

IoT Device-Fingerprinting mit dem ESP32 System-on-Chip

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Steven Hysi, BSc

Matrikelnummer 11830164

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.-Prof. Dipl.-Ing. Mag. Dr. techn. Edgar Weippl

Mitwirkung: Univ.Lektor Dipl.-Ing. Michael Pucher

Dipl.-Ing. Christian Kudera

Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik

Wien, 28. Jänner 2026

Steven Hysi

Edgar Weippl

Technische Universität Wien

A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



IoT Device Fingerprinting using ESP32 System-on-Chip

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Steven Hysi, BSc

Registration Number 11830164

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Prof. Dipl.-Ing. Mag. Dr. techn. Edgar Weippl

Assistance: Univ.Lektor Dipl.-Ing. Michael Pucher

Dipl.-Ing. Christian Kudera

Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik

Vienna, January 28, 2026

Steven Hysi

Edgar Weippl



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Steven Hysi, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 28. Jänner 2026

Steven Hysi



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Danksagung

Ein großer Dank gilt Michael Pucher, Georg Merzdovnik und Christian Kudera für die pragmatische, unkomplizierte und stets sehr hilfsbereite Betreuung.

Denselben und ganz besonderen Dank möchte ich meiner Familie, meinen Freunden und meiner Partnerin aussprechen, die mir in allen Lebenslagen ein Anker und große Unterstützung sind.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Contents	ix
1 Kurzfassung	1
2 Abstract	3
3 Introduction	5
4 Background	7
4.1 Networking Fundamentals	7
4.2 802.11 WLAN Standard and MAC Fundamentals	11
4.3 TCP/IP Protocol Stack	14
4.4 Multicast DNS and DNS-Based Service Discovery	21
4.5 ESP32 System-on-Chip	23
5 Related Work	31
5.1 Machine Learning-Based Network Traffic Analysis	31
5.2 Physical and Hardware-Layer Fingerprinting	32
5.3 Alternative Protocol and Application-Layer Methods	32
6 Methodology	35
6.1 Characteristic Identification	35
6.2 System Design Philosophy	36
6.3 Evaluation Methodology	39
7 Fingerprinting Methods and Characteristics	41
7.1 Scope of Device Fingerprinting	41
7.2 WLAN Characteristics	43
7.3 TCP/IP Characteristics	48
7.4 Application Layer Characteristics	51
7.5 Putting it all together - Fingerprinting Schema	54
8 Implementation	57
8.1 System Architecture	57

8.2	ESP32 Firmware	59
8.3	Frontend Web Application	69
9	Evaluation and Results	79
9.1	Testing Environment and Device Setup	79
9.2	Fingerprinting Effectiveness Analysis	81
9.3	Layer-Specific Performance Evaluation	88
9.4	mDNS and MAC Vendor Lookup Results	91
9.5	System Performance Metrics	92
9.6	Database Contribution and Validation	93
10	Discussion	95
10.1	Contribution of the Work	95
10.2	Limitations of the Study	96
10.3	Ethical Considerations	98
10.4	Future Work	98
11	Conclusion	101
	Overview of Generative AI Tools Used	103
	List of Figures	105
	List of Tables	107
	Acronyms	109
	Bibliography	111

Kurzfassung

Die zunehmende Verbreitung von Internet-of-Things(IoT)-Geräten erschwert das Netzwerkmanagement und erhöht die Sicherheitsrisiken – eine zuverlässige Geräteidentifikation ist daher unerlässlich. Diese Arbeit begegnet dieser Herausforderung mit einem mehrschichtigen Fingerprinting-Ansatz. Dabei werden Merkmale aus der WLAN-Schicht (IEEE 802.11), dem TCP/IP-Stack und der Anwendungsschicht kombiniert, um Gerätemodelle voneinander unterscheiden zu können. Die Grundannahme lautet, dass die Zusammenführung der Merkmale aller Netzwerkschichten etwaige Mehrdeutigkeiten einzelner Schichten beseitigen kann.

Zur Prüfung wurde ein praxisnahes, erweiterbares System entwickelt. Ein kostengünstiger ESP32-Mikrocontroller fungiert dabei als dedizierte Probing-Einheit für Low-Level-Netzinteraktionen, während eine Host-Anwendung die Steuerung, die Datenanalyse und ein Plug-in-basiertes Fingerprinting-Framework bereitstellt. Es wurde untersucht, inwieweit sich mit dieser ressourcenbeschränkten Hardware Aufgaben wie ARP-Scans, maßgeschneiderte TCP-Pakete und passives WLAN- bzw. UDP-Monitoring durchführen lassen.

Tests an einer breiten Palette von verbraucherorientierten IoT-Geräten bestätigten die Wirksamkeit des Ansatzes. Zwar zeigten einzelne Merkmale Schwächen, etwa Kollisionen bei Geräten mit identischer Hardwareplattform auf WLAN-Ebene, doch der kombinierte Fingerprint erreichte eine Eindeutigkeitsrate von 100 % über alle Gerätemodelle, für die ein Fingerprint erzeugt werden konnte. Die Arbeit belegt zudem die Eignung des ESP32 für diese Aufgabe sowie die Notwendigkeit aktiver Verfahren wie Deauthentifizierungen, um genügend Probe Requests zu generieren. Der Hauptbeitrag liegt weniger in einem neuen Fingerprinting-Merkmal als in der ganzheitlichen Konzeption, praktischen Validierung und Offenlegung eines integrierten, leicht zugänglichen und erweiterbaren Systems, welches die Sichtbarkeit und Sicherheit moderner IoT-Umgebungen erhöht.

CHAPTER 2

Abstract

The proliferation of Internet of Things (IoT) devices presents significant network management and security challenges, making robust device identification critical. This thesis confronts this challenge by investigating the efficacy of a multi-layered fingerprinting approach that combines network characteristics from the WLAN (IEEE 802.11), TCP/IP, and application layers to achieve granular, model-level identification. The central hypothesis is that synthesizing data from these distinct layers can resolve ambiguities inherent in single-layer methods.

To validate this approach, a practical and extensible fingerprinting system was designed and implemented. The system architecture features a low-cost ESP32 microcontroller as a dedicated network probing unit, responsible for low-level network interactions, and a host application that provides user control, data analysis, and a modular plugin system for different fingerprinting techniques. The research addressed the feasibility of using such resource-constrained hardware for tasks including ARP scanning, custom TCP packet crafting, and passive monitoring of Wi-Fi and UDP traffic.

The evaluation, conducted on a diverse set of consumer-grade IoT devices, confirmed the effectiveness of the multi-layered strategy. While individual layers showed limitations—such as collisions between devices sharing a common hardware platform at the Wi-Fi layer—the combined fingerprint achieved a 100% model-level uniqueness rate across all fingerprinted devices. The study also validated the viability of the ESP32 for this role and demonstrated the necessity of techniques like active deauthentication-based probe request instigation to ensure sufficient data collection. The primary contribution of this work is not a novel fingerprinting feature, but the holistic design and practical validation of an integrated, accessible, and extensible system that provides a powerful tool for enhancing visibility and security in modern IoT environments.

Introduction

By now, it is a well-observed fact that there is an ever-growing number of Internet of Things (IoT) devices interconnected to each other and/or, as the name suggests, to the internet. This upward trend is expected to continue, reaching approximately 30 billion devices by 2030 [1]. The IoT ecosystem ranges from industrial appliances as part of the so-called Fourth Industrial Revolution, to devices developed for the end user market such as the smart home sector.

The proliferation of IoT devices has resulted in intricate networks that are populated with a diverse array of connected devices coming from a similar wide range of device manufacturers. This, and the absence of standardization across the IoT landscape, makes it challenging to find a reliable method of identification [2], leading to vulnerabilities and management challenges. Thus, discovering and cataloging those devices is an important requirement to ensure integrity, resilience and reliability within a given network.

This thesis confronts the challenge of IoT device identification by exploring the nuances of their network communication patterns. The central hypothesis is that device-specific implementations of protocol behavior—even within standardized protocols—can reveal subtle but stable deviations that can be leveraged for identification. While any single characteristic might be insufficient for unique identification, a multi-layered approach that combines information from different network protocols can create a composite fingerprint of high granularity and reliability.

The focus of this work is directed toward the lower layers of the network stack, specifically the WLAN (IEEE 802.11) and TCP/IP layers. This decision is motivated by several key factors. Firstly, it builds upon prior work by Stöger [3], which focused on application-layer scanning, by exploring the rich, yet often overlooked, information present at the foundational layers of network communication. A central motivation for this thesis was also to create a fingerprinting solution on an easily accessible and portable platform, inspired by Stöger’s implementation on Android smartphones. The chosen hardware

platform, the ESP32 System-on-Chip (SoC), aligns perfectly with this goal. The ESP32 is not only low-cost and widely available but is—as identified in this work—also particularly well-suited for low-level network interaction, providing the necessary access to raw packet manipulation and monitor mode functionalities that are often restricted on higher-level platforms like Android devices.

A multi-layered fingerprinting strategy was chosen because it directly addresses the limitations inherent in single-layer methods. As the evaluation in this thesis demonstrates, characteristics from one layer can resolve ambiguities present in another. For example, while the Wi-Fi fingerprinting method encountered collisions between different device models that shared a common hardware platform, these collisions were successfully resolved by incorporating application-layer data, which revealed a unique product key for each device. This synergy between layers is the cornerstone of the developed methodology, enabling a more robust and precise identification process.

To address these challenges, this thesis is guided by the following research questions:

1. Which characteristics found in the WLAN standard and the overlying IP suite might be used for device fingerprinting and how could they be combined to create a unique fingerprint for an IoT device?
2. To what extent can these characteristics be extracted using the ESP32 SoC, and what system architecture would best support this?

To validate this approach, this thesis details the design, implementation, and practical evaluation of an integrated fingerprinting system. The system comprises an ESP32-based network probing unit, which performs the low-level network interactions, and a host application that provides user control, data analysis, and an extensible, plugin-based architecture for different fingerprinting methods. The primary contributions of this work include the practical validation of the multi-layered fingerprinting approach on accessible, low-cost hardware, the development of novel implementation techniques for network analysis on a resource-constrained platform, and the creation of a flexible and extensible framework for future research in IoT device identification.

This thesis is structured as follows: chapter 4 provides foundational knowledge on the core technologies and concepts relevant to this thesis. Chapter 5 reviews existing literature on IoT device fingerprinting, contextualizing the contributions of this work. Chapter 6 outlines the overall research methodology, detailing the systematic approach taken to develop and evaluate the fingerprinting system. Chapter 7 details the specific fingerprinting characteristics selected from the WLAN, TCP/IP, and application layers. Chapter 8 describes the system architecture and implementation of both the ESP32 firmware and the host application. Chapter 9 presents the results of the practical evaluation, analyzing the system's performance in terms of coverage, uniqueness, reproducibility, and stability. Finally, chapter 10 discusses the implications of the findings, acknowledges the limitations of the study, and proposes directions for future research, followed by a concluding summary of the work.

Background

This chapter establishes the technical foundation required to comprehend the device fingerprinting methodologies explored in this thesis. Identifying IoT devices by their network behavior requires a detailed understanding of their communication protocols and hardware platforms. This chapter provides that essential context, covering fundamental networking principles, key protocols, and the capabilities of the hardware platform central to this work.

The chapter begins with an overview of core networking concepts, including layered communication models like the OSI and TCP/IP models, which provide a structured framework for understanding network interactions. It then delves into the specifics of the IEEE 802.11 WLAN standard, with a focus on the MAC layer and management frames that are crucial for passive wireless fingerprinting. Following this, a detailed examination of the TCP/IP protocol suite is presented, covering essential protocols such as ARP, IP, TCP, and UDP, whose implementation details are leveraged for active fingerprinting. The discussion then moves to application-layer service discovery mechanisms, specifically mDNS and DNS-SD, which are prevalent in modern IoT ecosystems. Finally, the chapter introduces the ESP32 System-on-Chip for the practical implementation of the probing unit, including details of its programming environment and the lwIP TCP/IP stack, which are central to the system's design and capabilities.

4.1 Networking Fundamentals

A foundational understanding of network communication principles is essential for contextualizing the device fingerprinting techniques discussed in this thesis. Modern networking is deconstructed into a series of distinct tasks organized into layers. This layered approach, defined by standardized models, ensures that different technologies can interoperate seamlessly. This section provides an overview of these layered models and core networking

concepts that underpin the rest of this work, drawing primarily from the principles outlined in Kozierok (2005), *The TCP/IP Guide* [4].

4.1.1 The OSI Reference Model

The most significant innovation for managing network complexity is the concept of layering. A networking model divides the required functions into a stack of layers, each responsible for a specific set of tasks. This division provides several benefits, including modularity, which allows a technology at one layer to be updated or replaced without impacting the others, and abstraction, which hides the complex inner workings of lower layers from the higher ones.

The most widely recognized framework is the OSI Reference Model, which structures network communication into seven distinct layers. While few network protocol suites strictly adhere to this seven-layer structure, the model is an invaluable educational and descriptive tool. The layers are:

- **Layer 7: Application Layer:** The highest layer, providing services directly to user applications. Protocols at this layer define the rules for specific tasks, such as web browsing (Hypertext Transfer Protocol (HTTP)) or file transfer (File Transfer Protocol (FTP)).
- **Layer 6: Presentation Layer:** Responsible for data translation, compression, and encryption. It ensures that data sent from the application layer of one system can be read by the application layer of another.
- **Layer 5: Session Layer:** Manages the establishment, maintenance, and termination of sessions—persistent logical links—between applications.
- **Layer 4: Transport Layer:** Provides for end-to-end communication between software processes on different hosts. It is responsible for segmenting data and can offer either connection-oriented (e.g., TCP) or connectionless (e.g., UDP) services.
- **Layer 3: Network Layer:** Handles the routing of data across different networks to form an internetwork. It uses logical addresses (e.g., IP addresses) to identify devices on the broader network.
- **Layer 2: Data Link Layer:** Manages communication on a local network segment. It is responsible for framing data, performing error detection, and uses physical hardware addresses (MAC addresses) to identify devices on the same local network. The IEEE 802.11 standard operates at this layer.
- **Layer 1: Physical Layer:** The lowest layer, defining the physical and electrical specifications for the network hardware, including cables, connectors, and signaling methods for transmitting raw bits of data.

4.1.2 The TCP/IP Model

The TCP/IP protocol suite, which forms the basis of the Internet, uses its own, more condensed four-layer architectural model. This model is often seen as more practical as it aligns more closely with the implementation of TCP/IP. Figure 4.1 illustrates the relationship between the two models.

- **Application Layer:** Combines the functions of the OSI Application, Presentation, and Session layers.
- **Transport Layer:** Corresponds directly to the OSI Transport Layer, featuring TCP and UDP.
- **Internet Layer:** Corresponds to the OSI Network Layer, with the Internet Protocol as its core component.
- **Link Layer:** Corresponds to the OSI Data Link and Physical Layers, defining how TCP/IP interfaces with the underlying network hardware.

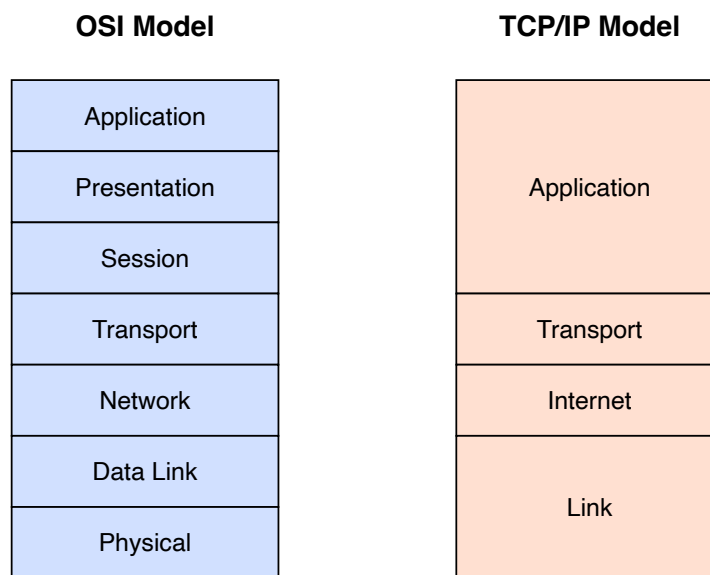


Figure 4.1: Comparison of the OSI and TCP/IP Layered Models (adapted from [4])

4.1.3 Packet Switching

Modern data networks, including those based on TCP/IP, predominantly use a technique called **packet switching**. In this paradigm, a larger piece of data is broken down into smaller, manageable blocks called packets. Each packet is independently addressed

and routed across the network. The packets may travel along different paths to reach their destination, where they are then reassembled into the original data stream. This contrasts with **circuit switching**, the method used by the traditional telephone system, where a dedicated, end-to-end connection (a circuit) is established for the duration of the communication. Packet switching allows for more efficient use of shared network resources, as multiple communications can be interleaved over the same infrastructure.

4.1.4 Data Encapsulation

As data moves down the protocol stack from a higher layer to a lower one, it undergoes a process called **data encapsulation** (illustrated in Figure 4.2). At each layer, the data received from the layer above (known as the Service Data Unit, or SDU) is wrapped with a new header (and sometimes a footer) containing control information specific to that layer. This newly wrapped message is the Protocol Data Unit (PDU) for that layer. For example, the Transport Layer encapsulates application data into a TCP segment (its PDU). This segment is then passed to the Network Layer, where it becomes the payload (SDU) and is encapsulated within an IP packet (the Network Layer's PDU). This IP packet is then encapsulated within a Data Link Layer frame (e.g., an Ethernet or 802.11 frame). This process ensures that each layer can perform its function without needing to interpret the data from other layers.

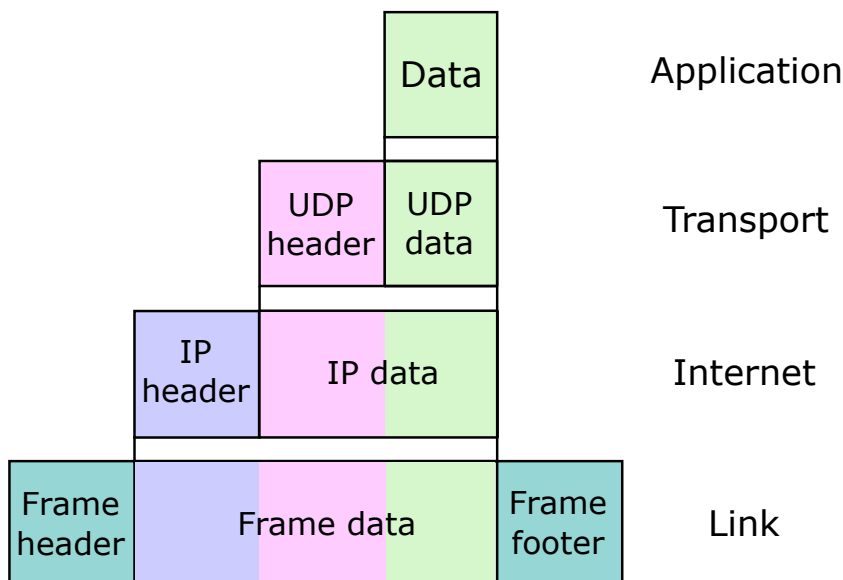


Figure 4.2: The process of data encapsulation as data passes down the protocol stack [5]

4.1.5 Network Addressing

For data to reach its correct destination, networks rely on addressing schemes. Two fundamental types of addresses are used in TCP/IP networks:

- **Physical Address (MAC Address):** A unique 48-bit hardware address assigned to a Network Interface Controller (NIC) by the manufacturer. It operates at the Data Link Layer (Layer 2) and is used for addressing on a local network segment (e.g., all devices connected to the same Wi-Fi AP).
- **Logical Address (IP Address):** A 32-bit (for IPv4) or 128-bit (for IPv6) address assigned to a device on a network. It operates at the Network Layer (Layer 3) and is used for routing data across different networks in an internetwork. Unlike the static MAC address, an IP address is typically assigned dynamically and can change.

4.1.6 Client/Server Model

Many network services and applications operate on a **client/server model**. In this paradigm, a powerful, centralized *server* provides resources or services. Multiple *clients* initiate communication by sending requests to the server, which then processes the requests and returns responses.

4.2 802.11 WLAN Standard and MAC Fundamentals

This section predominantly incorporates information presented in Gast (2005), *802.11 Wireless Networks: The Definitive Guide* [6].

The Institute of Electrical and Electronics Engineers (IEEE) 802.11 standard defines the specifications for Wireless Local Area Network (WLAN), commonly known as Wi-Fi. As part of the broader IEEE 802 family, 802.11 focuses on the two lowest layers of the OSI model: the Physical (PHY) layer and the Data Link layer. The Data Link layer in 802.11 is further subdivided into the Logical Link Control (LLC), typically conforming to the 802.2 standard shared by other 802 networks, and the Medium Access Control (MAC) sublayer. This section provides an overview of the 802.11 fundamentals, with a particular emphasis on the MAC layer, which governs how devices access the shared wireless medium.

4.2.1 Network Components and Topologies

An 802.11 network comprises several key components:

- **Station (STA):** Devices equipped with wireless network interfaces, such as laptops, smartphones, or IoT devices.

- **Access Point (AP):** Devices that function as a bridge between the wireless medium and a wired network infrastructure (the Distribution System). APs manage associated stations and coordinate access.
- **Wireless Medium:** The radio frequencies used for transmitting data frames. Various PHY layers (e.g., 802.11b/g/a/n/ac/ax) define different modulation and coding schemes for this medium.
- **Distribution System (DS):** The infrastructure, typically a wired Ethernet network, used to connect APs and integrate the WLAN with other networks.

These components form network topologies. The two most common (visualized in Figure 4.3) are: (1) **Independent Basic Service Set**, or ad hoc network, where stations communicate directly without an AP, and (2) the more widely used topology **Infrastructure Basic Service Set**, where all communication is relayed through a central AP. Multiple BSSs connected via a DS form an **Extended Service Set (ESS)**, allowing for larger coverage areas and station mobility (roaming) between APs within the same ESS, identified by a common Service Set Identifier (SSID).

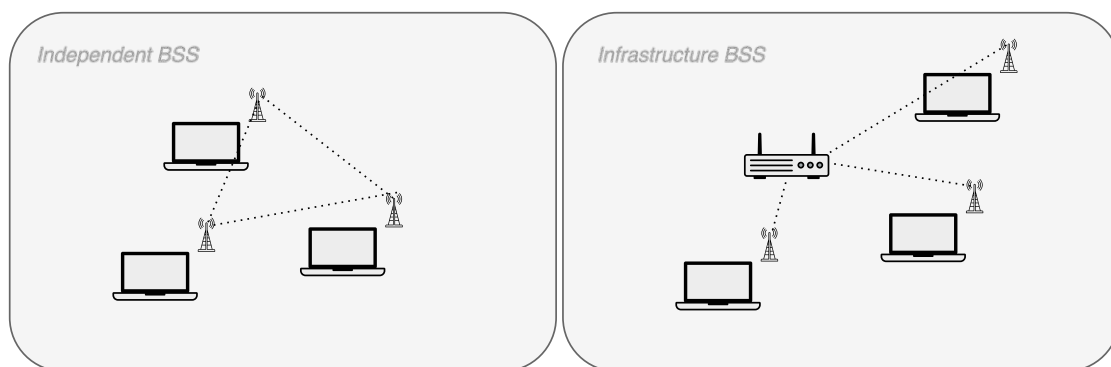


Figure 4.3: The 802.11 Independent Basic Service Set (BSS) and Infrastructure BSS topologies (adapted from [6])

4.2.2 The 802.11 MAC Layer

The 802.11 MAC layer is significantly more complex than its wired Ethernet (802.3) counterpart, primarily due to the challenges inherent in wireless communication, such as unreliable link quality and the hidden node problem. Its primary function is to coordinate access to the shared wireless medium using specific rules known as Coordination Functions. The most fundamental is the Distributed Coordination Function (DCF), a contention-based system relying on Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). Under CSMA/CA, stations listen before transmitting (carrier sense) and use random backoff timers and acknowledgments to avoid collisions, rather than detecting them post-occurrence like Ethernet.

MAC Frame Structure

Key fields of the 802.11 MAC frame (as seen in Figure 4.4) format include:

- **Frame Control:** Contains protocol version, frame type (Management, Control, Data) and subtype, flags indicating direction (To/From DS), fragmentation status, retry status, power management state, security status (Protected Frame), etc.
- **Duration/ID:** Sets the NAV or carries other information (e.g., Association ID in PS-Poll frames).
- **Address Fields:** Up to four 48-bit address fields are present, used variously for Destination Address (DA), Source Address (SA), Receiver Address (RA), Transmitter Address (TA), and Basic Service Set ID (BSSID), depending on the frame type and context (e.g., infrastructure vs. ad hoc).
- **Sequence Control:** Contains sequence and fragment numbers for reassembly and duplicate detection.
- **Frame Body:** Carries the payload (e.g., IP packet), typically encapsulated using 802.2 LLC/SNAP. Maximum payload size depends on the standard revision and configuration.
- **Frame Check Sequence (FCS):** A 32-bit CRC for error detection covering the frame body and MAC header.

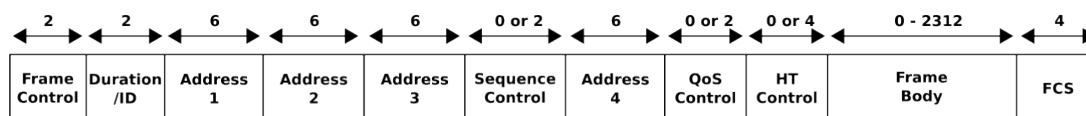


Figure 4.4: 802.11 MAC Frame [7]

Management Frames

As briefly mentioned in the description of the MAC frame control, 802.11 defines three main frame types: Management, Control, and Data frames. Each type serves distinct purposes within the network. While data frames are used to carry user payload and control frames manage medium access, management frames are essential for establishing and maintaining connections within an 802.11 network by handling tasks such as network discovery, authentication and association. The remainder of this section focuses on the management frames, which are crucial for the fingerprinting techniques outlined in chapter 7.

Management frames share a common MAC header structure but utilize their frame body to convey specific management information through a combination of fixed-length

fields and variable-length *Information Elements (IEs)*. IEs are tagged data structures, each identified by an Element ID and length, allowing for flexibility and extensibility. Examples include the SSID, Supported Rates, and security parameters like the Robust Security Network (RSN) IE.

A key example of a management frame is the **Probe Request**. Stations transmit Probe Requests to actively scan for nearby 802.11 networks. A Probe Request typically contains two crucial IEs: the SSID the station is looking for (which can be a specific network name or a broadcast SSID to discover any network) and the data rates the station supports (as can be seen in Figure 4.5). Networks receiving a Probe Request check these IEs for compatibility. If a match is found (e.g., the network's SSID matches the request and the station supports the network's mandatory rates), the network (specifically, the AP in an infrastructure BSS or a designated station in an independent BSS) responds with a **Probe Response**. The Probe Response contains detailed information about the network's capabilities, similar to a Beacon frame, allowing the probing station to initiate the authentication and association process. Other management frames handle authentication exchanges, association/reassociation requests and responses, disassociation/deauthentication notifications, and beacon transmissions announcing network presence and parameters.

```

▼ IEEE 802.11 Probe Request, Flags: .....C
  Type/Subtype: Probe Request (0x0004)
  ▶ Frame Control Field: 0x4000
    .000 0000 0000 0000 = Duration: 0 microseconds
    Receiver address: Broadcast (ff:ff:ff:ff:ff:ff)
    Destination address: Broadcast (ff:ff:ff:ff:ff:ff)
    Transmitter address: fc:3c:d7:5a:0f:95 (fc:3c:d7:5a:0f:95)
    Source address: fc:3c:d7:5a:0f:95 (fc:3c:d7:5a:0f:95)
    BSS Id: Broadcast (ff:ff:ff:ff:ff:ff)
    .... .... 0000 = Fragment number: 0
    0001 1010 0101 .... = Sequence number: 421
    Frame check sequence: 0x083c8935 [correct]
    [FCS Status: Good]
▼ IEEE 802.11 Wireless Management
  ▼ Tagged parameters (63 bytes)
    ▶ Tag: SSID parameter set: Magenta5933947
    ▶ Tag: Supported Rates 1, 2, 5.5, 11, [Mbit/sec]
    ▶ Tag: Extended Supported Rates 6, 9, 12, 18, 24, 36, 48, 54, [Mbit/sec]
    ▶ Tag: DS Parameter set: Current Channel: 9
    ▶ Tag: HT Capabilities (802.11n D1.10)

```

Figure 4.5: Probe Request of a Smart Ambient Light captured with Wireshark

4.3 TCP/IP Protocol Stack

The TCP/IP protocol suite is the set of communication protocols (including TCP and IP) that implements the protocol stack on which the Internet and most commercial networks run. The following subsections, which detail the core components of this stack, draw

heavily from the comprehensive overview provided by *The TCP/IP Guide* [4], unless another source is explicitly cited.

4.3.1 Address Resolution Protocol

While IP addresses govern the routing of datagrams across an internetwork, the actual delivery of a datagram between devices on the same local network segment (e.g., on the same Wi-Fi network) depends on Layer 2 hardware addresses, specifically MAC addresses. The Address Resolution Protocol (ARP), defined in RFC 826 [8], is the mechanism that bridges this Layer 3 to Layer 2 gap. Its function is to dynamically discover the MAC address associated with a given IP address, a critical prerequisite for any IP-based communication on a local network.

ARP messages have a simple format designed to be flexible across different network technologies. Key fields include the hardware and protocol types (e.g., Ethernet and IPv4), the length of each address type, and an opcode that specifies the message type (e.g., ARP Request or ARP Reply). The core of the message contains four address fields: the sender's hardware address, the sender's protocol address, the target's hardware address, and the target's protocol address.

The ARP transaction process, illustrated in Figure 4.6, is a stateful exchange involving several steps:

1. **Cache Check:** A source device wanting to send a packet first checks its local ARP cache to see if it already has a valid MAC address for the destination IP. If so, the remaining steps are skipped.
2. **ARP Request Generation:** If the address is not in the cache, the source device constructs an *ARP Request* message. It fills in its own MAC and IP addresses as the sender addresses and the destination's IP as the target protocol address. The target hardware address field is left blank, as this is the information being sought.
3. **Broadcast:** The source broadcasts the ARP Request to the entire local network.
4. **Processing by Network Devices:** All devices on the local network receive and process the request. Devices whose IP address does not match the target IP address in the request simply discard the packet.
5. **ARP Reply Generation:** The one device that owns the target IP address constructs an *ARP Reply*. It populates the sender address fields with its own MAC and IP addresses. For the target address fields, it uses the sender addresses from the original ARP Request.
6. **Cache Update (Destination):** As an optimization, the destination device adds the source's IP and MAC address from the request to its own ARP cache. This avoids the need for a separate ARP request if it needs to send a reply packet back to the source shortly after.

7. **Unicast Reply:** The destination device sends the ARP Reply directly (unicast) to the source device's MAC address, which it learned from the initial request.
8. **Processing and Cache Update (Source):** The source device receives the reply, extracts the sender's MAC address (which is the MAC address it was looking for), and adds the new IP-to-MAC mapping to its ARP cache for future use. It can now send its IP datagram.

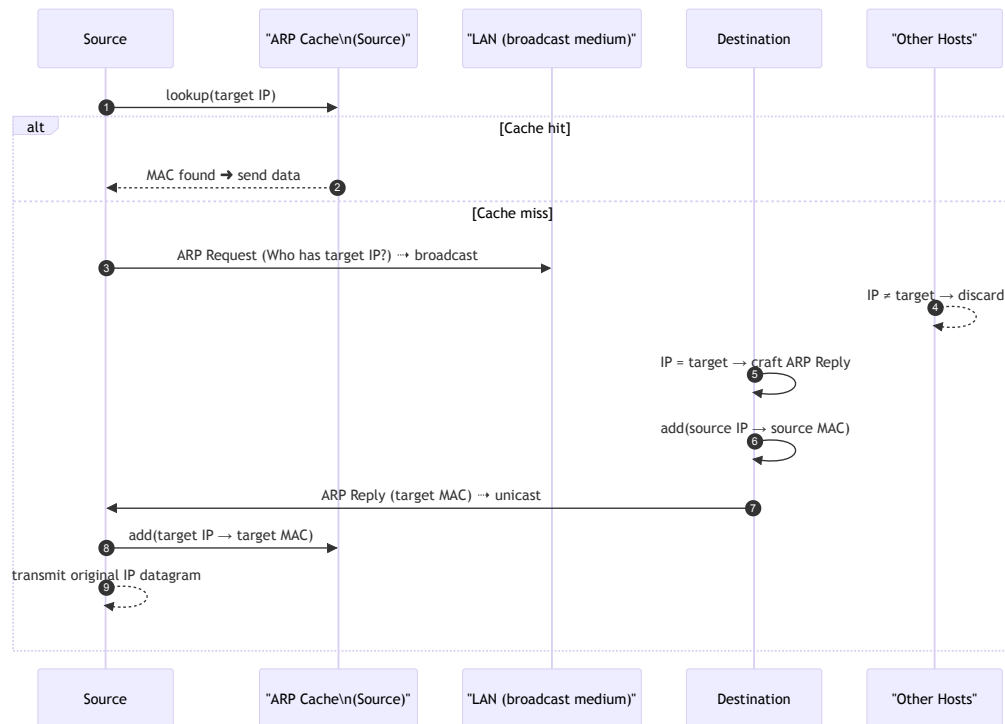


Figure 4.6: The Address Resolution Protocol (ARP) Transaction Process [4]

To improve efficiency and minimize the number of broadcast requests, operating systems maintain an ARP cache, which stores recent IP-to-MAC address mappings for a short period. Devices can also issue unsolicited ARP replies, known as *Gratuitous ARP*, to announce their IP-to-MAC mapping to the network (for example, upon connecting or changing an IP address). This mechanism is also useful for detecting IP address conflicts.

4.3.2 The Internet Protocol

Internet Protocol (IP) is the core protocol of the TCP/IP suite's Internet Layer, corresponding to the Network Layer (Layer 3) of the OSI model. Its primary responsibility is to provide a mechanism for addressing and routing packets of data, known as datagrams, between hosts on a potentially vast and complex internetwork [4]. IP is the "workhorse"

of TCP/IP, enabling the fundamental connectivity that all higher-level protocols rely upon.

A key design principle of IP is its connectionless and unreliable nature. It is connectionless because it does not establish a session before sending data; each datagram is treated as an independent unit and routed separately. It is considered unreliable because it does not guarantee delivery. IP makes a "best-effort" attempt to deliver datagrams, but it includes no mechanisms for error recovery, flow control, or acknowledgment of receipt. These reliability functions are intentionally left to higher-layer protocols, most notably TCP, allowing for a more flexible and efficient network layer that does not impose unnecessary overhead on applications that may not require it.

IPv4 Datagram Structure

To transmit data, IP encapsulates the payload received from the transport layer (e.g., a TCP segment) into an IP datagram. The structure of this datagram, defined in RFC 791 for IPv4, consists of a header and a payload. The header contains crucial control information for addressing and routing. The standard IPv4 header is 20 bytes long, with the possibility of including additional options. Figure 4.7 illustrates the format.

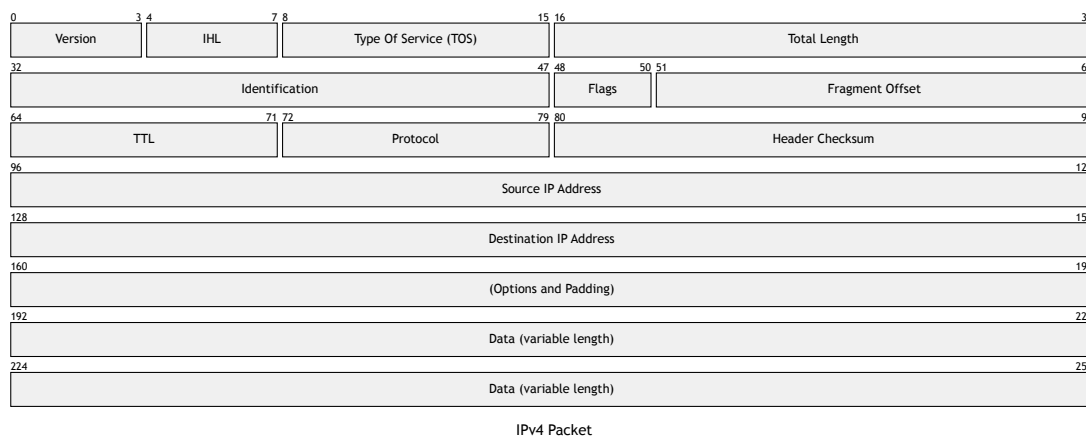


Figure 4.7: The IPv4 Datagram Header Format (adapted from [4])

Several fields within the IP header are particularly relevant to the fingerprinting techniques discussed in this thesis:

- **Version (4 bits):** Identifies the IP version. For IPv4, this is always 4.
- **Internet Header Length (IHL) (4 bits):** Specifies the length of the header in 32-bit words. The minimum value is 5 (for a 20-byte header).

- **Time to Live (Time to Live (TTL)) (8 bits):** A value that limits the lifespan of a datagram. It is decremented by each router that forwards the datagram. If the TTL reaches zero, the packet is discarded.
- **Protocol (8 bits):** Identifies the protocol of the encapsulated payload. A value of 6 indicates TCP, while 17 indicates UDP.
- **Header Checksum (16 bits):** An error-checking field calculated over the header to detect corruption.
- **Source and Destination Addresses (32 bits each):** The logical network addresses of the sending and receiving hosts.
- **Options (variable length):** Allows for additional control information, though not commonly used in typical traffic. The presence, absence, and type of options can be a distinguishing characteristic.

Fragmentation

IP must be able to handle data payloads larger than the Maximum Transmission Unit (MTU) of an underlying network link. If a datagram is too large, it can be divided into smaller pieces, a process known as fragmentation. The **Identification, Flags** (specifically the Don't Fragment (DF) and More Fragments (MF) bits), and **Fragment Offset** fields in the IP header are used to manage the fragmentation and reassembly of datagrams.

4.3.3 Transport Layer Protocols: TCP and UDP

Sitting above the Internet Protocol are the two primary transport layer protocols of the TCP/IP suite: the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). While both serve to bridge the gap between network-layer datagram delivery and application processes, they offer fundamentally different service models based on application requirements [4].

User Datagram Protocol

User Datagram Protocol (UDP) is a simple, connectionless transport protocol defined in RFC 768. It provides a minimalistic "wrapper" around IP, adding little more than transport-layer port addressing and an optional checksum for data integrity. Its key characteristics are:

- **Connectionless:** UDP does not establish a formal connection before sending data. Each UDP datagram is an independent transaction.
- **Unreliable:** It offers a "best-effort" delivery service, mirroring IP. There are no acknowledgments, no retransmission of lost packets, and no guarantee of ordered delivery.

- **Low Overhead:** The UDP header is only 8 bytes, resulting in minimal overhead and making it very fast.

Due to these characteristics, UDP is well-suited for applications where speed is prioritized over reliability. It is also used for simple, query-response protocols and supports broadcasting and multicasting, making it ideal for one-to-many service discovery mechanisms.

Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP), defined in RFC 793, is a cornerstone of the TCP/IP suite, providing a full-featured, connection-oriented transport service. It acts as a crucial bridge between higher-layer applications and the inherently unreliable IP, offering robust mechanisms for reliable, ordered, and error-checked data delivery [4]. Unlike UDP, TCP is designed for applications that require guarantees regarding data integrity and delivery.

Data Handling: Streams and Segments TCP treats data from applications as an unstructured *stream of bytes*. This stream-oriented approach offers maximum flexibility, as applications do not need to segment their data into fixed-size messages. Instead, TCP is responsible for packaging these bytes into discrete units called *segments* for transmission over the network. The size of these segments is dynamically determined based on factors such as the Maximum Segment Size (MSS) and the receiver's advertised window size.

Connection Establishment: The Three-Way Handshake Before any data exchange can occur, TCP establishes a connection between two devices using a process known as the *three-way handshake*. This handshake ensures that both the sender and receiver are ready to communicate and synchronizes their initial sequence numbers. The process involves three steps:

1. **SYN (Synchronize):** The initiating client sends a TCP segment with the SYN flag set to the server, indicating its desire to establish a connection and proposing an initial sequence number (ISN).
2. **SYN-ACK (Synchronize-Acknowledgment):** The server, upon receiving the SYN, responds with a SYN-ACK segment. This segment has both the SYN and ACK flags set. The ACK acknowledges the client's ISN (by incrementing it by one), and the SYN proposes the server's own ISN.
3. **ACK (Acknowledgment):** Finally, the client sends an ACK segment to the server, acknowledging the server's ISN. At this point, the connection is fully established, and data transfer can begin.

This handshake is crucial for reliable communication, as it sets the foundation for sequence numbering, acknowledgment, and flow control mechanisms that govern the subsequent data transfer. Figure 4.8 illustrates this process.

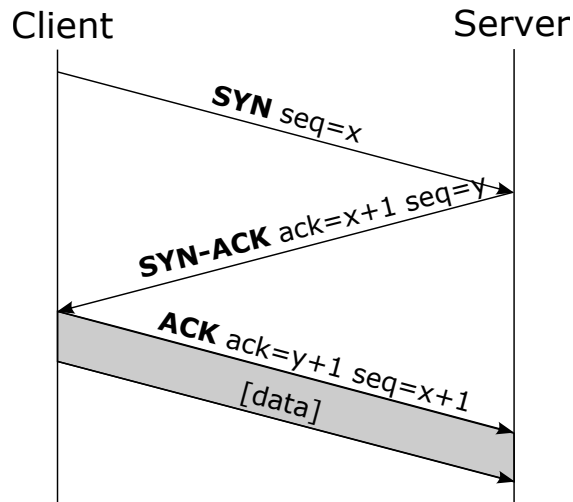


Figure 4.8: The TCP Three-Way Handshake Connection Establishment Procedure [?]

TCP Segment Format The complexity of TCP's features is reflected in its header, which is typically 20 bytes long (plus potential additional options). This header contains critical fields for managing connections, ensuring reliability, and controlling data flow. Figure 4.9 illustrates the TCP segment format.

- **Source Port (16 bits) and Destination Port (16 bits):** These fields identify the application processes at the source and destination hosts, respectively, enabling multiplexing and demultiplexing of data for multiple applications over a single IP address.
- **Sequence Number (32 bits):** Identifies the sequence number of the first byte of data in the current segment. During connection establishment, it carries the Initial Sequence Number (ISN).
- **Acknowledgment Number (32 bits):** If the ACK flag is set, this field contains the next sequence number the sender of this segment expects to receive, cumulatively acknowledging all prior bytes.
- **Data Offset (4 bits):** Specifies the length of the TCP header in 32-bit words, indicating where the actual data begins.
- **Control Bits (6 bits):** A set of flags that control the connection state and data flow. Key flags include:

- **SYN (Synchronize):** Used to initiate a connection and synchronize sequence numbers.
 - **ACK (Acknowledgment):** Indicates that the Acknowledgment Number field is valid.
 - **FIN (Finish):** Used to terminate a connection.
 - **RST (Reset):** Aborts a connection due to an error.
 - **PSH (Push):** Requests immediate delivery of buffered data to the application.
 - **URG (Urgent):** Indicates that the segment contains urgent data, pointed to by the Urgent Pointer field.
- **Window Size (16 bits):** Advertises the amount of receive buffer space available at the sender of this segment, used for flow control.
 - **Checksum (16 bits):** A crucial field for error detection, calculated over the entire TCP segment and a "pseudo-header" that includes fields from the IP header. This ensures data integrity and helps detect misdelivered segments.
 - **Urgent Pointer (16 bits):** If the URG flag is set, this field indicates the offset from the Sequence Number to the last byte of urgent data.
 - **Options (variable length):** Provides extensibility for additional TCP capabilities. Common options include:
 - **Maximum Segment Size (MSS):** Specifies the largest segment of data a device can receive.
 - **Window Scale (WS):** Allows for larger window sizes beyond the 16-bit limit.
 - **Selective Acknowledgment (SACK) Permitted:** Indicates support for selective retransmission of lost packets.
 - **Timestamps:** Used for more accurate Round-Trip Time (RTT) measurements and protection against wrapped sequence numbers.

4.4 Multicast DNS and DNS-Based Service Discovery

Multicast DNS (mDNS) and DNS-Based Service Discovery (DNS-SD) are key application layer protocols for zero-configuration networking, enabling devices to discover services on a local network without manual configuration or a designated DNS server [9, 10]. These protocols are particularly relevant in dynamic environments, such as those populated by IoT devices, where ease of deployment and self-organization are critical.

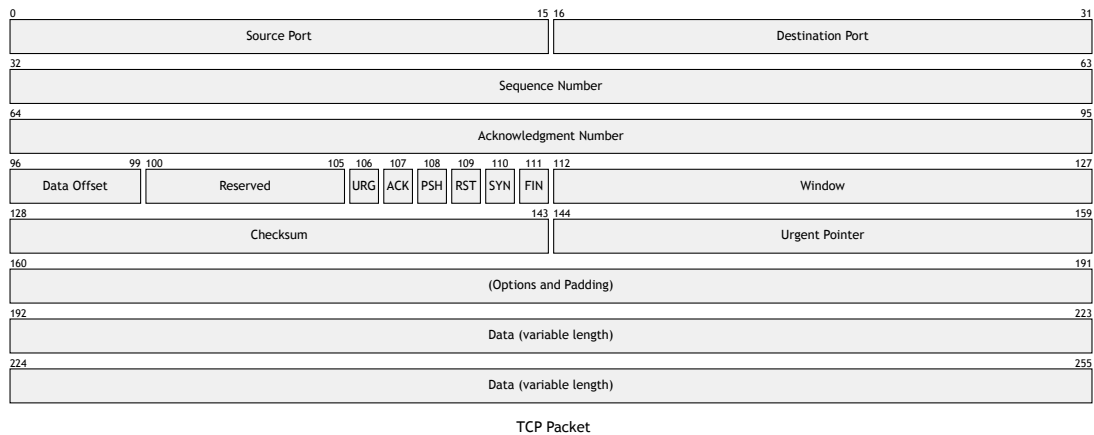


Figure 4.9: The TCP Segment Format (adapted from [4])

4.4.1 Multicast DNS

mDNS, defined in RFC 6762 [9], extends the traditional Domain Name System (DNS) to local network segments. Unlike conventional DNS, which relies on unicast queries to a designated server, mDNS utilizes IP multicast to perform DNS-like operations directly on the local link. This allows hosts to resolve hostnames and discover services within their broadcast domain without requiring a dedicated DNS server.

Key characteristics of mDNS include:

- **Link-Local Scope:** mDNS operates exclusively on the local network segment. It designates the top-level domain `.local.` for local use, ensuring that names within this domain are meaningful only on the link where they originate.
- **Multicast Addressing:** All mDNS queries and responses are sent to specific link-local multicast addresses: `224.0.0.251` for IPv4 and `FF02::FB` for IPv6. Communication occurs over UDP port 5353.
- **Self-Configuration and Conflict Resolution:** Devices using mDNS employ a probing mechanism to assert the uniqueness of their chosen names. If a conflict is detected, the protocol provides a mechanism for devices to automatically reconfigure with a new, unique name, ensuring network stability.
- **Standard DNS Message Format:** mDNS reuses the existing DNS message structure, name syntax, and resource record types, minimizing complexity and leveraging established DNS parsing capabilities.

The self-configuring nature of mDNS makes it highly suitable for environments where devices frequently join or leave the network, or where administrative overhead for network configuration is to be minimized.

4.4.2 DNS-Based Service Discovery

DNS-SD, specified in RFC 6763 [10], builds upon mDNS (though it can also operate with conventional Unicast DNS) to enable the discovery of network services. It defines a standardized way to name and structure DNS resource records to facilitate service enumeration and resolution.

The core mechanism of DNS-SD involves three primary DNS record types:

- **PTR (Pointer) Records:** Used for service instance enumeration. A client queries for a PTR record with a name like `<Service>.<Domain>` (e.g., `_http._tcp.local.`) to discover a list of named instances of a desired service.
- **SRV (Service) Records:** Once a service instance name is discovered (e.g., `MyPrinter._ipp._tcp.local.`), an SRV record provides the target hostname and port number where the service can be reached. This decouples the service's logical name from its physical network location.
- **TXT (Text) Records:** These records are crucial for conveying additional, often human-readable, information about a service instance. They store arbitrary key/value pairs (e.g., `model=XYZ, firmware=1.2.3`). This structured data allows clients to gain rudimentary information about a service without needing to connect to it, serving as a performance optimization and enriching the service's profile.

The ability of DNS-SD to convey rich metadata through TXT records is particularly valuable for device identification, as it can expose manufacturer, model, and capability details that are not readily available from lower-layer protocols.

4.5 ESP32 System-on-Chip

The ESP32 is a series of low-cost, low-power SoC microcontrollers developed by Espressif Systems. It has become a cornerstone of the Internet of Things (IoT) landscape due to its robust feature set, accessible price point, and strong community support. The chip integrates either a dual-core or single-core Tensilica Xtensa LX6 microprocessor or, in newer variants, a RISC-V processor, along with built-in Wi-Fi and dual-mode Bluetooth capabilities, making it a versatile platform for connected devices. Its architecture is designed for efficiency and performance in a small footprint, enabling its use in a wide array of applications, from simple sensor nodes to complex IoT gateways [11].

While the ESP32 SoC itself is a small, intricate component, it is most commonly utilized in the form of a development board, as shown in Figure 4.10. These boards are designed to make the SoC's powerful features accessible for rapid prototyping and development. They typically include essential supporting circuitry, such as a USB-to-serial converter for programming and communication, a voltage regulator to manage power, and breakout pins that expose the ESP32's General-Purpose Input/Output (GPIO) and other interfaces.

This integrated design significantly lowers the barrier to entry for developers, allowing them to focus on software development and hardware integration without needing to design custom printed circuit boards for basic functionality [12].

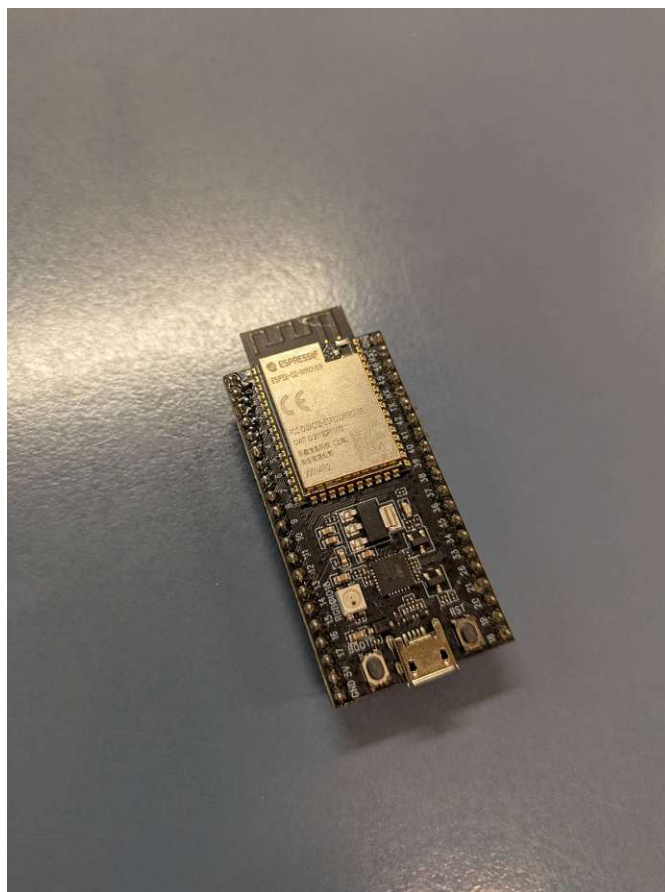


Figure 4.10: An ESP32 development board (ESP32-S2-WROVER) used for this thesis

4.5.1 Programming Environment

Developing software for the ESP32 SoC is facilitated by a structured ecosystem of frameworks and tools designed to manage the hardware's complexity. The primary development framework is the Espressif IoT Development Framework (ESP-IDF), which provides a comprehensive toolchain, libraries, and APIs for the ESP32. For this project, development was managed using PlatformIO, a versatile build system that integrates with ESP-IDF to streamline project configuration, library management, and the debugging process.

ESP-IDF and PlatformIO

The **ESP-IDF** [13] is the official development framework from Espressif. It is built upon the FreeRTOS real-time operating system and provides a rich set of software components, including drivers for peripherals (GPIO, I2C, SPI), middleware for networking (TCP/IP stack, TLS), and system-level features like power management and over-the-air (OTA) updates. ESP-IDF offers a C/C++ based Software Development Kit (SDK) that grants developers direct access to the hardware's capabilities, which is essential for the low-level network operations required for device fingerprinting.

While ESP-IDF can be used directly with its own build system, this project utilizes **PlatformIO** [14], an open-source ecosystem for embedded development. PlatformIO acts as a layer on top of ESP-IDF, simplifying many aspects of the development workflow. It automates the installation of toolchains and debugging tools, manages project dependencies, and provides a consistent command-line interface and integration with code editors like Visual Studio Code. This abstraction allows the developer to focus on the application logic rather than on the intricacies of the build environment.

4.5.2 lwip TCP/IP Stack

At the core of the ESP-IDF's networking capabilities is lightweight IP (lwIP), an open-source TCP/IP stack designed for embedded systems with limited resources [15]. It provides all the fundamental protocols of the TCP/IP suite, including IP, ICMP, UDP, and TCP. While lwIP offers a standard Berkeley Software Distribution (BSD) socket API for high-level network programming, certain advanced tasks, such as the active TCP/IP fingerprinting detailed in section 7.3, require more direct control over packet construction than the standard socket API allows.

To send custom IP datagrams, an application must be able to construct the entire packet, including the IP header. In traditional network programming using the BSD socket API, this is typically achieved by creating a raw socket and setting the `IP_HDRINCL` ("IP Header Included") socket option. This option signals to the operating system's network stack that the application will provide the IP header, and the stack should not prepend its own. However, the BSD socket compatibility layer provided by ESP-IDF does not support the `IP_HDRINCL` option, rendering it unsuitable for the kind of low-level packet crafting needed for active fingerprinting [16].

lwIP Raw API

To overcome this limitation, it is necessary to bypass the high-level socket API and interact directly with lwIP's native, lower-level "Raw API" [17]. This API provides fine-grained control over the IP layer and operates on two core data structures: the Protocol Control Block (PCB) and the Packet Buffer (pbuf). A PCB is a data structure that holds the state of a connection, acting as a conduit to the IP layer for sending or receiving packets. The memory management of the lwIP stack revolves around the pbuf structure.

A pbuf represents a packet buffer, designed to manage network packet data with high efficiency by minimizing memory copies. Instead of allocating a single, contiguous block of memory for an entire packet, a pbuf can consist of a chain of smaller buffers linked together. This architecture is particularly advantageous for network protocol layering. When a packet moves down the stack (e.g., from the application to the TCP layer), a new header can be prepended by simply allocating a new pbuf for the header and linking it to the front of the existing pbuf chain that holds the payload.

The process of sending a custom packet using the Raw API involves several steps, as illustrated in Listing 4.1. A critical responsibility when using the Raw API is the manual construction of protocol headers and the correct calculation of their checksums. While this offers maximum control, it is also error-prone. Fortunately, although not always prominently documented, lwIP provides a suite of helper functions and data structures within its header files that significantly simplify these tasks. Key among these are `ip.h` and `tcp.h`, which define the header structures (`ip_hdr`, `tcp_hdr`), and `inet_chksum.h`, which contains the necessary checksum calculation routines.

The IP header is constructed by populating an `ip_hdr` structure. The lwIP stack provides a set of macros (e.g., `IPH_VHL_SET`, `IPH_TTL_SET`) to correctly format and set the various fields. Once the header fields are populated, its checksum is calculated by calling the `inet_chksum()` function. The TCP checksum calculation is more complex because it must include a "pseudo-header" containing the source and destination IP addresses, the protocol number, and the TCP segment length. The lwIP stack abstracts this complexity away with the `ip_chksum_pseudo()` function.

Furthermore, since ESP-IDF runs on the FreeRTOS operating system, applications often interact with the lwIP stack from multiple threads. To optimize for performance and a low memory footprint, the core of the lwIP stack is not designed to be inherently thread-safe. Instead, it employs a dedicated 'tcpip_thread' that is responsible for all core processing. Direct calls into the stack's internal functions from other application threads could lead to race conditions and data corruption, for instance, by manipulating linked lists of PCBs or pbufs simultaneously.

To manage this, lwIP provides a mandatory core locking mechanism. Any application code that needs to call a function within the lwIP API must do so within a critical section, managed by the `LOCK_TCPIP_CORE()` and `UNLOCK_TCPIP_CORE()` macros. These macros ensure that the enclosed code block has exclusive access to the TCP/IP core, preventing concurrent modifications and maintaining the integrity of the stack's internal state. This approach provides thread safety without incurring the constant overhead of fine-grained locking throughout the stack, which is a critical design choice for resource-constrained embedded systems.

BSD Sockets for Transport Layer Communication

For applications requiring direct interaction with the transport layer (such as sending and receiving standard UDP or TCP packets) the most common approach is to use

```

1 // 1. Create a new raw Protocol Control Block (PCB) for IPv4 and TCP
2 struct raw_pcb *pcb = raw_new_ip_type(IPADDR_TYPE_V4, IPPROTO_TCP);
3 // Set flag to indicate the application provides the IP header
4 pcb->flags |= RAW_FLAGS_HDRINCL;
5
6 // 2. Construct the IP and TCP headers manually
7 ip_hdr_t iphdr;
8 tcp_hdr_t tcphdr;
9 // ... (populate iphdr and tcphdr fields: src/dest IPs, ports,
10 // TCP flags, etc.) ...
11
12 // 3. Calculate checksums using lwIP helpers
13 IPH_CHKSUM_SET(&iphdr, 0); // Set to 0 before calculation
14 IPH_CHKSUM_SET(&iphdr, inet_chksum(&iphdr, IP4_HDRLEN));
15
16 tcphdr.chksum = 0; // Set to 0 before calculation
17 // ip_chksum_pseudo handles the complex pseudo-header calculation
18 tcphdr.chksum = ip_chksum_pseudo(p_tcp, IPPROTO_TCP,
19                                 tcp_len, &source_ip, &dest_ip);
20
21 // 4. Allocate a pbuf and copy headers into it
22 u16_t packet_len = sizeof(ip_hdr_t) + sizeof(tcp_hdr_t);
23 struct pbuf *p = pbuf_alloc(PBUF_IP, packet_len, PBUF_RAM);
24 memcpy(p->payload, &iphdr, sizeof(ip_hdr_t));
25 memcpy((uint8_t*)p->payload + sizeof(ip_hdr_t), &tcphdr, sizeof(tcp_hdr_t));
26
27 // 5. Send the packet within a thread-safe block
28 LOCK_TCPIP_CORE();
29 raw_sendto_if_src(pcb, p, &destination_ip_address,
30                  netif, &source_ip_address);
31 UNLOCK_TCPIP_CORE();
32
33 // 6. Free the pbuf and the raw PCB when done
34 pbuf_free(p);
35 raw_remove(pcb);

```

Listing 4.1: Practical Example of lwIP Raw API Usage for Sending a Custom TCP Packet.

an implementation of the BSD socket interface [18, 17]. This widely adopted API provides a standardized, high-level abstraction for network programming. It abstracts the complexities of the underlying TCP/IP stack, allowing developers to work with a familiar model that treats network communication endpoints as file descriptors. A key advantage of the BSD socket layer is that it is thread-safe by default, automatically managing the core locking required by lwIP.

The socket API workflow begins with the creation of a socket, a communication endpoint, using the `socket()` function. This function takes three arguments: the address family (e.g., `AF_INET` for IPv4), the socket type, and the protocol. The type determines the communication semantics:

- **SOCK_STREAM:** Provides a connection-oriented, reliable, and stream-based service, which corresponds to TCP. Data is transmitted as a continuous stream of bytes.
- **SOCK_DGRAM:** Provides a connectionless, unreliable, and datagram-based service, corresponding to UDP. Data is transmitted in discrete messages (datagrams).
- **SOCK_RAW:** Allows for direct access to lower-level protocols, such as IP or ICMP. As discussed in subsection 4.5.2, this is of limited use for custom packet crafting in ESP-IDF due to the lack of the `IP_HDRINCL` option, making the lwIP Raw API the necessary choice for such tasks.

For connectionless communication with UDP, a socket is typically created with `SOCK_DGRAM`. The `bind()` function can be used to associate the socket with a specific local IP address and port, which is necessary for a server that needs to receive packets on a well-known port. Data is then exchanged using `sendto()` and `recvfrom()`. These functions allow the application to specify the destination address for each outgoing packet and retrieve the source address of each incoming packet, reflecting the connectionless nature of UDP.

For connection-oriented communication with TCP (`SOCK_STREAM`), the workflow is stateful and differs between the client and server.

TCP Server Workflow

1. **socket():** A listening socket is created.
2. **bind():** The socket is bound to a local IP address and port number.
3. **listen():** The socket is placed into a passive listening state, ready to accept incoming connections. This function also sets a limit on the queue for pending connections.

4. **accept ()**: The server waits for a client to connect. When a connection request is received, `accept ()` creates a new socket dedicated to that specific connection and returns its file descriptor. The original listening socket remains open to accept further connections.
5. **recv ()/send ()**: All subsequent communication with the client occurs on the new socket.
6. **close ()**: The connection-specific socket is closed to terminate the session.

TCP Client Workflow

1. **socket ()**: A socket is created.
2. **connect ()**: The client attempts to establish a connection to the server's address and port, initiating the TCP three-way handshake.
3. **send ()/recv ()**: Once the connection is established, data is exchanged with the server.
4. **close ()**: The socket is closed to terminate the connection.

This structured, stateful approach provided by the BSD socket API is ideal for the majority of application-level network tasks, offering a robust and portable alternative to the more complex lwIP Raw API.

The esp-netif Module and netif->input()

In the ESP-IDF, the `esp-netif` module serves as a crucial abstraction layer that connects the lwIP TCP/IP stack with the underlying network interface drivers, such as Wi-Fi or Ethernet [19]. This "glue" layer is responsible for passing network packets between the hardware-specific driver and the protocol stack. It provides a unified API for network interface management, simplifying the integration of different network interfaces with lwIP.

The core of the packet reception process is the `netif->input ()` function pointer within the `netif` structure. This function is set by the lwIP stack to point to its internal packet processing entry point. When a network interface driver receives a packet from the hardware, it encapsulates the packet in a buffer and calls this `input ()` function to pass it up to the lwIP stack. For example, in the ESP32 Wi-Fi driver, a function like `wlanif_input ()` is registered as a callback. When a Wi-Fi frame is received, this function is invoked. It then prepares the packet and calls the `netif->input ()` function of the appropriate network interface, thereby injecting the packet into the lwIP stack. This mechanism allows for a clean separation between the device driver and the TCP/IP stack, making the system more modular. The following code snippet shows a simplified representation of how a Wi-Fi driver passes a received packet to the lwIP stack through `esp-netif`:

4. BACKGROUND

```
1 esp_netif_recv_ret_t wlanif_input(void *h, void *buffer, size_t len, void*
  l2_buff)
2 {
3   // ...
4
5   /* full packet send to tcpip_thread to process */
6   if (unlikely(netif->input(p, netif) != ERR_OK)) {
7     LWIP_DEBUGF(NETIF_DEBUG, ("wlanif_input: IP input error\n"));
8     pbuf_free(p);
9     return ESP_NETIF_OPTIONAL_RETURN_CODE(ESP_FAIL);
10  }
11
12  return ESP_NETIF_OPTIONAL_RETURN_CODE(ESP_OK);
13 }
```

Listing 4.2: Simplified Wi-Fi driver packet input

Related Work

The proliferation of IoT devices has introduced significant security and management challenges, making robust device identification a critical area of research. While the core fingerprinting characteristics selected for this thesis are detailed in Chapter 7, it is essential to understand the broader landscape of techniques proposed in the literature. This chapter surveys diverse methodologies for IoT device fingerprinting, focusing on approaches that are complementary to or different from the specific methods implemented in this work. The goal is to contextualize the contribution of this thesis, which lies in the practical integration of fundamental techniques into an accessible, multi-layered system.

5.1 Machine Learning-Based Network Traffic Analysis

A dominant trend in recent IoT fingerprinting research is the application of Machine Learning (ML) and Deep Learning (DL) to automatically classify devices from their network traffic. These approaches typically extract a large number of statistical features from traffic flows and use sophisticated algorithms to learn the complex patterns that distinguish one device type from another [20]. As noted in comprehensive surveys by Chowdhury and Abas [20] and by Sanchez et al. [21], this has become the de facto standard for achieving high classification accuracy in complex environments.

For instance, Sivanathan et al. [22] developed a multi-stage ML pipeline to classify IoT devices based on statistical features of their network flows, such as traffic volume, packet sizes, and server interactions. Their work demonstrated that ML models, particularly Random Forest classifiers, could effectively distinguish between different device types with high accuracy. Similarly, Meidan et al. [23] used a set of statistical features from the first 12 packets of a device's communication to train a Random Forest model, successfully identifying various IoT device types in a smart home environment.

More advanced DL techniques have also been applied. Bai et al. [24] utilized a combination of Long Short-Term Memory (LSTM) and Convolutional Neural Network (CNN) models to classify devices from traffic streams, while Ortiz et al. [25] used LSTM-based autoencoders to learn device behaviors from raw TCP packets. These methods leverage the ability of deep neural networks to automatically learn relevant features, reducing the need for manual feature engineering. While powerful, these approaches often require large, labeled datasets for training and significant computational resources, which contrasts with the goal of this thesis to create a lightweight, rule-based matching system that can operate with a smaller, curated database.

5.2 Physical and Hardware-Layer Fingerprinting

Another significant branch of research focuses on characteristics derived from the physical hardware of a device, which are often unique to an individual unit, not just a device model.

Radio Frequency (RF) Fingerprinting is a prominent example of this approach. As introduced in chapter 7, this technique exploits minute, unintentional imperfections in a device's radio transmitter hardware, which create a unique and stable "fingerprint" on the analog waveform of the transmitted signal [20]. The broader literature, extensively surveyed by Soltanieh et al. [26], confirms that these fingerprints can be captured using Software-Defined Radio (SDR) and analyzed with advanced signal processing or DL techniques to identify individual transmitters with very high accuracy. While powerful for its resistance to spoofing, the general consensus is that this method requires specialized SDR hardware, which is contrary to this project's goal of using low-cost, commodity components.

Clock Skew Fingerprinting is another hardware-based method that relies on the small, stable imperfections in a device's crystal oscillator, which causes its internal clock to drift at a unique rate. This drift can be measured remotely by analyzing the timestamps in network packets (e.g., TCP timestamps) [21]. While effective for distinguishing devices, its reliability can be affected by network jitter and temperature variations, and as noted by Polčák and Franková [27], the uniqueness may not scale to very large numbers of devices.

5.3 Alternative Protocol and Application-Layer Methods

While this thesis focuses on a specific set of protocols, other researchers have explored a wide range of protocols for fingerprinting.

For example, Domain Name System (DNS) traffic has been used to identify devices [28], as many IoT devices contact specific service or update servers, revealing their vendor or function. Similarly, characteristics of the TLS/SSL handshake [29], such as the offered cipher suites, can serve as a fingerprint for the client-side implementation.

Other work has focused on IoT-specific protocols like MQTT [30], analyzing their specific communication patterns to distinguish devices. These approaches highlight the richness of the application layer for fingerprinting but also underscore the challenge of heterogeneity; a fingerprinting system must be extensible to handle the ever-growing number of protocols in the IoT ecosystem. This challenge is a primary motivator for the modular, plugin-based architecture designed in this thesis.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Methodology

The methodology of this thesis is structured in two phases. The first phase consists of a theoretical investigation to identify protocol-layer characteristics suitable for device fingerprinting. The second phase encompasses the conceptual design and implementation of a system to extract and utilize these identified characteristics.

Following are the guiding research questions for this work:

1. Which characteristics found in the WLAN standard and the overlying IP suite might be used for device fingerprinting and how could they be combined to create a unique fingerprint for an IoT device?
2. To what extent can these characteristics be extracted using the ESP32 SoC, and what system architecture would best support this?

6.1 Characteristic Identification

The identification of suitable characteristics focuses primarily on the lower layers of the network stack, specifically the Data Link Layer (as defined by the IEEE 802.11 WLAN standard) and the Network Layer (IP protocol), moving beyond the application layer focus of prior work by Michael Stöger [3].

The identification process involves reviewing existing literature on network fingerprinting techniques. This includes established methods like those used by NMAP [31] for OS detection, which capitalize on implementation differences in TCP/IP stacks, as well as techniques targeting the MAC layer as explored by Robyns et al. [32]. Particular attention is paid to characteristics that can be elicited through active probing—sending specifically crafted packets and analyzing the subsequent responses. Potential characteristics include variations in protocol implementations, response timings, supported features, and reactions to malformed or non-standard requests.

For instance, consider the TCP three-way handshake. While the basic sequence (SYN, SYN-ACK, ACK) is standardized, the exact values chosen for fields like the initial window size, or the specific TCP options included in the SYN and SYN-ACK packets (e.g., MSS, Window Scale, SACK Permitted, Timestamps) can vary between different TCP/IP stack implementations. An active probe could send a series of SYN packets with varying option combinations to a target device. As illustrated conceptually in Figure 6.1, Device A and Device B, despite both adhering to TCP, might respond with different sets of TCP options or window sizes, revealing underlying differences in their network stacks. These subtle variations, when systematically collected and analyzed, can form the basis of a distinguishing fingerprint.

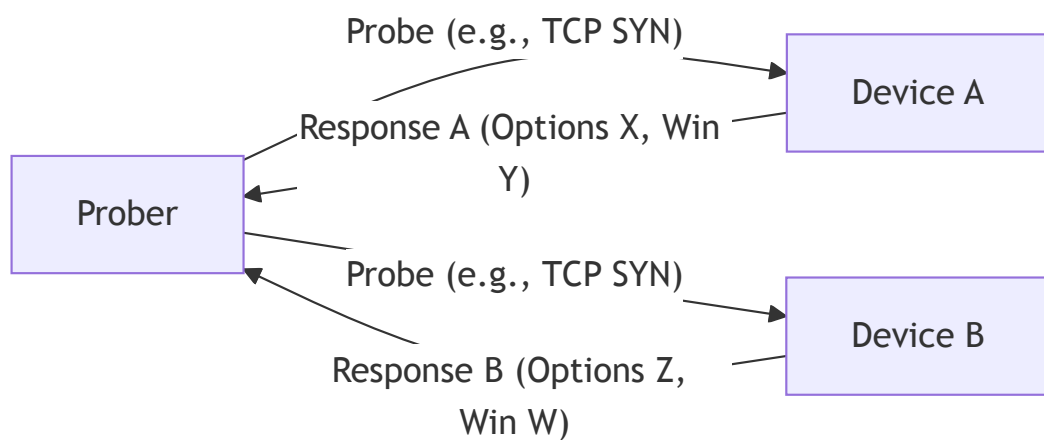


Figure 6.1: Conceptual example of active probing eliciting different responses. The prober sends an identical probe to Device A and Device B. Differences in their respective TCP/IP stack implementations lead to distinguishable responses (e.g., different TCP options or window sizes), which can be used for fingerprinting.

The literature described in chapter 5 presents a spectrum of fingerprinting methodologies, from highly specialized hardware-based techniques to computationally intensive ML models. Many of these studies focus on developing a novel classification algorithm or proving the efficacy of a single, specific feature set. This thesis takes a different approach: it does not aim to invent novel fingerprinting features, but rather to synthesize several well-established, fundamental techniques into a practical, integrated system.

6.2 System Design Philosophy

Addressing the second research question requires outlining a system capable of practically extracting and utilizing the identified characteristics. This section details the desired attributes and high-level architecture for such a system.

6.2.1 The ESP32 as the Dedicated Network Probing Unit

Effective fingerprinting, particularly active probing, necessitates direct interaction with network protocols at a low level. This includes the ability to craft and transmit arbitrary network packets (e.g., at the MAC and IP layers) and to capture raw network traffic. For this thesis, the ESP32 SoC is selected as the dedicated network probing unit, a choice made deliberately when contrasted with more general-purpose hosts like an Android device, as utilized in the work by Stöger [3]. While platforms such as Android offer broad functionality, they often lack native support for raw sockets, thereby restricting the ability to craft and send arbitrary TCP/IP packets. Furthermore, their Wi-Fi chipsets typically do not readily allow user-level access to promiscuous or monitor mode, which is necessary for capturing raw IEEE 802.11 MAC frames. In contrast, the ESP32, with its accessible hardware interfaces and supportive development frameworks like ESP-IDF, facilitates this essential direct packet manipulation and monitor mode configuration. Its low cost, integrated Wi-Fi capabilities, and sufficient processing power for these specialized network tasks make it a suitable platform for realizing a portable and flexible probing tool, aligning with the practical focus of this research to elicit and observe granular network characteristics.

However, the ESP32 is a resource-constrained microcontroller with inherent limitations in processing power, memory, and storage. These constraints make it challenging to handle extensive data analysis or complex user interfaces directly on the device. Therefore, a two-tier architecture is adopted. In this model, the ESP32 acts primarily as a specialized forwarding agent. Its main responsibilities are to execute low-level network operations as directed by a more powerful host system and to forward the raw or minimally processed results back. The host system then handles the more complex tasks of data analysis, fingerprint generation, database comparison, and user interaction. This division of labor leverages the strengths of each component: the ESP32's low-level network access and the host system's superior processing capabilities.

The data forwarding strategy from the ESP32 to the host system is a key consideration. A simple approach might involve forwarding all captured packets. However, to optimize data transmission and reduce the processing burden on the host, a more refined strategy is envisioned. The ESP32 can be designed to execute pre-defined "scanning primitives" or "probing patterns" (e.g., a sequence of TCP SYN probes, an ARP scan for a given subnet). These primitives would be triggered by commands from the host system, and the ESP32 would return structured results or summaries, rather than a raw stream of all network traffic. This approach aims to balance the need for detailed network data with the efficiency of the overall system. Figure 6.2 illustrates this conceptual architecture, with the ESP32 serving as the core of the Network Probing Unit.

By building the system on an accessible platform, this work addresses the practical engineering challenges of deploying a fingerprinting solution in a real-world, heterogeneous IoT environment, bridging the gap between theoretical research and practical network management and security.

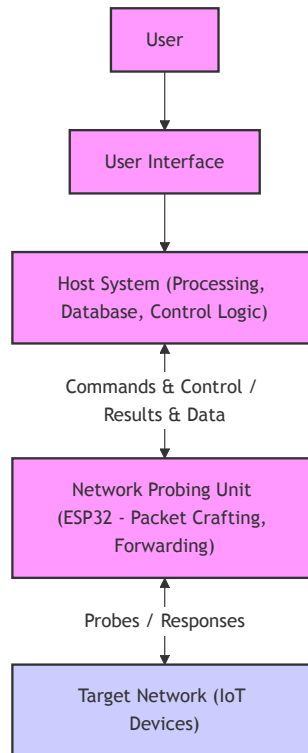


Figure 6.2: Conceptual overview of the proposed system architecture. The User interacts via a User Interface on a Host System, which controls the ESP32-based Network Probing Unit. The ESP32 interacts with devices on the Target Network.

6.2.2 User Interaction and System Control

To be broadly useful, the fingerprinting system should be accessible not only to network security experts but also to users with a general technical understanding. This necessitates an intuitive and effective user interface for managing the system and interpreting its results. The desired functional capabilities for such an interface include:

- **Configuration Management:** Allowing users to configure the network probing unit (e.g., its connection to the target network) and define parameters for scanning and fingerprinting tasks (e.g., target IP ranges, specific probes to use).
- **Task Orchestration:** Providing controls to initiate, monitor the progress of, and manage device discovery and fingerprinting operations.
- **Results Presentation:** Clearly displaying the list of discovered devices, their generated fingerprints, and any associated metadata.

- **Database Interaction:** Enabling the comparison of newly generated fingerprints against a stored database of known device fingerprints to facilitate identification.
- **Database Management:** Offering tools for users to view, manage, and contribute new or refined fingerprints to the database, thereby enhancing the system's knowledge base over time.

Furthermore, the system's design should ideally accommodate future enhancements and the integration of new fingerprinting techniques. A modular or plugin-based architecture for the control software on the host system is desirable. This would allow for the addition of new scanning tools or fingerprinting modules (e.g., for different protocols or device types) without requiring extensive modifications to the core system. This extensible architecture enables the system to leverage multiple fingerprinting layers to resolve ambiguities.

6.3 Evaluation Methodology

The evaluation aims to assess the practical applicability and effectiveness of the proposed multi-layered fingerprinting approach. The primary objectives are to determine the uniqueness, stability, reproducibility, and coverage of the generated fingerprints. The methodology involves deploying the system in a controlled network environment populated with a diverse set of IoT and consumer electronic devices. Data is collected over multiple sessions and under varying, yet controlled, conditions to ensure comprehensive analysis.

Key evaluation metrics are defined as follows:

- **Coverage:** The proportion of tested devices for which a meaningful and distinguishing fingerprint can be generated by each implemented fingerprinting method (WLAN, TCP/IP, Application Layer).
- **Uniqueness:** The ability of the fingerprinting schema to distinguish between different device models. This is quantified by analyzing fingerprint collision rates.
- **Stability:** The consistency of fingerprints generated for the same device over an extended period of time to determine long-term consistency.
- **Reproducibility:** The ability to obtain consistent fingerprinting results when the process is repeated in short timeframes.

The evaluation also considers the performance of the ESP32 firmware and the frontend application in terms of resource utilization and processing times. The outcomes of this evaluation process determine the extent to which the selected characteristics can be reliably extracted and used to form dependable fingerprints, thereby answering the second research question and assessing the practical viability of the proposed fingerprinting methodology and system design.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Fingerprinting Methods and Characteristics

Building upon the methodological framework outlined previously, which emphasizes the identification of reliable protocol-layer characteristics for IoT device fingerprinting, this chapter delves into the specific methods and characteristics selected for this thesis. The primary objective is to explore and define a set of network behaviors that can be extracted and combined to form a distinctive fingerprint for various IoT device models. This exploration spans multiple layers of the network stack, beginning with the WLAN (IEEE 802.11) MAC layer, progressing to the TCP/IP suite, and extending to application-layer interactions where necessary to achieve finer-grained differentiation.

The selection of these methods is guided by several factors, including their documented efficacy in existing literature, their potential to reveal unique implementation details across different devices, and critically, their feasibility for extraction using the ESP32 SoC as outlined in the second research question. This chapter will detail the rationale behind choosing specific characteristics within each protocol layer, discussing both active and passive collection techniques. It will further examine the challenges encountered, particularly in the context of active 802.11 probing with the ESP32, and the subsequent adaptations in methodology. Finally, it will define a comprehensive fingerprinting schema designed to systematically consolidate these diverse characteristics into a structured format, thereby laying the groundwork for the implementation and validation phases of this research.

7.1 Scope of Device Fingerprinting

Current literature on device fingerprinting defines the concept in various ways, often leading to confusion. Thus, the following clarifications are made to delineate the scope

of this work: Device fingerprinting in this thesis refers to characterizing IoT devices by their network communication patterns such that each fingerprint is indicative of a device model (and possibly its OS/firmware version) rather than an individually unique device instance. In other words, the goal is type identification – determining the device model or class – not assigning a unique ID to each physical device [2]. This scope aligns with the needs of security monitoring and inventory management: by matching an observed network fingerprint against a database of known device-model fingerprints, the system can recognize what types of devices are present on the network. This provides an inventory of device models and helps flag unexpected or potentially vulnerable device types, without attempting to uniquely distinguish every device of the same model.

7.1.1 Active vs. Passive Fingerprinting Methods

Two broad approaches exist for obtaining these network-based fingerprints: active and passive methods. In passive fingerprinting, an identifier is derived solely by eavesdropping on the device’s normal network traffic, with no direct interaction with the device [2]. The fingerprint is built from patterns the device naturally exhibits (e.g. typical packet sizes, timing, protocol headers) as it communicates. In active fingerprinting, the fingerprinting system actively probes the device by sending specially crafted packets or queries designed to elicit a distinctive response [2]. For example, an active probe might send an unusual TCP/IP packet sequence to the device and observe how its network stack responds, revealing traits characteristic of its implementation. Active methods can rapidly trigger devices to reveal identifying behaviors, whereas passive methods impose zero load and remain stealthy. This thesis prioritizes active fingerprinting techniques to ensure on-demand data for identification, while treating passive monitoring as a fallback if active probing is infeasible. In practice, active scans will be used to gather fingerprints when possible, and passive observation will be used only in cases where probes cannot be sent or do not yield results.

7.1.2 Network Protocols

The thesis focuses on fingerprinting devices over Wi-Fi (IEEE 802.11) and the IP network suite, but also, as described in section 7.4, by enhancing the fingerprinting accuracy through application-layer behaviors. While the Internet Protocol exists in two major versions, IPv4 and IPv6, this work will concentrate exclusively on fingerprinting techniques within the IPv4 context. This decision is grounded in the current state of consumer-grade IoT devices, the vast majority of which continue to operate predominantly over IPv4 in typical local network environments. Consequently, all subsequent discussions of IP and its related protocols refer to their IPv4 implementation unless explicitly stated otherwise. Other wireless interfaces such as Bluetooth or Zigbee are out of scope for this thesis, as they involve entirely different protocol stacks and would require separate fingerprinting techniques.

7.2 WLAN Characteristics

WLAN (IEEE 802.11) networks offer a rich source of characteristics for device fingerprinting, stemming from various aspects of their operation. These characteristics can be broadly categorized based on the layer of the 802.11 protocol stack from which they are derived, as well as the methods used to collect them. This section explores key WLAN characteristics pertinent to identifying IoT devices.

7.2.1 Radio-Frequency vs. MAC-Layer Fingerprinting in WLANs

Device fingerprinting in Wi-Fi networks can be achieved at different layers of the 802.11 protocol stack, notably at the physical radio-frequency (RF) layer or at the Medium Access Control (MAC) layer. Radio-frequency fingerprinting (also known as physical-layer fingerprinting) identifies devices based on unique analog signal characteristics caused by hardware imperfections [33]. Inherent variations in components (oscillators, amplifiers, etc.) introduce subtle differences in the transmitted RF waveforms. These differences, often stable over time, act as a fingerprint that can distinguish one device's transmissions from another's, even among devices of the same model. Prior work has shown that such RF imperfections (e.g., minute variations in signal phase noise, frequency offset, or modulation errors) can allow receivers to accurately identify the transmitting source [34]. As mentioned briefly before, the advantage of RF fingerprinting is its potential for identifying unique physical devices rather than just device types. However, RF-based techniques typically require specialized hardware and signal processing: raw radio signals must be captured (often with high sampling rates or custom radios) and analyzed to extract low-level signal features [32]. In the context of this thesis - which focuses on low-cost practical solutions - RF fingerprinting is largely infeasible. The ESP32's built-in Wi-Fi radio does not provide access to fine-grained analog waveform data, nor the high-fidelity sampling needed to capture hardware minutiae. Thus, while RF fingerprints are a powerful concept, they were not pursued for our implementation due to these hardware and practicality constraints.

In contrast, MAC-layer fingerprinting relies on differences in the way devices implement the 802.11 protocol and frame content, rather than on the analog signal characteristics. This is done by examining the fields and behaviors at the 802.11 data-link layer - for example, the sequence numbers, timing intervals between frames, supported protocol features, or Information Elements (IEs) in management frames. Because 802.11 standards leave certain implementation details to device manufacturers (for instance, how a device scans for networks or the exact composition of optional fields in a frame), different chipsets or drivers exhibit distinguishable patterns in their 802.11 traffic [32]. For instance, one early study [35] found that by passively analyzing the timing and rates of probe request frames, it was possible to fingerprint the wireless device driver in use on a device. Other works have identified differences in how various vendors order or populate the optional capabilities in their 802.11 frames, providing a basis for identification. The key benefit of MAC-layer techniques is that they can be performed with off-the-shelf Wi-Fi hardware

(e.g. a USB Wi-Fi adapter or microcontrollers with Wi-Fi capabilities such as the ESP32) capturing Wi-Fi frames, without needing specialized RF equipment [32]. Additionally, MAC-layer fingerprinting tends to reflect the device's make or chipset configuration rather than providing a unique identifier for an individual unit. For example, two IoT sensors of the same model (same Wi-Fi chipset and firmware version) will likely behave very similarly at the MAC layer [36]. This characteristic aligns well with the thesis's focus on per-device model fingerprinting.

In summary, RF fingerprinting can uniquely identify individual emitters but is impractical on low-cost platforms, whereas MAC-layer fingerprinting is feasible on devices like the ESP32 and can primarily help to differentiate device models, making the latter approach the focus of further research for our IoT device identification scheme.

7.2.2 Active Fingerprinting Approaches

The initial plan for this thesis included the investigation of active fingerprinting methodologies within the Wi-Fi (802.11) domain. A fundamental prerequisite for this was the technical feasibility of utilizing an ESP32 microcontroller platform for crafting and transmitting custom 802.11 frames, essential for active probing. While the standard ESP-IDF presents limitations due to its closed-source MAC layer components, the *ESP32 open Wi-Fi MAC* project [37] initially appeared promising. This open-source initiative offers a modified MAC layer implementation, potentially enabling direct transmission and reception of raw 802.11 frames. However, further investigation revealed that the project exhibited limitations at the time of this research. Its implementation was restricted to open Wi-Fi networks and did not reliably support the dual AP/STA Wi-Fi mode crucial for the system architecture detailed in chapter 8. These constraints hampered its utility for a real-world deployment.

Second, a review of existing literature indicated a relative scarcity of established and broadly effective active fingerprinting techniques specifically for 802.11 client devices. While some research, such as the work by Bratus et al. [38] on fingerprinting Access Points by sending non-standard or malformed frames, exists, applying these to diverse client stations has proven more challenging. Similarly, while Robyns et al. [32] explored Class 1 frames to instigate transmissions, the primary focus of their overall method remained passive, and the utility of such instigated responses for comprehensive fingerprinting was not definitively established for a wide range of client devices.

One specific avenue involving Class 1 frames, the ADDBA (Add Block Ack) Request, was briefly explored due to its potential for eliciting varied responses as noted by Robyns et al. [32]. Preliminary tests were conducted using the Python library Scapy [39] to send ADDBA Requests to a small set of available devices. While devices generally responded to these requests, the responses were largely homogeneous. The primary observed difference was in the `Number of buffer` value within the ADDBA response frames (see Figure 7.1). This limited variability suggested that, at least with the tested devices and this specific probe, the fingerprintable information might be minimal. To

ascertain the true potential of ADDBA responses for device fingerprinting, comprehensive novel tests involving a significantly larger and more diverse set of IoT devices would be necessary. Such extensive empirical investigation was beyond the practical scope of this thesis, particularly given the other identified challenges with active fingerprinting.

```

IEEE 802.11 Wireless Management
  ▾ Fixed parameters
    Category code: Block Ack (3)
    Action code: Add Block Ack Response (0x01)
    Dialog token: 0xd7
    Status code: Successful (0x0000)
  ▾ Block Ack Parameters: 0x1002, Block Ack Policy
    .... ..0 = A-MSDUs: Not Permitted
    .... ..1 = Block Ack Policy: Immediate Block Ack
    .... ..00 00.. = Traffic Identifier: 0x0
    0001 0000 00.. .... = Number of Buffers (1 Buffer = 2304 Bytes): 64
    Block Ack Timeout: 0x0000
  
```

(a) Response of a Google Pixel 7a Smartphone to an ADDBA request

```

IEEE 802.11 Wireless Management
  ▾ Fixed parameters
    Category code: Block Ack (3)
    Action code: Add Block Ack Response (0x01)
    Dialog token: 0xd7
    Status code: Successful (0x0000)
  ▾ Block Ack Parameters: 0x0102, Block Ack Policy
    .... ..0 = A-MSDUs: Not Permitted
    .... ..1 = Block Ack Policy: Immediate Block Ack
    .... ..00 00.. = Traffic Identifier: 0x0
    0000 0001 00.. .... = Number of Buffers (1 Buffer = 2304 Bytes): 4
    Block Ack Timeout: 0x0000
  
```

(b) Response of a Shelly smart powerplug to an ADDBA request

Figure 7.1: ADDBA responses as captured with Wireshark of Google Pixel 7a Smartphone and a Shelly smart powerplug

Therefore, due to the combined factors of platform limitations for generalized custom frame injection and the inconclusive nature of preliminary tests on broader active techniques, the primary focus of the WLAN fingerprinting methodology shifted. Instead of pursuing a wide array of active probes, the approach was narrowed to prioritize passive analysis of probe requests, augmented by a targeted active technique—deauthentication-based instigation—to improve data collection, as detailed in the implementation (section 8.2.1).

7.2.3 Passive Fingerprinting with Probe Requests

Among passive techniques, the analysis of WLAN probe request frames emerged as a particularly viable approach for this thesis. This decision is underpinned by two primary factors: first, the fingerprinting potential of information contained within probe requests is a well-documented and extensively researched area within the academic literature [33, 32, 40]. Second, capturing and dissecting these management frames is

readily implementable using an ESP32 microcontroller operating in promiscuous mode, aligning with the research question of leveraging low-cost, accessible hardware for device identification. Probe requests often contain a rich set of Information Elements (IEs) (as described in section 4.2.2) that can reveal vendor-specific configurations and capabilities, thereby offering a fertile ground for characteristic extraction. While probe requests are typically broadcast by client devices searching for known networks, Franklin et al.'s research [35] notes that most devices sent out periodic probe requests even if they are already associated and authenticated to an AP.

Information Elements Selection

Information Elements (IEs) within 802.11 Probe Request frames serve as a valuable source for passive device fingerprinting. To effectively leverage IEs, a systematic methodology is required to identify which elements and their constituent bits are most suitable for creating stable and unique device fingerprints.

The methodology employed for analyzing Information Elements, as detailed by Robyns et al. [32], involves quantifying two key characteristics for each bit within an IE: variability and stability. Variability is measured as the entropy of a bit across Probe Request frames from different devices, indicating its potential to uniquely distinguish devices. Stability, conversely, is measured as the inverse of entropy for a bit within frames from the same device, reflecting how consistently that bit's value remains over time. Robyns et al. applied this per-bit entropy analysis to a dataset of over 200,000 Probe Requests, including data from a music festival and a research lab, to empirically determine the suitability of various IE bits for fingerprinting. Their findings, summarized in Table 7.1, highlight the varying degrees of variability and instability across different IE types. Similarly, Matte [33] also conducted an analysis of IE entropy and stability across different datasets (Lab, Train station, and Sapienza), reinforcing the finding that IEs contain significant identifying information and that their characteristics vary in terms of stability and uniqueness.

A crucial consideration when selecting IEs for fingerprinting is the inherent trade-off between uniqueness (variability) and stability. As highlighted by Robyns et al., IEs or bits that are highly variable across devices contribute significantly to the uniqueness of a fingerprint. However, if these elements are not stable within a single device's transmissions over time, the resulting fingerprint will be unreliable for persistent identification. Conversely, highly stable IEs might be common across many devices, reducing their effectiveness in uniquely identifying a specific device model.

While IEs like SSID and Vendor Specific exhibit high variability, they also tend to have lower stability, making them less ideal for creating consistent device-model fingerprints. In contrast, capability-related IEs such as HT Capabilities, Extended Capabilities, and Supported Rates generally show higher stability. Matte's work [33] corroborates the high stability of many capability-related IEs and notes that elements like HT Capabilities are frequently present and contribute significantly to fingerprint entropy. He also points out

Information element	Σv_i	$\Sigma(1 - s_i)$
AP channel report	0.000	0.000
DS parameter set	0.625	0.411
Extended capabilities	32.790	0.061
Extended supported rates	28.373	1.716
HT capabilities	13.299	0.176
Information element order	40.327	2.529
Interworking	32.491	0.000
RSN information	0.000	0.000
SSID parameter set	87.570	30.590
Supported rates	22.529	1.317
VHT capabilities	10.424	0.000
Vendor specific	285.449	135.288

Table 7.1: Entropy and stability of Information Elements in Probe Request frames. The first column lists the Information Elements, while the second and third columns show the variability and stability values, respectively. The values are derived from Robyns et al.'s analysis [32].

that elements that are less stable over time typically stem from unstable values generated by a small subset of devices.

Based on this trade-off and the empirical findings from research in this area, the selection of Information Elements for device fingerprinting in this thesis focuses on those that demonstrate a favorable balance of variability and stability, and are commonly present in Probe Request frames. The selected IEs are:

- **Extended Capabilities, HT Capabilities, and VHT Capabilities:** These IEs provide details about the device's support for various IEEE 802.11 amendments and features, such as High Throughput (HT) and Very High Throughput (VHT). As highlighted by Robyns et al. [32], the capabilities announced by a device are dependent on its Wi-Fi chipset, leading to variations that are useful for fingerprinting and generally stable as they reflect intrinsic hardware capabilities.
- **Supported Rates and Extended Supported Rates:** These elements specify the data rates that the device is capable of transmitting and receiving. They are commonly included in Probe Requests as noted by Matte [33] and observed in preliminary testing. The elements exhibit sufficient variability and stability across devices to contribute to the fingerprinting scope of device-model fingerprinting.
- **Interworking:** This IE, defined in the IEEE 802.11u amendment, indicates a device's support for interworking with external networks, such as Hotspot 2.0. Its presence and content can serve as an identifying characteristic, particularly for devices supporting these advanced features.

- **Information Element Order:** Beyond the content of individual IEs, the order in which these elements appear within a management frame was also considered as a potential characteristic. While potentially less stable due to implementation variations, the sequence of IEs can provide distinguishing information, as noted by Robyns et al. [32]. However, as detailed in the evaluation (section 9.2.3), this characteristic was ultimately found to be too variable for reliable fingerprinting and was excluded from the final schema.

7.3 TCP/IP Characteristics

TCP/IP fingerprinting leverages the inherent variations in the implementation of the TCP/IP protocol suite across different operating systems and devices. While the core protocols are defined by standards (e.g., RFCs), manufacturers and developers often introduce subtle differences in how these standards are interpreted and implemented. These variations, though seemingly minor, can manifest in observable network traffic patterns that serve as distinctive fingerprints.

7.3.1 Methods and Outcomes from Literature

Various studies and tools have demonstrated the effectiveness of TCP/IP characteristics for device and operating system identification. Prominent examples from the literature reviewed include:

- **Nmap:** As a widely-used active network scanner, Nmap employs a sophisticated TCP/IP fingerprinting mechanism primarily for operating system detection [31]. Its approach involves sending a series of crafted probes, including six TCP SYN packets with variations in TCP options and window sizes, to open and closed ports. By analyzing the responses, Nmap extracts numerous attributes related to Initial Sequence Numbers (ISN), IP ID sequences, TCP options, window sizes, and TTL values. These attributes are combined to generate a fingerprint that is matched against a comprehensive database (`nmap-os-db`) to identify the target's operating system and version.
- **p0f:** In addition to Nmap, another notable open source project is p0f [41], which, unlike Nmap, creates a fingerprint based on passive network traffic. Fingerprints are created based on characteristics found in TCP (SYN and SYN+ACK packets) and HTTP network communication. The fingerprints are stored, quite similar to Nmap, in a textual form into a file. Analyzed header content includes various subtle implementation quirks in IP and TCP, such as window size and TCP timestamp progression. p0f's analysis provides insights into the target's OS, system uptime, network distance, and can detect network address translation (NAT) or load balancing.

- **SinFP:** SinFP is presented as a tool that unifies active and passive operating system fingerprinting for both IPv4 and IPv6 [42]. Its active method employs a small number of standard TCP probes (SYN without options, SYN with options, and SYN+ACK) sent to a single open port, analyzing the responses for various IP and TCP header fields such as TTL, ID, Don't Fragment bit, Sequence and Acknowledgment numbers, Flags, Window size, and Options (MSS, Timestamp). SinFP's passive mode analyzes captured packets and modifies them for compatibility with its active signature database. A key contribution is the introduction of deformation masks to handle variations in responses caused by intermediate devices or TCP/IP stack customizations. The primary outcome is operating system identification.
- **Yang et al. (2019):** As already mentioned in chapter 5 this research explores automatic fingerprinting of IoT devices by examining features across the network, transport, and application layers, utilizing neural networks for classification [43]. At the transport layer, they specifically also incorporate TCP window size (WIN) and TCP options (OPT) as significant features for differentiating devices.
- **Zhou et al. (2021):** Focusing on Industrial IoT (IIoT) devices, this work proposes a hybrid fingerprinting method that combines passive monitoring with active TCP SYN scans [44]. Their approach extracts TCP/IP header features, including Initial TTL, Window Size, Maximum Segment Size (MSS), Window Scaling Value, and Options Layout, alongside port-based features. Using machine learning classifiers (specifically Gradient Boosting), they demonstrate that these features are effective in identifying IIoT device types. Their findings highlight the continued relevance of these fundamental TCP/IP characteristics for fingerprinting even within specialized IoT domains.

7.3.2 TCP Options and Window Size

The TCP options and the TCP window size fields serve, due to their inherent flexibility and resulting diverse TCP/IP stack implementation, as the primary characteristics to be extracted.

This implementation diversity is empirically supported by the effectiveness of these characteristics in established fingerprinting tools and recent research. As demonstrated by widely-used tools like Nmap [31] and p0f [41], TCP options and window size are fundamental elements that exhibit significant variability across different TCP/IP stack implementations. The Nmap documentation explicitly states that the TCP options test provides “a veritable trove of information” [31] for fingerprinting. Similarly, regarding the TCP window size, it notes that this field “is quite effective, since there are more than 80 values that at least one OS is known to send” [31]. Furthermore, their relevance extends to the contemporary landscape of IoT and IIoT devices. Recent work by Yang et al. [43] and Zhou et al. [44] successfully employs TCP options and window size as

features in machine learning models for device identification, validating their continued discriminatory power in these domains.

An additional argument for the selection of these fields is their high entropy within the existing `nmap-os-db` database. As part of this research, an entropy analysis was conducted on the `nmap-os-db`, which reveals that parameters related to TCP options and window size exhibit particularly high entropy values. As shown in Table 7.2, several TCP option-related parameters (e.g., OPS:O4, OPS:O2, OPS:O1) and window size parameters (e.g., WIN:W2, WIN:W1, WIN:W6) are among the top parameters with the highest entropy. High entropy indicates a greater degree of variability in these fields across different fingerprints, making them highly informative for distinguishing between systems.

Parameter	Entropy
OPS:O4	8.1865
OPS:O2	8.1798
OPS:O1	8.1011
OPS:O5	8.0574
SEQ:SP	7.9112
OPS:O3	7.7484
ECN:O	7.4788
SEQ:ISR	7.3617
T3:O	6.5404
OPS:O6	6.1700
WIN:W2	5.6320
WIN:W1	5.5883
WIN:W6	5.5763
WIN:W5	5.5693
WIN:W3	5.5651
WIN:W4	5.5392
T3:W	5.3684
ECN:W	5.3243
SEQ:TS	3.2324

Table 7.2: Entropy Analysis of Top Parameters in `nmap-os-db`

7.3.3 Analogy to Nmap’s 6-Packet Sequence

The decision to employ a probing strategy involving a sequence of six TCP packets is driven by the objective of leveraging the already existing Nmap OS Database. Nmap’s OS detection process, as detailed in its documentation [31], relies on sending specific packets including six TCP SYN probes, each with chosen variations in TCP options and window sizes, to an open port on the target. The responses to these probes are then analyzed to generate a comprehensive fingerprint.

By mirroring this 6-packet sequence, the fingerprint data collected in this research can be directly compared or correlated with the extensive `nmap-os-db`. This database contains a vast collection of operating system fingerprints gathered over many years from a wide array of devices and systems. The fingerprints in `nmap-os-db` are structured based on the responses to Nmap’s specific probes, including the sequence of six TCP packets.

The primary advantage of this approach is the ability to potentially utilize the wealth of information contained within `nmap-os-db` to enrich the fingerprinting process. While the ultimate goal of this thesis is focused on IoT device identification rather than solely OS detection, the underlying OS and network stack play a significant role in a device’s network behavior. By generating fingerprints that are structurally compatible with Nmap’s format, it may be possible to:

- Infer the underlying operating system or network stack of an IoT device by finding matches or near-matches in `nmap-os-db`.
- Augment the collected IoT-specific fingerprints with known OS characteristics from the database.
- Leverage the continuous updates and community contributions to `nmap-os-db` to improve the accuracy and coverage of the fingerprinting process over time.

7.4 Application Layer Characteristics

Fingerprinting methodologies that rely solely on characteristics extracted from the WLAN (IEEE 802.11) and TCP/IP network stacks often provide insights into the underlying Wi-Fi chipset and the operating system’s network stack implementation, respectively. While this might at times be sufficient for identifying the device’s make and model, it may not always yield sufficient granularity to differentiate between devices that share similar hardware and firmware configurations. This observation is also noted by Yang et al. [43] and highlights a potential important limitation. Indeed, this phenomenon is observable within the validation phase of this research (see chapter 9), where distinct IoT devices, such as a door sensor and a temperature sensor, exhibit identical Wi-Fi and TCP/IP fingerprints. Both devices are built upon the Tuya stack (a well-known IoT-as-a-service provider) and share the same underlying hardware and firmware components.

Therefore, to achieve a more granular level of device identification that can expose specific model details and, in some cases, version information, it becomes advantageous to identify and fingerprint constant values (often human-readable strings) in the application layer. This approach aligns with techniques described in other literature, such as proposed by Feng et al. [45].

A notable downside to the application layer approach is its inherent specificity. Unlike the more agnostic fingerprinting approaches at the IEEE 802.11 and IP layers, which rely on standardized protocol behaviors and options, application layer fingerprints are

tightly coupled to the particular protocols and data formats used by specific applications or device ecosystems. This means that a fingerprint developed for one application layer protocol (e.g., Tuya’s proprietary protocol) is generally not applicable to devices using a different application layer protocol (e.g., MQTT, CoAP, or a vendor-specific HTTP API). This specificity necessitates developing distinct fingerprinting logic for each targeted application ecosystem.

To address this challenge and facilitate the future incorporation of diverse application layer fingerprinting techniques, the fingerprinting logic within this thesis is designed with a plugin-style architecture, as will be detailed further in chapter 8. This modular approach allows for the straightforward addition of new fingerprinting modules tailored to different application layer protocols without requiring extensive modifications to the core fingerprinting framework.

7.4.1 Tuya Application Layer Characteristics

As an initial demonstration of this application layer fingerprinting capability, this thesis will concentrate on Tuya-based devices. The prevalence of Tuya devices in the consumer IoT market makes them a relevant and impactful target for developing and validating application-specific fingerprinting techniques.

Tuya Wi-Fi devices use a proprietary application-layer protocol for local LAN communication. Unfortunately, the detailed specifications of this protocol are not publicly available, and the information provided here is based on reverse engineering efforts and community contributions, such as the `tinytuya` [46] and `localtuya` [47] projects. The protocol has evolved over time, with different versions (3.1, 3.3, 3.4, and 3.5) introducing various features and encryption methods.

However, the basic packet structure has been consistent across protocol versions (see Figure 7.2). Messages are wrapped with a fixed prefix and suffix (prior to version 3.5: 55aa at the start and aa55 at the end; starting with version 3.5: 6699 and 9966 [48]), followed by a sequence number, command byte, payload length, the payload itself, and a checksum or HMAC (for encrypted messages). Inside the frame, the payload consists of a command code and a JSON-formatted data section (often called the DP data) which represents device state or commands.

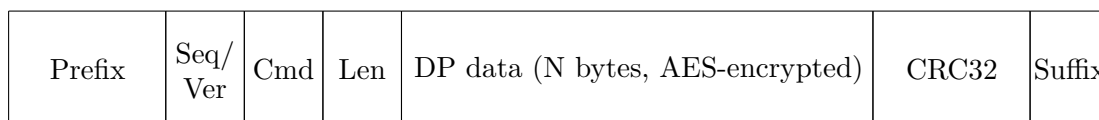


Figure 7.2: Diagram of a Tuya packet.

The communication pattern of Tuya devices as seen in the local area network can be categorized into two buckets:

- **Device Discovery:** Tuya devices periodically advertise themselves on the LAN via UDP broadcasts, which serve as a discovery/heartbeat mechanism to announce the device’s presence. These broadcasts contain a small UDP packet (to port 6666 or 6667) containing basic information about the device including device’s unique ID, its product model (product ID or “device type ID”), and the protocol version number. Notably, no actual state data is included in these broadcasts – the payload is strictly device metadata intended for discovery.
- **Command and Control:** Once a device is discovered, subsequent communication for status polling and control typically occurs over TCP. Tuya devices listen on TCP port 6668 for incoming connections. This communication involves the exchange of structured messages based on the Tuya proprietary protocol and varies significantly with the device’s protocol version.

A significant distinction between these two communication modes lies in their encryption. Device discovery packets, particularly in older protocol versions (e.g., v3.1), were often unencrypted or encrypted with a static, publicly known key. Even in later versions, the encryption applied to UDP discovery broadcasts is generally less complex, as their primary purpose is device announcement rather than secure data exchange. Consequently, these discovery packets can often be decrypted with relative ease, allowing for the identification of device parameters without requiring device-specific secrets.

In contrast, command and control communication over TCP, especially for protocol versions 3.3 and newer, employs more robust AES encryption for the payload. The decryption and encryption of these messages necessitate a device-specific `local_key`. This `local_key` is typically generated and assigned to the device during its initial registration and pairing process with the Tuya cloud infrastructure, usually facilitated through a companion mobile application. Therefore, to successfully intercept, decrypt, or issue valid local commands to a Tuya device, possession of its unique `local_key` is essential, effectively tying local control capabilities to a prerequisite of cloud registration.

This makes passive listening for UDP broadcast packets sent out by Tuya devices a more practical approach for large-scale fingerprinting. These broadcast packets are encrypted with a static, publicly known key, making their decryption feasible without the need for individual device keys.

An example of such a decrypted UDP broadcast message, as retrieved with the ‘scan’ module of the `tinytuya` [46] project, is shown below:

```
{ "ip": "192.168.1.13", "gwId": "bf82c872d3cb6bb9ukph",
  "active": 2, "ability": 0, "encrypt": true,
  "productKey": "keyhhtuxrsp7ueap", "version": "3.3" }
```

The message contains several key-value pairs that provide information about the Tuya device. Including the device’s current `ip` (IP address) on the local network and the `gwId`

(Gateway ID) which functions as a unique device identifier. Of particular significance for the thesis's scope of device-model fingerprinting is the `productKey` which, as noted in the Tuya documentation [49], is a unique identifier assigned to specific product model or a closely related batch of models. This inherent characteristic makes the `productKey` a stable and reliable indicator to be used as part of the overall fingerprint, increasing the granularity of device identification.

7.5 Putting it all together - Fingerprinting Schema

To effectively consolidate the diverse characteristics gathered from WLAN, TCP/IP, and application layers, a structured fingerprinting schema was developed. This schema dictates the format in which device fingerprints are stored and processed, ensuring consistency and enabling systematic comparison. The primary implementation of this schema is a textual database file, referred to as `iot-db` within the project, which serves as the repository for known device fingerprints.

The design of the `iot-db` format draws significant inspiration from the well-established `nmap-os-db` [31], which Nmap utilizes for its operating system detection capabilities. This inspiration is evident in several aspects of the schema:

- **Fingerprint Naming:** Each entry begins with a `Fingerprint` directive followed by a human-readable name or identifier for the device or device type (e.g., `Fingerprint LSC Ambient Light`).
- **Classification:** A `Class` directive provides a hierarchical classification for the device, including vendor, model, and sometimes firmware or OS details, along with a general device type (e.g., `Class IoT | Door sensor | Tuya`). This allows for grouping and broader identification.
- **TCP/IP Characteristics:** The schema incorporates `OPS` (TCP Options) and `WIN` (TCP Window Size) directives, which are directly analogous to Nmap's tests for TCP/IP stack fingerprinting. These capture the responses to a sequence of specifically crafted TCP probes.

A complete fingerprint entry in the `iot-db` typically comprises the following components:

```
Fingerprint <DeviceDescription>
Class <Vendor> | <Model> | <Firmware/OS Vers.> | <Category>
OPS(<TCP Options String>)
WIN(<TCP Window Sizes String>)
D11IE(<802.11 Information Elements String>)
TUYA(<Tuya Specific String>)
```

If a particular characteristic is not applicable to a device or could not be obtained during the fingerprinting process, it is often represented by R=N (e.g., TUYA (R=N)), indicating "Result Not Available".

The specific fingerprinting values within this schema are derived as follows:

- **OPS (TCP Options) and WIN (TCP Window Size):** These values are generated by actively probing the target device with a sequence of six TCP SYN packets, each with distinct TCP options and window size settings, mirroring Nmap's 'SEQ' probes as detailed in subsection 7.3.3.
 - The OPS string is a concatenation of results from the six probes, formatted as O1=<opts1>%O2=<opts2>%...%O6=<opts6>. Each <optsN> is a compact string representing the TCP options observed in the Nth response. This string is generated by parsing the hexadecimal TCP options field. The specific codes used to represent different options are as follows:
 - M<val>** Maximum Segment Size
 - W<val>** Window Scale
 - S** SACK Permitted
 - T<val1><val2>** Timestamp (indicating presence of TSval and TSecr)
 - N** No-Operation
 - L** End of Option List
 - The WIN string similarly concatenates window sizes from the six probes: W1=<size1>%W2=<size2>%...%W6=<size6>. Each <sizeN> is the hexadecimal representation of the TCP window size field from the Nth response.
- **D11IE (IEEE 802.11 Information Elements):** This characteristic is derived from passively captured IEEE 802.11 Probe Request frames transmitted by the device, as discussed in section 7.2.3. The string encapsulates several key Information Elements (IEs) and their values. While the order of IEs was initially considered, it was found to be too unstable for reliable fingerprinting (see section 9.2.3) and is therefore excluded from the final fingerprint string. The included IEs are:
 - EC=<hex_value>: Extended Capabilities (IE ID 127).
 - HC=<hex_value>: HT (High Throughput) Capabilities (IE ID 45).
 - IW=<hex_value>: Interworking (IE ID 107).
 - VC=<hex_value>: VHT (Very High Throughput) Capabilities (IE ID 191).
 - SR=<hex_value>: Supported Rates (IE ID 1).
 - ESR=<hex_value>: Extended Supported Rates (IE ID 50).
- **TUYA (Tuya Application Layer Data):** This fingerprint component is specific to devices utilizing the Tuya IoT platform, as described in subsection 7.4.1. It is

obtained by passively monitoring and decrypting UDP broadcast packets sent by Tuya devices.

- The primary value extracted is the `productKey`, a unique identifier assigned by Tuya to a specific product model or batch.
- The format is `PK="<productKey_value>"`.

This multi-faceted fingerprinting schema, by combining data from the MAC layer (802.11 IEs), the transport layer (TCP/IP options and window sizes), and the application layer (Tuya product key), aims to create a more distinctive and reliable signature for IoT devices. The schema is also designed with extensibility in mind, allowing for the future incorporation of additional fingerprinting characteristics as new methods are developed or other device ecosystems are targeted.

The practical effectiveness of this fingerprinting schema in identifying and differentiating various IoT devices is further analyzed and discussed in the evaluation chapter (chapter 9).

Implementation

Following the detailed exposition of fingerprinting methods and the fingerprinting schema established in the preceding chapter, this chapter transitions to the practical realization of the proposed IoT device identification system. The primary objective is to describe how the architectural philosophy outlined in the methodology is implemented. This includes the specific technologies chosen, the detailed architecture of the system components, and the core functionalities developed to achieve the desired device discovery and fingerprinting capabilities.

The first section will elaborate on the system architecture, detailing the ESP32 firmware and the host application designed to meet the user interaction and extensibility goals (subsection 6.2.2). Subsequently, this chapter will cover the specifics of the ESP32 firmware's API and low-level networking techniques, the host application's architecture including its user interface and plugin system for fingerprinting logic, and the concrete implementation of the fingerprinting modules derived from the methods discussed in chapter 7. This provides a comprehensive overview of the system's operational capabilities, laying the groundwork for the evaluation in the following results chapter.

8.1 System Architecture

The implemented system realizes the architecture proposed in subsection 6.2.1, featuring a decoupled, two-component design. This comprises an ESP32-based firmware module, serving as the dedicated network probing unit, and a host-based software application that provides user control, data processing, and visualization. This architectural choice is driven by the methodological goal of leveraging the ESP32's proficiency in low-level network interaction (acting as a forwarding agent) while offloading computationally intensive tasks, complex logic, user interface management, and data persistence to a more capable host computing platform.

The host application is implemented as a web-based frontend, fulfilling the user interaction and system control capabilities outlined in subsection 6.2.2. In more detail the functionalities include:

- Configuration and status monitoring of the ESP32 probing unit.
- Initiation and management of device discovery scans, including parameter definition (e.g., IP subnets, timeframes) and presentation of scan results.
- Orchestration of active and passive device fingerprinting processes.
- Display of fingerprinting outcomes, comparison against established databases (e.g., `nmap-os-db`, project-specific `iot-db`), and identification of matches.
- Tools for users to manage and contribute new entries to the `iot-db`.

As illustrated in Figure 8.1, the host application itself is structured into two main logical components to effectively meet these requirements:

1. A client-side User Interface (UI) developed using Vue.js. This choice facilitates the creation of a responsive and accessible UI, addressing the goal of usability for a broad range of technical users.
2. A server-side backend implemented as a Node.js web server. The Node.js backend is responsible for:
 - Managing persistent data, notably serving and updating the `iot-db` file.
 - Orchestrating and executing potentially long-running processes such as network discovery and device fingerprinting. The asynchronous, event-driven nature of Node.js is well-suited for these tasks, which may require extended operational periods not ideal for a purely client-side application.
 - Implementing the plugin-based architecture for fingerprinting methods (detailed in subsection 8.3.2), which directly addresses the extensibility goal from the methodology.

Communication between these components, and between the Node.js server and the ESP32, is primarily achieved through REST APIs using JSON as the data interchange format. For direct ESP32 configuration and status retrieval, the Vue.js client can communicate with the REST API exposed by the ESP32 firmware. For more complex operations like initiating and managing device discovery and fingerprinting tasks, the Vue.js client interacts with the Node.js server's REST API. The Node.js server, in turn, orchestrates these tasks by commanding the ESP32's REST API to perform low-level network operations (e.g., execute "scanning primitives") and retrieve the resulting data. This data is then processed by the Node.js server (e.g., by fingerprinting plugins) and made

available to the Vue.js client via the server's own API endpoints. This tiered approach ensures efficient background process control and data aggregation by the Node.js server, while the Vue.js client focuses on providing an interactive user experience.

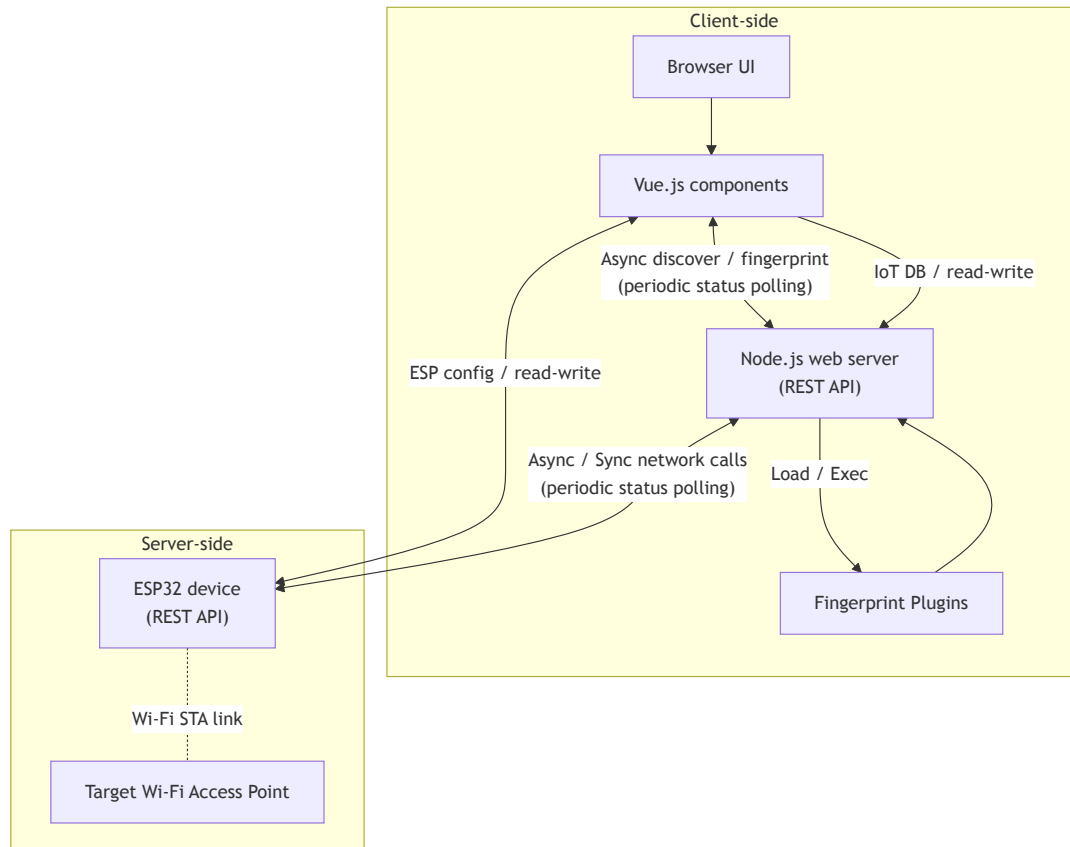


Figure 8.1: Diagram of basic architecture

8.2 ESP32 Firmware

The ESP32 firmware is designed to provide a lightweight network interaction layer for the frontend application. For this communication to work, the ESP32's Wi-Fi interface is configured to operate simultaneously in both STA and AP modes. The AP mode allows the frontend application, typically running on a separate computing device, to establish a direct Wi-Fi connection to the ESP32. This creates a dedicated local network segment for reliable command and control between the frontend and the firmware, independent of any existing infrastructure network. Concurrently, the STA mode enables the ESP32 to connect to the target Wi-Fi network where the network scanning and probing tasks are executed.

API Endpoints

The firmware exposes its functionalities through a set of RESTful API endpoints using JSON as the data format. The most significant endpoints are detailed below:

/esp (GET/POST) This subordinate endpoint provides essential status information and configuration options for the ESP32 and its network interfaces. This information is crucial for the frontend to monitor the ESP32's connectivity status and to display relevant network details to the user.

- The GET `/esp/info` endpoint returns the following data: IP address assigned to the STA interface, the SSID of the connected network and the Received Signal Strength Indicator (RSSI).
- The GET `/esp/wifiscan` endpoint returns the SSID and RSSI of all available Wi-Fi APs
- POST `/esp/wificonnect` is used to connect to a AP using the provided SSID and password

/arp (POST) Utilized for network discovery, this endpoint initiates an ARP scan across a specified IP subnet. The frontend sends a POST request containing the target subnet details. The ESP32 then sends ARP requests to each IP address within this subnet and collects the corresponding ARP responses. The endpoint responds with a list of discovered devices (IP and MAC address tuples).

/tcp (POST) This endpoint enables the transmission of custom TCP SYN packets. The POST request body includes the target IP addresses (either a list or a subnet range) and detailed specifications for the TCP header, including various flags and options. The ESP32 crafts and sends these packets and captures the responses, specifically looking for TCP packets with the SYN/ACK flags set. The collected response data is returned to the frontend for analysis and fingerprint generation.

/monitor/wifi (POST/GET) This endpoint manages passive monitoring of Wi-Fi traffic, specifically focusing on capturing IEEE 802.11 Probe Request frames.

- A POST request is used to initiate a Wi-Fi monitoring session for a defined duration, optionally filtering for probe requests from a specified list of MAC addresses.
- A subsequent GET request is used to retrieve the collected probe request data, including the source MAC address, timestamp, and the content of various IEs, after the monitoring period has concluded. This data is vital for passive 802.11 fingerprinting based on IE analysis.

/monitor/udp (POST/GET) This endpoint is dedicated to passive monitoring of UDP broadcast messages on the network.

- A `POST` request starts a UDP monitoring session, specifying the target IP addresses (or ranges), ports, and a timeout period.
- A `GET` request retrieves the captured UDP packets, including source IP, source port, and the packet payload, after the monitoring session has ended. This is particularly useful for capturing application-layer broadcast data, such as those used by Tuya devices, for fingerprinting purposes.

/mdns (POST) This endpoint triggers an active mDNS discovery process. The frontend sends a `POST` request specifying the service types to query (e.g., `_http._tcp.local`). The ESP32 sends out the corresponding mDNS queries and collects the responses from devices on the local network. The endpoint returns a list of discovered services, including their names, hostnames, IP addresses, ports, and any associated `TXT` records.

8.2.1 Implementation Details

This subsection delves into the specific technical details of key firmware components responsible for network interactions. Each of the following subsections describes the design choices, challenges, and solutions for a particular functionality.

Address Resolution Protocol (ARP) Scan Implementation

As elaborated in the subsequent subsection 8.3.1, ARP is a pivotal and effective method for host discovery within a given target network.

To support this mechanism, the lwIP library [50] — used by ESP-IDF as the backbone of its TCP/IP stack — exposes several ARP-related functions. Including access to the ARP table and sending/receiving ARP requests and responds. The ARP scan functionality, which is called through the `/arp` endpoint, is implemented by using lwIP's `etharp_query` function to broadcast ARP requests for all given IP addresses in a defined IP subnet range. All necessary low-level details of packet formatting and sending are then handled by the lwIP function itself.

The default ARP table size in the lwIP TCP/IP stack, which is limited to 10 entries, presents a notable constraint for device discovery, particularly when several hosts must be enumerated rapidly. This limitation is compounded by the ESP-IDF framework's lack of a configurable option for the ARP table size via `sdkconfig`. To mitigate the ARP table becoming a bottleneck for network scanning, a real-time ARP reply processing mechanism has been developed.

While the ESP-IDF provides an *ESP-NETIF L2 TAP Interface* for intercepting data-link layer frames on the fly, this feature is restricted to Ethernet interfaces and does not support WLAN connections [19]. To address this, a custom wrapper function was engineered and integrated into the lwIP stack's network input processing chain (see subsection 4.5.2 for details on lwIP). This wrapper captures incoming ARP responses, extracts the required information into a custom-managed array, and then passes the

packet to the original lwIP input handling routine (as shown in Listing 8.1). To ensure efficient ARP data collection while maintaining the responsiveness of the critical network input path, the wrapper incorporates conditional logic designed to minimize processing overhead for non-ARP traffic and offloads the storage of captured ARP responses to a worker thread using FreeRTOS Queues.

```

1 // Save the original input function and assign custom wrapper
2 original_input_fn = lwip_netif->input;
3 lwip_netif->input = netif_input_wrapper;
4
5 err_t netif_input_wrapper(struct pbuf *p, struct netif *inp)
6 {
7     // ... (quick checks if it's a packet that we want. returning to original
8     //     input function if not) ...
9
10    etharp_hdr_t *arphdr = (etharp_hdr_t *)((uint8_t *)ethhdr +
11        sizeof_ETH_HDR);
12    uint16_t opcode = arphdr->opcode;
13
14    arp_entry_t entry;
15    IPADDR_WORDALIGNED_COPY_TO_IP4_ADDR_T(&entry.ip_addr, &arphdr->sipaddr);
16    SMEMCPY(&entry.eth_addr, &arphdr->shwaddr, ETH_HWADDR_LEN);
17
18    // process ARP replies
19    if (opcode == PP_HTONS(ARP_REPLY)) {
20        xQueueSendToBack(arp_entry_queue, &entry, 0);
21    }
22
23    // process broadcast ARP announcements
24    else if (opcode == PP_HTONS(ARP_REQUEST) && ethhdr->dest.addr[0] == 0xFF)
25    {
26        // check if sender IP equals target IP (indicating ARP announcement)
27        if (ip4_addr_cmp(&sender_ip, &target_ip)) {
28            xQueueSendToBack(arp_entry_queue, &entry, 0);
29        }
30    }
31
32    return original_input_fn(p, inp);
33 }

```

Listing 8.1: Simplified Netif Input Wrapper.

Custom TCP SYN Packet Transmission

To implement the TCP SYN capabilities, the system utilizes the lwIP raw socket API. This API provides the necessary functions for constructing and transmitting custom TCP SYN packets on an individual basis.

The core of this packet crafting and transmission logic resides in the `send_tcp` function. Most of the IP header values are hardcoded, as they remain constant for the specific use

case. The TCP header largely derives its values from parameters received via the `/tcp` POST request. Within the `send_tcp` function, both the IP header checksum and the TCP checksum are computed and set using the checksum calculation functions provided by lwIP (through `inet_checksum.h`). Listing 8.2 shows a condensed excerpt of the packet construction and transmission logic.

Incoming raw IP packets are processed by a callback function. To correlate received responses with sent probes, a custom connection table (`conn_table`) is maintained containing a 4-tuple (source IP, source port, destination IP, destination port) similar to implementation found in operating systems.

A critical aspect of using lwIP's raw API for sending TCP-like packets, without engaging the full TCP state machine, is the management of Protocol Control Blocks (PCBs). When a raw PCB is created via `raw_new_ip_type` for sending a packet, it is not automatically deallocated by the lwIP stack if no corresponding "connection" is established or closed through standard TCP mechanisms. This can lead to PCB leakage and eventual resource exhaustion. To address this, a custom PCB management system was implemented:

- A linked list, `pcb_list`, is used to keep track of all active raw PCBs created by `send_tcp`. Each time a PCB is successfully used for sending, it is added to this list.
- When a valid SYN-ACK response is received and processed, it is scheduled for removal by using the `tcPIP_callback` function, which calls the passed callback method (for removing the `pcb`) safely within the lwIP's `tcPIP_thread` context. This prevents premature removal that would otherwise interfere with lwIP's internal processing.
- For probes that do not elicit a SYN-ACK response (e.g., packets sent to closed ports or non-responsive hosts), all remaining PCBs in the `pcb_list` are removed at the end of the TCP routine.

This explicit tracking and deferred removal strategy ensures that all allocated raw PCBs are properly deallocated, maintaining system stability during intensive scanning operations.

IEEE 802.11 Probe Request Monitoring and Instigation

The Wi-Fi monitoring functionality enables passive capture of IEEE 802.11 probe request frames for device fingerprinting based on Information Element (IE) analysis. This capability leverages the ESP32's ability to operate in promiscuous mode, allowing the capture and analysis of wireless frames that would otherwise be filtered by the network stack.

When monitoring is initiated, the ESP32's Wi-Fi interface is configured to capture all 802.11 management frames using `esp_wifi_set_promiscuous()` and a custom

```

1 static void send_tcp(in_addr_t targetip, tcp_hdr_ol_t* tcphdr_ol)
2 {
3     ip_hdr_t* iphdr = &(ip_hdr_t){0}; // Initialize IP header
4     // ... (Set source IP from ESP32's STA interface) ...
5     iphdr->dest.addr = targetip;
6
7     IPH_VHL_SET(iphdr, 4, 5); // Version 4, Header Length 5 (20 bytes)
8     IPH_TTL_SET(iphdr, 255);
9     IPH_PROTO_SET(iphdr, IPPROTO_TCP);
10    IPH_CHKSUM_SET(iphdr, 0); // Checksum calculated later
11    IPH_LEN_SET(iphdr, htons(IP4_HDRLEN + TCP_HDRLEN + tcphdr_ol->option_len)
12    );
13    IPH_CHKSUM_SET(iphdr, inet_chksum(iphdr, IP4_HDRLEN));
14
15    // ... (Add to custom connection tracking table: add_conn_entry) ...
16    // Calculate TCP checksum
17    struct pbuf* chk_buf = pbuf_alloc(PBUF_TRANSPORT, TCP_HDRLEN + tcphdr_ol
18    ->option_len, PBUF_RAM);
19    tcphdr_ol->hdr_with_o->base_hdr.chksum = 0;
20    memcpy(chk_buf->payload, tcphdr_ol->hdr_with_o, TCP_HDRLEN + tcphdr_ol->
21    option_len);
22    tcphdr_ol->hdr_with_o->base_hdr.chksum = ip_chksum_pseudo(chk_buf,
23    IPH_PROTO(iphdr), TCP_HDRLEN + tcphdr_ol->option_len,
24    &srcip_addr, &destip_addr); // srcip_addr, destip_addr are ip_addr_t
25    types
26    pbuf_free(chk_buf);
27
28    // Allocate pbuf for IP + TCP header + options
29    struct pbuf* buf = pbuf_alloc(PBUF_IP, IP4_HDRLEN + TCP_HDRLEN +
30    tcphdr_ol->option_len, PBUF_RAM);
31    memcpy(buf->payload, iphdr, IP4_HDRLEN);
32    memcpy(((uint8_t*)buf->payload) + IP4_HDRLEN, tcphdr_ol->hdr_with_o,
33    TCP_HDRLEN + tcphdr_ol->option_len);
34
35    LOCK_TCPIP_CORE();
36    struct raw_pcb *raw_ip_pcb = raw_new_ip_type(IPADDR_TYPE_V4, IPPROTO_TCP)
37    ;
38    raw_ip_pcb->flags |= RAW_FLAGS_HDRINCL; // Indicate header is included
39    raw_bind(raw_ip_pcb, &srcip_addr);
40    raw_recv(raw_ip_pcb, raw_ip_recv, NULL); // Set receive callback
41
42    esp_err_t send_err = raw_sendto_if_src(raw_ip_pcb, buf, &destip_addr,
43    lwip_netif, &srcip_addr);
44    UNLOCK_TCPIP_CORE();
45
46    if (send_err == ERR_OK) {
47        add_pcb_to_list(raw_ip_pcb); // Add to custom PCB tracking list
48    }
49    // ... (Error logging for send_err and buff freeing) ...
50 }

```

Listing 8.2: TCP Packet Crafting and Sending Excerpt.

callback function is registered via `esp_wifi_set_promiscuous_rx_cb()`. The promiscuous callback function, shown in Listing 8.3, implements frame filtering and processing logic. It verifies that the received frame is a management frame (`WIFI_PKT_MGMT`) and that the frame control field is of type probe request (type value = 0; subtype value = 4). The function then validates whether the source MAC address (`addr2`) matches any of the target MAC addresses specified via the `/monitor/wifi` POST request.

Once a valid probe request is identified, the system extracts the Information Element data, which contains the device-specific parameters used for fingerprinting. The IE section begins immediately after the 802.11 MAC header and extends to the FCS. The extraction process must account for the variable-length nature of probe request frames and properly calculate the IE data length by subtracting both the MAC header size and the 4-byte FCS from the total frame length indicated by `rx_ctrl.sig_len`.

The captured probe request data is stored in a structured format that includes the source MAC address, timestamp, and the complete IE data. To optimize memory usage and avoid redundant data, the system implements deduplication logic that updates existing entries when multiple probe requests are received from the same MAC address, retaining only the most recent capture.

Probe Request Instigation Mechanism To enhance the effectiveness of passive monitoring, particularly for devices that may not transmit probe requests spontaneously (as observed in chapter 9), the system implements an active instigation mechanism. This feature operates through a separate FreeRTOS task that periodically sends deauthentication frames to target devices, compelling them to disconnect from their current access point and subsequently transmit probe requests as they attempt to reconnect.

While the initial assessment in subsection 7.2.2 concluded that comprehensive active fingerprinting was impractical due to ESP-IDF limitations and the constraints of available solutions such as the ESP32 open Wi-Fi MAC project, subsequent investigation revealed an alternative approach for limited raw 802.11 frame transmission. This discovery occurred during the later stages of implementation when exploring probe request instigation mechanisms. Unlike the ESP32 open Wi-Fi MAC project, which required replacing the entire MAC layer implementation, this approach leverages the existing ESP-IDF framework's `esp_wifi_80211_tx` function for frame transmission.

However using this function requires circumventing ESP-IDF's built-in restrictions on Wi-Fi packet processing. The ESP-IDF framework includes sanity checks that prevent direct manipulation of certain Wi-Fi frames, particularly those used for security-sensitive operations. To bypass these limitations, the build configuration employs the linker flag `-Wl,-zmuldefs`, which allows multiple symbol definitions and prevents linker errors when overriding framework functions. Additionally, a dummy implementation of the `ieee80211_raw_frame_sanity_check` function is provided in the main application code, effectively disabling the framework's sanity checks by returning a success value (0)

```

1 static void wifi_promiscuous_cb(void *buf, wifi_promiscuous_pkt_type_t type)
2 {
3     if (type != WIFI_PKT_MGMT) return;
4
5     const wifi_promiscuous_pkt_t *ppkt = (wifi_promiscuous_pkt_t *)buf;
6     const wifi_ieee80211_packet_t *ipkt = (wifi_ieee80211_packet_t *)ppkt->
7     payload;
8     const wifi_ieee80211_mac_hdr_t *hdr = &ipkt->hdr;
9
10    // Check if it's a probe request (type=0, subtype=4) by masking with 0xFC
11    if ((hdr->frame_ctrl & 0xFC) != 0x40) return;
12
13    // Check if the source MAC matches any of our target MACs
14    bool mac_match = false;
15    int mac_index = -1;
16    for (int i = 0; i < g_state.config.num_macs; i++) {
17        if (memcmp(hdr->addr2, g_state.config.mac_addresses[i], 6) == 0) {
18            mac_match = true;
19            mac_index = i;
20            break;
21        }
22    }
23
24    if (!mac_match) return;
25
26    // Mark this MAC as received for early termination logic
27    if (mac_index >= 0) {
28        g_state.mac_received[mac_index] = true;
29    }
30
31    // Extract Information Elements from probe request
32    const uint8_t *ie_start = (const uint8_t *)ppkt->payload + sizeof(
33    wifi_ieee80211_mac_hdr_t);
34    size_t ie_len = 0;
35    if (ppkt->rx_ctrl.sig_len > sizeof(wifi_ieee80211_mac_hdr_t) + FCS_LEN) {
36        ie_len = ppkt->rx_ctrl.sig_len - sizeof(wifi_ieee80211_mac_hdr_t) -
37        FCS_LEN;
38    }
39
40    // Store probe request data with deduplication logic
41    // ... (packet storage and management code) ...
42 }

```

Listing 8.3: Wi-Fi Promiscuous Callback Function for Probe Request Processing.

for all frame validation attempts. This approach, which was adapted from the esp32-wifi-penetration-tool project [51], enables the transmission of custom deauthentication frames while maintaining compatibility with the ESP-IDF build system.

```

1 typedef struct __attribute__((packed)) {
2     wifi_ieee80211_mac_hdr_t hdr;
3     uint16_t             reason_code;
4 } wifi_deauth_frame_t;
5
6 wifi_deauth_frame_t deauth;
7 memset(&deauth, 0, sizeof(deauth));
8
9 deauth.hdr.frame_ctrl = 0xC0; // Deauthentication frame
10 deauth.hdr.duration_id = 0;
11 memcpy(deauth.hdr.addr1, target_mac, 6); // Destination MAC
12 memcpy(deauth.hdr.addr2, connected_ap_mac, 6); // Source (spoofed AP MAC)
13 memcpy(deauth.hdr.addr3, connected_ap_mac, 6); // BSSID
14 deauth.hdr.sequence_ctrl = 0;
15 deauth.reason_code = 0x0002; // Previous authentication no longer valid
16
17 esp_err_t tx_ret = esp_wifi_80211_tx(WIFI_IF_STA, &deauth, sizeof(
    wifi_deauth_frame_t), false);

```

Listing 8.4: Deauthentication Frame Construction for Probe Request Instigation.

The deauthentication frame is constructed with the frame control field set to 0xC0, indicating a management frame of subtype deauthentication. The frame spoofs the MAC address of the access point to which the ESP32's STA interface is connected, obtained via `esp_wifi_sta_get_ap_info()`. This approach leverages the fact that devices typically respond to deauthentication frames by immediately attempting to reconnect, during which they transmit probe requests containing their characteristic IE patterns.

UDP Broadcast Message Capture

The UDP monitoring functionality enables passive capture of UDP broadcast messages transmitted by IoT devices, particularly targeting application-layer protocols that utilize broadcast communication for device discovery or status announcements. This capability is essential for application-layer fingerprinting, as described in section 7.4, where protocol-specific characteristics can provide granular device identification beyond what is achievable through network and transport layer analysis alone.

The implementation utilizes standard BSD socket programming interfaces provided by the lwIP TCP/IP stack within ESP-IDF (see subsection 4.5.2 for more detail). When monitoring is initiated via the `/monitor/udp` POST endpoint, the system creates a UDP socket bound to a port specified using the REST endpoint and configures it to listen for incoming packets from a predefined list of target IP addresses. The monitoring process operates within a dedicated FreeRTOS task to prevent blocking the main application thread during potentially long-running capture sessions.

The core monitoring logic, implemented in the `monitor_task` function, employs a polling approach with configurable timeout intervals. As shown in Listing 8.5, the task continuously attempts to receive packets using `recvfrom()`, filtering incoming traffic to retain only packets originating from the specified target IP addresses. To prevent indefinite blocking, the socket is configured with a 100-millisecond timeout using `SO_RCVTIMEO`, allowing the monitoring loop to periodically check for termination conditions.

```

1 static void monitor_task(void *pvParameters) {
2     uint8_t buffer[1500]; // Standard MTU size
3     struct sockaddr_in src_addr;
4     socklen_t src_addr_len = sizeof(src_addr);
5     int64_t start_time = esp_timer_get_time();
6
7     while ((esp_timer_get_time() - start_time) < (monitor_state.config.
8         timeout_ms * 1000LL)) {
9         ssize_t len = recvfrom(monitor_state.socket_fd, buffer,
10             sizeof(buffer), 0, (struct sockaddr *)&src_addr, &src_addr_len);
11
12         // ... (filtering and error handling) ...
13
14         esp_err_t ret = udp_monitor_process_packet(buffer, len,
15             src_addr.sin_addr.s_addr, ip_index, ntohs(src_addr.sin_port));
16     }
17     // ... (cleanup) ...
18 }

```

Listing 8.5: UDP Monitoring Task Implementation.

Multicast DNS (mDNS) Service Discovery

The mDNS probing functionality implements active multicast DNS service discovery to identify and enumerate network services advertised by IoT devices. The implementation leverages ESP-IDF's built-in mDNS library while providing a structured interface for service discovery and data extraction.

The mDNS service discovery logic is encapsulated within the `query_mdns_services` function (Listing 8.6). This function supports two primary modes of operation: targeted queries for specific service types (e.g., `_http._tcp`) and a comprehensive meta-service discovery. In the meta-service discovery mode, an initial query is dispatched to the standard DNS-SD meta-service, `_services._dns-sd._udp`, to enumerate all service types advertised on the local network. Subsequently, for each unique service type discovered through this meta-query, the system initiates further mDNS queries to find specific instances of that service.

Discovered service data is extracted and stored systematically. The `process_mdns_service` function parses mDNS results, populating `mdns_service_t` structures with details such as service names, types, hostnames, IP addresses, ports, and TXT records.

```

1 static void query_mdns_services(const char* name, const char* service_type,
2     const char* proto, char queried_services[][32], size_t* queried_count,
3     size_t max_queried) {
4     // Meta-service enumeration for comprehensive discovery
5     if (name && !strcmp(name, "_services") &&
6         !strcmp(service_type, "_dns-sd") && !strcmp(proto, "_udp")) {
7         mdns_query_generic("_services", "_dns-sd", "_udp", MDNS_TYPE_PTR,
8             MDNS_QUERY_MULTICAST, 1000, 20, &results);
9         while (results) {
10            if (results->service_type) {
11                // Query for instances of discovered service type
12                mdns_query_ptr(results->service_type, results->proto, 500,
13                    10, &instances);
14                while (i) {
15                    process_mdns_service(i);
16                    i = i->next;
17                }
18                results = results->next;
19            }
20            // "Simple" service type queries
21        } else if (service_type && proto) {
22            mdns_query_ptr(service_type, proto, 500, 10, &results);
23            const mdns_result_t *i = results;
24            while (i) {
25                process_mdns_service(i);
26                i = i->next;
27            }
28        }
29    }

```

Listing 8.6: Simplified mDNS Service Query Logic.

TXT records are specifically processed to extract and store key-value pairs, preserving both keys and values. IPv4 addresses are extracted by iterating through the linked list of IP addresses provided by the mDNS library, ensuring that discovered services are linked to their respective host devices.

8.3 Frontend Web Application

The frontend web application serves as the primary user interface and control center for the device fingerprinting system. It is designed to abstract the complexities of low-level network interactions handled by the ESP32 firmware, providing an accessible platform for configuring the system, initiating network scans, orchestrating the fingerprinting process, and managing the resulting data. As outlined in the system architecture (section 8.1), the frontend is a composite application, consisting of a reactive Vue.js client for user

interaction and a Node.js server backend. The Node.js server component is pivotal for managing persistent data, such as the `iot-db`, and for coordinating the execution of potentially long-running tasks like device discovery and multi-stage fingerprinting operations.

8.3.1 Host Discovery

Host discovery is an essential preliminary phase in the network analysis workflow. Its primary objective is to identify devices within a specified network segment, creating a foundational map of reachable targets. The frontend application implements host discovery by orchestrating ARP and TCP SYN scans, leveraging the ESP32's low-level networking capabilities.

The Address Resolution Protocol (ARP) scan is a effective technique for discovering hosts on a local network segment. An ARP scan involves broadcasting ARP request packets for each IP address within a target subnet. Devices that are active and hold those IP addresses will respond with an ARP reply containing their MAC address. This method is particularly useful for host discovery within a local broadcast domain because it operates at Layer 2, bypassing potential Layer 3 firewalls that might block higher-layer probes such as ICMP or TCP [52]. Furthermore, it directly yields the MAC address, which is a critical identifier for device fingerprinting and subsequent analysis. ARP scans are only performed if the target network is on the same subnet as the ESP32's STA interface, as ARP is a link-local protocol and its requests do not traverse router boundaries [8].

Complementing ARP scans, TCP SYN scans are employed to identify open TCP ports on discovered hosts. This technique is a fundamental component of network scanners like Nmap [31]. A TCP SYN scan involves sending a TCP packet with only the SYN flag set to selected ports on a target IP address. If a port is open and a service is listening, the target system will respond with a TCP packet with both the SYN and ACK flags set (a SYN/ACK packet). This partial handshake allows for port state determination without establishing a full connection, reducing network overhead and logging footprint [52]. Conversely, a RST (reset) packet typically indicates a closed port, while no response often suggests a firewall is in place, dropping the packet. If the scan targets a network on the same subnet as the ESP32, TCP probes are directed only towards IP addresses previously identified by the ARP scan, optimizing the process. If the target network is on a different subnet, the ARP scan is skipped, and the TCP SYN scan will target the entire specified IP range.

The combined host discovery flow, involving both ARP and TCP SYN scans, is illustrated in Figure 8.2. The overall host discovery process is managed as a job by the Node.js server. When a user initiates a discovery task from the Vue.js frontend, they specify the target network, subnet mask, TCP ports to scan, and a duration. The server then enters a loop, repeatedly executing cycles of ARP (if applicable) and TCP SYN scans via the ESP32 for the specified duration. Results from each scan cycle, containing newly found hosts or updated information (like additional open TCP ports for an already discovered

host), are merged into a comprehensive list. This iterative approach allows the system to gradually build a more complete picture of the active devices on the network. The accumulated list of discovered hosts, including their IP addresses, MAC addresses (if found via ARP), and open TCP ports, is then made available to the frontend for display and for subsequent fingerprinting operations.

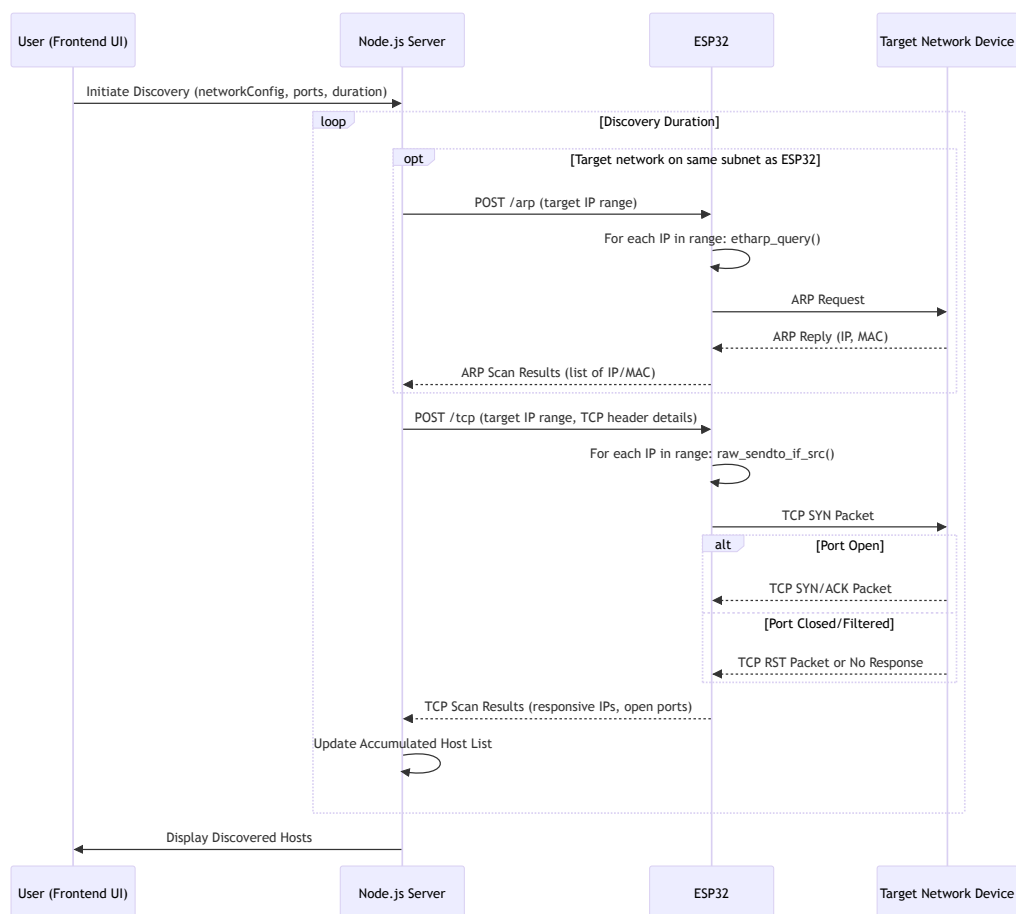


Figure 8.2: Combined Host Discovery Interaction Sequence (ARP and TCP SYN)

8.3.2 Frontend Fingerprinting Plugin Logic

Beyond host discovery, the frontend's Node.js server component is responsible for orchestrating the device fingerprinting process. To accommodate diverse fingerprinting methodologies and ensure extensibility, a plugin-based architecture is employed. This design allows for the modular inclusion of different fingerprinting techniques, each encapsulated as a self-contained plugin. The primary motivation for this approach is to simplify the addition of new methods, particularly for application-layer protocols which

vary significantly between IoT ecosystems, and to maintain a clear separation of concerns within the fingerprinting logic.

Plugins are located in the `fingerprint_methods` directory within the frontend's server-side codebase. A `manifest.json` file in this directory lists the JavaScript files that constitute the available plugins. Upon startup, the Node.js server reads this manifest and dynamically loads each specified plugin module using Node.js's `require()` mechanism. Listing 8.7 illustrates a conceptual excerpt of this dynamic loading process.

```

1 // Simplified excerpt from server.js
2 const fs = require('fs').promises;
3 const path = require('path');
4 const fingerprintingPlugins = [];
5
6 async function loadFingerprintingPlugins() {
7   try {
8     const manifestPath = path.join(process.cwd(), 'fingerprint_methods', '
9       manifest.json');
10    const manifestData = await fs.readFile(manifestPath, 'utf8');
11    const manifest = JSON.parse(manifestData);
12
13    for (const pluginFile of manifest.fpmethods) {
14      try {
15        const pluginPath = path.join(process.cwd(), 'fingerprint_methods',
16          pluginFile);
17        // Dynamically load the plugin module
18        const plugin = require(pluginPath);
19        fingerprintingPlugins.push(plugin);
20
21        // Optionally call a preload function if the plugin defines one
22        if (typeof plugin.preload === 'function') {
23          await plugin.preload();
24        }
25      } catch (err) {
26        console.error(`Failed to load plugin: ${pluginFile}`, err);
27      }
28    } catch (err) {
29      console.error('Failed to load plugin manifest:', err);
30    }
31  }

```

Listing 8.7: Conceptual Example of Dynamic Plugin Loading in Node.js Server.

Each loaded plugin is expected to export a common interface, typically including a name string (e.g., "TCP Fingerprinting"), an optional asynchronous `preload` function for one-time initializations (such as loading external database files like `nmap-os-db` or `nmap-mac-prefixes`), and a core asynchronous `method` function. This method function accepts the list of discovered hosts, the base URL of the ESP32's API, a timeout duration, and an optional flag for instigating probe requests (used by the Wi-Fi plugin).

When a fingerprinting task is initiated by the user through the Vue.js interface, the Node.js server invokes the `method` function of each loaded plugin. These methods then execute their specific fingerprinting logic. This often involves making HTTP requests to the ESP32's REST API to trigger network actions (e.g., sending TCP SYN packets via `/tcp`, capturing 802.11 Probe Requests via `/monitor/wifi`, or sniffing UDP broadcasts via `/monitor/udp`) and retrieving the results. Some plugins, like the MAC vendor lookup, perform their operations locally using preloaded data.

A key aspect of the plugin design is the structure of the data returned by each plugin's `method` function. For each host, a plugin returns an object (or an array of objects if a single plugin generates multiple pieces of information) containing:

- `hostId`: A unique identifier for the host.
- `type`: A string describing the nature of the information (e.g., "TCP Fingerprint", "Wi-Fi IE", "Tuya Product Key", "MAC Vendor Lookup", "mDNS Service").
- `fingerprint`: A string containing the primary characteristic(s) extracted by the plugin. This data is typically formatted to align with the fingerprinting schema defined in section 7.5 and is intended for storage in the `iot-db` (e.g., `OPS(...)` and `WIN(...)` from the TCP plugin, `D11IE(...)` from the Wi-Fi plugin, or `TUYA(PK="...")` from the Tuya plugin).
- `other`: A string field used to convey supplementary information that, while not part of the core fingerprint, provides valuable device context. This allows plugins to return richer details for display or further analysis. Examples include potential OS matches identified by the TCP fingerprinting plugin by comparing results against the `nmap-os-db` (e.g., "OS Matches: Linux 2.6.x"), the device vendor derived from its MAC address (e.g., "MAC Vendor: Espressif Inc."), or details of discovered mDNS services (e.g., "Service: `_http._tcp`, Host: `my-iot-device.local`, Port: 80").

This dual-member approach for returning results (`fingerprint` and `other`) enables the system to systematically collect structured data for fingerprint matching while also capturing diverse, human-readable information that enhances device identification and understanding. The Node.js server aggregates these results from all plugins for each host before making them available to the Vue.js client for display.

8.3.3 Implemented Fingerprinting Modules

The plugin-based architecture facilitates the integration of several distinct fingerprinting modules, each targeting different network characteristics. These modules are executed by the Node.js server, which orchestrates their operations, including interactions with the ESP32's API endpoints. The following modules form the core of the device fingerprinting capabilities:

MAC Vendor Lookup Plugin

This plugin determines the manufacturer of a device's NIC by examining its MAC address. It utilizes a locally stored database, derived from the `nmap-mac-prefixes` file, which contains mappings of Organizationally Unique Identifiers (OUIs) – the first three bytes of a MAC address – to vendor names. Upon receiving a list of discovered hosts, the plugin extracts the MAC address of each host, identifies the Organizationally Unique Identifier (OUI), and queries its internal database for the corresponding manufacturer. This information, while not part of the formal fingerprint string in `iot-db`, is provided in the supplementary `other` field (e.g., "MAC Vendor: Espressif Inc.") and offers valuable contextual data that can aid in the preliminary identification or classification of a device.

mDNS Service Discovery Plugin

The mDNS Service Discovery plugin actively queries the local network for services advertised using multicast DNS. It instructs the ESP32, via the `/mdns` API endpoint, to send out mDNS queries for a predefined list of common service types (e.g., `_http._tcp`, `_printer._tcp`, `_services._dns-sd._udp`). The ESP32 collects responses and forwards them to the plugin. The plugin then parses these responses, extracting service names, hostnames, IP addresses, ports, and particularly, information from TXT records. TXT records often contain human-readable details about the device, such as its model, manufacturer, or specific capabilities. This module helps in understanding the potential roles and functions of a device on the network. The collated service information and extracted device details are presented in the `other` field, providing a richer profile of the discovered device.

TCP Fingerprinting Plugin

This plugin aims to identify the underlying operating system or TCP/IP stack implementation of a target device. It emulates the TCP sequence probing technique used by Nmap for OS detection, as detailed in subsection 7.3.3. The plugin orchestrates a series of six TCP SYN packets sent to an open port on the target device via the ESP32's `/tcp` endpoint. Each packet in this sequence has distinct TCP options and window size settings, designed to elicit varied responses based on the target's TCP/IP stack peculiarities. The plugin parses the TCP options and window sizes from the responses, formatting them into `OPS(...)` and `WIN(...)` strings according to the schema defined in section 7.5. These strings constitute the `fingerprint` value for this plugin. Additionally, the plugin compares this generated fingerprint against a loaded version of the `nmap-os-db`. If a match is found, potential OS identifications are provided in the `other` field (e.g., "OS Matches: Linux 2.6.x").

Tuya Application Layer Plugin

Tailored for identifying IoT devices based on the Tuya platform, this plugin focuses on application-layer characteristics, as discussed in subsection 7.4.1. It instructs the ESP32,

via the `/monitor/udp` API endpoint, to passively listen for UDP broadcast messages on port 6667. Tuya devices commonly use this port and protocol for local discovery and status announcements. The plugin retrieves captured packets from the ESP32 and attempts to decrypt their payloads using a known, static decryption key common to these Tuya broadcasts. If successful, it parses the decrypted JSON data to extract the `productKey`. This `productKey` is a Tuya-specific identifier for a product model or batch. The extracted key is then formatted into the `TUYA(PK=<productKey>)` string and stored as the plugin's `fingerprint` contribution, offering a granular means of identifying specific Tuya device models.

Wi-Fi Fingerprinting Plugin

This module performs passive fingerprinting based on the Information Elements (IEs) present in IEEE 802.11 Probe Request frames, aligning with the methodology described in section 7.2.3. The plugin directs the ESP32, through the `/monitor/wifi` API endpoint, to enter promiscuous mode and capture Probe Request frames originating from the MAC addresses of discovered hosts. It can also optionally instruct the ESP32 to attempt to instigate the transmission of Probe Requests by sending deauthentication frames to target devices. Upon receiving captured frame data, the plugin parses the IEs, specifically extracting values for Extended Capabilities, HT Capabilities, VHT Capabilities, Interworking, Supported Rates, and Extended Supported Rates. It also records the order in which all IEs appear in the frame. These extracted values are then compiled into the `D11IE(...)` string format (e.g., `D11IE(EC=...%HC=...%IO=...)`) and stored as the `fingerprint` output.

Frontend User Interface

The frontend user interface (UI), built with Vue.js, provides a centralized dashboard for interacting with the fingerprinting system. It is organized into three main views, corresponding to the core operational workflow: Configuration, Host Discovery, and Device Fingerprinting.

The **Configuration** view (Figure 8.3) allows the user to set the IP address of the ESP32 module and manage its Wi-Fi connectivity. It displays the current Wi-Fi status (SSID, IP address) and allows scanning for available networks and connecting to a selected one, with an option to provide a password if required. This view also provides a function to clear locally cached data, such as previously discovered hosts and their fingerprints.

The **Host Discovery** view (Figure 8.4) is where users initiate network scans to find active devices. Key adjustable settings include the target IP network and subnet mask (which default to the ESP32's current network settings if available), a comma-separated list of TCP ports to probe (e.g., "80,443,8080"), and the overall duration for the discovery task in minutes. Once the discovery process is complete, this view presents a table of discovered hosts, listing their IP addresses, MAC addresses (if found), the method of discovery (e.g., ARP, TCP SYN), and any open TCP ports identified.

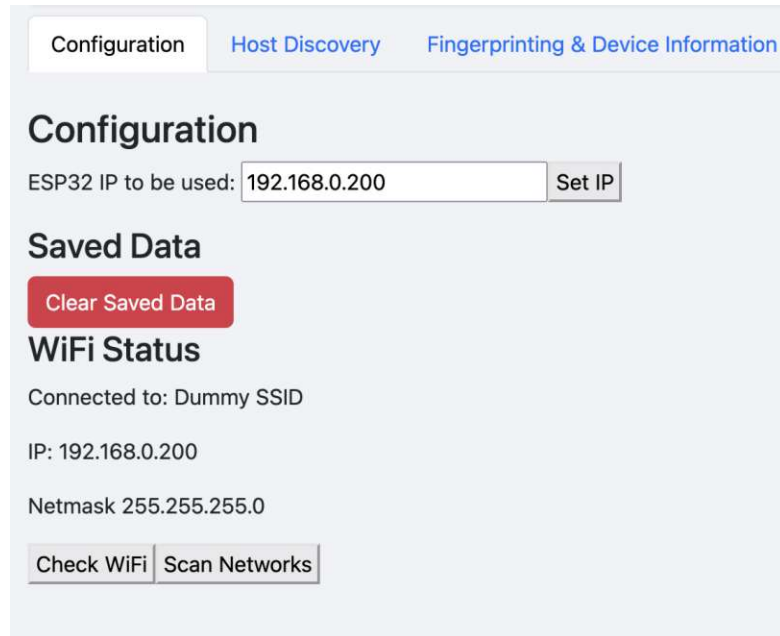


Figure 8.3: Frontend UI: Configuration View.

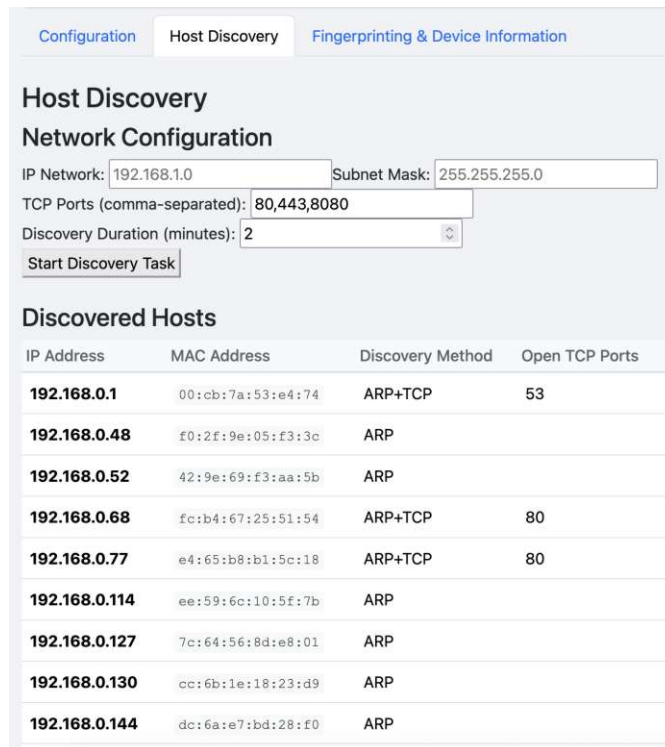


Figure 8.4: Frontend UI: Host Discovery View.

The **Device Fingerprinting** view (Figure 8.5) allows users to initiate the fingerprinting process for all hosts identified during the discovery phase. Users can set a timeout for the fingerprinting duration in minutes and toggle an option to “Instigate Additional Probe Requests”, which triggers the ESP32 to send deauthentication frames to encourage Wi-Fi devices to emit probe requests. For each discovered host, this view displays the generated fingerprint strings (e.g., `OPS(...)`, `D11IE(...)`, `TUYA(PK=...)`), any supplementary information gathered (such as potential OS matches from `nmap-os-db` or MAC vendor lookups), and highlights any matches found by comparing the new fingerprint against the project’s `iot-db`. A crucial feature of this view is the ability to save a newly generated fingerprint to the `iot-db`. This opens a modal dialog where the user can input descriptive information for the device, including a fingerprint name, manufacturer, model, version, type, and additional notes, thereby enriching the database with new device profiles.

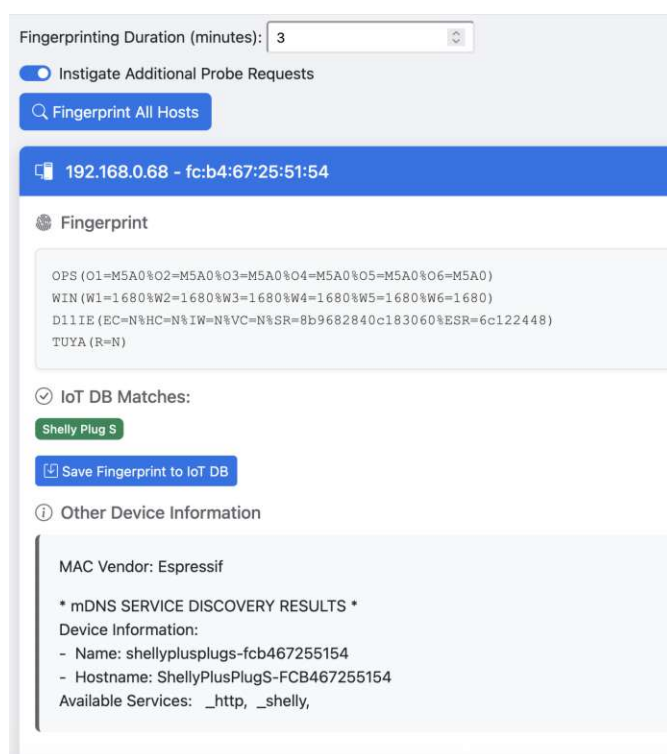


Figure 8.5: Frontend UI: Device Fingerprinting View.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Evaluation and Results

This chapter presents the evaluation of the implemented IoT device fingerprinting system. The evaluation follows the methodology detailed in section 6.3, focusing on the system's practical application and performance. The testing environment and key results are detailed, followed by an analysis of the fingerprinting effectiveness across different network layers. System performance and limitations are also discussed.

9.1 Testing Environment and Device Setup

The test network's central AP was a Vantiva Home Box Fiber Router/Modem, the standard equipment issued by the Magenta Internet Service Provider (ISP). All IoT devices and consumer electronics included in the evaluation (see Table 9.1) were connected to this AP, which provided Wi-Fi connectivity on the 2.4GHz and 5GHz band. The ESP32 probing unit, responsible for network interaction, connected to this Vantiva AP using its STA (Station) interface to communicate with the target devices.

The host system, a Raspberry Pi 4 Model B, executed the Node.js server backend and the Vue.js frontend application. This host system connected directly to the ESP32's software-enabled AP (SoftAP) interface, establishing a dedicated link for command, control, and data exchange related to the fingerprinting tasks. To ensure that test devices could perform their typical online activities, Internet connectivity was supplied to the main test network via the Vantiva router.

9.1.1 Device Inventory

A diverse set of 12 devices was selected for the evaluation, encompassing various categories such as smart home devices, consumer electronics, and general computing equipment. This diversity allows for testing the fingerprinting system's effectiveness across different

manufacturers, operating systems, and device functionalities. The detailed list of devices included in the testbed is presented in Table 9.1 (including the ESP32 prober itself).

Vendor	Model	Usage	Firmware/OS Ver.	MAC Address
Shelly	PlusPlugS	Power Socket	1.5.1	FC:B4:67:25:51:54
Shelly	PlusPlugS	Power Socket	1.0.7	E4:65:B8:B1:5C:18
Apple	Macbook Pro M1	Notebook	15.4.1 (24E263)	B2:03:74:EE:99:F2
Brother	MFC-L2710DW	Printer	Y	F8:89:D2:5D:01:C3
Amazon	Fire TV Stick 4k Max (2nd Gen)	Smart TV	Fire OS 8.13.6	F0:2F:9E:05:F3:3C
Samsung	TV UE49MU6170	Smart TV	T-KTMDEUC-1400.3	7C:64:56:8D:E8:01
Raspberry Pi	4 Model B Rev 1.2	Gen PC	Debian 12.10	DC:A6:32:6F:DF:AD
Espressif	ESP32	SoC	ESP-IDF 5.2.1	<i>N/A (Prober)</i>
Google	Pixel 7a	Smartphone	Android 15	12:DE:75:AF:76:F8
LSC	3204432 AL	Ambient Light	-	FC:3C:D7:5A:0F:95
Tuya	-	Door Sensor	-	D8:D6:68:7A:D7:7A
Tuya	-	Temperature Sensor	-	B8:06:0D:19:BD:9B
Vantiva	Home Box Fiber	Router	-	00:CB:7A:53:E4:74

Table 9.1: Inventory of Devices in the Test Network

9.1.2 Standard Evaluation Parameters

To ensure comparable and reproducible results across all subsequent evaluation tests, a default set of evaluation parameters was established. The selection of these parameters was directly informed by the findings of the coverage assessment detailed in section 9.2.1. The assessment revealed that an optimized configuration, particularly with an extended fingerprinting duration and a broader TCP port scan, significantly enhances the system's fingerprinting coverage.

Therefore, the following parameters, representing the optimized conditions from Test 3 of the coverage assessment, were adopted as the standard for all evaluation sessions unless explicitly stated otherwise:

- **Discovery Duration:** 180 seconds allocated for host discovery operations, including both ARP scanning and TCP SYN probing cycles. After this timeframe expires, the current ARP/TCP SYN cycle is allowed to complete before termination, ensuring that ongoing discovery operations are not interrupted mid-process. This approach provides sufficient opportunity for devices to respond while maintaining practical evaluation session lengths.
- **Fingerprinting Duration:** 300 seconds designated for passive monitoring and active fingerprinting operations. This extended period accommodates the capture

of intermittent device communications, particularly important for battery-powered IoT devices that may enter sleep modes.

- **TCP Port Selection:** The 1000 most commonly used TCP ports, as determined by the `nmap-services` file ranking [53].
- **Probe Request Instigation:** Active deauthentication-based probe request instigation enabled by default for Wi-Fi fingerprinting operations. This mechanism enhances the likelihood of capturing IEEE 802.11 probe requests from devices that may otherwise remain silent, as demonstrated in subsection 9.3.1.

9.1.3 Data Collection Procedure

For each device in the inventory, fingerprinting data was collected multiple times over several days using the standard evaluation parameters. Each data collection session involved:

1. Performing host discovery (ARP and TCP SYN scans) to confirm device presence and active ports.
2. Executing all implemented fingerprinting / device information plugins (MAC Vendor, mDNS, TCP, Tuya, Wi-Fi).
3. Recording the generated fingerprint strings (OPS, WIN, D11IE, TUYA) and any supplementary information.

Network conditions, such as background traffic, were kept relatively consistent, although minor variations due to device background activity were unavoidable and considered part of a realistic test scenario.

9.2 Fingerprinting Effectiveness Analysis

This section analyzes the effectiveness of the combined fingerprinting schema based on the collected data. The evaluation is structured around the core metrics defined in the methodology (section 6.3): coverage, uniqueness, stability and reproducibility. These metrics collectively provide a comprehensive assessment of the system's practical utility.

9.2.1 Coverage Assessment

The coverage assessment is performed with two primary objectives. Firstly, it aims to verify that all devices listed in the defined test inventory (Table 9.1) are successfully discovered by the system. Secondly, it evaluates whether each discovered device can be effectively covered by at least one of the implemented fingerprinting methods.

To quantify this, method-specific coverage rates are determined. These rates indicate the proportion of known test devices that were successfully fingerprinted by each individual

technique—specifically, WLAN (D11IE), TCP/IP (OPS, WIN), or Application Layer (TUYA) fingerprinting. Furthermore, the overall coverage is calculated, representing the proportion of known test devices for which the system could generate at least one valid fingerprint, irrespective of the particular method that yielded the result. This provides a comprehensive measure of the system’s ability to fingerprint the diverse set of devices in the test environment.

In this context, *fingerprinting success* refers to the successful capture and extraction of fingerprintable information. This occurs when the system receives network responses or transmissions that contain sufficient characteristic data to generate at least one valid fingerprint string (e.g., OPS, WIN, D11IE, or TUYA).

Coverage Test Comparison

To assess the fingerprinting system’s performance under varied parameters, three distinct coverage tests were conducted. These tests were chosen to evaluate the impact of the active probe request instigation mechanism, the number of TCP ports to be scanned, and the duration of the fingerprinting timeout. The specific configuration for each test was as follows:

- **Test 1 (Baseline):** This test established a baseline with minimal settings. It was configured with a fingerprinting timeout of 120 seconds, a scan of the 50 most common TCP ports, and the active deauthentication-based probe request instigation mechanism was **disabled**.
- **Test 2 (Instigation Enabled):** Building on the baseline, this test aimed to isolate the effect of the active instigation mechanism. It used the same 120-second timeout and 50-port TCP scan as Test 1, but with the deauthentication-based probe request instigation **enabled**.
- **Test 3 (Optimized Parameters):** This test represented the optimized configuration, employing more comprehensive settings. The fingerprinting timeout was extended to 300 seconds, the TCP scan was expanded to the 1000 most common ports, and the deauthentication-based probe request instigation was **enabled**.

The results of these three tests are summarized in Table 9.2. The analysis highlights key aspects of IoT device fingerprintability and the efficacy of the applied methods. While all three tests successfully discovered all 12 baseline devices (100% discovery rate), Test 3 yielded a superior overall fingerprinting success rate of 83.3%, a significant increase from Test 2’s 58.3% and Test 1’s 50%.

A critical factor influencing the success of WLAN fingerprinting is the ability to elicit Probe Requests. The comparison between Test 1 and Test 2 clearly demonstrates the impact of the deauthentication instigation mechanism. With instigation disabled in Test 1, the Wi-Fi fingerprinting success rate was only 16.7% (2 of 12 devices). Enabling the

Table 9.2: Coverage Test Results Comparison

Metric	Test 1	Test 2	Test 3
Total Devices Discovered	12	12	12
Device Discovery Rate	100%	100%	100%
TCP Fingerprinting Success	6/12 (50%)	6/12 (50%)	8/12 (66.7%)
Wi-Fi Fingerprinting Success	2/12 (16.7%)	6/12 (50%)	8/12 (66.7%)
Tuya Fingerprinting Success	1/12 (8.3%)	1/12 (8.3%)	3/12 (25%)
Overall Fingerprinting Success	6/12 (50%)	7/12 (58.3%)	10/12 (83.3%)

mechanism in Test 2 boosted the success rate to 50% (6 of 12 devices). This proves the mechanism is essential for obtaining Probe Requests from devices that would otherwise remain silent, such as the Shelly smart plugs.

The importance of the extended fingerprinting duration and expanded TCP port scan in Test 3 is also evident. The TCP fingerprinting success rate increased from 50% to 66.7% when the number of scanned ports was raised from 50 to 1000. This expanded range increased the likelihood of finding an open port that would respond to the SYN probes.

Furthermore, the extended 300-second timeout in Test 3 was crucial for fingerprinting battery-powered IoT devices, specifically the Tuya door and temperature sensors. These devices frequently enter sleep modes to conserve energy, ceasing all network activity. In the shorter tests, they did not transmit the necessary UDP broadcasts or Probe Requests to be fingerprinted. The longer observation window in Test 3 provided the opportunity to capture the periodic transmissions that occur when these devices wake, increasing the Tuya fingerprinting success from 8.3% to 25% and contributing to the improved Wi-Fi fingerprinting rate.

These results underscore that while individual techniques have limitations, their combination, enhanced by optimized parameters like active instigation and extended timeouts, maximizes the overall success rate across a diverse range of IoT devices.

9.2.2 Uniqueness Analysis

Uniqueness, as defined in the evaluation methodology (section 6.3), refers to the system's ability to generate distinct fingerprints for different device models. This is a critical metric for assessing the practical utility of the fingerprinting schema detailed in section 7.5. The analysis is based on data collected from the 12 devices in the testbed.

The uniqueness of each fingerprinting method was quantified by calculating the ratio of unique device model fingerprints to the total number of fingerprints generated for that method. The results, summarized in Table 9.3, demonstrate the varying effectiveness of each component and the synergistic benefit of their combination.

The TCP/IP fingerprint (combining OPS and WIN strings) demonstrated a uniqueness rate of 100%, successfully distinguishing all device models based on their network stack

Method	Total Fingerprints	Unique Fingerprints	Collision Groups	Uniqueness Rate
TCP (OPS/WIN)	8	8	0	100%
Wi-Fi (D11IE)	8	5	1	62.5%
Tuya (TUYA)	3	3	0	100%
Combined	10	10	0	100%

Table 9.3: Uniqueness Rates of Fingerprinting Methods

implementations. Notably, the two Shelly PlusPlugS devices in the testbed, despite having distinct firmware versions (1.5.1 and 1.0.7), exhibited identical TCP fingerprints. This is the expected behavior for the model-level identification goal of this thesis: both devices are the same model and thus share the same underlying network stack implementation. However, this observation also highlights a limitation in firmware version differentiation, further elaborated in section 10.2, where variations in firmware do not invariably lead to differences in network stack fingerprints.

The Wi-Fi fingerprint (D11IE) exhibited a moderate uniqueness rate of 62.5%. This was due to a collision that occurred among three different Tuya-based devices: the Tuya Door Sensor, the Tuya Temperature Sensor, and the LSC Ambient Light. These devices, although being different device models, are built upon a common Tuya hardware and firmware base, leading to identical Information Element patterns in their probe requests.

The Application Layer fingerprint (TUYA) proved to be also effective for its specific target group, achieving a 100% uniqueness rate among the three Tuya device models it covered. By extracting the `productKey` from their UDP broadcasts, the system could uniquely identify each model.

Ultimately, the strength of the combined fingerprint is evident in its ability to resolve ambiguities present at lower layers. By integrating characteristics from all network layers, the system achieved a 100% uniqueness rate across all fingerprinted device models. This score signifies that for every device model within the testbed with a fingerprint a unique fingerprint was assigned. The high granularity of the Tuya fingerprint successfully resolved the collision observed at the Wi-Fi layer, correctly differentiating the Door Sensor, Temperature Sensor, and LSC light models. This result validates the methodological decision to incorporate application-layer analysis, as it provides the necessary specificity to distinguish between different products that share a common hardware platform, a central goal of this thesis.

9.2.3 Stability and Reproducibility

The evaluation of fingerprint stability and reproducibility is crucial for determining the reliability and practical utility of the fingerprinting system. As defined in section 6.3, these two metrics address different aspects of consistency.

Stability refers to the consistency of a fingerprint generated for the same device over extended periods. This metric assesses whether the inherent characteristics of a device, as captured by its fingerprint, remain constant over time. For this evaluation, stability was measured by performing fingerprinting operations on each test device at several distinct time intervals, spanning from an initial baseline measurement to subsequent measurements taken after several minutes and up to 4 hours later. The variance in fingerprint components across these time intervals was then analyzed to determine the long-term consistency of the generated fingerprints. A high degree of stability indicates that the fingerprint attributes are not significantly affected by the device’s operational uptime or state.

Reproducibility, on the other hand, assesses the ability to obtain consistent fingerprinting results when the entire measurement process is repeated under similar conditions in a short timeframe. The fingerprints generated from these repeated runs were then compared to ascertain if the system consistently produces the same output for the same device. High reproducibility suggests that the fingerprinting methodology and its implementation are robust and yield dependable results upon repeated execution.

Quantitative Stability and Reproducibility Analysis

The quantitative analysis of the system’s stability and reproducibility yielded strong results, confirming the reliability of the fingerprinting methods. For the vast majority of devices, particularly mains-powered ones, both stability and reproducibility were exceptionally high. Representative tests indicated an overall fingerprint reproducibility of 91.7% and stability of 86.1%.

Table 9.4: Stability and Reproducibility Metrics by Fingerprinting Method

Method	Reproducibility		Stability	
	Consistent/Total	Rate	Stable/Total	Rate
TCP (OPS/WIN)	12/12	100%	12/12	100%
Wi-Fi (D11IE)	10/12	83.3%	9/12	75%
Tuya (TUYA)	11/12	91.7%	10/12	83.3%
Overall	33/36	91.7%	31/36	86.1%

Both metrics were evaluated using a consistent methodology. Reproducibility assessed whether each device-method combination produced identical fingerprints across all 5 consecutive runs performed in immediate succession, tracking 36 unique fingerprints (12 devices \times 3 methods). Stability assessed whether fingerprints remained identical across all 4 time intervals (0s, 300s, 600s, and 1800s), also tracking the same 36 fingerprints. The TCP method demonstrated perfect stability and reproducibility across all devices. The observed variations in the Wi-Fi and Tuya methods were primarily attributable to battery-powered devices, such as the Tuya door and temperature sensor. These devices are known for their power-saving sleep cycles, during which they cease network activity to conserve energy. While their intermittent network activity meant that a fingerprint

could not be captured in every single attempt, the extended 300-second fingerprinting window proved effective in maximizing the capture opportunities.

Most importantly, the analysis confirmed a critical finding: whenever a fingerprint was successfully obtained from these battery-powered devices, its content was perfectly stable and reproducible. The resulting fingerprint strings were identical across all successful test runs. This demonstrates that the fingerprinting characteristics themselves are inherently consistent. The challenge is not one of reliability but of timing—capturing the device during its brief active communication windows.

For practical deployments, the impact of occasional missed captures can be effectively managed by aggregating results over multiple observation periods. This strategy further enhances the system’s robustness, ensuring a comprehensive and reliable device identification process.

Observed Variability in Wi-Fi Information Elements

Beyond the general assessment of fingerprint component stability, preliminary investigations revealed significant variability in the ordering and presence of Information Elements within IEEE 802.11 Probe Request frames. This variability can substantially affect the stability and reproducibility of Wi-Fi based fingerprints (D11IE). Notably, this phenomenon was not isolated to a single device but was observed in 3 out of 8 devices (37.5%) devices in the testbed that successfully generated Wi-Fi fingerprints, as summarized in Table 9.5.

Table 9.5: Information Element Variability in Probe Requests

Device	Description
Raspberry Pi 4 Model B	Variable IE ordering and VSI presence due to dual-mode P2P/STA operation; firmware alternates between "full-featured" probes (with P2P VSIs) and "minimal" probes (baseline 802.11 IEs only)
LSC Ambient Light	HT Capabilities IE inconsistently present across probe requests (occurs in both broadcast and unicast frames)
Tuya Temperature Sensor	HT Capabilities IE inconsistently present across probe requests (occurs in both broadcast and unicast frames)

The most extensively analyzed example of variability was observed with the Raspberry Pi 4 Model B. This device, across different capture sessions and without intentional configuration changes, emitted Probe Requests with notable differences in IE composition and order. Figures 9.1 and 9.2 depict two such distinct Probe Request frames from the Raspberry Pi 4. The first frame (Figure 9.1) contains numerous Vendor-Specific IEs (VSIs) while the second frame (Figure 9.2) omits those VSIs, such as those related to WPS and P2P. This behavior is attributable to the Raspberry Pi’s networking stack, specifically its use of the `brcmfmac` driver and `wpa_supplicant` with P2P functionality enabled. This configuration results in the exposure of two logical interfaces: `wlan0` for standard station (STA) infrastructure scans and `p2p-device` for Wi-Fi Direct P2P

discovery. During an active scan, the firmware services both interfaces, often leading to the transmission of two types of broadcast probes on a given channel or in close succession: a "full-featured" probe from the `p2p-device` interface (akin to Figure 9.1), advertising extensive capabilities for P2P negotiation, and a "minimal" probe from the `wlan0` interface (resembling Figure 9.2), containing only baseline 802.11 IEs for maximum compatibility with legacy access points. While this provides a clear rationale for the Raspberry Pi's behavior, it also highlights a broader challenge: similar dual-mode operations or dynamic IE selection based on P2P or other service discovery protocols may exist in other Linux-based embedded IoT devices or devices utilizing comparable Wi-Fi chipsets and supplicant software.

The LSC Ambient Light and Tuya Temperature Sensor exhibited a different type of IE variability. These devices inconsistently included the HT Capabilities IE across multiple probe requests. This variability was observed in both broadcast and unicast probe requests, indicating that the frame destination type was not the determining factor. Interestingly, the Tuya Door Sensor, despite being from the same manufacturer and sharing a similar hardware platform with the Temperature Sensor, did not exhibit this variability, demonstrating that IE stability can vary even within a single vendor's product ecosystem.

This observed instability in IE composition and order across multiple devices reinforces the decision to exclude IE sequence as a feature in the final D11IE fingerprinting schema developed in this work. While core IEs such as Supported Rates, HT Capabilities, and Extended Capabilities generally exhibit consistency when present, the variability in their ordering and the inconsistent presence of certain IEs—driven by factors like concurrent P2P discovery, STA scanning, and firmware-specific behaviors—poses a significant challenge to creating stable D11IE fingerprints that rely on the complete order of IEs.

```

IEEE 802.11 Wireless Management
- Tagged parameters (247 bytes)
  > Tag: SSID parameter set: Wildcard SSID
  > Tag: Supported Rates 1, 2, 5.5, 11, [Mbit/sec]
  > Tag: Extended Supported Rates 6, 9, 12, 18, 24, 36, 48, 54, [Mbit/sec]
  > Tag: DS Parameter set: Current Channel: 13
  > Tag: HT Capabilities (802.11n D1.10)
  > Tag: Extended Capabilities (8 octets)
  > Tag: Vendor Specific: Microsoft Corp.: WPS
  > Tag: Vendor Specific: Wi-Fi Alliance: P2P
  > Tag: Vendor Specific: Wi-Fi Alliance: Multi Band Operation - Optimized Connectivity Experience
  > Tag: Vendor Specific: Epigram, Inc.: HT Capabilities (802.11n D1.10)
  > Tag: Vendor Specific: Microsoft Corp.: Unknown 8
  > Tag: Vendor Specific: Broadcom

```

Figure 9.1: Example of Probe Request from Raspberry Pi 4 (Observation 1), showing a directed probe with numerous Vendor-Specific IEs.

```

IEEE 802.11 Wireless Management
  ▾ Tagged parameters (112 bytes)
    ▶ Tag: SSID parameter set: Wildcard SSID
    ▶ Tag: Supported Rates 1, 2, 5.5, 11, [Mbit/sec]
    ▶ Tag: Extended Supported Rates 6, 9, 12, 18, 24, 36, 48, 54, [Mbit/sec]
    ▶ Tag: DS Parameter set: Current Channel: 9
    ▶ Tag: HT Capabilities (802.11n D1.10)
    ▶ Tag: Extended Capabilities (8 octets)
    ▶ Tag: Vendor Specific: Epigram, Inc.: HT Capabilities (802.11n D1.10)
    ▶ Tag: Vendor Specific: Microsoft Corp.: Unknown 8
    ▶ Tag: Vendor Specific: Broadcom

```

Figure 9.2: Example of Probe Request from Raspberry Pi 4 (Observation 2), showing a broadcast probe with fewer Vendor-Specific IEs.

9.3 Layer-Specific Performance Evaluation

Having established the overall effectiveness of the combined fingerprinting schema in terms of coverage, uniqueness, and reproducibility/stability, this section provides a more granular analysis of each constituent method’s performance. The following subsections evaluate the results obtained from each network layer individually: the WLAN layer (D11IE), the TCP/IP stack (OPS and WIN), and the Application layer (TUYA). This layer-specific examination offers deeper insights into the specific strengths, limitations, and characteristic behaviors observed for each fingerprinting technique.

9.3.1 WLAN Fingerprinting Results (D11IE)

The WLAN fingerprinting method, which relies on Information Elements from 802.11 Probe Requests, demonstrated varied but significant effectiveness, largely dependent on device behavior and the probing strategy employed. The selection of IEs for the D11IE fingerprint—focusing on stable yet variable elements like Extended Capabilities, HT/VHT Capabilities, and Supported Rates as justified in section 7.2.3—forms the basis of this method’s utility. The importance of actively instigating probe requests to achieve adequate coverage is detailed in the comparative analysis in section 9.2.1.

Even with instigation, several devices, including the Amazon Fire TV Stick, Samsung Smart TV, and Apple Macbook Pro, did not emit capturable Probe Requests. This is further discussed in section 10.2.

An interesting behavior was observed with Tuya devices, which send Probe Requests as both broadcast frames and unicast frames directed to the AP (see Figure 9.3 and Figure 9.4). As discussed in section 9.2.3, the LSC Ambient Light and Tuya Temperature Sensor exhibited inconsistent inclusion of the HT Capabilities IE across multiple probe requests. This variability was observed in both broadcast and unicast frames, indicating that the frame destination type is not the determining factor for IE composition. For consistency and to ensure representative device behavior, the system was configured to

only process broadcast Probe Requests for D11IE fingerprint generation, as these are more reliably captured across different network configurations.

The content of the captured IEs provided a moderate level of uniqueness, as analyzed in subsection 9.2.2. While effective in distinguishing many device types, collisions occurred among devices built on shared hardware platforms, most notably within the Tuya ecosystem, reinforcing the need for a multi-layered fingerprinting approach.

```
IEEE 802.11 Wireless Management
- Tagged parameters (63 bytes)
  ▶ Tag: SSID parameter set: Magenta5933947
  ▶ Tag: Supported Rates 1, 2, 5.5, 11, [Mbit/sec]
  ▶ Tag: Extended Supported Rates 6, 9, 12, 18, 24, 36, 48, 54, [Mbit/sec]
  ▶ Tag: DS Parameter set: Current Channel: 13
  ▶ Tag: HT Capabilities (802.11n D1.10)
```

Figure 9.3: Example of a broadcast Probe Request from a Tuya device.

```
IEEE 802.11 Wireless Management
- Tagged parameters (35 bytes)
  ▶ Tag: SSID parameter set: Magenta5933947
  ▶ Tag: Supported Rates 1, 2, 5.5, 11, [Mbit/sec]
  ▶ Tag: Extended Supported Rates 6, 9, 12, 18, 24, 36, 48, 54, [Mbit/sec]
  ▶ Tag: DS Parameter set: Current Channel: 6
```

Figure 9.4: Example of a unicast Probe Request to the AP from the same Tuya device.

9.3.2 TCP/IP Fingerprinting Results (OPS, WIN)

The TCP/IP fingerprinting method, as detailed in section 7.3, utilizes variations in TCP stack implementations across different devices. This is achieved by sending a sequence of six TCP SYN probes, analogous to Nmap’s approach (subsection 7.3.3), and analyzing the TCP options (OPS) and window sizes (WIN) in the responses. The implementation of this probing and analysis is handled by the TCP Fingerprinting Plugin described in subsection 8.3.3.

The evaluation of TCP/IP fingerprinting across the 12 devices in the testbed (as listed in Table 9.1) revealed notable variability in the generated OPS and WIN strings. Out of the 12 devices, 8 provided valid TCP fingerprints, corresponding to the 66.7% TCP fingerprinting success rate reported in the coverage assessment (Table 9.2). Devices such as the Vantiva Home Box Fiber router, the Amazon Fire TV Stick 4k Max, the Brother MFC-L2710DW printer, and the Raspberry Pi 4 Model B each produced distinct OPS and WIN signatures.

A key aspect of the TCP/IP fingerprinting module is its capability to compare generated fingerprints against the comprehensive `nmap-os-db` to infer the underlying operating system. This feature proved effective during the evaluation. For instance, the TCP fingerprint obtained from the Raspberry Pi 4 consistently matched entries for various Linux kernels in the `nmap-os-db`, aligning with its Debian 12.10 operating system. The

Vantiva Home Box Fiber router’s fingerprint also corresponded to a specific Linux kernel (2.6.30), demonstrating the method’s utility in identifying specialized operating systems.

Despite its strengths, TCP/IP fingerprinting encountered several challenges. Firstly, four devices did not yield TCP fingerprints ($OPS(R=N)WIN(R=N)$): the Google Pixel 7a, the Tuya Door Sensor, the Tuya Temperature Sensor, and the Apple Macbook Pro M1. This could be due to various factors, including host-based firewalls dropping the probes (which is typical for consumer devices such as smartphones and notebooks), devices being in a deep sleep state and not responding to network requests on the scanned ports within the test window, or not having any of the 1000 scanned TCP ports open.

Secondly, while this evaluation showed unique fingerprints for each device model, it is likely that a larger test set would reveal collisions where different models share the same TCP/IP fingerprint. This potential limitation is discussed in more detail in section 10.2.

9.3.3 Application Layer Fingerprinting Results (TUYA)

Application layer fingerprinting, as detailed in section 7.4, was specifically implemented to target Tuya-based devices by extracting their unique `productKey`. This method relies on passively capturing and decrypting UDP broadcast messages, from which the `productKey` is parsed. This identifier is assigned by Tuya to a specific product model or a closely related batch, offering a granular means of device identification.

The evaluation focused on three Tuya-based devices within the testbed: the LSC Ambient Light, the Tuya Door Sensor, and the Tuya Temperature Sensor. While the battery-powered nature of the door and temperature sensors introduced timing challenges due to their sleep cycles, as discussed in subsection 9.2.3, the method proved effective under optimized conditions. Using the standard evaluation parameters (Test 3, section 9.2.1), which included an extended fingerprinting timeout, the `productKey` was successfully extracted for all three devices.

In terms of uniqueness, the `productKey` proved highly effective. Each of the three distinct Tuya device models yielded a unique `productKey`:

- LSC Ambient Light: `keydhn7587yertgt`
- Tuya Door Sensor: `keyhhtuxrsp7ueap`
- Tuya Temperature Sensor: `1f36y5nwb8jkxwgg`

To validate that these `productKey` values are indeed model-specific identifiers rather than device-instance identifiers, external verification was performed. Searching for the extracted `productKey` values in online resources and developer communities yielded results associated with the corresponding device types. For instance, the `productKey` `keyhhtuxrsp7ueap` consistently appears in contexts related to Tuya door sensor products, while `1f36y5nwb8jkxwgg` is associated with temperature sensor implementations.

This external validation corroborates Tuya's documentation claim that `productKey` values are assigned at the product model level, confirming their suitability as stable fingerprinting characteristics.

This demonstrates that, for the tested devices, the application-layer fingerprint alone was sufficient to distinguish between these different Tuya products, achieving a 100% uniqueness rate among the devices covered by this method, as also noted in the overall uniqueness analysis (Table 9.3). This outcome underscores the value of application-layer data in providing model-specific identification where lower-layer methods might encounter collisions, particularly within a single vendor's ecosystem.

9.4 mDNS and MAC Vendor Lookup Results

Beyond the core fingerprinting methods that generate structured strings for the `iot-db`, the system also gathers supplementary device information through mDNS service discovery and MAC vendor lookups. While not part of the formal fingerprint used for matching, this information provides valuable context that aids in the manual identification and functional understanding of discovered devices.

The MAC vendor lookup, which correlates the OUI of a device's MAC address with its manufacturer, yielded useful results for a majority of the test devices. For instance, both Shelly PlusPlugS devices were correctly identified as using "Espressif" chipsets, and the Raspberry Pi 4 was identified as a "Raspberry Pi Trading" product. Similarly, the Tuya-based devices (Door Sensor, Temperature Sensor, and LSC Ambient Light) were all traced back to "Tuya Smart", confirming their shared hardware ecosystem. Other successful lookups included "Vantiva USA" for the router, "Amazon Technologies" for the Fire TV Stick, and "Samsung Electronics" for the smart TV. This method provides a quick, high-level classification of a device's hardware origin.

The mDNS service discovery provided deeper functional insights for several devices. The Brother MFC-L2710DW printer was particularly verbose, advertising a wide range of services including `_ipp` (Internet Printing Protocol), `_printer`, `_scanner`, and `_http` for its web-based administration interface. The TXT records in its mDNS responses further enriched this information, explicitly stating the manufacturer as "Brother" and the model as "Brother MFC-L2710DW series". The Shelly PlusPlugS also advertised its presence via mDNS, exposing a `_http` service for direct web access and a custom `_shelly` service. Other devices, like the Amazon Fire TV Stick, advertised services such as `_matterd`, indicating its compatibility with the Matter smart home standard. In contrast, many of the simpler or more locked-down devices, including the Tuya sensors and the LSC light, did not respond to mDNS queries.

These two methods, while not contributing directly to the uniqueness metrics of the primary fingerprinting schema, proved to be highly effective in augmenting the overall device profile. The MAC vendor lookup offers a reliable, albeit general, hardware-level

identifier, while mDNS can reveal specific functionalities and even self-reported device models, significantly enhancing the system’s descriptive power.

9.5 System Performance Metrics

This section evaluates the performance of the ESP32 firmware and the frontend application, focusing on memory utilization, task execution times, and overall system responsiveness. The analysis of these metrics is crucial for validating the feasibility of using a resource-constrained device for complex network probing tasks, as outlined in the methodology (subsection 6.2.1), and ensuring the practicality of the system as a whole.

9.5.1 Memory Consumption Analysis

The ESP32’s dynamic memory (heap) usage was monitored throughout the fingerprinting process to assess resource utilization across different operational phases. Figure 9.5 illustrates the minimum, maximum, and average free heap space measured during each phase of the fingerprinting workflow.

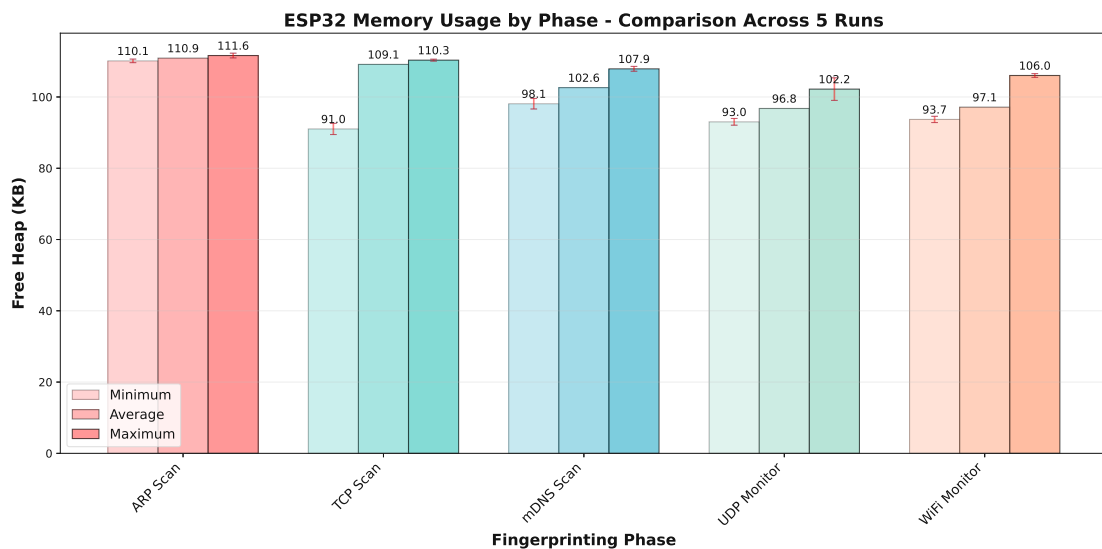


Figure 9.5: ESP32 heap memory consumption across different fingerprinting operations.

The ARP scan phase, which queries all hosts in the /24 subnet, demonstrated stable memory consumption with free heap ranging between 110.1 KB and 111.6 KB. The mDNS service discovery phase exhibited more variability, with minimum free heap values dropping to 98.1 KB during intensive service enumeration operations that require parsing and temporary storage of numerous service responses. The TCP scan phase, involving the creation of raw sockets and connection tracking tables as described in section 8.2.1, showed the widest range of memory utilization, with minimum free heap values of 91 KB, reflecting the dynamic nature of connection state management.

The continuous monitoring tasks (WiFi promiscuous mode and UDP packet capture) operated with free heap space ranging from 93 KB to 106 KB. Across all operational phases, the system maintained an overall minimum free heap of 91 KB and maximum of 111.6 KB. These measurements demonstrate that while the fingerprinting tasks are memory-intensive, the ESP32 firmware manages resources effectively. Even under peak load, the system maintains sufficient free heap space for stable operation, confirming the feasibility of implementing network analysis and tracking on resource-constrained embedded devices.

9.5.2 Task Execution Times

The execution time for specific probing tasks was also measured. An ARP scan across the entire /24 subnet consistently completed in approximately 13.7 seconds. The duration of TCP scans varied with the number of target IPs and open ports. For instance, a TCP scan targeting a list of 10-12 discovered IPs completed in about 1.5 seconds on average. The comprehensive mDNS service discovery process, including meta-service queries and subsequent instance queries, took approximately 12.2 seconds to complete. These execution times confirm that the ESP32 is capable of performing these network-intensive operations with reasonable efficiency.

The performance of the frontend application, comprising the Node.js backend and Vue.js user interface, introduces only marginal overhead to the overall fingerprinting process. The Node.js server efficiently orchestrates the fingerprinting workflow, processing results from various plugins with minimal delay. Database operations, such as loading the `nmap-os-db` and matching against the `iot-db`, are optimized for quick retrieval. Consequently, the Vue.js interface remains responsive, with device and fingerprint data populating the user interface almost instantaneously upon retrieval from the backend.

9.6 Database Contribution and Validation

The process of contributing to the `iot-db` is integrated into the frontend application's workflow, as described in section 8.3.3. After a fingerprinting task is completed, the user is presented with the generated fingerprint strings for each device. The user can then choose to save a new fingerprint to the database. This action opens a dialog where the user provides descriptive metadata, including the device's manufacturer, model, firmware version, and general type. This information, combined with the systematically generated fingerprint strings (OPS, WIN, D11IE, TUYA), is then formatted and appended to the `iot-db` file, creating a new reference entry for future comparisons.

During the evaluation, those devices in the test inventory (Table 9.1) that generated at least one of fingerprint were successfully added to the `iot-db`. As established in the uniqueness analysis (subsection 9.2.2), this resulted in the creation of 10 unique fingerprint entries, with the two identical Shelly PlusPlugS devices correctly sharing a

single fingerprint signature, and the Tuya-based devices being successfully differentiated by their application-layer characteristics.

The validation of the database's matching accuracy was performed by conducting subsequent rescans of the test devices and comparing the newly generated fingerprints against the now-populated `iot-db`. The system demonstrated a high degree of accuracy in re-identifying these known devices. This success is directly attributable to the high stability and reproducibility of the fingerprinting methods, as quantified in subsection 9.2.3. With overall fingerprint reproducibility 91.7% and stability measured at 86.1%, the system consistently generated fingerprints that matched their corresponding entries in the database. This confirms the effectiveness of the `iot-db` as a reliable mechanism for known device identification within the implemented fingerprinting framework.

This structured evaluation provides insights into the strengths and weaknesses of the implemented fingerprinting system, laying the groundwork for future improvements and research directions discussed in the conclusion.

Discussion

This chapter reflects on the findings presented in chapter 9, contextualizing the contributions of this work within the broader field of IoT security. It begins by summarizing the primary contributions of the developed fingerprinting system. Subsequently, it addresses the inherent limitations of the study, discussing constraints related to the hardware, methodology, and the devices themselves. The chapter also revisits the ethical considerations surrounding network fingerprinting and concludes by outlining promising directions for future research that build upon the insights gained.

10.1 Contribution of the Work

The primary contribution of this thesis is not the invention of a novel, isolated fingerprinting characteristic, but rather the design, implementation, and practical evaluation of an integrated, multi-layered IoT device identification system built upon accessible, low-cost hardware. This work provides a holistic solution that addresses the engineering challenges of creating a practical and extensible fingerprinting tool. The key contributions are detailed as follows:

- **Multi-Layered Fingerprinting on Accessible Hardware:** This research successfully designed and implemented a complete system that synthesizes fingerprinting characteristics from the WLAN (IEEE 802.11), TCP/IP, and Application layers. As detailed in chapter 8, the system architecture leverages an ESP32 for low-level network probing and a host system for control and analysis. This approach demonstrates the feasibility of building a powerful fingerprinting tool using commodity hardware, making it accessible for researchers, administrators, and enthusiasts, in contrast to methods requiring specialized equipment like Software-Defined Radios or significant computational resources for complex machine learning models.

- **Practical Validation of the Multi-Layered Approach:** The evaluation in chapter 9 provides empirical evidence for the core hypothesis that a multi-layered approach is more effective than relying on a single layer. This result validates the necessity of incorporating application-specific data to achieve granular, model-level identification where lower-layer characteristics may be insufficient.
- **Extensible Architecture:** Another contribution is the design of an extensible, plugin-based architecture for the host application’s fingerprinting logic, as described in subsection 8.3.2. This modularity makes the system adaptable and future-proof, allowing for the straightforward integration of new fingerprinting modules for different protocols (e.g., MQTT, CoAP) or vendor ecosystems without requiring core system modifications. This provides a flexible framework for future research and development in IoT device identification.
- **Novel Implementation and Empirical Insights:** The thesis contributes practical implementation solutions for conducting advanced network analysis on a resource-constrained platform. This includes the development of a custom `netif` input wrapper to overcome lwIP’s ARP table limitations (section 8.2.1), a robust management system for raw TCP Protocol Control Blocks (section 8.2.1), and an effective deauthentication-based mechanism to instigate probe requests (section 8.2.1). Furthermore, the evaluation provides valuable empirical insights, such as demonstrating the necessity of extended monitoring windows for fingerprinting battery-powered, intermittently connected devices (section 9.2.1).

10.2 Limitations of the Study

While the evaluation demonstrates the effectiveness of the multi-layered fingerprinting approach, it is important to acknowledge the system’s inherent limitations and edge cases as outlined below.

10.2.1 Hardware and Resource Constraints

A significant hardware limitation is the ESP32’s exclusive support for the 2.4GHz Wi-Fi band. This restricts the WLAN fingerprinting method to devices operating on this frequency. Devices that operate solely in the 5GHz spectrum, such as the Apple Macbook Pro and Amazon Fire TV Stick observed during testing, could not be subjected to Wi-Fi IE fingerprinting. While this can sometimes be mitigated by network configurations that encourage or force 2.4GHz operation, it remains a fundamental constraint of the chosen hardware platform. The adoption of newer microcontrollers with dual-band support, such as the ESP32-C5, presents a clear path to addressing this limitation in future work.

Furthermore, the ESP32’s limited resources affect the system’s scale. The firmware relies on statically allocated arrays to manage captured data, a necessity given the limited heap memory. These fixed limits cap the number of devices and data points that can be

processed in a single scan and could become a bottleneck in denser networks, as discussed further in the context of scalability.

10.2.2 Methodology and Scope

A key limitation is the small number and limited variety of devices used in the evaluation. The study was conducted with twelve consumer-grade IoT devices available in a home environment. This small sample size, while diverse in its selection, does not represent the full spectrum of IoT products, which includes industrial, healthcare, and other specialized devices. Consequently, the system's performance and the uniqueness of the generated fingerprints are likely to be optimistic. With a broader and more extensive set of devices, the uniqueness of fingerprints, particularly those from the TCP/IP stack, would likely decrease as more devices could share identical network stack implementations. Therefore, the results on fingerprint uniqueness should be interpreted with caution, as they may not hold when applied to a larger and more varied device population.

An initial hypothesis was that the methodology might also differentiate between firmware versions. However, the uniqueness analysis (subsection 9.2.2) showed that two Shelly Plus-PlugS devices with different firmware produced identical TCP/IP and Wi-Fi fingerprints. This outcome is expected, as a change in the fingerprint requires modifications to the underlying network stack, such as the Wi-Fi driver or the TCP/IP implementation. Since firmware updates often target application-level logic or security fixes without altering these core network components, the fingerprints remain unchanged, thus limiting the granularity for version detection.

10.2.3 Device Behavior and Network Dependencies

Certain devices proved inherently difficult to fingerprint. As noted in the TCP/IP results (subsection 9.3.2), devices like the Google Pixel 7a and the Tuya sensors did not respond to TCP probes. This unresponsiveness is likely due to host-based firewalls, deep-sleep power-saving modes that minimize network activity, or simply not having any of the scanned TCP ports open. Furthermore, the stability of Wi-Fi fingerprints was impacted by the variable behavior of some devices, which did not consistently transmit probe requests even with the deauthentication instigation mechanism enabled.

10.2.4 NMAP OS Database

The system's reliance on the `nmap-os-db` for OS identification introduces limitations tied to the scope and accuracy of the database. The database is heavily populated with fingerprints from general-purpose operating systems (e.g., Linux, Windows) and has limited coverage of the specialized Real-Time Operating System (RTOS) or bare-metal implementations common in the IoT landscape. Consequently, while it can provide accurate matches for devices running conventional Linux kernels, like the Raspberry Pi, it often fails to identify the bespoke systems found in many commercial IoT products.

10.2.5 Scalability and Deployment Environment

The system's performance during the discovery and fingerprinting phases is a potential limitation in large-scale networks. As the number of devices increases, the resource constraints of the ESP32 and the single-threaded nature of the probing process could lead to prolonged scan times and missed events. The system's design is tailored for small-scale, non-critical environments such as homes or small offices. The use of deauthentication frames to instigate probe requests, while effective, is an aggressive technique that could be disruptive in a production or enterprise network. The current implementation is not designed for large-scale network reconnaissance. Scaling the system to cover a larger physical area or a significantly higher number of devices would necessitate a more sophisticated architecture, likely involving multiple ESP32 probers and a central orchestration server to coordinate their actions and aggregate results.

10.3 Ethical Considerations

The system developed in this thesis is a dual-use technology, effective for both legitimate network defense and potentially malicious reconnaissance. Its intended purpose is defensive by providing network administrators with a tool for asset inventory, rogue device detection, and vulnerability assessment, thereby strengthening network security.

However, the same capabilities can be exploited by attackers to identify high-value targets and plan attacks based on a device's specific model and inferred OS. The system's methodology is not entirely passive since it employs active TCP SYN scans and, more significantly, a disruptive deauthentication-based mechanism to instigate probe requests. As demonstrated in the evaluation (section 9.2.1), this technique is highly effective but constitutes a denial-of-service action.

The use of such aggressive techniques without explicit, informed consent from the network owner is unethical and often illegal. Therefore, this system must be used responsibly, exclusively for security auditing and research on authorized networks. The goal of this work is to advance defensive capabilities by highlighting the practical realities of IoT device identification, enabling the security community to develop better countermeasures and improve device resilience.

10.4 Future Work

Building on the foundation of this thesis, several promising avenues for future research emerge that could enhance the system's capabilities and address its current limitations.

First, the discovery process could be made more performant and comprehensive. To improve scalability in larger networks, future work could optimize current scanning mechanisms. Incorporating additional discovery methods, such as ICMP echo requests or passive analysis of a wider range of broadcast traffic, could also improve the detection rate of devices that are firewalled or unresponsive to the current ARP and TCP probes.

Second, the extensible, plugin-based architecture provides a clear path for expanding the system's fingerprinting coverage. A key area for future work is the development of new application-layer fingerprinting plugins for other prevalent IoT protocols, such as MQTT, CoAP, and various vendor-specific HTTP APIs. This would significantly broaden the system's ability to perform granular identification across a more diverse range of device ecosystems.

Finally, a major extension would be to expand the system beyond Wi-Fi to other wireless technologies that are common in the IoT landscape, most notably Bluetooth Low Energy (BLE) and Zigbee. This would be a significant undertaking, requiring the integration of new hardware capabilities into the probing unit and the development of entirely new fingerprinting methodologies tailored to the unique characteristics and protocol stacks of these different wireless standards.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Conclusion

This thesis addressed the growing challenge of identifying Internet of Things (IoT) devices within network environments, a critical task for security monitoring and management. The central hypothesis was that a multi-layered fingerprinting approach, combining characteristics from the WLAN, TCP/IP, and application layers, could provide a more effective and granular means of device identification than single-layer methods. To validate this, the research focused on two primary questions: which protocol characteristics are suitable for fingerprinting, and to what extent can they be extracted using a low-cost ESP32 System-on-Chip?

The investigation confirmed that a multi-layered strategy is effective for achieving accurate device identification. The evaluation demonstrated the distinct strengths of each layer: both the TCP/IP and application-layer fingerprinting methods achieved 100% uniqueness for the device models they covered, while the Wi-Fi layer method encountered collisions among devices with shared hardware platforms, resulting in a 62.5% uniqueness rate. The coverage tests (83.3% overall coverage rate) also showed that the fingerprinting methods complement each other, as not all techniques were successful for every device. Thus, the true strength of the system was realized when these layers were combined to maximize device coverage and to achieve a perfect model-level uniqueness rate of 100% across all fingerprinted devices. This synergy, where application-layer data resolved lower-layer ambiguities, answers the first research question by affirming the value of a holistic, multi-faceted fingerprinting schema.

In response to the second research question, this work successfully demonstrated the viability of the ESP32 as a dedicated network probing unit. Despite its resource constraints, the implemented firmware was capable of performing all required low-level network operations. The research produced practical solutions to platform-specific challenges, such as managing lwIP's raw socket resources and implementing an effective deauthentication-based mechanism to instigate probe requests. This instigation technique

11. CONCLUSION

proved critical for improving Wi-Fi fingerprinting coverage, boosting its success rate from 16.7% to 50% in comparative tests.

The primary contribution of this thesis is the design, implementation, and practical evaluation of an integrated, reliable, and extensible fingerprinting system built on accessible hardware. The high measured stability (86.1%) and reproducibility (91.7%) of the generated fingerprints confirm the robustness of the chosen features. In contrast to research focusing on computationally intensive machine learning models or methods requiring specialized hardware, this work provides a practical blueprint for a tool that synthesizes well-established techniques from multiple network layers. Additionally, the investigation into the variability of Wi-Fi Information Elements provided valuable insights, explaining observed instabilities and informing the design of a more resilient fingerprinting schema. The system's modular, plugin-based architecture further enhances its utility, providing a flexible framework for future expansion.

However, the study has limitations. The evaluation was conducted on a limited, albeit diverse, set of consumer-grade devices, and a larger device set would be needed to provide more realistic uniqueness and coverage ratings. Additionally, the system's performance in large-scale networks and its scalability remain to be thoroughly evaluated in future work.

Overview of Generative AI Tools Used

Generative AI tools were used as an aid in this thesis. Primarily, they were used for grammar and spelling checks, as well as for occasional improvements to phrasing. In addition, AI tools were used for specific development tasks. The following sections list the main use cases, specifying the tool used and the prompt entered.

- **Decrypting Tuya Network Traffic:** An AI tool was used to translate the logic for decrypting Tuya data traffic (as developed by the TinyTuya project [46]) from Python to JavaScript.
 - **Tool:** Claude Sonnet 3.5 (Date: 04.02.2025)
 - **Prompt:** "can you create a java script file, that reimplements the decrypt_udp logic? so that when i give the function in js a encrypted string, it will decrypt it for me. as an example for decryption, this example should be used: '000055aa0000000000000000130000009c0000...?'"
- **Framework for Automated Evaluation:** An AI tool was also consulted for the design and creation of the test framework for automated evaluation.
 - **Tool:** Claude Sonnet 3.7 (Date: 17.04.2025)
 - **Prompt:** "help me to create a framework for automated evaluation based on the provided evaluation methodology and my nodejs server code"



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

4.1	Comparison of the OSI and TCP/IP Layered Models (adapted from [4])	9
4.2	The process of data encapsulation as data passes down the protocol stack [5]	10
4.3	The 802.11 Independent BSS and Infrastructure BSS topologies (adapted from [6])	12
4.4	802.11 MAC Frame [7]	13
4.5	Probe Request of a Smart Ambient Light captured with Wireshark	14
4.6	The Address Resolution Protocol (ARP) Transaction Process [4]	16
4.7	The IPv4 Datagram Header Format (adapted from [4])	17
4.8	The TCP Three-Way Handshake Connection Establishment Procedure [?]	20
4.9	The TCP Segment Format (adapted from [4])	22
4.10	An ESP32 development board (ESP32-S2-WROVER) used for this thesis	24
6.1	Conceptual example of active probing eliciting different responses. The prober sends an identical probe to Device A and Device B. Differences in their respective TCP/IP stack implementations lead to distinguishable responses (e.g., different TCP options or window sizes), which can be used for fingerprinting.	36
6.2	Conceptual overview of the proposed system architecture. The User interacts via a User Interface on a Host System, which controls the ESP32-based Network Probing Unit. The ESP32 interacts with devices on the Target Network.	38
7.1	ADDBA responses as captured with Wireshark of Google Pixel 7a Smartphone and a Shelly smart powerplug	45
7.2	Diagram of a Tuya packet.	52
8.1	Diagram of basic architecture.	59
8.2	Combined Host Discovery Interaction Sequence (ARP and TCP SYN)	71
8.3	Frontend UI: Configuration View.	76
8.4	Frontend UI: Host Discovery View.	76
8.5	Frontend UI: Device Fingerprinting View.	77
9.1	Example of Probe Request from Raspberry Pi 4 (Observation 1), showing a directed probe with numerous Vendor-Specific IEs.	87
9.2	Example of Probe Request from Raspberry Pi 4 (Observation 2), showing a broadcast probe with fewer Vendor-Specific IEs.	88
		105

9.3	Example of a broadcast Probe Request from a Tuya device.	89
9.4	Example of a unicast Probe Request to the AP from the same Tuya device.	89
9.5	ESP32 heap memory consumption across different fingerprinting operations.	92

List of Tables

7.1	Entropy and stability of Information Elements in Probe Request frames. The first column lists the Information Elements, while the second and third columns show the variability and stability values, respectively. The values are derived from Robyns et al.'s analysis [32].	47
7.2	Entropy Analysis of Top Parameters in nmap-os-db	50
9.1	Inventory of Devices in the Test Network	80
9.2	Coverage Test Results Comparison	83
9.3	Uniqueness Rates of Fingerprinting Methods	84
9.4	Stability and Reproducibility Metrics by Fingerprinting Method	85
9.5	Information Element Variability in Probe Requests	86



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acronyms

- AP** Access Point. 12, 44, 46, 59, 60, 79, 88
- ARP** Address Resolution Protocol. 7, 15, 37, 60, 80, 96
- BLE** Bluetooth Low Energy. 99
- BSD** Berkeley Software Distribution. 25, 67
- BSS** Basic Service Set. 12, 105
- CSMA/CA** Carrier Sense Multiple Access with Collision Avoidance. 12
- DL** Deep Learning. 31
- DNS** Domain Name System. 32
- DNS-SD** DNS-Based Service Discovery. 21, 68
- ESP-IDF** Espressif IoT Development Framework. 24, 37, 44, 61
- FCS** Frame Check Sequence. 13, 65
- FTP** File Transfer Protocol. 8
- HTTP** Hypertext Transfer Protocol. 8
- IEEE** Institute of Electrical and Electronics Engineers. 11, 35, 41, 60, 81, 95
- IEs** Information Elements. 14, 43, 60
- IoT** Internet of Things. 5, 7, 31, 35, 37, 41, 57, 67, 79, 95, 101
- IP** Internet Protocol. 5, 7, 16, 18, 19, 35, 41, 42, 58, 82, 95, 101
- ISP** Internet Service Provider. 79

LLC Logical Link Control. 11, 13

lwIP lightweight IP. 7, 25, 61, 96, 101

MAC Medium Access Control. 8, 11, 35, 41, 60, 81, 91

mDNS Multicast DNS. 21, 61, 81, 91, 92

ML Machine Learning. 31, 36

MSS Maximum Segment Size. 19, 36, 49

MTU Maximum Transmission Unit. 18

NIC Network Interface Controller. 11, 74

OSI Open Systems Interconnection. 7, 11

OUI Organizationally Unique Identifier. 74, 91

pbuf Packet Buffer. 25

PCB Protocol Control Block. 25

PHY Physical. 11

RF Radio Frequency. 32, 43

RSSI Received Signal Strength Indicator. 60

RTOS Real-Time Operating System. 97

SDK Software Development Kit. 25

SDR Software-Defined Radio. 32

SoC System-on-Chip. 6, 23, 35, 37, 41

SSID Service Set Identifier. 12, 60

STA Station. 11, 44, 59, 79

TCP Transmission Control Protocol. 5, 7, 16, 19, 32, 35, 41, 60, 80, 95, 101

TTL Time to Live. 18, 48

UDP User Datagram Protocol. 7, 18, 53, 60, 83

UI User Interface. 58

WLAN Wireless Local Area Network. 5, 7, 11, 35, 41, 43, 45, 51, 61, 82, 95, 101

Bibliography

- [1] Transforma Insights; Exploding Topics, “Anzahl der mit dem Internet der Dinge verbundenen Geräte weltweit bis 2030 [Number of devices connected to the Internet of Things worldwide by 2030].” <https://de.statista.com/statistik/daten/studie/1420315/umfrage/anzahl-der-iot-geraete-weltweit/>.
- [2] P. Yadav, A. Feraudo, B. Arief, S. F. Shahandashti, and V. G. Vassilakis, “Position paper: A systematic framework for categorising IoT device fingerprinting mechanisms,” in *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, (Virtual Event Japan), pp. 62–68, ACM, Nov. 2020.
- [3] M. B. Stöger, *Scanning for IoT Devices Using a Mobile Android Application: Detecting Devices in IP-based Networks and the Bluetooth Environment*. Suchen von IoT Geräten Mittels Einer Mobilen Android Applikation: Erkennen von Geräten in IP-basierten Netzwerken Und Über Bluetooth, Wien: Wien, 2023.
- [4] C. M. Kozierek, *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference*. San Francisco: No Starch Press, 2005.
- [5] U. User:Kbrose, “English: Encapsulation of application data descending through the layered IP architecture,” July 2008.
- [6] M. Gast, *802.11 Wireless Networks: The Definitive Guide*. Beijing ; Farnham: O’Reilly, 2nd ed ed., 2005.
- [7] Buhadram, “English: 802.11 Generic MAC Frame,” Mar. 2018.
- [8] D. Plummer, “Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware,” Tech. Rep. RFC0826, RFC Editor, Nov. 1982.
- [9] S. Cheshire and M. Krochmal, “Multicast DNS,” Tech. Rep. RFC6762, RFC Editor, Feb. 2013.
- [10] S. Cheshire and M. Krochmal, “DNS-Based Service Discovery,” Tech. Rep. RFC6763, RFC Editor, Feb. 2013.

- [11] “ESP32 Wi-Fi & Bluetooth SoC | Espressif Systems.” <https://www.espressif.com/en/products/socs/esp32>.
- [12] “ESP DevKits | Espressif Systems.” <https://www.espressif.com/en/products/devkits>.
- [13] “ESP-IDF Programming Guide - ESP32 - — ESP-IDF Programming Guide v5.4.1 documentation.” <https://docs.espressif.com/projects/esp-idf/en/stable/esp32/index.html>.
- [14] PlatformIO, “PlatformIO: Your Gateway to Embedded Software Development Excellence.” <https://platformio.org>.
- [15] A. Dunkels and lwIP Developers, “lwIP - A Lightweight TCP/IP stack - Summary [Savannah].” <https://savannah.nongnu.org/projects/lwip/>.
- [16] “lwIP - ESP32 - — ESP-IDF Programming Guide latest documentation.” <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/lwip.html>.
- [17] “lwIP: Overview.” https://www.nongnu.org/lwip/2_1_x/.
- [18] “Chapter 7. Sockets.” <https://docs.freebsd.org/en/books/developers-handbook/sockets/>.
- [19] “ESP-NETIF - ESP32 - — ESP-IDF Programming Guide v5.4.1 documentation.” https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/network/esp_netif.html#e-esp-netif-l2-tap-interface.
- [20] R. R. Chowdhury and P. E. Abas, “A survey on device fingerprinting approach for resource-constraint IoT devices: Comparative study and research challenges,” *Internet of Things*, vol. 20, p. 100632, Nov. 2022.
- [21] P. M. S. Sanchez, J. M. J. Valero, A. H. Celdran, G. Bovet, M. G. Perez, and G. M. Perez, “A Survey on Device Behavior Fingerprinting: Data Sources, Techniques, Application Scenarios, and Datasets,” *IEEE Communications Surveys & Tutorials*, vol. 23, no. 2, pp. 1048–1077, 2021.
- [22] A. Sivanathan, H. H. Gharakheili, F. Loi, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman, “Classifying IoT Devices in Smart Environments Using Network Traffic Characteristics,” *IEEE Transactions on Mobile Computing*, vol. 18, pp. 1745–1759, Aug. 2019.
- [23] Y. Meidan, M. Bohadana, A. Shabtai, M. Ochoa, N. O. Tippenhauer, J. D. Guarnizo, and Y. Elovici, “Detection of Unauthorized IoT Devices Using Machine Learning Techniques,” Sept. 2017.
- [24] L. Bai, L. Yao, S. S. Kanhere, X. Wang, and Z. Yang, “Automatic Device Classification from Network Traffic Streams of Internet of Things,” Dec. 2018.

- [25] J. Ortiz, C. Crawford, and F. Le, “DeviceMien: Network device behavior modeling for identifying unknown IoT devices,” in *Proceedings of the International Conference on Internet of Things Design and Implementation*, (Montreal Quebec Canada), pp. 106–117, ACM, Apr. 2019.
- [26] N. Soltanieh, Y. Norouzi, Y. Yang, and N. C. Karmakar, “A Review of Radio Frequency Fingerprinting Techniques,” *IEEE Journal of Radio Frequency Identification*, vol. 4, pp. 222–233, Sept. 2020.
- [27] L. Polčák and B. Franková, “On Reliability of Clock-skew-based Remote Computer Identification:,” in *Proceedings of the 11th International Conference on Security and Cryptography*, (Vienna, Austria), pp. 291–298, SCITEPRESS - Science and Technology Publications, 2014.
- [28] R. Perdisci, T. Papastergiou, O. Alrawi, and M. Antonakakis, “IoTFinder: Efficient Large-Scale Identification of IoT Devices via Passive DNS Traffic Analysis,” in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, (Genoa, Italy), pp. 474–489, IEEE, Sept. 2020.
- [29] J. Sun, K. Sun, and C. Shenefiel, “Automated IoT Device Fingerprinting Through Encrypted Stream Classification,” in *Security and Privacy in Communication Networks* (S. Chen, K.-K. R. Choo, X. Fu, W. Lou, and A. Mohaisen, eds.), vol. 304, pp. 147–167, Cham: Springer International Publishing, 2019.
- [30] M. Hasan, M. M. Islam, M. I. I. Zarif, and M. Hashem, “Attack and anomaly detection in IoT sensors in IoT sites using machine learning approaches,” *Internet of Things*, vol. 7, p. 100059, Sept. 2019.
- [31] “Chapter 8. Remote OS Detection | Nmap Network Scanning.” <https://nmap.org/book/osdetect.html#osdetect-intro>.
- [32] P. Robyns, B. Bonné, P. Quax, and W. Lamotte, “Noncooperative 802.11 MAC Layer Fingerprinting and Tracking of Mobile Devices,” *Security and Communication Networks*, vol. 2017, pp. 1–21, 2017.
- [33] C. Matte, *Wi-Fi tracking: Fingerprinting attacks and counter-measures*. PhD thesis, Université de Lyon, Networking and Internet Architecture, 2017.
- [34] S. Al-Hazbi, A. Hussain, S. Sciancalepore, G. Oligeri, and P. Papadimitratos, “Radio Frequency Fingerprinting via Deep Learning: Challenges and Opportunities,” Apr. 2024.
- [35] J. Franklin, D. McCoy, P. Tabriz, V. Neagoe, J. V. Randwyk, and D. Sicker, “Passive data link layer 802.11 wireless device driver fingerprinting.,” in *USENIX Security Symposium*, vol. 3, pp. 16–89, 2006.

- [36] C. Maurice, S. Onno, C. Neumann, O. Heen, and A. Francillon, “Improving 802.11 fingerprinting of similar devices by cooperative fingerprinting,” in *2013 International Conference on Security and Cryptography (SECRYPT)*, pp. 1–8, 2013.
- [37] J. Devreker and S. Neuenhausen, “ESP32 open MAC.” <https://esp32-open-mac.be/>.
- [38] S. Bratus, C. Cornelius, D. Kotz, and D. Peebles, “Active behavioral fingerprinting of wireless devices,” in *Proceedings of the First ACM Conference on Wireless Network Security*, (Alexandria VA USA), pp. 56–61, ACM, Mar. 2008.
- [39] “Scapy: Scapy: Interactive packet manipulation tool.” <https://scapy.net>.
- [40] L. Pintor and L. Atzori, “Analysis of Wi-Fi Probe Requests Towards Information Element Fingerprinting,” in *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, (Rio de Janeiro, Brazil), pp. 3857–3862, IEEE, Dec. 2022.
- [41] Michal Zalewski, “P0f v3.” <https://lcamtuf.coredump.cx/p0f3/>.
- [42] P. Auffret, “SinFP, unification of active and passive operating system fingerprinting,” *Journal in Computer Virology*, vol. 6, pp. 197–205, Aug. 2010.
- [43] K. Yang, Q. Li, and L. Sun, “Towards automatic fingerprinting of IoT devices in the cyberspace,” *Computer Networks*, vol. 148, pp. 318–327, Jan. 2019.
- [44] F. Zhou, H. Qu, H. Liu, H. Liu, and B. Li, “Fingerprinting IIoT Devices Through Machine Learning Techniques,” *Journal of Signal Processing Systems*, vol. 93, pp. 779–794, July 2021.
- [45] X. Feng, Q. Li, H. Wang, and L. Sun, “Acquisitional rule-based engine for discovering internet-of-thing devices,” in *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC’18, (Baltimore, MD, USA and USA), pp. 327–341, USENIX Association, 2018.
- [46] J. Cox, “TINYTUYA.” <https://github.com/jasonacox/tinytuya>, Jan. 2026.
- [47] rospogrigio, “Localtuya.” <https://github.com/rospogrigio/loclaltuya>, May 2025.
- [48] uzlonewolf, “Protocol notes · jasonacox/tinytuya · Discussion #260.” <https://github.com/jasonacox/tinytuya/discussions/260>.
- [49] “Basic Features-Tuya Developer Platform-Tuya Developer.” <https://developer.tuya.com/en/docs/iot/basic-features?id=Kaumiwx9b4kkp>.
- [50] “lwIP: Src/core/ipv4/etharp.c File Reference.” https://www.nongnu.org/lwip/2_1_x/etharp_8c.html.
- [51] risinek, “ESP32 Wi-Fi Penetration Tool.” <https://github.com/risinek/esp32-wifi-penetration-tool>, Feb. 2023.

- [52] “Chapter 3. Host Discovery (“Ping Scanning”) | Nmap Network Scanning.”
<https://nmap.org/book/host-discovery.html>.
- [53] “Well Known Port List: Nmap-services | Nmap Network Scanning.”
<https://nmap.org/book/nmap-services.html>.