Windows machine for a measurement study in Chapter 9. This is done manually, with external hardware, and requires that the system Bluetooth be turned off. While it is less likely that such a setup should exist to be exploited on normal user systems, it is still a consideration that should be factored in when implementing the solution.

7.8.2 Potential Barriers for Adoption

Our solution involves a modification to the BLE specification, which would then necessitate modifications to implementing devices in order to claim compliance with the specification. As per our analysis in §7.3, this has the greatest chance of ensuring adoption throughout the BLE ecosystem. However, there is a possibility of resistance from the Bluetooth SIG with regard to modifying the specification.

Despite BLE being a full-stack protocol, the core specification describes only the components below the application layer. This can lead to BLE being viewed as merely a transport-layer protocol, on top of which applications are developed, and may give rise to the view that application-level protection is the responsibility of either platform vendors or developers.

Further, the fact that the solution requires information external to the stack (as mentioned in §7.8.1), may also be viewed as a barrier to adoption, as the BLE stack has thus far not required any information from implementing platforms for its operation.

7.8.3 Potential for Extension

Reduction of covert fingerprinting by apps The ability to discover all services and characteristics on a BLE peer and thereby obtain a possible fingerprint for a device is a privacy concern that has been described in literature [46]. The Bluetooth specification [Version 5.2, Vol 3, Part G, Sections 4.4-4.6] states that such information about services and characteristics must be readable without authentication or authorisation, which means that a malicious application on a multi-application platform may be well placed to fingerprint the devices in the user's surroundings.

Because the AAM in our solution processes all ATT/GATT requests from applications, it can also intercept service discovery requests (the `Enum` requests in Figures 7.4a and 7.4b) and process them based on the requesting application's authorisation status. In this manner, the number of applications that may be able to perform covert fingerprinting can be greatly reduced. We highlight that, because of the construction of our POC in terms of intercepting `connectGatt`, which precedes service enumeration, prevention of fingerprinting by unauthorised applications is already accomplished.

Note that this does not violate the requirement in the specification in any way. The list of services and characteristics on the peer BLE device will be readable without authentication, including by the multi-application platform. However, this information will only be available to *applications* residing on the multi-application platform if they have been authorised by the user.

We also note that this does not defeat the fingerprinting as described in [46], as there the authors describe tracking a user by fingerprinting their BLE devices using some external device. However, our solution can prevent a malicious application on a user-owned Multi-App device from learning about the user via their different BLE devices.

Fine-grained access control Our solution and POC restrict application access to BLE data at the Peripheral level. That is, an application is either allowed to access all data on a Peripheral or none. This can be extended to enable fine-grained access control by access type (i.e., reads or writes) or even on a per-characteristic level (we note, however, that per-characteristic access control is inadvisable in most cases, as it would place too much decision burden on users who may not be aware of the purpose of individual characteristics).

7.9 Chapter Summary and Next Steps

In this chapter, we have presented a modified Bluetooth Low Energy stack to solve the unauthorised data access vulnerability on multi-application platforms. Our solution fulfils stringent security and system requirements, and takes into account practical considerations in its design. It ensures protection by default, while maintaining backward compatibility with existing systems. Concretely, no changes are required to applications or resource-constrained BLE Peripherals, nor are changes required to existing stack layers. We have also implemented a proof-of-concept on the Android-x86 platform to illustrate our solution and have demonstrated that the solution prevents unauthorised data access via explicit user awareness and authorisation.

This chapter concludes our discussion on the application-level unauthorised data access vulnerability. The next part of this thesis presents a set of measurement studies conducted against BLE-enabled applications, devices, and firmware, to better understand the prevalence of vulnerabilities within the BLE ecosystem and to aid in focusing research efforts.

Part III
Measurement of BLE Security and Privacy

8 Functionality Distribution and Impact Analysis via UUIDs

In this chapter, we analyse the utility of BLE UUIDs - extracted from companion mobile applications - to understand functionality distribution within the BLE ecosystem, identify vulnerabilities, and prioritise security analyses via functionality mapping.

8.1 Introduction

The presence of a vulnerability in a BLE system may not always be cause for concern. BLE devices can have a variety of functionalities, and the impact of exploiting a vulnerability will not be the same across all devices. For example, an unauthorised read of a thermostat will not have the same privacy implications as the unauthorised read of a glucometer. Writing to an attribute that controls the intensity of a smart bulb is in no way comparable to writing to an insulin pump. *Context* is therefore important.

With our mobile application analysis (Chapter 6), we found that the coarse-grained application categories utilised on app marketplaces, and the multi-functional nature of the apps themselves, meant that BLE-specific functionality was not easy to gauge. Therefore the impact of any identified BLE vulnerabilities may not be immediately apparent.

In this chapter, we explore the possibility of determining the functionality of BLE devices (that interface with mobile applications) using a novel source of information: Universally Unique Identifiers. UUIDs underpin data transactions in BLE and a single UUID will be associated with a single and specific piece of data. Where a BLE Peripheral implements a number of services, the set of UUIDs corresponding to the services and associated characteristics will be representative of the Peripheral's functionality. We utilise BLE-enabled Android APKs as our primary source for UUIDs (owing to the ease of obtaining and analysing APKs) and present a framework for the extraction, classification and functionality mapping of UUIDs. We further discover that some UUIDs can also provide useful indications regarding potential security issues.

We provide some background on the use of UUIDs within BLE in §8.2. We describe our UUID analysis framework, `BLE-GUUIDE` [31], in §8.3. The functionality distribution of BLE devices, as gleaned from performing functionality mapping against a dataset of 17,000+ Android APKs, is presented in §8.4. Observations are made regarding incorrect or anomalous use of UUIDs in §8.5. We discuss potential security considerations and describe security- and functionality-prioritised case studies in §8.6. Limitations of our technique are discussed in §8.7.

Related work Functionality mapping and the exploration of functionality distribution specifically within the BLE ecosystem have not been widely studied, although identification of device name and manufacturer (which could provide indications as to device functionality) from BLE advertisements has been explored in [69, 154]. However, device identification and classification for other types of IoT devices has been the subject of numerous analyses [155–159]. Most of these works employ machine learning techniques against network traffic. Some of the studies try to map traffic to a specific brand and model of device, while others classify the traffic at a higher level (e.g., as mobile phone, printer, etc.). These works are not directly applicable to BLE as they utilise TCP/IP traffic obtained by sniffing packets transmitted over Wi-Fi (or occasionally Ethernet). While BLE traffic can also potentially be sniffed over the wireless interface, it is somewhat more complicated to do so for multiple devices at a time due to the multiple channels and the frequency hopping employed by BLE.

The use of BLE UUIDs as a source of information has only recently begun to be explored. Zuo et al. [66] and Celosia et al. [46] explored the possibility of fingerprinting (and thereby tracking) BLE devices by using their UUIDs. To our knowledge, we are the first to use UUIDs as a data source for BLE functionality distribution analysis, or as indicators of vulnerabilities.

8.2 UUIDs as used in Bluetooth Low Energy

As described in §2.1.3.3, BLE organises and stores data in the form of attributes, where each attribute, i.e., each service, characteristic and descriptor, is identified by a UUID. The Bluetooth Special Interest Group has defined some standard UUIDs with specific meanings, which cover a variety of behaviours. For example, 0000180D-0000-1000-8000-00805F9B34FB is the SIG-defined UUID assigned to the Heart Rate service; 00002A19-0000-1000-8000-00805F9B34FB is the UUID assigned to the Battery Level characteristic. This means that these UUIDs can be tied to the defined behaviour. We refer to this type of UUID, i.e., one that has SIG-defined functionality, as an *adopted* UUID and the associated service/characteristic as an adopted service/characteristic in this thesis. Note that adopted services are, for the most part, optional and dependent on the use case of the BLE device. Two exceptions are the GATT Service and the GAP Service, which are mandatory [160]. With the remaining services and characteristics, there is no compulsion to use them, and if interoperability is not needed, then a developer may well choose not to do so. The Bluetooth specification allows for the creation of custom services and characteristics, where the developer has full control over the type and format of data. These services and characteristics will require *custom* UUIDs.

All UUIDs defined by the SIG are derived from the Base UUID 00000000-0000-1000-8000-00805F9B34FB [161]. The range of 2^{32} values, created by modifying the first 32 bits of the Base UUID (i.e., of the form XXXXXXXX-0000-1000-8000-00805F9B34FB), are reserved by the SIG. At present, all SIG-specified BLE UUIDs are defined by modifying only 16 bits of the Base UUID (of the form 0000XXXX-0000-1000-8000-00805F9B34FB) and are commonly referred to by only the 16-bit values. Therefore, the Heart Rate service UUID is often given as simply 0x180D and the Battery Level characteristic as 0x2A19.

UUIDs within the SIG-reserved range should not be defined by developers for their own use, although they can utilise the services and characteristics defined by the SIG [160]. That is, a custom UUID of the format XXXXXXXX-0000-1000-8000-00805F9B34FB should not be defined by developers. Any 128-bit value outside the Bluetooth SIG reserved range may be used by developers to create custom UUIDs for their own services and characteristics. To obtain a custom UUID *within* the reserved range, developers need to pay a fee to the SIG, which then assigns a member UUID [162].

8.3 A Framework for BLE Functionality and Security Measurement

Our goal is to obtain information regarding BLE-specific functionality from within APKs, such that the impact of BLE vulnerabilities can be determined and security analyses can be prioritised. We hypothesise that BLE UUIDs are a potential source of such information. However, an APK can have functionality apart from BLE interactions and even UUIDs can be used for non-BLE purposes within an APK. We therefore developed `BLE-GUUIDE`, a framework with specific mechanisms to extract and analyse only the information that is actually related to BLE.

`BLE-GUUIDE` comprises three main components:

1. **UUID Extractor & Classifier**: extracts BLE UUIDs from Android APKs and classifies them according to our custom categorisation.
2. **Functionality Mapper**: identifies BLE-relevant functionality within an APK.
3. **Security Analyser**: identifies vulnerabilities from raw UUIDs and performs an impact-centric prioritisation of security issues based on the outputs of the Functionality Mapper (also making use of findings and tools from Chapters 5 and 6).

§8.3.1- §8.3.3 describe the operation of each of these components in more detail.

8.3.1 UUID Extractor and Classifier

The functionality and security analyses performed by our framework both use BLE UUIDs as the starting point. We first extract UUIDs from an APK (§8.3.1.1) and then perform a high-level classification based on public knowledge of functionality (§8.3.1.2).

8.3.1.1 UUID Extraction*

In order to extract BLE-specific UUIDs from APKs, each APK within the dataset is analysed for standard API calls for BLE Peripheral interaction (e.g., `getService`, `getCharacteristic`, `setServiceUuid`). An adapted version of the tool `BLEScope` [66] is used to perform UUID extraction starting from such API calls. The final output is stored as a JSON file, containing the UUIDs as well as the method(s) within which the UUID was called, i.e., utilised.[1]

[1]This component was developed by collaborators. Sections within this chapter that are the contribution of co-authors are denoted with an asterisk in the section title (*).

8.3.1.2 UUID Classifier

We classify UUIDs into two broad categories: *Known Functionality UUIDs* or *Unknown Functionality UUIDs*, depending on whether or not the functionality provided by them is publicly known. For this, we utilise the Bluetooth SIG as our primary source of information, as the SIG describes all adopted service and characteristic UUIDs in terms of their functionality. We also use UUID information from BLE chipset manufacturers and device developers when they are uniquely defined and publicly documented.

Known Functionality UUIDs The Bluetooth SIG defines a number of adopted services and characteristics that can be used by device manufacturers to achieve functionality such as insulin delivery (0x183A) or user data gathering (0x181C), among others. In addition to this, many developers provide information within SDKs or other documentation about the custom services and characteristics used by their devices. We consider the UUIDs representing these services and characteristics as *Known Functionality UUIDs* or *KFUs*. That is, they have specific assigned meaning in the context of BLE that we can derive from publicly available information.

Our KFU database includes all the adopted services and characteristics defined by the Bluetooth SIG. We also include SIG-assigned member UUIDs (i.e., those assigned to members of the Bluetooth SIG upon payment of a fee), but only when unique functionality is associated with them. In addition, we include UUIDs that are uniquely defined and publicly documented by BLE chipset manufacturers or device developers. An example of these would be UUIDs defined by chipset vendors to enable Device Firmware Updates (DFU). For device developers, we focus on BLE devices that can be used without software modification by several applications. The most prominent example of this would be BLE beacons. We obtain such KFUs from manufacturer websites or, in a few cases, from developer sites (but only if multiple sites cite the UUIDs as belonging to the same device, and no sites assign different functionality to them).

Unknown Functionality UUIDs We consider UUIDs that are not classified as KFUs to be *Unknown Functionality UUIDs* or *UFUs*. UFUs are typically generated by BLE device developers when they are producing BLE devices and their corresponding applications. These UUIDs are expected to be randomly generated, to avoid collisions, and there is no formal or reliable source of information that is publicly available regarding them. Note that UFUs are always custom UUIDs, while KFUs can be adopted or custom. To put it another way, all adopted UUIDs are KFUs, whereas a custom UUID can be a KFU or a UFU depending on whether or not it has publicly-defined functionality.

Categorising BLE UUIDs The Classifier separates extracted UUIDs into KFUs and UFUs, where KFUs are used for deriving security implications, as described in §8.3.3.1, as well as for validating our framework before it is applied to UFUs.

- Medical
  - Measurement
  - Intervention
- Fitness
  - Activity Measurement
  - Body Metrics
- Device Mgmt
  - Device Info
  - DFU
  - Bootloader
  - Softdevice Mgmt
  - Battery
  - RSSI
  - Txpower
  - Connection Mgmt
  - Advertising
  - Scanning
  - Alerts
  - Factory Reset
  - Reboot
- Accessories
  - Audio Visual
  - Input Output
  - Gaming
  - Other
- Environment
  - Sensors
- Security
  - Access Control
  - Authentication
- Devices
  - Personal Comms
  - Office Mgmt
- Network
  - Mesh
  - Proxying
  - Configuration
- Communications
  - File Transfer
  - UART
  - Internet Protocol
  - SPP
- User Information
  - PII
- Smart Home
  - Device Mgmt
  - Environment Mgmt
- Transport
  - Personal Transport
  - Generic Transport
- Locnav
  - Positioning
  - Beacon

```
"network":{
  "mesh":{
    "mesh":{
      "blacklist":[
        "meshing"
      ],
      "meaning":[],
      "children":{}
    }
  },
  "proxying":{
    "proxying":{
      "blacklist":[],
      "meaning":[],
      "children":{}
    }
  },
  "configuration":{
    "router":{
      "blacklist":[],
      "meaning":[
        "router.n.02"
      ],
      "children":{}
    }
  }
}
```

Figure 8.1: Categories used for functionality mapping, with sample entry structure.

8.3.2 Functionality Mapper

The functionality mapper identifies the BLE-relevant functionality contained within an APK. This is thereafter used to derive a picture of the overall BLE functional landscape as well as to prioritise security analyses.

Building a database of functional categories In order to derive BLE functionality, we first build a database of possible functional categories, to be tested against. To build the database, we manually analysed several hundred APKs, their metadata, Google Play descriptions, and manufacturer websites, looking for the specific functionality of the BLE device. Thus, our categories represent the different functionalities that may be available on a BLE device. For each category, we provide a list of related words (e.g., microphone, camera, etc., for an `audio_visual` category). As a word can have several meanings (e.g., a "speaker" could be a loudspeaker or a speaker at a conference), we map each word to its corresponding WordNet definition [163].

The obtained functional categories are grouped according to their semantics, e.g., the high-level category *security* includes two sub-categories: *access control* and *authentication*. Overall, we have 13 categories and 40 sub-categories. Figure 8.1 depicts these categories and provides

```
.class La/b/c;
  .field HR_UUID:Ljava/util/UUID;
    ...

.class Lx/y/z;
    .field IRRELEVANT_FIELD:Lb/c/d;

    .method heartRateMeasuringMethod()V
        /* This method calls HR_UUID, so the method name, and strings & fields
           within the method, will be considered when mapping functionality for
           the HR_UUID */
        ...
        string "Read heart rate from device"
        iget La/b/c;->HR_UUID:Ljava/util/UUID;
        ...
    .end method

    .method irrelevantMethodInSameClass()V
        /* This method does not call HR_UUID, and will not be used for mapping its
             functionality */
        ...
        iget Lx/y/z;->IRRELEVANT_FIELD:Lb/c/d;
        ...
    .end method
```

Figure 8.2: Proximity-based approach of UUID Mapper.

the database structure for one category ("network"). The complete database structure, with descriptions, is provided in our code repository.

8.3.2.1 Mapping UUIDs to Functionality

We explore the possibility of using the method signatures, field names and logging strings defined within an APK - which may provide clues as to the APK's, but also in some cases specifically to a single UUID's, functionality - for our functionality mapping.

Our framework uses `Androguard` [111] to extract all strings, fields and method signatures from an APK. Each of these elements is individually tested against the functional category database, to obtain element-wise lists of functional category assignments, all of which are then combined to get a master list of possible categories. With method signatures, we found that the combination of the class and method names produced the most accurate results. For example, for a method with signature `Lcom/a/b/classA;->methodX(descriptor)`, we extract the `classA` and `methodX` components.

Using the entire set of elements found within an application would result in significant false positives. Our framework overcomes this by applying a proximity-based approach and considering only those methods that actually call the UUID. That is, it only considers the class and method names, as well as the fields/strings that are present within the method, for methods that actually *utilise* a UUID. This has been depicted with an artificial example in Figure 8.2.

Evaluation of the UUID Mapper KFUs have known functionality that we can easily ascertain. We were able to manually classify 376 KFUs against our keyword set, and these were used as ground truth against which to validate our framework. That is, we use the UUID Mapper to classify KFUs as if they were UFUs and then compare the obtained results against our manually annotated validation dataset. Note that the classification is on a per-UUID basis, not for a device as a whole, and therefore may not reflect the overall device functionality. For example, not all UUIDs used with a beacon device will necessarily have beacon-related functionality. Some may be related to signal strength, firmware updates, etc.

Given that three different sources of information (i.e., strings, fields and API method names) feed into the UUID Mapper, each of which will generate its own assignments, we opted to consider only those instances where the combined list consists of (possibly multiple instances of) a single unique category-subcategory. This technique provided an accuracy of 78% when compared to the alternative of taking a majority vote over all assigned category-subcategory pairs (which gave an accuracy of 74%). Here, accuracy was computed as $\frac{matches}{matches+mismatches}$.

Coverage obtained by UUID Mapper At a later stage (as described in §8.4.1.2), we observed that the UUID Mapper was able to map functionality for ∼18% of the available UFUs. This is a fairly low coverage and could have been due to a lack of logging or user interface strings, and also due to obfuscation techniques being applied to class, method and field names. Due to the poor coverage achieved via UUID mapping, we explore additional sources of information regarding BLE functionality: the SIG Product Database and Google Play. These are described in §8.3.2.2 and §8.3.2.3, respectively. They are described at a high level in this thesis, as they were not my contribution.

8.3.2.2 SIG Processor*

The Bluetooth SIG publishes details of qualified/declared components that incorporate the Bluetooth technology. If a specific product version is known, then searching for the product within the SIG Product Database would probably result in the most accurate description of BLE-specific functionality within the product (assuming such a description was provided). However, in the absence of product version information, we utilise application and library names.

For each APK under consideration, we compile the list of methods within which BLE UUIDs have been used. From these methods, we extract the main package name (i.e., first component after location domain names). For instance, for the call `com.exampleVendor.BLEManager.getHR()` we extract `exampleVendor`.[2] This is taken to be the developer or library-specific part of the package name, and is used to perform an automated search of the SIG Product Database.[3] If more than 20 items are returned, the term is considered to be too generic and is skipped. For each item returned, we get the product name and marketing description. If a functional category keyword is identified within the product name or description, we execute a Natural Language Processing (NLP) algorithm to identify the meaning of the word *within the context of the text*. If the meaning matches a WordNet definition within our database, we add the category to a list.

[2]This is the same technique as that employed in §6.6.2 and contains the same limitations.
[3]https://launchstudio.bluetooth.com/Listings/Search

Evaluation of the SIG Processor Because there is no ground truth available for SIG data, a manual approach was used for evaluation. We perform a manual verification of 50 libraries extracted from our dataset that had a single category mapping. For each of the cases, we inspect the SIG product results that the library produces and verify that the resulting categories map to the text describing the products (i.e., that they talk about the correct kind of BLE device). We obtained an accuracy of 78% for the SIG Processor via our manual evaluation.

8.3.2.3 Play Processor*

Descriptions on Google Play provide overall information about an APK's functions, including its BLE functionality, which can serve to augment the other data. We use Play descriptions as an input to our Functionality Mapper to determine the functionality of an application. For each APK, we download the Google Play description. After normalising, translating (if non-English), tokenising and stemming the app description, we iterate over it looking for appearances of the keywords defined in our functional category database. Such keywords, along with the app description, are processed using the NLP algorithm, similar to our approach for SIG categories.

Evaluation of the Play Processor Similar to SIG data, there is no ground truth when considering BLE-specific functionality within Play data either. We therefore manually analysed 100 randomly selected APKs, to verify that the classification offered by the Play Processor is correct. For this, we manually inspect the app description available in Google Play including any provided screenshots, and check the developer website for further information about the app. We glean the BLE-specific functionality from these sources and test it against the categories output by the Play Processor. This produced an accuracy of 62% across the 100 APKs. The lower accuracy obtained via Play descriptions is expected in some ways because, as we have observed before, the description will not focus solely on the BLE functionality, but will cover the functionality of the app as a whole.

8.3.2.4 Combined Functionality

We combine the information from UUIDs and the results from the NLP-processed SIG and Play descriptions to map APKs to BLE device functionalities. Our three methods work at different levels of granularity. The UUID Mapper assigns functionalities directly to UUIDs but the SIG and Play outputs are at the library/application level. Because of this, we only consider UUIDs that have been assigned a single functional category, but accept when the Google Play descriptions and the SIG product search return several functionalities (i.e., firmware update, heart rate measurement, notifications, etc.). Owing to the lower accuracy obtained from processing Google Play descriptions, we focus more on the results obtained from the other two sources.

8.3.3 Security Analyser

The Security Analyser component within our framework performs two types of security analysis/prioritisation: it identifies security indications from KFUs (§8.3.3.1), and prioritises security analyses for UFUs (§8.3.3.2).

8.3.3.1 Identification of Security Indications from KFUs

The very presence of certain UUIDs within a BLE device can sometimes be indicative of security vulnerabilities, when the functionality of the UUID is known. That is, some KFUs are associated with inherent vulnerabilities. The Security Analyser identifies such instances for two types of KFUs: adopted UUIDs and DFU UUIDs.

Adopted UUIDs As we have described in Chapter 5, the Bluetooth SIG defines a large number of services and characteristics, covering domains from environment to health and fitness. For all SIG-defined characteristics, including health and fitness, and excluding only those concerning insulin delivery, the maximum security mandated in the specifications at present is protection via the standard Bluetooth pairing mechanism. If the specification is adhered to, then protection at higher layers will not be implemented for these characteristics. However, we have shown that pairing/bonding alone is not sufficient protection when the data is accessed by apps on multi-application platforms such as Android if there is no application layer security. Therefore, by default, the vast majority of attributes with specification-compliant adopted UUIDs (specifically, characteristics) will be vulnerable to unauthorised data access.

Absent or insecure DFU A process for updating firmware is often necessary if bugs or security issues are discovered after a device has been released into the market. Some BLE chipsets allow for over-the-air firmware updates, i.e., Device Firmware Update (DFU). This process enables a BLE peripheral device to have its firmware modified by receiving updated firmware from a connected BLE application. However, if the update process itself is not secure, the BLE device would be vulnerable to unauthorised firmware modifications. Different chipset vendors implement different DFU procedures, some of which have security mechanisms built-in by default, some that require configuration by developers in order to be secure, and some that have no security options. Each of these DFU procedures use and therefore can be identified by a specific set of UUIDs. Table 8.1 lists the DFU UUIDs by chipset vendor, with an indication as to whether the procedure has security that is built-in, developer-dependent or unavailable. The presence of DFU UUIDs that are associated with processes that have known issues could indicate a vulnerability to malicious firmware updates.

8.3.3.2 Security Prioritisation of UFUs

While understanding security implications and their impact is fairly straightforward with KFUs, doing the same for UFUs requires greater effort and, in the general way, requires a case-by-case analysis. If no other information was provided, this would be a monumental task, given the potentially large number of custom UUIDs within applications. To focus our analysis, we first determine sensitive BLE data via functionality mapping (as described in §8.3.2). We then check, using our `BLECryptracer` tool (see Chapter 6), whether the BLE data has any app-layer security implemented. If no such protection is found, the sensitive BLE data is identified as being vulnerable to unauthorised reads and writes.

Table 8.1: Applications containing firmware update UUIDs.

Manufacturer	F/W Update UUID(s)	Secur.
Nordic Legacy [164]	0000153X-1212-EFDE-1523-785FEABCD123 (X=0-4)	×
Nordic Secure [145]	0000FE59-0000-1000-8000-00805F9B34FB	✓
	8E400001-F315-4F60-9FB8-838830DAEA50	✓
	8EC9000X-F315-4F60-9FB8-838830DAEA50 (X=1,2)	✓
	8EC90003-F315-4F60-9FB8-838830DAEA50	✓
	8EC90004-F315-4F60-9FB8-838830DAEA50	✓
Texas Instr. [165, 166]	F000FFXX-0451-4000-B000-000000000000	D
	(XX=C0,C1,C2,C3,C4,C5,D0,D1)	
Qualcomm [147, 167]	00001016-D102-11E1-9B23-00025B00A5A5	D
	0000110X-D102-11E1-9B23-00025B00A5A5 (X=0,1,2)	D
Silicon Labs [168]	1D14D6EE-FD63-4FA1- BFA4-8F47B42119F0	D
	F7BF3564-FB6D-4E53-88A4-5E37E0326063	D
	984227F3-34FC-4045-A5D0-2C581F81A153	D
	4F4A2368-8CCA-451E-BFFF-CF0E2EE23E9F	D
	4CC07BCF-0868-4B32-9DAD-BA4CC41E5316	D
	25F05C0A-E917-46E9-B2A5-AA2BE1245AFE	D
Cypress [146]	0006000X-F8CE-11E4-ABF4-0002A5D5C51B (X=0,1)	D
NXP [144]	003784CF-F7E3-55B4-6C4C-9FD140100A16	✓
	013784CF-F7E3-55B4-6C4C-9FD140100A16	✓
ST BlueNRG [169]	XXXXXXX0-8506-11E3-BAA7-0800200C9A66	D
	(XXXXXXX=8A97F7C,122E8CC,210F99F,2691AA8,2BDC576)	
ST STM32WB [170]	0000FE20-CC7A-482A-984A-7F2ED5B3E58F	✓
	0000FEYY-8E22-4541-9D4C-21EDAE82ED19 (YY=11,22,23,24)	✓

× = DFU with known security issues. D = DFU with developer-dependent security. ✓ = DFU with some security by default.

8.4 Large-Scale Functionality Measurement of the BLE Ecosystem

In this section, we present the results of applying BLE-GUUIDE against a dataset of 17,243 Android apps obtained from Google Play.[4] By executing the UUID Extractor against our dataset, we obtained 12,352 unique, valid UUIDs from 16,197 APKs (i.e., valid UUIDs could not be extracted from 1,046 APKs). Ultimately, 470 KFUs and 11,882 UFUs were obtained, with 5,735 APKs having only KFUs, 1,368 APKs having only UFUs, and 9,094 APKs having both.

We describe the functionality gleaned from the three data sources (UUIDs, SIG data and Play data) in §8.4.1 through §8.4.3. The combined BLE-relevant functionality for APKs is described in §8.4.4. We present additional observations regarding misuse of UUIDs in §8.5.

8.4.1 Functionality Mapping with UUID Data

Functionality derivation from UUIDs occurs directly for KFUs and via the UUID Mapper for UFUs. We first describe the two elements individually and then present a holistic view of the results of our functionality mapping in Table 8.3.

[4]This is a different dataset to that described in Chapter 6.

Table 8.2: Prevalence of adopted BLE services. ↓=Downloads in thousands.

Service	Apps	↓	Service	Apps	↓
Battery	1749	319015	Heart Rate Measure.	1672	173644
Device Information	1561	298719	GAP	898	127936
Common*	608	132132	Immediate Alert	371	24847
GATT	292	23979	Blood Pressure	181	115825
Glucose	158	5787	Link Loss	158	17162
Current Time	151	105716	Cycl. Speed&Cadence	141	25657
Body Composition	113	1561	Run. Speed&Cadence	110	5155
Health Therm.	100	2682	Tx Power	93	106507
Weight Scale	90	103237	Pulse Oximeter	69	720
User Data	63	811	Alert Notification	61	3516
Environ. Sensing	56	196	Cycling Power	54	11582
Transport Discovery	25	742	Cont. Glucose Mon.	23	30
Fitness Machine	10	691	HID	10	101
Location and Nav.	10	211	IP Support	7	17
Automation IO	6	7	Phone Alert Status	6	17
Scan Parameters	6	101011	Object Transfer Service	4	1
Indoor Positioning	3	0	Mesh Provisioning	3	1
Mesh Proxy	3	1	Next DST Change	3	10
Ref. Time Update	3	5	Bond Management	2	0
HTTP Proxy	1	0	Insulin Delivery	1	1
Unassigned**	105	3349			

**UUIDs that have been assigned to multiple services. **UUIDs that have been defined but apparently not assigned to any service.*

8.4.1.1 KFU Categorisation Results

We describe some of our observations regarding the most prevalent KFUs in this section.

Adopted UUIDs Adopted UUIDs were the most prevalent of the KFUs extracted from our dataset. This is unsurprising given that they make up a significant proportion of KFUs overall. We found that 12,289 of the 16,197 APKs contain adopted UUIDs. Table 8.2 presents the number of APKs that contain adopted UUIDs, grouped according to the services defined by the SIG, along with their cumulative download counts. The table shows that 3 of the top 10 most prevalent UUIDs are related to user health and fitness, of which heart rate-related services are the most common. This may reflect one of the most popular consumer applications of BLE - fitness trackers - which typically measure a user's heart rate. In fact, the distribution of these UUIDs across the various categories could provide some insights into the distribution of services across the general BLE eco-system. To gauge the *popularity* of the various applications, we analyse approximate download counts from Google Play, which show that 3 of the top 10 most downloaded (adopted) services are also related to user health.

Beacons Coming second to adopted UUIDs in prevalence were beacon UUIDs (10% of all tested APKs contained UUIDs with beacon functionality). As we have mentioned before, beacons are BLE-enabled transmitters that operate predominantly via advertisements. These advertisements contain a UUID, which a nearby listener (such as a mobile phone) can pick up, which normally triggers some location-based services. The prevalence of beacon UUIDs, with a cumulative download count of over 28 million for the containing apps, shows the popularity and potential of proximity-based marketing and services.

DFU A small percentage of APKs contained references to known DFU UUIDs. As these may have security implications, we discuss the exact ramifications of this, as well as of the individual UUIDs that were identified, in §8.6.1.

8.4.1.2 UFU Categorisation Results

We applied the UUID Mapper to the 11,882 UFUs extracted from our APK dataset, considering only unique UFU-methods pairs. This produced 25,157 UFU-methods pairs, out of which the UUID Mapper assigned a single unique category to 4,622 UFU-methods pairs. Examining this on a per-UFU basis, a single unique category-subcategory was assigned for 2,174 UFUs from our set of 12,195 UFUs (∼18% coverage).

A note on localised strings The original version of our framework did not use localised strings for UUID functionality mapping. A custom taint analysis script was used at a later stage to extract localised text from APKs. However, we found that very few APKs contained localised strings *within the BLE data access methods* (only 172 APKs contained such strings). A manual analysis revealed that most of the strings did not provide information regarding potential BLE functionality. Localised strings from only 50 APKs revealed useful information. Further, because class/method names were already used in functionality mapping, we found that only 47 methods (and the UUIDs therein) would have had functionality mapped *solely* due to localised strings, of which only 32 were assigned a *unique* category-subcategory.

8.4.2 Functionality Mapping with SIG Data*

The SIG functionality mapping component produced a very limited number of matches (22% in terms of apps but only 6.3% in terms of downloads). A manual inspection of the results showed that many of the companies that were being queried had incomplete information in the SIG product database about their products, e.g., only the codename for the device with no further information about it. Also, the type of names used in some of the cases made it difficult to map some of the libraries to the actual developers. As an example, we extracted *shenzhen* as one of our possible library names. However, Shenzhen is a well known location of chip manufacturers and a search in the Bluetooth SIG database reveals 1,210 companies with *Shenzhen* in their name. Most of the functionality matches we *were* able to achieve using the SIG Processor were related to location and fitness.

8.4.3 Functionality Mapping with Play Data*

Using our NLP processor over Google Play descriptions we were able to extract functional categories for 11,734 apps, accounting for 97.1% of the overall downloads. This makes the extraction of functionality via Play descriptions the method with the most coverage. However, as observed in §8.3.2.3, this method is also likely to be less accurate than the other two.

We observed that most of the apps (that were assigned a category) were assigned three functional categories when using information from Google Play. This is expected because, as we observed earlier, a BLE device may have different services each with different functionality and the Play description may indicate all such functionalities. For example, in the case of wearable devices, the Play descriptions will map to both Medical and Fitness categories because of their access to heart rate and activity data.

8.4.4 APK Categorisation Results*

We produce the final BLE-relevant functionality categorisation of an application by joining together the results produced by each of the sources (UUIDs, Google Play and SIG database). Note that the presence of a KFU automatically assigns its functionality to its app, while the presence of a UFU assigns its functionality if and only if all copies of the UFU found within the app are assigned to the same category across the whole dataset of UUIDs. Using all three information sources, we were able to assign BLE-specific functionality for 87.7% of the analysed apps. Our results are shown in Table 8.3. From these, we make the following observations:

Fitness and Medical are not the most popular BLE devices While there are more apps for medical and fitness related functionalities than any other category, the two most popular categories in terms of *downloads* are audio-visual and positioning. Devices within audio-visual include cameras, microphones but also speakers. For cameras, BLE is normally used to enable remote control of the device. For audio systems it is normally used as a wake up mechanism, but not to transmit music. The possibility for transmitting audio over BLE been added very recently into the Bluetooth specification as LE Audio [171] and we expect this to grow in the future.

Bluetooth is widely used for location-related tasks During our initial dataset filtering, we identified 50k+ apps that scanned for BLE advertisements but didn't connect to them. This, in addition to the fact that 16% of app downloads have some location-related functionality, point to widespread use of BLE as an additional method to incorporate location-tracking capabilities within applications. This is likely due to the very fine-grained indoor location tracking that BLE enables, via beacon technology.

Developers define custom UUIDs even when SIG-defined UUIDs exist Many applications use attributes with custom UUIDs even for functionalities that have already been defined by the official BLE specification. Although this means that the BLE devices will only able to function with their own application (meaning the user has less flexibility in their choice of apps), this may help protect against cross-application attacks, such as those described in Chapter 5, if application-layer security is applied to such attributes.

Table 8.3: Results of the functional categorisation of applications within the BLE ecosystem. $\downarrow$ = Downloads in millions

		Known Funct.UUIDs				Play Processor				SIG Processor				UUID Mapper				Total			
Category	**Subcategory**	**Apps**	%	$\downarrow$	%	**Apps**	%	$\downarrow$	%	**Apps**	%	$\downarrow$	%	**Apps**	%	$\downarrow$	%	**Apps**	%	$\downarrow$	%
Medical	Measurement	1915	8.2	198.3	7.1	4287	16.6	420.2	4.2	287	6.4	8.2	1.7	272	3.4	22.1	1.8	5031	9.0	541.2	3.9
	Intervention	1	0.0	0.0	0.0	39	0.2	4.6	0.0	7	0.2	0.0	0.0	0	0.0	0.0	0.0	47	0.1	4.6	0.0
Fitness	Activity measmnt.	198	0.8	36.1	1.3	4487	17.4	1302.6	13.0	393	8.8	136.0	28.3	499	6.3	243.3	19.8	5059	9.1	1623.6	11.8
	Body metrics	147	0.6	105.8	3.8	1549	6.0	87.4	0.9	14	0.3	10.8	2.2	33	0.4	0.8	0.1	1643	2.9	190.3	1.4
Security	Access control	52	0.2	0.1	0.0	726	2.8	40.8	0.4	115	2.6	10.4	2.2	63	0.8	3.7	0.3	944	1.7	54.4	0.4
	Authentication	2390	10.2	41.1	1.5	426	1.7	54.1	0.5	17	0.4	0.1	0.0	314	4.0	25.6	2.1	3103	5.6	109.3	0.8
Location	Positioning	13	0.1	0.2	0.0	2193	8.5	2139.2	21.3	2418	53.9	27.8	5.8	5	0.1	0.0	0.0	4285	7.7	2159.6	15.7
	Beacon	2382	10.2	28.0	1.0	371	1.4	11.0	0.1	20	0.4	0.1	0.0	1721	21.7	26.3	2.1	2849	5.1	43.0	0.3
Device mgmt.	Device info	4693	20.1	346.1	12.4	230	0.9	143.3	1.4	2	0.0	0.1	0.0	150	1.9	27.2	2.2	4936	8.9	498.7	3.6
	DFU	604	2.6	136.6	4.9	1420	5.5	247.7	2.5	9	0.2	0.2	0.0	2831	35.7	183.8	15.0	4480	8.0	531.8	3.9
	Bootloader	0	0.0	0.0	0.0	6	0.0	0.6	0.0	0	0.0	0.0	0.0	14	0.2	0.3	0.0	20	0.0	0.9	0.0
	Softdevice	0	0.0	0.0	0.0	2	0.0	0.6	0.0	7	0.2	10.6	2.2	0	0.0	0.0	0.0	9	0.0	11.2	0.1
	Battery	4125	17.7	354.8	12.7	965	3.7	161.2	1.6	1	0.0	0.0	0.0	399	5.0	83.9	6.8	5153	9.2	521.3	3.8
	RSSI	3	0.0	0.0	0.0	28	0.1	1.6	0.0	0	0.0	0.0	0.0	58	0.7	1.3	0.1	88	0.2	2.9	0.0
	TX Power	2509	10.7	135.6	4.9	13	0.1	0.0	0.0	8	0.2	0.0	0.0	15	0.2	1.0	0.1	2543	4.6	136.7	1.0
	Connection mgmt.	84	0.4	11.2	0.4	307	1.2	231.5	2.3	0	0.0	0.0	0.0	505	6.4	353.5	28.8	876	1.6	493.5	3.6
	Advertising	2408	10.3	28.0	1.0	7	0.0	0.1	0.0	0	0.0	0.0	0.0	0	0.0	0.0	0.0	2411	4.3	28.0	0.2
	Scanning	6	0.0	101.0	3.6	3	0.0	0.0	0.0	1	0.0	0.0	0.0	0	0.0	0.0	0.0	8	0.0	101.0	0.7
	Alerts	467	2.0	126.5	4.5	1221	4.7	111.7	1.1	16	0.4	1.6	0.3	87	1.1	12.5	1.0	1638	2.9	237.4	1.7
	Factory reset	65	0.3	0.1	0.0	9	0.0	1.0	0.0	0	0.0	0.0	0.0	0	0.0	0.0	0.0	74	0.1	1.1	0.0
	Reboot	1	0.0	0.0	0.0	10	0.0	0.1	0.0	5	0.1	0.1	0.0	15	0.2	1.0	0.1	31	0.1	1.2	0.0
Transport	Personal	0	0.0	0.0	0.0	82	0.3	2.8	0.0	0	0.0	0.0	0.0	0	0.0	0.0	0.0	82	0.1	2.8	0.0
	Generic	0	0.0	0.0	0.0	1023	4.0	90.6	0.9	33	0.7	0.1	0.0	29	0.4	1.8	0.1	1050	1.9	90.7	0.7
Smart home	Device mgmt.	0	0.0	0.0	0.0	827	3.2	199.3	2.0	39	0.9	4.0	0.8	205	2.6	23.2	1.9	1051	1.9	202.8	1.5
	Environment mgmt.	0	0.0	0.0	0.0	94	0.4	117.7	1.2	110	2.5	1.1	0.2	37	0.5	0.3	0.0	233	0.4	119.0	0.9
Environment	Sensors	118	0.5	10.5	0.4	1219	4.7	37.5	0.4	82	1.8	101.8	21.2	86	1.1	102.4	8.3	1458	2.6	252.2	1.8
Devices	Office mgmt.	0	0.0	0.0	0.0	122	0.5	41.9	0.4	15	0.3	12.1	2.5	143	1.8	0.9	0.1	265	0.5	53.8	0.4
Accessories	Audio visual	0	0.0	0.0	0.0	2682	10.4	2697.9	26.9	483	10.8	138.7	28.9	58	0.7	1.5	0.1	3081	5.5	2706.7	19.7
	I/O	10	0.0	0.1	0.0	116	0.5	138.0	1.4	22	0.5	0.1	0.0	7	0.1	0.1	0.0	152	0.3	138.3	1.0
	Gaming	0	0.0	0.0	0.0	679	2.6	1439.3	14.3	5	0.1	0.0	0.0	2	0.0	0.1	0.0	683	1.2	1439.3	10.5
	Other	0	0.0	0.0	0.0	391	1.5	166.2	1.7	258	5.8	15.7	3.3	58	0.7	102.4	8.3	691	1.2	182.0	1.3
Network	Mesh	3	0.0	0.0	0.0	91	0.4	3.8	0.0	78	1.7	0.5	0.1	100	1.3	0.4	0.0	231	0.4	4.6	0.0
	Configuration	0	0.0	0.0	0.0	40	0.2	24.8	0.2	0	0.0	0.0	0.0	0	0.0	0.0	0.0	40	0.1	24.8	0.2
Comms.	File transfer	4	0.0	0.0	0.0	9	0.0	120.0	1.2	0	0.0	0.0	0.0	1	0.0	1.0	0.1	14	0.0	121.0	0.9
	UART	1098	4.7	1126.2	40.4	27	0.1	0.9	0.0	20	0.4	0.1	0.0	205	2.6	8.4	0.7	1312	2.4	1134.9	8.2
	Internet protocol	7	0.0	0.0	0.0	0	0.0	0.0	0.0	0	0.0	0.0	0.0	0	0.0	0.0	0.0	7	0.0	0.0	0.0
	SPP	13	0.1	0.0	0.0	19	0.1	0.2	0.0	20	0.4	0.1	0.0	27	0.3	0.2	0.0	79	0.1	0.5	0.0
User info	PII	55	0.2	0.8	0.0	54	0.2	2.8	0.0	0	0.0	0.0	0.0	1	0.0	0.0	0.0	101	0.2	3.6	0.0

8.5 Observations Regarding UUID Usage

We make some observations regarding developers' use of UUIDs within their BLE devices, focusing on incorrect or anomalous use.

8.5.1 Incorrect Use of SIG-Reserved Range

As mentioned in §8.1, the range of 2^{32} UUIDs formed by modifying the first 32 bits of the Bluetooth Base UUID are reserved by the Bluetooth Special Interest Group. However, we found that 13% of UFUs, i.e., custom UUIDs, were actually formed using this base UUID. While such UUIDs don't have a functionality assigned to them (by the Bluetooth SIG) at present, the use of them in applications runs the risk of future conflicts.

8.5.2 Anomalies

Adopted UUIDs typically have clear meaning and functionality assigned to them. We made use of the defined functionality for adopted UUIDs to create a mapping between the UUIDs and the Google Play categories that they could be expected to fall under. Although Google Play categories don't provide fine-grained details about an application's BLE functionality, they do provide a very high-level functional overview, which can be useful in determining obvious anomalies. For example, a Heart Rate Measurement UUID may be expected to be used in a Medical, Sports, or Health & Fitness application. However, inclusion of this UUID within a Finance application would be highly incongruous.

We applied the mapping to the applications in our dataset, to identify possible anomalies between the inclusion of an adopted UUID within an Android application and the functionality of the application as indicated by its Google Play category. 10,556 applications within our dataset made use of at least one non-GATT/GAP adopted UUID as well as having a presence on Play at the time of testing. Only these were therefore used for anomaly detection.

From our mapping, we identified 333 instances of incongruous use of adopted UUIDs. From these, we separated out 123 applications that used UUIDs belonging to health-related BLE services. Using additional information from Play, we prioritised these APKs as follows:

(a) **Suspicious** - No obvious need for the UUID (taking Play description into consideration).
(b) **Somewhat suspicious** - There may be a use case for the UUID, but the app description doesn't explicitly state it.
(c) **Very likely benign** - Based on UUID and Play categories, as well as Play description,[5] there appears to be a legitimate use for the UUID.

Within the second category there were instances where adopted UUIDs were used for slightly different reasons than expected from the specification, e.g., Health Thermometer UUIDs to measure temperatures in industrial applications.

[5] We also used screenshots available in Play for this verification.

We manually analysed 95 APKs which were considered to be *suspicious*. The purpose of our analysis was to determine whether the apps were making use of their BLE access capabilities to access sensitive data, when their functionality clearly didn't require it.

Fifteen of the 95 APKs defined UUIDs that were not used anywhere within the app. In all such cases, the UUIDs were defined in a library, rather than in the core application. This would account for why a UUID might be present but not used, as libraries may define more functionality than is actually used by the calling application.

There was one APK that incorporated the nRF Toolbox by Nordic Semiconductor [172], which demonstrates a large number of BLE profiles. The APK itself apparently only used the DFU functionality within nRF Toolbox, but started the Heart Rate Measurement function because the code for doing so was present within nRF Toolbox.

We found one instance of a Blood Pressure Sensor service being defined within the code of an application that did not perform health functions. A closer analysis of this APK revealed that this particular BLE-specific code was being reused from another app by the same developer that actually did connect to a blood pressure monitor.

There were a number of instances of incorrect UUID usage, i.e., UUIDs used for purposes other than for what they were defined. In particular, 4 APKs from one developer denoted the Heart Rate Service as “RX Service”; one APK used it to store door lock parameters; and one APK used various adopted UUIDs to control a barometer.

There was also an unusual combination of UUIDs being used in a set of 42 applications which all enabled the control of music streaming to speakers. The applications used the service UUID 0x180A (Device Information) in conjunction with the Heart Rate Measurement characteristic. Given that this combination of service and characteristic does not conform to the BLE service specification, it is likely that this is due to a developer error.

One APK included functionality to interface with a popular fitness tracker, but with no indication of doing so within the application description. Whenever a developer intends for their application to interface with any sort of user device, particularly one that handles sensitive user data, some mention should be made within the app description.

Finally, there were ∼10 APKs where health-related services appeared to be defined within the application's own code, for no apparent reason and where the developer did not have other health-related apps on Play. It should be noted that around half of them defined the Heart Rate Service, which is the BLE service most commonly used in coding examples. Therefore, it is possible to assume that the UUIDs for the service were copied from online sources unintentionally [173]. We emulated a BLE peripheral with the services declared in each APK, to check if the application would attempt to surreptitiously connect to it, but found that none did. This finding supports our assumption of accidental use of adopted UUIDs.

Table 8.4: Firmware update UUIDs.

Manufacturer	#Apps	Manufacturer	#Apps	Manufacturer	#Apps
Nordic (Legacy)	291	Nordic (Secure)	94*	Texas Instr.	175
Qualcomm	39	Silicon Labs	37	Cypress	31
NXP	22	ST (BlueNRG)	8	ST (STM32WB)	0

**Partial overlap with Nordic Legacy.*

8.6 Security Analysis

In this section, we discuss security considerations derived from the extracted UUIDs, in accordance with §8.3.3.

8.6.1 Security Indications from KFUs

Adopted UUIDs As mentioned previously, of the 16,197 APKs for which at least one valid UUID was identified, we found that around 70% expose services defined by the Bluetooth SIG (excluding the GATT/GAP services, which are always included and therefore do not offer much insight into functionality). In addition, 1,457 APKs use *only* Bluetooth-defined services (i.e., no custom services and at least one non-GAP/GATT service). As we have mentioned before, BLE data within adopted UUIDs, if adhering to specification-compliant absence of higher-layer protections, will be vulnerable to access by unauthorised apps.

This observation is particularly concerning when the data in question is of a sensitive nature. For example, Table 8.2 shows that, of the 7,077 APKs that reference BLE UUIDs (excluding GATT/GAP/common/unassigned),[6] over 25% are concerned with user health data such as glucose level, blood pressure and heart rate measurements. If this information is combined with other data such as activity levels, then a malicious application could derive a complete health and fitness profile for a user. In addition to general privacy concerns, this data could also be exploited, unbeknownst to the user, by insurance agencies and other interested parties.

Absent or insecure DFU We found that, out of the 16,197 APKs in our dataset that returned at least one UUID during the extraction phase, 603 APKs contained references to DFU UUIDs. This implies that, for the remaining devices, even if a bug or security issue is identified, there will be no easy means by which to update the device. While other mechanisms of firmware update are possible, they will typically involve greater levels of user intervention and may therefore be neglected.

Table 8.4 summarises the number of APKs containing DFU UUIDs from various chipset manufacturers. It shows that the Nordic Legacy DFU UUID is the most prevalent among the DFU UUIDs. This particular DFU mechanism does not test the source of the firmware or the identity of the communicating BLE device, and is therefore vulnerable to malicious firmware overwrites by unauthorised entities. Nordic has since provided a new, more secure mechanism, using signed

[6]A single application may use multiple adopted UUIDs.

firmware. However, as the table shows, the proportion of devices using the new mechanism is much lower. Given that the SDK containing the Secure DFU functionality also contains the Legacy UUIDs, the number of APKs that *only* support the insecure firmware update mechanism was found to be 207. Obtaining download counts from Google Play for 170 of these APKs (the rest were no longer on Play), we found that this corresponded to at least 109 million downloads. This means that there are potentially over 100 million BLE devices that are vulnerable to unauthorised firmware updates. Forty of the 207 APKs contained at least one health or fitness-related adopted UUID. Malicious firmware modifications on such devices could enable incorrect health information being fed to the user.

Of the remaining DFU UUIDs identified by our framework, those corresponding to Nordic Secure, NXP and STM32WB have security mechanisms built into the DFU process. The rest have security options that require enabling by the developer, which runs the risk of those security mechanisms not being implemented.

8.6.2 Security- and Functionality-Prioritised UFUs

Within APKs that contained UFUs, `BLECryptracer` identified 8,593 APKs with no protection for BLE reads and 10,420 APKs with no protections for BLE writes. These were prioritised for manual analysis using the results from the Functionality Mapper component of `BLE-GUUIDE`. The APKs with the most sensitive BLE functionality were identified, with particular focus on functionality that relates to user health or personal data, or which has security consequences. We next present a selection of case studies based on our analysis of such APKs.

Case study - ECG applications: Two ECG measurement applications within the dataset read data from external ECG recording devices and display the results in the app with no protection between the two endpoints. This means that while the "official" app is reading information from the ECG recorder, so too can any other app on the same Android device (as we have described in Chapter 5). We have informed the developers of both applications of this vulnerability, but received no response from either.

Case study - user PII: The Functionality Mapper returned two UFUs that were mapped to the sub-category PII (Personally Identifiable Information). Both belonged to the same APK: a proximity-based "friend-finder". Manually analysing the app, we found code suggesting that the app advertises the user's first name, device address, and an ID within BLE advertisements, and scans for such advertisements to identify other app users in the area. The app also maintains a *Last Seen* parameter for each user it identifies, which can facilitate unauthorised tracking of users. When we installed the app, we found that it was in demo mode, and the data-collection functionality was not being executed. At the time of conducting this analysis, we hypothesised that the mere presence of code within the application for advertising user PII, along with burgeoning interest in user/device trackers [174], signalled an increase in the future of such tracking or "finding" apps. This hypothesis is now validated by the increasing use of contact-tracing applications for COVID-19.

Case study - door lock: From the results obtained using our framework, we identified an APK that interfaced with a BLE-enabled door lock in an insecure manner. Specifically, the application code logs the start of an authentication sequence, and the data that is read from the BLE lock is sent to a decryption method. However, the decryption code revealed that it did not employ any standardised algorithm, but rather a custom scheme.

Analysing the custom algorithm, we found that it comprises an array of fixed bytes, to which another array of bytes and a string are used as inputs. The authentication sequence begins with reading a value from a specific BLE characteristic. The read value (in bytes) is the first argument to the decryption algorithm. The key/PIN to the door lock is the second argument. This means that an attacker in the vicinity can eavesdrop on the initial authentication exchange (i.e., the challenge issued by the door lock and the reply from the app) and brute-force the PIN *offline*. That is, the attacker can submit the challenge and every possible value of the PIN to the decryption algorithm and identify the PIN that results in the correct response. The PIN can be 4, 5 or 6 digits long for this particular door lock, which means the keyspace is smaller than 3.5 million and offline brute-forcing can be performed within a reasonable timeframe. Once the correct PIN has been identified, the attacker can trigger the authentication protocol with the actual door lock and complete the authentication sequence with the identified PIN.

In general, the use of non-standard cryptographic algorithms is discouraged as their security has likely not been verified by a community of experts. Since a BLE lock may be used to secure a home or building, standardised strong cryptography is imperative. We have notified the developer about this issue, but have not received any response.

8.7 Limitations

UUID extraction UUIDs are sometimes generated over multiple iterations, which makes it difficult to extract them without complex static or dynamic analyses. This means that there is a possibility that we may not have obtained complete coverage of all UUIDs used by an APK. In addition, as discussed in §8.5.2, sometimes only a small subset of the UUIDs defined within an APK may actually be used. The extraction mechanism may not always capture this scenario, and may therefore extract even those UUIDs that are not used.

Native code We conducted the APK string/field extraction (§8.3.2.1) and localisation analysis (§8.4.1.2) using `Androguard` over smali code, i.e., analysis of Native code was not performed. There could therefore be instances of UUIDs whose descriptions are contained within native code that our analysis does not capture.

Functionality mapping The richness and accuracy of information obtained through the various sources (i.e., API, fields, strings, Play and SIG descriptions) depends entirely on the BLE device/app developer actually incorporating or publishing such information in the first place. If a developer chose to include no strings within an app, to obfuscate methods and fields, and also to publish vague or incomplete descriptions on Google Play and even when validating their device on SIG, then the amount of information we will be able to retrieve will be negligible. In addition, if a single method called all UUIDs, then they would all be assigned exactly the same categories, even if their functionality differed.

8.8 Chapter Summary

We have presented a framework for measuring functionality distribution within the BLE ecosystem and for determining the impact of a BLE vulnerability by performing functionality mapping for the BLE device. We have evaluated a novel source of information for this purpose: the Universally Unique Identifiers used to identify each data value on a BLE device. Our UUID functionality mapping resulted in a reasonably accurate estimate of device functionality but produced poor coverage. Augmenting this data with information from the Bluetooth SIG and Google Play improved the coverage slightly (where the SIG data produced greater accuracy but very poor coverage, while Play resulted in high coverage but lower accuracy). However, we have arrived at the conclusion that functionality mapping using such techniques leaves the analysis at the mercy of the developer, i.e., it is fully dependent on the amount of information provided by the developer, which may be insufficient in many cases.

We have also utilised UUIDs to identify the presence of certain types of vulnerabilities within BLE devices, and - making use of the functionality mapping - prioritised and presented case studies where our framework identified poorly-protected sensitive data.

In the next chapter, we present a measurement technique for identifying minimum access requirements for BLE data by interacting with physical devices.

9 Device Security Measurements

In this chapter, we describe a technique and Node.js tool that we have developed for ascertaining the minimum access requirements on a per-characteristic basis for data on BLE devices. We analyse our results in light of later research and present limitations of our technique.

9.1 Introduction

Of the different types of attributes within a BLE device (as described in §2.1.3.3), characteristic value attributes are the most interesting from a security standpoint. These hold the data value of interest, which could be user health measurements, activity levels or values controlling a device's functions (such as in the case of door locks or eScooters). Access to these characteristics can be restricted via attribute permissions (see §2.2.4). Of the different types of attribute permissions, *access*, *authentication* and *encryption* permissions are handled by the BLE stack, while *authorisation* permissions are implementation-specific.

A device may apply no protection to its characteristics or it may apply protection to only some of its characteristics. For example, a fitness tracker can specify a requirement for an authenticated/encrypted link prior to allowing access to some of its characteristics (e.g., heart rate measurement values). When doing so, it will specify the requirement in terms of the security modes and levels described in §2.2.3. A device (e.g., a mobile phone) that attempts to access the data over an unauthenticated/unencrypted link will be presented with an `Insufficient Authentication/Encryption` error. An `Insufficient Authentication` error normally triggers pairing between the two devices. However, it should be noted that the error will not indicate the level of protection that is required. Therefore, if the strength of pairing is insufficient, then data access requests will receive `Insufficient Authentication` errors even after pairing and link encryption, and the devices will have to re-pair (assuming both devices are capable of stronger pairing).

The pairing "generation" (i.e., LE Legacy or LESC) and the association model together play a large part in determining the strength of the pairing process and therefore the strength of the generated encryption key. The pairing generation and association model that are ultimately used are decided by the features that the devices exchange during Phase 1 of the pairing process (see §2.2.5). The association model in particular is strongly linked to the IO capabilities indicated by the two devices. When two honest devices undergo pairing, they will specify their true security and IO capabilities, which will result in the strongest pairing association model that their combined capabilities allow. An attacker, who is not bound by such principles, can specify the *lowest* possible IO capabilities when attempting to pair with a victim BLE device, to force the pairing to the least secure association model, i.e., *Just Works*. If the permissions applied to the victim device's characteristics are strong, e.g., Mode 1 Level 3 or Mode 1 Level 4, then regardless of the fact that weak pairing has taken place, the attacker will not be able to access the victim device's data. If, however, the applied permissions are not strong enough, e.g., Mode 1 Level 2, then the attacker will be able to read and potentially manipulate the data on the victim device (subject to characteristic properties).

In addition, Haataja et al. [175] and Antonioli et al. [52] pointed out a weakness of Bluetooth pairing, in the form of key entropy downgrade. That is, the Bluetooth standard allows for the entropy of the pairing and session keys to be reduced from the default entropy of 128 bits to as low as 56 bits for BLE (and 8 bits for Bluetooth Classic). It is up to individual implementations to check the entropy of the key and refuse a pairing request if the key size is insufficient via an `Insufficient Encryption Key Size` error. A low-entropy key would make it far easier for an attacker to brute-force, and is therefore another aspect of interest when testing devices.

In this chapter, we describe a mechanism by which we determine the *minimum* possible level of security at which data on a BLE device can be accessed, including checks for acceptance of low entropy keys, through direct device interaction. Specifically, we exploit the existing algorithm for determining the pairing association model (§9.2) to define an algorithm for incremental access checking (§9.3). We implement our algorithm as a Node.js tool (§9.4), and test it against real-world devices (§9.5). We also utilise device-specific analyses to discuss limitations of a generic testing mechanism (§9.6).

Related work Security and privacy testing of physical BLE devices has been conducted in a number of previous works. Such works tend to fall into one of two categories: analysis of a particular class of device (e.g., wearables or medical devices), or tests for a particular type of vulnerability. Examples for the first category of studies include security analyses of BLE locks [7], eScooters [8, 176] and fitness trackers [9, 23, 177, 178]. Some of these analyses have identified severe safety-related vulnerabilities in the respective devices, including the ability to overwrite the firmware in wearables, unlock smart locks without authentication and take control of eScooters with the potential to cause injury to the user.

Analyses that fall under the second category explore one aspect (or occasionally more, if they are related to one another) of BLE security or privacy. For example, Das et al. [9] and Fawaz et al. [10] tested for the presence and correct implementation of resolvable private addresses within BLE Peripherals by monitoring the device address within BLE advertisements over a period of time. Antonioli et al. [52] described tests for low-entropy key negotiation vulnerabilities against real-world devices. Key and pairing downgrade attacks against devices were explored in [47] as well. Our work also falls into the second category, in that we test for insufficient protection of BLE data. To our knowledge, we are the first to explore minimum access requirements on a per-characteristic basis for BLE data.

Algorithm 9.1: Pseudocode for determining association model during pairing.

Data: OOB flags, MitM flags, IO capabilities, pairingGen (pairing generation)
Result: Set association model (assocModel)

```
if pairingGen == LESC then
    if at least one device has OOB flag set then
        assocModel = OOB;
    else
        CheckMitm();
    end
else
    if both devices have OOB flag set then
        assocModel = OOB;
    else
        CheckMitm();
    end
end
Function CheckMitm():
    if neither device has MitM flag set then
        assocModel = Just Works;
    else
        CheckIO();
    end
end
Function CheckIO():
    if (pairingGen is LESC) and (both devices have DisplayYesNo or Keyboard+Display) then
        assocModel = Numeric Comparison;
    else if (one device has Display) and (other device has Keyboard) then
        assocModel = Passkey Entry;
    else
        assocModel = Just Works;
    end
end
```

9.2 Determining the Pairing Association Model

As mentioned previously, when two devices undergo pairing, the association model that is to be used for key generation will be determined based on the features that are exchanged during Phase 1 of pairing.

Algorithm 9.1 presents pseudocode for the process (as defined in the Bluetooth specification [Version 5.2, Vol 3, Part H, Section 2.3.5.1]) to determine the association model that will be used during Phase 2 of pairing. As can be seen from the pseudocode, unless Out Of Band data is used, the association model is determined by the IO capabilities of the communicating devices. While a peer device or MitM attacker can manipulate the exchanged features in order to downgrade the pairing to the weakest association model (*Just Works*), a BLE device can refuse to pair with a peer device that does not have sufficient IO capabilities. However, many real-world devices may not do so.

Algorithm 9.2: Incremental access checking for BLE characteristics.

Data: Device address

Result: Per-characteristic security levels.

```
secLevels ⟵ [0, ..., n];
smpObjects ⟵<list of command packets with incrementally strong pairing configurations,
  corresponding to secLevels >;
currSecLevel ⟵ min(secLevels);
accessTypes ⟵ [read, write, notify];
connect to device and enumerate services + characteristics;
get characteristic properties;
characteristicList ⟵ (characteristics, properties) ;
outputObject ⟵ (characteristics, properties);
repeat
    for accessType ∈ accessTypes do
        for characteristic ∈ characteristicList do
            if accessType is applicable to characteristic then
                attempt characteristic access;
                if access succeeds then
                    update outputObject with currSecLevel for characteristic and accessType ;
                    remove characteristic − accessType from characteristicList;
                end
            end
        end
    end
    if characteristicList is empty then
        break;
    end
    while currSecLevel < max(secLevels) do
        currSecLevel + +;
        pairingObject ⟵ smpObjects[currSecLevel];
        pair with pairingObject;
        if pairing succeeds then
            break;
        end
    end
until characteristicList is empty or currSecLevel == max(secLevels);
```

9.3 Incremental Access Checking

We begin with the assumption that there likely exist BLE devices which, while possibly undergoing pairing as part of normal operations, might have characteristics that are freely readable and writable, i.e., no authentication or authorisation required. The same devices may also have characteristics that *do* require an encrypted link prior to access but which may be accessible after pairing using the fairly insecure *Just Works* pairing model. Some other devices may require *Passkey Entry* pairing prior to allowing access to any characteristics. We define Algorithm 9.2 to determine the lowest possible protection level at which each individual characteristic on a BLE device can be accessed.

The algorithm outlines the enumeration of services and characteristics on a BLE Peripheral, followed by determining appropriate access types (read, write, etc.) for each characteristic, based on the characteristic's properties. The core of the algorithm is the incremental access component (line 9 onward). Characteristic access attempts are made at increasingly high security levels, and within each security level, different types of access are attempted, as applicable to the characteristic (i.e., determined by the characteristic's properties). When a specific type of access for a characteristic is successful, that characteristic-access pair is removed from further consideration. This process continues until all the characteristics have been accessed or the highest level of security has been reached.

9.4 Implementation

We have implemented Algorithm 9.2 as a Node.js tool, `ATT-Profiler` [32], to profile BLE Peripheral devices. The tool is written on top of `noble` [179], which is an open-source implementation of a BLE Central device. We modify `noble` to capture and manipulate pairing-related events, to enable custom pairing requests and Central behaviour.

`ATT-Profiler` scans for and connects to a user-specified BLE Peripheral device. It then attempts to access (i.e., read, write, or subscribe to) every applicable characteristic, where a characteristic is applicable if the access type is present in its properties set. That is, `ATT-Profiler` will attempt to read a characteristic if the characteristic has the `read` property and will attempt to subscribe to the characteristic if the characteristic has the `notify` property, etc. If access is denied because the characteristic is protected, then the tool will attempt to pair with the test Peripheral using the lowest level of security. If pairing fails, then the tool will disconnect and reconnect to the test Peripheral, increment the security level and re-attempt pairing. If pairing is successful, then it will re-attempt characteristic access. If the characteristic still cannot be accessed, another reconnection is performed, the security level is incremented again and another pairing request is sent. We enable successive pairing attempts in this manner by disabling bonding, i.e., by clearing the bonding flags in our pairing requests.[1]

In our implementation, we use four security levels with different configurations: None - No pairing; Low - Pairing attempt specifying no MitM protection requirement, no input-output capability and 64-bit key (more accurately, a key with 64 bits of entropy); Medium - Pairing attempt with no MitM protection requirement, no input-output capability and 128-bit key; High - Pairing attempt with MitM protection requirement, Keyboard+Display capability and 128-bit key.[2] Given these features, we would expect pairing attempts at levels Low and Medium to result in the *Just Works* association model and level High to result in *Passkey Entry* (assuming the target Peripheral has a display). We perform STK masking to enable encryption using the correct key in scenarios where the key entropy has been reduced (i.e., at level Low).[3]

[1]In some cases, when bonding is enabled, the target device will reject pairing attempts that follow a successful pairing because it will have stored bond information for the successful pairing. Additional steps would need to be taken in order to clear this information from the device. Since we do not require the preservation of bond information for our tests, we instead set the bonding flags to `0`. This feature was not present in the original version of our code, which resulted in some incorrect outputs in [35].

[2]Additional levels are possible by introducing additional pairing configurations. However, these four levels are sufficient to identify the main concerns of weak pairing and acceptance of low key entropy.

[3]The key entropy downgrade checks were present in the original version of our code [32] and manuscript [35]. However, STK masking had not been performed in the initial version of the code, which meant that the pairing process failed when issuing pairing requests at level Low.

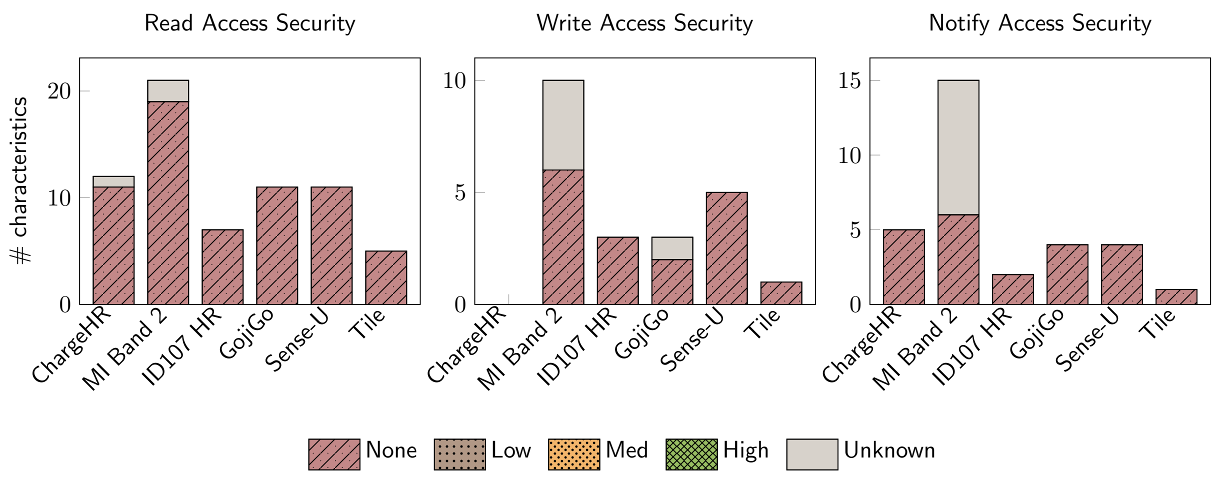


Figure 9.1: Security levels for read/write access in real-world devices.

Identification of static passkeys via dictionary attacks Some BLE devices utilise fixed passkeys (i.e., PINs made up of six decimal digits) with the Passkey Entry model. This somewhat defeats the purpose of passkeys since a known fixed passkey can be entered programmatically, with no need for the user intervention that is required with dynamically generated passkeys. We implement a brute-forcing component within `ATT-Profiler` to identify whether a BLE device uses a static passkey, by making repeated pairing attempts using a dictionary of commonly used passkeys, derived in part from an analysis of six-character passwords [180].

9.5 Real-World Device Testing

Tests were conducted against four fitness trackers of different prices and capabilities (Fitbit ChargeHR, Mi Band 2, ID107 HR, and GojiGo), a baby monitoring device (Sense-U), and an asset tracker (Tile). All devices were under our ownership and control.

The number of characteristics on each device accessible at each security level has been graphed in Figure 9.1. The level "Unknown" on the graph refers to when a characteristic was not accessible, even though its properties indicated that it should be.

The graphs show that four out of the six devices allowed all applicable characteristics to be read, four allowed all applicable characteristics to be subscribed to, and three also allowed all applicable characteristics to be written, without any authentication required. Here, 'applicable' refers to the presence of the relevant property (`read`, `notify` or `write`). These devices could therefore be vulnerable to unauthorised data read/writes and perhaps also MitM attacks. However, we see in the following discussion that there may be other factors to consider.

9.5.1 Per-Device Analysis

We present a per-device analysis of the characteristics that returned a result of Unknown or that returned incongruous results. This analysis feeds into our discussion on limitations of physical device testing (§9.6).

Fitbit ChargeHR The Fitbit ChargeHR had 12 characteristics with the `read` property, and 11 of these were freely readable. One characteristic returned `Insufficient Encryption` despite successfully pairing at level High. However, we observed that, while the `ATT-Profiler` sent a Pairing Request indicating Keyboard+Display capabilities for level High, the ChargeHR returned a Pairing Response indicating No Input Output capabilities (despite having a display). The same observation was made even when the ChargeHR paired with the official Fitbit mobile application. Another unusual finding was that the ChargeHR appeared to have no characteristics with the `write` property. This seemed unlikely since the device supports OTA firmware upgrades, which would require writing to a characteristic. Tracing over the BLE wireless interface while interacting with the ChargeHR via the official Fitbit application, we found that the ChargeHR has a writable characteristic that cannot be enumerated (i.e., is initially not visible in the list of services and characteristics) until a different characteristic has notifications enabled.

Mi Band 2 For the Mi Band 2, twenty-one characteristics had the `read` property, and all but two had no protection applied to them. The two that could not be accessed returned `Read Not Permitted` errors. Similarly, 8 of 15 characteristics that had the `notify` property returned `Write Not Permitted` errors for access attempts (recall from §2.1.3.3 that we need to *write* to a characteristic's CCCD in order to enable notifications). Further, one characteristic that had the `write` property also returned `Write Not Permitted`. Three other writable characteristics returned `Application Error 0x80` when write access attempts were made.

Analysing the errors individually, an `Application Error` is an obvious indication of application-layer restrictions. However, `Read` or `Write Not Permitted` errors for characteristics that have the `read` or `write` properties set is incongruous. Exploring BLE chipset vendor forums, we find a possible explanation for this behaviour. For example, with Nordic chipsets, when a characteristic has authorisation requirements, a security check is performed every time an access attempt is made for the characteristic. The type of check is developer-dependent; it could be a cryptographic authentication sequence, but could simply be that a specific character or character sequence is expected. If the check fails, the device can return a custom response, including `Read Not Permitted` or `Insufficient Authentication`, despite these errors not being inline with the specification's definitions. We believe that the Mi Band 2 may have implemented similar behaviour.

This surmise is validated by findings we made during later analyses. In particular, examination of the application code for the Mi Band 2 mobile app showed that access to the Heart Rate Measurement characteristic was "locked" and could be "unlocked" by writing a specific byte sequence to one of the freely writable characteristics (see §5.3.3).

ID107 HR The ID107 HR fitness tracker allowed all visible characteristics to be read, written and subscribed to without any security requirements. Tracing the message exchanges when the device communicated with its official mobile application, we found that a Nordic firmware update characteristic was made visible when the device was placed in Direct Firmware Update mode by writing to a specific characteristic.

GojiGo The GojiGo Activity Tracker had three characteristics with the `write` property, one of which was related to the Nordic DFU process. When a write attempt was made to the Nordic DFU characteristic, the device stopped responding to further requests. However, when the Nordic characteristic was removed from the tests, the remaining characteristics were found to be freely writable.

Tile The Tile asset tracker apparently had a single writable characteristic. However, this was found to be a SIG-defined Device Name characteristic. We knew from the product description that other configurations were also possible, which meant that at least one other characteristic with the `write` property had to be present on the device. Tracing over the BLE interface while interacting with the official Tile application, we found that the Tile, similar to the Fitbit, has writable characteristics that are hidden until a different characteristic's CCCD is written to.

9.5.2 Observations on Low-Entropy Keys

None of the devices, save for the ChargeHR, required pairing. When testing the ChargeHR, we found that it accepted the pairing request at security level Low, i.e., with a *Maximum encryption key size* value of 64 bits. This indicates that it is vulnerable to key downgrade attacks [52]. In addition, we modified `ATT-Profiler` to send pairing requests even when pairing was not required, to identify the prevalence of the key downgrade vulnerability. We found that the Mi Band, GojiGo and Sense-U baby monitor all successfully paired when requesting 64-bit key entropy. The Tile refused the pairing request with `Pairing Not Supported`; the same was returned even when requesting full-length keys. The ID107 HR appeared to complete pairing successfully, but then disconnected for no apparent reason. It did the same when requesting 16 byte key entropy. Observing message exchanges via traces when the device interacted with its official app, we found that pairing was not used during its normal operations.

9.6 Limitations and Future Work

Limitations of physical device testing Performing security tests with physical devices has many advantages. When testing individual devices manually, we are able to trace device communications over the wireless interface, explore its interactions with any companion application, and fuzz for device behaviour when provided with unexpected values. However, as we found from our tests of real-world devices, many devices exhibit custom, specification-deviant behaviour, which may require per-device analysis. This leads to a loss of generalisability and hampers *automated* bulk testing of devices. Further, conducting this type of testing at scale is also hindered by the cost of procuring devices.

Validation of SIG-defined services The `ATT-Profiler` enumerates and tests access for all visible services on a BLE Peripheral. Some or all of these services may be SIG-defined, and will therefore have predefined structures and security requirements. One possible option for extending `ATT-Profiler` would be to validate the obtained service structures against those defined by the SIG, to ensure that they match the SIG definitions. We have not implemented this functionality at present owing to the relatively small collection of devices that makes up our test set, and the lack of diversity of device types and of BLE services within the collection, which would not result in meaningful outcomes.

9.7 Chapter Summary and Next Steps

In this chapter, we have described a mechanism and purpose-built tool for determining the minimum access requirements for every characteristic on a BLE Peripheral. We have found through real-world device tests that many BLE Peripherals do not appear to protect their data sufficiently. However, we have also found through per-device analysis that devices may actually incorporate hidden and possibly protected characteristics that are not accessible via a generalised tool. In addition, bulk testing of physical devices suffers from the obvious drawback of the expense involved in purchasing the devices. We therefore explore an alternative source of security-relevant information in Chapter 10.

10 Firmware Analysis

In this chapter, we describe a mechanism for extracting security-relevant configuration data from a rich information source: device firmware. Our analysis technique is not confined to BLE firmware, but instead is applicable to generic ARM Cortex-M architectures, which are gaining popularity among IoT devices. We outline our approach to overcoming the challenges inherent to the analysis of stripped ARM Cortex-M binaries, and present case studies for extracting API arguments from BLE binaries that target Nordic Semiconductor and STMicroelectronics chipsets. Our results reveal widespread lack of security and privacy controls in real-world BLE devices.

10.1 Introduction

Numerous flaws have been uncovered in IoT devices in recent years, some of which have been exploited at a large-scale (e.g., Mirai [181–183] and Brickerbot [184]). Severe vulnerabilities have also been discovered in certain cardiac devices [76], baby heart monitors [77] and webcams [75]. The root cause on many occasions was poor device configuration, e.g., default passwords [185, 186] or poor protection for data [187–189].

The configuration of an IoT device can therefore be a vital source of information regarding possible vulnerabilities. While this could be obtained from the device itself, physical device testing has several disadvantages, particularly in terms of difficulty in automating large-scale analyses and the cost of procuring devices (as we observed in Chapter 9). The *firmware* running on the device is often a good alternative, as it generally reflects the device's configuration and functionality exactly. While firmware files were originally difficult to acquire, the advent of OTA firmware upgrade mechanisms in IoT devices has led to several vendors hosting firmware files on their websites or including them within mobile applications. However, even if firmware binaries are now easier to source, there are other challenges. The analysis of binary files is not straightforward in general, and IoT firmware tends to be made available as *stripped* binaries (without headers, symbol tables or section information), which further complicates analysis. With devices such as IoT hubs and gateways (e.g., mobile phones, routers), which often run some version of the Linux OS, familiar filesystem structures and commands may be identifiable within firmware, which can contribute towards the analysis. Even so, the analysis will generally not be straightforward. Analysis is much more complex for IoT peripheral/node devices (e.g., BLE Peripherals), which may feature custom operating systems, or sometimes have no operating system at all. This has resulted in far fewer analyses of IoT peripheral binaries.

In this chapter, we focus on peripheral firmware analysis. We do not limit our analysis to solely BLE binaries, but consider any ARM Cortex-M firmware, as this is a processor family that is gaining popularity with resource-constrained IoT devices. We outline the challenges involved in the analysis of stripped binaries, particularly in the case of IoT node devices (§10.2). We describe the design of our firmware analysis framework, `argXtract` [33], and detail how it overcomes each of the aforementioned challenges (§10.3). We implement and evaluate our framework against ground truth (§10.4) and present two case studies: the extraction of security and privacy-relevant configuration information from stripped Nordic (§10.5) and STMicroelectronics (§10.6) BLE binaries. §10.7 discusses limitations of our technique and potential avenues for future work.

Related work Various aspects regarding the analysis of firmware binaries and configuration security have been explored in previous studies. While it may seem like most aspects of firmware analysis have already been covered by previous studies, most such studies have focused on Linux-based systems [190]. We observe that the analysis of stripped binaries targeting non-traditional operating systems and the ARM Thumb instruction set, which is increasingly favoured by IoT peripherals and which is the focus of our analysis, has still not been explored sufficiently.

Analysis of stripped binaries: The analysis of stripped binaries, particularly function boundary identification, has been the subject of widespread study. Control flow analysis has been used in [191–195] to determine functions in PE, ELF, COFF and XCOFF binaries, and a QEMU+LLVM approach for function boundary identification was presented in [196]. These approaches may not be suited to ARM IoT analysis due to errors introduced by inline data and compiler-introduced constructs such as Thumb switch-case conditions. Machine Learning (ML) has also been proposed for identifying function entry points [197–199], but this approach requires a sufficiently large labelled training set, which is currently not available for IoT peripheral binaries. A semantics-based approach was used in Jima [200] for ELF x86/x86-64, which employs techniques for computing jump tables that are similar to those used in `argXtract` for computing table branch addresses. To the best of our knowledge, we are the first to employ the techniques we describe in this chapter for identifying the application code base,[1] the `.data` segment, as well as several sources of inline data. The inline data identification employed by `argXtract` also improves the performance of function identification (as we show in §10.4) and subsequent tracing.[2]

Function matching and labelling: Previous works have employed various strategies to achieve function pattern matching or similarity computations (primarily for non-ARM binaries). One approach for function pattern matching is to compute statistical similarities between instruction sequences of functions [201, 202], but this may suffer poor performance due to compiler-introduced variations and optimisations [203]. Dynamic similarity testing via function execution was employed in [204]. While this is in some ways similar to our approach, `argXtract` looks for functional *equivalence* based on *known* function behaviour,[3] while [204] considers function *simi-*

[1]Similar to our approach, Wen et al. [92] also uses vector table entries as one input to compute the application code base, but without considering default handlers as we do.

[2]This could be because the techniques we have developed are mainly applicable to stripped Cortex-M binaries, which have not been sufficiently studied by the research community.

[3]Our technique, while not seen in other works, is only applicable to functions that result in clearly distinguishable artefacts within memory or registers.

larity based on *random* executions. Most current approaches favour ML techniques [199,205–207] but, as mentioned previously, this requires sufficiently large training sets.

Security analysis and patching of IoT firmware: A number of previous works have described techniques for assessing different aspects of security for IoT devices via static or dynamic firmware analysis. Large-scale security analyses of embedded firmware files, predominantly Linux and VxWorks-based, were presented in [208, 209]. FIE [210], built from the KLEE symbolic execution engine, identifies vulnerabilities in embedded MSP430 firmware. Firmalice [211] detects authentication bypass vulnerabilities within the firmware of Linux and VxWorks-based binaries. FirmFuzz [212] specifically targets IoT firmware and is intended for security analysis. It uses QEMU and targets unstripped Linux-based binaries. These works analyse binaries that target at least pared-down versions of fully-fledged operating systems. They would not be suitable for analysing stripped firmware of embedded devices that do *not* have a proper operating system. `InternalBlue` [101] enables testing and patching of Broadcom Bluetooth firmware, while `LightBlue` [213] analyses and performs debloating of unneeded Bluetooth profiles and HCI commands within firmware to reduce the potential attack surface. The randomness of RNGs used in Bluetooth chipsets was measured via firmware analysis in [214].

BLE security analysis: On the BLE front, previous works (including our own) have explored the security and privacy configurations and behaviour of BLE peripherals by analysing devices [9, 10, 23, 47, 52, 177, 178], and mobile apps [34, 66]. However, as mentioned in §9.6, device analysis is expensive and may not be generalisable for bulk analysis, while mobile applications generally don't provide insights about low-level pairing mechanisms.[4]

Independently to us, Wen et al. [92] developed a tool named `FirmXRay`, built on top of `Ghidra`, that identifies BLE link layer configuration vulnerabilities by targeting supervisor calls on Nordic and ICalls on Texas Instruments BLE binaries. To compare `argXtract` and `FirmXRay`, we executed them against a random subset of 300+ binaries from the `FirmXRay` dataset. We found that a direct comparison was not possible due to insufficient information within `FirmXRay`'s output data structures. In general, while `FirmXRay` is geared towards BLE vulnerabilities, our work is capable of handling generic analysis of any technology that targets ARM Cortex-M binaries. Further, `FirmXRay` only handles specific supervisor calls and ICalls, whereas `argXtract` performs function pattern matching to identify *any* function (provided the requisite artefacts can be identified within memory/registers). The template-based approach used in our framework also enables easy addition of new test functions, and has greater format parsing capabilities. Within the BLE analysis, Wen et al. [92] have confined the discussion to link layer vulnerabilities, while we discuss application-layer issues as well.

[4]Some studies use the presence of pairing-related API calls within companion mobile applications as the only indication of pairing protection on the associated BLE devices [54,66]. However, we observe that mobile operating systems automatically handle pairing when relevant attribute access errors are encountered, even in the absence of such API calls. Therefore, the absence of pairing-related API calls within mobile application code does not automatically mean that the BLE device applies no protection for its data.

```
BLE_GAP_OPT_PASSKEY = 34;
uint8_t passkey[] = "123456";
ble_opt_t ble_opt;
ble_opt.gap_opt.passkey.p_passkey = &passkey[0];
err_code = sd_ble_opt_set(BLE_GAP_OPT_PASSKEY, &ble_opt);
```

(a) Source C code.

```
1eaba: 4ab8      ldr  r2, [pc,#736]
    ;(1ed9c)
1eabc: ab06      add  r3, sp, #24
1eabe: 6811      ldr  r1, [r2,#0]
1eac0: 2022      movs r0, #34 ;0x22
1eac2: 9106      str  r1, [sp,#24]
1eac4: 8891      ldrh r1, [r2,#4]
1eac6: 8099      strh r1, [r3,#4]
1eac8: 7992      ldrb r2, [r2,#6]
1eaca: a908      add  r1, sp, #32
1eacc: 719a      strb r2, [r3,#6]
1eace: 9308      str  r3, [sp,#32]
1ead0: f7fffe3a bl   1e748 <
    sd_ble_opt_set>

1ed9c: 00021f14 .word   0x00021f14


21f0c: 2528 2000 0001 0700 3231 3433
    3635 0000      (%. ....123456..
```

(b) Disassembly of unstripped binary.

```
0x3aba: b84a      ldr  r2, [pc,#0x2e0]

0x3abc: 06ab      add  r3, sp, #0x18
0x3abe: 1168      ldr  r1, [r2]
0x3ac0: 2220      movs r0, #0x22
0x3ac2: 0691      str  r1, [sp,#0x18]
0x3ac4: 9188      ldrh r1, [r2,#4]
0x3ac6: 9980      strh r1, [r3,#4]
0x3ac8: 9279      ldrb r2, [r2,#6]
0x3aca: 08a9      add  r1, sp,#0x20
0x3acc: 9a71      strb r2, [r3,#6]
0x3ace: 0893      str  r3, [sp,#0x20]
0x3ad0: fff73afe bl   #0x3748


0x3d9c: 141f      subs r4, r2, #4
0x3d9e: 0200      movs r2, r0

0x6f14: 3132      adds r2, #0x31
0x6f16: 3334      adds r4, #0x33
0x6f18: 3536      adds r6, #0x35
```

(c) Disassembly of stripped binary.

Figure 10.1: Differences in the disassembly of unstripped and stripped binaries.

10.2 Challenges Involved in the Analysis of Stripped IoT Binaries

IoT Peripherals may implement one or more communication technologies, such as BLE [215], Zigbee [216], ANT [217] or Thread [218]. Many of these technologies have fully-fledged stacks, with protocols defined from the physical up to application layers. To reduce development time and ease application development, many IoT SoC vendors implement the technology stacks themselves and provide APIs through which application developers can configure certain aspects (including security features) of the stacks [219–221]. In addition, developers may use library functions to perform other configurations. We present an example configuration function using sample C code in Figure 10.1a, where a fixed passkey is defined for the BLE pairing process using an API call `sd_ble_opt_set`. We will use this example throughout the chapter.

Focusing on this example, it is known that fixed passkeys are a vulnerability, as they reduce the security of the pairing mechanism. A security analyst would therefore want to identify such uses of fixed passkeys from devices' firmware (since in most cases, the source code will not be publicly available). That is, they would want to know whether `sd_ble_opt_set` (which we term a *Call Of Interest*, or COI) is called with a fixed passkey as its argument. To do this, we would need to pinpoint the location of the function call within the firmware binary, and then analyse the arguments that are passed to it. Figure 10.1b depicts the assembly instructions corresponding to this section of code, obtained by disassembling the firmware binary. From the instructions, we are able to identify that the relevant function call occurs at address `0x1ead0`, that the passkey bytes occur at address `0x21f14`, and that they are referenced by their absolute location at address `0x1ed9c`.

The ability to correctly deduce the above pieces of information depends on a set of **conditions**:

C1 Knowledge of function location and callers' addresses (i.e., knowing that the code for `sd_ble_opt_set` is at address `0x1e748` and that it is called at address `0x1ead0`).

C2 Knowledge of locations of inline data (i.e., knowing that the bytes at addresses `0x1ed9c` and `0x21f0c` should be interpreted as data rather than as code).

C3 Firmware being loaded at the correct address (such that the absolute address `0x21f14` results in bytes being loaded from the correct location).

This information is present within headers and symbol tables within the firmware. However, due to storage considerations, most IoT Peripherals tend to ship firmware with this information removed, i.e., as *stripped binaries*.

Figure 10.1c depicts the disassembly of the binary file with ELF headers and debugging symbols stripped out. The disassembly of the stripped binary does not contain information about function names, thereby making it difficult to pinpoint locations of function calls (failing Condition C1). Data segments have been incorrectly interpreted by the disassembler as code (failing Condition C2),[5] which leads to incorrect results when performing value tracing and precludes the use of emulation frameworks (e.g., QEMU [224], unicorn [225]). Further, the code has been loaded at the incorrect offset (failing Condition C3), which means absolute addressing will fail.

Contributing to this problem is the fact that many resource-constrained IoT devices feature ARM processors [226] with the Thumb and Thumb-2 instruction sets (as opposed to the ARM instruction set),[6] to achieve greater code densities [227]. In fact, the ARM Cortex-M processors, which are very popular in embedded systems, support *only* the Thumb and Thumb-2 instruction sets. These instruction sets are not yet fully supported by many disassemblers.

Out of the current state-of-the-art reverse-engineering tools, `IDA` (free) [228] does not currently support ARM,[7] while `Debin` [229] and `BAP` [230] do not fully support the Thumb instruction set. Testing free reverse-engineering tools that *do* support Thumb analysis against a simple stripped Cortex-M IoT binary, we found that `radare2` [231] failed to identify almost 40% of the functions within the binary (analysing using `aaa` and `aaaa`), `Ghidra` [232] failed to identify 30% of functions, while `angr` [233] was unable to produce a valid Control Flow Graph (CFG) - a step prior to analysis. Our observation regarding the robustness of `angr` and `radare2` for Thumb mode analysis is supported by [222], which also noted that `Ghidra` too has better support for the ARM instruction set than for Thumb. Further, because IoT peripheral binaries typically do not

[5]Note that inline data is far more common in ARM than in x86/x64 [222], and the misinterpretation of such data as code is a problem that is encountered even with state-of-the-art disassemblers [223].

[6]The ARM instruction set features fixed 32-bit instructions. The Thumb instruction set is a compact 16-bit encoding of a subset of the ARM instruction set. Thumb-2 extends the Thumb instruction set with additional 32-bit instructions.

[7]We consider only free tools, to increase reach and improve accessibility for researchers.

Algorithm 10.1: Application code base identification.

Result: Application code base

```
1  $vtAddresses = []$ ;                                    ▷ Vector Table (VT) addresses.
2  for $vtIndex \in [1, 2, 3, 4, 5, 6, 14, 15]$ do
3      $vtEntry = readBytesFromBinary(vtIndex, vtIndex + 4)$;
4      $vtAddresses.insert(vtEntry - 1)$;
5  end
6  for $branchIns \in disassembledInstructions$ do
7      if $target(branchIns) == address(branchIns)$ then
8          for $vtAddress \in vtAddresses$ do
               ▷ Consider addresses whose last 3 hex digits match those of a VT entry.
9              if $vtAddress[-3 :] == address(branchIns)[-3 :]$ then
10                 $appCodeBase = (vtAddress - address(branchIns))$;
11             end
12         end
13     end
14 end
```

include the technology stack or ROM data, *dynamic* analysis approaches are unsuitable. This reveals a gap in the automated IoT security analysis landscape and prompted the development of `argXtract`.

10.3 `argXtract`

We design `argXtract` to take as input the disassembly of a stripped Cortex-M binary, perform several levels of processing, and finally extract and output arguments to security-relevant COIs.

The input disassembly is obtained via any existing disassembler and will very likely feature the issues described in §10.2. The processing is divided into the following stages: §10.3.1 - **Application code base identification**, for correct absolute addressing; §10.3.2 - **Data identification**, such that data is not incorrectly interpreted as code; §10.3.3 - **Function boundary identification**, to enable call execution path generation and to enable function pattern matching; §10.3.4 - **COI identification** (function call or ARM supervisor call), to produce a list of trace termination points; §10.3.5 - **Tracing and argument processing**, to determine the input arguments passed to a Call Of Interest.

10.3.1 Application Code Base Identification

As described in §10.2, using an incorrect offset for instruction addresses will lead to the failure of absolute addressing. `argXtract` combines *known* address information with *obtained* addresses to compute the application code base using Algorithm 10.1.

The addresses of core interrupt handlers are known, as they are present at specific offsets (line 2) within the Vector Table (VT), which is located at `0x00000000` within the stripped binary [234]. The addresses will have a Least Significant Bit (LSB) of 1, indicating a switch to Thumb mode. The actual address is obtained by subtracting 1 (line 4). Corresponding interrupt handler *code* within the stripped binary is identified by exploiting the fact that at least one interrupt handler is usually the *default handler*, i.e., an endless loop or self-targeting branch. Addresses of self-targeting branches are extracted from the disassembly (lines 6-7) and compared against VT addresses to compute the correct offset (lines 8-10).

Once the application code base has been identified, the binary disassembly is reloaded at the correct offset. It now satisfies Condition C3 as presented in §10.2. `argXtract` then examines the first block of words within the file for valid address structures and uses this information to compute the size of the vector table; it sets the code start address as $application\_code\_base + vector\_table\_size$.

10.3.2 Inline Data Identification

Disassembly of an unstripped binary file produces a `.text` (i.e., code) segment and often a `.data` segment, with a clear demarcation between the two, gleaned from section information. *Stripped* Cortex-M binaries do not contain section information, which means that their disassembly will produce a block of instructions with no distinction between the `.text` and `.data` segments, and with the `.data` segment misinterpreted as code. The `.text` segment also tends to feature inline data, which is also often misinterpreted as code and thereby results in value tracing errors.

The data identification component of `argXtract` uses information from the Reset Handler, whose address is read from the Application Vector Table, to identify the location and correct starting address of the `.data` segment. It also identifies inline data using four primary sources: (i) PC-relative memory-loads (e.g., `ldr`, `ldrh`), (ii) direct write-to-PC operations (iii) table branches (`tbb`, `tbh`), and (iv) compact switch table helpers such as `__ARM_common_switch8` and `__gnu_thumb1` variants. These operations aid in satisfying Condition C2 (described in §10.2). We describe the data identification mechanism for each of these sources in further detail below.

Identification of `.data` The Reset Handler often contains the final address of the `.text` segment as well as the start and end addresses for the `.data` segment. This is present in the form of consecutive memory-loads, where the first memory-load reads in the address from which the `.data` segment starts and subsequent memory-loads read the (actual) start and end addresses for the `.data` segment. An example has been shown in Figure 10.2. `argXtract` analyses instructions within the Reset Handler to determine whether they match the required structure. If they do, then the addresses starting after the final address of the `.text` segment and ending at the end of the file are marked as data, i.e., as the `.data` segment. The addresses within the newly-identified `.data` segment are also modified according to the information extracted from the Reset Handler. In the example in Figure 10.2, the memory-loads at addresses `0x24176` and `0x24178` denote that the addresses from `0x263fc` onward need to be reinterpreted as data, and need to be re-addressed with addresses starting from `0x200033b0`.

PC-relative memory-loads A memory-load (i.e., `ldr` and variants) that loads data from an address within the firmware file will specify the source address relative to either the Program Counter (PC) or a register. Register-relative loads may require significant tracing in some cases. However, PC-relative loads are straightforward to analyse. `argXtract` performs a linear scan for

```
00024164 <Reset_Handler>:
  ...
  24172:  430a        orrs  r2, r1
  24174:  6002        str  r2, [r0, #0]
  24176:  4908        ldr  r1, [pc, #32]   ;(24198)
  24178:  4a08        ldr  r2, [pc, #32]   ;(2419c)
  2417a:  4b09        ldr  r3, [pc, #36]    ;(241a0)
  2417c:  1a9b        subs  r3, r3, r2
  2417e:  dd03        ble.n  24188 <Reset_Handler+0x24>
  24180:  3b04        subs  r3, #4
  24182:  58c8        ldr  r0, [r1, r3]
  24184:  50d0        str  r0, [r2, r3]
  24186:  dcfb        bgt.n  24180 <Reset_Handler+0x1c>
  24188:  f000 f812   bl  241b0 <SystemInit>
  2418c:  f7f6 ffc8   bl  1b120 <_mainCRTStartup>
  24190:  40000524   .word  0x40000524
  24194:  40000554   .word  0x40000554
  24198:  000263fc   .word  0x000263fc
  2419c:  200033b0   .word  0x200033b0
  241a0:  20003460   .word  0x20003460
```

Figure 10.2: Identification of `.data` using Reset Handler.

Algorithm 10.2: Inline data identification (memory load).

```
for $instruction \in disassembledInstructions$ do
    if $opcode(instruction) \in ldrInstructions$ then
        $ldrTarget = target(instruction)$;
        if $isPcRelativeAddress(ldrTarget)$ then
            $numBytes = 1$;
            if $opcode(instruction) == ldr$ then
                $numBytes = 4$;
            else if $(opcode(instruction) \in [ldrh, ldrsh])$ then
                $numBytes = 2$;
            $markAddressAsData(ldrTarget, numBytes)$;
            if $(numBytesAtTarget == 4) \& (numBytes < 4)$ then
                $reinterpretOverflowAsInstruction(ldrTarget + 2)$;
            end
        end
    end
end
```

PC-relative memory-loads, calculates the address from which data is loaded and marks it as data, reprocessing residual bytes as instructions where required. This is described in Algorithm 10.2.

Write-to-PC operations Direct write-to-PC operations are sometimes used to accomplish code branches. Figure 10.3 depicts an example. This operation loads a branch address from an address within the firmware and writes the branch address to the PC. The address from which the branch address is loaded (i.e., the `ldr` source at `0x1a844`, obtained in this example by adding the contents of `r2` and `r3`) must be interpreted as data, but is misinterpreted as code within the disassembly of stripped binaries.

When a write-to-PC is encountered (at `0x1a846` in Figure 10.3), `argXtract` examines the preceding instructions until an integer comparison is identified (`0x1a83c`). It then uses subsequent

```
1a83c: 2c17    cmp    r4, #23
1a83e: d8fc    bhi.n  1a83a
1a840: 4ac7    ldr    r2, [pc, #796]
1a842: 00a3    lsls   r3, r4, #2
1a844: 58d3    ldr    r3, [r2, r3]
1a846: 469f    mov    pc, r3
```

Figure 10.3: Write-to-PC operation.

```
2894a: 2e08          cmp     r6, #8
2894c: d219          bcs.n  28982
2894e: e8df f006     tbb    [pc, r6]
28952: 1b181804      ; data
28956: 172f2f22      ; data
2895a: 8901          ldrh    r1, [r0, #8]
```

Figure 10.4: Sample table branch structure.

conditional branches (`0x1a83e`) to determine the actual range of values for the comparison register ([0,23] in this example). The instructions following the branch and until the non-PC-relative memory-load (`0x1a844`) are executed for all possible values of the comparison register. This produces a range of addresses from which the branch addresses are loaded. The range of addresses is marked as data. A second execution until the PC-write instruction produces the set of branch addresses, which are stored to be used by the function boundary identification component (§10.3.3).

Table branches Table branch instructions (`tbb`, `tbh`) were introduced in the ARMv7-M architecture to handle complex branching conditions. Figure 10.4 depicts a sample table branch instruction (at address `0x2894e`). The instruction is immediately followed by a branch table (`0x28952` and `0x28956`). This table should be interpreted as data, but is misinterpreted by disassemblers as code in the absence of debugging symbols.

In the case of table branch instructions, an index value is used to index into the branch table. `argXtract` follows the procedure described in Algorithm 10.3, exploiting the comparison made with the indexing register (`0x2894a`) - similar to the process described for write-to-PC operations - to identify the range of addresses that make up the branch table. In the example in Figure 10.4, the comparison (`0x2894c`) and conditional branch (`0x2894a`) indicate that the branch table has 8 entries. Because the table branch instruction in our example is `tbb`, the table will consist of single-byte offsets (if the instruction had been `tbh`, the table would contain halfword offsets). `argXtract` processes this information and marks the 8 bytes from the PC onward as data.

Compact switch helpers Prior to the introduction of table branch instructions, helper functions were utilised to handle switch-case constructs. The GCC compiler produces `__gnu_thumb1` variants, while Keil produces `__ARM_common_switch8`. These helper functions have identifiable function prologues, and calls to the functions are followed by an index table, similar to table branch instructions. `argXtract` determines the locations of helper functions and applies function-specific processing to determine the size of the index table. It also determines addresses of resultant branches, to be used by the function boundary identification module (§10.3.3).

Algorithm 10.3: Inline data identification (table branch).

```
for $instruction \in disassembledInstructions$ do
    if $opcode(instruction) \in tableBranchInstructions$ then
        $mulFactor = (opcode(instruction) == tbh)?2 : 1;$
        $cmpValue = getTableSkipComparisonValue(address(instruction));$
        $maxBranchAddress = pcAddress + (cmpValue * mulFactor) + 2;$
        if $(opcode(instruction) == tbb) and (byte(maxBranchAddress) == 00)$ then
            $maxBranchAddress += 1;$
        end
        $tableBranchAddress = pcAddress;$
        while $tableBranchAddress < maxBranchAddress$ do
            $markAddressAsData(tableBranchAddress);$
            $tableBranchAddress += 2;$
        end
    end
end
```

10.3.3 Function Boundary Identification

Function boundary identification is used within `argXtract` to enable function pattern matching and call execution path determination. The challenges involved in function boundary identification have been widely studied. These include indirect function calls, absence of specific function prologues, indeterminate location of start instructions, absence of a clear exit point and presence of multiple exit points [235]. The presence of inline data within Cortex-M disassembly, which may be misinterpreted as code, can further complicate function boundary estimation [223].

`argXtract`'s function boundary identification is performed in two stages. First, an initial set of high-certainty candidates for function start addresses is generated by extracting the addresses of interrupt handlers from the Vector Table. That is, each interrupt handler is considered as a separate function and the addresses of the interrupt handler functions are obtained from the VT that occurs at the beginning of the binary file. Targets of branch-and-link (`bl`) instructions are added to this set; targets of branch (`b`) instructions are also added, but subject to satisfying requirements regarding function prologues.

In the second stage, a function estimation algorithm (Algorithm 10.4) is executed against each block of instructions beginning from one start address and ending prior to the next start. The algorithm operates on the basic principle that, while a function may have multiple exit instructions due to conditional executions, it must have mechanisms for bypassing all but one of the exit points. This could be via conditional branch instructions or a switch/branch table (as identified in §10.3.2). `argXtract` determines all potential exit points (e.g., `pop`, `bx lr`, unconditional branches to lower addresses or outside the current block, and data) within the block of instructions that is being analysed and marks the exit point that *cannot be bypassed* as the ultimate function exit. The next valid instruction is determined to be the beginning of the next function. This procedure is performed iteratively to obtain the final list of function boundaries.

We further illustrate this algorithm using the code example in Figure 10.5 as reference. This reference code contains two functions, denoted as `functionB` and `functionC`. Of these, `functionB`

Algorithm 10.4: Function estimation.

Result: Start addresses for estimated function blocks

```
1  $functionAddresses = [vtAddresses, branchTargets].sort()$;
2  for $functionAddress \in functionAddresses$ do
3      $start = functionAddress$, $end = getFinalAddressInBlock(functionAddress)$;
4      $exitPoints = getExitPoints(start, end).sort()$
5      for $i = start \rightarrow end$ do
6          if $opcode(i) \in conditionalBranchInstr$ then
7              $branchTarget_i = target(i)$ ;                          ▷ Target of branch.
8              for $exit \in exitPoints$ do
9                  if $doesBranchBypassExit(branchTarget_i, exit)$ then
10                     $exitPoints.remove(exit)$;
11                 end
12             end
13         else if $i \in [switchCalls, PCwrites]$ then
14             $maxBranchAddress = max(switchTable)$;
15             if $doesOverflowBlock(maxBranchAddress, end)$ then
16                 $combineAndReestimate(maxBranchAddress, start)$;
17             end
18             for $exit \in exitPoints$ do
19                 if $doesBypassExit(maxBranchAddress, exit)$ then
20                     $exitPoints.remove(exit)$;
21                 end
22             end
23         end
24     end
25     $nextFunctionStart = getFirstInstructionPostExit(exitPoints)$;
26     $functionAddresses.insert(nextFunctionStart))$;
27 end
```

```
  18228:  bl  182b0 <functionB>
000182b0 <functionB>:
  182b0:  push  {r4, lr}
  182b2:  cmp  r2, #32
  182b4:  blt.n  182c0  ;skips pop at 182be
  182b6:  mov  r0, r1
  182b8:  subs  r2, #32
  182ba:  lsrs  r0, r2
  182bc:  movs  r1, #0
  182be:  pop  {r4, pc}
  182c0:  mov  r3, r1
  182c2:  lsrs  r3, r2
  182c4:  lsrs  r0, r2
  182c6:  movs  r4, #32
  182c8:  subs  r2, r4, r2
  182ca:  lsls  r1, r2
  182cc:  orrs  r0, r1
  182ce:  mov  r1, r3
  182d0:  pop  {r4, pc}
  182d2:  nop
000182d4 <functionC>:   ;not called within code
  182d4:  ldrb  r2, [r0, #0]
  182d6:  adds  r0, r0, #1
```

Figure 10.5: Reference ARM assembly for function estimation.

```
0x18228:  bl    0x182b0
...
0x182b0:  push  {r4, lr}
0x182b2:  cmp   r2, #0x20
0x182b4:  blt   #0x182c0
0x182b6:  mov   r0, r1
0x182b8:  subs  r2, #0x20
0x182ba:  lsrs  r0, r2
0x182bc:  movs  r1, #0
0x182be:  pop   {r4, pc}
0x182c0:  mov   r3, r1
0x182c2:  lsrs  r3, r2
0x182c4:  lsrs  r0, r2
0x182c6:  movs  r4, #0x20
0x182c8:  subs  r2, r4, r2
0x182ca:  lsls  r1, r2
0x182cc:  orrs  r0, r1
0x182ce:  mov   r1, r3
0x182d0:  pop   {r4, pc}
0x182d2:  nop
0x182d4:  ldrb  r2, [r0]
0x182d6:  adds  r0, r0, #1
```

(a) Capstone disassembly (starting point).

```
0x18228:  bl 0x182b0
...
0x182b0:  push r4, lr ;start
0x182b2:  cmp   r2, #0x20
0x182b4:  blt   #0x182c0
0x182b6:  mov   r0, r1
0x182b8:  subs  r2, #0x20
0x182ba:  lsrs  r0, r2
0x182bc:  movs  r1, #0
0x182be:  pop   {r4, pc}
0x182c0:  mov   r3, r1
0x182c2:  lsrs  r3, r2
0x182c4:  lsrs  r0, r2
0x182c6:  movs  r4, #0x20
0x182c8:  subs  r2, r4, r2
0x182ca:  lsls  r1, r2
0x182cc:  orrs  r0, r1
0x182ce:  mov   r1, r3
0x182d0:  pop   {r4, pc}
0x182d2:  nop
0x182d4:  ldrb  r2, [r0]
0x182d6:  adds  r0, r0, #1
```

(b) Identify function blocks using `b`, `bl`.

```
0x18228:  bl    0x182b0
...
0x182b0:  push r4, lr ;start
0x182b2:  cmp   r2, #0x20
0x182b4:  blt   #0x182c0
0x182b6:  mov   r0, r1
0x182b8:  subs  r2, #0x20
0x182ba:  lsrs  r0, r2
0x182bc:  movs  r1, #0
0x182be:  pop {r4, pc} ;end?
0x182c0:  mov   r3, r1
0x182c2:  lsrs  r3, r2
0x182c4:  lsrs  r0, r2
0x182c6:  movs  r4, #0x20
0x182c8:  subs  r2, r4, r2
0x182ca:  lsls  r1, r2
0x182cc:  orrs  r0, r1
0x182ce:  mov   r1, r3
0x182d0:  pop {r4, pc} ;end?
0x182d2:  nop
0x182d4:  ldrb  r2, [r0]
0x182d6:  adds  r0, r0, #1
```

(c) Mark potential exit points such as `pop pc`

```
0x18228:  bl    0x182b0
...
0x182b0:  push r4, lr ;start
0x182b2:  cmp   r2, #0x20
0x182b4:  blt #0x182c0
0x182b6:  mov   r0, r1
0x182b8:  subs  r2, #0x20
0x182ba:  lsrs  r0, r2
0x182bc:  movs  r1, #0
0x182be:  pop {r4, pc} ;end?
0x182c0:  mov   r3, r1
0x182c2:  lsrs  r3, r2
0x182c4:  lsrs  r0, r2
0x182c6:  movs  r4, #0x20
0x182c8:  subs  r2, r4, r2
0x182ca:  lsls  r1, r2
0x182cc:  orrs  r0, r1
0x182ce:  mov   r1, r3
0x182d0:  pop {r4, pc} ;end?
0x182d2:  nop
0x182d4:  ldrb  r2, [r0]
0x182d6:  adds  r0, r0, #1
```

(d) Identify instructions that skip exit points.

```
0x18228:  bl    0x182b0
...
0x182b0:  push r4, lr ;start
0x182b2:  cmp   r2, #0x20
0x182b4:  blt   #0x182c0
0x182b6:  mov   r0, r1
0x182b8:  subs  r2, #0x20
0x182ba:  lsrs  r0, r2
0x182bc:  movs  r1, #0
0x182be:  pop   {r4, pc}
0x182c0:  mov   r3, r1
0x182c2:  lsrs  r3, r2
0x182c4:  lsrs  r0, r2
0x182c6:  movs  r4, #0x20
0x182c8:  subs  r2, r4, r2
0x182ca:  lsls  r1, r2
0x182cc:  orrs  r0, r1
0x182ce:  mov   r1, r3
0x182d0:  pop {r4, pc} ;end?
0x182d2:  nop
0x182d4:  ldrb  r2, [r0]
0x182d6:  adds  r0, r0, #1
```

(e) Remove skipped exit instructions.

```
0x18228:  bl    0x182b0
...
0x182b0:  push r4, lr ;start
0x182b2:  cmp   r2, #0x20
0x182b4:  blt   #0x182c0
0x182b6:  mov   r0, r1
0x182b8:  subs  r2, #0x20
0x182ba:  lsrs  r0, r2
0x182bc:  movs  r1, #0
0x182be:  pop   {r4, pc}
0x182c0:  mov   r3, r1
0x182c2:  lsrs  r3, r2
0x182c4:  lsrs  r0, r2
0x182c6:  movs  r4, #0x20
0x182c8:  subs  r2, r4, r2
0x182ca:  lsls  r1, r2
0x182cc:  orrs  r0, r1
0x182ce:  mov   r1, r3
0x182d0:  pop   {r4, pc}
0x182d2:  nop
0x182d4:  ldrb r2, [r0] ;start
0x182d6:  adds  r0, r0, #1
```

(f) Identify suitable start for next function.

Figure 10.6: Process used by `argXtract` for identifying function blocks.

is called via a `bl` instruction at `0x18228`, while `functionC` is called indirectly via a `blx` call (which means the starting address of `functionC` cannot be identified without some level of register tracing). Figure 10.6a depicts the equivalent assembly code obtained using the `Capstone` [236] disassembler against the *stripped* version of the binary. To estimate function boundaries for this disassembly, we first identify high-confidence function starts, including targets of `bl` instructions. This will result in `0x182b0` being identified as a function start (Figure 10.6b). This corresponds to `functionB`. We next apply our function boundary estimation algorithm to the block of assembly instructions beginning at `0x182b0` as follows:

- Mark out possible exit points, such as `pop pc`, `bx lr`, or data. As shown in Figure 10.6c, there are two such potential exit points, at addresses `0x182be` and `0x182d0`.
- Look for branches that skip the exit points. Figure 10.6d shows that the branch condition at `0x182b4` skips the exit point at `0x182be`. This exit point is therefore considered to be part of the existing function (i.e., the one beginning at `0x182b0`).
- Remove exit points that are skipped. We are left with one other potential exit point at `0x182d0`. It is considered the final exit point of the function (see Figure 10.6e).
- Identify the next valid instruction as the start of the next function. Initially, we consider the instruction at `0x182d2` to be the start of the next function. However, this address contains a `nop` instruction. Therefore, we skip it and mark `0x182d4` as the start of the next function, as shown in Figure 10.6f.

Comparing this with Figure 10.5, we can see that the algorithm has correctly identified the function starting address for `functionC`.

This process is then repeated from the start of the new function (i.e., from `0x182d4`), until the end of the block is reached. At the end of this step, we have a list of function start addresses (i.e., the addresses of functions) for the binary under test.

Function block annotation `argXtract` maintains a function block object, where each key corresponds to an identified function block. The object contains information on cross-references to and from a function, as well as the *call depth*. The call depth is a parameter indicating the maximum number of functions that get called iteratively by a function. This information tells `argXtract` which functions will likely take a long time to execute and are therefore candidates for exclusion when execution is time-limited. If a function contains perpetual loops, as is the case with some error handlers, then it is added to a deny-list, to prevent inadvertently calling such functions.

10.3.4 COI Identification

A COI in our framework could be a standard function call or could be translated to an ARM supervisor call (`svc`). `argXtract` identifies calls to the `svc`s or functions of interest using distinct techniques, as described in §10.3.4.1 and §10.3.4.2, respectively. In both cases, the addresses of the calling instructions are stored, to be used in the tracing step.[8] This then satisfies Condition C1 as described in §10.2. After this step, all pre-conditions for analysing a stripped ARM binary will be satisfied.

10.3.4.1 Supervisor Call Identification

An ARM supervisor call is simply an instruction with an opcode of `svc`. A supervisor call will have an associated `svc` number. If an IoT chipset vendor issues a technology stack that accepts

[8]Direct calls are identified. However, calls via `blx` are not.

configuration commands via supervisor calls, then the associated `svc` numbers will normally be available from the vendor SDK.

In `svc` analysis mode, an input object containing the `svc` numbers of interest is provided to `argXtract`. A linear scan is performed over the disassembly to obtain the addresses of the relevant `svc` instructions. These are stored, to be used in the tracing step (§10.3.5).

10.3.4.2 Function Pattern Matching

Identifying function calls of interest is far more complex than identifying supervisor calls, as functions cannot be immediately identified within assembly. We exploit the fact that configuration API functions (such as those provided by vendors for performing configurations to IoT stacks) accept inputs in a specific order, which are passed within registers in a specific sequence (`r0`, `r1`, ...) for Cortex-M. In addition, most functions generate artefacts that are detectable within memory and/or registers, i.e., as output or intermediate values.

For each function of interest, we define a "function pattern file", which is a collection of test sets containing register and memory inputs, and the corresponding outputs (which could be actual output values, stored in registers or memory, or intermediate values at detectable locations). In the case of functions that store identifiable values at binary-specific locations that *cannot* be predetermined, we propose wildcard addresses, where expected values are specified at some predetermined *offset* from the wildcard address.

The function pattern files are passed to each of the functions that have been identified (using the process described in §10.3.3) for the binary under test. The function instructions are executed with the input register and memory values specified in the pattern file. Output register and memory contents are compared against the expected values. If a single function matches the given pattern, then this is taken to be the function of interest. In the case of nested function calls, the function with the lowest call depth that satisfies the given pattern is taken to be the function of interest. Polymorphism will be detected if the processing of the inputs differs between the functions such that the artefacts/outputs are different.[9] Note however that if two functions produce the same outputs for any given inputs, then function pattern matching will fail.

10.3.5 Register Tracing and Argument Processing

Once COIs have been identified (as described in §10.3.4), `argXtract` performs backward interprocedural tracing to determine all call execution paths leading to the COI(s). It then forward-traces along the paths while updating the values of registers, memory and conditional flags with each instruction execution. Note that `argXtract` does not, in the interest of processing time and resources, follow every possible branch that it encounters during the trace.[10] Instead, it follows a set of rules, to determine whether or not to proceed with a branch:

[9]Note that input structures within function pattern files are provided in byte format. Therefore, differences in input *type* do not impact the analysis.

[10]A fairly simple firmware binary can have several thousand branch instructions, which rapidly compounds the required memory as well as analysis time.

- All branches that are within the identified call execution path are followed.
- All unconditional internal loops (i.e., branches to other addresses within the same function) are followed.
- Conditional internal loops are followed if the conditions are satisfied. If conditional checks cannot be made (due to the flags not being set as expected, for example), then the outcome is *indeterminate* and both possible paths are taken.
- Branches to functions on the deny-list are not taken.
- Branches to functions that are *not* present within the call execution path are only followed if their call depth is less than a configurable *max_call_depth* value (default=1).

If a branch check should fail, then `argXtract` skips it and proceeds to the next instruction. If a `bl` is skipped, then going by the ARM convention of using register `r0` for holding the output of subroutines [237], and the convention of returning 0 on execution success, we artificially assign a value of 0 to `r0` in order to bypass a possible subsequent branch to fault handlers due to non-zero status returns.

Once the COI is reached via the trace, its arguments are extracted for processing. The arguments to a COI are contained within registers `r0-r3` (or are on the call stack) [238–240]. Some of these register values may actually be pointers to data in memory. Therefore, when a COI is reached, the contents of both the register object and the memory map are returned to an argument analysis component for processing.

The type and format of data that are used as arguments to COIs are obtained from header files within vendor SDKs and provided to `argXtract` in the form of *Argument Definition Objects*. These are JSON files that describe the expected structure of data bits for each input argument using predefined keywords.[11] For example, Figure 10.7a depicts the Argument Definition Object for the `sd_ble_opt_set` COI discussed in §10.2. A corresponding trace output may look like that depicted in Figure 10.7b. Taking the second argument as an example, we note that it is defined as a pointer to a pointer to a 6-byte (48-bit) array. This argument is contained within register `r1`, which according to the trace output in Figure 10.7b contains a value of `20007f68`. As the Argument Definition Object indicates that this is a pointer, we refer to the contents of memory. The memory object in Figure 10.7b shows that the address `0x20007f68` contains the value `20007f60`. This (also being a pointer) is interpreted as a memory address, `0x20007f60`, which contains the hex value `0x313233343536`. This corresponds to the ASCII string "123456", which is the value specified as the fixed passkey in our example in Figure 10.1a. This results in the output depicted in Figure 10.7c.

Arguments that hold *outputs* (by being written to a memory location) will be specified in the Argument Definition Object to be fed back into the memory map and used in subsequent traces. This is used to tie together two separate traces. We use this functionality to link a BLE characteristic with its associated service when analysing Nordic binaries (§10.5).

[11]We adopt this template-based approach for greater flexibility, such that supporting additional COIs only requires including new Argument Definition Objects, rather than needing to add extra COI-specific code.

```
{
   "args": {
      "0": {...},
      "1": {
         "in_out": "in",
         "ptr_val": "pointer",
         "data": {
            "p_opt": {
               "ptr_val": "pointer",
               "length_bits": 48,
               "type": "hex"
            }
         }
      }
   }
}
```

(a) Argument definition object.

```
{"sd_ble_opt_set":{
   memory:{..., 0x20007f60:"31", 0x20007f61:"32", 0x20007f62:"33",
      0x20007f63:"34", 0x20007f64:"35", 0x20007f65:"36", 0
      x20007f66:"00", 0x20007f68:"60", 0x20007f69:"7f", 0x20007f6a
      :"00", 0x20007f6b:"20", ...},
   registers:{..., r0:"00000022", r1:"20007f68", ...}}}
```

(b) Sample register/memory contents.

```
"output": {
   "sd_ble_opt_set": [
       {
           "opt_id": 34,
           "p_opt": "313233343536"
       }
   ]
}
```

(c) Processed output file.

Figure 10.7: Template-based argument processing.

10.4 Evaluation

We implement `argXtract` using Python. We select `Capstone` as the disassembler, as it underpins ARM disassembly for a large number of existing reverse-engineering and analysis tools, including `radare`, `angr` and `binwalk`. Our implementation supports the entire ARMv6-M instruction set and the most prevalent instructions within ARMv7-M. In this section, we evaluate our implementation in terms of the accuracy of function boundary identification and pattern matching, and the correctness of extracted configurations.

10.4.1 Test Set and Ground Truth

There is no publicly available dataset of ARM Cortex-M binaries with known and annotated configurations. For this reason, we generated our own dataset, comprising 28 stripped binaries, for testing and verification purposes. The binaries target chipsets from NXP, STMicroelectronics, Nordic Semiconductor and Texas Instruments, for multiple IoT technologies including Zigbee,

Table 10.1: True Positive Rates (TPR) and Effective False Positive Rates (EFPR) for function block identification against test binaries. EFPR is computed by discounting misidentifications that do not impact the trace.

Bin File		argXtract		radare2[1]		ghidra[2]		Bin File		argXtract		radare2[1]		ghidra[2]	
ID†	#Fns‡	TPR	EFPR	TPR	EFPR	TPR	EFPR	ID†	#Fns‡	TPR	EFPR	TPR	EFPR	TPR	EFPR
0a1	324	100	0.29	95.68	2.19	87.96	0	0d2	841	0	100	69.32	7.4	69.68	2.85
1d7	951	93.27	0.97	74.24	3	73.08	0.82	3b1	204	99.02	0	88.24	0	82.35	0
443	598	100	0	83.78	3.05	83.95	1.95	4d7	1563	95.71	1.17	78.57	3.28	77.8	0.15
589	1486	97.51	0.68	83.24	1.59	84.79	0	5d3	398	99.50	0.73	94.97	0	93.47	0
646	166	98.80	0	80.72	0.73	77.71	0.76	67e	2138	99.16	0.05	82.69	0.54	83.4	0.22
681	1961	97.86	0.56	94.19	0.7	87.51	0.12	6ac	265	98.11	0.37	73.96	0.5	72.08	0
70b	115	95.65	0	67.83	0	73.04	0	7e8	1529	97.58	0.66	81.62	1.57	84.96	0
928	520	95.38	0.93	90.19	0	71.15	0	938	2764	99.57	0.74	85.53	0.59	83.90	0.09
989	762	95.80	9.7*	69.27	8.62	63.10	9.39	ade	1951	99.33	0.89	89.54	0.46	87.69	0.12
bad	839	92.25	0.79	69.85	5.43	68.53	0	be7	2035	99.71	0.39	5.11	87.43	4.72	88
cb5	92	94.57	1.11	61.96	0	67.39	0	cc8	1582	94.82	0.71	82.68	1.91	83	0
dd9	801	96.63	6.9*	95.93	6.36	88.15	6.66	e2a	495	95.15	0.39	89.7	0	69.29	0
e2d	698	96.42	0.35	94.99	0	86.25	0	f2b	1926	99.79	0.65	81.15	1.01	79.85	0.06
f37	1585	95.21	1.16	78.23	3.18	78.36	0	fe9	1007	99.40	0.1	61.27	0.80	56.21	0.70

†ID = First three characters of SHA256 of binary. ‡#Fns = Number of functions. [1]radare2 was executed using `aaa` analysis mode, [2]Ghidra was executed using the *ARM Aggressive Instruction Finder* option. Both were provided with the application code base manually.

ANT, BLE, Thread and 802.15.4. The binaries were compiled using GCC, IAR, Keil and Clang, depending on the options made available by the chipset vendor. We provide a detailed description of the test set binaries in our code repository. For ground truth, we obtained the actual configuration for each binary by disassembling its *unstripped* version using the GNU ARM embedded toolchain.

10.4.2 Accuracy of Function Boundary Identification

We evaluate the accuracy of `argXtract`'s function boundary identification (§10.3.3) by estimating function boundaries, i.e., function start addresses, for the 28 stripped binaries within our test set and comparing them against the actual functions from the unstripped versions. For comparison, we also do the same using `radare2` and `ghidra`. We present our results in Table 10.1.

Considering the True Positive Rate (TPR column), the table shows that for all but five binaries, more than 95% of functions are correctly identified by `argXtract`. The results are more variable for `radare2` and `ghidra`, but in general the TPRs obtained by these two tools are lower (often significantly lower) than those obtained by `argXtract`. Manual analysis of a sample of functions (across the test set) that were correctly identified by `argXtract` but not by `radare2` or `ghidra` showed that many such functions occurred after inline data or less traditional function exit points. The techniques employed by `argXtract` for inline data identification and function boundary identification enable it to handle such instances and identify a greater proportion of function start addresses correctly. There was a single exception (binary with ID=0d2), where `argXtract` resulted in a TPR of 0% while `radare2` and `ghidra` identified approximately 70% of functions. This was a binary where the `.text` segment was split into two sections, each with a different offset. `argXtract` was unable to compute the offsets in this case, which meant that further analysis was not possible. Additionally, manual analysis of the remaining four cases where `argXtract` produced a TPR $< 95\%$ showed that the unidentified functions were of unusual structure, e.g., functions accessed via direct conditional branches, or containing only a `bx lr` instruction. These are likely to be fragments of other functions or shared functions. We observe that for the vast majority of such cases, `radare2` and `ghidra` also failed to identify the functions.

Examining false positives (regardless of the analysis tool), we found that in many cases misidentified functions were either where unannotated data had been identified as the start of function blocks, or where a logical function start *can* be assumed, e.g., blocks of alternating `ldr` instructions and data bytes causing each `ldr` to be considered as the start of a new function. In the former case, these particular "functions" will never be called during the tracing phase. In the latter, the functions *are* directly addressed as if they are individual functions. Therefore, such FPs will not affect the trace. We thus consider an "Effective FPR" to denote the false positives *excluding* such instances. The EFPRs obtained by `argXtract` are fairly similar to those obtained by `ghidra` (within 1-2% of each other). `radare2` was more likely to result in a higher EFPR; manual analysis showed that this was often due to `radare2` incorrectly considering `push` instructions to be the start of a function. Overall, `argXtract` resulted in EFPRs of $< 1.5\%$ for all but two binaries (marked with * in the table). These were both compiled by IAR, which is the only compiler we have observed that uses `bl` instructions to branch and link *within* a function. This accounts for the higher EFPR for these two binaries. While this will not impact the actual branching functionality, it will influence the call depth calculation, which in turn *could* impact tracing. We observe that `radare2` and `ghidra` also resulted in high EFPRs for these two binaries.

10.4.3 Accuracy of Function Pattern Matching

We verified the pattern matching module against the `CryptoKeyPlaintext_initKey` function from the Texas Instruments SimpleLink Platform, the `mbedtls_ssl_conf_ciphersuites` from the mbedtls library, and the `ot::KeyManager::SetKeyRotation` OpenThread function. When testing for these functions, we generated stripped binaries using different vendor SDKs (where relevant), as well as different projects and compilers (Keil, IAR, Clang), to account for vendor- and compiler-introduced variations. `argXtract` was able to identify the correct function location from within the stripped binary in each case, which we verified using the unstripped versions of the binaries. To further check the accuracy of `argXtract`'s function pattern matching, we manually verified it against the `HAL_Write_ConfigData` and `aci_gap_init` functions within a real-world STMicroelectronics binary by comparing their functionality against the functions within an unstripped reference binary.

10.4.4 Correctness of Results

For correctness checks, we perform tests using binaries generated with known configurations, as well as verification using a real-world binary and associated device.

We use a subset of ten binaries from our test set, targeting Nordic and STMicroelectronics chipsets, compiled using GCC, Keil and IAR, and implementing ANT and BLE. For Nordic ANT binaries, we define different channel settings and encryption keys. For STMicroelectronics BlueNRG binaries, we define different advertising addresses and privacy configurations. For Nordic BLE binaries, we define 3 BLE services with very specific configurations as follows:

- Heart Rate Service: must include the Heart Rate Measurement and Body Sensor Location characteristics.
- Device Information Service: must include the Manufacturer Name String characteristic and *no other characteristics* (Nordic defines all possible characteristics associated with the Device Information Service within its BLE stack. However, an accurate trace should only identify those characteristics that we have explicitly included within our code).
- Custom service: must include our custom characteristic.

Each characteristic also has specific permissions. It is only if all these configurations are identified exactly that an output is taken to be correct.[12]

In our experiments, we found that all of the conditions were satisfied for all test binaries within our control set, i.e., the configurations were extracted exactly as expected.

We additionally verified `argXtract` against the Goji Go Activity Tracker, whose firmware we had extracted from its companion mobile application. The tracker had two SIG services, as well as the Nordic DFU service and a developer-defined service. Comparing the results obtained using `argXtract` with those we obtained from manual analysis (via a combination of device interaction using the nRF Connect app [241] and profiling using our `ATT-Profiler` (see Chapter 9)), we found that `argXtract` accurately extracted the configuration of the device.

10.5 Case Study I: BLE Security and Privacy (Nordic)

In this section we present a case study for the identification of BLE configuration vulnerabilities in binaries that target Nordic Semiconductor chipsets. For its BLE offerings, Nordic provides a BLE stack to which configuration requests are issued using supervisor calls.

Building the firmware dataset As we have observed previously, BLE Peripherals typically interface with one or more mobile applications. Many of these mobile applications implement a firmware upgrade and/or factory reset procedure. The firmware used for this purpose is either included within the mobile application itself or is downloaded from a server. The firmware for Nordic chipsets is identifiable due to its specific structure and included files.

We programmatically extracted 243 unique[13] Nordic BLE binaries from a large dataset of BLE-enabled Android APKs, obtained from AndroZoo [110] and Google Play. We check for the possibility of cloned firmware or different versions of firmware from a single developer, which can result in the same set of characteristics in different files. We use `ssdeep` for this purpose, with

[12]The configurations we have made depended on the options made available by the respective vendors.
[13]Determined by the SHA256 computed over the file bytes.

a threshold of 70%, to account for the fact that a lot of the Nordic baseline code will likely be the same across files. The output showed that 7 clusters were present within our dataset, with an average of 3 files per cluster. We account for these, where relevant, when presenting our results.

Execution environment We executed `argXtract` on a VM running Ubuntu 18.04.3 LTS with 64 GB RAM and 10 processor cores. Taking RAM usage into consideration, 8 parallel processes were used.

Section outline The remainder of this section describes our findings. We first review the protection (or lack thereof) applied to BLE data across the binaries for the link and application layers (§10.5.1). We then analyse instances of weakened pairing due to the use of fixed passkeys (§10.5.2). Finally, we examine privacy concerns identified for our dataset due to two reasons: static addresses (§10.5.3) and device/manufacturer names (§10.5.4). Each subsection describes the supervisor call that was targeted, mentions the obtained results, and discusses the security or privacy implications of the results.

10.5.1 Security of BLE Data

As mentioned in §2.1.3.3 and §2.2.4, BLE data attributes can have different permissions applied to them to restrict access. Authentication/encryption permissions are specified in terms of different security modes and levels and are applied to the link layer, while authorisation permissions are applied at the application layer.

Characteristics and specific descriptors can have authentication and authorisation permissions. According to the characteristic's properties, the method of applying security will differ. In particular, a characteristic that has the `read` property will have different security mechanisms to one that has `notify` or `indicate` properties, despite the outcome for both being somewhat similar (i.e., the connected device obtains the characteristic value). A read request requires that the connected device satisfy the read permissions specified for the characteristic value itself, while subscribing to notifications requires that the connected device satisfy the *write* permissions specified for the characteristic's CCCD.

Extracting characteristic security configurations The security requirements for BLE characteristics in Nordic devices are defined when the characteristics are declared. This is achieved using the `sd_ble_gatts_characteristic_add` call. To tie each characteristic to a service such that it can be uniquely identified, we also analyse the `sd_ble_gatts_service_add` call.

Executing `argXtract` against our dataset with a maximum execution time of 1.5 hours per trace returned 199 valid output files. Analysis was completed in less than 2 minutes for ∼50% of the binaries. 25% of the binaries required 2-10 minutes, while the remaining required more than 10 minutes. The 199 binaries contained 6 of the previously mentioned clusters. Manual examination showed that the corresponding output files within each cluster had the same service configurations. We therefore consider only 188 unique outputs.

Table 10.2: Protection applied to developer-defined data.

Access	Description	#Bins
Reads	Binaries with developer-defined readable characteristics	167
	Binaries with Mode 1 Level 2 link-layer protection for developer-defined readable characteristics	5
	Binaries with Mode 1 Level 3 link-layer protection for developer-defined readable characteristics	0
	Binaries with application-layer security for developer-defined readable characteristics	7
Writes	Binaries with developer-defined writable characteristics	169
	Binaries with Mode 1 Level 2 link-layer protection for developer-defined writable characteristics	4
	Binaries with Mode 1 Level 3 link-layer protection for developer-defined writable characteristics	0
	Binaries with application-layer security for developer-defined writable characteristics	69*

** 24 excluding Nordic DFU control point, from Nordic DFU library.*

10.5.1.1 Insufficient Protection for BLE Data

In this section, we discuss the protection applied to BLE data for the binaries in our dataset. Because BLE characteristics can either be defined by the Bluetooth SIG, with SIG-specified security configurations, or defined by the device developer with developer-specified security, we analyse the two instances separately.

(i) Protection for SIG-defined BLE data: `argXtract` extracted SIG-defined characteristics from 103 binaries. We compared the obtained security configurations with the expected values specified by the SIG and found that in all cases, the devices do follow the SIG specifications, in that most characteristics have no security applied to them.

The results also revealed an interesting observation for the SIG-defined characteristics that *do* have security requirements. In many such cases, the SIG specifies a choice of protection levels, normally Mode 1 Level 2 *or* Mode 1 Level 3. These can be achieved using *Just Works* or *Passkey Entry* pairing, respectively. While both *Just Works* and *Passkey Entry* are known to be vulnerable to passive eavesdropping attacks (as discussed in §3.4), *Passkey Entry* should be the choice for greater security, as it provides some MitM protection. However, our results show that device developers have invariably opted for the *lower* security level, i.e., Mode 1 Level 2. This may be due to insufficient IO capabilities on the devices precluding the use of *Passkey Entry*, although we have previously observed real-world devices using *Just Works* even when sufficient IO capabilities *were* available (see §9.5.1). We reiterate that, even if the BLE device *does* have sufficient IO capabilities, so long as Mode 1 Level 2 is specified, an attacker can often manipulate the pairing process such that *Just Works* is used.

(ii) Protection for developer-defined BLE data: `argXtract` extracted at least one developer-defined characteristic from 170 binaries. Table 10.2 summarises the link layer and application layer protection applied to the developer-defined characteristics, broken down into readable (including notifications/indications) and writable characteristics.

From the table, we conclude that protection for reads is virtually non-existent at the link-layer, with only five firmware binaries specifying Mode 1 Level 2 authentication requirements. Similarly, authorisation requirements are also not prevalent among readable characteristics, with only seven binaries specifying protection at higher layers.

Writable characteristics fare similarly to readable characteristics in terms of link-layer protection, with only four binaries specifying Mode 1 Level 2 authentication requirements. App-layer protection is slightly better for writable characteristics, but a significant proportion of binaries apply no protection at all to their writable characteristics (apart from those provided by Nordic itself, for its firmware upgrade procedure).

Security implications The security of a BLE device is strongly associated with the authentication and authorisation permissions applied to its data. As we have discussed in §3.4, having freely accessible BLE characteristics means any user in the vicinity of the BLE Peripheral will be able to read and write the data, subject to the characteristic being readable/writable. For that matter, even if the characteristic is protected by *Just Works* pairing, an attacker in the vicinity may be able to covertly pair with the device and access its data. Further, even if strong link-layer protection *is* present, absence of application-layer protection makes the data vulnerable to access by unauthorised apps, as we have shown in Chapter 5.

Implications for readable data within the dataset: Among the binaries that had no link-layer or application-layer protection for readable characteristics, we found numerous fitness tracking devices, as well as healthcare devices, all of which potentially store detailed information regarding a user's activity or health. No protection or only *Just Works* protection means that this personal and sensitive data is vulnerable to unauthorised access, via local *and* remote attacks.

Implications for writable data within the dataset: Within the binaries that had writable characteristics, we found one that contained the SIG-defined Human Interface Device (HID) service. This only had Mode 1 Level 2 link-layer protection applied to its characteristics. Again, this security requirement can be satisfied by *Just Works* pairing, which means that an attacker could transmit unsolicited messages to the HID device, and also read and modify the keyboard characters that are transmitted between the HID device and host via a MitM attack. This has been demonstrated in [242]. We have informed the developer of this vulnerability, but have not received a response.

The vast majority of devices applied no protection to *any* of their writable characteristics. Among these were smart switches, medical respiratory devices (nebulisers) and ECG monitoring devices. Writing random bytes to characteristics on such devices could cause the devices to function improperly or cease to function entirely. If the behaviour of the device corresponding to the written values is known to an attacker, then they can write carefully chosen values in order to modify the expected behaviour, possibly with harmful consequences.

10.5.1.2 Different Permissions for Read vs. Notify

As mentioned in §2.1.3.3, the value held within a characteristic can be accessed in different ways, either using a direct read request or via notifications/indications, and even though the mechanisms of access differ, the ultimate outcome is similar. We observed that one binary within our results contained characteristics that had both `read` *and* `notify` properties, but with different security properties set for the two types of access. Mode 1 Level 2 security was required to be able to *read* the characteristics' values, while the values could be freely accessed via *notifications* (their CCCDs were writable without the need for any pairing or higher-layer protection).

Security implications Different security levels for different value acquisition methods implies that the data can always be accessed using the less secure mechanism. In addition, there may be a false sense of security, as the protection will be assumed to be higher than it actually is. This finding shows that developers may unintentionally leave "gaps" in security, particularly when incorporating different functionalities. We have informed the developer about this issue, but have received no response.

10.5.2 Use of Fixed Passkeys

The *Passkey Entry* association model provides MitM protection by requiring that a user manually key in a passkey that is displayed on the BLE Peripheral. However, some developers choose to program a fixed passkey into the Peripheral. This might be because many BLE Peripherals don't have IO capabilities (i.e., keypad or display), but could also originate from bad practices when programming devices that *do* have these capabilities.

Identifying fixed passkeys The `sd_ble_opt_set` supervisor call enables setting multiple options on Nordic BLE devices, where each option is identified using an `opt_id`. The `opt_id` with value 34 denotes setting a fixed passkey. `argXtract` identified a binary within the dataset that contained a fixed passkey of `0x303030303030`, i.e. "000000".

Security implications Fixed passkeys undermine the security of the *Passkey Entry* model, particularly if the same passkey is used for all devices of a certain brand. In such a scenario, an attacker would only need to know the passkey for one device in order to be able to covertly connect to any device of the same brand. This effectively removes the MitM protection afforded by the *Passkey Entry* model. With the binary we identified, a fixed passkey of "000000" equates *Passkey Entry* to the *Just Works* model (since *Just Works* uses an all-zero key).

10.5.3 User Tracking due to Fixed Addresses

We have discussed the different types of addresses that BLE Peripherals can use in their advertising messages in §2.2.2.1. A Peripheral would need to use *private addresses* if it is to prevent address tracking.

Table 10.3: Address types used in BLE peripherals.

Address Type	#Binaries	Address Type	#Binaries	Address Type	#Binaries
Public	29	Private nonresolvable	1	Unknown	4
Random static	208	Private resolvable	1		

Extracting advertising address type By default, Nordic uses a random static address for each chipset, which is set at the time of manufacture and does not change for the lifetime of the device. However, developers are able to modify this behaviour to choose a different address type via the `sd_ble_gap_address_set` (in older versions of the stack) and the `sd_ble_gap_addr_set` and `sd_ble_gap_privacy_set` calls (in newer versions).

`argXtract` extracted the arguments to these supervisor calls (where present) for all binaries within the dataset. 35 out of the 243 firmware files included one of the `svc` numbers for performing address type selection/setting, which meant that the remaining 208 files used the default setting of a random static address, set at the time of manufacture and unchanging throughout the lifetime of the device.

Table 10.3 depicts a breakdown of the address types used within the BLE Peripherals in our dataset. Out of the 243 binaries in our dataset, only a single binary used resolvable private addresses. Combining this with information regarding the device name, we found that the binary was related to a personal protection device. One binary within the dataset used non-resolvable addresses, which means it will not be vulnerable to tracking but will also not be able to form bonds with other devices. We could not deduce its functionality from its device name. We found that overall, the results indicated that at least 95% of the BLE binaries use static (random or public) addresses.

Privacy implications Because BLE Peripherals tend to advertise constantly when not in a connection, the use of public or random static addresses in advertising messages opens the BLE device, and by extension (depending on the device) its owner, to tracking. Further, as we discussed in §3.7, even private resolvable addresses may be vulnerable if the associated IRK is not sufficiently protected. In crowded locations such as shopping centres, repeated visits by a user can be covertly tracked simply by monitoring BLE advertisements and logging the device addresses. This has been previously demonstrated in [9]. It has also been shown to be feasible to set up a botnet to track users across a range of locations [65]. These attacks are particularly relevant in the case of devices such as wearables, which are generally always on the user's person. We found that all of the wearable binaries within our dataset use public or fixed random static addresses.

10.5.4 Manufacturer/Device Names and Privacy

BLE advertising messages usually contain the Peripheral's name, which is often used by users to identify a device from (potentially) a number of other BLE devices that are also advertising in the vicinity. Peripherals may also include a Manufacturer Name String, normally obtained via a scan request. These advertising messages require no authentication in order to be read.

Extracting device and manufacturer names A Nordic BLE device's name is set programmatically using the `sd_ble_gap_device_name_set` supervisor call, while the Manufacturer Name String is included within the Device Information Service (obtained in §10.5.1). `argXtract` extracted non-null values for at least one of device or manufacturer name from 156 binaries.

An analysis of the device names revealed that our dataset contained firmware from a variety of BLE devices, including wearable fitness trackers, beacons, electric switch controls, parking aids, security devices, personal protection devices, medical equipment and behavioural monitoring devices.

Privacy implications Device names can reveal a lot about the nature of the device. This is particularly concerning when the device is related to a user's health, or is of an otherwise private nature. Because no active connections are required to read advertising data, an attacker would simply need to monitor the BLE advertising channels and perhaps send a scan request for additional information. By continuously scanning BLE advertisements, extracting the device and manufacturer name, and combining this information with the Received Signal Strength Indicator (RSSI), along with user observation, an attacker may be able to determine which devices belong to which users in the vicinity. This could defeat private addresses, as an attacker might instead be able to use the device name, along with other advertising data, to track the device [10,46,68]. Further, if a particular device has known issues (such as those identified in this thesis), then the attacker can take advantage of the device name to identify exploitable devices.

10.6 Case Study II: BLE Security and Privacy (STMicroelectronics)

In this section we present a case study for the identification of BLE configuration vulnerabilities in firmware that targets STMicroelectronics chipsets, specifically the BlueNRG family of processors. Configurations for BlueNRG are performed via function calls. We therefore use the function pattern matching feature of `argXtract` (§10.3.4.2) to determine the location of the relevant function within the disassembly.

We manually analysed 500 real-world .bin files extracted from APKs and found that two were STMicroelectronics BlueNRG binaries. `argXtract` identified that both had an application code base of `0x10051000`, which corresponds to BlueNRG-1 v2.1+ [243].

10.6.1 BLE Address Privacy

In this section, we describe our tests against the BlueNRG binaries to identify the use of resolvable private addresses, which are enabled by the `aci_gap_init` function.

Extracting address configurations BlueNRG provides the `aci_hal_write_config_data` function for configuring public addresses [244]. This function takes three arguments: an offset, a length and a pointer to a byte array. In the case of address configuration, the offset is expected to be 0, the length is 6 and the byte array contains the address. This can be used to validate obtained outputs. Internally, this function calls `HAL_Write_ConfigData` with the same arguments.

The output memory contains the configured address at a binary-specific location. Privacy is configured separately via the `aci_gap_init` function. This takes three inputs: the BLE role of the device, an integer indicating whether privacy is enabled (i.e., whether resolvable addresses are used), and the length of the device name. The function performs several tasks, most of which require runtime information. However, we exploit the fact that the function adds the GAP service to the database and specify test sets that look for the service structure within the output memory contents.

Executing `argXtract` against the real-world binaries revealed that one contained a public address derived from the BlueNRG example address. This along with the binary's name led us to conclude that the binary was for demonstration purposes. The second binary was a BLE-enabled cyclist safety aid. It did not have privacy enabled.

Privacy implications A cyclist safety aid is likely to be about the user's person whenever they are cycling. A fixed address emanating from the device at all times enables the user to be tracked over time, as discussed in §10.5.3.

10.6.2 BLE Pairing Security

With BlueNRG binaries, if a BLE characteristic has authentication requirements, then specific configurations must be performed to enable pairing. We exploit two pairing-related functions in our tests. We additionally check for authorisation requirements.

Extracting pairing configurations BlueNRG requires two function calls in order to enable BLE security: the first is the `aci_gap_set_io_capability` to set the device's input-output capability, and the second is the `aci_gap_set_authentication_requirement` to set the pairing requirements (such as bonding, MitM protection, etc) [244]. Authorisation permissions are set using `aci_gap_set_authorization_requirement`. Focusing on the cyclist safety aid, `argXtract` found that the binary had no calls to `aci_gap_set_io_capability`, nor did it have any calls to `aci_gap_set_authorization_requirement`. This means that BLE security was not enabled.

Security implications The fact that no security was present on the cyclist safety aid means that an attacker could connect to the device and send commands to it without the need for any authentication, which could have serious consequences for the cyclist's safety. We have informed the developer regarding the identified issues, but have not received any response.

10.7 Limitations and Future Work

Edge cases `argXtract` is able to analyse most Cortex-M binaries. However, as seen in one example in §10.4, there are edge cases where the `.text` segment is split into subsections, with different address offsets for each subsection, where `argXtract` is unable to obtain individual code bases and accurate function estimates. This improvement is left as future work.

Function boundary identification `argXtract` assumes that the instructions belonging to a function are laid out in a contiguous range. If a function is split up into disjoint blocks of instructions, then `argXtract` may identify each such block as a separate function.

COI and callsite identification As mentioned in §10.3.4, the function pattern matching performed by `argXtract` uses manually-defined test sets, when function outputs or artefacts are distinguishable. If two functions produce the same output for the same input and one is not nested within the other, then a single function cannot be matched. Further, the function pattern matching process can take several hours when a binary contains a large number of functions. An ideal alternative would be *automated* function pattern matching, *without* executing function code. The most popular method to achieve this at present is via machine learning techniques. However, this requires a sufficiently large annotated training set for each function of interest, which is not yet available for the types of vendor-specific configuration functions that are of interest here. With callsite identification, direct calls are identified. However, calls via `blx` are not. (Note that `blx` will be identified and handled during tracing, but not for COI identification.)

Register tracing Assigning a value of 0 to register `r0` when a branch is not followed due to exceeding the *max_call_depth* (as described in §10.3.5) can give rise to inaccurate results if the value is actually supposed to be non-zero. We have not encountered such errors for our use cases, because the data used as arguments to our COIs did not go through several nested branches of processing. If it is known that COI arguments are likely to be highly-processed, then increasing the *max_call_depth* can reduce the likelihood of errors at the expense of (much) longer trace times.

Binary patching `argXtract` is able to extract arguments to security-relevant configuration APIs at present. A natural extension would be to perform patching of the binary if vulnerable configurations *are* identified. This may be fairly straightforward in cases where the relevant API call is present within the binary and the configuration only requires modification of simple input structures. However, composite input structures would require complex handling. Additional complexities could arise due to vendor-specific integrity checks and firmware signing.

10.8 Chapter Summary and Next Steps

We have described a framework for extracting security-relevant configuration information from stripped ARM Cortex-M binaries, overcoming the challenges that are normally encountered with stripped ARM analysis. We have applied our framework to 200+ real-world BLE firmware files from two different vendors, and have identified widespread lack of protection for BLE data as well as serious privacy issues. Despite some inherent limitations, we find that firmware analysis provides rich information regarding a BLE device's security configuration, which would otherwise have to have been acquired using expensive device tests.

This chapter concludes our discussion on BLE measurement studies. The final part of this thesis summarises our contributions and presents some recommendations to BLE stakeholders, based on observations we have made at different points within this thesis.

Part IV
Concluding Remarks

11 Conclusion

In this chapter, we revisit our research questions in light of our work. We also provide some recommendations to the various stakeholders within the BLE ecosystem, based on our observations regarding BLE vulnerabilities, to improve the overall security of the BLE ecosystem.

11.1 Research Objective and Questions

In §1.2, we specified the objective of this thesis, which was to advance the understanding of the existing state of security and privacy within the BLE ecosystem, with a special focus on the security of BLE data, and with the overarching goal of improving the security of BLE systems. We defined four research questions that arose from this objective. In this section, we revisit and answer those research questions, based on findings from our research.

RQ01: Are existing BLE deployments secure?
In order to answer this question, we have comprehensively surveyed existing security and privacy studies related to BLE (Chapters 3 and 4). This survey has revealed the presence of numerous vulnerabilities in real-world BLE systems, originating within the specification as well as individual implementations.

We have also developed a number of analysis tools, all of which are available open-source, which can be used to identify the presence of vulnerabilities in existing BLE deployments by analysing different information sources: `ATT-Profiler` (Chapter 9) directly interfaces with devices to determine minimum access requirements for BLE characteristics; `BLECryptracer` (Chapter 6) and `BLE-GUUIDE` (Chapter 8) analyse companion Android APKs to determine the presence/absence of app-layer security and to obtain indicators to firmware update vulnerabilities; `argXtract` (Chapter 10) analyses firmware for configuration vulnerabilities. Combined, these tools have identified numerous vulnerabilities within real-world BLE systems, including poorly protected BLE data, weak proprietary cryptographic algorithms, weakened pairing due to fixed passkeys, and lack of privacy controls due to static addresses.

These results lead to the conclusion that, at present, not all BLE systems are secure. The actual extent and impact of such vulnerabilities are discussed in RQ03.

RQ02: Does the lack of clearly-defined application-layer security mechanisms result in a lack of protection for BLE data?
We have analysed application-layer security in BLE and have identified an application-level unauthorised data access vulnerability for multi-application platforms such as Android, iOS, etc. (Chapter 5). Mitigating this vulnerability requires the implementation of end-to-end pro-

tection by application developers. However, we have found through a large scale analysis that a significant proportion of BLE devices likely do *not* implement such protection mechanisms (Chapter 6). We theorise that it is a lack of clarity regarding application-layer security (particularly in terms of the stakeholders who should be responsible for implementing it) that is the cause for such widespread lack of protection.

RQ03: If vulnerabilities exist, what is their extent and impact?
The vulnerability survey discussed in Chapter 4 specifies the applicability (at a high level) of each BLE vulnerability, which can be used to gauge the potential extent of a vulnerability. Vulnerabilities within the specification tend to be applicable to a large proportion of BLE deployments. Implementation vulnerabilities, on the other hand, are typically confined to a particular brand of device, and the exact extent depends on the specific vulnerability. When considering the potential *impact* of vulnerabilities, in Chapter 3 we describe attacks that the vulnerabilities may give rise to. Our attack analysis has identified the possibility of unauthorised data access and tampering, spoofing, tracking and denial of service.

Our measurement studies are more specific. These have identified the extent of a vulnerability in terms of proportion within a certain dataset, while the impact is determined based on the BLE functionality. For example, we have conducted a large-scale study of several thousand BLE-enabled mobile applications and have identified that around 50% of the tested APKs did not implement end-to-end protection for BLE data, implying that the data is vulnerable to unauthorised reads/writes by malicious applications (Chapter 6). To determine the potential impact, we have mapped APKs to BLE functionality using UUIDs and have identified potential for user health and PII leakage, as well as a vulnerable security-critical application (Chapter 8). Through a small-scale study conducted against six real-world devices, we have identified that four of the devices allowed for reading/subscribing all applicable characteristics and three also allowed writing all applicable characteristics with no security. However, we have noted that implementation-specific behaviour may be present, which may cause invalid results. The impact is directly gleaned from the type of device (Chapter 9). Finally, our firmware analysis against 240+ real-world BLE binaries has determined that 97% of the tested binaries do not incorporate link-layer security for readable and writable characteristics, 95% do not implement app-layer protection for readable characteristics and 59% do not implement app-layer protection for writable characteristics. The analysis has also identified that 97% of tested real-world binaries do not incorporate private addresses, leaving the devices vulnerable to tracking. The actual impact of these vulnerabilities is determined using the device functionality, as gleaned from the device name (which is also extracted from the firmware). For example, we have identified that several health/fitness devices and an HID device were vulnerable to unauthorised data access, and that all binaries identified as wearables were vulnerable to tracking (Chapter 10).

RQ04: How should vulnerabilities be mitigated and who is responsible for mitigation?
Within the vulnerability survey in Chapter 4, we have specified mitigation options (as identified by previous studies) and determined the responsible stakeholder for each vulnerability. Our

analysis shows that around 50% of all vulnerabilities are implementation-related, which means that mitigation in these cases lies in the hands of developers or platform vendors, while the rest require modifications to the specification (or combined specification-implementation changes).

In the case of the application-layer unauthorised data access vulnerability we have identified, we have delved deeper. The way in which the specification has been defined makes it difficult to assign a single responsible stakeholder for this vulnerability. We have therefore performed a pragmatic analysis instead, and have devised a solution that has the greatest likelihood of being adopted. We have determined through a multi-faceted stakeholder analysis that a specification-level modification which places a greater burden of change on platform devices is more likely to be implemented. We have also designed and implemented a proof-of-concept for such a solution, which we have verified to be backward compatible with existing devices and applications (Chapter 7).

11.2 Recommendations for Stakeholders

The security and privacy issues described in this thesis demonstrate that vulnerabilities are not confined to a single stakeholder (i.e., vulnerabilities have in different instances been due to the SIG, platform vendors and device developers). We therefore make recommendations for the following stakeholders in order to ensure a secure BLE ecosystem: (i) The Bluetooth SIG, (ii) Central platform vendors (such as Android, iOS), (iii) End product developers, (iv) Lawmakers. We also discuss the importance of a unified approach to security.

Bluetooth SIG The Bluetooth SIG should utilise standard, secure cryptographic algorithms in its security protocols. If possible, protocols should undergo formal security verification. Protocols with known issues should not be included within the specification. Similarly, mechanisms that reduce security (e.g., key entropy reduction) should not be an option. If some security aspect is not within scope of the BLE specification, then this must be clearly stated. We also recommend that the BLE-relevant sections of the Bluetooth specification be extracted and formulated into a standalone document; at present the specification document is at 3250+ pages, containing details of not only BLE but also Classic Bluetooth and Alternative MAC/PHY (AMP). This makes the specification difficult to follow, which could be the cause for some of the vulnerabilities. We further recommend that the SIG issue a *security summary*, containing security- and privacy-relevant aspects that vendors and developers need to keep in mind when designing and developing their products.

Central platform vendors When implementing the BLE specification, platform vendors should formulate a threat model for the platform and identify any risks that might arise from entities outside the BLE stack. If some elements are outside the scope of the specification, then platform vendors should apply custom security controls for those areas. Further, standard secure coding practices should be followed when developing the platform.

End product developers Developers should formulate a threat model for their specific use case. They should understand and implement the security and privacy features that are available within the BLE specification, and consider whether additional measures are required to cover areas that do not fall under the specification's scope. When developing device firmware or companion applications using SDKs, developers should understand and correctly utilise any security features provided by the SDKs, and bridge any gaps with additional protection mechanisms. When incorporating SDKs that include BLE features that are *not* required within the end product, developers should turn off such features to limit the attack surface. They should also follow standard secure coding practices.

Further, as we have observed in Chapters 6 and 10, most developers do not respond at all when informed of vulnerabilities in their products (in fact, *none* of the end product developers we contacted responded to our emails). Ideally, every product manufacturer should have a communication channel specifically for reporting security vulnerabilities (e.g., dedicated email address), and should be open to discussion with security researchers.

Lawmakers Many BLE devices handle health data, and we have observed that their companion mobile apps tend to be classified on application marketplaces under *Health & Fitness*. However, the devices are not considered to be *medical* and so do not have the stringent requirements regarding data protection that apply to medical devices. We recommend that nations modify the scope of privacy requirements to include *any* device that handles a user's health or other personal data, regardless of its intended use. Further, we recommend that security regulations be introduced for any device that has safety implications for its user.

We observe that bringing about a legal framework for BLE end product security and privacy will require cross-country cooperation since most consumers purchase devices from online marketplaces, which may source products from different countries and which may not be properly regulated.

A unified approach When analysing vulnerabilities and determining responsible stakeholders, perhaps of most concern are vulnerabilities that are attributed to more than one stakeholder or where a responsible stakeholder cannot be identified. This is because the absence of a single responsible entity can result in each stakeholder assuming that the fix will be applied by an entity other than themselves, ultimately resulting in no fix being applied at all. We recommend that major stakeholders cooperate to create a secure BLE design, where security gaps are not likely to occur. We note that many chipset and platform vendors are already members of the Bluetooth SIG, and are therefore well-placed to bring forth product-specific security considerations for discussion by the SIG.

Bibliography

[1] Bluetooth Special Interest Group, “2020 Bluetooth market update,” 2020. [Online]. Available: `https://www.bluetooth.com/bluetooth-resources/2020-bmu` [Accessed 13-May-2020].

[2] R. Heydon, *Bluetooth Low Energy: The Developer’s Handbook.* Prentice Hall, 2013.

[3] J.-R. Lin, T. Talty, and O. K. Tonguz, “On the potential of Bluetooth Low Energy technology for vehicular applications,” *IEEE Communications Magazine*, vol. 53, no. 1, pp. 267–275, 2015.

[4] M. Collotta and G. Pau, “A solution based on Bluetooth Low Energy for smart home energy management,” *Energies*, vol. 8, no. 10, pp. 11916–11938, 2015.

[5] R. Karani, S. Dhote, N. Khanduri, A. Srinivasan, R. Sawant, G. Gore, and J. Joshi, “Implementation and design issues for using Bluetooth Low Energy in passive keyless entry systems,” in *India Conference (INDICON), 2016 IEEE Annual*, pp. 1–6, IEEE, 2016.

[6] A. H. Omre and S. Keeping, “Bluetooth Low Energy: Wireless connectivity for medical monitoring,” *Journal of diabetes science and technology*, vol. 4, no. 2, pp. 457–463, 2010.

[7] S. Jasek, “Blue picking: Hacking Bluetooth smart locks,” in *HITBSec-Conf*, 2017.

[8] R. Idan, “Don’t give me a brake – Xiaomi scooter hack enables dangerous accelerations and stops for unsuspecting riders,” 2019. [Online]. Available: `https://blog.zimperium.com/dont-give-me-a-brake-xiaomi-scooter-hack-enables-dangerous-accelerations-and-stops-for-unsuspecting-riders/` [Accessed: 02 Apr 2021].

[9] A. K. Das, P. H. Pathak, C.-N. Chuah, and P. Mohapatra, “Uncovering privacy leakage in BLE network traffic of wearable fitness trackers,” in *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications*, pp. 99–104, ACM, 2016.

[10] K. Fawaz, K.-H. Kim, and K. G. Shin, “Protecting privacy of BLE device users,” in *USENIX Security Symposium*, pp. 1205–1221, USENIX Association, 2016.

[11] W. Albazrqaoe, “A study of Bluetooth frequency hopping sequence: Modeling and a practical attack,” Master’s thesis, 2011. Michigan State University. Computer Science.

[12] H. Perrey, O. Ugus, and D. Westhoff, "Security enhancement for Bluetooth Low Energy with Merkle's puzzle," *SIGMOBILE-Mobile Computing and Communications Review*, vol. 15, no. 3, p. 45, 2011.

[13] J. Padgette, K. Scarfone, and L. Chen, "Guide to Bluetooth security," *NIST Special Publication*, vol. 800, no. 121, 2012.

[14] C. Gomez, J. Oller, and J. Paradells, "Overview and evaluation of Bluetooth Low Energy: An emerging low-power wireless technology," *Sensors (Basel, Switzerland)*, vol. 12, no. 9, pp. 11734–11753, 2012.

[15] T. Rosa, "Bypassing passkey authentication in Bluetooth Low Energy.," *IACR Cryptology ePrint Archive*, vol. 2013, p. 309, 2013.

[16] M. Ryan, "Bluetooth: With low energy comes low security," in *7th USENIX Workshop on Offensive Technologies, WOOT '13, Washington, D.C., USA, August 13, 2013*, 2013.

[17] D. A. Ortiz-Yepes, "BALSA: Bluetooth Low Energy Application Layer Security Add-on," in *2015 International Workshop on Secure Internet of Things (SIoT)*, pp. 15–24, IEEE, 2015.

[18] J. Uher, R. G. Mennecke, and B. S. Farroha, "Denial of Sleep attacks in Bluetooth Low Energy wireless sensor networks," in *MILCOM 2016-2016 IEEE Military Communications Conference*, pp. 1231–1236, IEEE, 2016.

[19] S. Jasek, "Gattacking Bluetooth Smart devices," in *Black Hat USA Conference*, 2016.

[20] H. J. Tay, J. Tan, and P. Narasimhan, "A survey of security vulnerabilities in Bluetooth Low Energy beacons," tech. rep., 2016.

[21] S. Bräuer, A. Zubow, S. Zehl, M. Roshandel, and S. Mashhadi-Sohi, "On practical selective jamming of Bluetooth Low Energy advertising," in *2016 IEEE Conference on Standards for Communications and Networking, CSCN 2016, Berlin, Germany, October 31 - November 2, 2016*, pp. 158–163, 2016.

[22] P. Gullberg, "Denial of service attack on Bluetooth Low Energy," 2016.

[23] A. Hilts, C. Parsons, and J. Knockel, "Every step you fake: A comparative analysis of fitness tracker privacy and security," vol. 11, 2016.

[24] Bluetooth Special Interest Group, "2018 Bluetooth market update." [Online]. Available: `https://www.bluetooth.com/bluetooth-resources/2018-bluetooth-market-update` [Accessed 22-Apr-2021].

[25] M. Sinda, T. Danner, S. O'Neill, A. Alqurashi, and H.-K. Kim, "Improving the Bluetooth hopping sequence for better security in IoT devices," *International Journal of Software Innovation (IJSI)*, vol. 6, no. 4, pp. 117–131, 2018.

[26] S. Sarkar, J. Liu, and E. Jovanov, "A robust algorithm for sniffing BLE long-lived connections in real-time," in *2019 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6, IEEE, 2019.

[27] M. R. Ghori, T.-C. Wan, and G. C. Sodhy, "Bluetooth Low Energy mesh networks: Survey of communication and security protocols," *Sensors*, vol. 20, no. 12, p. 3590, 2020.

[28] M. Woolley, "Bluetooth 5," 2019.

[29] P. Sivakumaran and J. Blasco, "BLECryptracer," 2018. `https://github.com/projectbtle/BLECryptracer`.

[30] P. Sivakumaran and J. Blasco, "Proof-of-concept solution for unauthorised data access vulnerability on multi-app Bluetooth Low Energy platforms," 2020. `https://github.com/projectbtle/BLE-MultiApp-POC`.

[31] P. Sivakumaran and J. Blasco, "BLE-GUUIDE," 2020. `https://github.com/projectbtle/BLE-GUUIDE`.

[32] P. Sivakumaran and J. Blasco, "ATT Profiler," 2018. `https://github.com/projectbtle/att-profiler`.

[33] P. Sivakumaran and J. Blasco, "argXtract," 2020. `https://github.com/projectbtle/argXtract`.

[34] P. Sivakumaran and J. Blasco, "A study of the feasibility of co-located app attacks against BLE and a large-scale analysis of the current application-layer security landscape," in *28th USENIX Security Symposium (USENIX Security 19)*, (Santa Clara, CA), pp. 1–18, USENIX Association, Aug. 2019.

[35] P. Sivakumaran and J. Blasco, "A Low Energy Profile: Analysing characteristic security on BLE peripherals," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pp. 152–154, ACM, 2018.

[36] Bluetooth Special Interest Group, "Bluetooth core specification v5.2," 12 2019.

[37] N. Ahmed, R. A. Michelin, W. Xue, S. Ruj, R. Malaney, S. S. Kanhere, A. Seneviratne, W. Hu, H. Janicke, and S. K. Jha, "A survey of Covid-19 contact tracing apps," *IEEE Access*, vol. 8, pp. 134577–134601, 2020.

[38] J. Padgette, K. Scarfone, and L. Chen, "Guide to Bluetooth security, revision 2," *NIST Special Publication*, vol. 800, no. 121, 2017.

[39] P. Cope, J. Campbell, and T. Hayajneh, "An investigation of Bluetooth security vulnerabilities," in *Computing and Communication Workshop and Conference (CCWC), 2017 IEEE 7th Annual*, pp. 1–7, IEEE, 2017.

[40] A. Lonzetta, P. Cope, J. Campbell, B. Mohd, and T. Hayajneh, “Security vulnerabilities in Bluetooth technology as used in IoT,” *Journal of Sensor and Actuator Networks*, vol. 7, no. 3, p. 28, 2018.

[41] C. Kolias, L. Copi, F. Zhang, and A. Stavrou, “Breaking BLE beacons for fun but mostly profit,” in *Proceedings of the 10th European Workshop on Systems Security*, EuroSec’17, (New York, NY, USA), Association for Computing Machinery, 2017.

[42] M. Ley, “DBLP,” *dblp. uni-trier. de*, 2005.

[43] G. Gu, “Computer security conference ranking and statistic,” 2015. [Online]. Available: `http://faculty.cs.tamu.edu/guofei/sec_conf_stat.htm` [Accessed 25-Nov-2019].

[44] MITRE Corporation, “CVE® List,” 2021.

[45] G. Celosia and M. Cunche, “Discontinued privacy: Personal data leaks in Apple Bluetooth Low Energy Continuity Protocols,” *Proc. Priv. Enhancing Technol.*, vol. 2020, no. 1, pp. 26–46, 2020.

[46] G. Celosia and M. Cunche, “Fingerprinting Bluetooth Low Energy devices based on the Generic Attribute Profile,” in *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, pp. 24–31, ACM, 2019.

[47] J. Wang, F. Hu, Y. Zhou, Y. Liu, H. Zhang, and Z. Liu, “BlueDoor: Breaking the secure information flow via BLE vulnerability,” in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pp. 286–298, 2020.

[48] M. Naveed, X. Zhou, S. Demetriou, X. Wang, and C. A. Gunter, “Inside job: Understanding and mitigating the threat of external device mis-binding on Android,” in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[49] E. Biham and L. Neumann, “Breaking the Bluetooth pairing - the fixed coordinate invalid curve attack,” in *International Conference on Selected Areas in Cryptography*, pp. 250–273, Springer, 2019.

[50] A. Y. Lindell, “Attacks on the pairing protocol of Bluetooth v2. 1,” *Black Hat USA, Las Vegas, Nevada*, 2008.

[51] T. Claverie and J. Lopes-Esteves, “Testing for weak key management in Bluetooth Low Energy implementations,” 2020.

[52] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen, “Key negotiation downgrade attacks on Bluetooth and Bluetooth Low Energy,” *ACM Trans. Priv. Secur.*, vol. 23, June 2020.

[53] A. C. Santos, J. L. Soares Filho, Á. Í. Silva, V. Nigam, and I. E. Fonseca, “BLE injection-free attack: A novel attack on Bluetooth Low Energy devices,” *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–11, 2019.

[54] J. Wu, Y. Nan, V. Kumar, D. J. Tian, A. Bianchi, M. Payer, and D. Xu, “BLESA: Spoofing attacks against reconnections in Bluetooth Low Energy,” in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[55] Y. Zhang, J. Weng, R. Dey, Y. Jin, Z. Lin, and X. Fu, “Breaking secure pairing of Bluetooth Low Energy using downgrade attacks,” in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 37–54, 2020.

[56] M. von Tschirschnitz, L. Peuckert, F. Franzen, and J. Grossklags, “Method confusion attack on Bluetooth pairing,” in *2021 IEEE Symposium on Security and Privacy (S&P)*, pp. 213–228, 2021.

[57] D. Antonioli, N. O. Tippenhauer, K. Rasmussen, and M. Payer, “BLURtooth: Exploiting cross-transport key derivation in Bluetooth Classic and Bluetooth Low Energy,” *arXiv preprint arXiv:2009.11776*, 2020.

[58] M. E. Garbelini, C. Wang, S. Chattopadhyay, S. Sun, and E. Kurniawan, “SweynTooth: Unleashing mayhem over Bluetooth Low Energy,” in *USENIX Annual Technical Conference (USENIX ATC)*, 2020.

[59] B. Seri, G. Vishnepolsky, and D. Zusman, “BLEEDINGBIT: The hidden attack surface within BLE chips,” 2018. [Online]. Available: `https://go.armis.com/bleedingbit`. [Accessed: 01-Dec-2018].

[60] J. S. Ruge, “Dynamic Bluetooth firmware analysis,” Master’s thesis, 2019.

[61] Z. Guo, I. G. Harris, Y. Jiang, and L.-f. Tsaur, “An efficient approach to prevent battery exhaustion attack on BLE-based mesh networks,” in *2017 International Conference on Computing, Networking and Communications (ICNC)*, pp. 1–5, IEEE, 2017.

[62] C. Hensler and P. Tague, “Using Bluetooth Low Energy spoofing to dispute device details,” in *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, pp. 340–342, ACM, 2019.

[63] A. Becker, “Bluetooth security & hacks,” July 2007. [Online]. Available: `https://www.emsec.rub.de/media/crypto/attachments/files/2011/04/slides_bluetooth_security_and_hacks.pdf` [Accessed 26-July-2017].

[64] A. Korolova and V. Sharma, “Cross-app tracking via nearby Bluetooth Low Energy devices,” in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY 2018, Tempe, AZ, USA, March 19-21, 2018*, pp. 43–52, 2018.

[65] T. Issoufaly and P. U. Tournoux, “BLEB: Bluetooth Low Energy Botnet for large scale individual tracking,” in *2017 1st International Conference on Next Generation Computing Applications (NextComp)*, pp. 115–120, IEEE, 2017.

[66] C. Zuo, H. Wen, Z. Lin, and Y. Zhang, “Automatic fingerprinting of vulnerable BLE IoT devices with static UUIDs from mobile apps,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1469–1483, ACM, 2019.

[67] J. Mussared, “Privacy issues discovered in the BLE implementation of the COVIDSafe Android app (ref: CVE-2020-12860),” 2020.

[68] J. K. Becker, D. Li, and D. Starobinski, “Tracking anonymized Bluetooth devices,” *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 3, pp. 50–65, 2019.

[69] G. Celosia and M. Cunche, “Saving private addresses: An analysis of privacy issues in the Bluetooth Low Energy advertising mechanism,” in *Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, pp. 444–453, 2019.

[70] D.-Z. Sun, L. Sun, and Y. Yang, “On Secure Simple Pairing in Bluetooth standard v5.0-part II: Privacy analysis and enhancement for Low Energy,” *Sensors*, vol. 19, no. 15, p. 3259, 2019.

[71] M. M. Lap Nagra, “How to change the MAC address of Bluetooth dongle in Ubuntu,” 2012. [Online]. Available: `https://stackoverflow.com/revisions/9213406/2` [Accessed 24-Jan-2021].

[72] G. Legg, “The Bluejacking, Bluesnarfing, Bluebugging blues: Bluetooth faces perception of vulnerability,” 2005. `https://www.eetimes.com/the-bluejacking-bluesnarfing-bluebugging-blues-bluetooth-faces-perception-of-vulnerability/`.

[73] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen, “BIAS: Bluetooth Impersonation AttackS,” in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[74] B. Seri and G. Vishnepolsky, “BlueBorne,” 2017. [Online]. Available: `https://www.armis.com/blueborne/`.

[75] M. Stanislav and T. Beardsley, “Hacking IoT: A case study on baby monitor exposures and vulnerabilities,” 2015. [Online]. Available: `https://www.rapid7.com/globalassets/external/docs/Hacking-IoT-A-Case-Study-on-Baby-Monitor-Exposures-and-Vulnerabilities.pdf`. [Accessed: 11 June 2020].

[76] S. Larson, “FDA confirms that St. Jude’s cardiac devices can be hacked,” 2017. [Online]. Available: `https://money.cnn.com/2017/01/09/technology/fda-st-jude-cardiac-hack`. [Accessed: 11 June 2020].

[77] I. Thomson, “Wi-Fi baby heart monitor may have the worst IoT security of 2016,” 2016. [Online]. Available: https://www.theregister.com/2016/10/13/possibly_worst_iot_security_failure_yet. [Accessed: 11 June 2020].

[78] European Union Agency for Network and Information Security, “Study on cryptographic protocols,” Nov 2014. [Online]. Available: https://www.enisa.europa.eu/publications/study-on-cryptographic-protocols [Accessed: 08 Mar 2021].

[79] M. Ryan, “Ubertooth,” 2020. https://github.com/greatscottgadgets/ubertooth.

[80] Nordic Semiconductor, “nRF Sniffer for Bluetooth LE,” 2020. [Online]. Available: https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Sniffer-for-Bluetooth-LE. [Accessed: 08 Mar 2020].

[81] D. Cauquil, “BtleJack: A new Bluetooth Low Energy Swiss-army knife,” 2018. https://github.com/virtualabs/btlejack.

[82] V. Parmar, “CC2564C: CC256x and WL18xx Bluetooth Low Energy - LE scan vulnerability,” 2016. [Online]. Available: https://e2e.ti.com/support/wireless-connectivity/bluetooth/f/538/t/856161. [Accessed: 18 Jan 2021].

[83] M. Garbelini *et al.*, “SweynTooth - unleashing mayhem over Bluetooth Low Energy,” 2020. https://github.com/Matheus-Garbelini/sweyntooth_bluetooth_low_energy_attacks.

[84] MITRE Corporation, “CVE-2019-2032.” Available from MITRE, CVE-ID CVE-2019-2032, 2019.

[85] Bluetooth Special Interest Group, “Bluetooth core specification v5.0,” 12 2016.

[86] M. Ryan, “Crackle, crack Bluetooth Smart (BLE) encryption,” 2013. [Online]. Available: https://lacklustre.net/projects/crackle [Accessed 02-July-2017].

[87] Apple Inc., “About the security content of iOS 8.1 (ref: CVE-2014-4428),” 2017.

[88] A. Pahwa, “Reverse engineering IoT devices (ref: CVE-2017-18642),” 2017.

[89] D. Su and A. Fletcher, “BlueSteal: Popping GATT safes (ref: CVE-2017-17436),” 2017.

[90] V. Casares, “Mimo Baby hack (ref: CVE-2018-10825),” 2018.

[91] H. Wen, Z. Lin, and Y. Zhang, “FirmXRay,” 2020. https://github.com/OSUSecLab/FirmXRay.

[92] H. Wen, Z. Lin, and Y. Zhang, “Firmxray: Detecting Bluetooth link layer vulnerabilities from bare-metal firmware,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 167–180, 2020.

[93] G. Celosia and M. Cunche, “Valkyrie,” 2020. https://github.com/gcelosia/valkyrie.

[94] G. Celosia and M. Cunche, “Valkyrie: A generic framework for verifying privacy provisions in wireless networks,” in *13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2020.

[95] G. Celosia and M. Cunche, “Venom: A visual and experimental Bluetooth Low Energy tracking system,” in *13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2020.

[96] M. Yaseen, W. Iqbal, I. Rashid, H. Abbas, M. Mohsin, K. Saleem, and Y. A. Bangash, “MARC: A novel framework for detecting MITM attacks in eHealthcare BLE systems,” *Journal of Medical Systems*, vol. 43, no. 11, p. 324, 2019.

[97] J. Wu, Y. Nan, V. Kumar, M. Payer, and D. Xu, “BlueShield: Detecting spoofing attacks in Bluetooth Low Energy networks,” in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pp. 397–411, 2020.

[98] S. Jasek, “GATTacker - a node.js package for BLE (Bluetooth Low Energy) security assessment using Man-in-the-Middle and other attacks,” 2016. https://github.com/securing/gattacker.

[99] J. Wu, Y. Nan, V. Kumar, M. Payer, and D. Xu, “BlueShield,” 2020. https://github.com/allenjlw/BlueShield.

[100] Y. Zhang, J. Weng, Z. Ling, B. Pearson, and X. Fu, “BLESS: A BLE application security scanning framework,” in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pp. 636–645, IEEE, 2020.

[101] D. Mantz, J. Classen, M. Schulz, and M. Hollick, “InternalBlue-Bluetooth binary patching and experimentation framework,” in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 79–90, 2019.

[102] European Telecommunications Standards Institute, “Cyber security for consumer Internet of Things: Baseline requirements.” [Online]. Available: https://www.etsi.org/deliver/etsi_en/303600_303699/303645/02.01.01_60/en_303645v020101p.pdf [Accessed 01 May 2021].

[103] Department for Digital, Culture, Media & Sport, “Government response to the call for views on consumer connected product cyber security legislation.” [Online]. Available: https://www.gov.uk/government/publications/regulating-consumer-smart-product-cyber-security-government-response/government-response-to-the-call-for-views-on-consumer-connected-product-cyber-security-legislation [Accessed 01 May 2021].

[104] "Bluetooth," June 2021. [Online]. Available: `https://source.android.com/devices/bluetooth`. [Accessed: 10-Jul-2021].

[105] Nordic Semiconductor, "BLE on Android v1.0.1," 2016. [Online]. Available: `https://devzone.nordicsemi.com/attachment/bdd561ff56924e10ea78057b91c5c642`. [Accessed: 05-Feb-2018].

[106] "Bluetooth Low Energy overview," Sept 2020. [Online]. Available: `https://developer.android.com/guide/topics/connectivity/bluetooth-le`. [Accessed: 06-Feb-2021].

[107] Bluetooth Special Interest Group, "Heart Rate Profile: Bluetooth profile specification v1.0," 07 2011.

[108] Apple Inc., "scanForPeripherals." [Online]. Available: `https://developer.apple.com/documentation/corebluetooth/cbcentralmanager/1518986-scanforperipherals`. [Accessed: 08-Mar-2021].

[109] Apple, Inc., "If an app would like to use Bluetooth on your device," Sept 2019. [Online]. Available: `https://support.apple.com/en-us/HT210578`. [Accessed: 20-Oct-2020].

[110] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of Android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, pp. 468–471, ACM, 2016.

[111] A. Desnos *et al.*, "Androguard: Reverse engineering, malware and goodware analysis of Android applications ... and more (ninja !)." `https://github.com/androguard/androguard`.

[112] "Java Cryptography Architecture (JCA) Reference Guide." [Online]. Available: `https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html`. [Accessed: 18-July-2018].

[113] "Security tips," June 2018. [Online]. Available: `https://developer.android.com/training/articles/security-tips`. [Accessed: 18-July-2018].

[114] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[115] F. Wei, S. Roy, X. Ou, *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1329–1341, ACM, 2014.

[116] F. Pauck, E. Bodden, and H. Wehrheim, "Do Android taint analysis tools keep their promises?," *arXiv preprint arXiv:1804.02903*, 2018.

[117] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in Android applications," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 73–84, ACM, 2013.

[118] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth, "Slicing Droids: Program slicing for smali code," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pp. 1844–1851, ACM, 2013.

[119] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! Analyzing unsafe and malicious dynamic code loading in Android applications.," in *NDSS*, vol. 14, pp. 23–26, 2014.

[120] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding Android obfuscation techniques: A large-scale investigation in the wild," in *International Conference on Security and Privacy in Communication Systems*, pp. 172–192, Springer, 2018.

[121] C. Fritz, S. Arzt, and S. Rasthofer, "Droidbench: A micro-benchmark suite to assess the stability of taint-analysis tools for Android." `https://github.com/secure-software-engineering/DroidBench`.

[122] X. Guo, Y. Yin, C. Dong, G. Yang, and G. Zhou, "On the class imbalance problem," in *Natural Computation, 2008. ICNC'08. Fourth International Conference on*, vol. 4, pp. 192–201, IEEE, 2008.

[123] L. A. Jeni, J. F. Cohn, and F. De La Torre, "Facing imbalanced data–recommendations for the use of performance metrics," in *Affective Computing and Intelligent Interaction (ACII), 2013 Humaine Association Conference on*, pp. 245–251, IEEE, 2013.

[124] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 176–186, ACM, 2018.

[125] Nordic Semiconductor, "Quarterly presentation Q1 2020." [Online]. Available: `https://www.nordicsemi.com/-/media/Investor-Relations-and-QA/Quarterly-Presentations/2020/Quarterly-presentation-Q1-2020.pdf` [Accessed: 03 July 2020].

[126] S. Krüger, J. Späth, *et al.*, "CogniCrypt_SAST: CrySL-to-static analysis compiler." `https://github.com/CROSSINGTUD/CryptoAnalysis/`.

[127] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, *et al.*, "CogniCrypt: Supporting developers in using cryptography," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 931–936, IEEE Press, 2017.

[128] U. du Luxembourg, "Lists of APKs." [Online]. Available: `https://androzoo.uni.lu/lists`. [Accessed: 12-Nov-2018].

[129] IDC, "Worldwide wearables market grows 7.3% in Q3 2017 as smart wearables rise and basic wearables decline, says IDC," 2017. [Online]. Available: `https://github.com/secure-software-engineering/DroidBench` [Accessed 16-Feb-2017].

[130] statcounter, "Operating system market share worldwide." [Online]. Available: `https://gs.statcounter.com/os-market-share`. [Accessed: 06-Feb-2021].

[131] Markets and Markets, "IoT chip market," 2021. [Online]. Available: `https://www.marketsandmarkets.com/Market-Reports/iot-chip-market-236473142.html` [Accessed 08-Feb-2021].

[132] Bluetooth Special Interest Group, "LaunchStudio:Listings," 2021. [Online]. Available: `https://launchstudio.bluetooth.com/Listings/Search` [Accessed 08-Feb-2021].

[133] H. Assal and S. Chiasson, "'Think secure from the beginning' A survey with software developers," in *Proceedings of the 2019 CHI conference on human factors in computing systems*, pp. 1–13, 2019.

[134] J. A. Halderman, "To strengthen security, change developers' incentives," *IEEE Security & Privacy*, vol. 8, no. 2, pp. 79–82, 2010.

[135] D. van der Linden, P. Anthonysamy, B. Nuseibeh, T. T. Tun, M. Petre, M. Levine, J. Towse, and A. Rashid, "Schrödinger's security: Opening the box on app developers' security rationale," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 149–160, IEEE, 2020.

[136] Bluetooth Special Interest Group, "Reporting security vulnerabilities." [Online]. Available: `https://www.bluetooth.com/learn-about-bluetooth/key-attributes/bluetooth-security/reporting-security`. [Accessed: 12-Feb-2021].

[137] Android, "Android security rewards program rules." [Online]. Available: `https://www.google.com/about/appsecurity/android-rewards/`. [Accessed: 12-Feb-2021].

[138] Apple Inc., "Report a security or privacy vulnerability." [Online]. Available: `https://support.apple.com/en-gb/HT201220`. [Accessed: 12-Feb-2021].

[139] Texas Instruments, "Report potential product security vulnerabilities." [Online]. Available: `https://www.ti.com/technologies/security/report-product-security-vulnerabilities.html`. [Accessed: 12-Feb-2021].

[140] Nordic Semiconductor, "Product security vulnerabilities and how to report them." [Online]. Available: `https://www.nordicsemi.com/About-us/PSIRT`. [Accessed: 12-Feb-2021].

[141] L. O'Donnell, "Consumers urged to junk insecure IoT devices." [Online]. Available: https://threatpost.com/consumers-urged-to-junk-insecure-iot-devices/145800/. [Accessed: 12-Feb-2021].

[142] R. Ramachandran, "IoT connected healthcare devices: Challenges in cybersecurity and the way forward," 2020.

[143] W. Schwartau, "Let's end pass-the-buck security," 2004.

[144] NXP, "QN902x OTA profile guide," 2018. [Online]. Available: https://www.nxp.com/docs/en/user-guide/UM10993.pdf [Accessed 07 Feb 2020].

[145] "Buttonless secure DFU service," 2017. [Online]. Available: https://infocenter.nordicsemi.com/topic/com.nordic.infocenter.sdk5.v14.0.0/service_dfu.html. [Accessed: 01 May 2019].

[146] "BLE external memory bootloader and bootloadable," 2015. [Online]. Available: http://www.cypress.com/file/228556/download. [Accessed: 21 Dec 2018].

[147] Qualcomm Technologies, "OTAU CSR102x," 2016. [Online]. Available: https://developer.qualcomm.com/qfile/34081/csr102x_otau_overview.pdf [Accessed 09 Aug 2019].

[148] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli, "Secure firmware updates for constrained IoT devices using open standards: A reality check," *IEEE Access*, vol. 7, pp. 71907–71920, 2019.

[149] "Set the application ID," Oct 2020. [Online]. Available: https://developer.android.com/studio/build/application-id. [Accessed: 20-Oct-2020].

[150] N. Momen, S. Bock, and L. Fritsch, "Accept-maybe-decline: Introducing partial consent for the permission-based access control model of Android," in *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*, pp. 71–80, 2020.

[151] J. Nielsen, "Response times: The 3 important limits," 1993.

[152] L. C. Hogan, "Performance is user experience," 2014.

[153] K. Micinski, D. Votipka, R. Stevens, N. Kofinas, M. L. Mazurek, and J. S. Foster, "User interactions and permission use on Android," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pp. 362–373, 2017.

[154] A. Heinrich, M. Stute, and M. Hollick, "BTLEmap: Nmap for Bluetooth Low Energy," in *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pp. 331–333, 2020.

[155] Y. Meidan, M. Bohadana, A. Shabtai, J. D. Guarnizo, M. Ochoa, N. O. Tippenhauer, and Y. Elovici, "ProfilIoT: A machine learning approach for IoT device identification based on network traffic analysis," in *Proceedings of the Symposium on Applied Computing*, SAC '17, (New York, NY, USA), p. 506–509, Association for Computing Machinery, 2017.

[156] M. Miettinen, S. Marchal, I. Hafeez, N. Asokan, A.-R. Sadeghi, and S. Tarkoma, "IoT sentinel: Automated device-type identification for security enforcement in IoT," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 2177–2184, IEEE, 2017.

[157] A. Aksoy and M. H. Gunes, "Automated IoT device identification using network traffic," in *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pp. 1–7, IEEE, 2019.

[158] J. Kotak and Y. Elovici, "IoT device identification using deep learning," in *Conference on Complex, Intelligent, and Software Intensive Systems*, pp. 76–86, Springer, 2020.

[159] S. Marchal, M. Miettinen, T. D. Nguyen, A.-R. Sadeghi, and N. Asokan, "Audi: Toward autonomous IoT device-type identification using periodic communication," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1402–1412, 2019.

[160] K. Townsend, C. Cufí, R. Davidson, *et al.*, *Getting started with Bluetooth Low Energy: tools and techniques for low-power networking.* O'Reilly Media, Inc., 2014.

[161] R. Heydon, "An introduction to Bluetooth Low Energy," 2016. [Online]. Available: `https://datatracker.ietf.org/meeting/interim-2016-t2trg-02/materials/slides-interim-2016-t2trg-2-7`. [Accessed: 18 Feb 2020].

[162] Bluetooth Special Interest Group, "16 bit UUIDs for members." [Online]. Available: `https://www.bluetooth.com/specifications/assigned-numbers/16-bit-uuids-for-members/` [Accessed 28 May 2020].

[163] G. A. Miller, "WordNet: A lexical database for English," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[164] "Device firmware update service," 2017. [Online]. Available: `https://infocenter.nordicsemi.com/topic/com.nordic.infocenter.sdk5.v11.0.0/group__ble__sdk__srv__dfu.html`. [Accessed: 01 May 2019].

[165] "OAD profile," 2016. [Online]. Available: `http://dev.ti.com/tirex/content/simplelink_cc2640r2_sdk_1_40_00_45/docs/blestack/ble_user_guide/html/oad-ble-stack-3.x/oad_profile.html`. [Accessed: 01 May 2019].

[166] "Over-the-Air download code example," 2017. [Online]. Available: `http://dev.ti.com/tirex/content/simplelink_msp432_sdk_bluetooth_plugin_1_20_00_42/examples/`

rtos/MSP_EXP432P401R/bluetooth/oad_firmware_update/README.html. [Accessed: 14 July 2019].

[167] "Consider blocklisting Qualcomm CSR firmware update service." [Online]. Available: https://github.com/WebBluetoothCG/registries/issues/20. [Accessed: 01 May 2019].

[168] Silicon Labs, "AN1086: Using the gecko bootloader with the Silicon Labs Bluetooth applications," 2018. [Online]. Available: https://www.silabs.com/documents/public/application-notes/an1086-gecko-bootloader-bluetooth.pdf [Accessed 04 Feb 2020].

[169] ST Microelectronics, "BlueSTSDK," 2019. [Online]. Available: https://github.com/STMicroelectronics/BlueSTSDK_GUI_iOS [Accessed 03 Feb 2020].

[170] ST Microelectronics, "AN4869 application note," 2016. [Online]. Available: https://www.st.com/resource/en/application_note/dm00293821.pdf [Accessed 01 May 2019].

[171] Bluetooth Special Interest Group, "LE Audio." [Online]. Available: https://www.bluetooth.com/learn-about-bluetooth/bluetooth-technology/le-audio/ [Accessed 23 Feb 2020].

[172] Nordic Semiconductor ASA, "nRF Toolbox." https://github.com/NordicSemiconductor/Android-nRF-Toolbox.

[173] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack overflow considered harmful? The impact of copy&paste on Android application security," in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 121–136, IEEE, 2017.

[174] L. Whitney, "How to locate your friends with the Apple 'Find My' app," Nov 2019. [Online]. Available: https://uk.pcmag.com/gallery/123522/how-to-locate-your-friends-with-the-apple-find-my-app. [Accessed: 03 Mar 2020].

[175] K. Haataja, K. Hyppönen, S. Pasanen, and P. Toivanen, *Bluetooth security attacks: comparative analysis, attacks, and countermeasures*. Springer Science & Business Media, 2013.

[176] L. Cameron Booth and M. Mayrany, "IoT penetration testing: Hacking an electric scooter," 2019.

[177] J. Classen, D. Wegemer, P. Patras, T. Spink, and M. Hollick, "Anatomy of a vulnerable fitness tracking system: Dissecting the Fitbit cloud, app, and firmware," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 2, no. 1, pp. 1–24, 2018.

[178] B. Cyr, W. Horn, D. Miao, and M. Specter, "Security analysis of wearable fitness devices (Fitbit)," *Massachusetts Institute of Technology*, 2014.

[179] S. Mistry, "noble: A Node.js BLE (Bluetooth Low Energy) central module," 2018. https://github.com/noble/noble.

[180] D. Malone and K. Mahern, "Investigating the distribution of password choices," 2011. Retrieved from https://arxiv.org/pdf/1104.3722.pdf.

[181] New Jersey Cybersecurity and Communications Integration Cell, "Mirai: NJCCIC threat profile," 2016. [Online]. Available: https://www.cyber.nj.gov/threat-center/threat-profiles/botnet-variants/mirai-botnet. [Accessed: 11 June 2020].

[182] B. Krebs, "KrebsOnSecurity hit with record DDoS," 2016. [Online]. Available: https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/. [Accessed: 11 June 2020].

[183] R. Chirgwin, "Finns chilling as DDoS knocks out building control system," 2016. [Online]. Available: https://www.theregister.com/2016/11/09/finns_chilling_as_ddos_knocks_out_building_control_system/. [Accessed: 11 June 2020].

[184] Radware, "'BrickerBot' results in PDoS attack," 2006. [Online]. Available: https://security.radware.com/ddos-threats-attacks/brickerbot-pdos-permanent-denial-of-service/. [Accessed: 11 June 2020].

[185] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, *et al.*, "Understanding the Mirai botnet," in *26th USENIX security symposium (USENIX Security 17)*, pp. 1093–1110, 2017.

[186] Mitre, "CVE-2015-2880." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2880 [Accessed: 14 July 2020].

[187] Mitre, "CVE-2019-16518." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16518 [Accessed: 14 July 2020].

[188] Mitre, "CVE-2018-10825." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-10825 [Accessed: 14 July 2020].

[189] K. Zetter, "Hackers can seize control of electric skateboards and toss riders." [Online]. Available: https://www.wired.com/2015/08/hackers-can-seize-control-of-electric-skateboards-and-toss-riders-boosted-revo/ [Accessed: 27 July 2020].

[190] A. Qasem, P. Shirani, M. Debbabi, L. Wang, B. Lebel, and B. L. Agba, "Automatic vulnerability detection in embedded devices and firmware: Survey and layered taxonomies," *ACM Computing Surveys (CSUR)*, vol. 54, no. 2, pp. 1–42, 2021.

[191] M. Prasad and T.-c. Chiueh, "A binary rewriting defense against stack based buffer overflow attacks," in *USENIX Annual Technical Conference, General Track*, pp. 211–224, 2003.

[192] L. C. Harris and B. P. Miller, “Practical analysis of stripped binary code,” *ACM SIGARCH Computer Architecture News*, vol. 33, no. 5, pp. 63–68, 2005.

[193] G. Ravipati, A. R. Bernat, N. Rosenblum, B. P. Miller, and J. K. Hollingsworth, “Toward the deconstruction of Dyninst,” *Univ. of Wisconsin, technical report*, p. 32, 2007.

[194] R. Qiao and R. Sekar, “Function interface analysis: A principled approach for function recognition in COTS binaries,” in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 201–212, IEEE, 2017.

[195] D. Andriesse, A. Slowinska, and H. Bos, “Compiler-agnostic function detection in binaries,” in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 177–189, IEEE, 2017.

[196] A. Di Federico, M. Payer, and G. Agosta, “rev. ng: a unified binary analysis framework to recover CFGs and function boundaries,” in *Proceedings of the 26th International Conference on Compiler Construction*, pp. 131–141, 2017.

[197] N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt, “Learning to analyze binary computer code,” in *AAAI*, pp. 798–804, 2008.

[198] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, “BYTEWEIGHT: Learning to recognize functions in binary code,” in *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 845–860, 2014.

[199] E. C. R. Shin, D. Song, and R. Moazzezi, “Recognizing functions in binaries with neural networks,” in *24th USENIX Security Symposium (USENIX Security 15)*, pp. 611–626, 2015.

[200] J. Alves-Foss and J. Song, “Function boundary detection in stripped binaries,” in *Proceedings of the 35th Annual Computer Security Applications Conference*, pp. 84–96, 2019.

[201] G. Myles and C. Collberg, “K-gram based software birthmarks,” in *Proceedings of the 2005 ACM symposium on Applied computing*, pp. 314–318, 2005.

[202] Y. R. Lee, B. Kang, and E. G. Im, “Function matching-based binary-level software similarity calculation,” in *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, pp. 322–327, 2013.

[203] M. Bourquin, A. King, and E. Robbins, “Binslayer: accurate comparison of binary executables,” in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, pp. 1–10, 2013.

[204] M. Egele, M. Woo, P. Chapman, and D. Brumley, “Blanket execution: Dynamic similarity testing for program binaries and components,” in *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 303–317, 2014.

[205] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 363–376, 2017.

[206] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "SAFE: Self-Attentive Function Embeddings for binary similarity," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 309–329, Springer, 2019.

[207] J. Patrick-Evans, L. Cavallaro, and J. Kinder, "Probabilistic naming of functions in stripped binaries," in *Annual Computer Security Applications Conference*, pp. 373–385, 2020.

[208] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 95–110, 2014.

[209] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: A case study on embedded web interfaces," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pp. 437–448, 2016.

[210] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *22nd USENIX Security Symposium (USENIX Security 13)*, pp. 463–478, 2013.

[211] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware.," in *NDSS*, 2015.

[212] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, "FirmFuzz: Automated IoT firmware introspection and analysis," in *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, pp. 15–21, 2019.

[213] J. Wu, R. Wu, D. Antonioli, M. Payer, N. O. Tippenhauer, D. Xu, D. J. Tian, and A. Bianchi, "LIGHTBLUE: Automatic profile-aware debloating of Bluetooth stacks," in *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2021.

[214] J. Tillmanns, J. Classen, F. Rohrbach, and M. Hollick, "Firmware insider: Bluetooth randomness is mostly random," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[215] Bluetooth SIG, "Intro to Bluetooth Low Energy." [Online]. Available: `https://www.bluetooth.com/bluetooth-resources/intro-to-bluetooth-low-energy/` [Accessed: 27 July 2020].

[216] Zigbee Alliance, "What is Zigbee?." [Online]. Available: `https://zigbeealliance.org/solution/zigbee/` [Accessed: 27 July 2020].

[217] Garmin Canada Inc., "What is ANT+." [Online]. Available: https://www.thisisant.com/consumer/ant-101/what-is-ant [Accessed: 27 July 2020].

[218] Thread Group, "What is Thread." [Online]. Available: https://www.threadgroup.org/what-Is-thread [Accessed: 27 July 2020].

[219] Nordic Semiconductor ASA, "SoftDevices." [Online]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fug_gsg_ses%2FUG%2Fgsg%2Fsoftdevices.html [Accessed: 03 July 2020].

[220] Texas Instruments, "A fully compliant ZigBee 3.x solution: Z-Stack." [Online]. Available: https://www.ti.com/tool/Z-STACK [Accessed: 02 July 2020].

[221] Texas Instruments, "Bluetooth Low Energy software stack." [Online]. Available: https://www.ti.com/tool/BLE-STACK [Accessed: 02 July 2020].

[222] M. Jiang, Y. Zhou, X. Luo, R. Wang, Y. Liu, and K. Ren, "An empirical study on ARM disassembly tools," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, (New York, NY, USA), p. 401–414, Association for Computing Machinery, 2020.

[223] J. Friebertshäuser, F. Kosterhon, J. Classen, and M. Hollick, "Polypyus–the firmware historian," 2020.

[224] F. Bellard, "QEMU, a fast and portable dynamic translator.," in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, p. 46, 2005.

[225] N. A. Quynh, "Unicorn: The ultimate CPU emulator," 2020. [Online]. Available: https://www.unicorn-engine.org [Accessed:25 Oct 2020].

[226] S. M. Mishra, *Wearable Android: Android Wear and Google Fit app development.* John Wiley & Sons, 2015.

[227] L. Goudge and S. Segars, "Thumb: reducing the cost of 32-bit RISC performance in portable and consumer applications," in *COMPCON'96. Technologies for the Information Superhighway Digest of Papers*, pp. 176–181, IEEE, 1996.

[228] Hex-Rays, "IDA Pro disassembler." [Online]. Available: https://www.hex-rays.com/products/ida/support/download_freeware/. [Accessed: 31 Jan 2021].

[229] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1667–1680, 2018.

[230] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *International Conference on Computer Aided Verification*, pp. 463–469, Springer, 2011.

[231] S. Alvarez, "radare2." `https://github.com/radareorg/radare2`.

[232] National Security Agency, "Ghidra." `https://github.com/NationalSecurityAgency/ghidra`.

[233] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*, pp. 8–9, IEEE, 2017.

[234] ARM, "Vector table." [Online]. Available: `https://developer.arm.com/documentation/dui0552/a/the-cortex-m3-processor/exception-model/vector-table` [Accessed: 03 July 2020].

[235] X. Yin, S. Liu, L. Liu, and D. Xiao, "Function recognition in stripped binary of embedded devices," *IEEE Access*, vol. 6, pp. 75682–75694, 2018.

[236] N. A. Quynh, "Capstone: The ultimate disassembler," 2020. `https://www.capstone-engine.org`.

[237] ARM, "Register usage in subroutine calls." [Online]. Available: `https://developer.arm.com/documentation/dui0473/m/writing-arm-assembly-language/register-usage-in-subroutine-calls` [Accessed: 08 July 2020].

[238] K. Wang, "Embedded real-time operating systems," in *Embedded and Real-Time Operating Systems*, pp. 401–475, Springer, 2017.

[239] ARM, "Calling SVCs from an application." [Online]. Available: `https://developer.arm.com/documentation/dui0471/m/handling-processor-exceptions/calling-svcs-from-an-application` [Accessed: 28 July 2020].

[240] ARM, "Supervisor calls." [Online]. Available: `https://developer.arm.com/documentation/dui0471/g/handling-processor-exceptions/supervisor-calls` [Accessed: 28 July 2020].

[241] Nordic Semiconductor, "nRF Connect for mobile," 2020. `https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Connect-for-mobile`.

[242] G. Klostermeier and M. Deeg, "Case study: Security of modern Bluetooth keyboards," 2018. [Online]. Available: `https://www.syss.de/fileadmin/dokumente/Publikationen/2018/Security_of_Modern_Bluetooth_Keyboards.pdf` [Accessed: 30 Nov 2020].

[243] STMicroelectronics, "AN4869: The BlueNRG-1, BlueNRG-2 BLE OTA (over-the-air) firmware upgrade," 11 2018.

[244] STMicroelectronics, "PM0257: BlueNRG-1, BlueNRG-2 BLE stack v2.x programming guidelines," 01 2019.