

MASTER THESIS

UNIVERSITY OF BERGEN

DEPARTMENT OF INFORMATICS

---

Hardware in the Hands of the Unaware:  
Remote BLE Keystroke Injection on  
USB-Armory Mk II and the Pedagogical  
Gap in Cybersecurity

---

*Author:*

Oskar Krystian Michalski

*Supervisor:*

Øyvind Ytrehus

June 1, 2025



## 0.1 Acknowledgment

I would like to sincerely thank my supervisor, Dr. Øyvind Ytrehus, for his invaluable guidance, constructive feedback, and continuous encouragement throughout this thesis. His motivation and support have been instrumental, offering critical insights while also allowing me the necessary creative freedom. His extensive expertise in cybersecurity significantly shaped my research direction and encouraged critical and independent thinking.

Special thanks to Bjørn Møller Greve for inspiring me to pursue the Bluetooth functionality within the Armory Device.

I am also grateful to my fellow students and colleagues for insightful discussions and continuous motivation throughout the development of this thesis. Additionally, I extend my thanks to the University of Bergen for providing the essential resources and tools required to complete my research.

I would also like to acknowledge ChatGPT, the artificial-intelligence language model, for helping me navigate the GitHub repository, clarify domain-specific concepts, and translate specialized terminology in the available documentation as well as comprehend complex aspects of this topic.

Finally, I wish to express my deepest gratitude to my family for their unwavering support, understanding, and patience during the entire process of writing this thesis. Their encouragement and consideration provided me the peace and clarity necessary to accomplish my goals.

Oskar Krystian Michalski  
June 1, 2025



## 0.2 Abstract

As our daily lives become increasingly digitalized, cybersecurity is critical for protecting personal and sensitive data. Yet, despite its importance, most people have limited knowledge and understanding of cybersecurity fundamentals. This gap in awareness is a significant vulnerability, potentially compromising even the most sophisticated security systems. To explore these challenges, this thesis investigates the usability and practical functionality of the USB-C Armory Device MK II, a portable security tool portrayed by the public as an advanced penetration testing platform. By implementing a keystroke injection attack via Bluetooth Low Energy (BLE), I demonstrate both the device's capabilities and the considerable barriers newcomers face when trying to acquire cybersecurity knowledge. This study reveals how poorly presented cybersecurity documentation can discourage proactive users, while motivated attackers easily exploit available vulnerabilities. Ultimately, this work aims to highlight the urgent need for more accessible and intuitive cybersecurity resources, promoting better awareness among everyday users.



## 0.3 Research Questions

This thesis addresses the following research questions:

Firstly; *How can the USB-C Armory Device MK II serve as an effective tool for testing personal cybersecurity for users without specialized expertise?*

Secondly; *What specific challenges do non-experts encounter when attempting to educate themselves about cybersecurity and current cyber threats?*

Lastly; *How can awareness of cybersecurity threats and countermeasures impact our safety in the digital world?*

In exploring these questions, I investigate specific aspects of cybersecurity that pose challenges for beginners, identifying obstacles in acquiring cybersecurity awareness and understanding. Additionally, I examine methods to clearly illustrate the real-world dangers posed by cyberattacks, even when performed by relatively inexperienced attackers. Ultimately, this research seeks to bridge the knowledge gap between cybersecurity experts and the general public, exploring ways to make powerful cybersecurity tools accessible and manageable for non-experts. As well as showcasing how easier accessibility of such tools and knowledge can enhance overall cybersafety of individuals.





# List of Figures

1.1	Order of Topics . . . . .	5
2.1	404 Server could not find client-requested page . . . . .	12
2.2	Overlooked resources . . . . .	13
2.3	Communication setup between the Linux system on USB-C Armory Device and my virtual machine. . . . .	17
3.1	Toggling Visibility . . . . .	23
3.2	Successful settings reading . . . . .	24
3.3	Default Sequence . . . . .	25
3.4	Read current configuration settings. . . . .	26
3.5	Results of the BLE scan. . . . .	28
3.6	Discovering ANNA, without being able to connect to it. . . . .	29
3.7	AT confirmation message from the ANNA module seen in Picocom interface. . . . .	30
3.8	Intercommunication between underlying system components. . . . .	35
3.9	Oracle VM settings connecting USB-C Armory Device MK II to my Kali Linux VM . . . . .	39
4.1	Terminal not connected to the Armory Device . . . . .	43
4.2	Terminal connected to the Armory Device . . . . .	44
4.3	Injected script into the not connected terminal . . . . .	45
4.4	Background Python string processing . . . . .	45
4.5	Python terminal seen by the user connected to the Armory Device . . . . .	46
4.6	Keystroke Injection Countermeasures and their limitations . . . . .	48



# List of Tables

3.1	BLE services and their characteristics . . . . .	30
3.2	The properties of the characteristics. . . . .	31



# Contents

Acknowledgements . . . . .	i
0.1 Acknowledgment . . . . .	i
Abstract . . . . .	iii
0.2 Abstract . . . . .	iii
Research Question. . . . .	v
0.3 Research Questions . . . . .	v
List of Figures . . . . .	vii
List of Tables . . . . .	ix
Contents . . . . .	xi
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Related Work . . . . .	2
1.3 Problem Statement . . . . .	3
1.4 Objective . . . . .	3
1.5 Methodology . . . . .	3
1.6 Thesis Organization . . . . .	4
1.7 Bluetooth Technology in Daily Life . . . . .	4
1.7.1 Bluetooth Low Energy (BLE) . . . . .	6
1.8 Cybersecurity Context and Threat Landscape . . . . .	6
1.8.1 Open-Source vs. Proprietary Security Approaches . . . . .	6
1.9 Social Engineering and Hardware-Based Threats . . . . .	7
1.9.1 Combined Threats: Bluetooth and BadUSB Attacks . . . . .	7
1.10 Incentive and Research Gap . . . . .	7
1.10.1 My Expertise and Its Relevance . . . . .	8
1.10.2 Bridging Cybersecurity and Education . . . . .	8
<b>2 Background</b>	<b>9</b>
2.1 USB-C Armory Device mk II. . . . .	9
2.1.1 Hardware Specifications . . . . .	9
2.1.2 Device Features . . . . .	10
2.1.3 Adaptability . . . . .	11
2.2 ANNA-B112 . . . . .	11
2.2.1 Bluetooth Isolation and Security Features of ANNA-B112 . . . . .	12
2.2.2 AT Commands and ANNA Configuration . . . . .	14
2.3 Experimental Setup . . . . .	15
2.3.1 Intended Communication Method . . . . .	15
2.3.2 Final Communication Setup . . . . .	16
2.4 Challenges . . . . .	17

2.4.1	Initial Setup Difficulties . . . . .	17
2.4.2	Documentation and Configuration Struggles . . . . .	17
2.4.3	Device Communication Exploration . . . . .	17
2.4.4	Connectivity Adjustments and Final Approach . . . . .	18
2.4.5	System Stability and Storage Issues . . . . .	18
2.4.6	Bluetooth Access Challenges . . . . .	20
2.4.7	Filesystem Integrity Problems . . . . .	20
2.4.8	Comments on the GitHub Repository . . . . .	21
<b>3</b>	<b>Thesis</b>	<b>22</b>
3.1	An practical introduction to ANNA . . . . .	22
3.1.1	Establishing BLE Communication Using Picocom . . . . .	22
3.2	Modifying ANNAs behavior . . . . .	23
3.3	ANNA-B112 Integration and Limitations . . . . .	27
3.4	Establishing my grounds with ANNA-B112 . . . . .	27
3.5	The Discovery of ANNA . . . . .	27
3.5.1	Finding ANNA . . . . .	28
3.5.2	Connecting to ANNA . . . . .	28
3.5.3	Exploring services of ANNA . . . . .	29
3.5.4	The Escape Character . . . . .	30
3.5.5	The final script . . . . .	31
3.6	Bridging Bluetooth with the Rest of the Device . . . . .	34
3.6.1	Bypassing ANNA . . . . .	34
3.6.2	Plugging Python to the Port . . . . .	35
3.6.3	listening.py in Greater Detail . . . . .	36
3.7	HID Emulation . . . . .	38
3.7.1	Explaining the <code>hidnet.sh</code> Script . . . . .	39
3.7.2	Modifying the <code>hidnet.sh</code> Script . . . . .	40
3.7.3	Using the Modified <code>hidnet.sh</code> Script for Keystroke Injection . . . . .	41
3.8	Preparing Automatic Launch on Boot . . . . .	41
<b>4</b>	<b>Results &amp; Future Work</b>	<b>43</b>
4.1	My Results . . . . .	43
4.1.1	The Impact of My Results . . . . .	44
4.1.2	Trusted Device Vulnerabilities . . . . .	47
4.1.3	Highlighting the Weakest Link . . . . .	47
4.1.4	Countermeasures and Limitations . . . . .	47
4.1.5	Improvements and Future Work . . . . .	49
	<b>References</b>	<b>50</b>
	<b>Appendices</b>	<b>53</b>
<b>A</b>	<b>Exciting results</b>	<b>54</b>
A.1	The HID emulation scripts . . . . .	54
A.1.1	Modified <code>hidnet.sh</code> bash script. . . . .	54
A.1.2	Original <code>hidnet.sh</code> script . . . . .	55
A.2	Beginning of BLE communication with ANNA-B112 . . . . .	57
A.3	Connection Attempts. . . . .	58

A.3.1	Attempting Connection after full iteration. . . . .	58
A.3.2	Attempting Connection instantly after discovering the target device. . . . .	59
A.4	Script discovering the services of the device. . . . .	60
A.5	User terminal connection . . . . .	62
A.6	Testing connection with multiple AT commands. . . . .	65
A.7	listening.py . . . . .	66
A.8	Services . . . . .	68
A.8.1	usb-gadget-hid-ecm.service . . . . .	68
A.8.2	rem-com-exec.service . . . . .	68





# Chapter 1

## Introduction

### 1.1 Motivation

Although cybersecurity concerns affect virtually everyone, only a minority actively pursue knowledge to enhance their digital safety. The rapidly evolving landscape of cybersecurity requires continuous learning, making it a demanding task even for professionals. For average users, staying updated on cybersecurity threats and effective protective measures is often overwhelming and impractical. Moreover, unclear or overly technical documentation of security assessment tools further discourages proactive cybersecurity engagement among non-expert users, resulting in uncertainty and potential vulnerability.

The USB-C Armory Device Mk II attempts to address these concerns through its transparent and open-source design, allowing users direct access to its source and offering an overview of its cryptographic capabilities via the publicly available GitHub repository [1]. Its compact yet powerful hardware offers extensive computing capabilities, including advanced features like Bluetooth Low Energy (BLE) Serial Port Service (SPS) for remote control and Human Interface Device (HID) emulation. Such features are particularly valuable in penetration testing scenarios, yet remain underutilized by novices due to documentation complexity and usability barriers.

In contrast, simpler tools like the Rubber Ducky emphasize the need for intuitive, user-friendly interfaces to engage non-expert users effectively. The practical difficulty associated with configuring the Armory Device, as compared to more user-friendly alternatives, highlights a significant usability gap.

This thesis, therefore, aims to address the critical balance between powerful functionality and usability in cybersecurity tools. By demonstrating how sophisticated devices like the USB-C Armory Device Mk II can become accessible for non-specialists, this research underscores the importance of reducing the knowledge gap between cybersecurity experts and everyday users. Enhancing user awareness and practical understanding is essential for strengthening individual digital security, ultimately contributing to broader societal cybersecurity versatility and robustness against threats.

## 1.2 Related Work

Several studies address the significant risks posed by Social Engineering and how user unawareness contributes to security breaches. The study "Impact of Human Vulnerabilities on Cybersecurity" by Maher Alsharif, Shailendra Mishra, and Mohammed AlShehri [2] emphasizes the critical role human behavior plays in cybersecurity. Their findings illustrate how robust cybersecurity measures can fail due to simple human errors. Despite numerous efforts to bridge the knowledge gap between cybersecurity professionals and general users, as highlighted by research such as "The Effect of Countermeasure Readability on Security Intentions" [3], there remains significant consumer unawareness regarding threats and best practices.

Moreover, despite Bluetooth being integral to daily routines, many users lack even basic knowledge of its security implications. Studies such as "USB Devices Phoning Home" [4] reveal severe consequences stemming from user ignorance regarding threats like badUSB attacks. Similarly, a comprehensive study by Armis researchers [5] identified eight zero-day vulnerabilities in Bluetooth protocols widely used in 2017, underscoring the urgent need for greater public awareness of commonly trusted digital technologies.

It is important to mention that Patrik Sandstad's master thesis [6] has helped me greatly in developing an understanding of the Armory Device which is of great importance in this thesis. Patrik Sandstad documented several downsides and technical challenges throughout his work with Armory Device. Some of these challenges were very similar to the troubles I encountered while working with the device.

## 1.3 Problem Statement

Cybersecurity is a rapidly evolving field impacting nearly every aspect of daily life, both professionally and personally. Despite its pervasive presence and necessity, the majority of users possess minimal cybersecurity understanding, often unable to differentiate genuine security practices from malicious threats such as phishing attacks. This widespread knowledge gap creates a critical vulnerability—the human factor.

In this thesis, I address the common lack of cybersecurity awareness among everyday digital users and examine barriers preventing individuals from acquiring essential cybersecurity knowledge. By practically engaging with a sophisticated cybersecurity tool (the USB-C Armory Device MK II) I aim to document the difficulties faced by newcomers navigating complex cybersecurity documentation and jargon. Implementing a keystroke injection script via Bluetooth Low Energy (BLE) remote control, I illustrate how confusing and inaccessible expert documentation can inadvertently facilitate dangerous cyberattacks, even from relatively inexperienced attackers.

## 1.4 Objective

This thesis aims to investigate the existing knowledge gap between novice cybersecurity users and industry experts. Through practical engagement with the USB-C Armory Device MK II, I document firsthand experiences of the complexities faced when encountering unintuitive cybersecurity concepts and documentation. Highlighting the challenges stemming from assumptions of prior foundational knowledge, I explore how such barriers retard the adoption of cybersecurity among non-specialists.

Additionally, by developing a keystroke injection functionality leveraging the Armory Device’s BLE capabilities, I will showcase both the device’s adaptability and critically assess the clarity and accessibility of its official documentation. Throughout the thesis, I maintain a balance between practical experimentation and theoretical analysis to effectively engage both technically experienced and inexperienced readers.

## 1.5 Methodology

Throughout this thesis, I adopt a firsthand, experimental approach, documenting my development of a unique functionality for the USB-C Armory Device MK II by leveraging its Bluetooth Low Energy (BLE) capabilities. Simultaneously, I aim to perceive the documentation and development experience from the perspective of a newcomer to cybersecurity. This dual perspective addresses both the general lack of cyber threat awareness among inexperienced users and the specific challenges newcomers face when seeking reliable and comprehensible cybersecurity resources.

This research primarily follows a quantitative, trial-and-error methodology. My foundational knowledge stems from my Bachelor’s degree in Cybersecurity, supplemented by introductory programming courses. Prior to this thesis, my experience with low-level embedded devices like the USB-C Armory Device MK II was minimal. My expertise predominantly lies in cryptographic functions, accompanied by

moderate Linux proficiency.

Throughout this work, I thoroughly document my approach to navigating GitHub repository information, overcoming encountered challenges, and leveraging device features for simulated attacks. These experiments highlight how devices specifically designed for security purposes can paradoxically expand the attack surface and introduce vulnerabilities, providing an insightful firsthand account of a newcomer's experience with complex cybersecurity implementations.

## 1.6 Thesis Organization

This thesis is organized as follows in an intuitive order such that while reading it one gets an intuitive understanding of the knowledge required to understand subsequent sections. (As demonstrated in Figure 1.1):

**Background:** Initially, I present an overview of the core issues this thesis addresses, highlighting the dangers associated with cybersecurity unawareness. This section discusses social engineering threats, Bluetooth vulnerabilities, and the open-source versus proprietary code debate to contextualize the specific challenges explored in this thesis.

**Prerequisite Concepts:** Following this, essential concepts necessary for understanding the subsequent results are introduced, including badUSB attacks, keystroke injection, and specific technical aspects of the USB-C Armory Device, such as the ANNA-B112 Bluetooth module.

**Implementation and Analysis:** The central part of this thesis details the practical implementation of my keystroke injection script, coupled with a technical breakdown of its functionality. This section provides insight into both the attacker's perspective in developing an exploit and the victim's perspective in gaining awareness and protection against such attacks.

**Results and Future Work:** Finally, I discuss the broader implications of my findings, summarizing what I've learned throughout this research. Additionally, I propose directions for future development, outlining potential improvements and advancements for the implemented script and exploring further cybersecurity research possibilities.

## 1.7 Bluetooth Technology in Daily Life

Bluetooth has become an integral feature of modern mobile devices, deeply embedded in our daily routines. As we carry our phones, smartwatches, and headphones everywhere, we are continuously surrounded by various Bluetooth signals. Conceptually, these signals resemble radio waves—just operating at different frequencies. Your phone detects nearby Bluetooth devices much like a radio tunes into stations, allowing users to establish connections effortlessly. Upon selecting a device to connect with, your device creates a "communication agreement," defining precisely how data will be exchanged, enabling multiple simultaneous connections without interference.

Bluetooth's convenience for short-range (typically around 10 meters) communication has made it ubiquitous for personal device interactions such as audio streaming, file transfers, and device synchronization. Most pairing procedures are automatic,

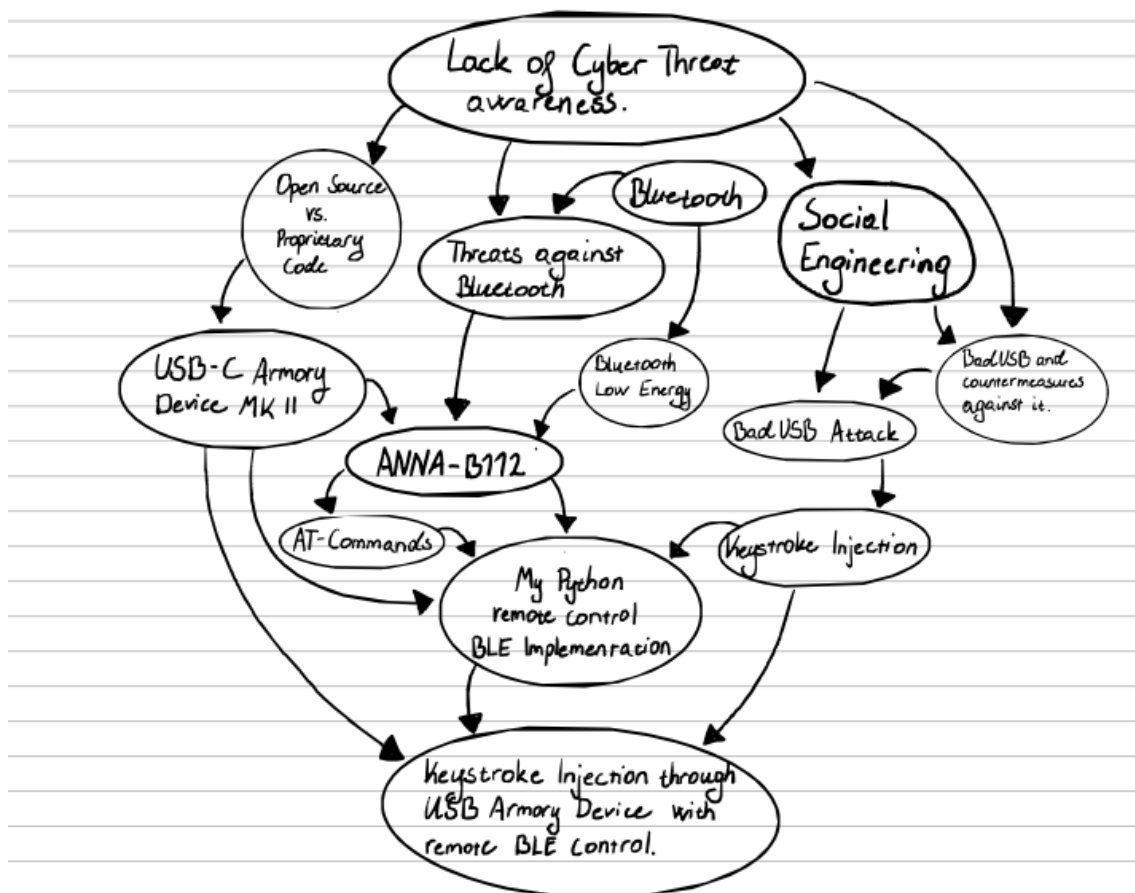


Figure 1.1: Order of Topics

often prompting users with simple confirmations like "Pair," "Connect," or "Disconnect." To streamline usability further, previously connected devices may be stored as "Trusted Devices," allowing automatic reconnections without repeated confirmations.

However, this ease of use introduces notable security risks. Each Bluetooth-enabled device represents a potential entry point for unauthorized access. Standard pairing protocols typically prevent unauthorized connections, yet sophisticated attackers can exploit vulnerabilities or misconfigurations, silently pairing with devices and potentially extracting sensitive data unnoticed. Consequently, while Bluetooth significantly enhances convenience, awareness and proactive management are crucial to maintaining personal and data security. Simply turning Bluetooth off when not in use can greatly mitigate such risks.

### **1.7.1 Bluetooth Low Energy (BLE)**

Bluetooth Low Energy (BLE) is designed specifically for devices demanding minimal power consumption, distinguishing itself from traditional Bluetooth by significantly enhancing battery life. Rather than maintaining continuous data streams, BLE transmits small data packets intermittently, enabling devices to remain mostly dormant and significantly conserve power.

BLE is commonly used in fitness trackers, smartwatches, medical devices, automotive sensors, and various IoT applications. Its efficient power consumption makes it ideal for devices requiring long-term operation without frequent recharging, emphasizing BLE's relevance in modern, battery-dependent applications.

## **1.8 Cybersecurity Context and Threat Landscape**

Today's cybersecurity landscape features numerous challenges, including threats posed by social engineering, hardware-based attacks like BadUSB, and user awareness gaps. Additionally, this section highlights how open-source security strategies, such as those employed by devices like the USB-C Armory Device MK II, differ from proprietary solutions, providing context for the subsequent discussion.

### **1.8.1 Open-Source vs. Proprietary Security Approaches**

The choice between open-source and proprietary software significantly impacts security strategies. Proprietary software hides its source code from the public, relying heavily on secrecy and vendor trust. However, exposure of even minor sections of proprietary code can severely compromise security, especially for large corporations that serve millions of users.

Open-source software, conversely, provides public access to its codebase, aligning closely with Kerckhoffs's principle, which argues that a secure system should remain secure even if all details except the encryption keys are public knowledge. Open-source approaches allow continuous public scrutiny, fostering community-driven improvements and potentially identifying vulnerabilities faster. However, openness alone doesn't guarantee security, as nearly half of open-source applications reportedly contain critical vulnerabilities, underscoring the necessity of comprehensive and ongoing community engagement for maintaining robust security.

## 1.9 Social Engineering and Hardware-Based Threats

Social engineering remains a pervasive cybersecurity threat, manipulating users into unknowingly compromising security measures. According to Cisco, social engineering remains one of the most common attack vectors, demonstrating the critical importance of user awareness and education. For instance, Chainalysis [7] reported record-high ransomware payments exceeding \$460 million in the first half of 2024, emphasizing the financial and operational impacts of successful social engineering attacks.

Hardware-based attacks, notably BadUSB, further compound this threat landscape. Attackers exploit human trust and curiosity by leaving malicious USB devices in accessible locations. Once connected to a victim’s computer, these devices can execute keystroke injections, silently installing malware or extracting sensitive information. Defenses like USB port blocking or keystroke monitoring are helpful but have limitations, such as users inadvertently overriding security measures.

### 1.9.1 Combined Threats: Bluetooth and BadUSB Attacks

Combining Bluetooth’s inherent vulnerabilities with BadUSB tactics creates an especially potent attack vector. Bluetooth, often overlooked in security protocols, lacks widespread specialized detection tools, making malicious connections challenging to detect. For instance, Bluetooth-based Serial Port Service (SPS) communications do not produce typical network footprints, unlike standard network-based attacks (e.g., SSH connections). Therefore, detecting unauthorized Bluetooth activities requires specialized monitoring tools and physical proximity, complicating defensive strategies.

Organizations frequently underestimate Bluetooth vulnerabilities, neglecting comprehensive Bluetooth-specific security measures. This oversight, when coupled with hardware attack vectors such as BadUSB, significantly expands potential vulnerabilities, emphasizing the importance of proactive user education and robust security awareness.

## 1.10 Incentive and Research Gap

The USB-C Armory Device MK II exemplifies powerful, versatile cybersecurity hardware, capable of facilitating penetration testing, secure communications, and everyday cryptographic applications. Its open-source nature offers adaptability crucial for high-trust environments.

However, despite its significant capabilities, the device suffers from usability challenges, primarily due to complex and poorly accessible documentation, hindering adoption by less experienced users. This difficulty reflects a broader issue within cybersecurity: the significant knowledge gap between experts and everyday users, creating vulnerabilities stemming from user misunderstanding or ignorance of security threats.

This thesis aims to explore and document these usability challenges through practical experience with the Armory Device, specifically implementing a remote-controlled keystroke injection attack via BLE. This approach illustrates not only

the device’s technical potential but also highlights barriers to effective utilization by non-experts.

### **1.10.1 My Expertise and Its Relevance**

My academic background in cryptography and familiarity with low-level programming provided a valuable foundation for working with the USB-C Armory Device MK II. Yet, despite relevant expertise, the complexity of implementing BLE-based functionalities posed considerable challenges. These difficulties underscored the extensive prerequisite knowledge required to effectively utilize sophisticated cybersecurity tools, illuminating how intimidating the learning curve might be for users new to the field.

This personal experience reinforced the notion that powerful cybersecurity tools are only as effective as their accessibility and ease of use. Improving documentation and educational approaches can significantly enhance tool adoption and effectiveness, emphasizing the critical intersection between technical security measures and accessible user education.

### **1.10.2 Bridging Cybersecurity and Education**

Cybersecurity requires not only advanced technical skills but also effective communication and educational methods. Unfortunately, cybersecurity professionals often lack the pedagogical expertise needed to convey complex topics effectively to non-specialists. As threats evolve, the gap between experts and everyday users continues widening, creating vulnerabilities due to user ignorance and misunderstandings.

Through direct engagement with the Armory Device, this thesis highlights the challenges users face when attempting to grasp cybersecurity fundamentals, emphasizing the urgent need for improved pedagogical strategies within the cybersecurity domain. Enhancing educational clarity can significantly mitigate inadvertent user-created vulnerabilities, strengthening overall cybersecurity resilience.



# Chapter 2

## Background

This chapter introduces key topics and concepts relevant to this thesis. It provides an overview of the primary device explored in this study, the USB-C Armory Device MK II, along with its integrated Bluetooth module, ANNA-B112. It discusses the role of the ANNA-B112 module in safeguarding and managing Bluetooth Low Energy (BLE) communications on behalf of the Armory Device. Additionally, an introduction to the initial experimental setup and a brief overview of the primary challenges encountered during the course of this research will be presented.

### 2.1 USB-C Armory Device mk II.

The USB-C Armory Device Mk II, widely known as "the Swiss Army knife of cybersecurity," has earned its reputation among enthusiasts, professionals, and the cybersecurity community for its versatility and wide-ranging applications. The device is equipped with powerful hardware components as well as features that are perfectly maintaining the security within the system on the USB-C Device, and can easily be applied for penetration testing purposes. These features and the means of their application will be discussed in this chapter.

#### 2.1.1 Hardware Specifications

The USB-C Armory Device MK II comes with its own dedicated processor, RAM, and persistent storage, making it effectively a fully functional computer in the size of a USB flash drive. The processor used is the NXP i.MX6ULZ ARM Cortex-A7, known particularly for its strong security features and energy efficiency. The device includes 512 MB of RAM, an embedded 16 GB eMMC internal storage, and a MicroSD card slot for additional external storage. Furthermore, it allows users to change the boot method between the internal eMMC storage and an external MicroSD card.

A notably powerful feature of this device is its embedded Secure Element (eSE). Due to its robust security and isolation from the main system, the eSE is well-suited for securely storing sensitive information, such as cryptographic keys or even cryptocurrencies. This component, combined with the flexibility and compact size of the device, makes it particularly relevant to the cryptographic community.

Additionally, the USB-C Armory Device MK II is well-equipped for both cryptographic and penetration testing purposes. It can emulate various USB device types,

such as Ethernet adapters, Human Interface Devices (HID) like keyboards, and mass storage devices. This capability significantly expands its utility and broadens the potential attack vectors it can access during penetration testing.

## **2.1.2 Device Features**

### **Security Features**

The USB-C Armory Device MK II incorporates multiple security features aimed at enhancing both the speed and security of encryption and decryption processes. Notably, it includes a True Random Number Generator (TRNG) and Cryptographic Acceleration capabilities that allow faster cryptographic operations compared to a standard CPU.

Additionally, the device offers tamper resistance, which includes a secure boot mechanism that verifies the authenticity of firmware during startup, and protects the system against unauthorized firmware modifications. The compact size of the device itself can also be viewed as a security advantage; the integrated hardware components make physical tampering just as challenging as digital tampering.

The ANNA-B112 Bluetooth module further enhances security by strictly managing Low Energy Bluetooth (BLE) functionality. ANNA acts as a security gatekeeper, isolating and supervising Bluetooth interactions. This isolation significantly reduces potential attack vectors, as described in Section 1.7. Additionally ANNA offers a wide spread of security settings which can be applied by leveraging AT commands to communicate with ANNA and modify its behavior. (Discussed in greater detail in section 3.2)

### **USB Device Emulation**

The USB-C Armory Device MK II can emulate various USB device types, offering great adaptability for penetration testing purposes. Depending on its configuration, the device can function as a keyboard for command injection attacks or as a network adapter capable of facilitating man-in-the-middle attacks. This adaptability greatly enhances how versatile and practical for penetration testing this device might be.

### **CDC Ethernet Emulation**

By emulating a network adapter using TCP/IP protocols over USB-C, the device can communicate with a host system, making the device sort of an agent handling the communication on the hosts behalf. Furthermore, the device can utilize the host's authenticated network access to interact with remote resources.

### **Flash Drive Emulation via Mass Storage Gadget**

The USB-C Armory Device can emulate a mass storage device (such as a flash drive) using its internal eMMC storage. This feature allows users to conveniently load new firmware, store data, and facilitate file transfers between the host and the device. Combined with the previously mentioned features, this capability positions the device as an effective tool for data exfiltration from secured or isolated networks. Combining this feature with my implementation could potentially improve the capabilities of the attack even further, thus allowing an attacker to flash the whole

storage of a target device after downloading sensitive data from the cloud. This topic deserves further discussion in the potential further work chapter (Section 4)

## Stand-alone Mode

Finally, the USB-C Armory Device MK II can operate independently, performing computational tasks without the involvement of a host system. This capability effectively transforms the device into a flash drive-sized standalone computer, suitable for use in various IoT applications or isolated computing tasks.

### 2.1.3 Adaptability

The USB-C Armory Device MK II is highly adaptable, offering significant adaptability properties through its hardware and software features. One of the most prominent examples is the inclusion of a 10-pin General-Purpose Input/Output (GPIO) header. This GPIO header lets users easily communicate with external hardware components by configuring each pin for different purposes—such as powering external devices, transmitting data, or controlling hardware directly.

In addition to hardware adaptability, the Armory Device offers software flexibility. All software for the USB-C Armory Device MK II is publicly accessible through its GitHub repository [1]. This open-source availability allows developers to customize and adjust the code to perfectly align with their specific goals and needs.

Lastly, the Armory Device supports Linux-based operating systems and applications written in the Go programming language, which is recognized for effectively balancing security and performance. This combination allows for secure and efficient development and deployment of new applications for this device.

## 2.2 ANNA-B112

Due to the ANNA module completely isolating the Bluetooth adapters from the rest of the device. I encountered several challenges during my initial attempts to interact with Armory Devices Bluetooth functions. Given that this type of hardware-level security was entirely new to me, my progress slowed significantly, as I needed to thoroughly understand ANNA's functionality and the appropriate methods for interacting with it.

Initially, I was uncertain about how Bluetooth operated on such specialized hardware. However, based on my observations, I assumed that the primary objective of ANNA-B112 was to enhance security through extensive configuration options, advanced cryptographic algorithms, and complete control over Bluetooth Low Energy (BLE) implementations, and by also including aspects like authentication. Furthermore, by isolating Bluetooth interactions, ANNA minimizes the potential security risks associated with wireless connections.

While trying to familiarize myself with ANNA, I attempted to access its documentation through links provided in the USB-C Armory Device GitHub repository. Unfortunately, most of these links were outdated or broken, displaying a "404" error, as illustrated in figure 2.1. Eventually, I navigated directly to the U-Blox homepage,

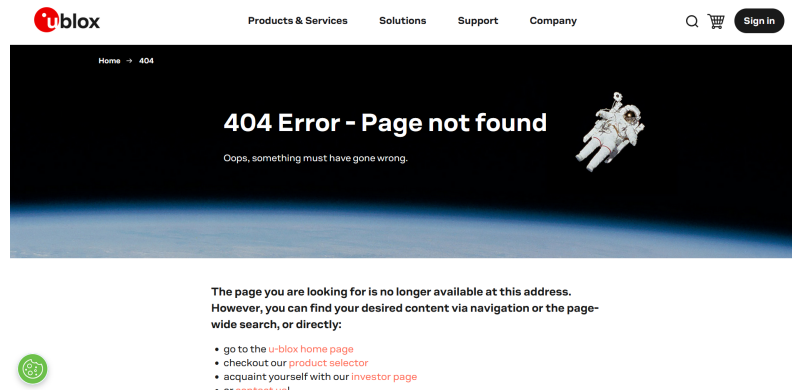


Figure 2.1: 404 Server could not find client-requested page

but even there, I initially only found basic resources such as data sheets and product summaries. It seemed like a comprehensive manual wasn't available. Much later, after I completed most of my work, I revisited the website and discovered that additional resources—including a user guide—was available on their homepage.[2.2](#).



Given these difficulties, my main strategy shifted to searching online using commands found in the initial Bluetooth configuration section of the USB-C Armory GitHub repository. By combining these commands with the module's name—"ANNA-B112 AT commands"—I successfully located the AT command manual, which provided the critical information I needed to configure the ANNA module.

Navigating the overwhelming amount of documentation and numerous repository pages to pinpoint the essential steps required for setting up the module was extremely time-consuming. Despite this challenge, discovering the "Bluetooth" section in the repository was crucial—it explicitly confirmed that the ANNA-B112 module was intended for "out-of-band" Bluetooth interactions, motivating further investigation.

My ultimate objective was to familiarize myself with the ANNA-B112 module and establish a functional Bluetooth connection with my laptop. Specifically, I wanted the device to be discoverable and provide transparent UART functionality to directly send Bash commands from my terminal to the USB Armory Device. Strangely enough, the commands listed in the GitHub repository appeared to configure the device as "non-discoverable, non-pairable, non-connectable, and disabled in any role (central or peripheral)," exactly the opposite of what I needed. This caused initial confusion, as I intuitively assumed setting these parameters to 1 (often implying "true") would activate these features, while 0 (false) would deactivate them. As I later discovered by closely reviewing the AT command manual, my assumptions were entirely incorrect.

### 2.2.1 Bluetooth Isolation and Security Features of ANNA-B112

When I first started working with implementing a simple Python script that would establish a Bluetooth connection to receive and execute commands (utilizing the "BlueZ" library on the Debian OS running on Armory Device) from my laptop to the Armory Device, I quickly encountered an unexpected roadblock. It turned out that the Bluetooth adapters were completely inaccessible. The operating system did

ANNA-B112 Data Sheet 	ANNA-B112 Product Summary 	EVK-ANNA-B112 	
---	--	--	--

Product selection	Product description	Documentation & resources
-------------------	---------------------	---------------------------

File Category ▾	Current X Reset	Expand
-----------------	-----------------	--------



Product Summary	▼
User Guide	▲
u-connectXpress user guide 7-Jan-2025 Product evaluation	 PDF 1.69 MB
s-center User Guide 31-Mar-2023 Product evaluation	 PDF 0.74 MB
Data Sheet	▼
Integration Manual	▼
Interface Manual	▼
Application Note	▼
CAD/CAE Libraries	▼
Evaluation Software	▼
Release Note	▼
Information Note	▼

Figure 2.2: Overlooked resources

not recognize them, and it wouldn't even allow me to check their status or enable Bluetooth using standard commands like `"systemctl enable bluetooth"` or `"systemctl status bluetooth"`.

This unexpected issue was due to the ANNA-B112 module having exclusive access to the Bluetooth module, effectively blocking any other applications running on the OS from accessing the BLE adapters. Although I knew ANNA-B112 was the Bluetooth module, I wasn't initially aware that it also restricted direct access to Bluetooth functionalities. This crucial detail was likely assumed to be obvious by the documentation on the GitHub repository. However, for someone like me, who didn't have extensive expertise in this specific area of firmware, this restriction was far from intuitive. Initially, I worried that my boot image might have been improperly configured, and thus spent considerable time troubleshooting potential misconfigurations rather than immediately realizing that the lack of access was intentional and enforced by ANNA.

Once I realized that ANNA-B112 intentionally restricted Bluetooth adapter access, it became clear that any communication had to pass exclusively through ANNA using AT commands. At this stage, I hoped that the ANNA module supported a UART Transparency mode or similar functionality, allowing remote command execution. Unfortunately, this isolation mechanism significantly complicated the process, forcing me to employ *ad hoc* scripting and alternative methods to bypass ANNA's isolation measures and successfully achieve remote command execution.

The ANNA-B112 module provides several security options that highlight its robust capability for securing Bluetooth communication in addition to BLE isolation:

- **SPS Encryption and Communication** – providing Serial Port Service communication and security for BLE communication.
- **Security Modes** – allowing different connection and pairing modes tailored to varying application-specific security requirements.
- **Advanced Security Types** – employing elliptic curve cryptography for secure pairing, combined with AES encryption algorithms for robust authentication and communication protection.
- **User Confirmation and Passkey Entry** – configurable features enhancing device security by requiring explicit user authentication.

These advanced security measures underscore ANNA-B112's design intent as a secure and isolated Bluetooth management solution.

## 2.2.2 AT Commands and ANNA Configuration

Briefly speaking, Attention Commands (AT Commands) are special commands used to communicate with modules such as ANNA-B112. They allow configuration and control of functionalities such as Bluetooth or Wi-Fi. For the ANNA module specifically, these commands enable adjustments to settings like device name, advertisement intervals, and power consumption. They also control connections, like for instance, connecting or disconnecting devices when ANNA is operating as a central or peripheral device and provide the ability to query current device states and settings.

The ANNA-B112 module supports AT commands via a UART serial interface. To interpret and execute these commands, ANNA-B112 relies on u-blox's "u-connectXpress firmware" which contains an AT command parser. Without this firmware, the module cannot understand or respond to AT commands.

The UART interface for ANNA-B112 on my USB-C Armory Device, running Debian 11, is located at the path `"/dev/ttyMXC0"`. This particular port became a central focus during my script development, where I monitored traffic passing through it, as discussed further in Chapter 3.

Leveraging these AT commands, I modified several settings on ANNA to facilitate a Bluetooth SPS (Serial Port Service) connection, in attempt to enable remote command execution via UART transparency from my laptop to the Armory Device. To simplify and speed up the Bluetooth connection process, I significantly lowered the advertisement interval using the command `"AT+UBTLECFG=1"`, reducing the minimum advertisement interval to 800 units (0.5 ms), down from the default of 1600 units (1 second). I also reduced the maximum interval to 1000 units (0.625 ms), down from the default of 1.25 seconds, using `"AT+UBTLECFG=2"`. These changes increased how often ANNA broadcasted BLE advertisements, making the device easier and faster for my laptop to discover, though this improvement came at the cost of higher power consumption.

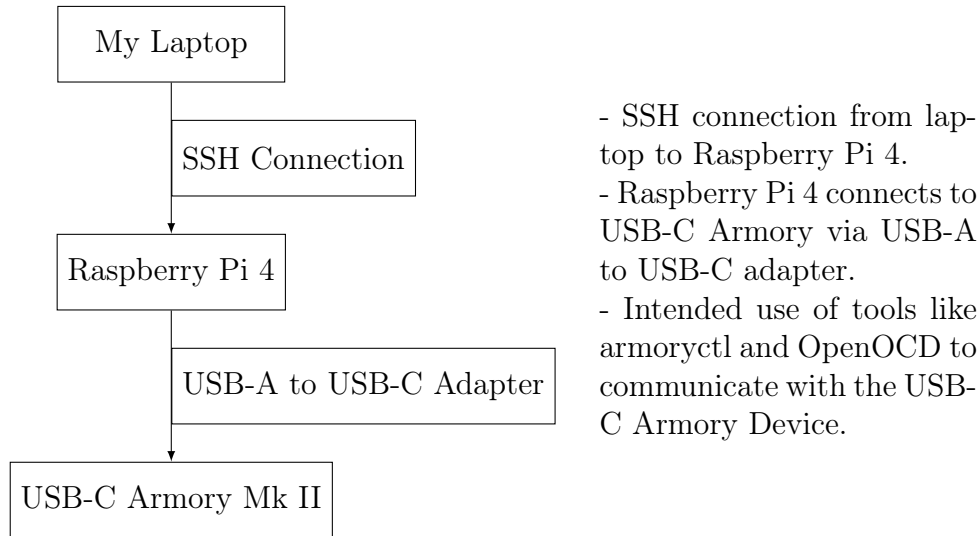
I ensured the SPS server was activated using the command `"AT+UDSC=0"`, verifying this with the response `"+UDSC:0,6"`, indicating that SPS was indeed enabled. SPS functionality essentially simulates a wired serial connection, enabling BLE devices to communicate as though physically connected.

Finally, to enable remote configuration capability, I set the server flag parameter `"AT+UDSF=0,1"`, allowing remote access and management of the ANNA-B112 module, which was initially disabled by default.

## 2.3 Experimental Setup

### 2.3.1 Intended Communication Method

When I started working with the Armory Device, my initial goal was to find an operating system that would let me easily explore and test the device's capabilities. I decided on using the Raspberry Pi OS because of its simplicity and close resemblance to Debian, which is the OS used for the original Armory Device image. My first setup attempt closely mirrored Patrik's approach, connecting the Armory Device to a Raspberry Pi 4 and then accessing the Pi remotely via SSH. The setup looked like this:



However, I quickly realized this setup introduced unnecessary complexity, especially during debugging. Since the Raspberry Pi 4 only has USB-A ports, I was forced to use a USB-A to USB-C adapter. Tools like OpenOCD and armoryctl require precise, direct configurations, and the adapter complicated things considerably. For instance, while following Inverse Path’s instructions from their GitHub repository [1], I struggled to detect or configure the important "imx\_gpio" pin needed by OpenOCD.

Before diving deeper into debugging, I made sure the Armory Device was recognized by running the `lsusb` command, confirming that it showed up correctly as a "Linux-USB Ethernet/RNDIS Gadget":

```
Bus 001 Device 003: ID 0525:a4a2 Netchip Technology,
Inc. Linux-USB Ethernet/RNDIS Gadget
```

Despite multiple attempts, the OpenOCD setup constantly failed, suggesting issues with the USB-A to USB-C adapter itself. Given these complications, I chose to abandon this initial approach and shifted focus toward a simpler, more practical method of exploring the device’s capabilities.

### 2.3.2 Final Communication Setup

The final setup used in this thesis is shown in Figure 2.3. I directly connected the USB-C Armory Device MK II to my laptop, running Kali Linux in a virtual environment. I configured port forwarding carefully to minimize common communication issues, especially those arising with Windows systems. This allowed a successful SSH connection and provided internet access to the Armory Device.

### Acknowledgments and Limitations

Usually, the Armory Device operates independently, routing internet traffic securely through its USB-C port. In my setup, however, the approach is reversed: I focused primarily on exploring the penetration testing features of the Armory Device itself. Because of this, evaluating the built-in security measures provided by the Armory Device was beyond the scope of this thesis.



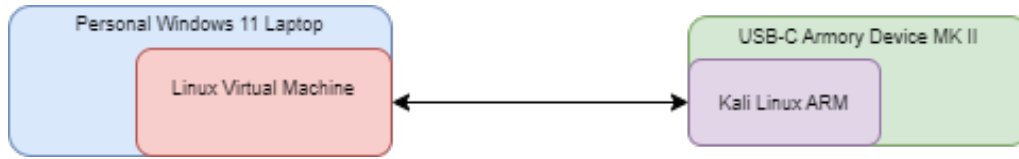


Figure 2.3: Communication setup between the Linux system on USB-C Armory Device and my virtual machine.

## 2.4 Challenges

While setting up my Armory Device MK II and implementing BLE functionality with the ANNA-B112 module, I encountered several challenges. These ranged from hardware compatibility issues to unclear documentation, requiring persistent troubleshooting and research.

### 2.4.1 Initial Setup Difficulties

One of the first major obstacles was simply understanding how the Armory Device worked. I initially knew very little about its internal mechanics, and the available documentation (Inverse Path’s GitHub repository) was extremely detailed but often confusing. Important details got buried within massive amounts of information, and clear, step-by-step guides were scarce.

My initial approach, connecting the Armory Device through a Raspberry Pi 4 using a USB adapter, quickly presented issues. The Pi didn’t correctly recognize the device, and I suspected the adapter was not properly translating the USB signals. Directly connecting to a Linux machine instead immediately identified the Armory Device as a network adapter, reinforcing my suspicion that the adapter setup was problematic.

### 2.4.2 Documentation and Configuration Struggles

The GitHub documentation for essential tools like `armoryctl` and `OpenOCD` was confusing, leading me to misunderstand whether these tools belonged to the host or directly on the Armory Device. My difficulties compiling `OpenOCD` with the necessary flags (like `"imx_gpio"`) likely arose from hardware incompatibility or incorrect setups. Further complications arose because I relied on a virtual machine (Kali Linux running in VirtualBox on Windows 11), which made USB access and external hardware configuration notably challenging.

### 2.4.3 Device Communication Exploration

#### Exploring Vendor-Defined Messages

Initially, the Armory Device identified itself as a "Freescale Semiconductor i.MX 6ULL SystemOnChip in RecoveryMode." Inspired by Gunnar Alendal’s work [8] on exploiting vendor-defined messages in USB protocols, I attempted to investigate this communication pathway. I created a preliminary C script to test brute-force interactions, which I’ll discuss further in Chapter 4.

## Operating System Installation

I chose the recommended Debian 12 OS from Inverse Path’s repository, downloaded the provided image, and flashed it onto a 32GB SD card. This straightforward step helped ensure compatibility and simplify later configurations.

## Challenges with armoryctl and Dependencies

Inspired by Patrik Sandstad’s previous work [9], I explored the armoryctl tool, aiming to manage device communication. However, vague instructions and unclear warnings made setting it up confusing. It wasn’t immediately obvious whether the tool was meant for the host machine or the Armory Device itself. Misinterpreting these instructions led me down unproductive paths, struggling with dependencies like OpenOCD and the "imx\_gpio" configuration.

### 2.4.4 Connectivity Adjustments and Final Approach

After continuous issues, I decided to simplify my setup by connecting the Armory Device directly to my laptop running Kali Linux within a virtual machine. Even though this transition introduced new challenges, like complex port forwarding configurations, it ultimately allowed me to overcome IP conflicts and establish reliable SSH communication. However, issues with compiling OpenOCD persisted, eventually leading me to bypass armoryctl entirely and rely directly on AT commands to communicate with ANNA-B112, as described further in Chapter 3.

Even then, unexpected complications arose, such as the device becoming unresponsive after power cycles, adding further hurdles to my progress.

### 2.4.5 System Stability and Storage Issues

#### Permanent "Off" State Problem

As I mentioned before, navigating the GitHub repository was not straightforward, causing me to miss some critical instructions—especially around correctly partitioning the SD card and properly flashing the device. This oversight led me into a frustrating situation where the Armory Device would abruptly stop functioning exactly 44 seconds after being connected.

After numerous trials and errors, I discovered the issue was related to improper partitioning of the SD card. Rather than following the overly complicated and unclear walkthrough provided by the repository, I developed my own simpler and more intuitive partitioning method, which I discuss in Chapter 4.

Once the partitioning was fixed, I could install dependencies and continue configuring armoryctl without issues. My theory is that without correct partitioning, essential boot or runtime instructions got overwritten due to extremely limited storage (around 0.2GB). This was probably why the device initially booted fine but failed after running dependency installations or system upgrades. The entire debugging process cost me a lot of time, and I even tried different operating systems suggested by the repository moderators, which ultimately didn’t address the actual problem—partitioning. This situation highlighted how my lack of specialized knowledge significantly complicated the process.

## Storage Space Problem and My Hypothesis

While investigating these issues, I noticed the Armory Device only used about 3.2GB of storage out of the available 32GB on the SD card. My hypothesis was that running `sudo apt upgrade` on limited storage space overwrote essential boot data. Initially, the device seemed functional, possibly because critical system files were temporarily stored in RAM. However, after power cycling, these files disappeared, leaving the device unable to boot.

## Resolving the Issue: Resizing the Filesystem

After understanding the storage limitation, I identified two ways to resize the filesystem: a complicated method recommended by the repository moderators and my own simplified approach. Naturally, I chose the simpler option:

After inserting the SD card into my PC via an adapter, I identified the SD card as `/dev/sdb` and then ran:

```
sudo parted /dev/sdb
```

Inside `parted`, I listed partitions with `print` and resized the first partition:

```
resizepart 1 100%
```

After quitting `parted`, I extended the filesystem itself:

```
sudo resize2fs /dev/sdb1
```

I confirmed the expansion was successful by using `df -h`, which showed significantly more available storage.

## Difficulty Locating Partitioning Instructions

Due to the repetitive flashing and boot setups I performed, having easy access to clear partitioning instructions became crucial. Unfortunately, finding these instructions within the GitHub repository was challenging. The recommended instructions were buried deep within multiple sub-links:

1. **Getting Started** linked to `usbarmory-debian-base_image`.
2. `usbarmory-debian-base_image` linked to `Pre-Compiled releases`.
3. **Pre-Compiled releases** contained numerous unrelated links, with resizing instructions hidden under question 5.
4. **Resizing the microSD/eMMC Partition** finally provided relevant instructions.

Even then, the provided method recommended using a "BeagleBoard," additional hardware costing between 600-1400 NOK. This seemed unnecessarily complex and expensive, prompting me to stick with my simpler manual resizing method.

## 2.4.6 Bluetooth Access Challenges

### Issues Accessing Bluetooth on Armory Device

A major obstacle was accessing Bluetooth functionality. The ANNA-B112 module, managing the BLE connections, had exclusive control over Bluetooth, making typical Linux Bluetooth methods useless. Attempts using `armoryctl` or directly interacting with hardware failed repeatedly, making it clear that ANNA-B112 operated independently, requiring specific commands and communication protocols.

### GPIO Pin Configuration Problems

Another significant headache was the unclear GPIO pin configuration needed for compiling OpenOCD with the required `"imx_gpio"` flags. The GitHub documentation was vague about what these pins were or how they should be configured, forcing me to rely heavily on trial-and-error. The documentation clearly assumed a deeper familiarity with ARM-based hardware than I possessed.

### Confusion on Tool Installation Location

Determining whether to install tools like `armoryctl` and OpenOCD on my host machine or directly on the Armory Device itself was confusing due to ambiguous documentation. This confusion led me to repeatedly set up the same tools in both environments, significantly delaying my progress and causing unnecessary complications.

### Adjusting Bluetooth Discovery Intervals

Based on my understanding of BLE protocols, I initially tried increasing the discovery intervals, expecting faster and more reliable connections. Contrary to expectations, this change didn't improve connection stability, suggesting deeper underlying issues between the system and ANNA-B112 firmware. This experience highlighted how complex BLE configuration is and how limited conventional tools were for resolving these issues.

## 2.4.7 Filesystem Integrity Problems

### System Remounting to Read-Only

When attempting system updates with `apt upgrade`, I encountered an issue where the filesystem suddenly became read-only, despite proper permissions. Checking system logs with:

```
dmesg | grep -i 'error|ext4|remount'
```

revealed filesystem corruption. To fix this, I had to physically remove the SD card, connect it to my PC, and debug from within a virtual machine—another challenging process on its own.

## Mounting the SD Card to My Virtual Machine

Mounting the SD card inside my VM required a rather complicated approach, involving creating a Virtual Machine Disk (VMDK) linked directly to the SD card. First, I identified the SD card path, verified it through Windows Disk Manager, and then executed:

```
C:\Program Files\Oracle\VirtualBox>
.\VBoxManage internalcommands createrawvmdk
-filename "C:\path\to\store\the\sdcard.vmdk"
-rawdisk \\.\PhysicalDriveX
```

After generating the .vmdk file, I added it through Oracle VM's storage settings, finally enabling direct filesystem editing.

### 2.4.8 Comments on the GitHub Repository

Navigating the GitHub repository was notably difficult due to numerous links, overly technical jargon, and a lack of clear structure. Even when users know what they're looking for, it's challenging to find relevant information.

From extensive experience, I consider Inverse Path's repository more like a cheat sheet than an actual guide. Cheat sheets are not intended as learning resources, and this analogy perfectly fits the GitHub repository. If you're already knowledgeable, you might find it useful. However, for newcomers trying to get started with the Armory Device, the repository can easily become overwhelming and, frankly, a bit of a nightmare.

# Chapter 3

## Thesis

In this chapter, I provide a practical introduction to the ANNA-B112 Bluetooth module, which proved essential in my efforts to develop a remote keystroke injection script using the USB-C Armory Device MK II. By carefully documenting each step of this process, I was able to highlight specific challenges that a non-expert user would likely encounter while trying to acquire knowledge within this field.

I describe in detail my experience getting familiarized with the ANNA-B112 module, including how I learned to interact with it through Python scripts and AT commands. I also discuss the specific modifications I made to the module and outline the various ways in which ANNA restricted some aspects of my development process. Similar descriptions are provided regarding my exploration of the Armory Device itself, particularly focusing on my experience using Debian 11 and its HID Emulation capabilities.

Each section concludes with a comprehensive explanation of the scripts I developed. My goal is to clearly illustrate the types of barriers that newcomers might face when attempting to understand complex cybersecurity tools and methodologies. Ultimately, this chapter serves as motivation for enhancing accessibility and adopting a more effective pedagogical approach to documentation—because knowledge shared without clear communication cannot truly be considered shared.

### 3.1 An practical introduction to ANNA

#### 3.1.1 Establishing BLE Communication Using Picocom

To ensure that establishing a connection with the ANNA BLE module on my Armory Device was possible, I needed to interact with the module through an appropriate interface. This interface facilitates communication between the Linux OS running on the Armory Device and the ANNA BLE module, allowing me to modify settings such as discoverability, pairability, connectability, and device role selection, including central or peripheral modes.

Initially, I used Minicom for communication, but it quickly proved to be inefficient due to several issues, including errors caused by incorrect line endings. Consequently, I switched to Picocom, which provided a more reliable and automated way of communicating with the ANNA module.

With Picocom, I was able to confirm the current state of the device. However, to fully utilize the AT commands provided by Ublox, I had to implement certain

workarounds.

## AT Command Workarounds

The Bluetooth section of the USB-C Armory GitHub repository provides a general command execution sequence, followed by a state save command and a restart, as illustrated in Figure 3.1. However, finding proper documentation or even documentation relating to these commands turned into an extensive internet search due to the outdated links in the repository.

My objective was not to configure the BLE module as non-discoverable, non-pairable, non-connectable, and role-disabled, as suggested in the default instructions. Instead, I wanted to verify whether it was possible to establish a connection with the module. To achieve this, I had to validate the parameters shown in Figure 3.1.

Fortunately, I was able to locate the AT commands documentation via ChatGPT's web search functionality [10]. In Section 6: Bluetooth, I found relevant commands to verify the current settings of the ANNA BLE module.



Figure 3.1: Toggling Visibility

It is also important to mention that the command execution sequence was not as straightforward as depicted in Figure 3.1. Using the provided sequence often resulted in errors, as shown in Figure 3.3, and even using the read mode commands (Figure 3.4) led to unexpected issues.

To address these inconsistencies, I applied an ad hoc approach, leveraging the read settings command and the restart command, as demonstrated in Figure 3.2.

## 3.2 Modifying ANNAs behavior

When I first started working with AT commands, I used a terminal interface called "picocom," which allowed me to interact directly with the ANNA-B112 module and modify its settings and behaviors through AT commands.

Initially, I spent several hours familiarizing myself with the U-blox manual for the AT commands, primarily due to its structure with overwhelming information but also because I wasn't entirely sure what information I was searching for. As mentioned earlier, my expertise lies primarily in coding and cybersecurity rather

```

AT+UPROD=1
OK
+STARTUP
AT+UPRODLFCLK=0,16,2
OK
AT+UBTDM?
ERROR
AT+UBTDM?
ERROR
AT+CPWROFF
OK
+STARTUP
AT+UBTDM?
+UBTDM:3
OK
AT+UBTPM
ERROR
AT+UBTPM?
+UBTPM:2
OK
AT+UBTCM?
+UBTCM:2
OK
AT+UBTLE?
+UBTLE:2
OK

```

Figure 3.2: Successful settings reading

than hardware programming or establishing Bluetooth Low Energy (BLE) connections. Consequently, understanding the intricacies of BLE communication required considerable effort. I had to grasp the fundamental differences between BLE and traditional Bluetooth connections, focusing especially on factors such as connection procedures and communication protocols unique to BLE.

Additionally, before beginning my work with this device I did not know anything about its capabilities and if I would be able to make use of the implementations available on the device. I needed to confirm whether the ANNA-B112 module itself supported the specific BLE functionalities required for my project. Although the documentation provided by the USB-C Armory Device Mk II indicated the presence of a "u-blox ANNA-B112 BLE" module [11], it did not explicitly detail the specific Bluetooth capabilities or services that the module supported. This lack of specificity led to further uncertainty and required additional exploration on my part to clearly identify the BLE features available on the ANNA-B112.



```

picocom v3.1
port is      : /dev/ttyMXC0
flowcontrol  : none
baudrate is  : 115200
parity is    : none
databits are : 8
stopbits are : 1
escape is    : C-a
local echo is : no
noinit is    : no
noreset is   : no
hangup is    : no
nolock is    : no
send_cmd is  : SZ -vv
receive_cmd is : RZ -vv -E
imap is      : crclrf,
omap is      :
emap is      : crclrf,delbs,
logfile is   : none
initstring   : none
exit_after is : not set
exit is      : no

Type [C-a] [C-h] to see available commands
Terminal ready
AT+UPROD=1

OK

+STARTUP

AT+UPRODLFCLK=0,16,2

Debian_bac...
OK

AT+UBTDM=1

ERROR

And flashin...
AT+UBTPM=1

ERROR

AT+UBTCM=1

ERROR

AT+UBTLE=0

ERROR

```

Figure 3.3: Default Sequence

```
AT+UBTDM?  
  
ERROR  
AT+UBTCM?  
dev2.txt  
  
ERROR  
AT+UBTBLE?  
  
ERROR
```

Figure 3.4: Read current configuration settings.

### 3.3 ANNA-B112 Integration and Limitations

The core purpose of the Armory Device's BLE interface is to remotely configure and manage the ANNA-B112 module from an external device. Typically, AT commands are used to modify ANNA's radio behavior, including how it advertises itself, accepts connections, and operates as either a central or peripheral device. However, these commands are not designed for large data transfers or comprehensive control over the Armory Device. Initially, I explored the possibility of establishing a more direct, high-bandwidth communication channel. Unfortunately, after extensive research through firmware documentation and available resources, I found no explicit support for such functionality. Given the often overly complicated or unclear documentation on how ANNA-B112 integrates with the Armory Device, I still cannot entirely exclude the existence of an indirect method to create a more transparent BLE link that I simply overlooked.

### 3.4 Establishing my grounds with ANNA-B112

Before establishing the Bluetooth connection, I needed to make several essential modifications to the ANNA-B112 module, detailed extensively in subsequent chapters. The most crucial adjustments included activating the Serial Port Service (SPS) to enable remote AT command execution and reducing the advertisement interval to expedite connection establishment.

The SPS is effectively a wireless counterpart of a physical serial connection, allowing two devices to communicate as if they were physically connected by a cable. In this context, the relevant port was identified as `"/dev/ttyMXC0"`. Initially, my assumption was that connecting to this port via BLE would allow me to directly inject terminal commands into the device. However, I misunderstood this implementation. The port was dedicated exclusively to receiving AT commands, not general terminal commands, compelling me to employ a somewhat ad-hoc solution.

Another initial misunderstanding involved the advertisement interval setting. Initially, I assumed increasing the interval to 10 seconds would offer more time for connection establishment. In reality, this adjustment significantly prolonged the connection time, increasing it from approximately two to four minutes. Further clarification, notably through discussions with ChatGPT, revealed that reducing the advertisement interval (thus increasing advertisement frequency) was preferable for Windows-based systems and considerably improved connection reliability and speed.

Navigating these configurations proved challenging and time-intensive, but ultimately, I achieved the correct setup, enabling me to proceed confidently with further exploration of Bluetooth Low Energy communication.

### 3.5 The Discovery of ANNA

During my research on practical methods for establishing BLE connections, I discovered a particularly helpful Medium article titled "TLDR: How to Control a Bluetooth LE Device with Python" [12]. The author shared valuable insights from personal experiences with BLE programming and recommended the Python library "Bleak"

for interacting effectively with BLE devices. Leveraging this advice and utilizing the built-in Bluetooth adapter on my Windows laptop, I successfully identified the ANNA-B112 module. This marked a significant milestone, as it allowed me to establish a stable and reliable connection, forming the foundation for my continued experimentation and deeper exploration of BLE capabilities.

### 3.5.1 Finding ANNA

Starting with the simple Python snippet from Proto Bioengineer’s Medium [13] post (shown below), I was able to scan for BLE peripherals and retrieve both the MAC address and the friendly name of the Armory Device. I used this later to connect with ANNA. The results of that scan are displayed in Figure 3.5.

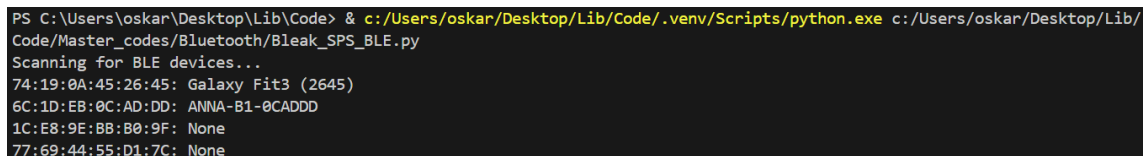
```
# Bluetooth LE scanner
# Prints the name and address of every nearby Bluetooth LE device

import asyncio
from bleak import BleakScanner

async def main():
    devices = await BleakScanner.discover()

    for device in devices:
        print(device)

asyncio.run(main())
```



```
PS C:\Users\oskar\Desktop\Lib\Code> & c:/Users/oskar/Desktop/Lib/Code/.venv/Scripts/python.exe c:/Users/oskar/Desktop/Lib/Code/Master_codes/Bluetooth/Bleak_SPS_BLE.py
Scanning for BLE devices...
74:19:0A:45:26:45: Galaxy Fit3 (2645)
6C:1D:EB:0C:AD:DD: ANNA-B1-0CADD
1C:E8:9E:BB:B0:9F: None
77:69:44:55:D1:7C: None
```

Figure 3.5: Results of the BLE scan.

### 3.5.2 Connecting to ANNA

After discovering the device, the next step was to connect and maintain that connection—but it was not as straightforward as I would hope. As a Bleak newcomer, I naturally assumed I’d need to run a full scan each time, much like pairing in Windows: discover first, then connect. However, the Armory Device would only advertise itself for a few seconds before returning to sleep mode to conserve power. By the time my script spotted the device and tried to connect, it had often already stopped advertising, and go to sleep, resulting in repeated “Connection failed” errors. The original script used for this initial connection method is provided in Appendix A.3.1, and the complication this method entailed can be seen in Figure 3.6

Initially, I considered several potential reasons for these connection failures. First, I thought my laptop might not be compatible with establishing this specific BLE connection. However, given that the Bleak scan successfully detected the

```
Found ANNA-B1-0CADD with address 6C:1D:EB:0C:AD:DD
Attempting to connect...
Error during connection or communication: Device with address 6C:1D:EB:0C:AD:DD was not found.
```

Figure 3.6: Discovering ANNA, without being able to connect to it.

ANNA-B112 module, I figured that it should also be capable of connecting to the device as well. Second, there was the possibility that the Armory Device actively rejected connection attempts from my laptop, because of some settings prohibiting external device connection. Since the picocom interface provided detailed logs of all Bluetooth procedures, any rejection would have likely triggered explicit notifications, which I never observed. Finally, the most plausible scenario was that the device simply went into sleep mode before my connection attempt could complete. This suspicion was reinforced by observing the module's irregular advertising through the "U-blox BLE" application on my iPhone, which showed ANNA appearing and disappearing unpredictably. To address this, I adjusted my script to immediately verify if a newly discovered device was the ANNA module and, if so, connect to it immediately. Rather than performing a complete discovery scan first and then searching the discovered devices afterward. This direct verification helped significantly reduce the delay between discovery and connection attempts.

Once I successfully connected to the ANNA module, I could proceed with exploring additional capabilities of the module directly from my laptop. According to the Medium blog that guided part of my approach, the next critical step was identifying the device's *characteristics*. At the time, I was unfamiliar with the concept, prompting me to seek clarification from ChatGPT. The explanation I received clarified that characteristics in a BLE device represent small data points exposed by the device, indicating how external systems should communicate with it. Characteristics typically have properties such as *read* and *write*, which allow users to send data to the device or receive updates through notifications. Additionally, ChatGPT mentioned that subscribing to these notifications would allow me to receive automated feedback directly from the device. This realization greatly motivated me, as it suggested a viable pathway toward implementing remote command execution on a Linux machine via Bluetooth.

Parallel to establishing a stable BLE connection, I explored the possibility of executing remote commands through BLE communication. My research indicated that an SPS (Serial Port Service) connection or a UART transparency mode needed to be active on the target device to facilitate such communication. Consequently, I examined the ANNA manual and identified the relevant subsection titled *Server Configuration*, which enabled me to configure and activate the SPS connection on the ANNA module. These configuration details and the module's response to these adjustments are discussed in Section 2.2.2.

### 3.5.3 Exploring services of ANNA

Having achieved a reliable connection to ANNA, the subsequent task was to iterate through its available services and their respective characteristics, aiming to identify a suitable characteristic that would enable command execution. To effectively explore these characteristics, I first needed to understand the concept of UUIDs (Universally Unique Identifiers), which uniquely identify specific Bluetooth services. With this

understanding, I developed a dedicated script tailored specifically to scan for UUIDs and extract their properties from the Armory Device (refer to Appendix A.4).

The results (as you can see in table 3.1 and 3.2) had one service that seemed to be particularly good fitted for the purpose of communication with properties like "read, write-without-response, write, notify". Which had all the properties required to read current state, write commands to the current state, and get notifications about the "writes".

After successfully identifying the service UUID, I attempted a direct connection and write operation using this UUID. Although the connection appeared successful (as depicted in Figure 3.7), the commands sent were not reflected in the Picocom interface used to monitor the ANNA module. Clearly, there was still an unresolved issue.

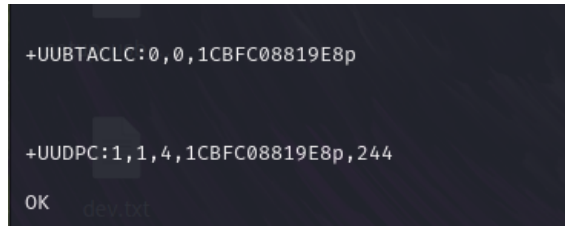


Figure 3.7: AT confirmation message from the ANNA module seen in Picocom interface.

Table 3.1: BLE services and their characteristics

Service UUID	Characteristic UUID
00001800-0000-1000-8000-00805f9b34fb	00002a00-0000-1000-8000-00805f9b34fb
00001800-0000-1000-8000-00805f9b34fb	00002a01-0000-1000-8000-00805f9b34fb
00001800-0000-1000-8000-00805f9b34fb	00002a04-0000-1000-8000-00805f9b34fb
00001800-0000-1000-8000-00805f9b34fb	00002aa6-0000-1000-8000-00805f9b34fb
00001801-0000-1000-8000-00805f9b34fb	00002a05-0000-1000-8000-00805f9b34fb
0000180a-0000-1000-8000-00805f9b34fb	00002a29-0000-1000-8000-00805f9b34fb
0000180a-0000-1000-8000-00805f9b34fb	00002a24-0000-1000-8000-00805f9b34fb
0000180a-0000-1000-8000-00805f9b34fb	00002a26-0000-1000-8000-00805f9b34fb
0000180a-0000-1000-8000-00805f9b34fb	00002a28-0000-1000-8000-00805f9b34fb
2456e1b9-26e2-8f83-e744-f34f01e9d701	2456e1b9-26e2-8f83-e744-f34f01e9d703
2456e1b9-26e2-8f83-e744-f34f01e9d701	2456e1b9-26e2-8f83-e744-f34f01e9d704

### 3.5.4 The Escape Character

Resolving this issue proved particularly challenging, taking approximately one month of my research. Initially, I was uncertain if my approach would even function correctly and anticipated several potential obstacles. These included improper initialization of the BLE connection, formatting constraints such as requiring specific

Table 3.2: The properties of the characteristics.

Characteristic UUID	Properties
00002a00-0000-1000-8000-00805f9b34fb	read
00002a01-0000-1000-8000-00805f9b34fb	read
00002a04-0000-1000-8000-00805f9b34fb	read
00002aa6-0000-1000-8000-00805f9b34fb	read
00002a05-0000-1000-8000-00805f9b34fb	indicate
00002a29-0000-1000-8000-00805f9b34fb	read
00002a24-0000-1000-8000-00805f9b34fb	read
00002a26-0000-1000-8000-00805f9b34fb	read
00002a28-0000-1000-8000-00805f9b34fb	read
2456e1b9-26e2-8f83-e744-f34f01e9d703	read, write-without-response, write, notify
2456e1b9-26e2-8f83-e744-f34f01e9d704	write-without-response, write, notify

termination characters (e.g., "\r\n"), or potential issues in how the ANNA module processed incoming commands. Additionally, I considered the possibility of overlooked configuration settings necessary for the communication.

After a prolonged period without progress and nearing resignation, I revisited the ANNA manual with the intention of identifying any overlooked configuration options. Although unsure precisely what I was seeking, I intuitively searched for terms related to command processing, utilizing the browser's word search function for the keyword "command." Fortunately, I encountered the obscurely located section named "Escape Character S2." This section described how the ANNA module recognized the escape character, specifically requiring the configured escape character ("+") to be transmitted three times consecutively within a single data frame.

Additionally, this section referenced another configuration subsection titled *Configuration section* (5.6 +UDCFG), mentioning that some modules allowed adjustments to the timing required for entering data mode. Intrigued by this timing parameter, I investigated further and discovered detailed explanations in section 5.6.2 of the AT commands manual. Specifically, I learned about the "escape sequence timing," which mentioned a one-second period of silence before and after transmitting the escape character sequence for the ANNA module to recognize the switch to command mode communication.

Implementing these timing adjustments in my script finally resolved the issue, enabling successful command execution from my laptop to the ANNA module, with commands correctly appearing in the Picocom interface as intended.

### 3.5.5 The final script

Roughly speaking, the script *USB\_Connect.py* (see Appendix A.5) begins with my latest update that makes it compatible with any operating system that supports

Python. This update takes care of automatically installing all required packages using Python's `importlib` library, removing the need for manual setup.

```
# Auto-install dependencies if missing
_deps = ("bleak",)
for pkg in _deps:
    try:
        importlib.import_module(pkg)
    except ImportError:
        subprocess.check_call([sys.executable,
                                "-m",
                                "pip",
                                "install",
                                "--upgrade",
                                pkg])
```

The top part of the script is used to import all necessary libraries and define global variables, such as the friendly name of the ANNA BLE module for faster identification, along with the specific service and characteristic UUIDs required for communication.

```
import asyncio
from bleak import BleakScanner, BleakClient

TARGET_NAME = "ANNA-B1-0CADDD"
CONNECTION_HOLD = 5 # Time (in seconds) to keep the connection open
buffer = "" # Buffer to store the response from the device.
started = False # Flag to indicate if the response has started.

# Custom service/characteristic UUIDs as discovered:
SPS_SERVICE_UUID = "2456e1b9-26e2-8f83-e744-f34f01e9d701"
SPS_WRITE_CHAR_UUID = (
    # Supports write and notify
    "2456e1b9-26e2-8f83-e744-f34f01e9d703"
)
SPS_NOTIFY_CHAR_UUID = (
    # Same characteristic used for notifications
    "2456e1b9-26e2-8f83-e744-f34f01e9d703"
)
```

From there, the `main()` function kicks in. It initializes the Bleak scanner to check if the ANNA module is available. If the device is found, the `main()` function proceeds by calling `connect_device()`, which uses `BleakClient()` to establish a connection. Once the connection is set up, the script prompts the user for input. The input is then parsed and passed back to the `connect_device()` function, which handles sending the command to the USB Armory device where the ANNA module is implemented.



## **main()**

As previously mentioned, the `main()` function is responsible for initializing and performing the Bluetooth scan using `BleakScanner()`. It does so by registering the callback function `detection_callback()` through `register_detection_callback()`, which processes each device found during the scan. If the ANNA module is detected, its friendly name is matched, and the dynamic global variable `DeviceHolder` is set accordingly. After the scan, the script checks this variable to determine if ANNA-B112 is available. If the device is not found, the script ends with the message "Device not found."

## **detection\_callback()**

Each time the `detection_callback()` function is triggered during the scan, it compares the name of the currently detected device against the `TARGET_NAME` defined in the script. If the name matches ANNA-B112, the function sets the device property of the `DeviceHolder` class to the correct device instance. This approach simplifies and accelerates the connection process by storing the appropriate target device for later use.

## **connect\_device()**

The `connect_device()` function establishes a notification channel using the characteristic UUID that supports notifications. This is necessary to receive status updates and feedback from the USB Armory device via ANNA. The function sets `notification_handler()` as the callback to handle incoming messages.

Once the notifier is active, the script executes the escape character sequence required by ANNA to switch from data mode to command mode. This sequence follows the instructed sequence required from the Armory Device to initialize the Command Mode Communication (as mentioned in previous sections). Consisting of a one-second silence, followed by three consecutive plus signs (+++) sent in a single frame, and then another one-second pause.

If this transition is successful, the function enters a while loop that continuously prompts the user for input. If the user types exit, the session ends. Otherwise, the input command is parsed and sent to the ANNA module via `write_gatt_char()`, targeting the write-enabled characteristic. After the command is sent, the script briefly sleeps before re-entering the loop for the next user command.

## **notification\_handler()**

The `notification_handler()` function is tasked with interpreting and displaying the feedback messages returned from the ANNA terminal. It is registered by `connect_device()` and is called each time a notification is received. The function begins by accessing the global variables `buffer` and `started`, then decodes the received data.

It checks whether the decoded data contains both the `!start!` and `!end!` markers. These are my custom delimiters I implemented to signify the beginning and end of a message. If both markers are present in a single payload, the complete message is printed directly. If only one of the two markers is found (either `!start!` or `!end!`), the function buffers the data appropriately, either initializing or finalizing

the message, supposedly ensuring that even fragmented messages are reconstructed correctly before being displayed.

This approach offers a robust and user-friendly mechanism for receiving clear and ordered messages. However, it is not well-suited for large data transfers; for instance, attempting to cat a file with more than five lines may result in the `!end!` signal not being received, thus preventing the message from being displayed at all.

### **terminal\_emulation()**

The `terminal_emulation()` function serves a simple but essential role: it prompts the user for input, checks whether the entered command is `exit`, and if not, transforms the command into a byte string. It adds a custom start symbol `"#"` at the beginning (to aid recognition on the ANNA-B112 side) and appends a newline character (`\n`) to properly end the command before it is sent.

## **3.6 Bridging Bluetooth with the Rest of the Device**

The ANNA module provides a standalone Bluetooth "administrator" that functions independently from the rest of the Armory Device having all Bluetooth processing handled by the ANNA-B112 module. The module effectively acts as a security "guard," positioned between the publicly accessible Bluetooth antenna for the Armory Device user and the private USB Armory Device itself. This design inherently prevents direct Bluetooth interaction with the system, making it impossible for me to directly implement solutions that rely on Bluetooth adapters (like `bluez` or `bluetoothctl`), as these adapters are not exposed to the underlying operating system. Consequently, all Bluetooth communication must pass through the ANNA module, effectively isolating Bluetooth interactions from the main device.

This isolation required me to find a workaround to enable remote control of the device via Bluetooth. Figure 3.8 illustrates my solution, highlighting the interdependence between the scripts I developed and the underlying system components. This approach is detailed in the following sections.

### **3.6.1 Bypassing ANNA**

As previously mentioned, I consistently utilized an interface called `picocom`, which allowed me to configure ANNA's behavior using AT commands. `Picocom` also provided live status updates whenever ANNA processed incoming messages. Additionally, I noticed that `picocom` could effectively monitor all communication between my laptop and the ANNA module, displaying everything sent to or received from the SPS service.

This observation led me to an idea on how to effectively bypass ANNA using `picocom`. Initially, my approach involved configuring `picocom` to log all communication into a `".log"` file, which my script could then process. The script was intended to filter the logged commands, extract relevant kernel commands, and subsequently execute them directly on the Armory Device's terminal.

However, during implementation, I discovered that `picocom` simply read the input from the `ttymxc0` port, which was also accessible by the system kernel. Consequently, this meant the port received inputs simultaneously from my laptop and



Python and executed in the shell.

This method also has a protective function: it prevents any user command from being misinterpreted by the ANNA module as an actual AT command, which could otherwise cause the module to change its notification settings—or worse, brick the device, cutting off the only available access channel.

With that, I was able to reliably identify incoming commands and ensure only valid commands were processed. However, a new issue appeared: ANNA would still return an "ERROR" message after each user command starting with "#". To fix this, I implemented a filter to discard irrelevant notifications and separate the kernel response intended for the user, from default noise generated by ANNA.

It's important to note that disabling ANNA's echo function altogether caused a different problem—no data would be read from `/dev/ttymx0` anymore. This was because ANNA fully manages the Bluetooth communication protocol, and turning off echo resulted in python being unable to read the data from the port. So instead of disabling echo, I added a second layer of control—similar to the "#" prefix—by introducing a custom symbol to signal the start of a response. Later complications, such as larger data chunks being transferred out of order, led me to also define an end-of-message symbol. Both are visible in the `USB_Connect.py` script (see Appendix A.5).

To forward the kernel's response back to the user via Bluetooth, I used the Python subprocess library to execute commands and handle both standard output and error output. Then, using the same serial library, I sent the response through `/dev/ttymx0` by writing the message in the following sequence: start symbol, response payload, and end symbol. This allowed the ANNA module to send the message over BLE back to the user. All of this is implemented in my script called `listening.py` (Appendix A.7), which I describe in more detail in the following subsections.

### 3.6.3 `listening.py` in Greater Detail

The main goal of the `listening.py` script is to: detect incoming user commands, execute them in the system kernel, handle the output, and send the response back to the user in a proper and recognizable format.

#### Listening on the Port

With the use of Python's `try:` and `except:` structures, the script is able to manage the communication process robustly, even in the case of errors. In the initial `try:` block, I instantiate the listening interface using the serial library, applying key settings—like baudrate and timeout—that I retrieved from the official GitHub repository of the USB-C Armory Mk II device [1].

```
try:
    # Open the serial port for reading
    # (and optionally writing, if required)
    with serial.Serial('/dev/ttymx0', baudrate=115200, timeout=1) as ser:
        print("Listening on /dev/ttymx0...")
```

## Reading the Serial

After setting up the serial connection, the script enters a while loop that runs indefinitely (while True:). Inside this loop, the script continuously listens to the port using the `.readline()` method. This method both keeps track of the current position in the stream and fetches the next available line from the serial port buffer.

If a new line is available, the script attempts to decode the incoming data and parse it into a string. If the decoding fails it simply parse the line into string format and proceeds with that string. This ensures that the system does not crash or stall due to encoding issues, and that all received data is processed in one form or another.

```
while True:
    # Check if data is waiting in the serial buffer
    line = ser.readline()
    if line:
        # Read all available data
        try:
            # Decode the incoming bytes into a string.
            text = data.decode('utf-8', errors='replace')
            text = str(text)
        except Exception as e:
            text = str(line)
```

## String Processing

To process the incoming strings correctly, I use the `.strip()` method, which trims any leading and trailing whitespace or newline characters. This gives me a cleaner string to work with and allows me to iterate through the string and access specific indexes for conditional checks.

Although using “absolute position checks”—where you check for specific characters at fixed positions (like the one I used)—can be a bit fragile (since formatting differences or noise might make the character appear in a different position, which would result in the entire command being unrecognizable), I found this method to be quite effective in practice. Once a command is successfully identified, I know it starts with `"b'#"` and ends with a newline character and a single quote mark: `"\n'".` These parts are removed using one simple instruction: `user_command = fixed_text[3:-3]`.

## Executing the Command

At this point, the string has been validated as a user command and is ready to be executed. If the command begins with `cd`, the `cd` prefix is removed. By leveraging Python’s `os` library, I change the current working directory to the user-provided path using the `chdir()` function. Once the directory has been updated, I encode an acknowledgment as a byte string and transmit it back to the user, framed by the previously defined start and end symbols.

I pass the (possibly modified) command to the system shell using Python’s `subprocess` library. If the command executes successfully, it returns a `.stdout` response, which is captured in the `ker_resp` variable. This response is then sent

through the serial port using the `write()` method of the `serial` library, again enclosed between the start and end markers.

After sending the response, the script pauses (`sleep(1)`) to allow the ANNA module time to complete the transmission. Finally, I reset the input buffer with `ser.reset_input_buffer()` to prevent the script from reading its own outgoing message on the next loop iteration.

If the provided string is not a valid command, the shell returns an error via `.stderr`. In that case, the error message is handled similarly—sent back over the serial port with the same formatting.

Important note: While writing this part, I realized that the current error-handling branch does not reset the input buffer after sending the error message. This could lead to the script getting stuck in a loop—constantly reading the same error line it just sent. To avoid this, I recommend adding a `ser.reset_input_buffer()` call immediately after handling the error message, just like in the successful command case.

```
if execution.stdout:
    ker_resp = execution.stdout
    print("Command Output:\n", ker_resp)

    # Formating the kernel_feedback string to send it to the user:

    print("Sending the following command:", ker_resp)
    #Begin the incoming message:
    ser.write(ker_start)
    ser.write(ker_resp)
    ser.write(ker_end)
    sleep(1)
    ser.reset_input_buffer()

if execution.stderr:
    error = execution.stderr
    print("Error Output:")
    print(error) # Debugging output in Kali.
    ser.write(ker_start)
    ser.write(error)
    ser.write(ker_end)
    sleep(1)
```

## 3.7 HID Emulation

To leverage HID emulation on the Armory Device, I first familiarized myself with the script provided by Collin Mulliner [14], which is included in the Armory Device's GitHub repository. This script instructs the device to change its behavior, presenting itself as an HID keyboard rather than its default USB-C Armory Device MK II gadget. Figure 3.9 shows the Oracle VM settings I used during this process. Mulliner's script effectively transforms the device from an Ethernet Gadget into

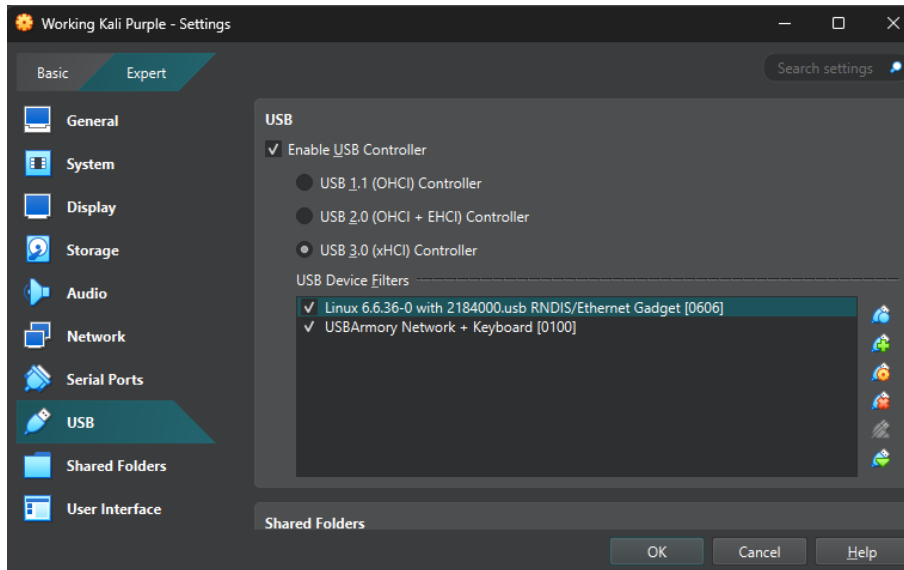


Figure 3.9: Oracle VM settings connecting USB-C Armory Device MK II to my Kali Linux VM

a combined network and keyboard interface. His repository contains several bash scripts for different use cases; I employed `hidnet.sh` to preserve network emulation (allowing SSH access if the BLE connection failed) while simultaneously exposing the device as a keyboard.

### 3.7.1 Explaining the `hidnet.sh` Script

To understand how I leveraged and modified this script, we first review its default behavior. A brief analysis of Mulliner’s `hidnet.sh` (see Appendix A.1.2) is presented below.

The script begins by requesting `sudo` privileges to avoid **ERROR: Permission Denied** when manipulating kernel modules and configuration directories. It then removes the modules responsible for the Ethernet Gadget and loads those needed for HID and ECM:

```
modprobe -r g_ether usb_f_ecm u_ether
modprobe usb_f_hid
modprobe usb_f_ecm
```

Next, the script creates configuration directories under: `"/sys/kernel/config/usb_gadget/"`, instantiates `g1`, and sets the `hid` and `ecm` functions for the `usb0` interface. Finally, it specifies the HID property as a keyboard, explicitly informing the kernel and host to recognize the device as an HID keyboard.

```
cd /sys/kernel/config/
mkdir usb_gadget/g1
cd usb_gadget/g1
mkdir configs/c.1
mkdir functions/hid.usb0
mkdir functions/ecm.usb0
```

```

echo 1 > functions/hid.usb0/protocol
echo 1 > functions/hid.usb0/subclass
echo 8 > functions/hid.usb0/report_length
echo -ne "\x05\x01\x09\x06\xA1\x01\...
...\x29\x65\x81\x00\xC0" > functions/hid.usb0/report_desc
mkdir strings/0x409
mkdir configs/c.1/strings/0x409

```

The next few lines handle how the gadget presents itself in a human-readable way, for both the user and the connected host. These lines declare product information, power consumption, and the active configuration:

```

echo 0x1d6b > idVendor    # Linux Foundation
echo 0x0104 > idProduct   # Multifunction Composite Gadget
echo 0x0100 > bcdDevice   # v1.0.0
echo 0x0200 > bcdUSB      # USB2
echo "deadbeef9876543210" > strings/0x409/serialnumber
echo "USBArmory" > strings/0x409/manufacturer
echo "USBArmory Network + Keyboard" > strings/0x409/product
echo "Conf1" > configs/c.1/strings/0x409/configuration
echo 120 > configs/c.1/MaxPower

```

The script then binds the functions to the configuration by creating symbolic links. This ensures that both the keyboard interface (`hid.usb0`) and the ECM interface (`ecm.usb0`) are included in configuration `c.1`. Finally, it attaches the gadget `g1` to the physical USB controller, altering how the USB-C Armory Device MK II is presented to the host:

```

ln -s functions/hid.usb0 configs/c.1/
ln -s functions/ecm.usb0 configs/c.1/
echo ci\_hdrc.0 > UDC

```

### 3.7.2 Modifying the `hidnet.sh` Script

**Disclaimer:** This work extends beyond my core expertise; some modifications may be redundant. Also with the aid of modern tools like ChatGPT, I enhanced the script for safer and more reliable execution.

The most significant change I introduced was the initialization of the network interface, effectively opening the “network doors” on the USB Armory Device:

```

USB\_IF="usb0"
ip link set \${USB\_IF} up
ip addr add 10.0.0.1/24 dev \${USB\_IF}

```

I also improved error handling by suppressing failures when loading or removing modules that may already be in the desired state:

```

modprobe -r g\_ether usb\_f\_ecm u\_ether 2>/dev/null || true
modprobe usb\_f\_hid 2>/dev/null || true
modprobe usb\_f\_ecm 2>/dev/null || true

```



Next, I verify whether the `g1` gadget directory exists. If it does, I unbind and remove it safely before reconfiguration:

```
if \[ -d /sys/kernel/config/usb\_gadget/g1 ]; then
echo "" > /sys/kernel/config/usb\_gadget/g1/UDC 2>/dev/null || true
rm -rf /sys/kernel/config/usb\_gadget/g1
fi
```

I then group the procedure for defining HID and ECM functions. For HID, I create the function directory and set protocol, subclass, and report length. The report descriptor is defined in hexadecimal:

```
mkdir functions/hid.usb0
echo 1 > functions/hid.usb0/protocol
echo 1 > functions/hid.usb0/subclass
echo 8 > functions/hid.usb0/report\_length
echo -ne "\x05\x01\x09\x06\xA1...
...\x29\x65\x81\x00\xC0" > functions/hid.usb0/report\_desc
```

To create the ECM (Ethernet) function, I simply make its directory:

```
mkdir functions/ecm.usb0
```

Finally, I bring up the `usb0` interface with IP `10.0.0.1/24` for SSH access. This interface is exclusively exposed to the directly connected host via the USB port.

### 3.7.3 Using the Modified `hidnet.sh` Script for Keystroke Injection

Now that the USB Armory Device is recognized as a keyboard, the host accepts keystrokes as if they were entered by a user. This behavior is exactly why we configured the device as a reliable HID keyboard. To inject keystrokes, I use the compiled version of `string2hid.c`, also provided by Collin Mulliner. Briefly, this program accepts a single argument of arbitrary length and translates it into HID reports. Using it on my Kali Linux VM, I was able to perform tasks such as launching a web browser and searching for a specified URL, or opening a terminal and typing “Hello World.” This demonstrates HID emulation with virtually no limitations—equivalent to an SSH shell on a remote host.

The final requirement for fully remote keystroke injection is to execute the modified `hidnet.sh` script at boot. This ensures that, upon power-up, the device immediately presents itself as a combined network+keyboard gadget, allowing instant SSH access and arbitrary code injection.

## 3.8 Preparing Automatic Launch on Boot

To start the application automatically after boot, I created systemd services—analogous to Bluetooth or network services—that can be enabled or disabled with `systemctl`. This approach lets me control the state of the HID gadget and BLE listener independently.

First, I copied the modified `hidnet.sh` script to the system binary directory (`/usr/local/bin`) for global accessibility, alongside other administrative scripts. I placed the BLE listener (`listening.py`) in `/opt/rem_com_exec`. Once the directory structure was organized, I created two service unit files in `/etc/systemd/system`: `usb-gadget-hid-ecm.service`[A.8.1](#) and `rem-com-exec.service`[A.8.2](#). The system attempts to start these services during boot as follows.

`usb-gadget-hid-ecm.service` ensures the USB gadget configuration is mounted before running:

```
[Unit]
After=local-fs.target sysinit.target
Wants=local-fs.target

[Service]
Type=oneshot
ExecStart=/usr/local/bin/usb_gadget_hid_ecm.sh
RemainAfterExit=yes
```

`rem-com-exec.service` runs only after `usb-gadget-hid-ecm.service` is active, launches the BLE listener, and restarts on failure:

```
[Unit]
After=usb-gadget-hid-ecm.service

[Service]
ExecStart=/opt/rem_com_exec
Restart=on-failure
RestartSec=5
```

After creating both units, I reloaded `systemd` and enabled the services:

```
sudo systemctl daemon-reload
sudo systemctl enable usb-gadget-hid-ecm.service
sudo systemctl enable rem-com-exec.service
```

Finally, I rebooted the device, allowing a complete keystroke injection immediately after startup.

# Chapter 4

## Results & Future Work

This chapter summarizes the key outcomes of my research and discusses their broader implications for cybersecurity awareness and practice. I address how user unawareness can significantly diminish the effectiveness of even the most sophisticated security implementations, emphasizing the need of applying some pedagogical approaches in cybersecurity documentation which are intended for general public.

Furthermore, I evaluate the practical limitations and countermeasures relevant to the remote keystroke injection methods developed in my thesis. Additionally, I outline potential enhancements to my existing scripts and suggest promising directions for future exploration and further development of the USB-C Armory Device MK II, highlighting areas that could be of interest for cybersecurity research and practice.

### 4.1 My Results

I successfully connected the Armory Device MK II to my host computer and forwarded its USB-C traffic to a Kali Linux virtual machine (VM) for a controlled demonstration. Figure 4.1 illustrates the terminal state before SSH connection, while Figure 4.2 shows a terminal with an ongoing SSH connection.

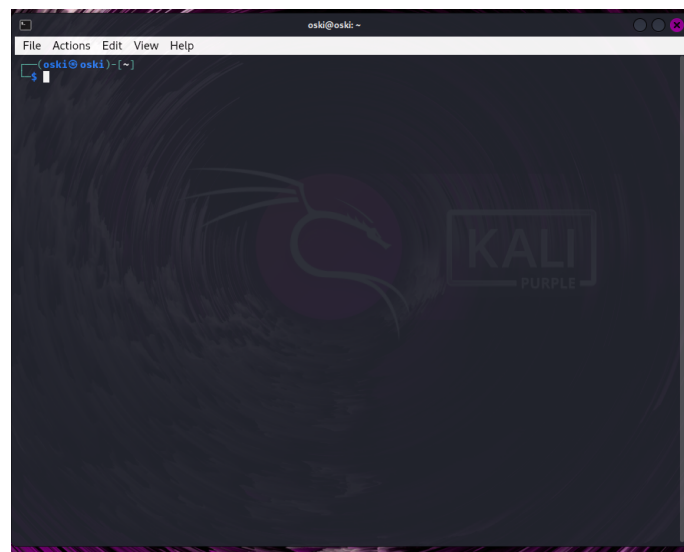


Figure 4.1: Terminal not connected to the Armory Device

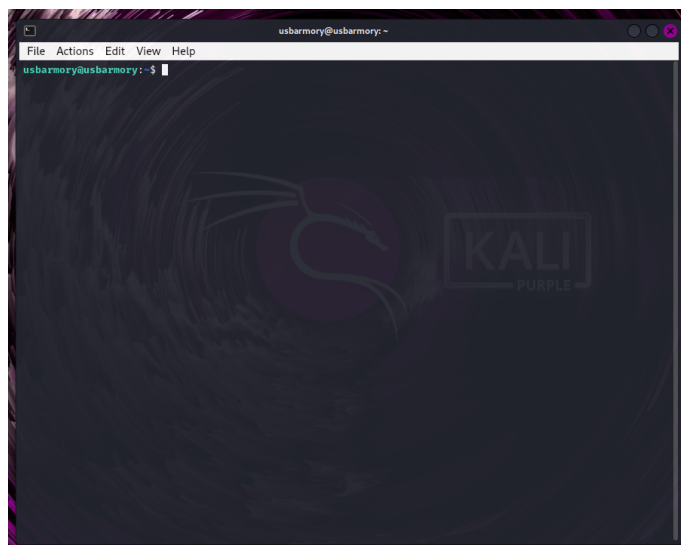


Figure 4.2: Terminal connected to the Armory Device

To perform keystroke injection, I utilized the compiled `string2hid.c` binary named `Inject`, located in `/home/usbarmory/GIT_Backup/`. Additionally, the BLE listener script (`listening.py`) runs from `/opt/rem_com_exec`. My Python script includes a `cd` command, allowing it to navigate to any directory on the host filesystem, showcasing the flexibility and adaptability of the USB-C Armory Device MK II. Although rapid keystroke injections occasionally resulted in missed characters, stable command injection was reliably achievable when executed at a moderate pace. Figure 4.3 demonstrates keystrokes injected into the unconnected terminal, Figure 4.4 illustrates background device-side processing via `/dev/ttymxc0`, and Figure 4.5 presents the client-side output.

#### 4.1.1 The Impact of My Results

Social engineering attacks and badUSB exploits remain highly relevant threats in cybersecurity, primarily because ordinary users are neither adequately informed nor actively interested in understanding these risks. This thesis clearly illustrates how challenging it can be (even for someone with a strong background in cryptology) to access and interpret cybersecurity information due to its complexity and poor readability.

I managed to achieve substantial results within a one-year timeframe due to my motivation and daily dedication. However, typical users who might genuinely care about their privacy rarely have the luxury of dedicating such significant time and effort. For these individuals, navigating cybersecurity resources can become overwhelming and discouraging, ultimately leading them to ignore proactive security measures altogether.

Attackers benefit from clear incentives like financial gain or strategic advantages, making them inherently motivated to learn and exploit vulnerabilities efficiently. On the other hand, preventing such attacks provide no immediate reward, making proactive cybersecurity efforts challenging for users without clear and accessible information.

The current state of my script provides a practical demonstration of how simply a trusted device can miss use this trust. However it is only for performing tests on

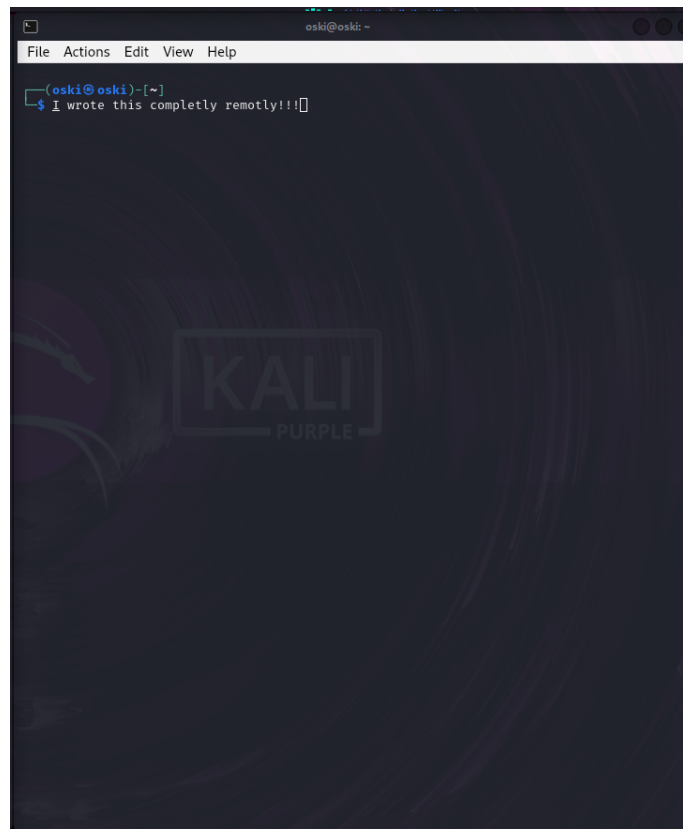


Figure 4.3: Injected script into the not connected terminal

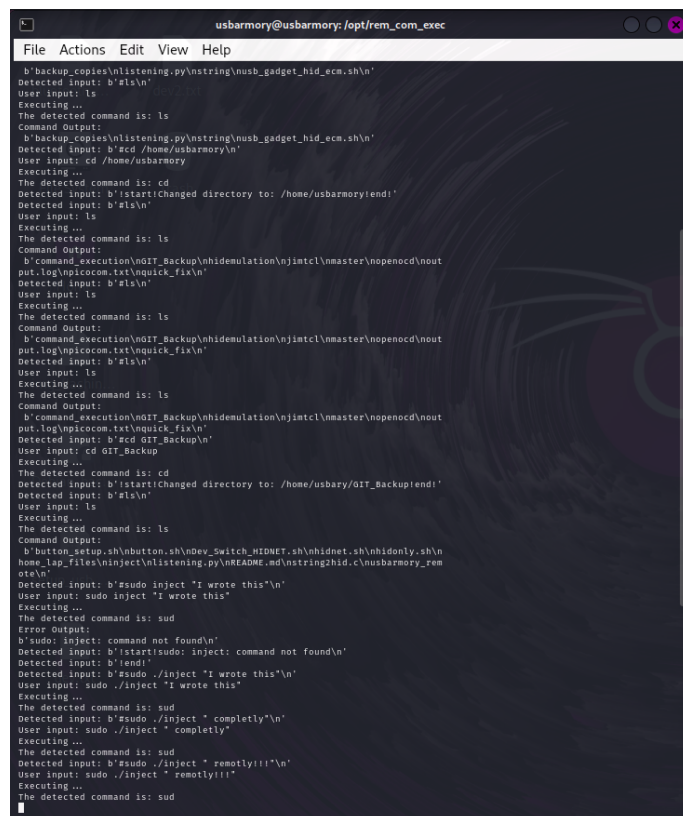


Figure 4.4: Background Python string processing

```

Type your input: ls
-----START-----
.....
Type your input: ls
.....
-----START-----
Type your input: ls
.....
-----START-----
command_execution
GIT_Backup
hidemlation
jimtbl
master
openocd
output.log
picocom.txt
quick_fix
-----END-----

Type your input: cd GIT_Backup
-----START-----
.....
Changed directory to: /home/usbarmory/GIT_Backup
-----END-----

Type your input: ls
-----START-----
.....
button_setup.sh
button.sh
Dev_S#####.s#####ng.#####armory_remote
-----END-----

Type your input: sudo inject "I wrote this"
-----START-----
sudo: inject: command not found
-----END-----

Type your input: sudo ./inject "I wrote this"
.....
Type your input: sudo ./inject "completely"
.....
Type your input: sudo ./inject "remotly!!!"

```

Figure 4.5: Python terminal seen by the user connected to the Armory Device

personal equipment, in an ethical manner. This demonstration gives an intuitive display of how dangerous cyber threats might be. Also highlighting the fact that cyber threats does not solely come from the digital channel.

### **4.1.2 Trusted Device Vulnerabilities**

Typically, devices become "trusted" once physically connected or paired via Bluetooth. This research emphasizes how easily this trust can be exploited. Malicious devices capable of emulating trusted devices can effortlessly gain unauthorized access, exploiting the user's assumption that physical or Bluetooth connections inherently imply safety.

### **4.1.3 Highlighting the Weakest Link**

A key insight from my research is that even the most advanced security protocols become entirely ineffective if there is an overlooked physical or digital vulnerability. Cybersecurity tends to emphasize technical solutions like protocol enhancements, cryptographic implementations, or software defenses, while often neglecting the importance of user awareness. Just as Health, Safety, and Environment (HSE) training is crucial in industries like construction, cybersecurity awareness should be equally prioritized within organizations dealing with sensitive user data.

### **4.1.4 Countermeasures and Limitations**

#### **Keystroke Injection Countermeasures**

One of the most effective defenses against keystroke injection attacks via badUSB is simple physical prevention—restricting access to device USB ports. However, this is not always practical in environments where USB access is necessary. Operating systems like Windows and macOS implement various software-based protections, such as blocking unknown USB Vendor IDs or preventing payload execution. Yet, these countermeasures often have significant downsides, including complexity, user inconvenience, and vulnerability to dynamically spoofed IDs.

To illustrate this, Figure 4.6 summarizes common keystroke injection prevention techniques and their limitations, emphasizing the critical need for proactive user education to reduce vulnerability.

#### **Theoretical Limitations**

This research aims solely at educational and testing purposes and does not endorse unethical activities. It offers insight into the learning curve faced by newcomers in cybersecurity. However, acknowledging theories like Bayesian Brain Theory highlights that individual experiences and perceptions vary greatly, suggesting the necessity for further research into enhancing cybersecurity education.

#### **Practical Limitations**

Due to limited documentation and time constraints, my research on developing a BLE connection with the Armory Device MK II did not reach the desired levels of automation, user-friendliness or security robustness. Numerous dead ends slowed

Countermeasure	OS	Weakness or Downside
UAC Secure Desktop	Windows	Only protects elevation prompts; user-space payloads are unaffected.
Device ID Whitelisting (GPO or Registry)	Windows	Can be bypassed by spoofing known device IDs; requires admin rights to configure.
WDAC / Device Guard	Windows	Difficult to configure; doesn't prevent the keystroke injection itself.
EDR Tools (e.g., Defender for Endpoint)	Windows	Reactive; doesn't stop initial injection, and requires enterprise setup.
TCC / Input Monitoring Consent	macOS	Applies only to software; hardware keystroke injection bypasses it.
Secure Keyboard Entry	macOS	Blocks software keyloggers, but does not prevent injected keystrokes.
Gatekeeper and SIP	macOS	Prevents unauthorized code, but not native system command execution via USB input.
USB Port Disable / BIOS Lock	Both	Highly restrictive; impractical for users who rely on USB input.
3rd-Party USB Authorization Tools	Both	Not native to the OS; can reduce usability and require manual maintenance.

Figure 4.6: Keystroke Injection Countermeasures and their limitations



progress, underscoring the need for clearer guidance and accessible resources in this field.

Additionally, my relative inexperience with BLE development and the lack of readable documentation significantly impacted the depth and pace of my progress, highlighting the importance of improved educational materials and resources.

## **Application Drawbacks**

On the practical side, my application currently handles only small data streams at a time (20-30 characters), requiring close proximity (3-5 meters) for effective Bluetooth communication. Furthermore, manual initiation of the keystroke injection application is necessary before using the script, reducing its ease of use. The script also lacks user notifications regarding connection status, potentially causing confusion when a connection is lost or terminated.

Addressing these practical issues in future developments could significantly improve usability and reliability, further emphasizing the need for clear, intuitive cybersecurity solutions accessible to all users.

## **4.1.5 Improvements and Future Work**

### **Security Enhancements**

Several improvements could significantly strengthen my application's security, showcasing its inherent flexibility. The ANNA-B112 module offers multiple encryption algorithms and key negotiation protocols that were not fully utilized in my current implementation. Employing these cryptographic tools would significantly reduce the risk of Bluetooth spoofing, connection hijacking, or similar security breaches. Additionally, leveraging ANNA's built-in handshake protocol, which requires user authentication before granting access, would further protect the device against unauthorized use.

### **Script Development and Automation**

Currently, my script requires further refinement before it could perform real keystroke injection prevention testing tool. Below are several suggestions for future development:

Firstly, automating the keystroke injection process would greatly enhance usability. Instead of manually navigating to the directory containing the injection file, the user could be immediately prompted to begin injecting keystrokes into the target device. Additionally, providing an option to seamlessly switch back to "Command Armory Device Mode" would increase flexibility and user convenience.

Further improvements could include adding predefined automated functions, such as "Open Command Prompt on Target" or "Open Link on Target." These user-friendly commands would simplify testing for individuals without extensive technical knowledge.

Improving the BLE connection itself could also significantly enhance performance. Extending the effective range and optimizing packet handling would enable the transmission of longer data strings, improving usability and reducing the need for sending multiple short messages.

Another promising area for further development would be combining remote keystroke injection with the Armory Device’s mass storage emulation capabilities. This combination could allow rapid exfiltration or targeted writing of data to and from the target device.

**Disclaimer:** These recommendations are presented strictly for educational and demonstration purposes, aiming to increase awareness about the versatility and broad attack surface of cybersecurity threats. They emphasize the importance of bridging the knowledge gap between cybersecurity professionals and everyday users. They are not intended to encourage malicious or unethical behavior.

### **Further Exploration of Vendor Defined Messages**

Before installing the operating system on the Armory Device, I briefly explored communication using Vendor Defined Messages (VDM). Although my initial exploration was limited, I successfully extracted some byte-level data from the device using a basic script. This approach reveals a potential vulnerability common in many USB devices, as previously demonstrated by Gunnar Alendal in his paper [15], which I have briefly interpreted and summarized in my paper [16].

Future research could explore the automation of communication using vendor-defined messages, leveraging the USB-C Armory Device MK II’s capabilities. This approach could delve deeper into this niche but significant area of USB security vulnerabilities, further expanding our understanding of potential threats and necessary countermeasures.

# Bibliography

- [1] Barisani, Andrea or InversePath. Mk ii introduction. [Online]. Available: <https://github.com/usbarmory/usbarmory/wiki/Mk-II-Introduction>
- [2] Maher Alsharif, Shailendra Mishra, and Mohammed AlShehri, “Cybersecurity awareness and phishing attacks,” *Computer Systems Science and Engineering*, vol. 40, no. 3, 2022, accessed: 2025-05-07. [Online]. Available: <https://www.techscience.com/csse/v40n3/44582/html>
- [3] Björn Hjelm, Niklas von Maltzahn and Michael Fischer, “Cybersecurity communication for the general public—challenges and opportunities,” *Journal of Cybersecurity and Privacy*, vol. 1, no. 4, pp. 649–661, 2021, accessed: 2025-05-07. [Online]. Available: <https://www.mdpi.com/2624-800X/1/4/34>
- [4] Unknown, “A device for executing keystroke injection attacks,” 2022, accessed: 2025-04-20. [Online]. Available: <https://tore.tuhh.de/dspace-cris-server/api/core/bitstreams/c8e23b88-214b-4006-9597-1795e1c8cc55/content>
- [5] Armis Researchers. (n.d.) Blueborne vulnerability summary. Accessed: 2025-05-01. [Online]. Available: <https://westoahu.hawaii.edu/cyber/vulnerability-research/vulnerabilities-weekly-summaries/blueborne/>
- [6] Patrik Alnehagen Sandstad, “Open source cots equipment for transparent security: A qualitative study,” Master’s thesis, University of Oslo, Department of Informatics, Faculty of Mathematics and Natural Sciences, 2024, 30 ECTS. [Online]. Available: <https://www.mn.uio.no/>
- [7] Chainalysis. (2024) 2024 crypto crime mid-year update: Part 1. Chainalysis. Accessed: 2025-05-21. [Online]. Available: <https://www.chainalysis.com/blog/2024-crypto-crime-mid-year-update-part-1/>
- [8] Gunnar Alendal, *Digital Forensic Acquisition of mobile phones in the Era of Mandatory Security*. NTNU, 2022.
- [9] Patrik Alnehagen Sandstad, “Open source cots equipment for transparent security,” Master’s thesis, University of Oslo, 2024.
- [10] u-connect. u-connectxpress, at commands manual. [Online]. Available: [https://content.u-blox.com/sites/default/files/u-connectXpress-ATCommands-Manual\\_UBX-14044127.pdf?utm\\_content=UBX-14044127](https://content.u-blox.com/sites/default/files/u-connectXpress-ATCommands-Manual_UBX-14044127.pdf?utm_content=UBX-14044127)
- [11] Hacker Warehouse, “Usb armory mk ii,” 2024, accessed: 2025-04-25. [Online]. Available: <https://hackerwarehouse.com/product/usb-armory-mkii/>

- [12] Proto Bioengineering, “How to control a bluetooth le device with python,” 2019, accessed: 2025-04-25. [Online]. Available: <https://medium.com/@protobioengineering/how-to-control-a-bluetooth-le-device-with-python-3541c0cd2223>
- [13] ——. (2021) How to control a bluetooth le device with python. Medium. Accessed: 2025-03-24. [Online]. Available: <https://medium.com/@protobioengineering/how-to-control-a-bluetooth-le-device-with-python-3541c0cd2223>
- [14] Collin Mulliner. hidemulation. [Online]. Available: <https://github.com/crmulliner/hidemulation>
- [15] Gunnar Alendal, “Digital forensic acquisition of mobile phones in the era of mandatory security: Offensive techniques, security vulnerabilities and exploitation,” Doctoral thesis, Norwegian University of Science and Technology (NTNU), Department of Information Security and Communication Technology, 2022, doctoral theses at NTNU, 2022:94.
- [16] Oskar Krystian Michalski, “Digital forensic acquisition: Assessment of gunnar alendal’s work,” Paper, University of Bergen, Department of Informatics, 2024, supervisor: Øyvind Ytrehus.

# Appendices

# Appendix A

## Exciting results

### A.1 The HID emulation scripts

#### A.1.1 Modified hidnet.sh bash script.

Listing A.1: Dev\_switch\_HID.sh

```
1  #!/bin/bash
2  #
3  # Modification of Collin Mulliner's USB gadget script for the USB
   Armory Device.
4  # It automatically re-executes with sudo if not already root.
5  # Then it sets up an IP address on the new ECM interface for SSH access
   at IP 10.0.0.1.
6  #
7  # Collin Mulliner + modifications
8
9  #set +e # Exit on any error
10
11 # This will remove the old gadget drivers that already exists.
12 # Therefor if an SSH connection is open, it will be dropped.
13 # (Inspired by Collin Mulliners code.)
14 modprobe -r g_ether usb_f_ecm u_ether 2>/dev/null || true
15 modprobe usb_f_hid 2>/dev/null || true
16 modprobe usb_f_ecm 2>/dev/null || true
17
18 # Removing existing usb_gadgets
19 if [ -d /sys/kernel/config/usb_gadget/g1 ]; then
20     echo "" > /sys/kernel/config/usb_gadget/g1/UDC 2>/dev/null || true
21     rm -rf /sys/kernel/config/usb_gadget/g1
22 fi
23
24 # Instantiate new gadget by Collin Mulliner.
25 mkdir /sys/kernel/config/usb_gadget/g1
26 cd /sys/kernel/config/usb_gadget/g1
27
28 # This device description was provided by Collin Mulliner.
29 echo 0x1d6b > idVendor      # Linux Foundation
30 echo 0x0104 > idProduct     # Multifunction Composite Gadget
31 echo 0x0100 > bcdDevice     # v1.0.0
32 echo 0x0200 > bcdUSB        # USB2
33 mkdir strings/0x409
34 echo "deadbeef9876543210" > strings/0x409/serialnumber
```

```

35 echo "USBArmory" > strings/0x409/manufacturer
36 echo "USBArmory Network + Keyboard" > strings/0x409/product
37 mkdir configs/c.1
38 mkdir configs/c.1/strings/0x409
39 echo "Conf1" > configs/c.1/strings/0x409/configuration
40 echo 120 > configs/c.1/MaxPower
41
42 # Instantiating the HID function for the device (provided by Collin
    Mulliner):
43 mkdir functions/hid.usb0
44 echo 1 > functions/hid.usb0/protocol
45 echo 1 > functions/hid.usb0/subclass
46 echo 8 > functions/hid.usb0/report_length
47 echo -ne "\x05\x01\x09\x06\xA1\x01\x05\x07\x19\xE0\x29\xE7\x15\x00\x25\x
    x01\x75\x01\x95\x08\x81\x02\x95\x01\x75\x08\x81\x03\x95\x05\x75\x01
    \x05\x08\x19\x01\x29\x05\x91\x02\x95\x01\x75\x03\x91\x03\x95\x06\x
    x75\x08\x15\x00\x25\x65\x05\x07\x19\x00\x29\x65\x81\x00\xC0" \
48 > functions/hid.usb0/report_desc
49
50 # Create ECM (Ethernet) function
51 mkdir functions/ecm.usb0
52
53 # Create symbolic link to the functions in the config directory. (As
    Collin Mulliner did.)
54 ln -s functions/hid.usb0 configs/c.1/
55 ln -s functions/ecm.usb0 configs/c.1/
56
57 # Connect the gadget to the physical USB controller.
58 echo "Binding to ci_hdrc.0..."
59 echo ci_hdrc.0 > UDC
60
61 # Instantly enable the network interface for ssh access.
62 # Network interface that I was working with was named "usb0".
63 # And the 10.0.0.1/24 IP address was the default for the USB Armory.
64 USB_IF="usb0"
65 ip link set $USB_IF up
66 ip addr add 10.0.0.1/24 dev $USB_IF

```

## A.1.2 Original hidnet.sh script

Listing A.2: hidnet.sh by Collin Mulliner

```

1 #!/bin/bash
2
3 #
4 # Collin Mulliner <collin AT mulliner.org>
5 #
6
7 modprobe -r g_ether usb_f_ecm u_ether
8 modprobe usb_f_hid
9 modprobe usb_f_ecm
10
11 cd /sys/kernel/config/
12 mkdir usb_gadget/g1
13 cd usb_gadget/g1
14 mkdir configs/c.1
15 mkdir functions/hid.usb0

```

```

16 mkdir functions/ecm.usb0
17 echo 1 > functions/hid.usb0/protocol
18 echo 1 > functions/hid.usb0/subclass
19 echo 8 > functions/hid.usb0/report_length
20 echo -ne "\x05\x01\x09\x06\xA1\x01\x05\x07\x19\xE0\x29\xE7\x15\x00\x25\x
    x01\x75\x01\x95\x08\x81\x02\x95\x01\x75\x08\x81\x03\x95\x05\x75\x01
    \x05\x08\x19\x01\x29\x05\x91\x02\x95\x01\x75\x03\x91\x03\x95\x06\x
    x75\x08\x15\x00\x25\x65\x05\x07\x19\x00\x29\x65\x81\x00\xC0" >
    functions/hid.usb0/report_desc
21 mkdir strings/0x409
22 mkdir configs/c.1/strings/0x409
23 echo 0x1d6b > idVendor # Linux Foundation
24 echo 0x0104 > idProduct # Multifunction Composite Gadget
25 echo 0x0100 > bcdDevice # v1.0.0
26 echo 0x0200 > bcdUSB # USB2
27 echo "deadbeef9876543210" > strings/0x409/serialnumber
28 echo "USBArmory" > strings/0x409/manufacturer
29 echo "USBArmory Network + Keyboard" > strings/0x409/product
30 echo "Conf1" > configs/c.1/strings/0x409/configuration
31 echo 120 > configs/c.1/MaxPower
32 ln -s functions/hid.usb0 configs/c.1/
33 ln -s functions/ecm.usb0 configs/c.1/
34 echo ci_hdrc.0 > UDC

```



## A.2 Beginning of BLE communication with ANNA-B112

Listing A.3: Bleak\_SPS\_BLE.py

```
1 import asyncio
2 from bleak import BleakScanner, BleakClient
3
4 async def main():
5     print("Scanning for BLE devices...")
6     devices = await BleakScanner.discover(timeout=10.0)
7
8     for d in devices:
9         if d.name == "ANNA-B1-0CADD":
10             print("Found ANNA-B1-0CADD")
11             try:
12                 async with BleakClient(d.address) as client:
13                     """ print(f"Connected to {d.name} at {d.address}")
14                     response = await send_command(client, "ls")
15                     if response:
16                         print(f"Response: {response}") """
17             except Exception as e:
18                 print(f"Failed to connect: {e}")
19
20 asyncio.run(main())
```

## A.3 Connection Attempts.

### A.3.1 Attempting Connection after full iteration.

Listing A.4: Sub optimal connection attempts.

```
1 import asyncio
2 from bleak import BleakScanner, BleakClient
3
4 TARGET_NAME = "ANNA-B112"
5 MAX_RETRIES = 3
6
7 async def main():
8     print("Scanning for devices...")
9     devices = await BleakScanner.discover(timeout=5.0)
10
11     target_device = None # Place holder for the target device to be
12                          # found.
13     for device in devices:
14         print(f"Found device: {device.name} [{device.address}]")
15         if device.name == TARGET_NAME:
16             target_device = device # Store the found device
17             break
18
19     # If the target device was not scanned then return.
20     if target_device is None:
21         print(f"Device '{TARGET_NAME}' not found.")
22         return
23
24     print(f"Found target device: {target_device.name} [{target_device.
25         address}]")
26
27     for attempt in range(1, MAX_RETRIES + 1): # In case the connection
28         # establishment time out too early.
29         print(f"Attempt {attempt} to connect...")
30         try:
31             async with BleakClient(target_device.address) as client:
32                 connected = await client.is_connected()
33                 if connected:
34                     print(f"Connected to {TARGET_NAME} successfully!")
35                     services = await client.get_services()
36                     print("Services:")
37                     for service in services:
38                         print(service)
39                     return # Success, exit after connection
40                 else:
41                     print(f"Failed to connect on attempt {attempt}.")
42         except Exception as e:
43             print(f"Error on attempt {attempt}: {e}")
44
45         await asyncio.sleep(2) # Wait 2 seconds before retrying
46
47     print(f"Failed to connect to {TARGET_NAME} after {MAX_RETRIES}
48         attempts.")
49
50 if __name__ == "__main__":
51     asyncio.run(main())
```

### A.3.2 Attempting Connection instantly after discovering the target device.

Listing A.5: Successful BLE Connection attempt.

```
1 import asyncio
2 from bleak import BleakScanner, BleakClient
3
4 """ This is a more efficient way to connect to a BLE device by
   instantly establishing a connection once it is detected."""
5 TARGET_NAME = "ANNA-B1-0C4DDD"
6 CONNECTION_HOLD = 60
7
8 # Container for storing the targets name.
9 class DeviceHolder:
10     device = None
11
12 found_event = asyncio.Event()
13
14 def detection_callback(device, advertisement_data):
15     if device.name == TARGET_NAME:
16         print(f"Detected {device.name} at {device.address}.")
17         DeviceHolder.device = device # Save the device immediately.
18         found_event.set() # Signal that the target has been found.
19
20 async def connect_device(device):
21     try:
22         async with BleakClient(device) as client:
23             if client.is_connected:
24                 print(f"Connected to {device.name} ({device.address})")
25                 # await asyncio.sleep(CONNECTION_HOLD) # This can
26                 # potentially prolong the connection time.
27                 print(f"Disconnecting from {device.name}...")
28     except Exception as e:
29         print(f"Connection failed: {e}")
30
31 async def main():
32     scanner = BleakScanner()
33     scanner.register_detection_callback(detection_callback)
34
35     print(f"Scanning for '{TARGET_NAME}'...")
36     await scanner.start()
37     await found_event.wait() # Wait until the name of the target is
38                             # assigned.
39     await scanner.stop()
40
41     # Connect to the target name.
42     if DeviceHolder.device:
43         print(f"Found {TARGET_NAME} at {DeviceHolder.device.address}.
44             Attempting connection...")
45         await connect_device(DeviceHolder.device)
46     else:
47         print("Device not found.")
48
49 if __name__ == "__main__":
50     asyncio.run(main())
```

## A.4 Script discovering the services of the device.

Listing A.6: Iterating through services.

```
1 import asyncio
2 from bleak import BleakScanner, BleakClient
3
4 TARGET_NAME = "ANNA-B1-0CADD"
5 CONNECTION_HOLD = 60 # Adjust if you want to keep the connection
    longer
6
7 # Shared container to hold the target device reference
8 class DeviceHolder:
9     device = None
10
11 found_event = asyncio.Event()
12
13 def detection_callback(device, advertisement_data):
14     if device.name == TARGET_NAME:
15         print(f"Detected {device.name} at {device.address}.")
16         DeviceHolder.device = device # Save the device immediately.
17         found_event.set() # Signal that the target has been found.
18
19 async def connect_device(device):
20     try:
21         async with BleakClient(device) as client:
22             if client.is_connected:
23                 print(f"Connected to {device.name} ({device.address})")
24
25                 # Discover and print available services, here I needed
26                 # some help from chatGPT for formatting purposes.
27                 services = await client.get_services()
28                 print("\nDiscovered Services and Characteristics:")
29                 for service in services:
30                     print(f"\nService: {service.uuid} | {service.
31                         description}")
32                     for char in service.characteristics:
33                         print(f"    Characteristic: {char.uuid} | {char.
34                             description}")
35                         if char.properties:
36                             print(f"        Properties: {char.properties}")
37
38                 await asyncio.sleep(CONNECTION_HOLD) # Keep the
39                 # connection open.
40                 print(f"Disconnecting from {device.name}...")
41     except Exception as e:
42         print(f"Connection failed: {e}")
43
44 async def main():
45     scanner = BleakScanner()
46     scanner.register_detection_callback(detection_callback)
47
48     print(f"Scanning for '{TARGET_NAME}'...")
49     await scanner.start()
50     await found_event.wait() # Wait until the name of the target is
51     # assigned.
```

```

49     await scanner.stop()
50
51     if DeviceHolder.device:
52         print(f"\nFound {TARGET_NAME} at {DeviceHolder.device.address}.
53             Attempting connection...")
54         await connect_device(DeviceHolder.device)
55     else:
56         print("Device not found.")
57
58 if __name__ == "__main__":
59     asyncio.run(main())

```

## A.5 User terminal connection

Listing A.7: The script that prompts the user with a python terminal allowing for remote bash command execution.

```
1  #!/usr/bin/env python3
2  import sys
3  import subprocess
4  import importlib
5
6  # Auto-install dependencies if missing
7  _deps = ("bleak",)
8  for pkg in _deps:
9      try:
10         importlib.import_module(pkg)
11     except ImportError:
12         subprocess.check_call([sys.executable, "-m", "pip", "install",
13                                "--upgrade", pkg])
14
15 import asyncio
16 from bleak import BleakScanner, BleakClient
17
18 TARGET_NAME = "ANNA-B1-0C4DDD"
19 CONNECTION_HOLD = 5 # Time (in seconds) to keep the connection open
20 buffer = "" # Buffer to store the response from the device.
21 started = False # Flag to indicate if the response has started.
22
23 # Custom service/characteristic UUIDs as discovered:
24 SPS_SERVICE_UUID = "2456e1b9-26e2-8f83-e744-f34f01e9d701"
25 SPS_WRITE_CHAR_UUID = (
26     "2456e1b9-26e2-8f83-e744-f34f01e9d703" # Supports write and notify
27 )
28 SPS_NOTIFY_CHAR_UUID = (
29     "2456e1b9-26e2-8f83-e744-f34f01e9d703" # Same characteristic used
30     for notifications
31 )
32
33 class DeviceHolder:
34     device = None
35
36 found_event = asyncio.Event()
37
38
39 def detection_callback(device, _):
40     if device.name == TARGET_NAME:
41         print(f"Detected {device.name} at {device.address}.")
42         DeviceHolder.device = device
43         found_event.set()
44
45
46 def terminal_emulation(exit):
47     command = input("Type your input: ")
48     if command.lower() == "exit":
49         return True, None
50     else:
51         return (
```

```

52         exit ,
53         b"##" + command.encode("utf-8") + b"\n",
54     )
55
56
57 def notification_handler(_, data):
58     global buffer
59     global started
60
61     try:
62         decoded = data.decode("utf-8", errors="replace")
63     except Exception:
64         decoded = str(data)
65
66     if "!start!" in decoded and "!end!" in decoded:
67         start_index = decoded.find("!start!") + len("!start!")
68         end_index = decoded.find("!end!")
69         content = decoded[start_index:end_index].strip()
70         print("-----START-----")
71         print(content)
72         print("-----END-----\n")
73         return
74
75     if "!start!" in decoded:
76         start_index = decoded.find("!start!") + len("!start!")
77         print("-----START-----")
78         started = True
79         buffer = decoded[start_index:]
80         return
81
82     if started:
83         buffer += decoded
84
85         if "!end!" in decoded:
86             end_index = buffer.find("!end!")
87             final_content = buffer[:end_index].strip()
88             print(final_content)
89             print("-----END\n")
90
91             buffer = ""
92             started = False
93
94     async def connect_device(device):
95         try:
96             async with BleakClient(device) as client:
97                 if client.is_connected:
98                     print(f"Connected to {device.name} ({device.address})")
99                     exit = False
100
101                     await client.start_notify(SPS_NOTIFY_CHAR_UUID,
102                                             notification_handler)
103                     print("Establishing Communication...")
104                     await asyncio.sleep(1)
105                     await client.write_gatt_char(
106                         SPS_WRITE_CHAR_UUID, b"+++", response=True
107                     )
108                     print(".....")

```

```

108         await asyncio.sleep(1)
109
110         print("Entering remote command execution...\n")
111
112         while True:
113             exit, command = terminal_emulation(exit)
114             if exit:
115                 break
116             else:
117                 await client.write_gatt_char(
118                     SPS_WRITE_CHAR_UUID, command, response=True
119                 )
120                 print(".....")
121                 await asyncio.sleep(1)
122
123         print("Connection was finished...")
124         await client.stop_notify(SPS_NOTIFY_CHAR_UUID)
125         print(f"Disconnecting from {device.name}...")
126
127     except Exception as e:
128         print(f"Connection failed: {e}")
129
130
131 async def main():
132     scanner = BleakScanner()
133     scanner.register_detection_callback(detection_callback)
134
135     print(f"Scanning for '{TARGET_NAME}'...")
136     await scanner.start()
137     await found_event.wait()
138     await scanner.stop()
139
140     if DeviceHolder.device:
141         print(
142             f"Found {TARGET_NAME} at {DeviceHolder.device.address}.
143             Attempting connection..."
144         )
145         await connect_device(DeviceHolder.device)
146     else:
147         print("Device not found.")
148
149 if __name__ == "__main__":
150     asyncio.run(main())

```



## A.6 Testing connection with multiple AT commands.

Listing A.8: In case one or more of the commands were not delivered.

```
1 import asyncio
2 from bleak import BleakScanner, BleakClient
3
4 TARGET_NAME = "ANNA-B1-0CADD"
5 CONNECTION_HOLD = 30 # Time (in seconds) to keep the connection open
6
7 # Custom service/characteristic UUIDs as discovered:
8 SPS_SERVICE_UUID = "2456e1b9-26e2-8f83-e744-f34f01e9d701"
9 SPS_WRITE_CHAR_UUID = "2456e1b9-26e2-8f83-e744-f34f01e9d703" #
10 SPS_NOTIFY_CHAR_UUID = "2456e1b9-26e2-8f83-e744-f34f01e9d703" # Same
    Supports write and notify
    characteristic used for notifications
11
12 # Shared container to hold the target device reference.
13 class DeviceHolder:
14     device = None
15
16 found_event = asyncio.Event()
17
18 def detection_callback(device, advertisement_data):
19     if device.name == TARGET_NAME:
20         print(f"Detected {device.name} at {device.address}.")
21         DeviceHolder.device = device # Save the device immediately.
22         found_event.set() # Signal that the target has been found.
23
24 # Chat GPTs suggestion: Use a notification handler to process incoming
    notifications.
25 def notification_handler(sender, data):
26     try:
27         decoded = data.decode("utf-8", errors="replace")
28     except Exception as e:
29         decoded = str(data)
30     print(f"Notification from {sender}: {decoded}")
31
32 async def connect_device(device):
33     try:
34         async with BleakClient(device) as client:
35             if client.is_connected:
36                 print(f"Connected to {device.name} ({device.address})")
37
38                 # Start notifications to receive the device's response.
39                 await client.start_notify(SPS_NOTIFY_CHAR_UUID,
                    notification_handler)
40
41                 # Allow a short delay to ensure notifications are
                    properly set up.
42                 await asyncio.sleep(1)
43
44                 # Sending multiple AT commands in case of packet loss
                    or timing issues.
45                 print("Sending AT command...")
46                 await client.write_gatt_char(SPS_WRITE_CHAR_UUID, b"AT\r\n", response=True)
47                 await client.write_gatt_char(SPS_WRITE_CHAR_UUID, b"AT\r\n")
```

```

        r\n", response=True)
48     await client.write_gatt_char(SPS_WRITE_CHAR_UUID, b"AT\r\n", response=True)
49     await client.write_gatt_char(SPS_WRITE_CHAR_UUID, b"AT\r\n", response=True)
50     await client.write_gatt_char(SPS_WRITE_CHAR_UUID, b"AT\r\n", response=True)
51     await client.write_gatt_char(SPS_WRITE_CHAR_UUID, b"AT\r\n", response=True)
52     print("AT command sent. Waiting for response...")
53
54     # Keep the connection open for a period to receive any
        responses.
55     await asyncio.sleep(CONNECTION_HOLD)
56
57     # Stop notifications before disconnecting.
58     await client.stop_notify(SPS_NOTIFY_CHAR_UUID)
59     print(f"Disconnecting from {device.name}...")
60 except Exception as e:
61     print(f"Connection failed: {e}")
62
63 async def main():
64     scanner = BleakScanner()
65     scanner.register_detection_callback(detection_callback)
66
67     print(f"Scanning for '{TARGET_NAME}'...")
68     await scanner.start()
69     await found_event.wait() # Wait until the callback detects the
        target.
70     await scanner.stop()
71
72     if DeviceHolder.device:
73         print(f"Found {TARGET_NAME} at {DeviceHolder.device.address}.
            Attempting connection...")
74         await connect_device(DeviceHolder.device)
75     else:
76         print("Device not found.")
77
78 if __name__ == "__main__":
79     asyncio.run(main())

```

## A.7 listening.py

Listing A.9: the script reading input from ttymxc0 and executing recognized commands.

```

1  #!/usr/bin/env python3
2
3  import serial
4  import subprocess
5  import os
6  from time import sleep
7
8  def main():
9     ker_start = b'!start!'
10    ker_end = b'!end!'

```

```

11     try:
12         with serial.Serial('/dev/ttymx0', baudrate=115200, timeout=1)
13             as ser:
14                 print("Listening on /dev/ttymx0...")
15
16                 while True:
17                     line = ser.readline()
18                     if line:
19                         try:
20                             text = data.decode('utf-8', errors='replace')
21                             text = str(text)
22                         except Exception as e:
23                             text = str(line)
24
25                     print("Detected input:", text)
26                     fixed_text = text.strip()
27
28                     if fixed_text[2] == "#":
29                         user_command = fixed_text[3:-3]
30                         print(f"User input: {user_command}")
31                         print("Executing...")
32
33                         try:
34                             print("The detected command is:",
35                                   user_command.strip()[3:])
36                             # Handle cd command separately
37                             if user_command.startswith('cd '):
38                                 try:
39                                     directory = user_command[3:].strip()
40                                     ()
41                                     os.chdir(directory)
42                                     response = f"Changed directory to:
43                                         {os.getcwd()}"
44                                     ker_resp = response.encode()
45                                     ser.write(ker_start)
46                                     ser.write(ker_resp)
47                                     ser.write(ker_end)
48                                 except Exception as e:
49                                     error = str(e).encode()
50                                     ser.write(ker_start)
51                                     ser.write(error)
52                                     ser.write(ker_end)
53                             else:
54                                 # Handle other commands as before
55                                 execution = subprocess.run(user_command
56                                                            , shell=True, capture_output=True)
57                                 if execution.stdout:
58                                     ker_resp = execution.stdout
59                                     print("Command Output:\n", ker_resp
60                                           )
61                                     ser.write(ker_start)
62                                     ser.write(ker_resp)
63                                     ser.write(ker_end)
64                                     sleep(1)
65                                     ser.reset_input_buffer()
66
67                                 if execution.stderr:
68                                     error = execution.stderr

```

```

63         print("Error Output:")
64         print(error)
65         ser.write(ker_start)
66         ser.write(error)
67         ser.write(ker_end)
68         sleep(1)
69
70     except Exception as cmd_exception:
71         print("Error executing command:",
              cmd_exception)
72         ser.write(ker_start)
73         ser.write(b'Critical Error')
74
75     except KeyboardInterrupt:
76         print("\nStopped by user.")
77     except Exception as e:
78         print("Error:", e)
79
80 if __name__ == '__main__':
81     main()

```

## A.8 Services

### A.8.1 usb-gadget-hid-ecm.service

Listing A.10: The service that runs the hidnet.sh file at boot up.

```

1 # /etc/systemd/system/usb-gadget-hid-ecm.service
2 [Unit]
3 Description=USB Gadget HID+ECM configuration
4 # ensure configs is mounted before this runs
5 After=local-fs.target sysinit.target
6 Wants=local-fs.target
7
8 [Service]
9 Type=oneshot
10 ExecStart=/usr/local/bin/usb_gadget_hid_ecm.sh
11 RemainAfterExit=yes
12
13 [Install]
14 WantedBy=multi-user.target

```

### A.8.2 rem-com-exec.service

Listing A.11: The service that runs the listening.py application at boot up.

```

1 # /etc/systemd/system/rem-com-exec.service
2 [Unit]
3 Description=Run rem_com_exec listener after USB gadget is up
4 After=usb-gadget-hid-ecm.service
5 Wants=usb-gadget-hid-ecm.service
6
7 [Service]
8 Type=simple
9 # run your Python script

```

```
10 ExecStart=/opt/rem_com_exec
11 Restart=on-failure
12 RestartSec=5
13
14 [Install]
15 WantedBy=multi-user.target
```