Andreas Karner, BSc

Analysis of Low-level iOS Lightning Protocols

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisors

Florian Draschbacher, Dipl.-Ing. BSc

Stefan Mangard, Univ.-Prof. Dipl.-Ing. Dr.techn.

Institute of Applied Information Processing and Communications

Graz, September 2023

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date, Signature

Acknowledgements

This master thesis was written in 2023 in cooperation with the Institute of Applied Information Processing and Communications.

First and foremost, I want to thank my supervisors, Florian Draschbacher and Stefan Mangard, for their excellent support throughout this master thesis. Their constant feedback has contributed significantly to the realization of this work, and I am grateful that they always took time out of their busy schedule to support my research.

I also want to thank my family, especially my parents, for their backing, education, and support. Finally, I want to thank my partner Beate for her constant motivation and understanding throughout my studies. Without her, I would certainly not be at this point in my life.

Abstract

The importance of mobile devices has tremendously increased over the past few years. This rise goes hand in hand with a broader field of application and an increased amount of sensitive data stored on these devices, making data protection essential. Therefore, complex security systems have emerged and been incorporated into the smartphone operating system to avoid data leakages.

However, Juice Jacking attacks are often overlooked, which aim to break this confidentiality by stealing sensitive data from the device through a tethered connection. An example of such a tethered connection is for charging the device battery. To illustrate the impact of Juice Jacking attacks, this thesis proposes a state-of-the-art modified charging cable called TuWire, which can stealthily inject HID events and capture the screen of an iPhone alongside charging after establishing a tethered connection. To access the system interfaces that enable the previously mentioned attack vectors, TuWire must first complete a proprietary IDBUS handshake. This handshake informs the iPhone about a newly connected accessory and allows TuWire to acquire access to the USB interface. Second, TuWire authenticates itself as a trustworthy accessory to the iPhone, leveraging the proprietary iAP protocol. As a trustworthy accessory, TuWire can put the iPhone into USB host mode via the iAP protocol. Third, the malicious charging cable performs various handshakes on iAP and USB levels to gain access to the iPhone's HID, WiFi, and screen-mirroring interfaces. Finally, TuWire can infiltrate the iPhone by i) enrolling a custom Mobile Device Management (MDM) profile, ii) extracting the access credentials for the WiFi network the iPhone is currently connected to, and iii) streaming the current screen content to flash memory.

Additionally, the thesis proposes a novel protection solution, called Lightning Condom, which resembles the idea of a USB Condom. The Lightning Condom secures the charging process by filtering the IDBUS handshake messages and preventing the iPhone from exposing high-level protocols like the USB interface, eliminating all kinds of Juice Jacking attacks.

Keywords: Smartphone security · Charging Attacks · Juice-Jacking/Filming

Kurzfassung

Smartphones haben in den letzten Jahren immer mehr an Bedeutung gewonnen. Diese Entwicklung impliziert einen signifikant größeren Anwendungsbereich, sowie auch eine größere Anzahl an sensiblen, gespeicherten Daten auf den Geräten. Infolgedessen ist Datenschutz unerlässlich geworden und Sicherheitsmaßnahmen wurden fest in das Betriebssystem des Smartphones verankert.

Eine bisher oft unterschätzte Möglichkeit, diese Sicherheitsmaßnahmen zu umgehen und sensible Daten vom Smartphone abzugreifen, stellt Juice Jacking dar. Diese Angriffsklasse infiltriert das Smartphone via einer physischen Verbindung, welche oftmals für das Laden der Gerätebatterie notwendig ist. Zur Verdeutlichung der Relevanz von Juice Jacking Angriffen, wurde in dieser Masterarbeit ein modifiziertes Ladekabel namens TuWire entwickelt. Nach Herstellung einer physischen Verbindung ist TuWire in der Lage, HID-Events einzuschleusen und den Bildschirminhalt abzugreifen. Hierfür verwendet TuWire zuerst das proprietäre IDBUS-Protokoll, um sich als ein legitimes Ladekabel gegenüber dem iPhone auszugeben und Zugriff zur USB-Schnittstelle anzufordern. Im nächsten Schritt wird das iPhone durch das proprietäre iAP-Protokoll in den USB-Host Zustand versetzt. Um Zugriff zu dieser Option zu erhalten, muss zuerst ein Authentifizierungsprozess, Bestandteil des iAP-Protokolles, erfolgreich abgeschlossen werden. Nach weiteren anwendungsspezifischen Handshakes, ausgeführt auf der iAP und USB-Ebene, ist TuWire in der Lage, das iPhone wie folgt zu infiltrieren: i) Registrierung eines Mobile-Device-Management (MDM) Profils aus dem Internet, ii) das WiFi-Passwort vom aktuell verbundenen Access Point zu extrahieren und iii) streamen des Bildschirminhaltes in den Flash-Speicher.

Um der Gefahr von Juice Jacking Attacken, wie TuWire, entgegenzuwirken, inkludiert diese Arbeit auch die Implementierung eines Lightning Condom, welches in der Lage ist, diese abzuwehren. Dafür wird der IDBUS Handshake gefiltert, um die Freigabe von high-level Protokollen, wie USB, zu verhindern und somit wird ein sicherer Ladevorgang garantiert.

Schlagwörter: Smartphone Sicherheit · Ladestation Attacke · Juice-Jacking/Filming

Contents

List of Figures

List of Tables

Chapter 1.

Introduction

The importance of smartphones has increased tremendously over the last decade[1]. Due to the fast technological advancements, the scope of smartphones has expanded from their original telecommunication purpose to include functions such as personal bank account management and use for multifactor user authentication. These expanded capabilities rely on additional stored sensitive user data on the device, which enables the device to perform the previously mentioned actions on the user's behalf. Consequently, if an attacker gains access to this sensitive user data, it could potentially impersonate the legitimate user.

Therefore, data protection is essential and established in multiple hardware and software layers. For example, on the hardware level, recent smartphones encrypt data via dedicated secure enclaves, which only provide an API for data encryption and decryption and never reveal their private keys. In addition, smartphones leverage biometric authentication mechanisms, such as fingerprint scanners and face IDs, to prove the user's unique biologic characteristics before granting access. In contrast to the hardware, the operating system aims to enhance security on the software level by providing secure API functions to the applications, enabling features such as establishing a secure transport channel with remote resources.

Since the introduction of multipurpose connectors, such as USB-C and Lightning, a single connector allows charging the device battery via a universal USB cable and connecting accessories. An accessory may extend the smartphone's usability by connecting external monitors, input devices such as a keyboard and a mouse, network interfaces, or entire docking stations. In order to integrate their functionalities smoothly into the rest of the smartphone's operating system, deep system integration is required. Unfortunately, this deep system integration introduces new security concerns, such as the injection of invisible functionalities. An accessory can accomplish this by providing its functionalities via a multipurpose protocol like Universal Serial Bus (USB). USB allows exposing multiple components concurrently so that the accessory can inject malicious and invisible features alongside the desired accessory behavior. The same problem applies to tethered charging because the employed charging cable usually provides additional functions beyond power. We distinguish between two major cable types: i) charging-only, which only conveys power, and ii) data cables, which convey power and provide access to a device data interface like USB. Therefore, charging the phone via a data cable implies that the charging outlet gains access to the device data interface.

An attacker can exploit this concept by arming a public charging kiosk with manipulated data cables. Once a victim establishes a connection for charging, the attacker gains access to the device data interfaces and may try to infiltrate the device. Depending on the device being infiltrated, the attacker may attempt to steal or manipulate sensitive data or install applications via the developer interface. The academic research community denotes this attack class as Juice Jacking, encompassing different subclasses with distinct attack vectors.

[1] https://web.archive.org/web/20230518005608/https://www.pewresearch.org/internet/fact-sheet/mobile/

Initially, Juice Jacking [Kum20] attacks infiltrated the smartphone as a USB host device. Running as the USB host forced the phone to operate in USB peripheral mode and to expose its USB functionalities, such as access to the internal device storage via the USB Mass Storage Device interface and debug interface. In the early days, the smartphone operating system exposed these functionalities to the host without user confirmation. However, this has changed with the release of iOS 7[2] and Android 6[3]. Since then, the user must explicitly trust the USB host computer before the mobile operating system exposes its functionalities, including the previously mentioned examples. To circumvent this mitigation strategy, a novel category of Juice Jacking attacks emerged, leveraging the smartphone in USB host mode instead of USB peripheral mode. One of them is Juice Filming [MLMK15], which captures the entire screen content during the charging phase by emulating an HDMI adapter. Later, in a postprocessing step, the attacker may automatically extract valuable information from this video feed via optical character recognition (OCR) algorithm. HID Event injection attacks also belong to the class, which leverages the victim device in USB host mode. Compared to Juice Filming, these attacks emulate input devices, such as a keyboard or a mouse, and gain complete user interface control over the victim's device. Combining the HID Event injection and Juice Filming attacks, an attacker can control the victim's device like a regular user.

1.1. Problem Statement

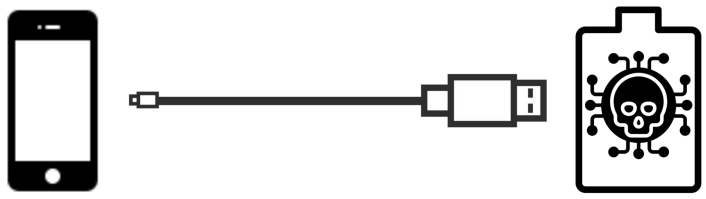
All mobile devices have one unique property in common; they are battery powered. Therefore, recharging the battery after a particular time is necessary, which is still primarily accomplished through a tethered connection, i.e., a cable. Public charging kiosks are deployed in airports, bus stations, and shopping centers to serve the high charging demand and keep the device battery charged on the go. However, these kiosks are prone to the previously introduced Juice Jacking attacks because a malicious attacker can easily equip these kiosks with tampered data cables, allowing them to attack the device.

Various security researchers have already addressed Juice Jacking attacks in the past and proposed the following mitigation strategies: i) USB Condoms [Kum20], a USB-to-USB adapter with removed data lines, ii) SandUSB [LHK+16], a Juice Jacking detection tool, iii) powering off the device during the charging phase [Kum20], and iv) using the own charger [Kum20].

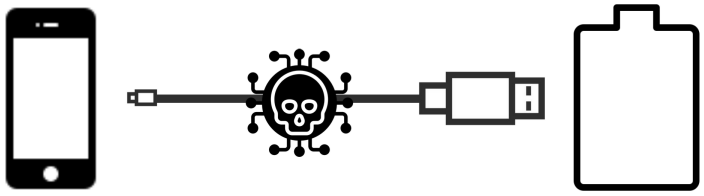
Powering off the device during charging defeats Juice Jacking attacks very effectively. However, it comes with the undesired cost that the device is unusable during the charging process. Another effortless way to erase the risk of falling victim to a Juice Jacking attack is using a personal charger. Unfortunately, this approach requires public power outlets, which are only occasionally available. Instead, charging kiosks offer native smartphone connectors like USB-C, micro-USB, or Lightning for charging the smartphone. Furthermore, mitigation strategies, such as the USB Condom and SandUSB, assume a USB based Juice Jacking attack scenario composed of a dumb cable and a malicious USB outlet, as illustrated in Figure 1.1a. However, the latest trend has shown that smartphone manufacturers prefer using an innovative, single multipurpose connector, which simultaneously provides access to different protocols. These connectors facilitate new attack scenarios, such as GhostTalk [WGY22],

[2] https://web.archive.org/web/20221222152315/https://theta44.org/2014/07/28/ios-lockdown-diag-services.html

[3] https://web.archive.org/web/20230408024942/https://developer.android.com/about/versions/marshmallow/android-6.0-changes



(a) Dumb cable and malicious outlet



(b) Malicious cable and dumb outlet

Figure 1.1.: The difference between a malicious charging outlet and a malicious charging cable.

which leveraged the audio channels, part of the Lightning and USB-C device connectors, to extract sensitive data. Furthermore, the Lightning connector of Apple mobile devices provides access to a proprietary protocol allowing the dispatch of various internal high-level protocols to the connector pins. As a result, mitigation objects that guarantee security by protecting the USB interface at the outlet side, such as classic USB Condoms and SandUSB, cannot mitigate Juice Jacking attacks where the attack logic is incorporated into the charging cable, as shown in Figure 1.1b.

1.2. Thesis Contribution

This thesis contributes a state-of-the-art Juice Jacking attack against an Apple iPhone called TuWire and a novel countermeasure against Juice Jacking attacks, such as TuWire, called Lightning Condom. To assemble these two main contributions, further reverse engineering and implementing three Apple proprietary protocols, realized as separate libraries, from scratch was necessary.

TuWire

The implemented attack can bypass all USB outlet mitigation strategies by moving the attack logic from the outlet into the charging cable. Doing so lets us control the device's user interface, extract the access credentials for the WiFi network the iPhone is currently connected to, enroll an external Mobile Device Management (MDM) profile, and stream the device screen content to a storage device. To access the required interfaces for accomplishing the previously defined goals, TuWire must communicate with the iPhone via three Apple proprietary protocols.

Lightning Condom

The Lightning Condom ensures a secure charging process through untrusted public charging kiosks by mitigating Juice Jacking attacks directed at Lightning based Apple devices. Therefore, it mitigates attacks such as TuWire and delivers a secure charging interface to users. It accomplishes this by transforming the present Lightning power source into a charging-only

cable, regardless of whether it is a power supply accessory or an ordinary cable. Hence, the Lightning Condom ensures that no high-level protocols, such as USB, are exposed to the Lightning accessory.

1.3. Thesis Structure

Outline. The remaining structure of the thesis is as follows. Chapter 2 introduces various commonly used protocols and standards, which are required to accomplish the defined goals. Chapter 3 provides a detailed list of related works and exhibits the significant differences in contrast to this work. Information about the Apple internal protocols required for the attack and mitigation phase is provided in Chapter 4, followed by the implementation in Chapter 5. The mitigation, the Lightning Condom, is explained in detail in Chapter 6. The evaluation results are shown in Chapter 7, which also discusses the limitations and future work. Chapter 8 finishes the thesis with a conclusion.

Chapter 2.

Background

This chapter provides essential information about all the open standards and protocols used throughout the master thesis. First, all the relevant communication protocols are introduced in Section 2.1. Second, Section 2.2 explains the crucial parts of the H.264 video standard, and Section 2.3 provides a rough overview of the iOS security model.

2.1. Communication Protocols

This section provides essential information about all the open standards and communication protocols used throughout the master thesis.

2.1.1. Universal Asynchronous Receiver Transmitter

The Universal Asynchronous Receiver-Transmitter (UART) protocol implements a device-to-device architecture and is commonly used by microcontrollers and embedded systems for debugging, short-range communications, and firmware updates. Furthermore, the data is exchanged asynchronously, implying that the protocol does not require a dedicated clock signal, and two lines are sufficient for connecting two communication partners. These wires are called RX and TX for receiver and sender, respectively. The actual data, the raw information we would like to transmit, is wrapped by a UART frame composed of

- a start bit responsible for synchronizing the receiver,
- a data frame holding the actual data,
- an optional parity bit for error checking, and
- a stop bit indicating the end of the frame.

The sender transfers the UART frame serially, one bit at a time, to the receiver. To ensure proper interpretation of the data stream, both communication partners prior agree on the same so-called BAUD rate, which denotes the transfer speed in bits per second. Commonly used BAUD rate values are 9600 or 115200, but newer UART components support rates up to 1500000.

2.1.2. Inter Integrated Circuit

The Inter-Integrated Circuit (I2C) standard implements a serial bus architecture capable of connecting multiple masters and slaves simultaneously. However, in the case of a multi-master architecture, synchronization is mandatory to ensure that only a single master interacts with the bus at a time. Due to its simplicity, I2C is widely deployed in the microcontroller field to

interconnect sensors and displays. The bus only requires two lines to operate; one for data (SDA) and one for the clock signal (SCL). Due to the single data line, I2C only supports half-duplex communications between the master and the slave. Each connected slave on the bus possesses a unique address to which it listens. Data transfers are always initiated by the master and arranged in I2C messages, consisting of a single address frame followed by one or multiple data frames. The address frame holds the address of the slave, and the data frame is the actual value to transfer. The bus supports bidirectional and unidirectional communication with bit rates from 100 kbit/s up to 3.4 Mbit/s and 5 Mbit/s, respectively [Sem12].

2.1.3. Serial Peripheral Interface

The Serial Peripheral Interface (SPI) describes a 4-wire serial bus architecture with a single master and multiple slaves. It is widely deployed in the industry to establish short-range connections between a microcontroller and sensors, displays, or flash memories. The host is responsible for the synchronization of the bus via a dedicated clock line (SCLK) and can exchange data in a full-duplex mode due to a separate read (MISO - Master-In-Slave-Out) and write (MOSI - Master-Out-Slave-In) line. Before a host can communicate with a slave, they must activate a dedicated chip-select (CS) line between him and the slave. This implies that the host must allocate as many pins as slaves connected for proper activation. To overcome this scalability issue, slaves can also be daisy-chained. In this constellation, the slaves are connected in series and shift data from the input to the output until it has reached the correct slave. However, only some devices support this mode, requiring more clock cycles than addressing them directly via the chip-select line. Furthermore, the utilized hardware limits the data rates. Microcontrollers such as the stm32f723e microcontroller support a data rate of up to 54 Mbit/sec [STM22].

2.1.4. Quad Serial Peripheral Interface

The Quad Serial Peripheral Interface (QSPI) extends the previously explained SPI interface by utilizing four data lines instead of a single one, and it is primarily used to interact with onboard flash memories. Furthermore, by enabling the Dual Data Rate (DDR) mode, the data rate can be doubled by leveraging every edge of the clock signal. Besides the data rate, QSPI supports natively addressing a memory space of up to 4 GB via a 32-bit addressing mode and a higher clock frequency than SPI devices. For example, the stm32f723e microcontroller supports a QSPI data rate of up to 100 Mbit/sec [STM22].

2.1.5. Universal Serial Bus

Nowadays, the Universal Serial Bus (USB) is the de facto standard for establishing a connection between a host computer and peripherals, such as Mass Storage Devices, printers, and Human Interface Devices (HID). The first revision of the standard (USB 1.0 [USB96]) was released in 1996 and came with two different modes of operation: i) a "Low Speed" (LS) mode, with a maximum transfer rate of 1.5Mbits/s and ii) a "Full Speed" (FS) mode, with a maximum transfer rate of 12 Mbit/s. Due to emerging of high-capacity storage devices, USB 2.0 [USB00] was released in 2000 and offered a maximum data rate of 480 Mbit/s (53 MB/s), called "High Speed" (HS). Since then, newer versions such as USB 3.0 and USB-3.1 have been released, but this thesis will only focus on USB 2.0 HS mode.

The bus is organized within a star topology and controlled by the host. This implies that the host device initiates the entire traffic on the bus, and no interconnection between the

individual peripherals exists. A connection between a peripheral and a host is established via a USB Connector, which embeds at least four pins. The four mandatory pins to establish a USB connection are GND and VBUS, responsible for powering up the peripheral, and D+ and D-, transferring the data differentially. A wide variety of connectors has emerged to integrate them properly into the different form sizes of the devices. However, this thesis will only mention Type A and Micro-USB because these are still the dominant connectors for central-purpose computers and microcontrollers, respectively.

USB is Plug'n'Play ready, which means that a user can connect a new peripheral via USB, and the host device can detect and load the proper driver automatically. This process is called enumeration and loads the USB descriptor hierarchy in multiple iterations, shown in Figure 2.1. After detecting a new USB peripheral, the host first requests the so-called Device Descriptor, which includes high-level information about the freshly connected device, such as vendor-id, product-id, and the USB revision. Second, it requests the Configuration Descriptor, which holds power consumption-related information. A USB peripheral can offer the host different power configurations, such as a lower power and high power mode, to support him in optimizing his power consumption. However, the host can only activate one at a time. In the next step, the host fetches all Interface Descriptors and their associated Endpoint Descriptors from the USB peripheral. The Interface Descriptor with the fields i) bInterfaceClass, ii) bInterfaceSubClass, and iii) bInterfaceProtocol guides the host to load the correct driver. The Endpoint Descriptors provide information about USB peripheral side buffers, to which the host can read and write. The host driver uses these endpoints or buffers to interact with the USB peripherals, which can be unidirectional or bidirectional. Unidirectional endpoints are either used to transfer data to the host device, denoted as an IN-endpoint in the USB standard, or away from the host device, denoted as an OUT-endpoint. A bidirectional endpoint can handle IN and OUT data simultaneously and is primarily used during initialization. The USB standard defined the following types of endpoints:

- Control Endpoints: primarily used during the initiation and enumeration process
- Interrupt Endpoints: used for non-periodic and small events
- Isochronous Endpoints: are used for continuous and periodic data
- Bulk Endpoints: used for extensive burst data

To support multipurpose USB peripherals, such as a mouse and keyboard combination, each Configuration Descriptor can consist of multiple Interface Descriptors. Depending on the USB interface, one or more endpoints are needed per interface to ensure proper functionality.

USB On-The-Go

The classical USB standard does not allow a USB-compatible device to switch between a device and host role. This inflexibility was acceptable in the early days of USB, but the need to connect peripherals directly to smartphones grew. Changing roles would allow to i) connect a smartphone as usual to a host and manage it from there and ii) connect a different peripheral, such as a USB keyboard or Mass Storage Device, to the smartphone and leverage their benefits. In 2012, the USB-IF released a USB 2.0 extension called USB On-The-Go (OTG) [USB12], which allows a USB peripheral to switch roles. After establishing a connection between a USB OTG device and a peripheral, a host is determined by probing an additional connector pin. After this additional process, the elected host will start to enumerate the USB peripheral by loading the relevant descriptors shown in Figure 2.1.

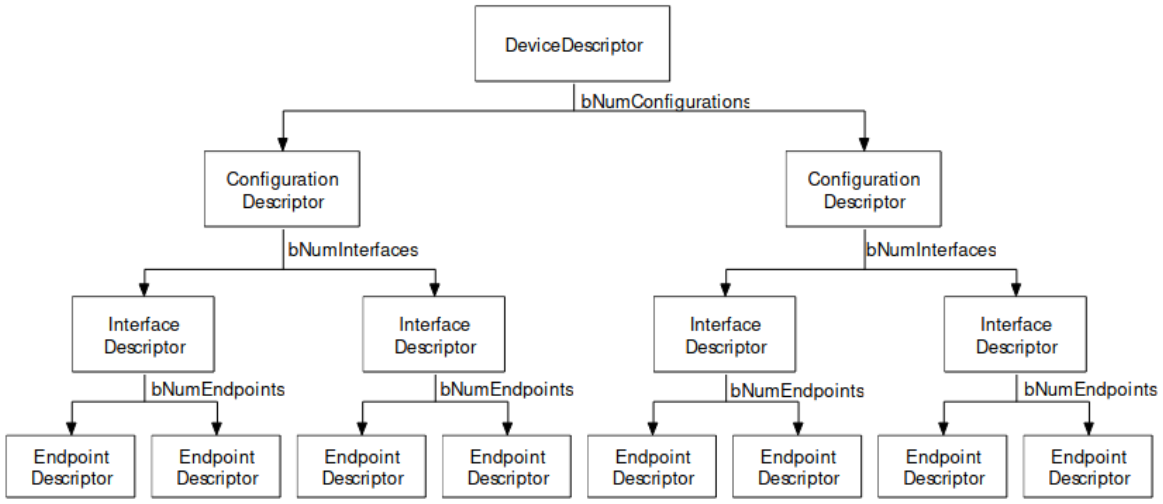


Figure 2.1.: Loaded USB descriptors and their hierarchical relationship to each other of the enumeration process.

USB Human Interface Device

The USB Human Interface Devices (HID) [USB01] extension, released in 2001, standardizes how a user can leverage input devices such as mice and keyboards to control a host computer. An HID device leverages the USB standard as the communication layer and updates the host device about its internal device state by sending so-called HID reports. Each change of state, caused by pressing or releasing a key on a keyboard, for instance, triggers a new report generation and transfer to the host device. In addition, the HID device must advertise the report structure to the host device during the USB enumeration process to keep the report size small and adaptable. This structure, called Report-Descriptor, provides sufficient information to the host device for correctly parsing the afterward received reports.

USB Mass Storage Device

The USB Mass Storage Device (MSC) [USB99] standard, released in 1999, allows interaction with Mass Storage Devices through the USB interface. In this sense, MSC abstracts the underlying storage technology, such as flash drives, solid-state drives, or smartphones. Furthermore, MSC is file system independent and therefore allows the use of the storage device with any format supported by the host device.

2.2. Advanced Video Coding H.264

The H.264, also called Advanced Video Coding (AVC) [1621], is nowadays de facto the standard for streaming video content over the network. Its development purpose was to enhance the video compression ratio and to make the video frames network friendly. The standard covers many concepts, but only the Network Abstraction Layer (NAL) container relevant for this work with its subclasses Sequence Parameter Set (SPS), Picture Parameter Set (PPS), and Bitstream is explained. A NAL unit is the primary transfer container of the H.264 standard and is used to either transfer video frames or parameter sets such as the SPS and PPS. The first byte of each NAL unit contains its type, which allows the decoder

to interpret the received data correctly. The SPS contains parameters that affect the entire video sequence and remain unchanged until a newer SPS overwrites them. Such parameters are frame rate and video resolution. Conversely, the PPS contains the parameters necessary to decompress the video frames successfully. Such parameters are picture size and color space. Also, these parameters remain unchanged until a newer PPS overwrites them. Bitstreams are a series of consecutive send NAL units containing video frames, which contain flagged with a static `[0x00, 0x00, 0x00, 0x01]` preamble to enable performant parsing on the receiver side.

2.3. iOS Platform Security

Apple's mobile operating system, called iOS, powers around one-third of the worldwide used mobile devices[1]. This substantial share results from the fact that users consider Apple devices to be secure. This section aims to prove this perception by discussing a subset of the essential building blocks, focusing on the system, data, app, and accessory security.

Apple ensures system security by utilizing two fully featured system-on-chip (SoC) instances with well-defined responsibilities and a secure boot process. Figure 2.2 shows the interconnection of the two SoCs, which are known as the Application Processor (AP) and the Secure Enclave (SE). The Application Process operates the main operating system (iOS) and its applications (user apps) and consults the Secure Enclave to encrypt and decrypt user-specific data. In order for the SE to perform these operations, the SoC possesses two hardware keys, unique ID (UID) and group ID (GID), which Apple fuses into its memory at manufacturing time. Moreover, these hardware keys act as the root to derive software keys for file-based encryption, as explained later in this section. Besides the fused root keys, the Secure Enclave also possesses other components, such as a Secure Nonvolatile Storage device holding the user data keys, accessible via I2C, and a true random number generator (TRNG), to name two. Furthermore, Apple guarantees system security by enforcing a secure boot process consisting of multiple components (Boot ROM, LLB, iBoot, Kernel, iOS) executed one after the other. Each chain component validates the following component's code signature, ensuring that only Apple-signed code is loaded. The security of this chain relies on the Boot ROM, which embeds the Apple Root certificate authority's (CA) public key for code signature validation. The Boot ROM is explicitly trusted because it is immutable (read-only memory) and written during manufacturing time. On a signature mismatch, the boot process fails, and the Apple device goes into recovery mode [Inc22].

Another building block of Apple's iOS security guidance is data security, established via file-based data protection. Their design relies on a dedicated AES-256 crypto engine positioned between the memory and the processors (AP and SE). If the AP interacts with the memory, this engine receives the file keys from the SE and performs real-time encryption and decryption of the file content. Hence, this architecture avoids revealing file keys to the AP and keeps the keys secure even if the operating system is compromised. Furthermore, iOS encrypts files using the key hierarchy shown in Figure 2.3. More specifically, if iOS writes a new file, it generates a fresh file key that encrypts the file content and gets embedded into the corresponding metadata object. Next, it encrypts this file metadata using one out of four class keys, which are: i) Class A, discarded ten seconds after locking the device, ii) Class B, available when the device is locked to enable background tasks, iii) Class C, available after

[1] `https://web.archive.org/web/20230523084626/https://gs.statcounter.com/os-market-share/mobile/worldwide/`

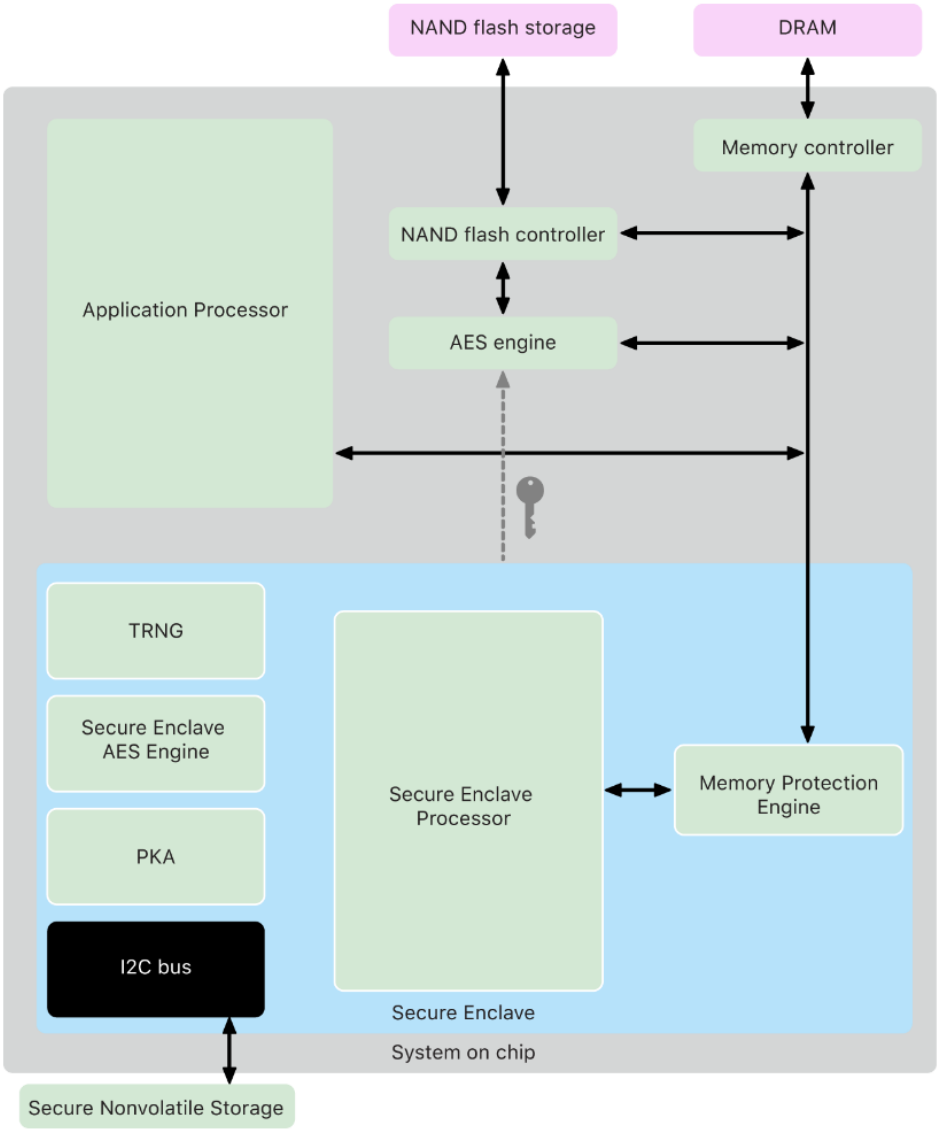


Figure 2.2.: Block diagram of an Apple mobile SoC, showing the Application Processor and Secure Enclave with its components, taken from the official Apple Platform Security Guide [Inc22].

the first unlock, and iv) Class D, available after boot. Finally, Class D keys are encrypted by the UID key, and the other three class keys are encrypted via a key derived from the UID and user's passcode [Inc22].

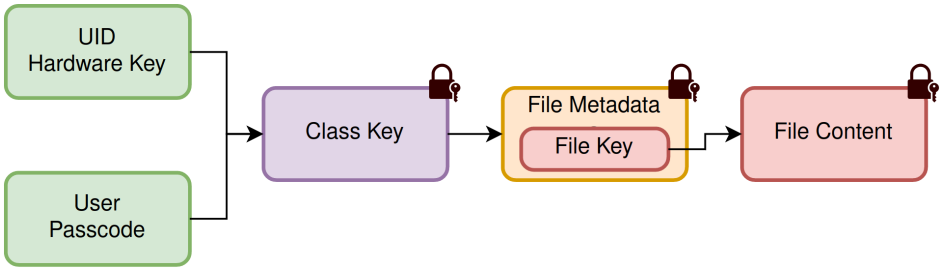


Figure 2.3.: Key hierarchy used by Apple's file-based data protection.

To ensure that apps cannot infiltrate the operating system or other apps, Apple applies a per-app sandbox to all third-party apps. By default, this sandbox only allows access to the files that belong to the current running app and denies access to other files and resources. If the application needs access to specific system information or resources that are not yet assigned to the application, it must consult a well-defined API provided by the operating system. Furthermore, the app must own the correct entitlements to access these functions successfully. Example entitlements are access to user-specific information or iCloud. In addition, Apple also leverages code signing to ensure that the operating system only loads apps signed by an Apple-issued certificate. Enforcing the entitlement and code signing reduces the risk of loading manipulated apps, which, for example, have been binary patched or contain injected entitlement. To reduce the risk of installing malware, Apple does not allow running untrusted code or installing applications from untrusted sources. The official Apple App Store is the only trusted source for installing iOS apps. Before an app is available for download in the Apple App Store, it must complete various security reviews performed by Apple [Inc22].

The Apple Platform Security Guide[Inc22] also encompasses external connectable accessories through Lightning, USB-C, or Bluetooth. Once a connection is made, the accessory must prove its authenticity to the iOS device via a challenge-response process. The accessory must use a custom IC, provided by Apple to authorized manufacturers, to sign the challenge. If this process fails, the Apple device limits access to analog audio and basic UART commands [Inc22].

Security researchers continue discovering new implementation flaws despite all previously shown security mechanisms. These vulnerabilities potentially allow bypassing Apple's security measures, leading to privilege escalation and the possibility of executing untrusted code. The process of acquiring this device state is commonly known as jailbreaking. Jailbreaks primarily focus on compromising the secure boot process, as attacking earlier yields better system control. For example, the jailbreak called checkm8 [2] exploits the earliest stage of the boot process, the read-only Boot ROM, which results in an unpatchable exploit and even provides access to the SE's AES crypto engine.

[2] https://twitter.com/axi0mX/status/1177542201670168576

Chapter 3.

Related Work

This chapter provides facts about related works concerning this master's thesis.

3.1. Juice Jacking

Evil Lightning Cables

Companies like Lambda-Concepts and Hak5 sell off-the-shelf Lightning-to-USB charging cables with an evil implant, allowing them to infiltrate mobile devices, which are named Graywire[1] and O.M.G. cable[2], respectively.

The Graywire cable [Con19] is manufactured for Apple devices only and supports the injection of keyboard and mouse events and capturing the device screen. After establishing a connection with the target phone, an attacker can remotely control the device via an exposed web portal by the evil implant. Unfortunately, Graywire from Lambda-Concepts is private, implying that most concrete implementation details are missing. Nevertheless, their blog explains that Graywire operates as a USB peripheral and states that the USB based Nero protocol provides access to the device screen content.

Conversely, the O.M.G. cable is available as an Apple and Android variation. Its primary function is injecting keystrokes and mouse movements, implying that it does not support video-capturing of the device screen.

Conceptually, the functionality of these two cables is very similar to TuWire. However, they are closed source and undocumented. Furthermore, our work can steal the credentials from the current WiFi connection and proposes a mitigation strategy against USB based Juice Jacking attacks.

Mactans

In 2014, Lau et al. [LJS+13] showed that malware uploads to a connected Apple device are feasible by exploiting Juice Jacking during the charging phase. Their malicious charger, denoted as Mactans, could compromise an Apple device within one minute after connecting it to the charging port. They have exploited the default trust of the iOS version (iOS 6) of that time into tethered connected USB host devices. Mactans could install hidden malware, emulate touches, and take screenshots without user interaction or jailbreak. However, this default trust into USB host devices changed with iOS 7[3]. After connecting the Apple device to a USB host, the user must explicitly grant access to permit communication.

[1] https://blog.lambdaconcept.com/post/2019-09/graywire/

[2] https://shop.hak5.org/products/omg-cable

[3] https://web.archive.org/web/20221222152315/https://theta44.org/2014/07/28/ios-lockdown-diag-services.html

Compared to Mactans, where the Apple device acts as a USB peripheral, our work aims to extract sensitive information by letting the device act as a USB host and overcome the beforehand mentioned trust approval mechanism.

Juice Filming

Meng et al. [MLMK15] proposed a Juice Filming attack applicable to Android and Apple devices. They aim to hide a USB to VGA adapter inside a public charging kiosk and capture the entire screen content during the charging phase. If the user interacts with the device while charging, their attack can automatically extract valuable information like pin codes in a successive postprocessing step. Furthermore, the attack does not require any installed software or explicit permission from the user. Their case study, performed at a public airport, has shown that around 191 test candidates out of 210 charged their phones via a multimedia cart, which proves that users are still unaware of Juice Filming attacks.

Compared to Meng et al. work, this thesis proposes a Juice Filming attack, which does not require any commodity adapters. Instead, our work reimplemented the proprietary Apple protocols, allowing us to emulate the necessary commodity adapters in software. Consequently, the size of the attack logic could be minimized to fit into a standard Lightning-to-USB charging cable. Additionally, our attack device can extract the access credentials for the WiFi network the iPhone is currently connected to and provides a mitigation strategy against USB based Juice Jacking attacks.

JuiceCaster

In their successor work called JuiceCaster, Meng et al. [MLMK16] proposed a tool capable of extracting user input information via OCR through a Juice-Filming attack applicable to Android and Apple devices. Their primary work [MLMK15] captured the victim's device screen content after trusting the USB to VGA adapter and storing it on disk. However, one minute of screen capture produced around 70 to 80 MB of video material. Therefore, JuiceCaster aims to analyze and extract user inputs on the fly by applying state-of-the-art OCR recognition against the grayscale frames. JuiceCaster has achieved an extraction accuracy of over 95%.

GhostTalk

In their work GhostTalk, Wang et al. [WGY22] proposed how an attacker can Juice Jack a smartphone by leveraging the loudspeaker and microphone interface. They took advantage of the multipurpose connector type from contemporary smartphones, which provides access to the loudspeaker and microphone interface alongside the charging functionality. Consequently, the charging outlet also gets implicit access and can mount Juice Jacking attacks against the connected smartphone. Furthermore, tethered microphone access is more resilient to ambient noises than attacks relying on over-the-air voice injection. Their evaluation has shown that GhostTalk can make phone calls and gather personal information via the voice assistant with an accuracy of 100%.

Additionally, Wang et al. proposed GhostTalk-SC, an adaptive eavesdropping system that can extract spoken words from the charging current by leveraging a deep neuronal network. This attack does not require a modified charging cable and delivers a word recognition accuracy of 92%.

3.2. Lightning

Tamarin Cable

Thomas Roth proposed the Tamarin Cable [sta22], a low-cost Lightning debug cable. It is open source and includes a minimal implementation of the IDBUS protocol, leveraging the programmable I/O interface of the Raspberry Pi Pico. The debug cable supports resetting the device, booting into Device Firmware Update (DFU) mode, or accessing internal protocols such as UART or JTAG. Resetting and requesting the desired protocol is accomplished by replaying prior captured IDBUS messages. Furthermore, Thomas Roth has also released a Saleae analyzer plugin for the Apple proprietary IDBUS protocol[4].

However, the Tamarin project does not provide an open read/write API to interact and exchange arbitrary IDBUS messages with an Apple device. Instead, the IDBUS messages that enable Tamarins functions are hardcoded into code. In addition, Tamarin's IDBUS module can only operate as HiFive (Lightning plug) and does not provide an associated response for every request sent by Tristar (Lightning port). In contrast, the IDBUS library developed during this work provides an open read/write API serviceable as HiFive or Tristar.

3.3. Screen Mirroring

QuickTime Video Hack

Daniel Paulus[5] reverse-engineered the USB based screen-sharing functionality between an iPhone and MacBook. His work reveals that the iPhone streams the video frames in real-time in the H.264 format via an Apple proprietary USB protocol to the MacBook. This thesis shows that the USB based Nero protocol exchanged between the iPhone and Haywire adapter, the codename for the Lightning Digital AV Adapter, resembles the same screen mirroring protocol.

However, Daniel Paulus application is tailored for desktop computers and operates in USB host mode, implying that explicit trust between the Apple device and computer is mandatory. Compared to his work, this thesis implemented the screen mirroring protocol for constrained devices operating in USB peripheral mode.

[4] https://github.com/nezza/SDQAnalyzer
[5] https://github.com/danielpaulus/quicktime_video_hack

Chapter 4.

Accessory Link Specifications

This chapter provides a comprehensive summary of Apple's proprietary protocols associated with Lightning, conducted by NyanSatan [Satb], Ramtin Amin [Ami18], and Daniel Paulus[1]. In addition, it incorporates the findings of this thesis. As a result, this section explains the fundamentals of the Lightning connector, how an accessory configures the Apple device leveraging the IDBUS and iAP protocol, and how the Nero protocol enables screen mirroring. This knowledge is required to assemble the TuWire attack and the Lightning Condom mitigation. Section 4.1 introduces the general Lightning connector for mobile accessories and Section 4.4 provides information about the Lightning HDMI adapter. The remaining sections of this chapter cover Apple's proprietary protocols to initiate the connection, authenticate and communicate with the mobile device.

4.1. Lightning Connector

Apple designed and patented the Lightning connector [Inc15], which describes an 8-pin adaptive connector consisting of a plug and port. Their mobile devices, such as the iPhone or iPad, use this connector to establish a tethered connection for charging, data transfer with a host computer, or extending the device capability by connecting accessories. Since the introduction of Lightning in 2012 together with the iPhone 5[2], it replaced the previously used and outdated 30-pin connector. To replicate the same set of functionalities of the 30-pin connector to the significantly reduced 8-pin Lightning connector, the latter allocates its pins adaptively. In practice, an adaptive 8-pin Lightning accessory requests a specific pin assignment from the Apple device so that only the required protocols are dispatched to the pins, significantly reducing the pin demand. The Apple proprietary protocol for requesting the desired pin assignment occupies two of the eight pins. NyanSatan has already reverse-engineered the pinout and functionality of the plug and port [Satb].

4.1.1. Lightning Plug

The Lightning plug is wired on both sides and can be plugged into the port in either direction to improve usability, as shown in Figure 4.1. Moreover, it is also visible that both sides provide the same functions, but the opposite pins are mirrored and not wired straight.

4.1.2. Lightning Port

Contrary to the plug, the Lightning port contains only a single connector lane at the bottom, as shown in Figure 4.2.

[1] https://github.com/danielpaulus/quicktime_video_hack

[2] https://web.archive.org/web/20230000000000*/https://www.apple.com/newsroom/2012/09/12Apple-Introduces-iPhone-5/

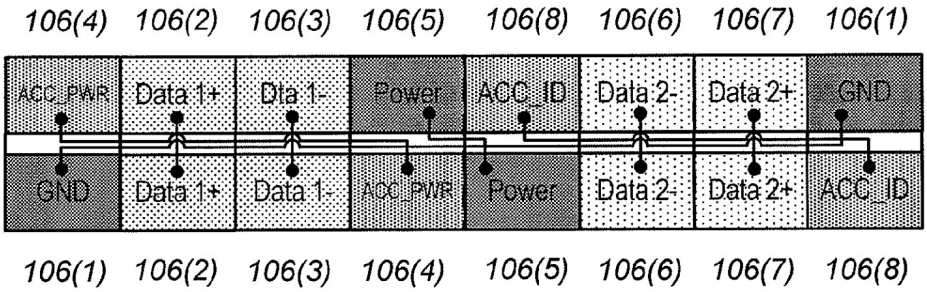


Figure 4.1.: Illustration of a Lightning plug, showing the pin locations of both sides, taken from Apple's patent US8,573,995 B2 [Inc15].

In order to simplify the explanation of the Lightning pinout, we distinguish between i) charging, ii) data, and iii) control groups. The charging group, which includes GND and P_IN, conveys the power and is responsible for charging the mobile device. P_IN and GND's default power levels are +5V and +0V, respectively. Data pairs A and B belong to the data group, and their purpose is adjustable. The iPhone and an accessory agree dynamically on allocating these pins. For example, a reasonable allocation would be UART on Data pair A and USB on Data pair B. Therefore, the final selected protocol determines the exact power level of each data pair. The control group, consisting of ACC1 and ACC2, conveys the IDBUS protocol, introduced later in more detail in Section 4.2. The task of the control group leveraging the IDBUS protocol is to initialize a newly connected accessory and determine the allocation of the data group pins. The power level of this group is fixed to +3.0 V, which makes a level converter mandatory in case of interaction with microcontrollers like the Arduino, which maintains a GPIO output voltage of +5 V.

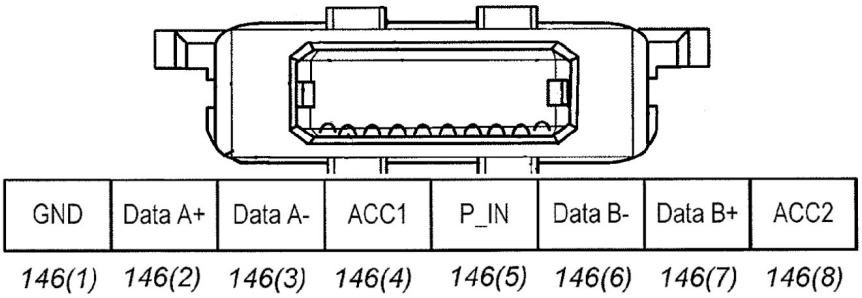


Figure 4.2.: Illustration of a Lightning port, showing the bottom-located single wired connector lane, taken from Apple's patent US8,573,995 B2 [Inc15].

4.2. IDBUS

NyanSatan and Ramtin Amin were one of the first reverse engineering and documenting the physical layer of the IDBUS protocol [Satb; Ami18]. Their work addressed how IDBUS translates binary information into physical data words, its communication partners, and its message structure. This thesis enriched their research by adding message semantics and the structure of the authentication handshake. This handshake initializes a newly connected Lightning accessory, handles the adaptive pin allocation, and contains a power handshake. In

addition, this thesis introduces two approaches to how the IDBUS handshake can be further analyzed on a jailbroken Apple device.

The protocol acts as the communication layer for Tristar and HiFive, which are two integrated circuits (IC) chips introduced in Section 4.2.1 and Section 4.2.2, respectively. Tristar is part of the Lightning port, and HiFive of the plug. After establishing a new physical Lightning connection, these two chips start communicating and are responsible for the initial setup by exchanging operational information and determining the adaptive pinout. The IDBUS protocol resembles a single-wire protocol on the physical layer and exchanges messages based on the request-response pattern. As a result, messages are exchanged without a clock signal at a fixed data rate and through either ACC1 or ACC2, depending on the orientation of the Lightning plug. Except for a single reset message, Tristar initiates the communication by sending the request, and HiFive responds with the reply. However, this work has shown that two exceptions to this protocol definition exist, which do not require a response.

4.2.1. Tristar

As NyanSatan and Ramtin Amin have shown in their works, the Tristar chip is part of the Apple device, and its primary task is to mux the higher-level protocols, such as USB or UART, via the data group to the accessory [Satb; Ami18]. The schematic in Figure 4.3 depicts this assumption, the higher-level protocols are connected on the right, and the Lightning pins are on the left side.

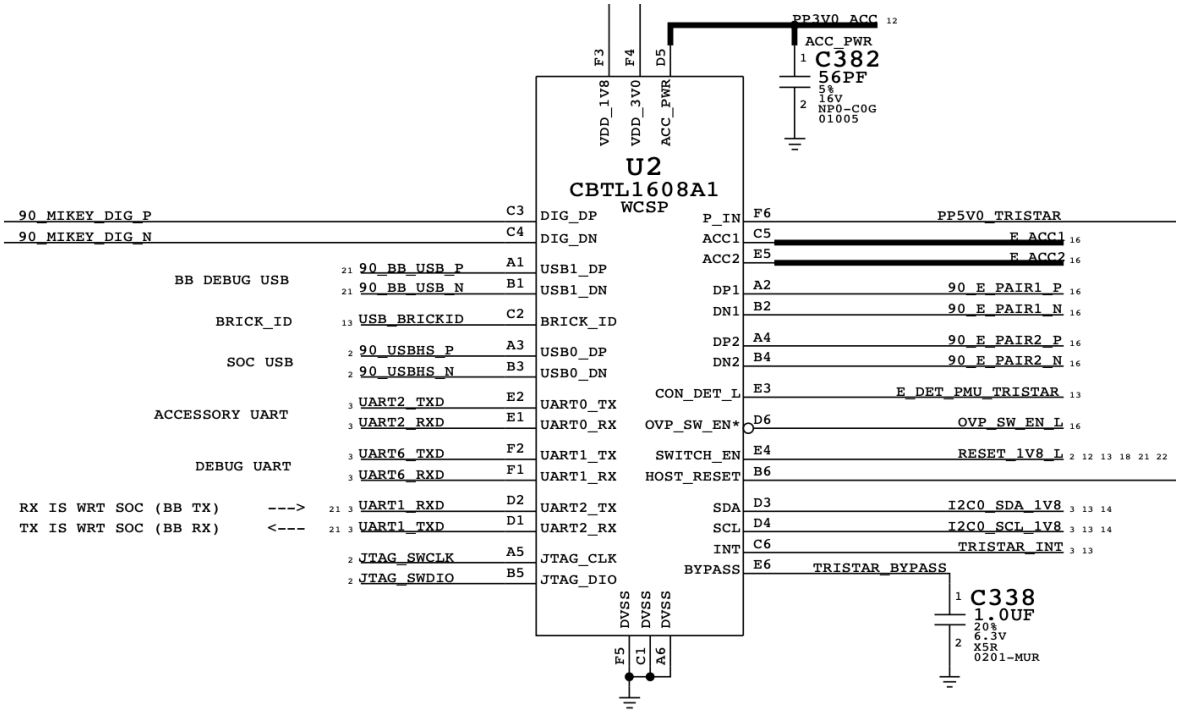


Figure 4.3.: Schematic published by NyanSatan [Satb] showing a Tristar chip with the product number CBTL1608A1 manufactured by NXP.

The exchanged IDBUS messages guide Tristar to determine which left-sided higher-level protocols it should connect to the right-sided Lightning pins. Furthermore, we can observe on the bottom right side an I2C interface (SCL and SDA pins). This interface allows

Tristar to communicate with the main system-on-chip (SOC) from the Apple device. Tristar receives power from either the connected Lightning plug (ACC_PWR) or the Apple device (VDD_3V0, VDD_1V8). This power redundancy ensures operability in the event of a dead device battery or a missing Lightning connection. Until now, three major Tristar revisions have been documented: Tristar, Hydra, and Kraken. The first Lightning-compatible devices incorporated Tristar, which got replaced by its successor Hydra, which added support for wireless charging. Regarding the newest revision, Kraken, only a few details are officially known. However, NyanSatan speculates that this revision of Tristar comes with a new debugging functionality called KIS [Sata]. This thesis will use the term Tristar to refer to any of these chip revisions for simplicity.

4.2.2. HiFive

As shown by NyanSatan, the HiFive chip is part of the Lightning plug and replies to the requests sent by Tristar [Satb]. Its main task is to authenticate itself to Tristar and request access to the high-level protocols accessible through the Lightning data pins. Furthermore, HiFive does not have a battery and is powered by Tristar's control pins.

4.2.3. Data Words

Like other one-wire protocols, the IDBUS protocol transmits data coded as a combination of timing and voltage. Therefore, a byte is converted into a serial bit stream and transmitted one bit at a time. A short or longer-lasting low voltage level is transmitted, representing either a low or high bit value. In addition to data words for transmitting the payload, IDBUS maintains a few control words for synchronization. Each IDBUS word transfer consists of a low and high phase, initiated by a falling and rising edge, respectively, as shown in Figure 4.4. The first phase carries the data word information, and the second phase provides sufficient time for storing the received information. Table 4.1 lists all observed IDBUS words with their respective low and high phase times during this work.

IDBUS Word	**Low**	**High**	**Description**
ZERO	7μs	3μs	zero bit
ONE	2μs	8μs	one bit
ZERO + STOP	7μs	15μs	tailing zero byte bit
ONE + STOP	2μs	20μs	tailing one byte bit
BREAK	14μs	4μs	Tristar pre/postamble
WAKE	24μs	-	HiFive reset signal

Table 4.1.: List of known IDBUS words and their respective meaningful and recovery stage times

Figure 4.4 illustrates a time-voltage transmission diagram of a logical ZERO and ONE in the green and red boxes, respectively. A low phase of 7μs followed by a 3μs high phase transfers a logical ZERO. Conversely, a logical ONE consists of a 2μs low and an 8μs high phase. Each byte is terminated by a 12μs high potential STOP postamble, as shown in the yellow box of Figure 4.4. This detente allocates enough time for processing the received byte value. Additionally, only Tristar inserts a so-called BREAK period at the beginning and the

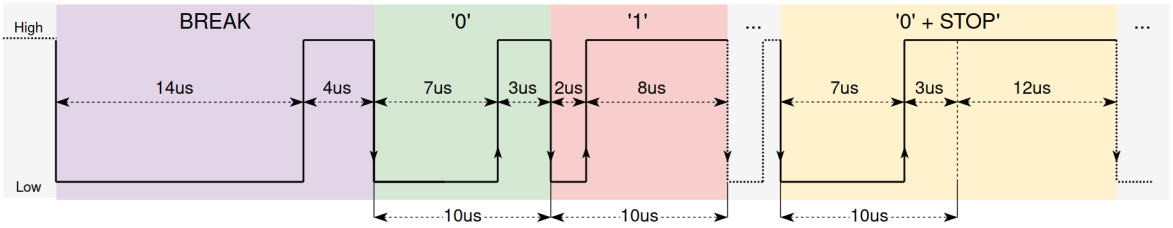


Figure 4.4.: IDBUS data words timing diagram containing a BREAK, ZERO, ONE and ZERO with a STOP postamble.

end of each IDBUS command, whereby Tristar first pulls the line low for 14μs and, second, keeps it high for 4μs, as shown in the purple box from Figure 4.4. Sending a BREAK period announces a new data transmission and allows HiFive to switch into receiving mode in time. Considering the above protocol definition, transferring a single byte via IDBUS consumes 92μs, containing eight bits ($8 * 10\mu$s) and the STOP preamble (12μs). This results in a bandwidth of $86,952 kbits$ per second, neglecting BREAK periods. In addition, the IDBUS protocol supports resets via the so-called WAKE word, which HiFive initiates toward Tristar. This message comprises a 24μs low phase and a high phase with an open end.

The word timings listed in Table 4.1 should act as a reference. They can be observed when connecting the Lightning control pins to a Logic-Analyzer and have been validated by testing them with an iPhone 6. NyanSatan has published possible boundaries for each IDBUS word in his work [Satb].

4.2.4. Commands

The previous Section 4.2.3 presented all known data words of the IDBUS protocol investigated in this work. This section builds upon this knowledge and provides essential information about the IDBUS semantics and available commands.

Each IDBUS packet consists of a header, payload, and footer region as listed in Table 4.2. The header byte specifies the command explicitly and the length implicitly. The payload region is either zero or n bytes long in size and holds the command parameter value as a byte array. The final byte of the packet, the footer, is an 8-bit Cycle Redundancy Check (CRC) checksum computed over the entire IDBUS packet and functions as an error-detection code.

Packet region	**Size**	**Description**
header	1	command identifier
payload	N	N data bytes
footer	1	CRC-8 checksum

Table 4.2.: List of IDBUS packet regions and their respective size.

Like other one-wire protocols that follow the request-response pattern, IDBUS must also associate the sent request with the received response correctly. IDBUS accomplishes this by exchanging a request-response pair at a time and prohibiting interleaved conversations. Moreover, the header value of the response must hold the increment header value of the request. For example, if the header value of a request is 0x74, the header value of the response

must be 0x75. Table 4.3 illustrates all the seen IDBUS commands with their corresponding field names and bindings throughout this thesis.

Message	Region	#	Field/Value	Message	Region	#	Field/Value
Get DigitalID	header	1	0x74	**Ret DigitalID**	header	1	0x75
	payload	2	0x00, 0x02		payload	6	digital ID
	footer	1	0x1F		footer	1	0xNN
Set Charging State	header	1	0x70	**Ack Charging State**	header	1	0x71
	payload	1	0x80 (active) 0x00 (inactive)		payload	0	-
		1	0x00				
	footer	1	0xNN		footer	1	0x93
Get Interface Details	header	1	0x76	**Ret Interface Details**	header	1	0x77
	payload	0	-		payload	1	vendorId
						1	productID
						1	revision
						1	flags
						6	IF serial #
	footer	1	0x10		footer	1	0xNN
Get Interface ModuleNumber	header	1	0x78	**Ret Interface ModuleNumber**	header	1	0x79
	payload	0	-		payload	20	zero terminated serial number
	footer	1	0x0F		footer	1	0xNN
Get Accessory SerialNumber	header	1	0x7A	**Ret Accessory SerialNumber**	header	1	0x7B
	payload	0	-		payload	20	zero terminated serial number
	footer	1	0xB3		footer	1	0xNN
Get Lightning DeviceState	header	1	0x72	**Ret Lightning DeviceState**	header	1	0x73
	payload	0	-		payload	4	device state
	footer	1	0x71		footer	1	0xNN
Apple SystemInfo Notification	header	1	0x84				
	payload	2	property key 0x00, 0x00 (model) 0x01, 0x00 (iOS)				
		1	info length				
		N	info value				
	footer	1	0xNN				

Table 4.3.: IDBUS command structure definition and request-response mapping based on an iPhone 6.

We organized the IDBUS commands into a notification and request group. Commands belonging to the notification group are `SetChargingState` and `AppleSystemInfoNotification`, permitting Tristar to advertise supplementary information to HiFive, such as the current charging status, the model number, and the iOS version of the Apple device. HiFive acknowledges these advertisements by sending no or an empty payload response. Tristar can retrieve essential information from the newly connected HiFive chip by leveraging the request command group. Therefore, Tristar receives the inquired information in the payload region of the response sent by HiFive. The most important command of this group is `GetDigitalID`, which requests a six-byte-long device identifier from HiFive. This identifier determines the purpose of the connected Lightning accessory and settles the port pinout. Furthermore, the messages `GetInterfaceDetails`, `GetInterfaceModuleNumber`, and `GetAccessorySerialNumber` allow Tristar to obtain supplementary information such as serial numbers, vendorId, and

productId from the connected HiFive chip. The command `GetLightningDeviceState` allows retrieving the internal HiFive device state. Unfortunately, the individual bytes' purpose for this message is still unknown.

4.2.5. Authentication

This section describes the handshake procedure between Tristar and HiFive. Once a Lightning connection exists, Tristar must ascertain at which control pin HiFive is present, corresponding to the Lightning plug's orientation. Tristar accomplishes this by alternatively performing a polling process against the ACC1 and ACC2 lines until HiFive responds on one of these lines, enabling Tristar to determine the orientation.

A single polling process comprises three phases: i) charging, ii) request, and iii) response. As stated in Section 4.2.2, Tristar entirely powers the HiFive chip via the control pins. Consequently, Tristar must charge HiFive's internal capacitors at the beginning of each poll by keeping the control pin high for approximately 1 ms. Second, Tristar verifies HiFive's presence by challenging it via the `GetDigitalID` request. If HiFive does not respond with the associated `RetDigitalID` command within 2ms, Tristar encounters a timeout and restarts the same polling process on the other control pin. This process alternates for a total duration of 5s. If HiFive is not present, meaning that neither control pin provides a response, Tristar stops polling, and the handshake fails.

In contrast, if HiFive proves its presence by responding with the `RetDigitalID` command, Tristar continues with the illustrated handshake from Figure 4.5 at the corresponding control pin. The purpose of the other control pin changes now to power delivery only. At this point and throughout the handshake, Tristar announces its current charging state via the command `SetChargingState`, as further explained in Section 4.2.6. Additionally, it requests essential module and accessory information via the commands `GetInterfaceDetails`, `GetInterfaceModuleNumber`, and `GetInterfaceSerialNumber`. After successfully processing each response, Tristar queries the internal HiFive state, leveraging the `GetLightningState` command. Unfortunately, the purpose of these bytes remains unknown, as stated previously. Tristar concludes the IDBUS handshake by informing HiFive about the Apple model number and the iOS build number via two consecutive `AppleSystemInfoNotification` messages. Furthermore, this investigation provides enough information to conclude that the IDBUS handshake contains no cryptographic authentication mechanism.

If an error occurs, Tristar retransmits the same request repeatedly until reaching a command-specific retry count. Errors are timeouts or incorrect payloads, such as mismatched header values, lengths, or checksums. Furthermore, the timeout for responses other than `GetDigitalID` increases from 2ms to approximately 100ms. Additionally, challenging an official Apple HiFive chip by emulating Tristar uncovered that the order of commands following the `GetDigitalID` is inconsequential and that sending this single command is sufficient for pins allocation. The same applies to the charging process. The Apple device will not charge its battery until HiFive acknowledges an active `SetChargingState` notification command. When HiFive receives unrecognized commands, it silently discards them.

After a complete handshake, Tristar changes into listening mode, allowing HiFive to reset the connection by sending WAKE data words. If Tristar detects a WAKE request, it initiates a minimized handshake consisting of the `GetDigitalID` and `SetChargingState` commands. HiFive can reset the connection up to three times. After reaching this limit, Tristar disregards the WAKE data word silently.

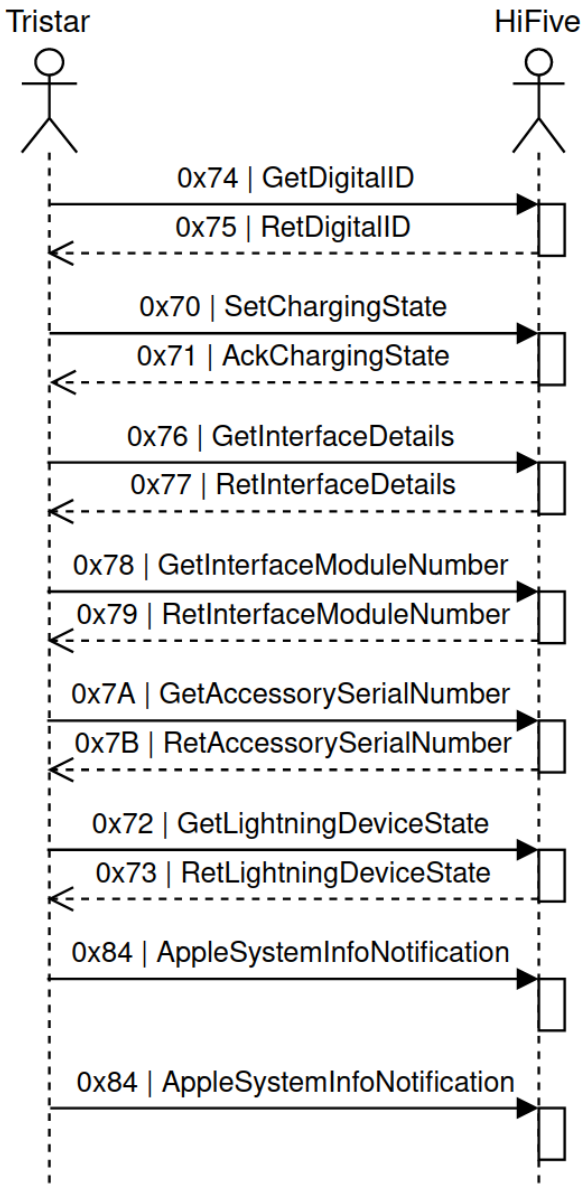


Figure 4.5.: Sequence diagram of an IDBUS handshake from an iPhone 6. Furthermore, the diagram illustrates the command name, header value and data direction.

These observations were made with an iPhone 6, meaning a different Apple device may exchange the messages in a different order or use different IDBUS messages. An obsolete list of possible IDBUS messages was published on Twitter by Nazurbek Kamazov[3].

4.2.6. Power Handshake

The previous section discusses the IDBUS handshake initiated by Tristar and used by HiFive to authenticate itself as a legitimate accessory and request a specific Lightning connector pin allocation. As part of this handshake, Tristar announces whether it draws current from

[3] `https://web.archive.org/web/20220306065229im_/https://pbs.twimg.com/media/D4YO1QwWsAIKHGb.jpg`

the P_IN pin, for charging the Apple device's battery, via the `SetChargingState` command. Analyzing different IDBUS handshakes revealed that the charging behavior of Tristar depends on the digital ID returned within the `RetDigitalID` command. Suppose HiFive returns a digital ID representing an official Lightning-to-USB cable. In that case, once the P_IN pin goes high, Tristar informs HiFive about the attempt to draw current by sending out a `SetChargingState` command, including the payload {`0x80,0x00`}. After receiving the `AckChargingState` command, Tristar draws current and starts to charge the device's battery. If the P_IN goes low, Tristar announces the end of the current charging phase by sending a `SetChargingState` command, including the payload {`0x00,0x00`}. This procedure iterates as often as the P_IN changes its voltage level.

However, the thesis demonstrates that the HiFive chip of other Lightning accessories must perform an additional power handshake to activate Tristar's charging functionality. One example of these accessories would be a Lightning USB-OTG adapter from an unauthorized 3-party manufacturer. NyanSatan already addressed this power handshake in his work, which is carried out as follows [Satb]. HiFive initiates the power handshake by polling the P_IN pin to high. As a result, Tristar initiates two `SetChargingState` commands holding {`0x00,0x00`} as payload, indicating that charging is disabled. After HiFive finishes handling the second `SetChargingState` command, it must temporarily pull the P_IN line to low for 24ms. Once P_IN moves back to high, Tristar sends a `SetChargingState` command, holding {`0x80,0x00`} as payload, which HiFive must acknowledge and completes the power handshake.

Furthermore, we noticed that if HiFive emulates an unauthorized Lightning USB-OTG adapter, it must also inform the Apple device about its charging capability leveraging the iAP commands introduced in Section 4.3.6. Hence, the unauthorized Lightning USB-OTG adapter must perform the IDBUS handshake, including the power handshake, and share its charging capacity with the Apple device to successfully enable charging.

4.2.7. iOS Components

The iOS command line utility `accctl` is very convenient for translating the raw IDBUS binary data into a human-readable format. Listing 4.1 exhibits an example output of an official Apple Lightning-to-USB cable. The output reflects exactly the requested information by Tristar during the IDBUS handshake. Unfortunately, iOS is locked down by default, preventing access to the command line interface. However, users can bypass this restriction by jailbreaking the device, which exploits security vulnerabilities to gain full system control. A jailbroken device permits users to remotely access the command line interface via the secure shell (SSH) protocol. Unfortunately, official iOS builds are significantly stripped down, and only Apple's internal iOS builds include these system utilities. Luckily, a handful of these internal builds have been leaked, allowing a user to extract and transfer the executable to the jailbroken device.

Listing 4.1: Output of accctl system utility executed against a Lightning-to-USB cable.

```
iPhone:/usr/local/bin root# ./accctl list
IOAccessoryManager primary port 1
    Accessory ID: 91 - Digital ID
    Digital ID: 10 0c 00 00 00 00
    Interface device info: vid=2 pid=1 rev=2 flags=0x80
    Interface device serial number: 0x600126ddaa6f
    Interface module serial number: DYG7285UP9VFJYHAY
    Accessory serial number: FC973152926G0NHAS
```

```
    USB connect state: host (active)
    USB current limit: 1500 (base 1500 offset none max none)
    Battery pack: no
    Power Mode: on (supported: off<0mA> on<100mA> high-current<500mA>)
    Sleep Power:  disabled
IOAccessoryPort device port 1 (manager primary port 1, transport type 0)
IOAccessoryPort device port 2 (manager primary port 1, transport type 2)
```

Besides unofficial utilities, the official iOS logging system also exposes valuable IDBUS communication information. Tracing the system log while connecting a new Lightning device shows entries containing the substring `IOAccessoryManager`. Examining these entries in more detail reveals that various processes, such as the `kernel` or `UserEventAgent`, interact with the IDBUS accessory through the `IOAccessoryManager` module. Tracing the logs must be accomplished wirelessly by combining a log reader application like iTools[4] and the `Sync with this [device] over Wi-Fi`[5] feature offered from iTunes[6].

4.3. iAP

iAP (iPod Accessory Protocol) describes an Apple proprietary protocol exchanged between their mobile devices and accessories and allows the accessory to customize the system behavior to their own needs. iAP1, the first protocol revision, was released with the iPod 3G in 2003 and allowed connected accessories to change playback options such as starting, stopping, or seeking the current track. Around 2014, iAP2[7] was released and was intended to replace the outdated iAP1. Both revisions differentiate between privileged and unprivileged commands, and unlocking the latter requires completing a cryptographic authentication process. Despite the availability of iAP2 for several years, recent Lightning accessories like the Lightning Digital AV Adapter[8] still utilize the deprecated iAP1 revision. Since this thesis implements the behavior of the previously mentioned adapter, it will only cover the iAP1 revision.

It is unnecessary to reverse engineer the iAP protocol because multiple official revisions of this standard have already been leaked to the public. The upcoming sections explain the iAP protocol based on a leaked iAP1 standard specification released in 2012.

4.3.1. Transport Layer

In order to keep the iAP adjustable to new emerging technologies, Apple has designed the iAP protocol to be independent of the underlying transport layer. Both releases support exchanging iAP messages over UART, Bluetooth, and USB (device and host mode), which provides high flexibility and enables comprehensive device coverage from tethered to wireless. Low-resource devices can leverage the UART transport layer for communication, which uses the standard 19200–8-N-1 configuration. This configuration uses a baud rate of 19200, a single start bit, eight data bits, no parity bit, and a single stop bit. More powerful devices can employ USB instead of UART. The USB interface for the iAP protocol consists of three endpoints: i) bulk IN, ii) bulk OUT, and ii) interrupt IN. The host uses the bulk endpoints to

[4] https://web.archive.org/web/20230102062918/https://www.itools4.com/

[5] https://web.archive.org/web/20230317193715/https://support.apple.com/guide/itunes/wi-fi-syncing-itns3751d862/windows

[6] https://web.archive.org/web/20230322182947/https://www.apple.com/itunes/

[7] https://web.archive.org/web/20230325182912/https://wiomoc.de/misc/posts/mfi_iap.html

[8] https://web.archive.org/web/20230000000000*/https://www.apple.com/shop/product/MD826AM/A/lightning-digital-av-adapter

handle the data transfer, while the interrupt endpoint allows checking for data availability. An accessory establishes a physical UART or USB connection with the Apple device via one of the two data pairs from the 8-pin Lightning connector. This thesis does not address Bluetooth.

4.3.2. Commands

The iAP packet comprises three regions: i) header, ii) payload, and iii) footer. The header consists of a start field, carrying a byte signature, followed by the payload size, a lingoId, a commandId, and terminated by a transaction number. The lingoId represents a command class, while the commandId represents a specific command within that class. The sender uses the transaction number to uniquely associate a received response to the previously sent request. The payload region contains the parameter values serialized as a byte array. The footer carries a checksum computed over the packet content, except for the start field, which the receiver uses for error detection. Furthermore, the iAP standard groups similar commands by assigning them the same lingoId, and a subset of these groups, called privileged lingos, are only available to the accessory after completing the authentication process explained in Section 4.3.5.

4.3.3. Secure Coprocessor

Section 4.3.2 states that the accessory must complete a cryptographic authentication process to access privileged lingos, explained in Section 4.3.5. Completing this process requires a secure coprocessor sold by Apple only to certified third-party companies. Therefore, delegating the cryptographic calculation to the secure coprocessor implicitly ensures that their mobile devices only trust certified manufacturers' accessories. The communication with the secure coprocessor is accomplished via the I2C protocol. Over time, Apple has released two major revisions.

The first revision (1.0) uses private and public keys, where the private key is fused into the secure coprocessor, and the Apple device stores the public key. Because the first revision is already obsolete, this thesis has not investigated further to retrieve the key size used back then and if each accessory implements the same key pair to confine the amount of stored public keys on the Apple device.

With the introduction of the second major revision (2.0), Apple moved to an authentication process based on standard X.509 Version 3 certificates. Listing 4.2 shows the certificate data extracted from a secure coprocessor embedded into an unauthorized Lightning USB-OTG adapter. Analogous to the first revision, the private key stays inaccessible for the accessory and is fused into the coprocessor. The public key is packed into the certificate, which is signed by a trustworthy certificate authority (CA). This approach eliminates the need to store the coprocessor's public key in advance on the Apple device because it can retrieve it on demand via the certificate. Another benefit of using certificates is that the Apple device can validate the correctness of the certificate and the public key via the signature and issue field. The signature value, in conjunction with the embedded public key, allows the Apple device to verify if the certificate is untampered, and the issuer field, together with the public key infrastructure (PKI), allows for validating if a trustworthy CA generated the certificate. Additionally, the second revision encompasses three sub-releases: 2.0A, 2.0B, and 2.0C. These sub-releases offer the same capabilities and only differ in package sizes and total timeout

threshold. The Apple device triggers a timeout when a certain number of invalid attempts have been reached or a certain amount of time has elapsed.

In the case of the current revision (2.0), a 20-byte long hexadecimal value generated by the Apple device acts as the cryptographic challenge and is transferred to the secure coprocessor. The coprocessor encrypts the challenge using its encapsulated 1024-bit private key and returns a 20-byte long response to the Apple device. Finally, the Apple device decrypts the received response and validates if the result satisfies the challenge requirements. Unfortunately, the exact validation logic is unknown because the publically leaked iAP standard definition does not address this aspect. Nevertheless, this thesis attempts to derive a potential validation procedure, explained later in this section, by accounting for a single authentication iteration containing the challenge, response, and certificate data.

Analyzing the certificate dump, shown in Listing 4.2, of the previously mentioned unauthorized Lightning to USB-OTG adapter allows concluding the following two facts: First, the issue name indicates that Apple introduced a dedicated CA for issuing secure coprocessor certificates for iPod and iPhone accessories with the Common Name (CN) `Apple iPod Accessories Certification Authority`. Second, the Apple devices used for research (iPhone 6 and iPhone 7) do not enforce time validity because they still trusted the unauthorized Lightning USB-OTG adapter, even though the certificate has expired.

Listing 4.2: iAP coprocessor X.509 certificate information from an unauthorized USB-OTG adapter generated via openssl.

```
$ openssl x509 -noout -text -in ./usb_otg_cert.der
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            12:12:aa:13:02:07:aa:04:aa:35:19:aa:95:94:84
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C = US, O = Apple Inc.,
            OU = Apple Certification Authority,
            CN = Apple iPod Accessories Certification Authority
        Validity
            Not Before: Feb  8 17:55:08 2013 GMT
            Not After : Feb  8 17:55:08 2021 GMT
        Subject: C = US, O = Apple Inc., OU = Apple iPod Accessories,
            CN = IPA_1212AA130207AA04AA3519AA959484
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (1024 bit)
                Modulus:
                    00:a9:e2:61:63:b6:22:02:...
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                9A:35:8F:3B:AD:AC:33:63:...
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Authority Key Identifier:
                FF:4B:1A:43:9A:F5:19:96:...
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment,
```

```
            Data Encipherment, Key Agreement
    Signature Algorithm: sha1WithRSAEncryption
    Signature Value:
        7d:39:32:80:9b:4e:26:30:18:ec:...
```

To better understand the secure coprocessor's steps, Listing 4.3 shows how to decrypt a received response with the certificate's public key. The challenge and response payloads of the demonstration example were extracted from a randomly selected handshake iteration between an iPhone 7 and the unauthorized Lightning to USB-OTG adapter. The first two commands display the 20-byte long challenge and the 128-byte long response payloads in hexadecimal. The next two subsequent commands illustrate how to extract the public key information from the certificate and how to decrypt the response. Since the decrypted response already contains the raw challenge bytes, the output of the fourth instruction indicates that the secure coprocessor does not explicitly hash the challenge before encryption. However, it appends the 14-byte long sha1 digest algorithm identifier as stated in the PKCS #1 standard[9]. This analysis indicates that the Apple device may validate the challenge by binary comparing the decrypted response with the challenge after removing the preluding digest identifier.

Listing 4.3: Emulating challenge-response authentication steps, performed by the secure coprocessor and Apple device.

```
# 1) hex payload of the challenge send to the coprocessor
$ xxd challenge.bin
00000000: bac7 4ff9 ae40 d111 3e09 18ff a88d 8218  ..O..@..>.......
00000010: a064 f321                                .d.!

# 2) hex payload of the response received from the coprocessor
$ xxd response.bin
00000000: 7bc3 bbea d120 01a6 9f79 363e 22db ee33  {.... ...y6>"..3
00000010: a1cc 5d88 f53f dcd1 f423 e60b 162c d16f  ..]..?...#...,.o
00000020: 36dd fe04 c039 c7bb e96a 557d 6656 e79b  6....9...jU}fV..
00000030: 6c2d 6e2b 992a b604 ab7a 579d 0d32 f3a9  l-n+.*...zW..2..
00000040: a52c dd49 a7db db40 5d27 9e48 4413 326f  .,.I...@]'.HD.2o
00000050: 4ab9 06ac 4069 f659 dc2c 6902 c552 816c  J...@i.Y.,i..R.l
00000060: ccab 754c d7bb a6b6 0db2 3b36 6a82 e49d  ..uL......;6j...
00000070: 4dd9 1d85 bc60 c1ec dbdc 590f 69c7 9ab4  M....`....Y.i...

# 3) public key extracted from the coprocessor certificate
$ openssl x509 -noout -in usb_otg_cert.der -pubkey
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCp4mFjtiICrYo4XjxjmJ6t3Imt
8J7W1imVYAqGulG9jyJp8mjx+i169YKUu6v8HVW/Si3/fB86NhJRFvSXnoyeTmYu
Iay8eZuDOck5qdHkFwE7Bgla0D+gPNfa/06zReWGqcLgS3dXWelSxvjuaaHaMtu2
cywpwKesgCVuLswvrQIDAQAB
-----END PUBLIC KEY-----

# 4) response decryption using the previously extracted public key
$ openssl rsautl -inkey usb_otg_pkey.pub -pubin -in response.bin | xxd
00000000: 3021 3009 0605 2b0e 0302 1a05 0004 14ba  0!0...+.........
00000010: c74f f9ae 40d1 113e 0918 ffa8 8d82 18a0  .O..@..>........
00000020: 64f3 21                                  d.!
```

[9] https://www.rfc-editor.org/rfc/rfc3447#section-9.2

```
# 5) remove digest AlgorithmIdentifier bytes
$ dd if=decrypt.bin skip=15 bs=1B | xxd
00000000: bac7 4ff9 ae40 d111 3e09 18ff a88d 8218   ..O..@.. >.......
00000010: a064 f321                                 .d.!
```

4.3.4. Identification

Before an accessory can use the iAP protocol, it must announce its presence to the Apple device by finishing the Identify Device Preferences and Settings (IDPS) process. The accessory initiates the IDPS process by sharing its device information and elaborates, together with the Apple device, a commitment to the commonly supported device capabilities, lingos, and transfer sizes. After this process, both parties know whom they are talking to, and which features the communication partner supports.

4.3.5. Authentication

After completing the IDPS handshake, the Apple device initiates the process to authenticate the accessory. The revision of the incorporated secure coprocessor, introduced in Section 4.3.3, guides the Apple device to generate a valid cryptographic challenge. Although iAP1 supports both major coprocessor revisions, this thesis will only discuss the 2.0B authentication sequence.

Figure 4.6 illustrates the authentication process and shows that the cooperation of the Apple device, the accessory, and the secure coprocessor is required to fulfill the handshake. All commands described later belong to the general lingo class. The Apple device initiates the process by sending the `GetAccessoryAuthenticationInfo` request, and the accessory acknowledges it via the `RetAccessoryAuthenticationInfo` command, including the coprocessor revision number and the certificate data. If a single `RetAccessoryAuthenticationInfo` transmission, containing the entire certificate data, exceeds the maximum transfer size negotiated in the IDPS process, the certificate data must be fragmented and iteratively sent to the Apple device. After receiving the entire authentication information, the Apple device validates the certificate data and reports its status to the accessory via the `AckAccessoryAuthenticationInfo` command.

On success, the Apple device proves the accessory's authenticity by transmitting a cryptographic challenge to the accessory via the `GetAccessoryAuthenticationSignature` command. Because only the coprocessor can solve the challenge by leveraging the fused private key, the challenge is forwarded unchanged to it. Next, the accessory starts the signing process and fetches back the signature from the coprocessor. The accessory returns the signature to the Apple device via the `RetAccessoryAuthenticationSignature` command. The Apple device validates it by leveraging the previously received public key and concludes the authentication process by communicating its result embedded into the `AckAccessoryAuthenticationSignature` command.

On success, the accessory gains unrestricted access to all lingos and subcommands. Despite this security wall, our test device, an iPhone 6, trusted the accessory even before the authentication handshake was completed and granted access to all lingos. The Apple device revokes this trust in case of running into a timeout. Considering the coprocessor release 2.0A with a timeout of 75 seconds, an accessory has ample time to harm the system before the trust is revoked. Furthermore, the accessory can reset the timeout counter by re-initiating the handshake, implying access without an official coprocessor. However, this comes at the cost of losing previously requested changes, such as HID handlers or USB host mode.

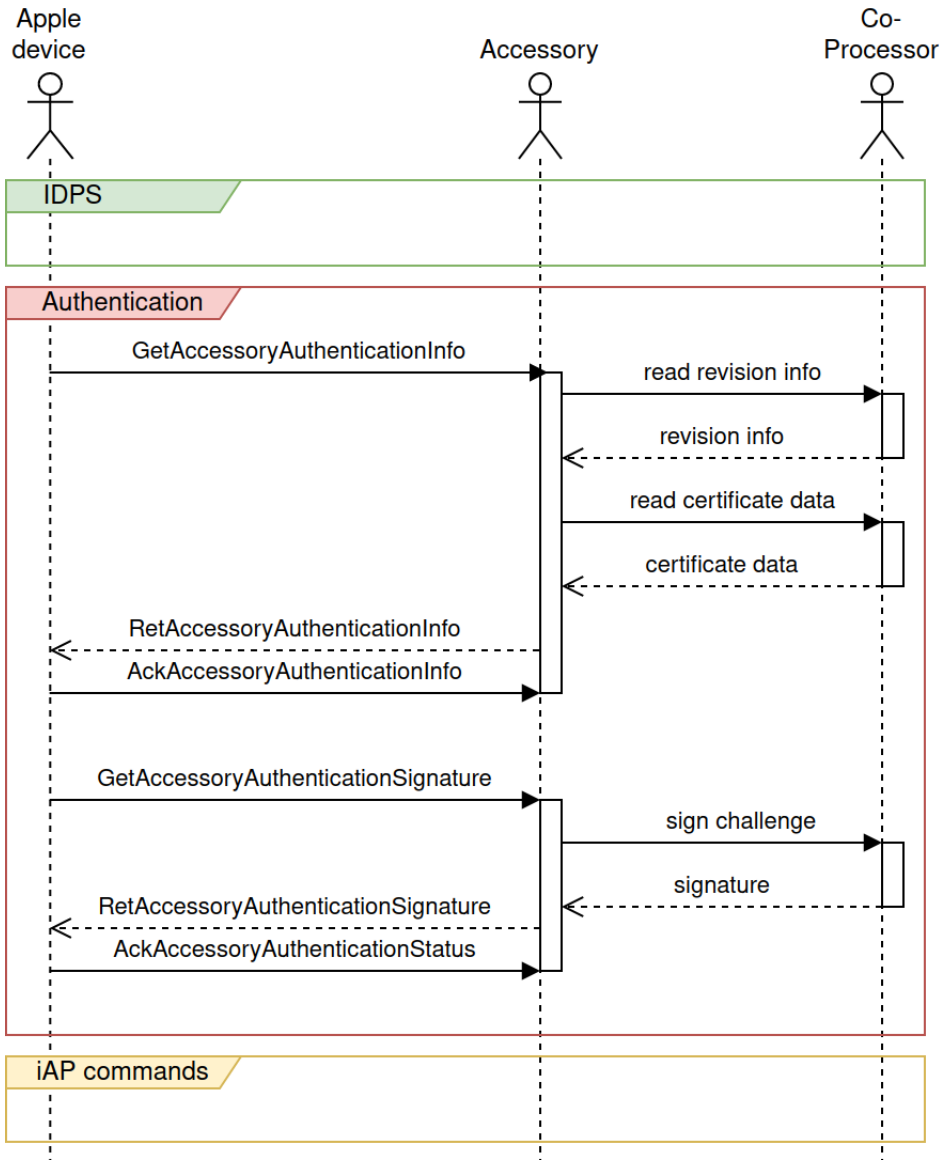


Figure 4.6.: Sequence diagram of an iAP1 authentication handshake utilizing a 2.0B secure coprocessor.

4.3.6. Features

An authenticated accessory can leverage all iAP features to modify and alter the system behavior. This section introduces the essential iAP commands to realize the contributions of this thesis.

USB Host Mode

By default, the Apple device acts as a USB peripheral, allowing for swift data exchange with a host device after establishing a tethered connection. The command `SetiPodUSBMode`, part of the iAP, allows an accessory to change the USB behavior of an Apple device and reinitiate the USB interface in host mode.

HID Events

The iAP protocol offers the capability to control the device's user interface via the injection of HID events. Moreover, the protocol supports handling multiple HID devices concurrently, allowing the accessory to register and simultaneously use multiple input devices, such as a keyboard and mouse. This feature corresponds to the simple remote lingo. It leverages the `RegisterDescriptor` command for registering a new device by providing the HID report descriptor and the `AccessoryHIDReport` command, which wraps the HID event. In this sense, iAP eliminates the overhead of starting an entire USB stack and enables using HID over the lightweight UART protocol.

WiFi Credentials

Another very security-relevant command is the `RequestWiFiConnectionInfo`, which requests the plaintext credentials of the currently connected WiFi access point. The Apple device returns the plaintext credentials via the `WiFiConnectionInfo` command, but only if the user confirms a dedicated sharing prompt. Nevertheless, the user could confirm this prompt unintentionally, or an attacker could temporarily take over the device and confirm the prompt.

Power Control

Accessories designed to power the Apple device can regulate the charging process via the iAP messages `SetAvailableCurrent` and `SetInternalBatteryChargingState`. The first command specifies the maximum current a connected Apple device can draw, and the second permits the accessory to turn on or off charging. These two commands combined allow an authorized accessory to enable charging if the Apple device has not started immediately after the IDBUS handshake, as explained in Section 4.2.6.

4.4. Haywire

The Lightning Digital AV Adapter, also known under its codename Haywire, allows mirroring of the device screen content to external monitors or TVs via an HDMI interface. NyanSatan has shown in his research work[10] that the architecture of the Haywire adapter is straightforward and boils down to a USB peripheral, which exposes a firmware update (DFU) interface. The USB interface is constantly accessible via the Lightning data group A or B, depending on the orientation. The unused control pin, which is not utilized by the IDBUS handshake, provides power to the adapter. Therefore, connecting the Haywire adapter as an ordinary USB peripheral to any host computer is possible. Furthermore, it is also feasible to connect Haywire as a regular USB peripheral through a Lightning USB-to-OTG adapter and bypass its Lightning plug entirely. This experiment concludes that the adapter's functionality only relies on the USB stack, which implies that no specific IDBUS handshake is needed to initialize Haywire and that any USB capable device can replicate its functionality.

Once connected to an Apple device, Haywire receives its firmware from the Apple device via the DFU interface. Therefore, each iOS version comes with a preinstalled version of the Haywire firmware and performs the upload in three stages. At first, the Apple device transfers the bootloader to the device, followed by iBoot, an advanced bootloader, and finally, the actual operating firmware. The final Haywire firmware exposes an iAP and Nero USB

[10] `tinyurl.com/yeand3hx`

interface, which perform an iAP handshake and request the screen content as explained in Section 4.3.5 and Section 4.5, respectively. This thesis extends NyanSatan's work by adding the boot sequence order and the final USB descriptor definition. The boot sequence order was derived from USB traces recorded between the Haywire adapter and a jailbroken iPhone, as explained in Section 4.5. The final USB description was extracted by connecting a booted Haywire adapter to a desktop computer, as explained in Appendix A.

4.5. Nero

Nero is the name of Apple's proprietary USB based screen-mirroring protocol used by the QuickTime player and the Haywire adapter. Initially, we knew that the Haywire adapter relies on the Nero protocol, but the investigation of this thesis revealed that the QuickTime player also utilizes the identical protocol. Daniel Paulus reverse-engineered and documented the protocol with all its commands for the case where the Apple device acts as the USB peripheral. In this setup, the Apple device streams its screen content to the QuickTime application, running on a computer operating in USB host mode[11].

The Juice Filming attack from this thesis extends his work by providing a Nero implementation for resource-constrained devices running in USB peripheral mode. As a result, this thesis analyzes and documents the Nero handshake from a resource-constrained USB peripheral perspective, which slightly differs from Daniel Paulus's work, possibly because his work is tailored for unconstrained USB host devices. In addition, this work proposes a packet structure definition optimized for implementations based on low-level programming languages like C. This work has also proven that it is possible to capture the raw Nero USB traffic via a jailbroken Apple device running the command line utility `tcpdump`, installed via Cydia. Before capturing, it is mandatory to enable the USB interface via the command line utility `ifconfig`, extracted from an Apple internal iOS build and transferred to the jailbroken device or installed via the Cydia package `network-cmds`. Furthermore, combining the outputs from the command line tool `ioreg` and the system logs revealed that the system daemon `mediaserverd` handles the Nero communication on the iOS side. The Apple device identifies a Nero capable USB peripheral, such as the Haywire adapter, by matching the following values: i) 0x12 for idProduct, ii) 0xff for bInterfaceClass and bInterfaceProtocol, and iii) 0x2a for bInterfaceSubClass. Analyzing the official Haywire USB captures and the `lsusb` dump, as shown in Listing A.1, fixed these values. In addition to the USB descriptor values, a Nero USB interface must contain a bulk endpoint for IN and OUT data transmissions. Although the official Haywire adapter uses high-speed mode, the Nero protocol supports USB 2.0 FS and USB 2.0 HS.

4.5.1. Data Objects

The Nero protocol supports exchanging configuration objects serialized into a byte stream format. This section will examine the structure of this byte stream format and guides how to deserialize them into a human-readable format. The building block of these configuration objects are entries, which are either simple key-value pairs or dictionary containers wrapping multiple key-value pairs. To precisely know the size of a dictionary container, the binary representation starts with a four-byte, little-endian encoded size identifier followed by the byte signature `dict`, as highlighted in yellow in Figure 4.7. By nesting dictionary containers,

[11] `https://github.com/danielpaulus/quicktime_video_hack`

the format supports representing complex configuration structures. Figure 4.7 shows the byte stream, ASCII character representation, and the human-readable representation of an entire configuration object. At the bottom, enclosed by the green box, it exhibits all three representations of a single key-value pair. Furthermore, the color coding in all three images is consistent, referring to the same content in all three representations.

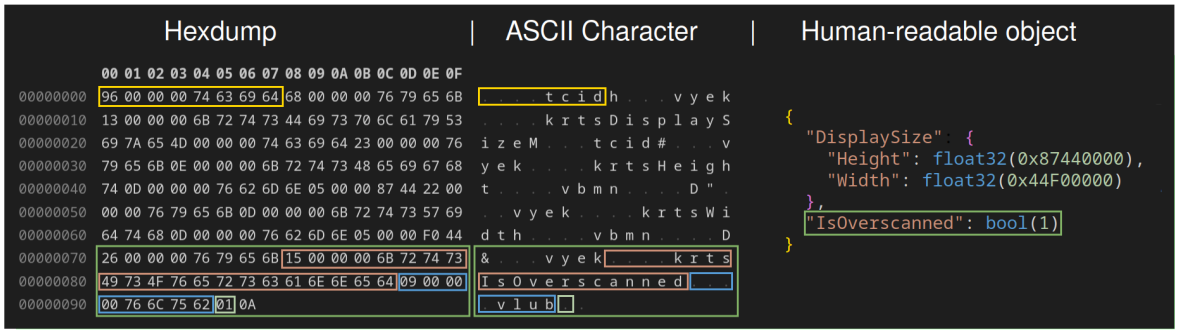


Figure 4.7.: Showing the binary representation of a Nero configuration object containing dictionaries and key-value pairs, highlighted in yellow and green, respectively.

A flat key-value entry, such as `IsOverscanned=true` from Figure 4.7 highlighted in green, is deserialized as follows: The first four bytes, little-endian encoded, represent the total length of the entry, followed by the byte signature `keyv` identifying that the next object is the key. A key object contains again a four-byte little-endian encoded length qualifier at the beginning, followed by the key type signature. The second group from Table 4.4 shows that the key can be either a string or integer identified by the signatures `strk` or `idxk`, respectively. The key's value is stored right after the signature and concludes the key entry. Next, we receive the actual value of the entry. Also, the value is preceded by a four-byte little-endian encoded length qualifier followed by the data type signature and the actual value. The first section of Table 4.4 shows all the known data types. The value of primitive data types, such as boolean, string, or byte array, is appended to the byte stream after the type signature. This logic differs for complex data types like the `NSNumber`[12] and `CMFormatDescription`[13] classes. The `NSNumber` class abstracts different numeric data types based on the official Apple documentation. The first byte after the `nmbv` signature defines the data type, and the rest represents the value in binary. The following byte-to-datatype mappings are known: i) 0x03 = uint32, ii) 0x04 = uint64, iii) 0x05 = float32, iv) 0x06 = float64. The same construction logic applies to the `CMFormatDescription` class, where the binary representation is a concatenation of the fields: i) `CMMediaType`, a 32-bit integer; ii) `CMVideoDimensions`, containing two 32-bit integers; iii) `CMVideoCodecType`, also a 32-bit integer; and iv) an extension dictionary as shown in the last subsection of Table 4.4.

4.5.2. Commands

The structure of a Nero package consists of a header and a command region, as shown in Table 4.5. The header region is mandatory and represents the simplest version of a Nero package. The command region is optional and contains the command, a four-byte ASCII

[12] `https://web.archive.org/web/20230205010711/https://developer.apple.com/documentation/foundation/nsnumber/`

[13] `https://github.com/phracker/MacOSX-SDKs/blob/master/MacOSX10.9.sdk/System/Library/Frameworks/CoreMedia.framework/Versions/A/Headers/CMFormatDescription.h`

Signature Group	Signature (little-endian)	Signature (big-endian)	Description	Example Value
Data Types	vlub	bulv	Boolean value	true, false
	vrts	strv	String value	”ABCabc”
	vtad	datv	Byte array	[0x01, ..]
	vbmn	nmbv	NSNumber	10, 0.10, ..
	csdf	fdsc	CMFormatDescription	see CVRP
	tcid	dict	Dictonary container	{ 1:1, ..}
Key Types	vyek	keyv	key-value pair	vyek
	krts	strk	string key	”key”:
	kxdi	idxk	index key	1:
Subtypes of CMFormat-Description	aidm	mdia	CMMediaType	0x00007363
	midv	vdim	CMVideoDimensions	[0x000002ee, 0x00000536]
	cdoc	codc	CMVideoCodecType	0x61766331
	ntxe	extn	Extension container	{ 1:1, ..}

Table 4.4.: Known byte signatures for the Nero configuration objects, grouped into data, key and subtypes.

value, and the parameter serialized as a byte array. Nero offers four different message types for exchanging data: i) PING, ii) SYNC, iii) RPLY, and iv) ASYN.

The PING message type consists only of the header region, and the communication partners use it to announce their presence. The other three message types, SYNC, REPLY, and ASYNC contain a header and command region, and the communication partners use them to exchange data synchronously or asynchronously. Combining the SYNC and RPLY message types facilitates synchronous data exchange. The sender initiates the data exchange by sending a SYNC request containing a correlation id in the parameter section. Subsequently, upon successful processing, the receiver acknowledges the request by replying a RPLY message and stores the received correlation id in the corresponding header field. Conversely, the ASYN message type allows the Nero communication partners to exchange messages asynchronously, so that no replies are required.

Table 4.6 shows a comprehensive list of known Nero commands with their associated message type and the direction of the data flow. Furthermore, the table exhibits the synchronicity between the SYNC and RPLY message groups.

Packet region	Packet field	Size	Description
header	size	4	total packet size
	message	4	message type
	correlation	8	correlation to bind messages
command	command id	4	four bytes unique command id
	parameters	N	zero to N bytes params stream

Table 4.5.: Nero packet structure and fields with their respective size in bytes.

Message	Command	Direction	Description	Message	Command	Direction	Description
PING	-	DEV → ACC	no command used request presence	**PING**	-	ACC → DEV	no command used confirm presence
SYNC	CWPA	DEV → ACC	init audio clock	**RPLY**	CWPA	ACC → DEV	return new audio clock reference
	AFMT	DEV → ACC	request audio config		AFMT	ACC → DEV	return audio configuration
	CVPT	DEV → ACC	init video clock contains PPS and SPS		CVPT	ACC → DEV	return new video clock reference
	CLOK	DEV → ACC	init additional clock		CLOK	ACC → DEV	return additional clock reference
	TIME	DEV → ACC	request current clock time		TIME	ACC → DEV	return current time of clock
ASYN	SPRP	DEV → ACC	parameter set notification	**ASYN**	HDP1	ACC → DEV	share additional screen configuration
	TBAS	DEV → ACC	time base notification		HDA1	ACC → DEV	share additional audio configuration
	TJMP	DEV → ACC	time jump notification		NEED	ACC → DEV	request new H.264 video frame
	SRAT	DEV → ACC	time rate and anchor notificaiton				
	FEED	DEV → ACC	return new H.264 video frame				

Table 4.6.: Message types and associated commands of the Nero protocol.

4.5.3. Handshake

Before the Apple device permits access to the video frames, the accessory must perform the handshake depicted in Figure 4.8. The handshake aims to create three new clocks on the accessory side for audio, video, and a currently unknown purpose. A 64-bit long value represents a single clock and is incorporated into the message payload whenever a Nero command references one of them.

After establishing a physical USB connection, the two communication partners initiate the handshake by exchanging the PING message with each other and announcing their presence. The Apple device then initiates the clock creation phase, responsible for creating the audio, video, and unknown clock.

This phase starts by creating and exchanging the audio clock references. The clock reference is stored in the parameter field as a flat byte value and exchanged via the SYNC[CWPA], sent to the accessory, and RPLY[CWPA] messages from the accessory. The accessory must store the received audio clock reference for later use.

Before the accessory returns to receiving mode, it transfers two consecutive ASYN messages to the Apple device, containing serialized data objects in the parameter field. The first message, an ASYN[HPD1], contains information about the device display, such as height, width, and whether over-scanning is supported. This message contains a dummy correlation value because it is unrelated to any clock. The second message, an ASYN[HPA1], is related to the audio clock, so it includes the clock reference in the correlation field. Furthermore, it provides supplementary information about the built-into audio device to the Apple device via the parameter field.

Subsequently, the Apple device configures the accessory's audio clock by sending a serialized version of the `AudioStreamBasicDescription`[14] structure and including the clock reference in the header correlation field. The accessory acknowledges the configuration request by

[14] `https://github.com/phracker/MacOSX-SDKs/blob/10dd4868459aed5c4e6a0f8c9db51e20a5677a6b/MacOSX10.9.sdk/System/Library/Frameworks/CoreAudio.framework/Versions/A/Headers/CoreAudioTypes.h`

returning a RPLY[AFMT] message containing a serialized configuration dictionary in the parameter field, holding the error status, and the preferred and default audio channel layout.

This concludes the audio clock setup. Next, the Apple device initializes the video clock, accomplished by exchanging the CVPT command. The request (SYNC[CVPT]), sent by the Apple device, contains the video clock reference and a configuration dictionary, including a serialized version of the `CMFormatDescription` struct. Besides neglectable information, this structure notifies the accessory about the selected video frame format and embeds the PPS and SPS NAL units. After extracting the aforementioned essential information, the accessory returns its clock reference by sending the RPLY[CVPT] response and concludes the video clock setup.

Immediately after the CVPT response, the accessory requests the Apple device's first video frame by sending the ASYN[NEED] message and referring to the video clock. However, instead of returning the frame content immediately, the Apple device postpones the reply and continues the clock creation process. This includes announcing additional configuration about empty video markers via the ASYN[SPRP] message and the creation of the third clock via the CLOK command. Until now, the definite purpose of this clock is unknown, but an internal video frame synchronization algorithm probably uses the clock. This assumption is based on the observation that the creation request (SYNC[CLOK]) refers to the video clock. The accessory returns the freshly created clock reference in the response and receives a TIME command from the Apple device referring to this clock afterward. In return, the accessory provides a serialized `CMTime`[15] object to the Apple device. The actual value is irrelevant and can be extracted from the official Haywire USB capture. The Apple device finalizes the handshake by sending the TBAS, TJMP, and SRAT commands, referring to the accessories video clock. The parameter values of these commands are unknown besides SRAT, which contains a serialized `CMTime` object.

After handling all these messages, the Apple device responds to the first ASYN[NEED] command with the associated ASYN[FEED]. This response carries a serialized `CMSampleBuffer`[16] instance in the parameter field, which encompasses one or multiple raw H.264 NAL unit containers. Each container initially holds the NALU size as a 32-bit big-endian integer, followed by the raw H.264 bitstream data. The data of multiple containers is concatenated and appended after the `sdat` signature from the ASYN[FEED] message. By iteratively polling the Apple device via sending a new ASYN[NEED] command, the accessory can request more video frames as needed. The amount of received video data depends on the current screen activity. Consequently, if the screen content is not changing, the Apple device stops transferring new frame data until a screen content change occurs. This polling approach also serves as a control flow mechanism because the accessory is entirely responsible for requesting new data if prepared.

4.5.4. H.264 Video Composition

The previous section explained how a USB peripheral could obtain the screen content of an Apple device running in USB host mode via the Nero protocol. The takeaway of the section was that the response RPLY[CVPT] encapsulates the H.264 configuration data, and

[15] `https://github.com/phracker/MacOSX-SDKs/blob/10dd4868459aed5c4e6a0f8c9db51e20a5677a6b/MacOSX10.9.sdk/System/Library/Frameworks/CoreMedia.framework/Versions/A/Headers/CMTime.h`

[16] `https://github.com/phracker/MacOSX-SDKs/blob/10dd4868459aed5c4e6a0f8c9db51e20a5677a6b/MacOSX10.9.sdk/System/Library/Frameworks/CoreMedia.framework/Versions/A/Headers/CMSampleBuffer.h`

the ASYN[FEED] the H.264 bitstream data. However, the thesis must still address how to reconstruct a valid H.264 video file from the two Nero messages. Both response messages carry a variable-sized, serialized Nero configuration data object in their parameter field, which makes parsing via C-structs impossible. In order to process these data objects, this thesis suggests developing a dedicated parser that facilitates the value extraction from these data objects.

Important to note is that the proposed reconstruction approach of this thesis does not include timing information, resulting in an inconsistent playback timeline if viewed with an ordinary player such as VLC[17]. However, video playback is still possible without any player or file adjustments. By excluding the timing information, the H.264 file structure becomes straightforward and compiles to a concatenation of the configuration NAL units, SPS and PPS, followed by the bitstream NAL units. Before appending the received NAL unit to the video file, the file processor must add the appropriate preamble, as explained in Section 2.2.

As indicated, the SYNC[CVRP] response contains the SPS and PPS information stored together within a byte array. Unfortunately, this array is stored nested within the extension data object from the `CMFormatDescription` instance, again a component of the main data object from the SYNC[CVRP] response. Unrolling this nested structure requires four steps. First, searching for the string-based `FormatDescription` key within the main data object yields the serialized `CMFormatDescription` instance. Second, the relative extension data object offset within the serialized `CMFormatDescription` must be determined by leveraging the available C-Header files part of the official SDKs. In the third step, we can extract the combined SPS and PPS information from the extended data object by accessing the numeric key `0x00`. Finally, byte manipulation functions allow extracting the PPS and SPS objects from the byte array.

Determining the relative start offset of the bitstream data within the ASYN[FEED] message is less complex than for the SPS and PPS array because it is not stored nested. As stated in the previous section, this response contains a serialized instance of the `CMSampleBuffer` class, which stores multiple bitstream NAL units concatenated together after the `sdat` signature. Therefore, extracting the bitstream data requires first finding and parsing the `sdat` container, consisting of its size and a file signature, followed by iteratively parsing the concatenated NAL units. The size of the `sdat` container represents the total length of the concatenated bitstream NAL units and indicates when to stop reading new NAL units. Furthermore, each NAL unit holds its own size within the first four bytes, determining when a new NAL unit begins.

[17] `https://www.videolan.org/vlc/`

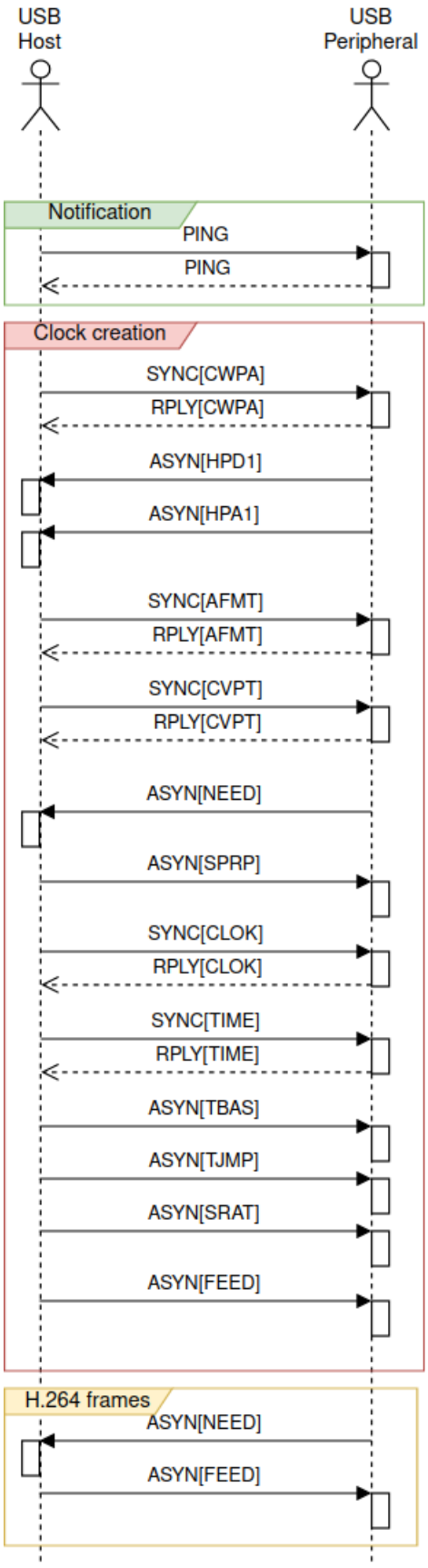


Figure 4.8.: Sequence diagram between an Apple device operating as the USB host and a Nero USB peripheral.

Chapter 5.

Attack Implementation

This chapter explains the structure and implementation of TuWire, a state-of-the-art malicious charging cable for attacking Apple Lightning devices, in more detail. As explained in Section 1.2, the chosen design infiltrates the Apple device by extracting the current WiFi credentials the iPhone is connected to, enrolling a custom MDM profile, and acquiring the device screen content. In order to accomplish these goals, the malicious charging cable must complete an IDBUS, iAP, and Nero handshake, as explained in Section 4.2, Section 4.3, and Section 4.5, respectively. TuWire was entirely designed and developed, from scratch, during this thesis.

The remaining chapter is outlined as follows: Section 5.1 introduces the general design of TuWire on a high level, based on its final schematic. Section 5.2 introduces the hardware components, while Section 5.3 introduces the utilized software libraries to construct TuWire. Section 5.4 concludes the chapter by presenting how TuWire infiltrates an Apple device by leveraging the previously introduced hardware and software elements.

5.1. System Architecture

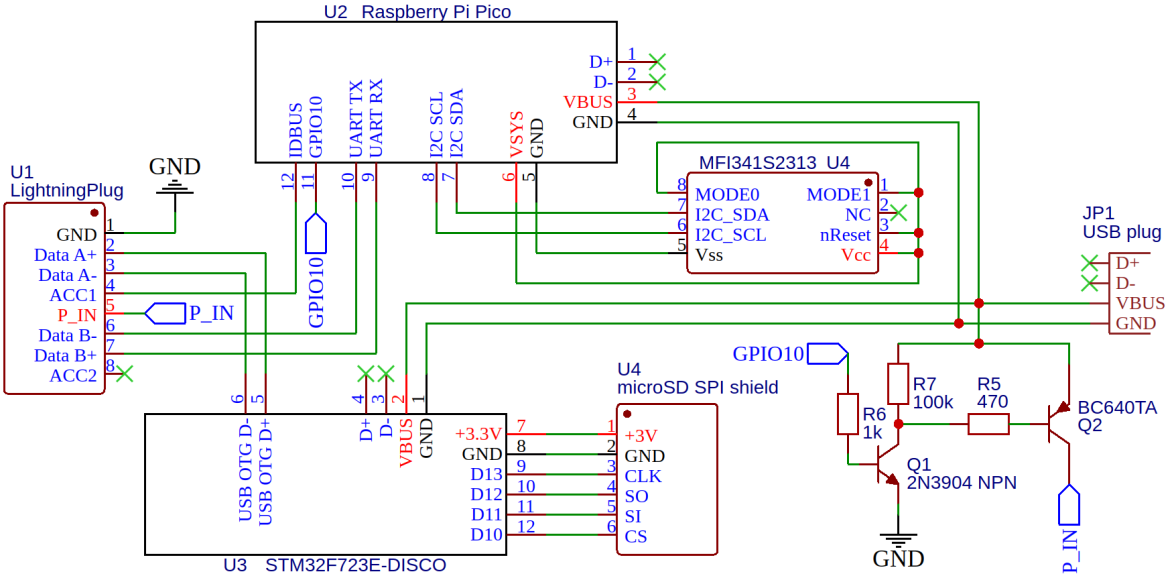


Figure 5.1.: Schematic of the malicious TuWire charging cable.

This section presents the system architecture of TuWire, shown in Figure 5.1, as a schematic diagram. As the schematic reveals, the final architecture consists of the Lightning

and USB connectors, two microcontrollers, a secure coprocessor, an optional SD card breakout board, and some electronic components. Furthermore, the schematic exhibits that the two microcontrollers do not share a common communication channel, implying that they are isolated. This isolation leads to a more modular and flexible architecture because it allows moving part of the logic or replacing entire microcontrollers independently. The selection of the microcontrollers is further explained in Section 5.2.

As the bottom right side of the schematic reveals, the USB plug powers both microcontrollers and delivers power to a connected Apple device through the P_IN pin. To regulate the power source and perform the power handshake, explained in Section 4.2.6, TuWire utilizes the transistor circuit depicted in the bottom right side of the schematic. This circuit facilitates the regulation of the +5 V voltage supply via a +3 V GPIO pin. Because both microcontrollers operate at a +3 V level, powering them via a connection between the VSYS pins and the USB plug would require a dedicated level converter. Powering them via the existing USB voltage regulation stack can circumvent this additional level converter. Furthermore, because TuWire does not aim to mimic a valid USB device on the USB plug, the USB pins (D+ and D-) are kept open.

The Raspberry Pi Pico handles the IDBUS and iAP protocol. Part of the IDBUS protocol is to request from the Apple device to expose the high-level protocols to the Lightning pins. TuWire uses a nice trick that allows wiring the Lightning plug internally statically. By leaving the ACC2 open, we force the Apple device to use ACC1 for the IDBUS handshake, which further settles the pinout for the iAP and Nero handshake to the data groups B and A, respectively. Conversely, the pinout is mirrored if the IDBUS handshake is established via ACC2, implying that iAP utilizes data group A and Nero data group B.

As previously mentioned, the Raspberry Pi Pico also handles the iAP protocol because the official library quickly facilitates the I2C communication with the secure coprocessor (MFI341S2313). This IC is required to complete the iAP authentication process as stated in Section 4.3.3. Consequently, the Raspberry Pi Pico possesses a UART connection with the Lightning Plug to communicate with the Apple device via iAP and maintains an I2C connection with the secure coprocessor. The secure coprocessor also operates at +3 V, implying that the USB plug is unsuitable for powering the device because it may break it. TuWire powers the secure coprocessor via the VSYS pin from the Raspberry Pi Pico, which provides precisely the needed +3 V. To enable the device and the I2C communication interface, the remaining pins of the secure coprocessor (MODE0, MODE1, and nReset) are connected to high, as stated in publically leaked iAP standards.

Compared to the Raspberry Pi Pico, the integration of the STM32 is less complex. This microcontroller only handles the Nero protocol, resulting in a single USB connection between the controller and the Lightning plug. Due to the previously mentioned pin allocation trick, the USB interface of the STM32 chip is fixed wired with the data group A of the Lightning plug. Furthermore, as explained in Section 5.3.3, the Nero module running on the STM32 controller may store the received device screen frames on an external SD card via SPI, which justifies the connection with the SPI SD card shield.

5.2. Hardware Components

This section introduces the used hardware components and discusses their selection. The final electronic design of TuWire, as explained in Figure 5.1, was not known at the outset of this thesis. Therefore, the hardware components have been selected based on the initial research,

concluding that it is mandatory to emulate the IDBUS protocol and the USB based Nero protocol. We selected the Raspberry Pi Pico because of the existing Tamarin Cable project, introduced in Section 3.2, which already provided a minimal, working IDBUS implementation for this board. Unfortunately, the Raspberry Pi Pico does not support developing USB-HS applications. Therefore, we selected the STM32F723-Disco as the second microcontroller because it embeds a USB-HS PHY module, simplifying the implementation of USB-HS applications. The necessity of the iAP protocol, including the secure coprocessors, was not known at the beginning of this work and emerged as we proceeded. Once it was evident that iAP is crucial to complete TuWire, we incorporated the secure coprocessors, with the product number MFI341S2313.

5.2.1. Raspberry Pi Pico

The Raspberry Pi Pico is a low-cost development board designed and maintained by the Raspberry Pi Ltd[1] and incorporates the self-designed ARM-based RP2040[2] SoC. The RP2040 provides two processing cores with an adjustable clock frequency of up to 133 MHz. Among essential microcontroller components, such as I2C, UART, and USB (only 1.1), the SoC also features two Programmable I/O (PIO) instances.

A PIO is a separate processing unit, running a very constrained assembly-like firmware, which allows configuring the general purpose input/output (GPIO) pins very efficiently. Each PIO instance has a dedicated instruction memory capable of holding a firmware image with up to 32 instructions. Furthermore, each PIO contains four state machines, which fetch and interpret the firmware image, implying that these state machines control the GPIO pins. The state machine interacts with the GPIO pins via two 32-bit stateful shift registers, one for reading and one for writing. Because they are stateful, they maintain their state independently and, if configured, reload or flush their current data to the main CPU before running out of capacity. In addition, these shift registers allow the state machine to read and write several pins in parallel. Each state machine can communicate with the central CPU cores via two FIFOs. The direction of each FIFO can be freely set, so they can either transfer from or to the state machine. Two 32-bit long general purpose registers, called `X` and `Y`, allow each state machine to maintain an internal state.

Multiple toolchains are available, encompassing MicroPython[3] and C/C++[4], to program the Raspberry Pi Pico. This thesis will only address the latter mentioned toolchain.

The source code from the Tamarin cable, explained in Section 3.2, already provides a working implementation of the IDBUS protocol for emulating the HiFive chip, leveraging the PIO capability of the Raspberry Pi Pico. Furthermore, the official C/C++ toolchain enables building applications, relying on standard microcontroller protocols, such as UART and I2C, which are vital to assemble TuWire.

[1] `https://web.archive.org/web/20230000000000*/https://www.raspberrypi.com/`
[2] `https://web.archive.org/web/20230410101427/https://www.raspberrypi.com/documentation/microcontrollers/rp2040.html`
[3] `https://web.archive.org/web/20230521132502/https://www.raspberrypi.com/documentation/microcontrollers/micropython.html`
[4] `https://web.archive.org/web/20230424132043/https://www.raspberrypi.com/documentation/microcontrollers/c_sdk.html`

5.2.2. STM32F723-Disco

The STM32F723E-DISCO board is a low-cost development board designed and maintained by ST Microcontrollers[5]. The board is powered by the 216 MHz fast STM32F723IE[6] chip, and allows for rapid application development, leveraging basic and more specific microcontroller interfaces, such as USB-HS and QSPI. Furthermore, the chip has an embedded USB-PHY module, simplifying the implementation of USB-HS applications.

The board is also supported by the Zephyr[7] project, which aims to provide a ready-to-use real-time operating system abstraction layer (RTOS) to constrained devices. Using Zephyr simplifies application development significantly because it includes a large bundle of hardware drivers, covering UART, USB device mode, and even USB Mass Storage Device.

Zephyr has a dedicated configuration and building system built on top of CMake, which utilizes KConfig and Devicetree files for configuration. A KConfig file contains simple key-value pairs, which guide the CMake building system to only include the required Zephyr modules and holds static value definitions, such as thread stack sizes and USB product and vendor IDs. The Devicetree file contains a standardized human-readable representation of the hardware and guides Zephyr to load the correct device drivers and initialize the hardware as specified. Utilizing these Devicetree files alongside a well-defined abstraction layer makes it possible to decouple the application code from the underlying hardware. For instance, if a SD card and flash driver share the same abstraction layer with the application code, the storage backend can be altered via the Devicetree file without touching the application code.

To summarize, the combination of the STM32F723E-DISCO development board and the Zephyr RTOS is well suited for USB-HS applications, such as the Nero protocol.

5.2.3. MFI341S2313

The MFI341S2313 IC belongs to Apple's Made For iPhone (MFI) project. Apple sells these chips to certified third-party manufacturers to permit them to build trustworthy accessories, as explained in Section 4.3.3. However, unauthorized companies sell these chips on the gray market.

The exact chip used in this thesis offers an I2C or SPI interface for communication with the accessory. The signal state of the configuration pins MODE0 and MODE1 determines the communication mode. This thesis utilized and only addresses the I2C interface because it is easier to set up, and Apple removed the SPI option for subsequent versions of these MFI chips. Activating the I2C interface requires setting both configuration pins to a high state. Once activated, the IC operates as a standard 7-bit I2C addressable slave.

The accessory communicates with the IC by writing to or reading from its registers. Each register has a fixed size in bytes, with a maximum capacity of 128. However, some objects, such as the certificate data, exceed the size limit of a single register. In order to store these objects, their data is divided and stored across multiple registers. The accessory can generate a signature of the challenge by writing the challenge to the corresponding IC's register and triggering the signing process by setting the correct bit within the control register. Once the IC has signed the challenge identified via the control register, the accessory can retrieve the signature value by reading the IC's result register.

[5] https://web.archive.org/web/20230311225009/https://www.st.com/en/microcontrollers-microprocessors.html

[6] https://web.archive.org/web/20220818053613/https://www.st.com/en/microcontrollers-microprocessors/stm32f723ie.html

[7] https://www.zephyrproject.org/

This thesis has chosen this specific version of the MFI chip because it was available on the gray market, and its packaging format (SOP-8 2313) enables its incorporation without the need for a dedicated PCB.

5.3. Software Components

This section introduces the required software components developed as part of this thesis, such that TuWire can infiltrate the Apple device. More specifically, this thesis provides, alongside the main contributions, a dedicated library to handle the IDBUS, iAP, and Nero protocols. By separating the protocol-specific logic into distinct libraries, this thesis also provides building blocks for future applications and ensures modularity and reusability. Furthermore, each library employs the platform available logging system to offer additional debug information, which can be turned on or off at compile time. This feature provides valuable information for debugging but introduces additional runtime overhead if enabled.

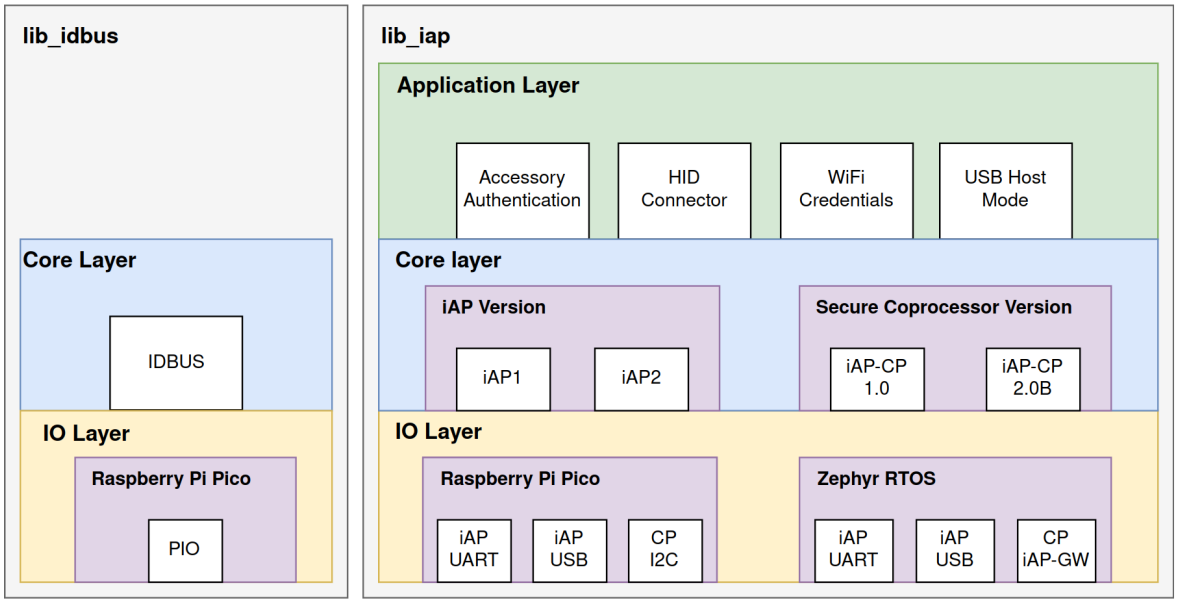


Figure 5.2.: Blockdiagram of the internal layers of the lib_idbus and lib_iap libraries.

5.3.1. IDBUS Communication

For establishing an IDBUS communication channel between an accessory, such as TuWire, and an Apple device, this thesis provides the library called `lib_idbus`. It is designed and implemented during this thesis and provides support to emulate the HiFive or Tristar chip individually or simultaneously. The publicly available API of the library provides high-level functions to initialize a new IDBUS communication partner and to perform the handshake. Furthermore, it separates the hardware-specific IO operations via an internal abstraction layer from the core IDBUS logic, as shown in Figure 5.2. The interface between these two layers is a set of shared IO functions (open, read, write, close) defined and used by the upper core layer while implemented in the lower IO layer. As a result, the task of the core layer is to translate, back and forth, the high-level IDBUS message structures into a binary representation, which is passed on to the IO layer. The IO layer finally performs the actual

I/O operations. This separation makes the library more modular, and porting it to a new microcontroller only requires the implementation of the IO layer. The build system used to assemble the library allows selecting which IO layer should be used at compile time. The current release of the library comes with a single IO layer implementation tailored for the Raspberry Pi Pico leveraging the PIO.

The library's initial iteration builds upon the existing code provided by Tamarin Cable. Unfortunately, the existing code was unsuitable for developing a library capable of emulating both IDBUS components (HiFive and Tristar) and providing a flexible read/write API. Therefore, we started to write the IDBUS library from scratch and incorporated Tamarin's idea of translating the binary information via the PIO into the associated IDBUS data words.

Core Layer

The core layer of the `lib_idbus` library aims to abstract the complexity of the IDBUS protocol by providing a high-level API to the user. This API includes functions to initialize a new IDBUS communication partner (HiFive or Tristar) and subsequently perform the respective IDBUS handshake, as explained in Section 4.2.5. If arbitrary message exchange is needed, the core layer also serves a low-level API granting access to the shared IO functions between the core and IO layers.

If an application wants to communicate with an IDBUS device through the library's high-level API, it must perform the following steps. Initially, the application must reset and prepare the initial state of the internal data structures by calling the `idbus_init` function. After the initialization phase, the application must create a new Tristar or HiFive configuration object, holding all the properties of the chip to be emulated. Using the created configuration structure jointly with the initialization functions `idbus_init_hifive` or `idbus_init_tristar`, the library allows creating a new HiFive (`idbus_hifive_t`) or Tristar (`idbus_tristar_t`) instance. Finally, to perform the handshake either as HiFive or Tristar, the library provides the functions `idbus_do_handshake_hifive` and `idbus_do_handshake_tristar`, respectively. As the library supports emulating multiple IDBUS components concurrently, the functions above accept an optional configuration object as the second parameter, which assists the IO layer in identifying the correct instance. For the Raspberry Pi Pico PIO IO layer, the configuration object contains the GPIO pin that should be used for the handshake. If a handshake as HiFive is triggered, the library will block until Tristar initiates the handshake and leverages the GPIO pin 10 to control the power delivery via the transistor network shown in Figure 5.1. Conversely, if emulating Tristar, the library assumes an already connected HiFive chip and immediately starts with the handshake. Listing 5.1 shows how to initialize a new HiFive instance and perform the IDBUS handshake against a Tristar, leveraging the high-level API of the library `lib_idbus`.

Listing 5.1: Code showing how to initiate and perform an IDBUS handshake as HiFive, utilizing the high-level API of the lib_idbus library.

```
// initialize lib_idbus by calling the idbus_init function
err = idbus_init();

// initiate a new HiFive configuration object
idbus_hifive_info_t hifive_info = {
    .vendor_id = 0x01,
    .product_id = 0x25,
    .revision = 0x80,
```

```
    .digital_id = {0x11, 0xF0, 0x00, 0x00, 0x00, 0x00},
    .interface_serial_number = {0xA0, 0x6A, 0x8D, ..},
    .interface_module_number = {0x44, 0x57, 0x48, ...},
    .accessory_serial_number = {0x43, 0x30, 0x38, ...},
    .accessory_state = {0x00, 0x00, 0x00, ...}
};

// allocate some memory for the new HiFive instance
idbus_hifive_t hifive;

// initiate new HiFive instance, based on previously
// defined configuration object
err = idbus_init_hifive(&hifive, &hifive_info);

// perform the handshake against an Tristar chip
// and pass the IO related configuration in the second argument
err = idbus_do_handshake_hifive(&hifive, (void *)IDBUS_GPIO_PIN);
```

An application that requires exchanging arbitrary IDBUS messages can utilize the library's low-level API. An example of such an application would be a protocol fuzzer. Before an application can read or write arbitrary IDBUS messages, it must first open a new HiFive or Tristar instance using the function `idbus_open_hifive` or `idbus_open_tristar`, respectively. Compared to the high-level API, which returns an instance object, the low-level API functions return a reference to the IO layer instance. In conjunction with the functions `idbus_write` and `idbus_read`, this IO layer reference allows exchanging of arbitrary IDBUS messages. Once finished, the application should close the low-level instance by calling the `idbus_close` function. The following subsections will discuss the IO layer in more detail.

IO Layer

The IO layer of the `lib_idbus` library receives the IDBUS messages from the core layer and is responsible for translating them into the correct physical IDBUS data words, introduced in Section 4.2.3. The data exchange between the layers occurs through the core's low-level API functions, defined in the core layer, while implemented in the IO layer.

The current release of the library only includes a single IO layer implementation, leveraging the PIO of the Raspberry Pi Pico. Therefore, this section only focuses on this implementation. As explained in Section 5.2.1, a PIO can run a tiny custom firmware and communicates with the central CPU via two FIFOs. Even though the instruction memory is tiny, it is sufficiently large to hold a conversion logic, which either allows the translation of a byte value into the correct IDBUS data words or vice versa. Moving the translation logic into the PIO significantly simplifies the main program running on the CPU. On a write, the library loads the sending firmware into the PIO and pushes the bytes to send to the state machine via the FIFO. Because the PIO and the main code are running independently, and the library restricts interleaving message sending, the main code of this implementation blocks until the PIO has finished sending. On a read, the prepared PIO reading firmware is loaded, and the main program can pull the byte values from the FIFO after the state machine has received and pushed them properly.

While implementing the IO layer, we encountered multiple problems and found the following solutions. As stated before, when sending data to a connected IDBUS device, the PIO code receives the bytes to send via the FIFO. For synchronization, the main code informs the PIO code about the number of bytes to send, which allows the PIO code to report back when it

completes the entire sending process precisely. However, because the PIO code knows how many bytes to send, it will block if the FIFO is empty and the sending counter is still greater than zero. This blocking introduces an undefined behavior on the physical IDBUS layer. To lower the risk of blocking the PIO code during sending, this thesis serializes the high-level IDBUS message structure into a consecutive array. The rationale behind this approach is that accessing a single, compact byte array results in a more consistent execution time than retrieving the individual IDBUS message parts from different memory regions, allowing the main code to fill the FIFO in time.

If emulating the reading process of the HiFive chip, it is essential to interpret the BREAK signal sent by Tristar. As stated in Section 4.2.3, the BREAK signal identifies the start and the end of a single IDBUS message transaction. Hence, reading and interpreting the BREAK signal yields the fastest way to identify the end of Tristar's transaction. Because Tristar expects the response of the first handshake message within 2ms, stopping reading and handling the received IDBUS request as fast as possible is vital. An earlier version of the library utilized a timeout to terminate reading. Unfortunately, this approach proved inefficient because it wastes unnecessary time waiting until the timeout expires and may lead to violating Tristar's response time constraints.

In the context of emulating Tristar, it is crucial to start reading HiFive's response immediately after finishing sending the request. Our research concludes that there is no mandatory bus idle time between the end of Tristar's request and the first byte of HiFive's response. This lack of delay regulation implied that some unauthorized HiFive chips sent their response immediately after the request. To support these HiFive chips, the PIO code responsible for sending data, as Tristar, also incorporates the reading procedure. As a result, the main code does not need to reload any PIO code, enabling efficient handling of HiFive responses that are sent as rapidly as possible.

5.3.2. iAP Communication

Like for the IDBUS protocol, this thesis also provides an iAP library, called `lib_iap`, which allows accessories, such as TuWire, to establish an iAP communication channel with an Apple device. It was written from scratch during this thesis and is based on an official, publicly leaked iAP standard released in 2012. The library supports instantiating multiple iAP channels concurrently. In addition, the library includes support to interact with the secure coprocessors. The exposed high-level API allows a client application to authenticate, inject HID input events, extract the WiFi credential, and switch the Apple device into USB host mode.

To keep the library extendable and maintainable, we have separated it into an application, core, and IO layer, as shown in Figure 5.2. The interaction between two layers is accomplished via shared functions, defined in the upper and implemented on the lower layer. The application layer exposes the previously defined high-level API and leverages the core layer. The core layer contains the iAP version-specific logic and is responsible for translating the high-level iAP message structures into a binary format and vice versa. During this translation, the core layer validates the correctness of the received and sent iAP messages. The IO layer handles the actual physical IO operations. The building system allows the selection of the desired components at compilation time. This selection encompasses the hardware, the iAP, and the secure coprocessor version.

Despite this modular design, the current release only implements the essential modules to operate TuWire. Therefore, on the core layer, only `iAP1` and `iAP-CP 2.0B` are implemented,

and on the IO layer, the USB support for the Raspberry Pi Pico is missing. The following subsections will discuss the individual layers in more detail.

Application Layer

The application layer aims to abstract the iAP protocol stack via a high-level API that includes the abovementioned functions.

An application, such as TuWire, can leverage the high-level API of the library to authenticate itself as a legitimate accessory, as shown in Listing 5.2. To accomplish this, first, it must construct a new transport instance (`iap_transport_t`) by calling the `iap_init_transport` function. This function initializes the hardware and returns a valid transport instance on success. Furthermore, it must construct a new secure coprocessor instance (`iap_cp_t`) by calling the `iap_cp_init` function. After creating a new transport and secure coprocessor instance, the application can authenticate itself against the Apple device by executing the function `iap_authenticate_accessory` consuming the two newly created instances as parameters. On success, the Apple device trusts the application and grants access to the privileged lingos.

After the authentication process, tasks such as changing the USB mode, extracting the current WiFi credentials, enabling charging, or injecting HID events are accomplished via the high-level API functions `iap_request_usb_host_mode`, `iap_request_wifi_credentials`, `iap_enable_charging`, and `iap_hid_send_report`, respectively.

Listing 5.2: Code showing how to authenticate an accessory against an Apple device via UART, leveraging the high-level API of the lib_iap library.

```
// open a new iAP transport instance via UART with default config (NULL)
iap_transport_t iap_trans
err = iap_init_transport(&iap_transport, IAP_UART, NULL);

// open a new iAP coprocessor instance with default config (NULL)
iap_cp_t iap_cp;
err = iap_cp_init(&iap_cp, NULL);

// authenticate the accessory
err = iap_authenticate_accessory(&iap_transport, &iap_cp);
```

Core Layer

The library's core layer contains all the version-specific implementations of the iAP and iAP-CP protocols. However, at compilation time, only a single combination of iAP and iAP-CP can be activated via the build system. The rationale behind enabling only a single iAP and iAP-CP version is that this allows exposing an iAP and iAP-CP version independent API to the application layer. Section 4.3 describes the protocols mentioned above in more detail.

IO Layer

The primary objective of the IO layer is to abstract the underlying hardware and to provide a consistent set of IO functions to the core layer. These IO functions encompass physical communication with the secure coprocessor and an iAP device like an iPhone. Analogous to the core layer, the build system also influences this layer. For the current iteration, the build system allows choosing if the IO layer should be compiled for the Raspberry Pi Pico or

a Zephyr board. The library supports iAP over UART for both hardware types and USB only for Zephyr boards.

Regardless of the selected hardware, the IO layer exposes the following platform independent functions to the core layer. For initializing the UART or USB hardware and the iAP instance itself, the IO layer provides the internal library functions `_iap_init_transport_uart` and `_iap_init_transport_usb`, respectively. Subsequently, the newly constructed iAP transport instance in conjunction with the internal functions `iap_transfer_out_{uart,usb}` and `iap_transfer_in_{uart,usb}` facilitate communication with a connected iAP partner. In addition, the IO layer also provides an initialization (`iap_cp_init`), read (`iap_cp_read_reg`), and write (`iap_cp_write_reg`) function to interact with the secure coprocessor.

While implementing the IO layer, we encountered multiple problems and found the following solutions. First, it is vital to always listen for incoming messages sent by the Apple device because the protocol also supports aperiodic notification messages. Dropping these messages involves losing information and could corrupt the internal iAP library state. Therefore, we recommend using an interrupt-based receiving logic and storing the received bytes temporarily in a buffer. On a read request from the upper core layer, the IO layer returns the bytes from this buffer instead of physically listening for new incoming data. This design guarantees that all bytes are received and processed correctly.

The second problem was a temporary misconception regarding the necessity to perform an iAP handshake by the STM32 board during the Nero handshake. This suspicion arose during the analysis of the official haywire adapter. However, this thesis proves that Nero does not require an iAP handshake to function, leading to the exclusion of this solution from the final architecture of TuWire. Nevertheless, we would like to address the problem and solution we found. The problem is the lack of compatibility of the I2C interface from the Zephyr RTOS and the secure coprocessor. Specifically, the coprocessor only supports a clock speed of up to 50 kHz, while the slowest supported speed by Zephyr is 100 kHz. To overcome this limitation, we connected the coprocessor to a Raspberry Pi Pico and leveraged it as an iAP-UART gateway. The Pico constantly listens for incoming iAP messages from the Zephyr board, which leverage a custom lingoId and payload structure. Once it receives a new custom iAP message, the Pico extracts the request and forwards it to the secure coprocessor. After receiving the coprocessor's result, the Pico returns it to the Zephyr board by wrapping it into the agreed custom iAP format.

5.3.3. Nero Module

This thesis implements the Nero protocol as an independent Zephyr module, allowing client applications to enable the module via the configuration system (KConfig). The current iteration of the module offers two modes of operation. It can either solely request the screen frames without storing them on a storage device or with storing them. If storing the frames is enabled via the KConfig file, the module collects, and stores all received frames in a file called `IDEV_REC.264` until a predefined data threshold is reached. This threshold is configurable via the KConfig system. The current release of the Nero module supports storing the received frames either on raw flash memory or on an SD card.

If storing the frames on a raw flash is enabled, the Nero module also exposes a Mass Storage Device alongside the Nero USB interface for retrieving the recording. In addition, it initializes a LittleFS file system on the flash memory for storing the frames. We chose LittleFS as the file system because of the following points. Firstly, it comes with Zephyr, making it a convenient option for integration. Second, LittleFS offers good stealthiness because common

operating systems, including the thesis evaluation device, neither inform the user about a newly connected storage device nor automount the partition. Using a file system technology offering these properties is crucial because of the embedded Mass Storage Device interface. The user could get suspicious if the target operating system (iOS) automounts and provides access to the Mass Storage Device. In addition, this lack of automount eliminates the necessity to interconnect the STM32 chip with the Raspberry Pi Pico. The STM32 can always expose the Mass Storage Device because the iPhone ignores it due to the chosen file system. Therefore, LittleFS results in a stealthier and more straightforward implementation if flash memory is used as the storage backend. The flash storage device is configured via the Devicetree file and must contain a `fixed-partition` node[8] with the ID `storage_partition`.

If the Nero module leverages an SD card as the storage backend, it uses the FAT partition schema and does not expose a USB Mass Storage Device interface. The necessity of the USB Mass Storage Device interface drops because retrieving the recording is accomplished by ejecting and inserting the SD card into a host computer. This elimination also prevents the exposure of the storage medium to the victim device, allowing the selection of a more common file system like FAT. Besides the already mentioned simplification, using an SD card with FAT allows consolidating frame data within a single file up to 4 GB in size. The SD card storage device is configured via the Devicetree file and must contain a `sdhc` node[9].

During the boot process, the Nero module exposes the USB interface, explained in Section 4.5, and starts a dedicated thread, which handles the Nero protocol as follows.

After an Apple device announces its presence via the PING message, the module performs the handshake and continuously requests the device screen content. As shown in the left column of Figure 5.3, the module immediately blocks the USB stack after receiving a new USB packet from the Apple device. Finishing the current received USB packet before releasing the USB stack and accepting new packets enforces backpressure against the Apple device and prevents the module from running out of RAM. After each USB packet, the module verifies that enough space is available on the storage device if explicitly enabled via KConfig. If not, the module closes the file and terminates. If dumping the video frames is disabled, the module will never stop and continuously request new video frames from the Apple device.

The right column of Figure 5.3 illustrates how the module processes a single USB packet. A USB packet contains either single or multiple Nero messages. Consequently, this function must process the content iteratively until no more data is available. Each iteration performs three steps: i) parsing, ii) processing, and iii) responding. However, ASYN messages do not require sending a response, implying that the last step is neglected for this message type.

First, the module must parse the Nero message type and the corresponding command. Second, it parses the necessary information, such as clock references, and populates the internal data structure. When receiving the CVRP command and dumping is enabled, the module extracts the PPS and SPS and stores them in the recording file. The same applies to the FEED command, but the module parses and stores the H.264 frames instead of the parameter sets. Handling the H.264 frames is the most sophisticated process of the entire Nero module because it requires temporarily releasing the USB stack lock to paginate the FEED message due to its size. Section 4.5.4 provides more information about how the module parses the PPS, SPS, and FEED information. Finally, the module prepares and transmits

[8] `https://web.archive.org/web/20230208005341/https://docs.zephyrproject.org/latest/build/dts/api/bindings/mtd/fixed-partitions.html`

[9] `https://web.archive.org/web/20230521054250/https://docs.zephyrproject.org/latest/services/storage/disk/access.html`

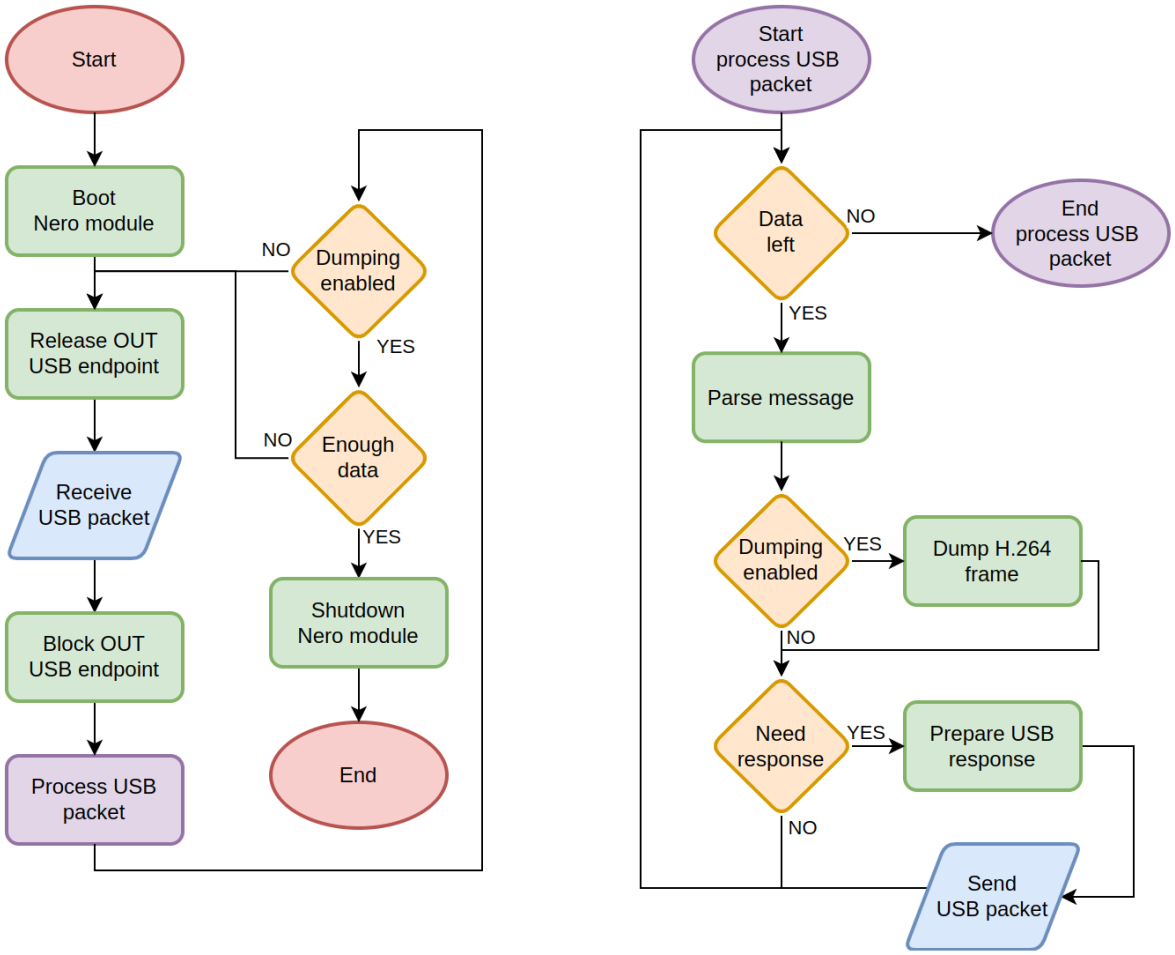


Figure 5.3.: Flowchart of the Nero module implementation in the left column and the Process USB packet processes illustrated in the right column.

one or more responses if required. If no more data is available for processing, it returns and releases the USB lock to accept a new USB packet.

5.4. Attack Approach

This section discusses how TuWire, a state-of-the-art malicious charging cable for Apple Lightning devices, can successfully leverage the previously introduced software elements to mount Juice Jacking attacks. The hardware components and their interconnection have already been addressed in Section 5.2. The proposed attack of this thesis describes a public charging kiosk equipped with TuWire. These charging kiosks allow hiding the microcontrollers within the charging station and only expose the Lightning cable. Furthermore, the two microcontrollers remain powered on and continuously listen for new coming requests. Once a victim connects their Apple device for charging, the two microcontrollers, part of TuWire, can infiltrate the device as follows. As discussed in Section 5.3.1, the Raspberry Pi Pico initially utilizes the `lib_idbus` library to perform the IDBUS handshake. Specifically, it initializes a new HiFive instance with the value {`0x11, 0xF0, 0x00, 0x00, 0x00, 0x00`} as digital ID and performs the handshake by leveraging the high-level API functions of the

`lib_idbus`. Using this exact digital ID is crucial because it ensures that the Apple device grants access to the iAP and USB interface. We extracted it from an arbitrary IDBUS handshake between an iPhone 7 and an unauthorized Lightning USB-OTG adapter. On success, the Raspberry Pi Pico continues with the iAP handshake, using the library `lib_iap` introduced in Section 5.3.2. It creates a new instance for the iAP transport and the secure coprocessor, followed by performing the authentication processes. As a result, the Apple device trusts the Raspberry Pi Pico and permits access to the privileged lingos. To avoid any suspicion, the Raspberry Pi Pico immediately informs the Apple device about its charging capabilities, which finalizes the power handshake and enables the Apple device's charging functionality. In the next stage of the attack, the Raspberry Pi Pico opens two new iAP HID handlers to emulate keyboard and mouse inputs. If not secured via a pin code, the keyboard handler unlocks the device by sending the `Enter` key. If a pin code secures the device, the current iteration of our attack cannot perform the wifi credential extraction and MDM profile enrollment attack. Section 7.3 further addresses this limitation. Succeeding the unlock procedure, the Raspberry Pi Pico requests the current WiFi credentials via the `lib_iap` library. For security reasons, the user must confirm a credential-sharing prompt before the Apple device returns the WiFi credentials in plaintext via the iAP protocol. However, the Raspberry Pi Pico can bypass this prompt by emulating the user via the previously requested HID handlers. Using input emulation (mouse) allows it to confirm the prompt and gain access to the WiFi credentials on the iAP protocol layer. Leveraging the same HID handlers, the Raspberry Pi Pico can also download and activate a MDM profile from the internet. For the current release, the Raspberry Pi Pico must know the precise position of the Safari and Settings app icons within the App launcher. However, this could be optimized in further work by starting the apps via the iAP protocol. Knowing the app's position, TuWire can open Safari and download the MDM profile. After the download, it closes the browser by emulating the `Home` button and opens the Settings app. This app allows the Raspberry Pi Pico to activate the MDM profile. Finally, the Raspberry Pi Pico turns the Apple device into USB host mode via the `lib_iap` library and hands control over to the STM32 board. Once the Apple device runs as the host, it detects the exposed Nero USB interface on the STM32 board and initiates the handshake. The Nero module on the STM32 chip also detects the Apple device, completes the handshake, and continuously requests the device screen content, as explained in Section 5.3.3. Depending on the chosen configuration at compile time, the Nero module consolidates the received frames in a single file either stored on the onboard flash memory leveraging the LittleFS file system or on an external SD card connected via SPI and utilizing the FAT file system. After receiving frames that occupy no more than a threshold, selected at compile time, the Nero module stops requesting new frames and closes the file. This concludes the attack phase. Finally, the attacker can view the recorded screen content on a host computer by connecting the storage device and mounting the partition. Mounting the flash based LittleFS partition requires connecting the STM32 USB port with the host computer and executing the command shown in Listing 5.3. Accessing the recording on the SD card is less complex because modern operating systems support FAT out-of-the-box. Therefore, inserting the SD card into the host computer is sufficient. Regardless of the chosen storage device option, the finally mounted file system contains a single recording file named `IDEV_REC.264`.

Listing 5.3: Example of mounting the LittelFS partition of the exposed USB Mass Storage Device and viewing the content of the H.264 file.

```
# mount LittleFS partition with block device name /dev/sda
$ sudo lfs \
       --read_size=16 \
       --prog_size=16 \
       --block_size=4096 \
       --block_count=3840 \
       --cache_size=64 \
       --lookahead_size=32 \
       /dev/sdc /mnt/lfs

# list mounted directory
$ sudo ls -al /mnt/lfs
total 2.3M
-rwxr-xr-x 1 root     root     2.3M Apr 18 18:57 IDEV_REC.264
```

Chapter 6.

Mitigation Implementation

This chapter explains the design and functionality of the Lightning Condom, a novel Lightning-to-Lightning adapter that facilitates secure charging of an Apple device via untrusted public charging kiosks. After placing it between the Apple device and a malicious charging cable, it secures the charging process by forcing the malicious Lightning accessory to act as a charging-only cable. This behavior is achieved by proxying and filtering the IDBUS handshake and terminating the data groups of the Lightning connector accordingly. The Lightning Condom was entirely designed and implemented during this thesis.

The remaining chapter is structured as follows: Section 6.1 introduces the system architecture, and Section 6.2 illustrates its functionality.

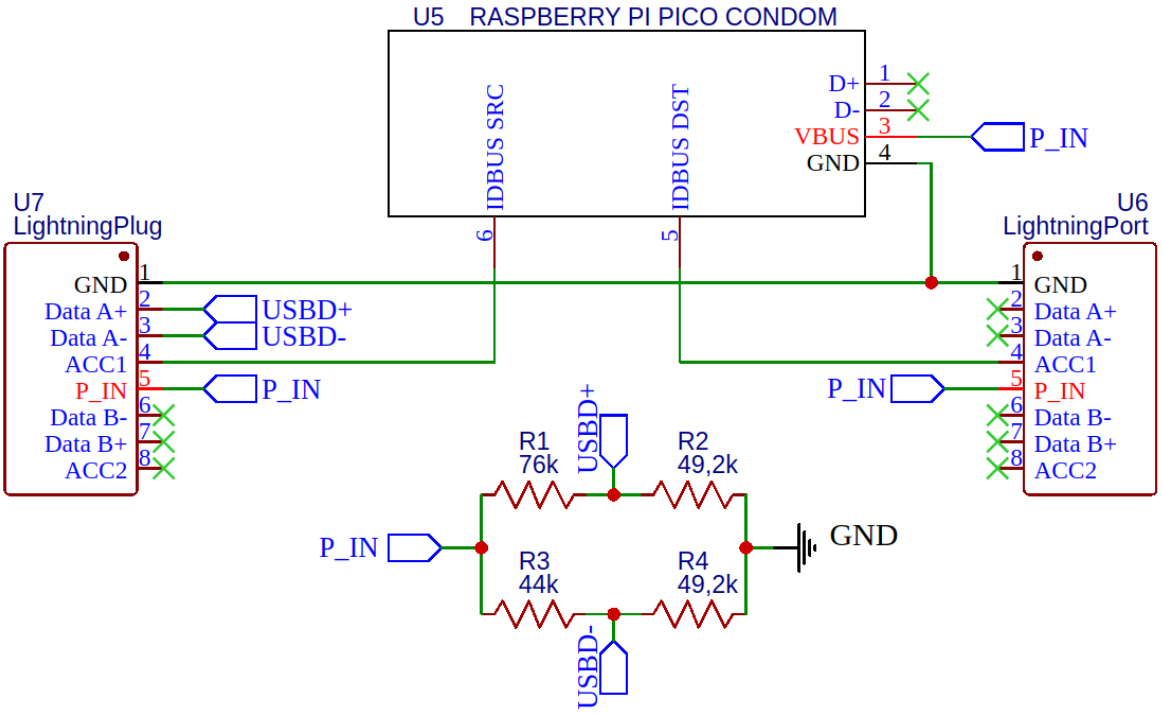


Figure 6.1.: Schematic of the Lightning condom to mitigate Juice Jacking attacks.

6.1. System Architecture

Figure 6.1 illustrates the schematic of the condom and shows that a Raspberry Pi Pico, further introduced in Section 5.2.1, is responsible for securing the IDBUS handshake. The embedded microcontroller and the Apple device receive power from the potentially malicious

charging cable via the shared Lightning port, labeled as `U6`. This architecture implies that connecting the malicious charging cable first, followed by the Apple device, is necessary. In order for the microcontroller to secure the IDBUS handshake, it has connections with the ACC1 pin of the Lightning plug and port. This pin was selected because the employed Lightning-to-USB cable during research consistently used the ACC1 pin regardless of its orientation. If other Lightning accessories exclusively operate through the ACC2 pin, the Lightning Condom must also establish a connection with this pin. To reduce the attack vector, the Lightning Condom does not maintain any other connections between the plug and port. As a result, even if the Apple device exposes high-level protocols, like USB, on data group A or B, the condom terminates them internally, implying that the malicious charging cable does not gain access to it. In more detail, the condom forces the IDBUS handshake to utilize a specific digital ID, representing either a charging-only or Lightning-to-USB cable. The Apple device detects the connected cable type by sampling the data group A pins. If these pins are terminated by a resistor network, as specified in their official Accessory Design Guideline[1], the Apple device classifies the connected Lightning accessory as a charging-only cable and simultaneously deducts the maximal charging current it is allowed to draw. Based on their official Accessory Design Guideline, the implemented resistor network, used by the Lightning Condom, provides a maximum charging current of 1000 mA. The pins of the other data group (B) stay unallocated and can be left floating.

6.2. Mitigation Approach

This section presents how the Lightning Condom defeats potential Juice Jacking attacks and provides a secure charging interface to the Apple device. Security is accomplished on two layers. First, as stated in the previous section, only the essential pins are connected between the plug and the port. Second, based on an allowlist approach, the condom filters the digital ID exchanged between the potentially malicious charging cable and the Apple device during the IDBUS handshake. This filtering step is crucial to mitigate all kinds of Juice Jacking attacks because specific digital IDs allow multiplexing of high-level protocols to the Lightning IDBUS pins. For example, the digital ID {`0x75,0xA0,0x00,0x00,0x00,0x00.0x00,0x40`} requests to expose the Serial Wire Debug (SWD) interface to the Lightning accessory through the IDBUS pins, as stated by Thomas Roth in his work Tamarin Cable [sta22]. Furthermore, filtering the IDs based on a predefined allowlist eliminates the need to identify all critical digital IDs, requiring analyzing all available Lightning accessories or using Apple's internal diagnostic firmware called `diags`. Booting `diags` requires recompiling or patching the iBoot source code and is not covered by this thesis due to its complexity. Nevertheless, security researchers, such as NyanSatan, have successfully booted `diags`[2].

To secure the IDBUS handshake, the embedded Raspberry Pi Pico leverages the high-level API of the `lib_idbus` library, introduced in Section 5.3.1. After establishing a connection with the malicious charging cable and the Apple device, the Raspberry Pi Pico handles the IDBUS handshake. To serve the IDBUS request from the Apple device, the Pico requests the IDBUS response internally from the malicious charging cable. As a result, the Pico does not need to hardcode any IDBUS related information because it acts as a proxy between the Apple device and the charging cable. Furthermore, proxying the IDBUS messages allows

[1] `https://web.archive.org/web/20230404002944/https://developer.apple.com/accessories/Accessory-Design-Guidelines.pdf`

[2] `https://web.archive.org/web/20221228120427/https://nyansatan.github.io/boot-diags/`

the condom to mimic the Lightning charging cable as good as possible. Consequently, the microcontroller performs two IDBUS handshakes. The first occurs between the Apple device's Tristar chip and the Raspberry Pi Pico emulating an HiFive chip, and the second occurs between the Raspberry Pi Pico emulating a Tristar chip and the malicious charging cable's HiFive chip. Figure 6.2 shows the message flow of the two IDBUS handshakes, whereby the left column represents the communication with the Tristar chip of the Apple device, and the right column shows the interaction with the HiFive chip of the malicious charging cable. Once the condom receives a new IDBUS request from Tristar, it performs a cache lookup to validate if it needs to request the response from HiFive. The condom implements a simple key-value cache instance, which aims to allow responding as fast as possible to subsequent attempts of the same request. The cache uses the conjunction of the header and checksum value as the key. This cache is essential to handle the first IDBUS request (`GetDigitalID`) because it maintains a tighter timeout (2 ms) than the other requests. Because proxying and filtering the request consumes more time than delivering a predefined response, the condom may fail to react in time for this request. This delayed response led to a timeout error on Tristar's side. In order to prevent process starvation, the condom incorporates the previously mentioned single-key cache, allowing it to respond in time for subsequent attempts. If the current received request is not in the cache, the condom requests the response from HiFive. As stated in Section 6.1, the notification message group of the IDBUS protocol does not require sending a response. Instead of hardcoding which messages require a response or not, the condom exploits the return value of the HiFive chip. If it receives a response, it updates the cache, secures the IDBUS message, and returns the secured version to Tristar. If HiFive does not respond within a predefined timeout value, the condom concludes that no response is needed and switches back into listening mode.

The received IDBUS messages are secured via policies based on the received HiFive response header value. The current release defines a single policy, which enforces the payload of the `RetDigitalID` command (header value `0x75`) to be always {`0x10, 0x0c, 0x00, 0x00, 0x00, 0x00`}, representing a Lightning-to-USB cable. In conjunction with a proper resistor network connected to data group A, it ensures that the Apple device classifies the Lightning Condom as a charging-only cable.

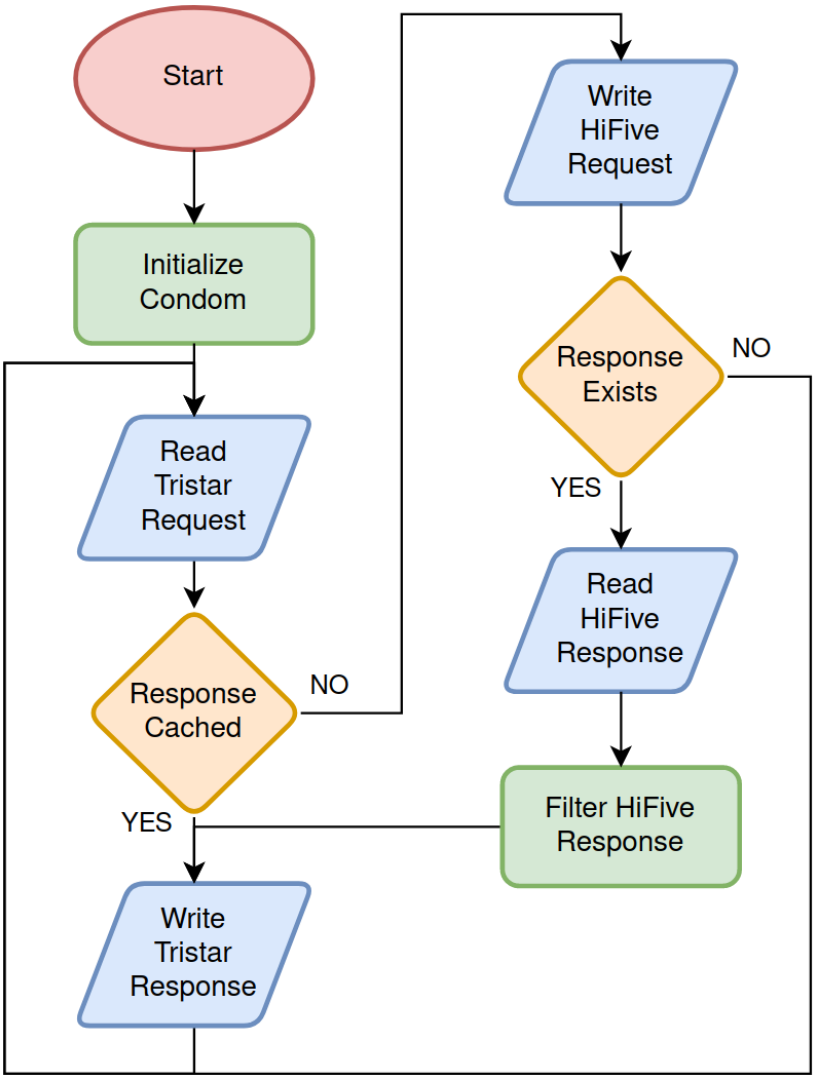


Figure 6.2.: Flowchart of how the Lightning Condom secures the IDBUS handshake. The left column shows the communication with the Apple device's Tristar chip, and the right column the message exchange with the malicious charging cable's HiFive.

Chapter 7.

Evaluation

This chapter evaluates at first the functionality of TuWire and the Lightning Condom, presented in Chapter 5 and Chapter 6, respectively. Afterward, in Section 7.3, it discusses the limitations of this work. Finally, it suggests future works, which can be built upon this work, in Section 7.4.

7.1. TuWire

This section evaluates the state-of-the-art Juice Jacking attack, called TuWire, which is designed and implemented during this thesis from scratch. The evaluation encompasses three aspects: i) the performance of the IDBUS, iAP, and Nero protocol implementations, ii) the assessment of power delivery to the Apple device, and iii) the examination of mounting a Juice Jacking attack.

7.1.1. Experimental Setup

The experimental setup contains an Apple iPhone 7 operating iOS 15.7 and TuWire. We selected the iPhone 7 because it was the newest device available for this work. Furthermore, the current release of TuWire can only perform the WiFi credential extraction and MDM profile enrollment attack in conjunction with this specific target device. These two attacks rely on HID input injection, and the current iteration does not track the mouse movements in real time. As a result, the malicious charging cable must know the exact mouse movements in advance to accomplish these tasks. Consequently, these attacks may fail for different Apple mobile devices because the essential apps may be located elsewhere in the app launcher. Furthermore, these attacks require that the Apple device is locked and can be unlocked without any authentication method, such as a pin code. Compared to the target device, TuWire consists of multiple components, which encompasses two microcontrollers, the secure coprocessor, the Lightning plug, and a transistor circuit, as stated in Section 5.1. In addition, the Nero module supports storing the received frames either on raw flash memory or on an SD card, as explained in Section 5.3.3. Therefore, the experimental setup also employs the onboard flash memory (MX25L51245G) with a capacity of 512 Mbits (64 MB) and an external 32 GB SanDisk Ultra SDHC SD card connected via SPI, running at a frequency of 50 MHz. Since our approach to evaluating the Nero module only requires an active streaming duration of 10 seconds, we limited the amount of retrieved data such that the Nero module stops requesting frames after approximately this amount of time. The final selected data thresholds that lead to sufficiently long, but not excessively long, streaming times are 2 MB for the flash memory and 20 MB for the SD card. Both microcontrollers receive power from the same host computer via the debug USB interface and provide in-exchange system logs.

The power to charge the connected Apple device is sourced from the +5 V pin of the STM32 board. Figure 7.1 shows the final experimental setup without the host computer.

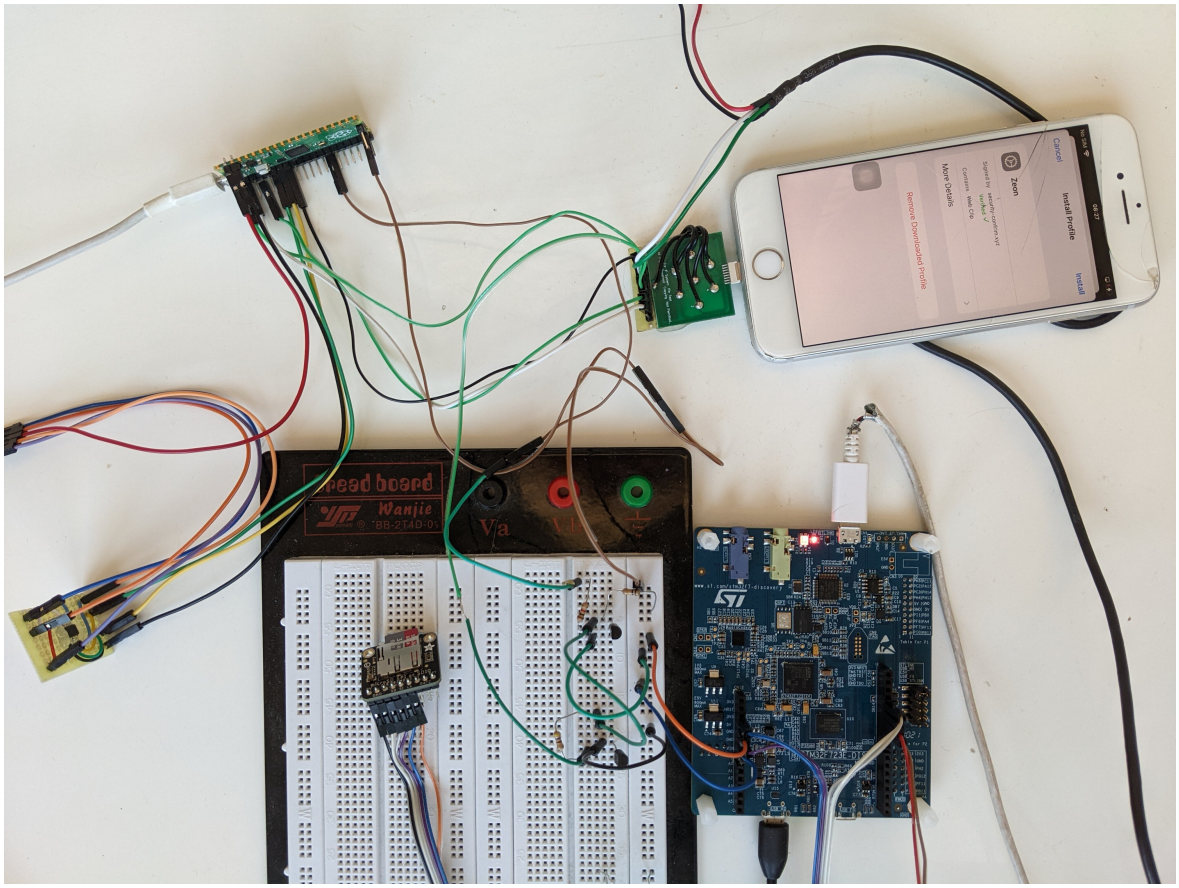


Figure 7.1.: Experimental setup of TuWire.

7.1.2. System Evaluation

This section presents the evaluation results from TuWire. After preparing the experimental setup, as explained in the previous section, we infiltrated the test device and captured the various handshakes. We used a logic analyzer to capture the IDBUS and iAP and a jailbroken test device to capture the Nero handshake. The presented results are derived from these captured handshakes.

IDBUS Stack

To derive a benchmark for the IDBUS stack of TuWire, we compare it against an official Lightning-to-USB cable. The evaluation compares the link quality, focusing on average response times, the overall handshake duration, and the frequency of required retransmissions. The response time refers to the elapsed time between the last byte of Tristar's request and the first byte of HiFive's response. The overall handshake time represents the total elapsed time when the IDBUS communication partners actively used the bus. Thus, it is the sum of the duration of all individual IDBUS transactions, where a single transaction represents the elapsed time between the first byte sent by Tristar and the last byte sent by HiFive. This

approach ensures that the outcome is not influenced by bus idle times. In addition, to ensure a consistent evaluation of the response time and overall handshake duration across all the various HiFive implementations, this evaluation considers an IDBUS handshake abstraction, which contains each command only once, as shown in Figure 4.5. Please note that this abstraction may overlook charging updates if issued multiple times by Tristar during the power handshake, as explained in Section 4.2.6. Retransmission frequency refers to how often Tristar must resend the same request because HiFive cannot respond in time. Moreover, it is important to differentiate whether the TuWire's logging system is enabled. This is especially relevant for the employed microcontroller, Raspberry Pi Pico, which transfers log messages synchronously, implying that every call to the log system injects delays. Therefore, this evaluation performs two iterations, one with and one without the logging system enabled. Table 7.1 summarizes the results of the evaluation.

IDBUS Communication Partner	Avg. Response Time (us)	Overall Time (us)	Avg. Retry Count
Lighting-to-USB	22.82	10188	0
TuWire with logging	507.38	13199	0
TuWire without logging	44.12	10416	0

Table 7.1.: Comparison of the average response time, overall handshake time, and retry count between the TuWire's IDBUS stack and an official Lightning-to-USB cable.

In the first column, the experiment states that the average response time of the implemented IDBUS stack is more than 22 times slower than the official Lightning-to-USB cable if the logging system is enabled. However, turning off the logging system reduces the overhead significantly, resulting in an average response time that is only twice as long as the official Lightning-to-USB cable. This noticeable delay is caused by the currently used IO layer implementation of the IDBUS library (`lib_idbus`), which must replace the loaded PIO code to switch from the reader into the writer mode, as explained in Section 5.3.1. The second column of Table 7.1 illustrates that the overall handshake time remains almost the same if TuWire's IDBUS stack is leveraged without the logging system, even though the average response time doubles. This observation can be explained by the fact that sending the bytes accounts for the majority of the overall handshake time. Specifically, the previously proposed abstracted IDBUS handshake exchanges 108 bytes, where a single byte transfer via the IDBUS protocol consumes, on average, 94 μs, resulting in a minimum exchange time of 10152 μs. Compared to the 108 bytes, the response time influences the overall handshake time only by a factor of six (amount of request-response transactions), justifying the observation. If logging is enabled, the overall handshake time increases by 29.55 %, and if disabled, only by 2.37 %. Neither of the two variants requires retransmissions, even when the logging system is enabled. As mentioned in Section 4.2.5, the IDBUS protocol maintains various timeout intervals, whereby the tightest one is 2 ms. Both variants can satisfy this timeout interval, implying zero retransmission count. As this experiment shows, if the logging system is disabled, the implemented IDBUS stack performs the handshake with nearly the same performance as the official Lightning-to-USB cable. However, the stack allows emulating arbitrary HiFive and Tristar instances compared to the official Apple cable.

iAP Stack

Because the UART iAP stack, used by TuWire, was implemented based on the publically leaked standard, and the Haywire adapter performs the handshake via USB, we cannot compare its performance against an official Apple product. Therefore, this evaluation presents the final runtime durations for completing the authentication process, enabling charging, registering HID handlers, extracting the WiFi credential, enrolling the MDM profile on the target device, and moving the test device into USB host mode. Table 7.2 shows the final runtime of each previously mentioned task, observed dynamically during code execution through the provided timing system from the Raspberry Pi Pico's SDK. Each runtime evaluation starts with the first iAP request and ends with the last iAP response received from the Apple device required to perform the current task.

iAP Task	Overall Duration (s)
iAP authentication	3.51
Enabling charging	0.20
HID handlers registration	1.70
WiFi credential extraction	5.02
MDM profile enrollment	21.05
USB host mode activation	0.03

Table 7.2.: Runtime of all performed iAP tasks by TuWire.

Table 7.2 indicates that the various iAP task's runtime is relatively long. This long runtime results from the main emphasis of this thesis, which focuses on feasibility rather than performance. Thus, the code contains a lot of system logging and sleep phases to enhance traceability. For example, during the authentication process, the microcontroller loads the certificate data iteratively, in 128-byte pages, on-demand from the secure coprocessor, even though the certificate data is static. This loading process consumes 2.6 seconds out of 3.51 seconds from the entire authentication process, which could be eliminated by pre-storing the certificate data on the microcontroller. Also, the HID handler registration runtime contains a raw wait time of 1.5 seconds, distributed across three HID handler acquisitions evenly. Likewise, the stack waits 10 ms after each HID event injection, significantly blowing up the WiFi credential extraction and MDM profile enrollment task's runtime. Consequently, optimizing or removing these wait phases allows easy optimization of these tasks. However, as previously stated, this thesis focuses on feasibility rather than optimization. In summary, the iAP stack works as expected and supports TuWire to accomplish its tasks, at the expense of performance.

Nero Stack

After evaluating the IDBUS and iAP stack, this subsection focuses on TuWire's Nero module. Analogous to the IDBUS evaluation, we compare the performance of the implemented Nero module against Apple's official Haywire adapter. The comparison encompasses the handshake duration and the average frame throughput.

As explained in Section 5.3.3, the implemented module supports two modes of operation. Either it requests and drops the frames or requests and stores them on the configured storage device (flash memory or SD card). Unlike our Nero implementation, the official

Haywire adapter does not store the frames and only forwards them to the HDMI connector. Nevertheless, to get an accurate comparison, we evaluate the module's performance with enabled and disabled storage functionality, including both storage backends. In addition, comparing the different combinations of modes and storage devices with each other allows for identifying potential bottlenecks of the current implementation. The overall handshake duration is determined by measuring the elapsed time starting from the first PING message and ending with the arrival of the first ASYN[FEED] message. To calculate the frame throughput, we sample the total data amount of the received frames within 10 seconds and divide it by the sampling time, yielding the average throughput in bytes per second. As stated in Section 4.5.3, the number of received frame data is coupled with the change in screen content. Therefore, maintaining the same screen activity among all three test cases is vital. For this evaluation, we manually switched between the same two home pages repeatedly with a constant frequency.

Both metrics are extracted from the raw USB data traffic, captured once per objective on the target device, and analyzed via Wireshark[1]. To gain access to the raw USB data traffic, we substitute the previously introduced test device with an iPhone 6 running iOS 12.5.7. The rationale behind this device substitution is that the iPhone 6 can be jailbroken and allows capturing the raw USB packets, as explained in Section 4.5.

Nero Implementation	Overall Handshake Time (ms)	Frames Throughput (kB/s)
Haywire	738.72	1680.67
TuWire without storing	136.72	1454.37
TuWire with flash memory	148.08	105.71
TuWire with SD card	159.87	387.74

Table 7.3.: Handshake duration and frames throughput of the Haywire adapter and TuWire.

As the Table 7.3 shows, the Nero implementation of this thesis performs the handshake significantly faster than Haywire. The runtime increases by four to five times depending on the active storage backend. The observable runtime difference is caused by an embedded iAP handshake, which Haywire performs as part of the Nero handshake. Haywire performs this iAP handshake to notify the Apple device of its current battery charging capability, depending on whether it is connected to an external power source. In the case of TuWire, the necessity of this iAP handshake during the Nero handshake drops because the Raspberry Pi Pico already shares this information with the Apple device after performing the iAP authentication processes. Furthermore, the evaluation shows that TuWire's Nero handshake takes slightly longer if storing the frames is enabled. The reason is that the module receives and stores the SPS and PPS information during the handshake. Based on the raw USB captures, the SPS and PPS processing time also causes the timing difference between the two storage options. Unfortunately, due to the huge abstraction layer provided by Zephyr regarding file systems, it is challenging to identify the root cause of this timing difference. However, a reasonable explanation would be that the LittleFS file system maintains a per-file cache and whether the current write triggers a page erase.

The second column indicates that the implemented Nero driver is nearly as fast as the original Haywire adapter if storing the frame content is disabled. It maintains a throughput of 86.53 %, compared to Haywire. Unfortunately, the throughput drops significantly after enabling storing the frame content independently of the activated storage backend. The

[1] `https://www.wireshark.org/`

evaluation shows that after enabling the flash based device storage, the throughput drops to 6.2 %. The selected flash memory causes this deterioration because erasing and writing a single page, 4096 bytes, consumes, on average, 35 ms. Consequently, this configuration of the Nero driver needs approximately 17 seconds to store 2 MB of frame data. The throughput can be raised to 23.07 % by activating the SD card based storage backend. This is still slow compared to the official Haywire, but it increases the frame throughput by more than three times compared to the flash memory option. More precisely, this configuration requires, on average, 10 ms to store a single data page (4096 bytes).

Initially, the plan was to utilize an SD card connected via the SDIO interface as the storage device. Unfortunately, the chosen board, STM32F723-Disco, does not provide access to the SDIO interface of the embedded STM32F723IE chip. Therefore, we switched during the thesis to use the available QSPI flash and an SD card via SPI as an alternative storage device, which turned out to be a suboptimal decision.

Power Delivery

After evaluating Apple's proprietary protocol implementations, this section focuses on the power delivery functionality of TuWire. To Juice Jack a device without arousing suspicion, TuWire must use the operating system to inform the user about the newly initiated charging process after establishing the tethered connection. Depending on the digital ID embedded into the `RetDigitalID` IDBUS message, the Apple device starts drawing current and charging its internal battery after completing the IDBUS handshake solely or requires an additional power handshake and activation via the iAP protocol, as explained in Section 4.2.6.

TuWire utilized a digital ID, which requires the additional power handshake and activation via the iAP protocol, as explained in Section 4.3.6. After completing all handshakes, the test device notifies the user about charging and draws, on average, 200 mA. Unfortunately, the empirically observed charging current is relatively low compared to other charging utilities. However, it is acceptable because charging is not its primary objective. In addition, optimizing the transistor circuit and leveraging a different power source than the STM32 board may increase the charging current.

Juice Jacking Attack

Finally, this subsection evaluates the entire process of how TuWire infiltrates an Apple device. Section 5.4 explains this process theoretically, and this section enriches this explanation with empirically observed timing values. The entire evaluation process is available on YouTube[2].

Once a connection exists, TuWire starts registering as a legitimate Lightning accessory against the Apple device via the IDBUS and iAP handshake. As evaluated in Section 7.1.2 and Section 7.1.2, the IDBUS and iAP handshake completes in approximately 3.64 seconds. Next, it enables charging via the iAP protocol. Consequently, the operating system (iOS) informs the system user about the charging process after approximately 3.84 seconds. Immediately after enabling charging, it unlocks the phone, allocates the HID handlers, and steals the WiFi credentials to which the Apple device is currently connected. As elaborated in Section 7.1.2, allocating and stealing the credentials takes approximately 6.72 seconds. As the next step, it opens the Safari browser, downloads the Zeon MDM profile from the internet, and aims to enroll it. The entire process consumes around 21.05 seconds. As a result, both attacks, leveraging HID input injection, are completed within 31.41 seconds. Finally, TuWire turns

[2] `https://youtu.be/imAbaG3W6TI`

the Apple device into USB host mode, which triggers the Nero implementation to capture the device screen content. As Section 7.1.2 already examined, the currently used storage device provides a low writing speed. Therefore, to limit the recording duration, the configurable video size threshold, offered by the Nero module, is set to 2 MB. This threshold results in a recording duration of approximately 17 seconds.

The experiment shows that once a tethered connection exists, TuWire's can successfully infiltrate the test device within approximately 48.41 seconds. Within this duration, the malicious charging cable can steal the WiFi credentials to which the Apple device is currently connected to, enroll a custom MDM profile, and stream the device screen content to a flash storage device. Thus, an attacker can infiltrate Apple devices by arming a public charging kiosk with TuWire, consisting entirely of low-budget microcontrollers.

7.2. Lightning Condom

This section reports the evaluation steps performed against the novel mitigation contribution of this work, called Lightning Condom. The evaluation focuses on the three objectives: i) performance, ii) power delivery, and iii) mitigation of USB based Juice Jacking attacks.

7.2.1. Experimental Setup

The experimental setup to prove the previously stated points is leveraging an official Lightning-to-USB cable instead of TuWire, even though the latter is the main objective to mitigate in this thesis. However, we select the Lightning-to-USB cable because of its simplicity in setup, and from the condom's perspective, it is inconsequential if it mitigates the cable or TuWire. This statement relies on the fact that the condom physically cuts all connections between the plug and the port except one IDBUS, the voltage, and the ground line, as explained in Section 6.1. Therefore, the condom terminates, by design, the exposed protocols on data groups A and B and the second IDBUS pin. Additionally, the remaining IDBUS pin serves exclusively as the communication channel for the IDBUS handshake. The Lightning Condom enforces this behavior by filtering the IDBUS handshake and permitting only digital IDs that utilize this pin solely for the handshake process. Besides this theoretical proof of equality, from the condom's perspective, we also conducted an empirical test to confirm this. A comprehensive demonstration video of mitigating the Lightning-to-USB and TuWire is available on YouTube[3].

The final experimental setup consists of a Lightning Condom, a Lightning-to-USB cable, and an iPhone, as shown in Figure 7.2. As explained in Section 6.1, the condom consists of a Raspberry Pi Pico, operating the condom firmware, a resistor network, and the Lightning connectors. The microcontroller maintains a USB connection to a host computer to receive power and provide system logs. Likewise, the Lightning-to-USB cable is also connected to the same host computer to validate later whether the condom effectively safeguards the USB interface of the Apple device. For testing, an iPhone 7 running iOS 15.7 was selected because it was the newest device available for this work.

7.2.2. System Evaluation

To evaluate the previously stated objectives, the iPhone and Lightning-to-USB cable must be connected as stated in Section 6.1.

[3] `https://youtu.be/olzWqD0d64o`

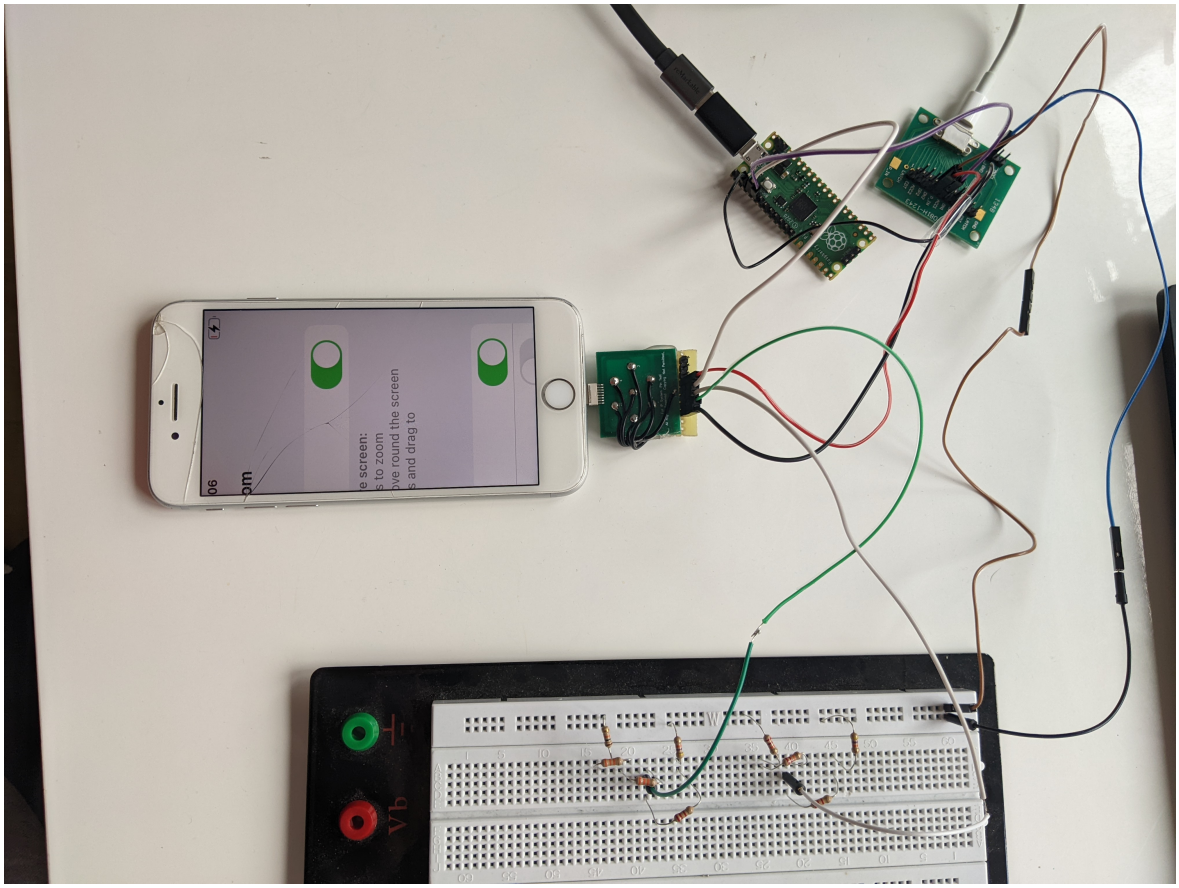


Figure 7.2.: Experimental setup of the Lightning Condom.

Once this connection exists, the Apple device initiates the IDBUS handshake, which is handled by the condom as stated in Section 6.2. Because the condom proxies and secures the IDBUS messages, it introduces a runtime overhead, leading to performance deterioration. To quantify this performance deterioration, we analyzed the runtime of the various IDBUS transactions. An IDBUS transaction starts with the first byte of the request sent by the Apple device and ends with the last byte of the response sent to the Apple device. We extracted these values from previously performed IDBUS handshakes, which have been captured via a logic analyzer. Table 7.4 shows the introduced overhead by comparing the IDBUS transaction's runtime differences between using and not using the Lightning Condom. The second column states the transaction times achieved by securing the handshake via the Lightning Condom. The third column shows the runtime without using the condom, and the fourth column exhibits the overhead when the condom is used. The last row, containing the overall handshake time, reveals that the condom introduces an overhead of more than four times the original duration. Especially, the first transaction consumes significantly more time because this message maintains a shorter timeout value, and the condom fails to answer within the first iteration. Consequently, the condom must wait until Tristar repolls the same IDBUS line, with an interval of 2 ms. Once Tristar repolls it, the condom acknowledges the request with the prior retrieved and cached response. In addition, compared to the insecure handshake, the Lightning Condom performs two IDBUS handshakes to secure the charging process. The first occurs between the Apple device's Tristar chip and the condom's emulated HiFive, and

the second occurs between the condom's emulated Tristar and the Lighting-to-USB cable's HiFive. Additionally, the logging system and IDBUS command length impact the runtime, which explains the variations among the different IDBUS transactions within the overhead column.

IDBUS Transaction	Secure (us)	Insecure (us)	Overhead
Digital ID	9025	1140	7.91
Charging State	3398	589	5.76
Interface Details	5033	1314	3.83
Interface Module Number	7394	2230	3.31
Accessory Serial Number	7350	2230	3.29
Lightning Device State	3629	767	4.73
Apple System Info Notification	3606	953	3.78
Overall	39435	9223	4.27

Table 7.4.: Message types and associated commands of the IDBUS protocol.

Next, we evaluate if the condom influences power delivery. More precisely, this experiment examines how much current the test device draws during charging with and without using the condom. In both situations, the battery level of the test device is fixed at 70 %. To ensure a stationary charging source, the experimental setup for this test was slightly modified by connecting the Lightning to USB cable to an official Apple 1000 mA power adapter instead of the host computer. If charging is established with the condom, the test device draws on average 400 mA. Compared to that, if charged without the condom, the test device draws on average 600 mA, implying that the condom reduces the power delivery by 20 %, relatively to the charger capacity of 1000 mA. However, the implemented resistor network within the condom uses approximated values due to availability issues, which may cause the observed deviation.

Finally, we evaluate if the condom successfully mitigates USB based Juice Jacking attacks. If connected via the condom, both the test device and the host computer fail to recognize each other's presence. The system log of the host computer does not show any newly connected device, and the test device never raises the host computer authentication prompt. Conversely, if connected without the condom, the host computer identifies the test device, and the user receives the authentication prompt.

As this evaluation shows, using the Lightning Condom decreases the performance but ensures that USB based Juice Jacking attacks are no longer possible. Furthermore, both handshake differences are within the range of milliseconds, which is not noticeable to humans.

7.3. Discussion

The experiments above prove the functionality of TuWire and the Lightning Condom. However, the following limitations of the proposed method should be addressed.

iOS Version

This work evaluated TuWire and the Lightning Condom against an iPhone 7, operating iOS 15.7. This device was selected because no other device offering a higher iOS version was

available during the thesis. However, the currently newest iOS version is iOS 16, raising the question if the latest iOS is also compatible with the two contributions of this work.

iAP2 Version

This work implements the outdated iAP1 standard over the UART protocol to authenticate, enable charging, inject HID events, and switch the Apple device into USB host mode. It would be advisable to replace iAP1 with iAP2, the successor, to accomplish the same tasks. Furthermore, substituting UART with a more modern transport layer, such as USB or Bluetooth, and upgrading to the latest 2.0C version of the secure coprocessor complements the above recommendations. These upgrades make the proposed attack contribution more resilient to upcoming improvements.

TuWire Stealthiness

The work supplies a working proof-of-concept implementation of the malicious TuWire charging cable. However, the stealthiness of the final design, encompassing two microcontrollers, is questionable. The size is neglectable if applied in a public charging kiosk attack scenario, where only the Lightning plug is visible to the victim. Nevertheless, this is untrue if TuWire should be embedded into a Lightning-to-USB cable. In this particular case, the proposed design exceeds the available space. To address the space limitation, TuWire could be exclusively enabled by a custom, designed board solely equipped with the STM32 chip. The chip can perform the IDBUS via the GPIO pins, and the thesis already provides an iAP implementation for the UART and USB transport layer. The only remaining challenge is communication with the secure coprocessor, which requires adjusting the Zephyr kernel to slower I2C transmission speeds.

TuWire Storage Device

As stated in Section 5.3.3, the current Nero module implementation offers support for storing the requested frames either on the onboard QSPI flash memory (MX25L51245G) or on an external SD card connected via the SPI protocol. Unfortunately, the write speed of both options is insufficiently fast enough to catch up with the number of frames we receive from the Apple device, resulting in a jumpy recording. Section 7.1.2 illustrates the problem in more detail. In order to enhance the recording quality, it would be advisable to use a NAND based flash or to connect the SD card via the SDIO interface.

TuWire Bypassing Pin Codes

The current release of TuWire can perform the WiFi access point credential and MDM enrollment attack only against unlocked or locked devices, which can be unlocked without a pin code. Implying that the current version can not perform these attacks against locked devices, which are secured by a pin code, because it maintains a consistent attack procedure without checking the current device state. Adding a validation logic that delays the attack phase until the user manually unlocks the phone would expand TuWire's capability to include devices secured via a pin code.

7.4. Future Work

IDBUS and iAP protocol fuzzing

The thesis studied and consolidated the fundamentals of the IDBUS and iAP protocols. As a result, it provides almost complete documentation of the semantics and effects of the seen IDBUS messages, alongside a basic introduction to iAP. Because the iAP standard was already publically leaked, further reverse engineering this protocol is unnecessary. Additionally, the work provides software libraries, which facilitate establishing a communication channel with an Apple device via the previously mentioned protocols. These contributions may be used as a starting point for fuzzing the two protocols to reveal new attack vectors.

TuWire Feed Streaming

The current TuWire implementation only allows storing the requested video frames to a local storage device. Consequently, the devices must possess enough storage to hold the entire video data, whether valuable or not. Additionally, an attacker must swap the storage devices repeatedly to avoid running out of storage. This situation could be enhanced by streaming the received data directly to a more powerful remote server instead of storing the frames locally. Besides the storage problem, this would enable near-real-time video feed analysis on the remote server to extract and store only valuable information.

Expand Device Comparability

The recent iteration of TuWire is tailored to attack an iPhone 7, which can be unlocked without any authentication method, and where the location of the Safari and Settings app is precisely known within the app launcher. If these preconditions are not fulfilled, the MDM profile enrollment and the WiFi credential extraction attack fail because the mouse movements are hardcoded for this specific model. These preconditions can be eliminated by using the Nero module and extracting the position of the needed apps from the received video feed. Additionally, this enables support for arbitrary Apple devices and allows mounting dynamic HID attacks against the device depending on whether an app is installed.

USB-C Connector

The European Union (EU) announced in 2022 a new regulation that by the end of 2024, all new smartphones and other mobile devices must leverage a USB-C connector[4]. Due to this regulation, Apple can no longer sell its mobile devices with a Lightning connector in countries belonging to the EU. As a result, once the new revisions of Apple mobile devices without a Lightning connector are released, further research is required to investigate if the concept of TuWire can be applied to them.

[4] https://web.archive.org/web/20230423181848/https://www.europarl.europa.eu/news/en/press-room/20220930IPR41928/long-awaited-common-charger-for-mobile-devices-will-be-a-reality-in-2024

Chapter 8.

Conclusion

Over the past decade, the significance of smartphones has increased tremendously, leading to a remarkable expansion of their field of applications. This increased significance made smartphones attractive targets for attackers because compromising them yields valuable information or may allow the attacker to impersonate the device user. Among the various security threats, an often overlooked approach to extracting sensitive information or installing malware represents Juice Jacking, which uses a tethered connection to infiltrate the device. This thesis emphasizes the relevance of Juice Jacking attacks targeting Apple devices leveraging the Lightning connector for establishing a tethered connection. The major contribution of this work encompasses the design and implementation of a low-budget malicious charging cable called TuWire and a mitigation object called Lightning Condom.

TuWire consists of two microcontrollers and infiltrates connected Apple devices by exploiting the functionality of three proprietary protocols, known as IDBUS, iAP, and Nero. These protocols facilitate TuWire to infiltrate a connected device by extracting the current WiFi access point credentials the device is currently connected to, enrolling a custom MDM profile, and streaming the device screen content to a configured storage device, alongside charging the device's battery. Specifically, the IDBUS protocol allows registering a newly connected Lightning accessory and requesting iAP and USB access. After gaining access to the iAP interface, it completes the authentication process by leveraging an unauthorized secure coprocessor. Completing the authentication process allows TuWire to access the HID input handler, part of the iAP standard, and to switch the Apple device into USB host mode. Once in USB host mode, the Nero protocol allows streaming the device screen content to disk. Complementary to the major contributions, the thesis also provides a dedicated library to handle each of the previously mentioned proprietary protocols. The evaluation of TuWire showed that it could perform the IDBUS handshake nearly as fast as an official Lightning-to-USB cable if the debug logging system is disabled. Due to the lack of a matching Lightning accessory, we evaluated the iAP implementation by itself, which showed that it is relatively slow. The last component, TuWire's Nero module, was evaluated against the official Haywire adapter. This evaluation showed that Haywire requires more time to perform the Nero handshake but provides a better frame throughput than the reimplementation. If the Nero module of this thesis drops the frames, it maintains a throughput of 86.53 % relative to the official Haywire adapter. However, if storing the frames to flash or SD card is enabled, the frame throughput rate drops to 6.20 % and 23.07 %, respectively.

To mitigate the risk of falling victim to Juice Jacking attacks, this thesis also proposes a novel protection solution called Lightning Condom. The condom is a Lighting-to-Lightning adapter, which allows charging the Apple device securely via untrusted public charging kiosks. It ensures a secure charging process by forcing any potential malicious charging cable to act as a charging-only cable by eliminating all unnecessary connections and filtering the IDBUS handshake. However, the evaluation shows that the secure interface comes with costs. The

Lightning Condom introduces a fourfold average IDBUS runtime overhead and reduces the charging current by 20 %.

In summary, this thesis proves the criticality of Juice Jacking attacks because it allows attackers to infiltrate an Apple device with solely low-budget microcontrollers. However, this threat can effectively be mitigated by leveraging the proposed Lightning Condom.

Appendix A.

Haywire USB Interface

As stated in Section 4.4, Haywire, the codename for Apple's Lightning Digital AV adapter, is an ordinary USB peripheral, connectable to any desktop computer, and receives its firmware from the Apple device. In order to emulate the adapter's behavior with an arbitrary USB-capable microcontroller, it is vital to comprehend its exposed USB interface. One way to gain this information was to run the system utility `lsusb` on a desktop computer against a booted Haywire adapter. Unfortunately, extracting this information from an unbooted adapter is infeasible because it receives its firmware from the Apple device, as stated previously. Consequently, before connecting it to the desktop computer, it is essential to establish a connection with an Apple device and let the firmware upload proceed.

To accomplish the USB interface extraction, we have used a breakout board for the Lightning plug and port, a USB cable with connectors on one end, and an iPhone 7. To prepare the adapter's firmware, we emulated a typical Lightning cable by wiring the two breakout boards straight and connected the adapter and iPhone to the port and plug, respectively.

Once the device screen content appears on the TV connected to Haywire's HDMI port, we can remove the `USB+` and `USB-` pins, which occupied the Data Group A in our specific situation. The rest of the pins stay untouched because the adapter must remain powered on by the Apple device. Finally, the desktop computer can connect with the booted haywire via the modified USB cable by joining the `GND` and USB pins. Listing A.1 shows produced USB interface dump of the Haywire adapter.

Listing A.1: USB descriptor information from a booted Haywire adatper generated via lsusb.

```
Bus 001 Device 012: ID 05ac:12ad Apple, Inc. iAccy1
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass            0
  bDeviceSubClass         0
  bDeviceProtocol         0
  bMaxPacketSize0        64
  idVendor           0x05ac Apple, Inc.
  idProduct          0x12ad
  bcdDevice            2.11
  iManufacturer           1 Apple Inc.
  iProduct                2 iAccy1
  iSerial                 3 nomac?123456
  bNumConfigurations      1
  Configuration Descriptor:
    bLength                9
    bDescriptorType        2
    wTotalLength       0x003e
```

```
bNumInterfaces           2
bConfigurationValue      1
iConfiguration           5
bmAttributes          0xc0
  Self Powered
MaxPower               500mA
Interface Descriptor:
  bLength                9
  bDescriptorType        4
  bInterfaceNumber       0
  bAlternateSetting      0
  bNumEndpoints          3
  bInterfaceClass      255 Vendor Specific Class
  bInterfaceSubClass   240
  bInterfaceProtocol     0
  iInterface             6
  Endpoint Descriptor:
    bLength              7
    bDescriptorType      5
    bEndpointAddress  0x02  EP 2 OUT
    bmAttributes         2
      Transfer Type          Bulk
      Synch Type             None
      Usage Type             Data
    wMaxPacketSize    0x0200  1x 512 bytes
    bInterval           10
  Endpoint Descriptor:
    bLength              7
    bDescriptorType      5
    bEndpointAddress  0x81  EP 1 IN
    bmAttributes         2
      Transfer Type          Bulk
      Synch Type             None
      Usage Type             Data
    wMaxPacketSize    0x0200  1x 512 bytes
    bInterval           10
  Endpoint Descriptor:
    bLength              7
    bDescriptorType      5
    bEndpointAddress  0x83  EP 3 IN
    bmAttributes         3
      Transfer Type          Interrupt
      Synch Type             None
      Usage Type             Data
    wMaxPacketSize    0x0040  1x 64 bytes
    bInterval           10
Interface Descriptor:
  bLength                9
  bDescriptorType        4
  bInterfaceNumber       1
  bAlternateSetting      0
  bNumEndpoints          2
  bInterfaceClass      255 Vendor Specific Class
  bInterfaceSubClass    42
```

```
bInterfaceProtocol     255
iInterface               7
Endpoint Descriptor:
  bLength                7
  bDescriptorType        5
  bEndpointAddress     0x85  EP 5 IN
  bmAttributes           2
    Transfer Type            Bulk
    Synch Type               None
    Usage Type               Data
  wMaxPacketSize     0x0200  1x 512 bytes
  bInterval             10
Endpoint Descriptor:
  bLength                7
  bDescriptorType        5
  bEndpointAddress     0x04  EP 4 OUT
  bmAttributes           2
    Transfer Type            Bulk
    Synch Type               None
    Usage Type               Data
  wMaxPacketSize     0x0200  1x 512 bytes
  bInterval             10
```

Bibliography

[1621] ITU-T Study Group 16. *Advanced video coding for generic audiovisual services*. https://handle.itu.int/11.1002/1000/14659. Accessed: 2023-03-26. Aug. 2021.

[Ami18] Ramtin Amin. *Tristan*. https://web.archive.org/web/20180524101120/http://ramtin-amin.fr:80/tristar.html. Accessed: 2022-09-14. 2018.

[Con19] Lambda Concept. *Graywire Lightning Cable Implant*. https://web.archive.org/web/20230223150856/http://blog.lambdaconcept.com/post/2019-09/graywire/. Accessed: 2022-09-04. 2019.

[Inc15] Apple Inc. *DUAL ORIENTATION CONNECTOR WITH EXTERNAL CONTACTS AND CONDUCTIVE FRAME*. Patent US 8,573.995 B2. Accessed: 2023-01-16. 2015. URL: https://web.archive.org/web/20230317081240/https://patentimages.storage.googleapis.com/2d/8d/88/479b141c173fe5/US8573995.pdf.

[Inc22] Apple Inc. *Apple Platform Security Guide*. https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf. Accessed: 2023-05-26. May 2022.

[Kum20] Yuvraj Kumar. "Juice Jacking - The USB Charger Scam". In: *Available at SSRN 3580209* (Apr. 2020).

[LHK+16] Edwin Lupito Loe, Hsu-Chun Hsiao, Tiffany Hyun-Jin Kim, Shao-Chuan Lee, and Shin-Ming Cheng. "SandUSB: An installation-free sandbox for USB peripherals". In: *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. 2016, pp. 621–626. DOI: 10.1109/WF-IoT.2016.7845512.

[LJS+13] Billy Lau, Yeongjin Jang, Chengyu Song, Tielei Wang, Pak Ho Chung, and Paul Royal. "Mactans: Injecting malware into iOS devices via malicious chargers". In: *Black Hat USA* 92 (2013).

[MLMK15] Weizhi Meng, Wang Hao Lee, SR Murali, and SPT Krishnan. "Charging me and I know your secrets! Towards juice filming attacks on smartphones". In: *Proceedings of the 1st ACM workshop on cyber-physical system security*. 2015, pp. 89–98.

[MLMK16] Weizhi Meng, Wang Hao Lee, SR Murali, and SPT Krishnan. "JuiceCaster: towards automatic juice filming attacks on smartphones". In: *Journal of Network and Computer Applications* 68 (2016), pp. 201–212.

[Sata] Nyan Satan. *Apple Kraken*. https://twitter.com/nyan_satan/status/1543156168855617538?t=JphDkoW8hEDo-TH1YTLpQg&s=19. Accessed: 2023-01-17.

[Satb] Nyan Satan. *Apple Lightning*. https://web.archive.org/web/20230211225113/https://nyansatan.github.io/lightning/. Accessed: 2022-09-14.

[Sem12] NXP Semiconductors. *UM10204: I2C-bus specification and user manual*. https://web.archive.org/web/20121017153327/http://www.nxp.com/documents/user_manual/UM10204.pdf. Accessed: 2023-03-08. Oct. 2012.

[sta22] stacksmashing. "The hitchhacker's guide to iPhone Lightning & JTAG hacking". In: *DEFCON 2022*. 2022.

[STM22] STMicroelectronics. *DS11853: Arm® Cortex® -M7 32b MCU+FPU, 462DMIPS, up to 512KB Flash 256+16+4KB RAM, USB OTG HS/FS, 18 TIMs, 3 ADCs, 21 com IF*. https://web.archive.org/web/20220510082950/https://www.st.com/resource/en/datasheet/stm32f723ie.pdf. Accessed: 2023-03-08. July 2022.

[USB00] Inc. USB Implementers Forum. *The Universal Serial Bus Specification, Rev. 2.0*. https://archive.org/download/usb_20_202303/usb_20.pdf. Accessed: 2023-03-07. Apr. 2000.

[USB01] Inc. USB Implementers Forum. *Device Class Definition for Human Interface Devices (HID)*. https://web.archive.org/web/20220628021053/https://usb.org/sites/default/files/hid1_11.pdf. Accessed: 2023-03-07. June 2001.

[USB12] Inc. USB Implementers Forum. *On-The-Go and Embedded Host Supplement to the USB Revision 2.0 Specification, Revision 2.0 version 1.1a*. https://ia903009.us.archive.org/23/items/usbspec_1.0/usbspec.pdf. Accessed: 2023-03-07. July 2012.

[USB96] Inc. USB Implementers Forum. *The Universal Serial Bus Specification, Rev. 1.0*. https://ia903009.us.archive.org/23/items/usbspec_1.0/usbspec.pdf. Accessed: 2023-03-07. Jan. 1996.

[USB99] Inc. USB Implementers Forum. *Universal Serial Bus Mass Storage Class*. https://web.archive.org/web/20230322032206/https://www.usb.org/sites/default/files/usbmassbulk_10.pdf. Accessed: 2023-04-11. Sept. 1999.

[WGY22] Yuanda Wang, Hanqing Guo, and Qiben Yan. "GhostTalk: Interactive Attack on Smartphone Voice System Through Power Line". In: *Network and Distributed Systems Security (NDSS) Symposium*. 2022.