DOKUZ EYLÜL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

ENHANCED AUTHENTICATION IN WIRELESS PERSONAL AREA NETWORKS

by
Süleyman Yiğit KAYGISIZ

February, 2026
İZMİR

ENHANCED AUTHENTICATION IN WIRELESS PERSONAL AREA NETWORKS

A Thesis Submitted to the
Graduate School of Natural And Applied Sciences of Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering, Computer Engineering Program

by

Süleyman Yiğit KAYGISIZ

February, 2026

İZMİR

M.Sc. THESIS EXAMINATION RESULT FORM

We have read the thesis entitled "**ENHANCED AUTHENTICATION IN WIRELESS PERSONAL AREA NETWORKS**" completed by **SÜLEYMAN YİĞİT KAYGISIZ** under supervision of **PROF. DR. GÖKHAN DALKILIÇ** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

.....

Prof. Dr. Gökhan DALKILIÇ

Supervisor

.....

Assoc. Prof. Dr. Yunus DOĞAN

Jury Member

.....

Assoc. Prof. Dr. Ömer AYDIN

Jury Member

Prof. Dr. Abdullah SEÇGİN
Director
Graduate School of Natural and Applied Sciences

ACKNOWLEDGEMENTS

I am sincerely indebted to my supervisor, Prof. Dr. Gökhan DALKILIÇ, for his guidance, inputs, and supportive feedback throughout my graduate education and during the writing of this master's thesis. His knowledge and patience were a very important part in influencing my research.

I am profoundly thankful to my parents for their unconditional love, continuous support, and for always trusting in me throughout my education.

Lastly, I would like to specially thank my girlfriend, Melike, for being my greatest supporter. Her patience, cheerfulness, and constant encouragement helped me being continuously motivated and successfully completing this master's thesis.

Süleyman Yiğit KAYGISIZ

ENHANCED AUTHENTICATION IN WIRELESS PERSONAL AREA NETWORKS

ABSTRACT

The Internet of Things (IoT) has changed our daily lives by integrating numerous devices into and delivering smart automation in fields like healthcare, smart cities, agriculture, and industrial manufacturing. Nonetheless, the rapid growth in the IoT has been met with many challenges regarding device security, authentication, and the integrity of data. These issues are mostly due to the limited computing capabilities of IoT devices and the use of inadequate authentication schemes. Battery-powered, resource-constrained IoT devices cannot use conventional cryptographic approaches that have complex computations. To overcome this issue, these cryptographic calculations are offloaded to a powerful gateway server. Due to limitations, using a gateway device may not be possible. Additionally, placing the gateway server unattended can increase security risks.

This thesis proposes a lightweight mutual authentication scheme for IoT devices that is not reliant on a gateway server during the authentication process. The scheme uses Elliptic Curve Diffie-Hellman (ECDH) for secure key exchange and HMAC to verify the integrity of transmitted messages. The scheme uses a distributed trust ledger model, where each device has a local ledger that contains trusted devices' hashed IDs, distributed by a trusted gateway before the initial handshake. Initial messages are encrypted with Pre-Shared Key (PSK) until devices establish a mutual secret key using ECDH; the PSK is also distributed by a trusted gateway before the auth session. The protocol switches between BLE and Wi-Fi channels to balance battery life and communication speed while also increasing authentication's trust, security, and performance. Merkle trees facilitate tamper-proof ledgers that devices can verify data integrity efficiently combined with Bloom filter lookups. The scheme is formally verified using the BAN logic representation and the Scyther formal verification tool, proving mutual authentication and establishing a secure session key

with forward and backward secrecy.

The scheme is tested on the ESP32-WROOM-32 microcontroller. We demonstrate that our proposed scheme is a secure, scalable, and lightweight authentication scheme for various IoT applications without requiring a gateway device during authentication processes. While mitigating attacks such as Man-in-the-Middle (MitM), replay, DoS, side-channel attacks, key escrow, physical tampering, eavesdropping, and impersonation. In real-world test runs, the scheme reliably performed as expected, validating its reliability and robustness. The results show that our proposed scheme offers a secure, lightweight authentication solution for resource-constrained IoT devices.

Keywords: Internet of things, Lightweight mutual authentication, ESP32 authentication scheme, ECDH, Distributed trusted ledger, Bloom filters, Merkle trees

KABLOSUZ KİŞİSEL ALAN AĞLARINDA GELİŞMİŞ KİMLİK DOĞRULAMA

ÖZ

Nesnelerin İnterneti (IoT) sağlık hizmetleri, akıllı şehirler, tarım ve endüstriyel üretim gibi alanlarda birçok çeşitli cihazı entegre ederek ve akıllı otomasyonu sağlayarak günlük yaşamımızı değiştirdi. IoT'deki hızlı entegrasyon, cihaz güvenliği, kimlik doğrulama ve veri bütünlüğü konularında birçok sorun açığa çıkarmıştır. Bu sorunlar, çoğunlukla IoT cihazlarının sınırlı hesaplama yetenekleri ve yetersiz kimlik doğrulama şemalarının kullanılmasından kaynaklanmaktadır. Pil gücüyle çalışan, kaynak kısıtlı IoT cihazları, geleneksel karmaşık hesaplamalar gerektiren kriptografik yaklaşımları kullanamaz. Bu sorunu aşmak için karmaşık hesaplama işlemleri güçlü bir ağ geçidi sunucusu üzerinden çözülmeye çalışılır. Sınırlamalar nedeniyle, bir ağ geçidi cihazı kullanmak mümkün olmayabilir. Ayrıca, ağ geçidi sunucusunun denetimsiz bırakılması güvenlik risklerini artırabilir.

Bu tez, kimlik doğrulama süreci sırasında bir ağ geçidi sunucusuna bağımlı olmayan, IoT cihazları için hafif bir karşılıklı kimlik doğrulama şeması önermektedir. Şema, güvenli anahtar değişimi için Eliptik Eğri Diffie-Hellman (ECDH) ve iletilen mesajların bütünlüğünü doğrulamak için HMAC kullanır. Şema, her cihazın başlangıçtaki el sıkışmasından önce, güvenilir bir ağ geçidi tarafından dağıtılan güvenilir cihazların hashlenmiş kimliklerini içeren yerel bir deftere sahip olduğu dağıtılmış bir dağıtık defter teknolojisi kullanır. İlk mesajlar, cihazlar ECDH kullanarak karşılıklı bir gizli anahtar oluşturana kadar Önceden Paylaşılan Anahtar (PSK) ile şifrelenir; PSK kimlik doğrulama oturumundan önce güvenilir bir ağ geçidi tarafından cihazlara dağıtılır. Protokol, pil ömrü ve iletişim hızını dengelemek için BLE ve Wi-Fi kanalları arasında geçiş yaparak kimlik doğrulamanın güvenilirliğini, mesajların güvenliğini ve performansını artırır. Merkle ağaçları, cihazların veri bütünlüğünü Bloom filtreleri ile verimli bir şekilde doğrulayabileceği, değiştirilmesi imkânsız defterleri sağlar. Şemanın, BAN mantık temsili ve Scyther resmi doğrulama

aracı kullanılarak karşılıklı kimlik doğrulama sağladığı ve ileri ve geri gizlilikle güvenli bir oturum anahtarı oluşturduğu kanıtlanmıştır.

Şemamız, ESP32-WROOM-32 mikrodenetleyicisinde test edilmiştir. Önerdiğimiz şemanın, kimlik doğrulama süreçleri sırasında bir ağ geçidi cihazı gerektirmeden çeşitli IoT uygulamaları için güvenli, ölçeklenebilir ve hafif bir kimlik doğrulama şeması olduğunu gösteriyoruz. Ortadaki Adam (MitM), tekrar oynatma, DoS, yan kanal saldırıları, anahtar saklama, fiziksel müdahale, dinleme ve taklit gibi saldırılara karşı şemamızın korunduğunu gösteriyoruz. Gerçek dünya testlerinde, şema beklendiği gibi güvenilir bir şekilde performans göstererek güvenilirliğini ve sağlamlığını doğrulamıştır. Sonuçlar, önerdiğimiz şemanın kısıtlı kaynağı olan IoT cihazları için güvenli, hafif bir kimlik doğrulama çözümü sunduğunu göstermektedir.

Anahtar kelimeler: Nesnelerin interneti, Hafif karşılıklı kimlik doğrulama, ESP32 kimlik doğrulama şeması, ECDH, Dağıtılmış güvenilir defter, Bloom filtreleri, Merkle ağaçları

CONTENTS

Page

M.Sc. THESIS EXAMINATION RESULT FORM ii
ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZ vi
LIST OF FIGURES xv
LIST OF TABLES xviii
ABBREVIATIONS xix

CHAPTER ONE – INTRODUCTION 1

1.1 Study Area 1
1.2 Motivation 3
1.3 Scope and Contributions 5

CHAPTER TWO – BACKGROUND 9

2.1 Foundational Technologies 9
2.1.1 Internet of Things (IoT) 9
2.1.2 Wireless Personal Area Networks (WPANs) 10
2.1.3 Bluetooth Low Energy (BLE) 10
2.1.4 Wireless Fidelity (Wi-Fi) 11
2.1.5 Zigbee 11
2.1.6 Thread 12
2.1.7 ESP32 12
2.1.8 Secure Boot 12
2.1.9 Flash Encryption 13
2.1.10 Mesh Network 13
2.1.11 Gateway Device 14
2.1.12 Symmetric Key Encryption 14

2.1.13 Advanced Encryption Standard Galois Counter Mode (AES-GCM) . 15
2.1.14 Public Key Encryption (PKE) 15
2.1.15 Public Key Infrastructure (PKI) 16
2.1.16 Elliptic Curve Cryptography (ECC) 16
2.1.17 Elliptic Curve Diffie-Hellman Algorithm (ECDH) 17
2.1.18 Hash-based Message Authentication Code (HMAC) 17
2.1.19 HMAC-based KDF (HKDF) 18
2.1.20 Pre-Shared Key (PSK) 19
2.1.21 Blockchain 19
2.1.22 Distributed Ledger 20
2.1.23 Merkle Tree and Merkle Root 20
2.1.24 Bloom Filters 21
2.2 Related Works 21

CHAPTER THREE – SYSTEM DESIGN AND METHODOLOGY 34

3.1 Scheme Assumptions and Device Initial States 34
3.1.1 Pre-Shared Key 34
3.1.2 Ledger Model 35
3.1.3 No Online Authentication Server 35
3.1.4 Device Capabilities 35
3.1.5 Communication Channels 35
3.1.6 Nonce Generation 36
3.1.7 Timeout and Timekeeping Assumption 36
3.2 Design Goals 37
3.2.1 Mutual Authentication 37
3.2.2 Secure Session Key Establishment 37
3.2.3 Forward Secrecy 38
3.2.4 Using Distributed Ledger Technology 38
3.2.5 Lightweight Operations 38
3.2.6 Comprehensive Security 39
3.3 Scheme Overview 39

3.3.1 Initial State 42
3.4 Phase 1: Initial Handshake and Temporary Session Key Generation 42
3.4.1 Step 1.1: BLE Connection Establishment & Start Timer 43
3.4.2 Step 1.2: Temporary Key, Nonce and HMAC Generation 43
3.4.3 Step 1.3: Initial Key Exchange and HMAC Verification 44
3.4.4 Step 1.4: Response Key Exchange and HMAC Verification 45
3.4.5 Step 1.5: Temporary Session Key Derivation 46
3.4.6 Step 1.6: Timeout Check 47
3.5 Phase 2: Initial Ledger Verification 48
3.5.1 Step 2.1: Hashed ID Exchange 48
3.5.2 Step 2.2: Ledger Lookup 49
3.5.3 Step 2.3: Result Comparison 49
3.5.4 Step 2.4: Timeout Check 50
3.6 Phase 3: Merkle Tree Verification 51
3.6.1 Step 3.1: Merkle Root Exchange 51
3.6.2 Step 3.2: Merkle Roots Equality Comparison 52
3.6.3 Step 3.3: Timeout Check 52
3.7 Phase 4: Merkle Tree Reconciliation 53
3.7.1 Step 4.1: Wi-Fi Channel Switch and Iterative Proof Exchange 53
3.7.2 Step 4.2: Trust Threshold Evaluation 55
3.7.3 Step 4.3: Timeout Check 55
3.8 Phase 5: Finalizing Authentication and New Session Key Derivation 56
3.8.1 Step 5.1: New Session Key Derivation 56
3.8.2 Step 5.2: Final Timeout Check 57
3.9 BAN Logic Analysis 58
3.9.1 Notations 58
3.9.2 Initial State Assumptions 59
3.9.3 Implementation Assumptions 59
3.9.4 Scheme Message Formalization 60
3.9.5 Phase 1: Initial Handshake (Key Agreement) 60
3.9.6 Phase 2: Ledger Verification (Identity Exchange) 61
3.9.7 Phase 3: Merkle Root Exchange 62

3.9.8 Phase 4: Merkle Tree Reconciliation 63
3.9.9 Phase 5: Final Key Derivation and Confirmation 63
3.9.10 Security Properties Proven 65
3.9.10.1 Mutual Authentication 65
3.9.10.2 Key Freshness 65
3.9.10.3 Forward Secrecy 66
3.9.10.4 Replay Protection 66
3.9.11 Message Summary 66

CHAPTER FOUR – SCHEME IMPLEMENTATION AND TESTING 68

4.1 Implementation Overview 69
4.1.1 Message Type Classification 69
4.1.2 SHA-256 Hash Function 70
4.1.3 Shared ECDH Computation 70
4.1.4 AES-256-GCM 72
4.1.5 HMAC-Based Message Authentication 73
4.1.6 HMAC-Based Key Derivation (HKDF) 73
4.1.7 Bloom Filter Device Lookup 74
4.1.8 Merkle Tree Construction 75
4.1.9 Merkle Tree Proof Generation and Verification 76
4.1.10 BLE Communication Layer 77
4.1.11 WiFi Channel Switching 79
4.1.12 State Management 80
4.1.13 Restart and Recovery Mechanism 81
4.1.14 Metrics Collection and Energy Estimation 82
4.1.15 Secure Memory Handling 82
4.1.16 Device Ledger Management 83
4.2 Testing 84
4.2.1 Wokwi Simulation 84
4.2.2 Real World Standalone Testing 85
4.2.3 Real World Two-Device Testing 86

4.3 Metrics and Analysis 86

CHAPTER FIVE – SECURITY ANALYSIS AND PERFORMANCE EVALUATION 90

5.1 Security Terms 90
5.1.1 Confidentiality 90
5.1.2 Integrity 91
5.1.3 Mutual Authentication 91
5.1.4 Secure Session Key Agreement 92
5.1.5 Forward/Backward Secrecy 92
5.1.6 Formal Verification 92
5.2 Security Attack Types 93
5.2.1 Replay Attacks 93
5.2.2 Impersonation Attacks 93
5.2.3 Man-in-the-Middle Attacks 94
5.2.4 Key Escrow 94
5.2.5 Denial of Service (DoS) Attacks 95
5.2.6 Side-channel Attacks 95
5.2.7 Eavesdropping 96
5.2.8 Physical Tampering 96
5.3 Our Countermeasures Against Specific Attacks 97
5.3.1 Replay Attack 97
5.3.2 Impersonation Attack 97
5.3.3 Man-in-the-Middle (MitM) Attack 98
5.3.4 Key Escrow 98
5.3.5 DoS Attack Resistance 98
5.3.6 Side-Channel Attack 99
5.3.7 Eavesdropping 99
5.3.8 Physical Tampering 100
5.4 Lightweight Design 100
5.4.1 Efficient Cryptography 101

5.4.2 Optimized Data Structures 101
5.4.3 Communication Efficiency 101
5.4.4 Computational Efficiency 101
5.4.5 Energy Efficiency 101
5.4.6 Reduced Storage Overhead 102
5.4.7 Hardware Acceleration 102
5.5 Formal Security Tool Analysis 102
5.5.1 Scyther Tool Features 102
5.5.1.1 Verification and Falsification 102
5.5.1.2 Problem Identification 103
5.5.1.3 Model Checking 103
5.5.2 Security Properties 103
5.5.2.1 Alive 103
5.5.2.2 Nisynch (Non-injective Synchronization) 103
5.5.2.3 Niagree (Non-injective Agreement) 103
5.5.2.4 Weak-agree 104
5.5.2.5 Secret (Confidentiality) 104
5.5.2.6 SKR (Session Key Reveal/Secrecy of Session Key) 104
5.5.2.7 Support for Sessions 104
5.5.3 Usage and Workflow 104
5.5.3.1 Protocol Description Language (SPDL) 104
5.5.3.2 Claim Events 105
5.5.3.3 Graphical User Interface and Command-Line 105
5.5.3.4 Analysis Modes 105
5.5.4 Limitations of SPDL 105
5.5.5 Our Scheme's SPDL Representation 107

CHAPTER SIX – CONCLUSION AND FUTURE WORK 112

APPENDICES 118

Appendix 1: Scyther Complete Code 119

Appendix 2: ESP32-WROOM-32 Detailed Code Snippets..... 121
Appendix 3: Wokwi Simulation Output 145
Appendix 4: ESP32-WROOM-32 Standalone Output..... 149
Appendix 5: ESP32-WROOM-32 Two-device Testing Output 153

LIST OF FIGURES

Page

Figure 1.1 Application areas of IoT technology (generated by Gemini Nano Banana Pro) 2
Figure 1.2 Smart devices controlled via smartphone as IoT gateway (generated by Gemini Nano Banana Pro) 3
Figure 1.3 Widespread usage domain examples of IoT (generated by Gemini Nano Banana Pro) 4
Figure 1.4 Some application examples of IoT Part 1 (generated by Gemini Nano Banana Pro) 6
Figure 1.5 Some application examples of IoT Part 2 (generated by Gemini Nano Banana Pro) 7
Figure 3.1 Initial state of devices before auth session 42
Figure 3.2 BLE handshake and timer starts 43
Figure 3.3 Temporary key, nonce and HMAC generation 44
Figure 3.4 Initial device key exchange and HMAC verification 45
Figure 3.5 Responding device key exchange and HMAC verification 46
Figure 3.6 Temporary shared session key derivation 47
Figure 3.7 Timeout check 47
Figure 3.8 Hashed ID exchange 48
Figure 3.9 Ledger lookup 49
Figure 3.10 Result comparison 50
Figure 3.11 Timeout check 50
Figure 3.12 Merkle root exchange 51
Figure 3.13 Merkle roots equality comparison 52
Figure 3.14 Timeout check 53
Figure 3.15 Wi-Fi channel switch and iterative proof exchange 54
Figure 3.16 Trust threshold evaluation 55
Figure 3.17 Timeout check 56
Figure 3.18 New session key derivation 57

Figure 3.19 Final timeout check 58
Figure 4.1 Message type classification 69
Figure 4.2 SHA-256 hash function..... 70
Figure 4.3 Key pair generation 70
Figure 4.4 Shared ECDH computation 71
Figure 4.5 AES-256-GCM encryption..... 72
Figure 4.6 AES-256-GCM decryption..... 72
Figure 4.7 HMAC functions..... 73
Figure 4.8 HKDF implementation 73
Figure 4.9 HKDF usage in phase 1 74
Figure 4.10 Bloom filter positions 74
Figure 4.11 Bloom filter operations 75
Figure 4.12 Merkle tree build..... 75
Figure 4.13 Merkle tree proof generation 76
Figure 4.14 Merkle proof verification 76
Figure 4.15 BLE server initialization 77
Figure 4.16 BLE message reception..... 78
Figure 4.17 WiFi channel switching..... 79
Figure 4.18 State structure 80
Figure 4.19 Restart handler 81
Figure 4.20 Phase metrics 82
Figure 4.21 Secure zero function 82
Figure 4.22 Secure cleanup usage..... 83
Figure 4.23 Ledger entry..... 83
Figure 4.24 Add device to ledger..... 83
Figure 4.25 Lookup devices in ledger 84
Figure 4.26 Wokwi simulation output 85
Figure 4.27 ESP32 standalone output..... 85
Figure 4.28 ESP32 two device testing output..... 86
Figure 5.1 Declarations..... 107
Figure 5.2 Role definitions..... 108
Figure 5.3 Claims..... 109

Figure 5.4 Verified scheme 110
Figure 5.5 Verified automatic claims 111

LIST OF TABLES

Page

Table 2.1 Security features and technologies 31
Table 2.2 Attack resilience analysis 33
Table 3.1 Summary of notations 41
Table 3.2 Complete scheme message summary 67
Table 4.1 Test average metrics 87

ABBREVIATIONS

WPAN	: Wireless Personal Area Network
WLAN	: Wireless Local Area Network
WBAN	: Wireless Body Area Network
IoT	: Internet of Things
IIoT	: Industrial Internet of Things
IoMT	: Internet of Medical Things
MitM	: Man-in-the-Middle Attack
PKI	: Public Key Infrastructure
BLE	: Bluetooth Low Energy
Wi-Fi	: Wireless Fidelity
6LoWPAN	: IPv6 over Low-Power Wireless Personal Area Network
AES	: Advanced Encryption Standard
AES-GCM	: Advanced Encryption Standard Galois Counter Mode
RG	: Relay Gateway
MG	: Main Gateway
CA	: Certificate Authority
ECC	: Elliptic Curve Cryptography
ECDSA	: Elliptic Curve Digital Signature Algorithm
ECDH	: Elliptic Curve Diffie-Hellman
MAC	: Message Authentication Code
HMAC	: Hash-based Message Authentication Code
HKDF	: HMAC Based Key Derivation Function
PUF	: Physical Unclonable Function
VDT	: Vehicular Digital Twin
NFT	: Non-Fungible Tokens
PSK	: Pre-Shared Key
BAN	: Burrows-Abadi-Needham Logic
HOTP	: HMAC-based One-Time Passwords

CHAPTER ONE
INTRODUCTION

The advancements in the Internet of Things (IoT) have changed technology by making it easier to connect, automate, and share information in many diverse areas. As IoT devices become a progressively important part of our foundation and everyday habits, it is more significant than ever to use secure, effective procedures for them to connect and authenticate. This chapter provides an outline on the matter, describes why secure IoT authentication is necessary, and outlines the main objectives and contributions of this thesis.

1.1 Study Area

Wireless Personal Area Networks (WPANs) allow short-range, energy-efficient communication. In contrast to broader-coverage Wireless Local Area Networks (WLANs), WPANs are optimized for close-proximity communications. It is built on the IEEE 802.15.4 standard; WPAN technologies such as Bluetooth Low Energy (BLE) and ZigBee support uninterrupted data transfer between nearby devices while decreasing power use. Through these networks, endpoint devices, also known as edge devices, can connect with gateways and higher-level infrastructure. This model forms the ground layer of the IoT ecosystem. The IoT ecosystems are composed of low-power computing devices, incorporating all types of sensors, cameras, microcontrollers, RFID tags, and smart cards (Şeker, 2024).

The figures in Chapter 1 were produced using the Gemini-3-Pro-Image-Preview image generation model, also referred to as Google's Nano Banana Pro model (Google DeepMind, 2025). IoT devices are now essential in our daily lives, as shown in Figure 1.1. Building on our knowledge of their wide existence, we see that they can radically transform sectors such as smart cities, smart homes, remote healthcare, and industrial automation. These devices collect and exchange important data, including temperature, blood pressure, and heart rate, which considered personal information,

must be shared securely.



Figure 1.1 Application areas of IoT technology (generated by Gemini Nano Banana Pro)

A typical IoT device has a low-power processor with limited computational capability. It also has low energy consumption and limited storage capacity. Traditional encryption algorithms are challenging or, in some cases, infeasible to implement on these devices. This leads to privacy and security challenges, making new authentication schemes critical. The IoT has seen major changes recently. Internet-connected devices and sensors have become part of daily life. These devices need to communicate to share information or perform certain tasks.

Breakthroughs in microchip technology have allowed affordable Internet-connected devices and sensors, as shown in Figure 1.2. This has made IoT products and systems popular in everyday life. However, as more devices are deployed, they become targets for malicious adversaries who exploit weaknesses. IoT devices commonly lack computing power, memory, and energy. This makes them vulnerable to security threats. Dealing with these problems is vital. To prevent fraud or exploitation, authentication must make certain that both participants are genuine and trustworthy.



Figure 1.2 Smart devices controlled via smartphone as IoT gateway (generated by Gemini Nano Banana Pro)

These devices can operate in public, in some cases dangerous surroundings without reliable security, leaving them susceptible to both active and passive attacks. Particularly, adversaries can intercept to perform eavesdropping, replay, and MITM attacks. Additionally, without appropriate authentication, an attacker can impersonate a node to inject deceitful data or impersonate a gateway to gather confidential data. These attacks can be very costly, exposing private data and damaging important systems and industries.

1.2 Motivation

The rapid expansion of IoT and Industrial IoT (IIoT), as shown in Figure 1.3, brings these technological advancements into higher-risk fields such as healthcare and industry. This growth increases the risk of security and privacy vulnerabilities. These devices are unable to use traditional encryption schemes due to their limited hardware. New methodologies that are adapted to the limits of these devices must therefore be considered.

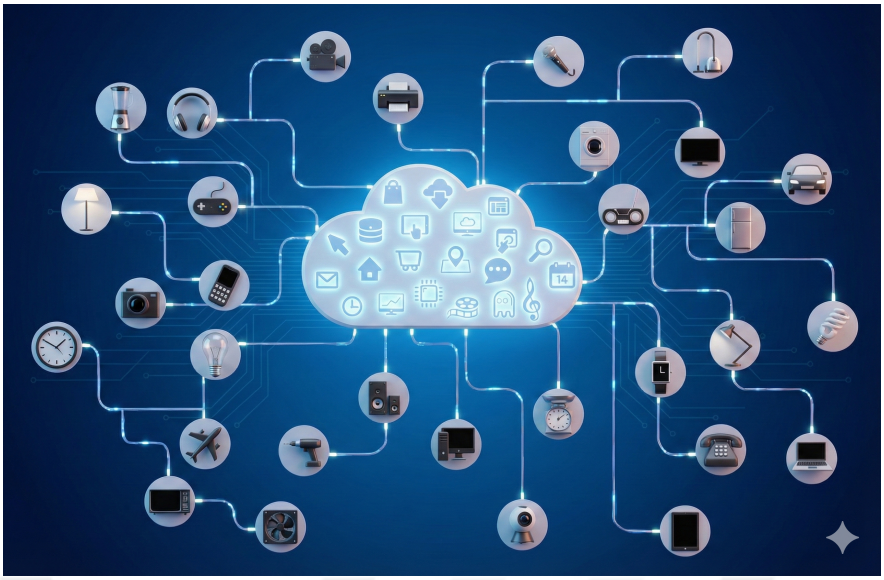


Figure 1.3 Widespread usage domain examples of IoT (generated by Gemini Nano Banana Pro)

The significant effects of IoT and the pressing demand for robust authentication are increasingly obvious, underlining the weaknesses of traditional security approaches. Many existing authentication approaches either require too much power for basic sensors or weaken security, leaving devices vulnerable. In addition, basic password-based protections remain popular, even though they provide only limited protection against even the most fundamental and widely known threats. The extensive number of devices, frequently positioned in easily accessed areas, presents an enormous obstacle: poor default credentials and lack of encrypted memory security make them particularly susceptible to brute-force and dictionary attacks. In addition, managing passwords across numerous devices is impractical. Discovering a vulnerability suddenly transforms the process of updating all hard-coded passwords across several devices into a tedious and resource-demanding task.

Similarly, implementing Public Key Infrastructure (PKI) presents significant obstacles within the IoT framework. Conventional PKI systems are complex and demand complex procedures on constrained devices. The security features required for PKI, including certificate management, validation, secure key distribution, storage, and revocation, impose substantial demands on the processing, memory, and energy resources of IoT nodes. The reliable security provided by PKI frequently

conflicts with the limitations of IoT devices, necessitating the exploration of alternate, more suitable authentication methods.

Furthermore, most authentication schemes depend on a centralized server, usually a reliable computer, to carry out computational and cryptographic tasks. In certain deployments, a central server may be unavailable; even when there is one, it is prone to fail. To address these issues, we present an innovative mutual authentication scheme designed for the capabilities of commonly used IoT devices. The scheme's security is thoroughly verified with the Scyther tool against established attack models. Additionally, we demonstrate a working prototype developed on the ESP32 platform utilizing the Arduino framework and open-source libraries, demonstrating the practical feasibility of our approach.

1.3 Scope and Contributions

We focus on mutual authentication of IoT devices within a peer-to-peer network. In smart homes, IoT devices like locks, lights, and thermostats cooperate to create an integrated security system. Automobiles analyze their systems and conduct diagnostics, suitable to both regular operations and automated vehicle technologies. They interact with each other to facilitate coordinated decisions. Wearable devices, such as smartwatches, health trackers, and diabetes sensors, must be verified with each other to enable safe data exchange. In agriculture, it can be applied to a humidity sensor for an indoor plant or a network of remotely controlled sensors and water valves distributed throughout an extensive field. In residential and industrial settings, IoT devices monitor energy production from solar panels. In urban areas, autonomous IoT systems and city grids collaborate to regulate traffic and enhance energy efficiency. The examples illustrated in Figure 1.4 and Figure 1.5 demonstrate the wide utilization and integration of IoT devices. They highlight the significance of authentication across applications.



Figure 1.4 Some application examples of IoT Part 1 (generated by Gemini Nano Banana Pro)



Figure 1.5 Some application examples of IoT Part 2 (generated by Gemini Nano Banana Pro)

This thesis proposes a lightweight mutual authentication scheme. It combines Diffie-Hellman key exchange, distributed ledger technology, and Merkle Tree-based verification to ensure secure and efficient authentication in the intricate Internet of Things ecosystem. For easier implementation in real life, we based our scheme in the characteristics and functionalities of conventional IoT devices. This thesis uses the ESP32 platform to model the characteristics of WPAN, including low power consumption, short range, and restricted bandwidth. This approach aims to develop efficient, robust safety mechanisms that enhance the potential of a reliable and secure IoT environment. The paper explains the algorithms and technologies used to create this scheme.

This thesis aims to design and implement a secure, mutual authentication scheme for IoT devices that does not rely on a gateway during authentication process. Our focus is limited to enhancing security during the authentication process; the device is compatible with Wi-Fi and Bluetooth Low Energy. Our priority on authentication techniques isn't limited to a particular field, such as healthcare, industry, or agriculture. This scheme was not designed to tackle specific domain challenges; therefore, such domain-specific problems fall outside of our focus. Our research intends to investigate and address the question at hand: How can a lightweight mutual authentication scheme operate on IoT devices that have limited CPU, RAM, and flash storage, while ensuring low power consumption and minimal data storage requirements?

We introduce a lightweight mutual authentication scheme that integrates AES-256-GCM, ECDH key exchange, HMAC, HKDF, Merkle Trees, Bloom Filters, DLT, and session timeout. We deliver a formal security verification proof using the Scyther tool, and we demonstrate a prototype implementation on actual IoT hardware, proving the scheme's practicality.

CHAPTER TWO
BACKGROUND

Recently, various lightweight authentication techniques have been proposed to address important challenges, including low-power encryption and the prevention of security and authentication flaws. This section provides an overview of the essential technologies and encryption methods utilized in the proposed mutual authentication scheme. Consequently, recent research on IoT authentication is reviewed, and two comparison tables are produced to evaluate various methods based on their key technology and security attributes. This chapter concludes with a review of the core IoT technologies of these schemes.

2.1 Foundational Technologies

This section provides an overview of foundational technologies relevant to IoT and mutual authentication, including the Internet of Things, Wireless Personal Area Networks, Bluetooth Low Energy, Wi-Fi, Zigbee, Thread, ESP32, Secure Boot, Flash Encryption, Mesh Networks, Gateway Devices, Symmetric Key Encryption, AES, Public Key Encryption, Public Key Infrastructure, Elliptic Curve Cryptography, Elliptic Curve Diffie-Hellman Algorithm, Hash-based Message Authentication Code, HKDF, Pre-Shared Key, Blockchain, Distributed Ledger, Merkle Tree and Merkle Root, and Bloom Filters. The following subsections explain these technologies to establish the required background to understand the proposed mutual authentication solutions.

2.1.1 Internet of Things (IoT)

IoT refers to a massive network of physical devices called nodes, such as sensors, switches, cameras, household appliances, vehicles, industrial machines, and wearable devices that have wired and/or wireless communication capabilities. The Internet and other communication technologies connect these devices together, allowing

autonomous perception, data exchange, and mutual task implementation with minimal human participation. IoT platforms are used in many areas, such as smart homes, smart healthcare, and industrial automation. They make it possible to collect a lot of data and stay connected all the time (Öksüzer, 2022).

2.1.2 Wireless Personal Area Networks (WPANs)

WPANs are networks intended for communication between devices with restricted resources in close-range communication, which are customary in IoT networks. Energy efficient communication technologies such as Bluetooth Low Energy (BLE), Zigbee, and Thread enhance communication between low-powered devices in the IoT ecosystem (Şeker, 2024).

2.1.3 Bluetooth Low Energy (BLE)

Bluetooth version 4.0, which was introduced in 2009, presented BLE, a standardized wireless communication protocol. It was developed for devices that need to use less power and works in the 2.4 GHz frequency band over short distances. BLE works inside a WPAN framework and can send and receive data at speeds of 1 to 2 Mbps, with payloads of up to 251 bytes. Nodes save energy by staying in sleep mode until they need to connect.

BLE has two main structures: the Generic Access Profile, which is used to find and pair devices, and the Generic Attribute Profile, which is used for communication between connected devices. The security of pairing is decided by the devices' input and output functions. This can be done using BLE methods like Just Works, numeric comparison, out-of-band, and passkey. The ESP32 devices support both Wi-Fi and BLE communication (Şeker, 2024).

2.1.4 Wireless Fidelity (Wi-Fi)

Wi-Fi is a wireless communication technology that transmits data between devices using radio waves in the 2.4 GHz and 5 GHz frequencies. It is defined by IEEE standards such as 802.11a/b/g/n/ac/ax, which vary in frequency bands and data transfer rates. The most recent 802.11ax standard (Wi-Fi 6), for example, can handle data rates of up to 9.6 Gbps, enhancing its performance in densely populated environments. Wi-Fi security relies on protocols like WPA2 and WPA3, which utilize robust encryption algorithms, including AES. Wi-Fi is compatible with a lot of devices, such as smartphones, laptops, and tablets, making it appropriate for real-time IoT applications that necessitate immediate responses. However, its high power consumption makes it less suitable for battery-powered IoT devices in comparison to BLE and ZigBee (Şeker, 2024).

2.1.5 Zigbee

ZigBee is a wireless communication technology developed for the IoT ecosystem to transfer minimal data very efficiently. It is intended for edge IoT devices with low power consumption. ZigBee, founded on the IEEE 802.15.4 wireless standard, encompasses the physical, medium access control, network, and application layers. It is used for short-range communication, between 10 and 100 meters. ZigBee works within the 2.4 GHz frequency range at a data rate of 250 kbps.

ZigBee is extensively applied in IoT applications due to its minimal power consumption, compatibility with diverse network topologies such as star, mesh, and cluster, and low-latency message transmission capability. It is regularly employed in smart systems domains such as healthcare, infrastructure, and houses. ZigBee utilizes AES encryption for data safety. Unlike other technologies, ZigBee is not supported on smartphones, tablets, or computers and lacks IP addressing, requiring a gateway device to manage records and employ protocols such as MQTT. Its technological adoption is hindered by its higher costs in comparison to Wi-Fi and BLE (Rzepecki

and Ryba, 2019).

2.1.6 Thread

The Thread network protocol is a wireless mesh technology that rivals ZigBee, utilizing 6LoWPAN (IPv6 over Low-Power Wireless Personal Area Networks). Thread is especially useful for mid-range IoT applications, especially when devices are in proximity to network nodes, and it is explicitly engineered for smart home products. Thread's key feature is its IPv6 connectivity, which enables device communication, cloud service access, and direct user interaction via the Thread mobile applications. IoT devices such as the Seeed Studio XIAO NRF52840 support data communication via Thread and additional communication channels, including BLE and ZigBee. The Thread Group alliance was established in July 2014, rendering Thread a relatively recent protocol. As companies such as Google, ARM, Samsung, and Apple join the Thread Group, it is expected to become a standard for IoT development soon (Wen et al., 2024, Rzepecki and Ryba, 2019).

2.1.7 ESP32

The ESP32 is a low-cost, energy-efficient system-on-chip microprocessor created by Espressif Systems. It is widely used in IoT applications. The ESP32 is preferred as an edge device in IoT implementations due to its cost-effectiveness, energy efficiency, and compatibility with WPAN technologies like Wi-Fi and BLE. It is designed to deliver high performance and adaptability, incorporating integrated wireless connectivity and resilient hardware-based security features (Espressif Systems, 2025).

2.1.8 Secure Boot

Secure boot is an important security feature of ESP32 devices. It makes sure that only verified firmware can run on the device. The bootloader checks the

cryptographic signature of each piece of software before it runs when this feature is turned on. This makes a chain of trust that goes from the bootloader to the application code. The mechanism prevents unauthorized firmware changes, which can be malicious code, even if someone gets physical access. During setup, the public key for ESP32 chips is stored in eFuse memory. The chips use RSA-based digital signatures. You can't turn off secure boot once it's on without reflashing the device. This makes sure that only trusted code from the manufacturer can run on the hardware and protects it from firmware tampering (Abdellatif et al., 2023, El Hanine et al., 2025).

2.1.9 Flash Encryption

Flash encryption is another feature of ESP32 devices that keeps the data and firmware stored in the ESP32's external flash memory secure. When this feature is turned on, all data written to flash is automatically encrypted with AES and a unique key that is stored in the device's eFuse memory. Hackers can't recognize important data, like application codes, security keys, or user records, even if they try to get to the flash chip directly or intercept the communication between the processor and flash memory. ESP32's specialized hardware handles encryption and decryption so it doesn't affect performance. When used with secure boot, flash encryption makes sure that only authenticated code runs and that all stored code and data stay private (Abdellatif et al., 2023, El Hanine et al., 2025).

2.1.10 Mesh Network

A mesh network is a type of ad hoc network where nodes send data directly to their neighbors without needing a central access point or gateway. Mesh networks use nodes to send traffic instead of a central node, which solves the problems that come with having a central node. In more advanced mesh setups, nodes share information in layers that are arranged in a hierarchy. As more devices are added to the mesh network, packet delays may happen. The optimal network size depends on how it is designed and setup (Şeker, 2024).

2.1.11 Gateway Device

A gateway device is a part of a network that has more processing power, storage, and memory than resource-constrained edge devices. In machine-to-machine communication networks, gateways connect IoT devices and handle communication between networks that are both internal and external. Relay and Main are the two main types of gateways. An edge gateway, or Relay Gateway (RG), collects data from many IoT devices and lets them connect to bigger networks. RGs are located at the edge of the network and provide storage, processing, and caching services to speed up connections and help the main gateway with authentication. They often support BLE, Zigbee, and Wi-Fi mesh protocols, which let them communicate with other devices that use those protocols. The Main Gateway (MG) is the most important part of the network. It usually works as an MQTT broker. It connects the whole network to the Internet (Hasan et al., 2024).

2.1.12 Symmetric Key Encryption

Symmetric-key encryption is a cryptographic method to encrypt and decrypt data using the same secret key. As it is fast and doesn't require a lot of computing power, it is used commonly in resource-constrained IoT devices that don't have a lot of memory or processing power. The safety of symmetric-key encryption relies solely on the key's confidentiality; if it is breached, the encryption becomes useless. AES is a well-known encryption model because it is supported in many devices and works efficiently in situations where resources are limited. However, it is still very challenging to send secret keys safely. To solve this problem, symmetric and asymmetric cryptography are often used together to make hybrid security systems (Fei et al., 2024).

2.1.13 Advanced Encryption Standard Galois Counter Mode (AES-GCM)

AES-GCM is an authenticated encryption cryptographic primitive that is widely used in IoT architectures due to its efficacy and ability to protect data at a high level. Standard block cipher modes may only address confidentiality. However, AES-GCM is an authenticated encryption with an associated data scheme that provides confidentiality, integrity, and authenticity in one pass. The algorithm transforms live data into an encrypted byte stream, which stops both passive eavesdropping and active data tampering. To do this, a mutual secret key is used to create a unique ciphertext and an authentication tag. This is usually done through Elliptic Curve Diffie-Hellman (ECDH).

On the server side, the process is the opposite: the server extracts the unique device ID, uses the shared secret to decrypt the payload, and checks the data's integrity at the same time to make sure it wasn't changed while it was being sent. The Galois Counter Mode (GCM) lets the system check the integrity of messages, allowing it to detect and stop attempts to change or fabricate information by quickly identifying single-bit alterations. Furthermore, GCM performs authentication and encryption at the same time, enabling the simultaneous execution of both tasks for resource-constrained IoT devices, rather than requiring their individual implementation (Asif et al., 2025).

2.1.14 Public Key Encryption (PKE)

Public Key Encryption (PKE), also known as asymmetric-key cryptography, uses two keys: a public key to encrypt and a private key to decrypt. The only way to decrypt a message that has been encrypted with the public key is with the corresponding private key. The private key cannot be derived from the public key. Key pairs are created using key derivation functions that are unique to the encryption algorithm that was chosen. The public key can be shared with others on the network, but the private key must stay confidential and not be distributed.

Traditionally, PKE is used for key exchange and authentication. It allows digital

signatures and secure communication without the need for pre-distributed keys, which is distinctive from symmetric-key cryptography. However, public-key encryption is slower and needs more computing power than symmetric encryption. As a result, it is usually used during the key generation phase instead of to encrypt data that is already in a session (Şeker, 2024).

2.1.15 Public Key Infrastructure (PKI)

PKI is a scheme for creating, distributing, storing, and revoking digital certificates, and furthermore for encrypting data with public keys. PKI is a system of rules, roles, and procedures for handling public key certificates. Certificate Authorities (CAs) are in charge of issuing PKI certificates that link the user's public key to their identity digitally. Certificate-based authentication usually uses the X.509 certificate formatting. For resource-constrained IoT devices, PKI is very difficult to implement because of the costs of certificate validation, storage requirements, and the complexity of the validation process (Gupta and Varshney, 2023).

2.1.16 Elliptic Curve Cryptography (ECC)

Elliptic Curve Cryptography is a public-key encryption method that uses the algebraic properties of elliptic curves over finite domains. It is widely adopted among asymmetric encryption methods. ECC has strong encryption and much smaller key sizes than other algorithms. For example, a 256-bit ECC key offers similar security to a 3072-bit RSA key. This efficient operation uses less memory and processing power, which makes ECC a great choice for resource-limited IoT. The security of ECC is based on the difficulty of computing the Elliptic Curve Discrete Logarithm Problem. The general form of an elliptic curve's equation is $y^2 = x^3 + ax + b$ The Elliptic Curve Digital Signature Algorithm (ECDSA) is a cryptographic algorithm that lets you sign things digitally using public and private keys made with ECC. It is widely used in blockchain applications. Elliptic Curve Diffie-Hellman (ECDH) is a way for two parties to generate a shared secret key. Because of its low computing overhead,

ECDH is considered to be a lightweight solution for IoT devices with limited resources (Kumar et al., 2024).

2.1.17 Elliptic Curve Diffie-Hellman Algorithm (ECDH)

The ECDH algorithm is a key-exchange protocol way for two parties to share a secret key. ECDH is based on the ECC algorithm, which is based on the mathematically difficult Elliptic Curve Discrete Logarithm Problem. The protocol lets two devices that are communicating with each other set a shared starting point and use their own private keys to derive the same standard secret key independently. This is very important, especially protecting the secret key of symmetric encryption. Because it is based on ECC, ECDH costs less to process than other asymmetric key exchange algorithms. This feature makes ECDH a lightweight solution that works well with resource-limited IoT devices.

Two parties, A and B, randomly select an integer to be the private key SK_A. Each party derives its public key by multiplying its private key by the common Generator Point (G), for example, $PK_A = SK_A * G$. After that, the two sides send each other their public keys over a channel that isn't secure. Finally, each party multiplies the other's public key by its own private key to compute the same secret key ($SK = SK_A * PK_B = SK_B * PK_A$). The main point is that neither side has to share their private keys in order to derive the shared secret key. In literature reviews, ECDH is used to create a public key for ephemeral, secure data exchange between an IoT device and a gateway device (Öksüzer, 2022).

2.1.18 Hash-based Message Authentication Code (HMAC)

HMAC is a specific type of Message Authentication Code (MAC) that checks the authenticity and integrity of a message. HMAC is a mix of a secret cryptographic key and a cryptographic hash function, such as SHA-256. The receiver can use this code to verify that the message is from the right person and that an attacker hasn't altered it.

HMAC works by firstly having the sender use the message and the shared secret key to calculate the HMAC and then send both the message and the HMAC together. Then the receiver calculates the message's HMAC value and compares it to the value it received. It is established that the message is not corrupted if the values are the same (Mershad and Cheikhrouhou, 2023).

2.1.19 HMAC-based KDF (HKDF)

The HMAC-based Key Derivation Function (HKDF) is a cryptographic primitive that generates one or more cryptographic session keys from an initial keying material, like a shared secret key created through a Diffie-Hellman exchange. Its main purpose is to make sure that the keys that come out are unique, secure, and uniformly distributed, which makes them valuable to use in certain symmetric schemes like AES-GCM or HMAC.

The function usually works in two steps: first, extraction, and then expansion. Extraction phase takes entropy from a shared secret that may be a non-uniform source and turns it into a pseudorandom key with a fixed length. Expansion Phase takes the extracted key and turns it into several new keys of the required lengths, each of which is created for a different security purpose. For example, one key is for encryption, and another is for message authentication.

HKDF is better for real-world IoT uses because it offers high security with low overhead. For instance, testing on platforms with limited resources shows that an HKDF can run in about 2.9 ms, which makes it easy to set up a session key quickly without using up too much battery life. The generation process is computationally irreversible because it is based on HMAC, which uses one-way hash functions like SHA-256. This protects the system from session-key disclosure even if an attacker intercepts a public nonce (Hasan et al., 2024, Fei et al., 2024).

2.1.20 Pre-Shared Key (PSK)

Pre-Shared Key (PSK) is a way for every participant involved in a secure communication session to know the secret key before the connection is established. This mechanism is often used along with symmetric encryption and Transport Layer Security (TLS) to keep communications between endpoints safe. But using TLS with PSK makes computation overhead. In some architectures, a server generates device IDs that are then shared and used as pre-shared keys to encrypt the distribution of a secondary key. A pre-shared hashed secret value can be used in symmetric-based authentication and key agreement schemes to create a standard shared key. However, storing multiple pre-shared keys across different proxies and using them for each authentication session can make the system more vulnerable to side-channel attacks (Şeker, 2024).

2.1.21 Blockchain

Blockchain, introduced by Satoshi Nakamoto in 2008, is essentially a distributed database or ledger. It works as a decentralized system, which means that there is no need for a central authority or a separate third-party app to control and store data. There are blocks in the technology, and each block contains transactions in it. A cryptographic hash links each block to the previous one. This hash-chaining method makes sure that if old data is changed, the hash value changes too. This makes it easy to detect if the data has been changed and keeps the records tamper-proof and permanent. This feature of data being unchangeable and traceable is very helpful for the IoT ecosystem because it makes it safer and easier to inspect. Blockchain systems are classified based on data visibility (Public or Private) and access control (Permissioned or Permissionless) (Altaş, 2022).

2.1.22 Distributed Ledger

In a blockchain network, a distributed ledger is the shared storage system where all the data from all transactions is stored forever. All the nodes in the network possess a copy of this shared ledger so that the data is always uniform. This copying of histories prevents a single point of failure attacks. Once recorded in the ledger, the data records are tamper-proof. Blockchain technology is a system that uses a distributed ledger to verify and securely store transactions across all network nodes. The ledger keeps track of all the transactions that happen on the blockchain network. In some structures, like the Quorum blockchain, nodes can have more than one ledger (Arshad et al., 2023).

2.1.23 Merkle Tree and Merkle Root

A Merkle tree is a tree-based data structure and is used in blockchain systems to store and check large amounts of data, like data transactions or info from IoT sensors, as a tamper-proof method. It works by putting data into a tree-based structure of cryptographic hashes, which makes it easy to detect if any data changes. Leaves are the nodes at the bottom of the tree. In the context of the IoT, each leaf is a hash that includes several pieces of information, like the message, its digital signature, the corresponding public key, and a batch ID. The ledger's storage overheads are much lower because it hashes the data instead of keeping it in its original form.

The tree is built by pairing and hashing leaf nodes repeatedly until there is only one hash at the top, which is called the Merkle root. This root serves as a digital fingerprint for all the data in that block. Each block is connected to the previous one through these hashes. This means that if the data in a block changes, its hash will also change, which the network can easily detect. This allows a recipient to check that a certain message was part of a recorded batch without having to process the entire dataset. The user simply just compares the Merkle proof against the Merkle root that is publicly available on the blockchain (Öksüzer, 2022).

2.1.24 Bloom Filters

Bloom filters are a type of probabilistic data structure that are used in security schemes, especially in IoT and blockchain environments where resources are limited, due to their efficiency. They use a bit array and multiple hash functions to check if an element is in a set. When initialized, all the elements in the array are set to zero. To add an element, it goes through multiple hash functions, each of which gives an index position in the bit array. The bit at that corresponding position is then set to 1. The Bloom filter method has been proposed as a lightweight alternative to cryptographic operations that require a lot of processing power, like ECDSA.

In some decentralized systems designed for lightweight nodes, a random group of blockchain nodes called witnesses is chosen to validate and confirm transactions. Bloom filters are used to keep track of these witnesses' selection, confirmation status, and whitelist membership. This usage displays that the core functionality of Bloom filters is really good at giving resource-limited IoT devices a lightweight technique to validate blocks in a blockchain environment. This space-efficient, probabilistic data structure cuts down on cryptographic overhead, making it easier for nodes with limited resources to query and verify the integrity of blockchain ledgers (Mershad and Cheikhrouhou, 2023).

2.2 Related Works

Recent studies have proposed many lightweight authentication methods for IoT devices to address key IoT challenges, from low-power cryptographic design to robust protection against security and authentication threats. In this section, we review the recently published literature on IoT authentication in light of the technologies discussed earlier. Based on these reviews, we create two comparison tables that assess major schemes by their key features and their resilience to common attacks.

Gupta and Varshney proposed a lightweight, secure PUF-based authentication protocol to address two key IoT concerns: physical tampering and secure key storage.

Their design combines PUFs with lightweight hashing and XOR operations, removing the requirement to store long-term secrets on the device. This approach increases resistance to physical capture and cloning attacks. Formal verification with AVISPA tool indicates that the protocol can provide mutual authentication, anonymity, and forward secrecy. However, the protocol's practical deployment can be challenging, since the stability of PUF's challenge-response pairs can vary under changing environmental conditions, which often requires robust error-correction mechanisms and can increase computational overhead (Gupta and Varshney, 2023).

Satpathy et al. propose a mutual authentication protocol intended just for the three-tiered IoT-Fog-Cloud architecture. It uses key technologies such as ECC to create public and private key pairs and mutual secrets, SHA-512 hash function is used for integrity checks and key derivation, and nonces and timestamps are used to guarantee that the data is fresh and to prevent replay attacks. A unique feature is that the fog layer gives the local authority node most of the responsibility for authentication verification. This decreases the dependence on the cloud during active sessions, which lowers latency and resource use. The protocol aims to deliver mutual authentication and session key security, perfect forward secrecy, along with anonymity and untraceability. Some known limitations are the strong trust assumed in the local authority and the difficulty of managing keys across different nodes, especially as the network scales further (Satpathy et al., 2025).

Nkenyereye et al. present the Lightweight Authentication Protocol for Smart Grids, targeted at mitigating the significant limitations met by resource-constrained Smart Meters (SMs) within widespread Smart Grid networks. The main concept is to use ECC and hash functions to accomplish that SMs can safely authenticate with each other or a trusted server while keeping the overhead low. ECC operations, the SHA-256 secure hash function, and XOR operations are the key technologies used in this protocol. All of these are implemented to operate best with battery-operated devices. One major weakness is that the protocol relies solely on tamper-proof devices for the protocol's security. If they are compromised, the protocol is susceptible to multiple attacks (Nkenyereye et al., 2025).

Jain et al. suggest a secure and direct method for mutual authentication between Industrial IoT (IIoT) devices and a server, as well as generating session keys for future secure communication within the IIoT network. The protocol uses basic cryptographic operations like XOR and hash functions to keep limited-resource IIoT devices from processing overhead. The process begins with an IIoT device (A) sending a request to another IIoT device (B), which then sends it to a central server. The central server checks the devices, generates a session key (SK) using intermediate values from device IDs, secret values, and random numbers, and then securely delivers this key to devices A and B. The protocol also includes timestamp checks to prevent any replay attacks. Even though the protocol is robust, its reliance on XOR and hash functions might make it vulnerable to more advanced attacks, when compared against stronger protocols like ECC or AES. The security model depends on a central, trusted server to generate keys and register devices. This could be a single point of failure attack target (Jain et al., 2025).

Ali and Anwer's framework defend IoT-cloud systems from security vulnerabilities by combining multifactor user authentication, mutual authentication between IoT devices and the cloud, and a new hybrid encryption scheme for sensor data. Users sign up for the Cloud Service Provider (CSP) with their credentials and biometric information. The first step is to encrypt the sensor data using ECC to generate an intermediate ciphertext. This ciphertext is then processed further with Genetic Algorithm (GA) operations, two-point crossover, and mutation on the binary representation to generate the final ciphertext. ECC offers better efficiency with smaller key sizes, while GA is meant to increase complexity and advance the avalanche effect. Decryption operations reverse the GA operations and then use the ECC private key to get to the initial sensor data. The paper does not provide a formal proof that the ECC-GA combination is significantly more secure than a robustly implemented ECC alone, even though GA adds more complexity. This could lead to an unnecessary processing overhead without apparent security benefits (Ali and Anwer, 2025).

Gupta et al.'s SLMAS framework aims for secure data communication and joint

authentication within Telecare Medical Information Systems (TMIS), focused on the protection of sensitive patient data exchanged between patient (P), doctor (D), the cloud server (C) and healthcare center (H). P signs up with the H to get the credentials and ECC-based public system parameters. Then H uploads P's data to the C. Then D can access this data by interacting with C. The core of SLMAS is a multi-phase mutual authentication process that is meant for different TMIS interactions. ECC point multiplication, secure hash functions, and nonce exchange are all used in authentication processes. During an authentication attempt, P and H send each other messages that include their identities, new nonces, and values that they have calculated using their private keys and the other party's public information. They check and confirm each other's identity and freshness, and then they generate a shared session key for secure communication. ECC is intended to provide strong security with relatively small key sizes, which makes it perfect for resource-constrained TMIS devices (Gupta et al., 2025b).

Hu et al. presented BAMA, a new mutual authentication method that is based on blockchain and therefore anonymous. This method protects confidentiality and security in IoT systems by using smart contracts, ring signatures, and ECC to allow anonymous mutual authentication and a session key agreement. BAMA is formally verified using ProVerif; it is confirmed that it is secure from replay and man-in-the-middle attacks, as well as having forward secrecy and high user anonymity. Even though it has strong security guarantees, using blockchain and ECC operations adds extra computational and storage overhead, which could be an issue for resource-constrained IoT devices (Hu et al., 2024).

Asif et al. present an intelligent two-phase dual-authentication framework for the security of the Internet of Medical Things (IoMT), aimed to reduce processing overhead. This proposed framework has two parts: a registration phase that uses ECDH for key exchange and a live communication phase that uses AES-GCM for data encryption. The framework uses ECDH, AES-GCM, and MQTT to make secure and fast communication in IoT networks. Still, it is limited by constraints on energy efficiency and the growing capability of large networks. Future work on improving

energy efficiency and integration of blockchain technologies can be used to make systems more scalable (Asif et al., 2025)

Burgos and Pustišek present a framework to guarantee data authenticity in IoT deployments by utilizing edge-aggregating servers and Ethereum Layer 2 rollups to increase scalability and security. The proposed framework incorporates signature aggregation, Zero-Knowledge Succinct Non-Interactive Argument of Knowledge utilizing Nova and Risc0 proving systems, Merkle trees for data validation, and the Elliptic Curve Digital Signature Algorithm for digital signatures. The protocol moves the processing load of checking signatures and making proofs to servers, making it easier on resource-constrained IoT and cutting down on the cost of storing data on-chain through Layer 2 rollups. The framework effectively tackles scalability and cost concerns with using blockchain-based IoT authentication. Still, the security and reliability of edge-aggregating servers are critical for future implementations. Future works may study strategies to reduce reliance on trust assumptions in edge servers, potentially by a distributed server architecture or an improved server-side security measure (Burgos and Pustišek, 2024)

Almuhaideb and Al-Khulaifi proposed an efficient authentication and key agreement scheme for the Internet of Underwater Things, where water limits the ability to communicate. The scheme utilizes fuzzy extractors and ECC algorithms to process biometric data. These features guarantee that both parties can mutually authenticate with each other and that users' identities are kept secret, even when communicating over acoustic channels, which causes high latency. Both BAN logic and the AVISPA tool have been used to verify the scheme, which offers robust security assurances, including forward secrecy and mitigation against physical tampering. The scheme's discussed trade-off is the high cost of ECC point multiplication, which is still a big issue for underwater sensors with limited battery life and hard to replace (Almuhaideb and Al-Khulaifi, 2024).

Kumar et al. present a secure and efficient framework for Vehicular Digital Twin networks, specifically constructed to address important security and privacy issues,

such as unauthorized access and data integrity. The framework utilizes ECC for robust authentication between vehicles, their digital twins, and a central cloud server. The scheme's security claims are analyzed in the random-oracle model, formally verified using the BAN logic representation, and the Scyther formal tool to display its ability to mitigate against a wide range of attacks. The suggested framework offers robust security assurance with smaller key sizes with ECC. This reduces processing, computing, and communication overhead, making it suitable for vehicles with resource-limited hardware. The framework's security claims depend on the central cloud server being reliable and unattainable. This dependence can result in a single point of failure. Future improvements could include using blockchain technologies to improve data reliability, using adaptive security methods such as machine learning algorithms to detect threats in real-time (Kumar et al., 2024).

Qiu et al. proposed a lightweight mutual authentication scheme designed for the Internet of Medical Things (IoMT) that uses blockchain and NFT technologies innovatively. The scheme uses the Chinese Remainder Theorem and ECC for quick and easy managing of users' identity registration and authentication, while making sure that each user's identity is unique and unalterable. This scheme's security claims are formally verified with the AVISPA tool, proving the scheme provides user anonymity and forward secrecy while communication overhead is reduced. However, combining NFTs with blockchain technologies in IoMT settings, we need to oversee the ledger growth and its latency. The scheme's resistance to certain DoS attacks needs more exploration. Future research can enhance device security and user confidentiality and also improve blockchain contract performance by utilizing more efficient NFT minting standards, such as ERC-1155, that can reduce registration latency (Qiu et al., 2025).

Alghamdi suggested a lightweight mutual authentication framework for IIoT based on the ECC. The framework suggests a registration process facilitated by a gateway with key insulation to mitigate the key escrow attack and prevent any distribution problems that occur in earlier identity-based and certificateless cryptography schemes. The framework demonstrates that optimizing cryptographic components lowered

communication overhead 51.44% and computation overhead 91.88% in comparison to the compared schemes. The framework is formally verified by both AVISPA and ProVerif tools, proving mitigation against replay and impersonation attacks. Alghamdi proposes a quantum-resistant design, adding Quantum Key Distribution to improve security against AI advancements and future quantum threats (Alghamdi, 2025).

Şeker et al. presents MARAS, a mutual authentication role-based authorization scheme devised for the security of MQTT-based IoT applications. This scheme uses standard encryption methods like AES, ECDSA, HOTP, and hash chains along with dynamic access tokens and a server running on OAuth 2.0 to ensure message integrity and secrecy between edge nodes and a gateway. MARAS offers a robust mutual authentication and authorization scheme through a relay gateway design. Their test measurements show that moving hash chain generation to the gateway limits client-side processing, which keeps the total CONNECT message overhead to 127 bytes. As a result, this scheme avoids the high battery usage of asymmetric encryption and limits message latency to 163% of the MQTT defaults. The reliance on a gateway poses a risk of potential single point of failure. The scheme also might have problems in decentralized or multi-broker environments where access control lists need to be updated on the fly. This indicates that future studies may benefit from decentralized trust methodologies to enhance resilience (Şeker et al., 2023).

Tanveer et al. proposed LEAF-IIoT, a lightweight and efficient authentication framework designed for the IIoT. The framework utilizes AES-GCM to establish the secure channels between users, gateways, and sensors. Their test results show that the framework has a very fast execution time of 0.004 ms on a Raspberry Pi 3 as a gateway. The framework's security claims are validated by the AVISPA tool, proving that it provides mutual authentication, anonymity, and perfect forward secrecy. However, the framework's reliance on pre-shared keys and symmetric primitives presents challenges in distributing keys in larger deployments. To overcome this, combining with different techniques could provide a balance between scalability and the current remarkable low-latency performance (Tanveer et al., 2024).

Soni and Singh presented LAKA, a lightweight authentication and key agreement protocol purposely designed for WBAN wearables, which are mostly used in medical settings. The protocol ensures secure communication between sensors and hub nodes with a low processing overhead by using simple XOR operations, SHA-256 hashing, and bio-hashing techniques instead of ECC. Their test results show communication overhead reduced by 57% and storage needs reduced by 15% when compared to recent protocols. Implementing a strict timestamp validation mitigates against replay and impersonation attacks. The protocol's security relies on keeping the bio-hashing parameters secret, which can be improved with strong encryption methods. Additionally, the protocol assumes that devices' clocks are perfectly synchronized for timestamp verification; however, in distributed low-powered edge devices, communication latency and battery-related clock synchronization issues (Soni and Singh, 2022).

Syam et al. proposed a distributed group authentication protocol for IoT devices in smart home settings. The protocol employs Shamir's Secret Sharing for token generation and mutual authentication, safeguarding the group's security against compromised nodes. Their test results reveal that rotating group administration and using nonces efficiently lower computational overhead, with an average total execution time of 0.52 seconds. The protocol is successfully formally validated by the Scyther tool, having forward secrecy and proficiently administering group keys. However, their dependence on the secure channel assumption proposes potential implementation problems when initializing SSL and TLS connections requires a high processor load (Syam et al., 2023).

Ukpebor et al. developed a secure end-to-end communication scheme that utilizes the KATAN32 block cipher in resource-constrained IoT networks. This scheme uses buffer manipulation and bitwise operations to deliver lightweight data encryption. The scheme provides confidentiality, but it was primarily developed as an encryption mechanism rather than a complete authentication scheme, missing essential features such as mutual authentication, forward secrecy, and data anonymity. As a result, it is suited for being a part of a complex security scheme (Ukpebor et al., 2023).

Naser et al. proposed a lightweight authentication framework for IoMT, employing IoT-optimized symmetric encryption techniques like TinyAES and SPECK. The paper shows that selecting the appropriate primitives can reduce overheads, like lowering encryption latency to 0.07 ms. The suggested framework provides mutual authentication and end-user privacy, but it utilizes symmetric encryption methods without forward secrecy protection. To overcome this, temporary key-exchange methods, like ECDH key exchange, can be added to (Naser et al., 2025).

Lo et al. proposed a lightweight mutual authentication protocol for low-powered IoT devices that are vulnerable in high-risk situations, like FinTech, where their limited battery and restricted processing make it unproductive to use secure cryptographic schemes like RSA and ECC. The protocol utilizes XOR operations and one-way hash functions to cut down on energy use by avoiding modular and iterative computations. The protocol effectively transfers the processing load away from the limited edge device to the authentication server by adding an additional registration phase. This phase takes only 0.00032 ms to compute, which is much lower than the 17 ms ECC-based techniques. The protocol's security is based on the idea that the initialization process happens in a safe place where device IDs and secret numbers are given out, even though these performance improvements have been made. The protocol's security is verified against network-based attacks like eavesdropping and replay attacks by utilizing timestamps. However, protocol relies on pre-shared secrets; improving physical security of IoT devices is critical (Lo et al., 2025).

Yu et al. developed a dynamically obscured hybrid PUF-based mutual authentication scheme designed for fog-enabled IoT infrastructures. The scheme uses PUFs and recursive hashing to make challenge-response pairs obscure, thus blocking adversaries from reverse engineering PUF's core. PUF is also used to mitigate against the increasing threat of AI-based modeling attacks. The scheme's security claims are verified formally and informally; by using ROR oracle model analysis and the AVISPA tool, both validate the scheme's mutual authentication, anonymity, and forward secrecy. While prior PUF-based schemes prioritize reducing cryptographic overhead and mitigating security attacks, this hybrid method focuses directly on the

vulnerability of AI-driven PUF model analysis and building. The scheme's security assurance is based on the complete isolation of the weak PUF hardware, safe from physical extraction (Yu et al., 2025).

Mo et al. suggested a two-factor authentication model for intellectual property transactions that uses an improve zero-knowledge proof architecture like the PLONK proof system that merges biometric data with property features. The model uses the PLONK proof system and Poseidon hashing that keeps users' data private and authenticates them without revealing any private data, while reducing proof generation time to 0.42346 seconds, efficient compared to similar cryptographic models. The model offers security against replay and identity-spoofing attacks. However, their dependence on the only simulated local hardware makes the system's high throughput efficiency unproven in real-world conditions (Mo et al., 2025).

There are many different security models and cryptographic approaches designed for IoT security. The schemes in Table 2.1 are grouped by the application domain and network architecture. This grouping is essential for understanding the operational limitations and framework dependencies of each scheme. The table also shows the types of communication and the main technologies used, like smart contracts and PUFs. These features were selected to demonstrate how non-cryptographic technologies can work together to make secure and efficient schemes when combined with cryptographic technologies. Classifying the communication channel is also important for figuring out how vulnerable the proposed system is to security attacks. Lastly, the table helps us to compare the processing load of each scheme by listing primitives such as AES, ECDH, and HMAC. This difference of processing overhead is important for figuring out if a proposed scheme is intended to work on resource-constrained devices or with powerful gateway devices.

Table 2.1 Security features and technologies

Schemes	Domain	Arch	Comm	Core Tech	Crypto	MA	PSK	ECC	DH/ECDH	HKDF	HMAC	MT	BF	TO	DL
Our Scheme (Kaygısız and Dalkılıç, 2025)	IoT	Decent.	BLE/Wi-Fi	Merkle Tree, DL	AES,HMAC,ECDH	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
MARAS (Şeker et al., 2023)	IoT	Cent.	MQTT	Tokens	AES, ECDSA	Y	N	Y	N	N	Y	N	N	Y	N
BAMA (Hu et al., 2024)	IoT	Decent.	Wireless	Smart Cont.	ECC, Hash	Y	N	Y	Y	Y	N	Y	N	Y	Y
(Qiu et al., 2025)	IoMT	Decent.	Wireless	NFT, CRT	ECC, Hash	Y	N	Y	Y	N	N	N	N	N	Y
LEAF (Tanveer et al., 2024)	IIoT	Hier.	Wireless	Auth. Enc.	AES-GCM	Y	Y	N	N	Y	N	N	N	Y	N
LAKA (Soni and Singh, 2022)	WBAN	Hier.	Wireless	Bio-hash	SHA, XOR	Y	Y	N	N	N	N	N	N	Y	N
(Syam et al., 2023)	Home	Group	Wireless	SSS	Hash, XOR	Y	N	N	N	N	N	N	N	Y	N
(Ukpebor et al., 2023)	IoT	P2P	WiFi	Buffer Ops	KATAN32	N	Y	N	N	N	N	N	N	N	N
(Naser et al., 2025)	IoMT	Edge	Wireless	Optim.	TinyAES	Y	Y	N	N	N	N	N	N	N	N
(Gupta and Varshney, 2023)	IoT	Dev-Srv	Wireless	PUF	Hash, XOR	Y	N	N	N	N	N	N	N	N	N
(Almuhaideb and Al-Khulaifi, 2024)	IoUT	Ad-hoc	Acoustic	Bio-met	ECC, Hash	Y	N	Y	N	Y	N	N	N	Y	N
(Yu et al., 2025)	Fog	Fog/Edge	Wireless	PUF	Hash, XOR	Y	N	N	N	N	N	N	N	Y	N
(Mo et al., 2025)	IP	Chain	Net	ZK Proofs	Poseidon	Y	N	Y	N	N	N	N	N	N	Y
Legend: Arch: Architecture, MA: Mutual Auth, MT: Merkle Tree, BF: Bloom Filter, TO: Session Timeout, DL: Distributed Ledger. Y: Yes, N: No.															

Since IoT devices are often left unattended and out in the open, the security of IoT schemes must be tested against strict threat models. Table Table 2.2 gives a detailed evaluation of how each scheme resists common malicious threats. This table was made to show a clear yes or no to the question of whether a scheme mitigates against major security attacks like replay, man-in-the-middle, and impersonation. The table has columns for physical tampering and side-channel as well. These attack resistance features are very important for IoT devices. The inclusion of anonymity and end-user confidentiality metrics tackles the increasing demand for privacy-protecting authentication in industries like healthcare and smart grids, where regulations mandate user identity protection. It also reflects the strength of each scheme's security claims by indicating whether a formal authentication methods were applied or not; use of BAN logic proofs and formal verification tools such as Scyther, AVISPA, and ProVerif. This distinction allows researchers to understand the level of assurance provided by each scheme.

This thesis focuses on designing a peer-to-peer mutual authentication scheme for IoT devices that does not depend on a gateway during authentication, an approach that has been rarely explored in recent studies. Our proposed scheme is lightweight, widely applicable, and resilient against common security threats.

Table 2.2 Attack resilience analysis

Schemes	Replay	MitM	Spoofing	Side-Chn.	DoS	Escrow	Eaves.	Phys.	Anon.	F/BS	BAN Logic	Formal Tool
Our Scheme (Kaygısız and Dalkılıç, 2025)	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Scyther
MARAS (Şeker et al., 2023)	Y	Y	Y	N/A	Y	N	Y	N	Y	N	N	Scyther
BAMA (Hu et al., 2024)	Y	Y	Y	N/A	Y	N	Y	N	Y	Y	Y	ProVerif
(Qiu et al., 2025)	Y	Y	Y	N/A	N/A	N	Y	N	Y	Y	N	AVISPA
LEAF-IIoT (Tanveer et al., 2024)	Y	Y	Y	N/A	Y	N	Y	N	Y	Y	Y	AVISPA
LAKA (Soni and Singh, 2022)	Y	Y	Y	N/A	Y	N	Y	Y	Y	Y	Y	AVISPA
(Syam et al., 2023)	Y	Y	Y	N/A	Y	N	Y	N	N	Y	N	Scyther
(Ukpebor et al., 2023)	N/A	N/A	N/A	N/A	N/A	N/A	Y	N	N	N	N	None
(Naser et al., 2025)	Y	Y	Y	N/A	N/A	N	Y	N	Y	N	N	None
(Gupta and Varshney, 2023)	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	AVISPA
(Almuhaideb and Al-Khulaifi, 2024)	Y	Y	Y	N/A	Y	N	Y	Y	Y	Y	Y	AVISPA
(Yu et al., 2025)	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	ROR/AVISPA
(Mo et al., 2025)	Y	N/A	Y	N/A	N/A	N	Y	N	Y	N	N	None
Legend: Y: Protected/Yes, N: No/Vulnerable, N/A: Not Applicable. F/BS: Forward/Backward Secrecy.												

CHAPTER THREE
SYSTEM DESIGN AND METHODOLOGY

Our scheme provides robust and lightweight mutual authentication and resilient session key establishment for resource-constrained IoT devices. We assessed similar literature research, pointing out their key advancements while indicating their limits. Our scheme was initially published as a conference paper. It uses lightweight cryptographic tools such as HMAC, ECDH key exchange, Bloom filters, and Merkle trees. The scheme can switch channels, initializing with BLE for the beginning and switching to the faster option, Wi-Fi, when more speed is needed and the data transferred size increases, improving security and performance further. A session timeout limit is imposed during the authentication process to mitigate against possible resource-exhaustion attacks.

3.1 Scheme Assumptions and Device Initial States

First, we talk about the key ideas, boundaries, and expectations that provision the scheme's security and performance. For example, the minimum hardware requirements of an IoT device, communication ability using both BLE and Wi-Fi, and the dependence on distributed ledgers instead of a gateway device. These boundaries should be made clear at the start to help understand our design choices for the scheme and why the conventional PKI-based solutions aren't ideal for resource-constrained IoT devices.

3.1.1 Pre-Shared Key

Each device maintains a common pre-shared key. During the device's setup process, this key is securely provided by an authenticating gateway server and stored safely in the device's protected memory. During the authentication session, PSK is only used to encrypt messages until a shared session key is established.

3.1.2 Ledger Model

Each device maintains a local ledger of trusted device IDs, which is provided by the authenticating server during the device's setup process. The ledger is managed by Merkle trees and verified using Bloom filters.

3.1.3 No Online Authentication Server

The scheme is devised for a peer-to-peer authentication without relying on an online gateway server during the session's runtime. An authentication gateway server is still involved, but only during the device's setup process.

3.1.4 Device Capabilities

Each IoT device is expected to have basic cryptographic capabilities on its firmware and hardware, including hashing, HMAC computation, and ECC-based asymmetric functions, and to be able to store a local ledger with a size of 100 entries.

3.1.5 Communication Channels

Devices should be able to communicate over both BLE and Wi-Fi. In our scheme, BLE is used for the initial phases due to its low energy consumption and the simplicity of making a connection. When larger data messages are transferred, the devices switch to Wi-Fi, a higher-throughput channel, to reduce latency, improve connectivity issues that might occur, add additional security, and shorten the total time of the authentication process.

3.1.6 Nonce Generation

ESP32 devices employ their built-in hardware random number generator to produce nonces, ensuring unpredictable entropy. In our setup, each device retrieves fresh random bytes from the ESP32 RNG at the initiation of every authentication session, generates a nonce of fixed length, and uses it only for that session. This nonce is straightaway integrated into the handshake, ensuring that each message and the resulting session keys are unique for each session, even if the same two devices are involved in communication every time. To avoid unintentional reusage after resets or interruptions, the device always generates a fresh nonce for every session and never stores it as a long-term value; any intermediate secrets dependent on the nonce are cleared from RAM once the session key is established.

3.1.7 Timeout and Timekeeping Assumption

ESP32-WROOM-32 model we implemented our scheme on has on-chip hardware timers and the SoC real-time clock subsystem that measures time and enforces timeouts. For normal active operation, the firmware uses the general-purpose timer groups and the system tick, which comes from these timers, to make a counter that always goes up and has a resolution of microseconds to milliseconds. This counter can be used for timestamp features and computing elapsed time for different operations, such as imposing session timeouts. ESP32 also has a separate real-time clock subsystem that keeps track of time even when the main CPU is paused. This allows the device to keep track of time even when it is in low-power modes or sleeping. Other solutions for timekeeping also include connecting to the Network Time Protocol server, using GPS modules to get clock info from satellites, or using external RTC chips to keep track of time. Timeouts have many security advantages. For instance, they can prevent slow-motion attacks where an attacker takes too long to send a message, which can hold resources indefinitely by incomplete sessions.

3.2 Design Goals

This section clarifies the security and efficiency goals that led to the scheme's construction. The main goals are mutual authentication, establishing a robust shared session key, protection against known IoT vulnerabilities, and overcoming the restrictions on computation, storage, and energy use of IoT devices.

3.2.1 Mutual Authentication

Mutual authentication is defined basically as two edge nodes or an edge node, and the gateway server must mutually prove their identities to each other. Using methods like ECC and HKDF, our scheme ensures neither device is an attacker trying to capture private data. This creates a root of trust in the network, which keeps unauthorized attackers from accessing critical systems. Mutual authentication is key to stopping MitM attacks, as it guarantees that the data is sent only to the intended recipient. (Mansour, 2021).

3.2.2 Secure Session Key Establishment

The scheme must create a unique, temporary shared session key that only the authenticated devices will use. This session key is used to keep the remaining conversations secure and private. Typically, ECDH key exchange method is used to generate a shared secret over an insecure channel without sending any private key. The session key stays unpredictable and valid only for a short period of time; hence, each session uses new nonces and ECC key pairs to establish the shared session key. This setup prevents attackers from reading messages, even if they capture the messages. A strong session key is the core of keeping data secure from attackers (Babu et al., 2022).

3.2.3 Forward Secrecy

Forward secrecy makes it impossible to use any previous and currently captured communication data to decrypt and analyze future sessions. If long-term credentials are captured, the keys created for past sessions are still secure. Forward secrecy is ensured by creating temporary session keys with temporary values that will be erased from the device's memory after each session. Therefore, even if an attacker captures a device, it won't be able to find out previous session keys to decrypt messages. This feature is very significant for the IoT devices that could be captured. (Lo et al., 2025).

3.2.4 Using Distributed Ledger Technology

ECDH, HKDF, and HMAC provide integrity, privacy, and session key freshness, but they can't deliver a shared, tamper-proof record of trusted identities for decentralized IoT deployments. Therefore, our scheme uses the DLT technology to maintain and store device IDs that is distributed by all of the nodes in the system. This removes reliance on an online gateway authority for authentication. DLT lowers the risk of a single point of failure during an authentication session. In our scheme, Bloom filters and Merkle tree technologies efficiently reconcile and verify the ledger entries. These two technologies allow devices to check membership existence with minimal proof and latency and detect inconsistencies without revealing the entire ledger. The addition of DLT doesn't change the underlying cryptographic protections for authentication processes; instead, it allows for robust auditability, ledger consistency, and detection of any alteration.

3.2.5 Lightweight Operations

Our scheme is designed to lower processing, bandwidth, and energy load on resource-constrained IoT devices. Instead of using processing-heavy operations like modular exponentiation or bilinear pairings, we utilize efficient methods such as one-way hash functions, XOR, and other lightweight encryption techniques.

Lightweight operations reduce the IoT device's overall load and preserve battery life. By making messages smaller and simplifying the authentication steps, the scheme also decreases network traffic and device storage requirements. All of these combined techniques keep security resilient without causing latency and processor overload (Fei et al., 2024).

3.2.6 Comprehensive Security

The scheme must be able to prevent and resist different kinds of security attacks, such as replay, MitM, DoS, and impersonation. It incorporates creating new nonces for each session to prevent replay attacks, in which the attacker can't resend captured messages. Data integrity validations and a successful mutual authentication help mitigate MitM and impersonation by ensuring that attackers can't intercept and change any message and device data without being detected (Alghamdi, 2025).

3.3 Scheme Overview

Before the scheme is initiated, an authenticating gateway server provides PSK and a ledger containing pre-trusted devices during the device's setup process. One of the devices initiates the handshake and starts the authentication session using BLE communication. After the BLE connection is established, both devices generate an ECC public-private key pair and a fresh nonce. Then, they send each other their public keys, nonces, and HMAC output of the pair, encrypted with PSK. The receiving device checks the HMAC output it received and compares it with calculating its own HMAC output with the received pair. Both devices carry out this process. The first step ensures that both devices possess the same PSK, the same HMAC function, and the HMAC verification ensures that the handshake messages are valid for both devices, which mitigates impersonation and MitM attacks at the beginning of the session. If both devices pass the HMAC verification, they derive a temporary session key by using these shared public keys and nonces with ECDH key exchange and HKDF method.

Authentication is based on a distributed trust model in which devices search opposing devices' hashed IDs in their local ledgers, which store previously verified device information. As previously mentioned, authenticating gateway servers provides these ledgers containing pre-trusted devices during the devices' setup processes. Then, devices use Bloom filters to search their ledgers for the opposing devices' hashed IDs. If one or both devices don't possess each other's hashed ID in their ledgers, they compare their Merkle tree roots. If they don't match, the scheme switches to a higher-throughput Wi-Fi communication for the iterative Merkle tree leaf proof reconciliation step with a trust threshold, phase timeout check, and message quota restrictions.

If the 60% trust threshold is achieved within the timeout and quota limits, the authentication process is affirmative, and the reconciliation phase is finished. They create a new shared session key to use for subsequent communications. If the reconciliation phase fails from any condition, devices terminate communication, and both parties' information is added to opposing blacklists, which will be distributed among the system. This multi-phase method, which includes PSK, ECDH, HMAC, HKDF, DLT, Bloom filters and Merkle trees, and condition-based communication channel switching, aims to protect against security attacks such as replay, MitM, eavesdropping, and DoS. All while keeping computational, communication, and storage overhead minimal (Kaygısız and Dalkılıç, 2025) The notations are provided in Table 3.1.

Table 3.1 Summary of notations

Notation	**Description**
A, B	Device A and Device B
$(PK_A^{\text{temp}}, SK_A^{\text{temp}})$	A's temporary ECC public-private key pair
$(PK_B^{\text{temp}}, SK_B^{\text{temp}})$	B's temporary ECC public-private key pair
n_A, n_B	Random nonces generated by A and B
PSK	Pre-Shared Key shared by A and B
HMAC_A	HMAC calculated by A over its initial message to B
$\text{ECDH}(SK_A^{\text{temp}}, PK_B^{\text{temp}})$	Elliptic Curve Diffie-Hellman algorithm
K_{DH}	Shared secret derived using ECDH with ECC public keys and nonces
$\text{HKDF}(n_A \parallel n_B \parallel K_{DH})$	HMAC-based Key Derivation Function
K_{temp}	Temporary shared session key
K_{new}	New shared session key
ID_A, ID_B	Unique identifiers for A and B
Hash	Cryptographic hash algorithm
$Root_A, Root_B$	Merkle Tree roots of A and B
$\text{Hash}(\text{Proof}(LR))$	Merkle tree leaf proof of ledger entry or a leaf LR
$CurrentTime$	Elapsed time in the session
T_{start}	Starting time of the session
T_{max}	Session timeout limit

3.3.1 Initial State

Before the authentication process, each device is assumed to be provisioned in a trusted setup phase shown in Figure 3.1. Each device stores the pre-shared key (PSK) and the initial local ledger received from the trusted authenticating gateway server. These values are treated as long-term information and are stored in protected memory. Session-specific values such as nonces and ephemeral session keys are generated fresh for each authentication run and are removed from memory once the session key is established or terminated.

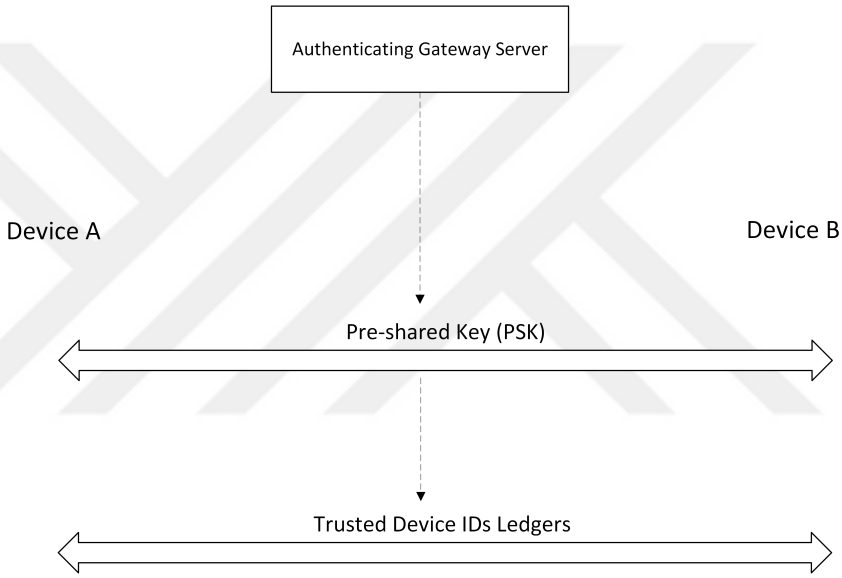


Figure 3.1 Initial state of devices before auth session

3.4 Phase 1: Initial Handshake and Temporary Session Key Generation

Phase Overview: This phase establishes a secure foundation for the entire scheme. Each device is preloaded with pre-shared key (PSK) and a local ledger, provided by the authenticating gateway server during the device's setup process. Devices connect over BLE, generate a temporary session key, and mutually verify that both devices run the exact scheme. Messages are encrypted using PSK until temp session key K_{temp} is derived. K_{temp} encrypts all subsequent communication until the new session key K_{new} is established.

This phase is achieved by the following steps:

3.4.1 Step 1.1: BLE Connection Establishment & Start Timer

In Figure 3.2, $Device\ A$ initiates a BLE server-client connection with $Device\ B$. When the BLE connection is established, both devices start measuring the session time, recording the current start time as T_{start}. This mutual timekeeping is critical for enforcing timeout checks throughout the session at the end of each phase, preventing any replay attacks.

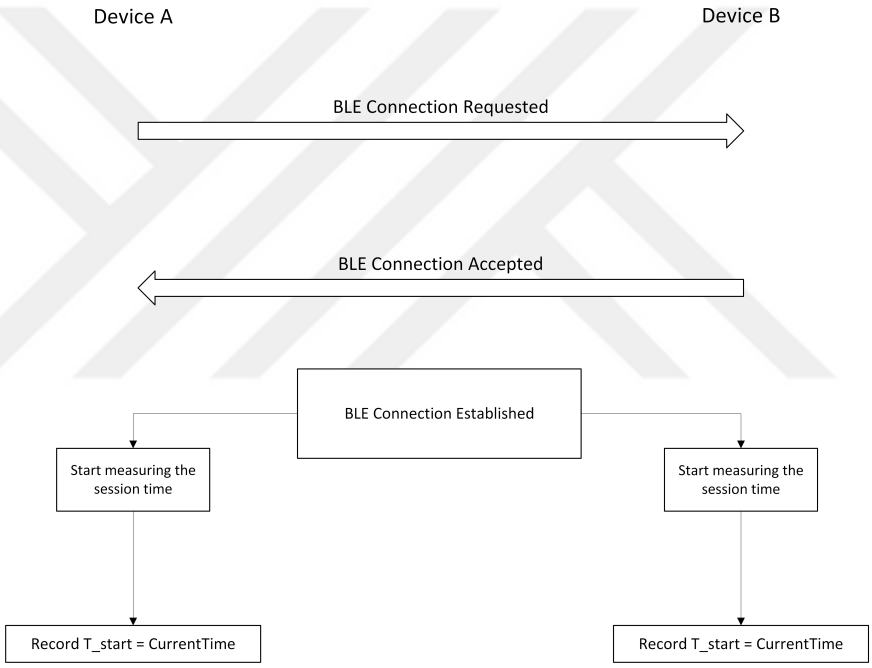


Figure 3.2 BLE handshake and timer starts

3.4.2 Step 1.2: Temporary Key, Nonce and HMAC Generation

As depicted in Figure 3.3, each device generates a temporary ECC public-private key pair independently. The public key (PK^{temp}) will be shared, while the secret key (SK^{temp}) remains private. These are temporary and used only in this session. Each device also generates a random nonce. Nonces ensure that each session is unique and protected against replay attacks Using the pre-shared key PSK, each device computes

$HMAC_A$ and $HMAC_B$ using its PSK, newly generated ECC public key, and its nonce, $HMAC(PSK, PK_A^{temp}, n_A)$ and $HMAC(PSK, PK_B^{temp}, n_B)$. These $HMAC_A$ and $HMAC_B$ will show to the other device that the sender has the valid PSK, without revealing it.

A generates its ECC public-private key pair $(PK_A^{temp}, SK_A^{temp})$ and fresh nonce n_A. A calculates $HMAC_A = HMAC(PSK, PK_A^{temp}, n_A)$. B generates its ECC public-private key pair $(PK_B^{temp}, SK_B^{temp})$ and random nonce n_B. B calculates $HMAC_B = HMAC(PSK, PK_B^{temp}, n_B)$

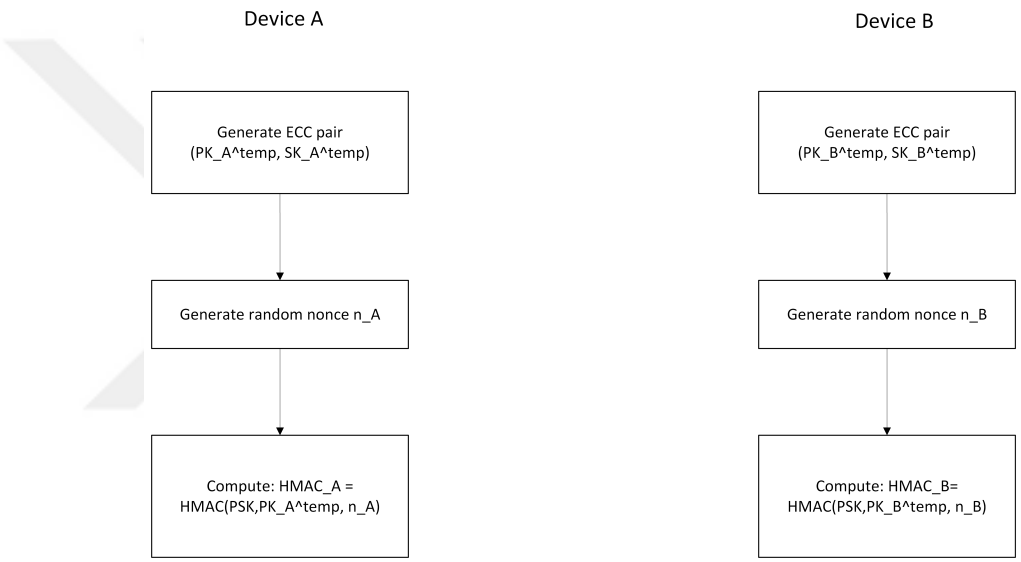


Figure 3.3 Temporary key, nonce and HMAC generation

3.4.3 Step 1.3: Initial Key Exchange and HMAC Verification

As shown in Figure 3.4, $Device\ A$ sends its temporary public key, nonce, and $HMAC_A$ to $Device\ B$. This message is encrypted with PSK. Upon receiving the message, $Device\ B$ uses PSK to decrypt it and then derives $HMAC'_A$ with its own $HMAC$ function. $Device\ B$ then compares $HMAC'_A$ with the received $HMAC_A$. If they are identical, $Device\ B$ can be confident that $Device\ A$ possesses the correct PSK and valid $HMAC$ function and that the message was not tampered with during transmission. If verification fails, $Device\ B$ immediately terminates the connection.

Also adding $Device\ A$ to its blacklist, marking it as a malicious device. The blacklists are regularly shared with gateway devices and across edge nodes. This prevents additional communication with potentially malicious devices at the handshake phase.

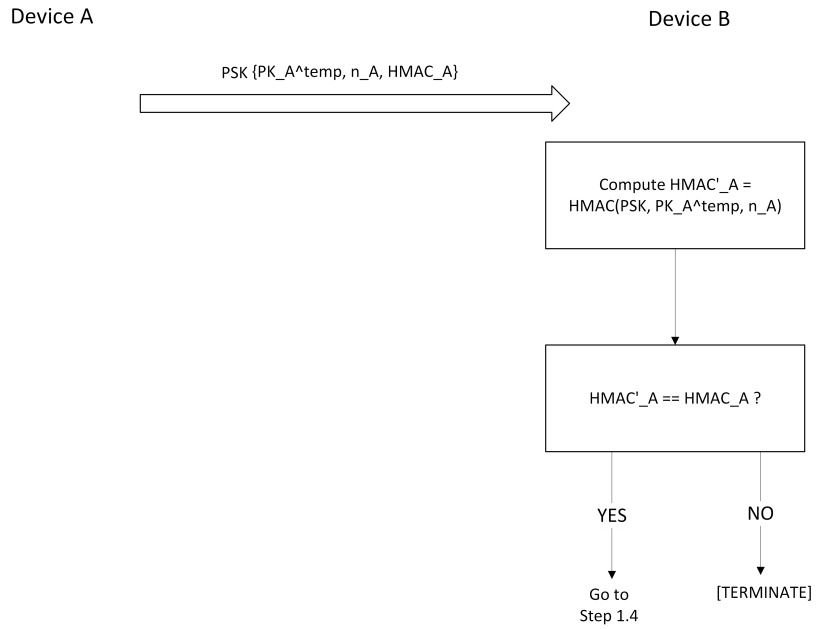


Figure 3.4 Initial device key exchange and HMAC verification

3.4.4 Step 1.4: Response Key Exchange and HMAC Verification

This step, which is shown in Figure 3.5, is similar to the previous step, only the roles are reversed. $Device\ B$ sends $Device\ A$ its temporary public key, nonce, and $HMAC_B$. $Device\ A$ uses PSK to decrypt the message and derives $HMAC'_B$ with its own $HMAC$ function. $Device\ A$ then compares the $HMAC'_B$ with the received $HMAC_B$. This two-way verification is necessary for both parties to prove who they are. Demonstrating that both devices share the identical PSK and $HMAC$ functions.

If verification fails, following the same procedure as in the previous step, $Device\ A$ immediately terminates the connection. Also adding $Device\ B$ to its blacklist, marking it as a malicious device. The blacklists are regularly shared with gateway devices and across edge nodes. This prevents additional communication with potentially malicious devices at the handshake phase. If verification is successful by both devices, they move

on to the temporary session key derivation step.

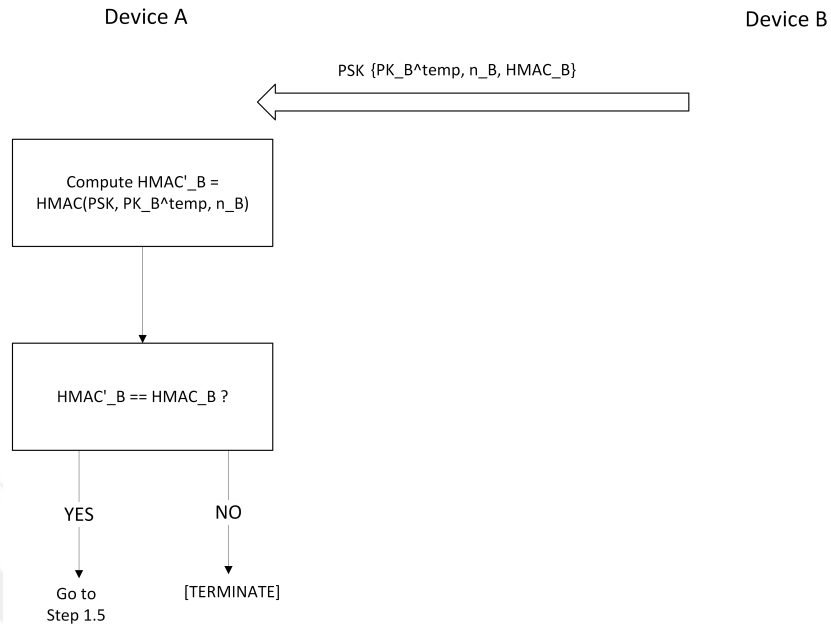


Figure 3.5 Responding device key exchange and HMAC verification

3.4.5 Step 1.5: Temporary Session Key Derivation

Both devices now possess each other's temporary public keys. In Figure 3.6, using the ECDH key exchange function, each device combines its secret key with the other's public key to create a mutual secret K_{DH}, calculated with $K_{DH} = ECDH(SK_A^{temp}, PK_B^{temp}) = ECDH(SK_B^{temp}, PK_A^{temp})$.

Due to the mathematical attributes of ECDH, both devices derive the same shared secret without ever transmitting it. The shared secret alone is not used directly as a key; hence, both devices use an HMAC-based Key Derivation Function ($HKDF$) to create the temporary shared session key K_{temp}, which is calculated using $K_{temp} = HKDF(n_A||n_B||K_DH)$.

$HKDF$ incorporates both nonces (n_A and n_B) to guarantee that the generated session key is unique to this session. From this step forward, all messages will be encrypted with K_{temp}.

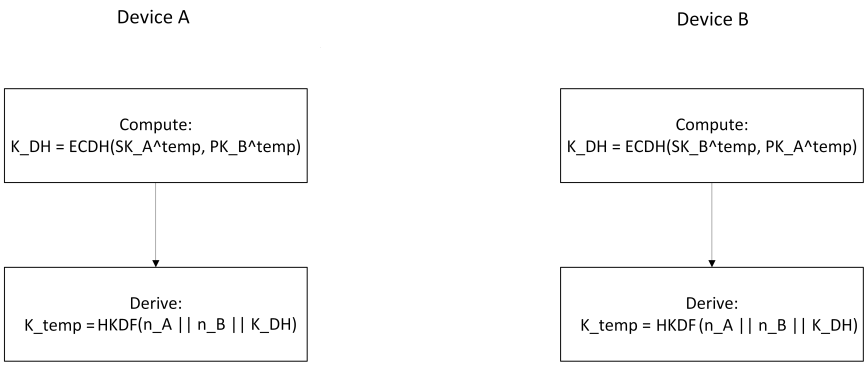


Figure 3.6 Temporary shared session key derivation

3.4.6 Step 1.6: Timeout Check

Before moving on to the next phase, both devices check if the time passed since T_{start} is longer than the maximum allowed time $T_{max} = 10\ Minutes$, which is $(CurrentTime - T_{start} > T_{max})$, as shown in Figure 3.7. At the end of each phase, this timeout mechanism kicks in. If the timeout limit is reached, the connection is immediately terminated, and all session data is erased. If not reached, both devices move on to Phase 2.

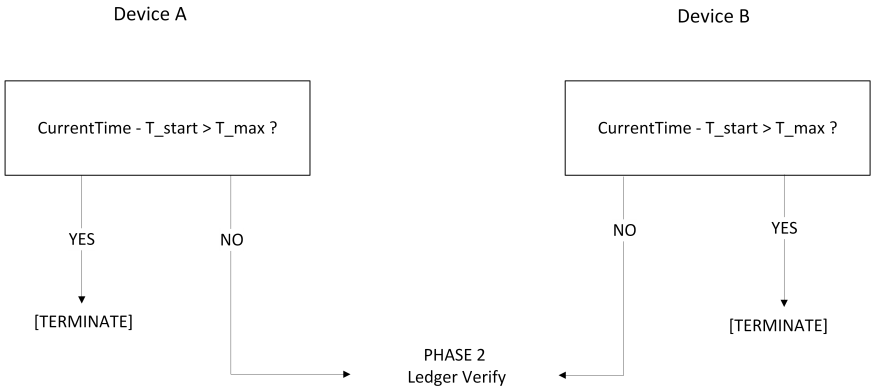


Figure 3.7 Timeout check

3.5 Phase 2: Initial Ledger Verification

Phase Overview: In this phase, the devices quickly check to see if they identify each other from previous communications. Each device maintains a local ledger of trusted device IDs. Devices exchange their own hashed IDs and use Bloom filters to quickly search them in their respective ledgers. If both devices can locate each other in their ledgers, the authentication process is successful, and they move to the final phase. We carry out this phase in the following order:

3.5.1 Step 2.1: Hashed ID Exchange

As seen in Figure 3.8, each device computes the hash of its unique identifier. Instead of sending plaintext IDs, hashing is performed to provide identity secrecy. If an attacker intercepts this message, they can't reach the plain ID, only the hash of it. Devices exchange hashed IDs and the other device's nonce. Adding the other device's nonce proves freshness and prevents replay attacks.

A sends $Hash(ID_A), n_B$ encrypted with K_{temp}. B sends $Hash(ID_B), n_A$ encrypted with K_{temp}.

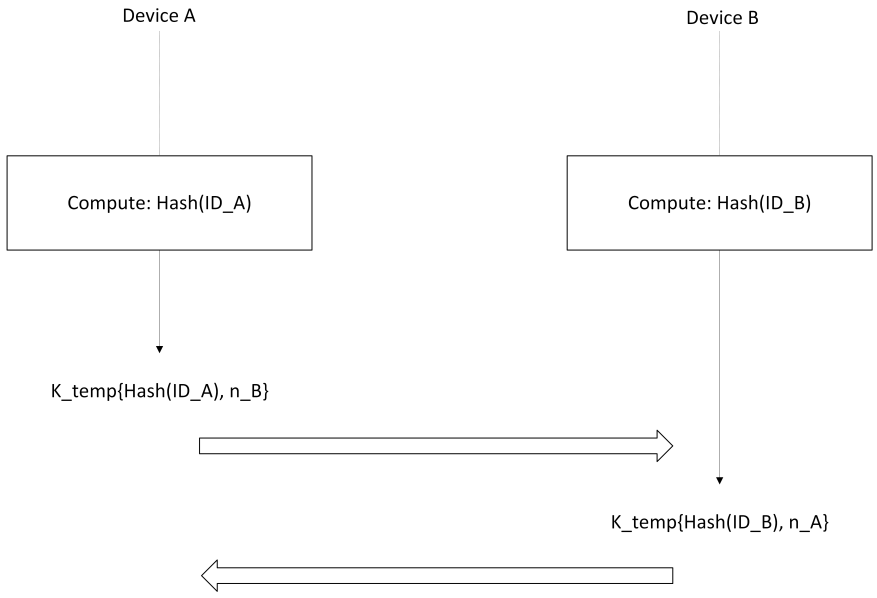


Figure 3.8 Hashed ID exchange

3.5.2 Step 2.2: Ledger Lookup

As shown in Figure 3.9, each device checks whether the other device's hashed ID exists in its local ledger using Bloom filter lookup. Bloom filters can rapidly clarify whether an element is definitely not in the list or can be in the list. Sometimes, they may produce false positives, meaning an element is reported as existing when it isn't, but they never produce false negatives. If it says not found, it's definitely not in the list.

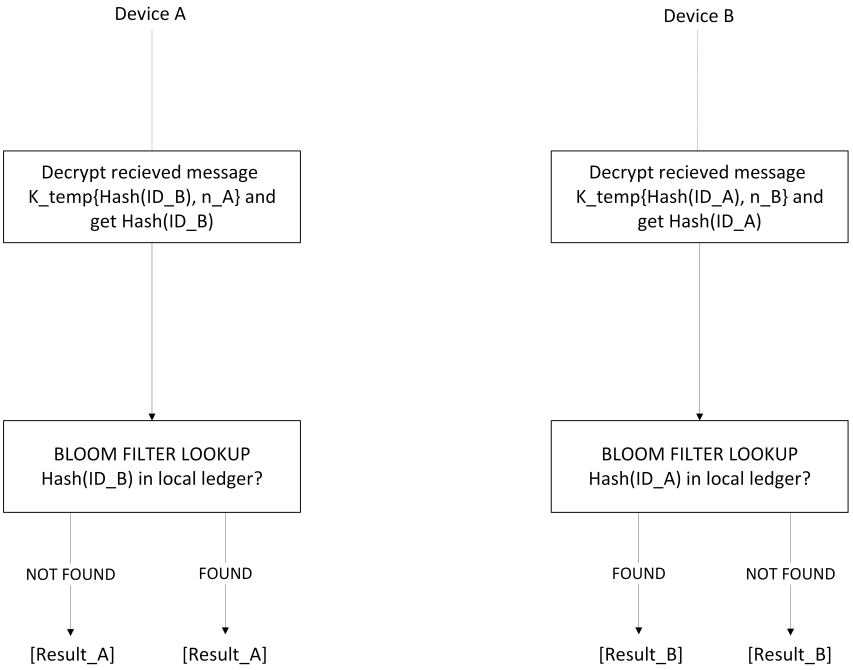


Figure 3.9 Ledger lookup

3.5.3 Step 2.3: Result Comparison

As seen in Figure 3.10, devices send each other their Bloom filter lookup results. If both devices found the other device in their local ledger, it means they share a record of a successful authentication. In this case, they can skip the Merkle tree verification phase and move directly to Phase 5 to finalize the authentication process. If one or both devices can't locate the other in their local ledgers, they must move to Phase 3 to use Merkle tree roots for verification.

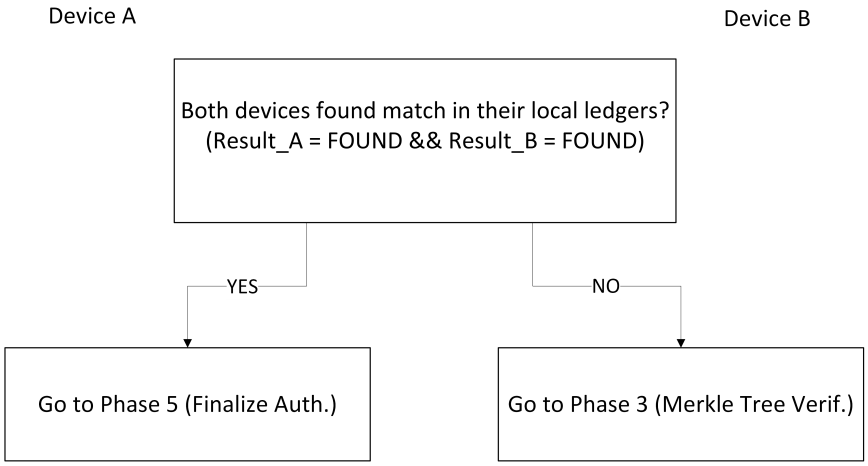


Figure 3.10 Result comparison

3.5.4 Step 2.4: Timeout Check

The standard timeout check, which is shown in Figure 3.11, is performed. To keep security assurances, this pattern repeats at the end of each phase. Before moving on to the next phase, both devices check if the time passed since T_{start} is longer than the maximum allowed time T_{max}, which is $(CurrentTime - T_{start} > T_{max})$, as shown in Figure 3.7. At the end of each phase, this timeout mechanism kicks in. If the timeout limit is reached, the connection is immediately terminated, and all session data is erased. If not reached, both devices move on to Phase 3 or Phase 5 according to the previous step's result.

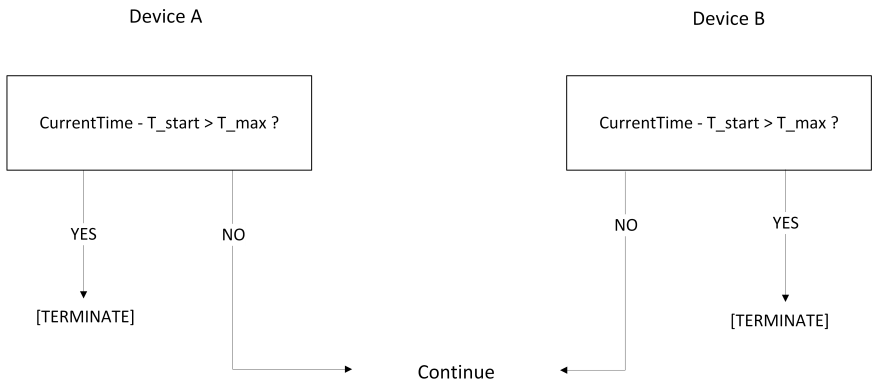


Figure 3.11 Timeout check

3.6 Phase 3: Merkle Tree Verification

Phase Overview: In Phase 2, if devices cannot locate each other's device IDs in their respective ledger, they compare their Merkle tree roots. There is a Merkle tree in every device that is created by their local ledgers. Devices can quickly find out if they share the exact ledgers or not by comparing their Merkle tree roots. The specific steps in this phase are outlined as follows:

3.6.1 Step 3.1: Merkle Root Exchange

As shown in Figure 3.12, devices exchange their Merkle tree roots with each other. Utilizing the Merkle tree roots is highly efficient, regardless of the number of the local ledger entries; only one hash value can be used to summarize the entire ledger. A Merkle tree is constructed in a way that any change to a node will yield a completely different root. Hence, the Merkle tree root serves as a fingerprint of the device's local ledger. A sends $K_{temp}(Root_A)$ to B. B sends $K_{temp}(Root_B)$ to A.

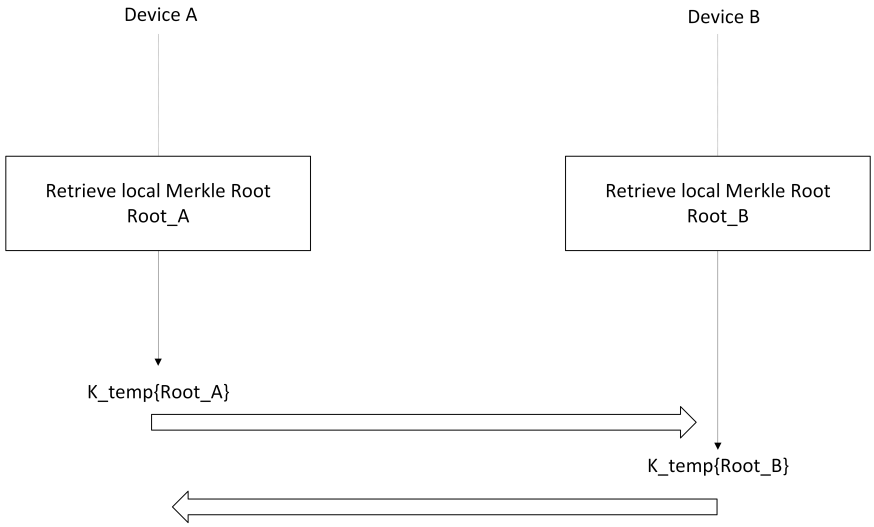


Figure 3.12 Merkle root exchange

3.6.2 Step 3.2: Merkle Roots Equality Comparison

As shown in Figure 3.13, both devices compare the received Merkle tree root with their own. If the roots are an exact match, it proves that both devices have identical ledgers. This is a very robust, impossible-to-replicate trust indicator; the authentication is successful, and they move on to Phase 5. If the roots are different, either the devices were setup during different time periods and have not received the updated ledger from the gateway or another trusted node, or they may be an adversary. Devices proceed to Phase 4 to reconcile these ledger differences.

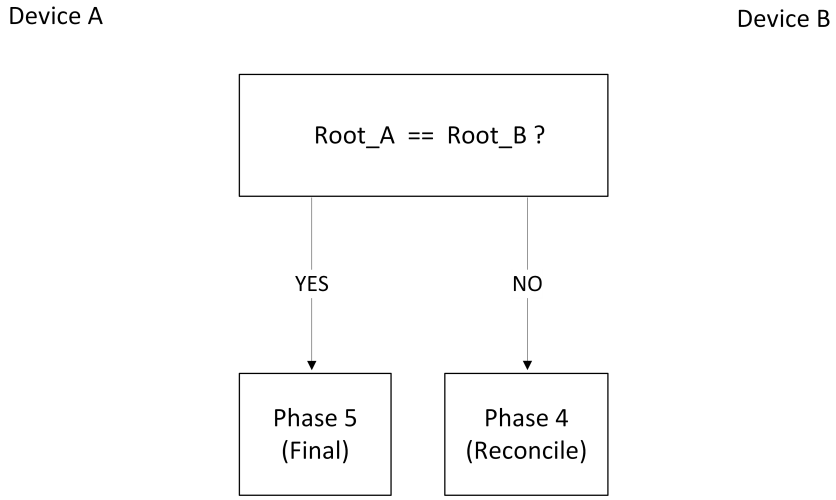


Figure 3.13 Merkle roots equality comparison

3.6.3 Step 3.3: Timeout Check

The standard timeout check, which is shown in Figure 3.14, is performed. To keep security assurances, this pattern repeats at the end of each phase. Before moving on to the next phase, both devices check if the time passed since T_{start} is longer than the maximum allowed time T_{max}, which is $(CurrentTime - T_{start} > T_{max})$, as shown in Figure 3.7. At the end of each phase, this timeout mechanism kicks in. If the timeout limit is reached, the connection is immediately terminated, and all session data is erased. If not reached, both devices move on to Phase 4 or Phase 5 according to the previous step's result.

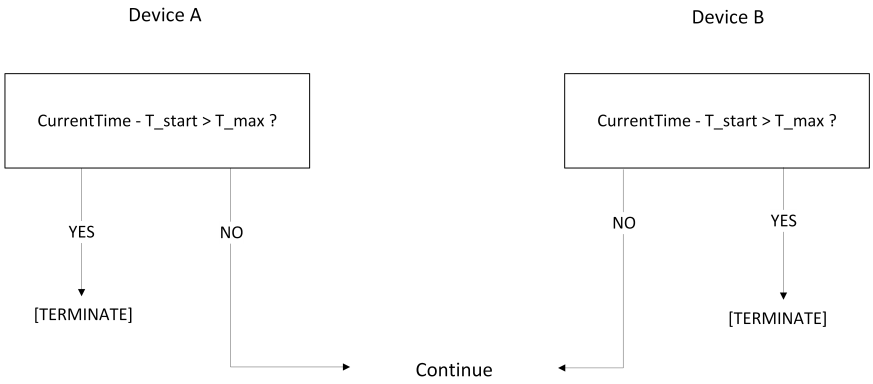


Figure 3.14 Timeout check

3.7 Phase 4: Merkle Tree Reconciliation

Phase Overview: When Merkle roots don't match, devices need to compare the entries in their ledgers. This phase involves switching to Wi-Fi to enable quick message transfer and exchanging iterative Merkle tree leaf proof to authenticate each other. The aim is to reach the trust threshold of 60% so that they can trust each other. This phase consists of the following steps:

3.7.1 Step 4.1: Wi-Fi Channel Switch and Iterative Proof Exchange

As shown in Figure 3.15, the reconciliation process involves exchanging multiple Merkle tree leaf proofs, which demands significant data transfer. BLE communication is energy-efficient but has limited bandwidth. Wi-Fi offers much higher data transfers with larger message capacity. Device A sends a hashed leaf of its Merkle tree $Hash(Proof(LR))$ to Device B and requests the corresponding Merkle tree leaf proof $Proof(LR)$. If Device B detects the matching leaf in its Merkle tree, it responds with the plain leaf proof $Proof(LR)$. Device A verifies this proof by comparing it to its own Merkle tree leaf hash. Since hash functions are one-way, they cannot be reverse engineered. If they match, it is verified that Device B indisputably possesses those entries.

This turn-by-turn exchange is reasonable and efficient. It continues until either a

sufficient trust threshold is achieved, or the 5-minute phase 4 specific timeout or message quota is reached. In addition to the standard timeout checks, in this phase, there exists another timeout limit of 5 minutes. Adding another timeout limit for the reconciliation step prevents resource exhaustion and DoS attacks.

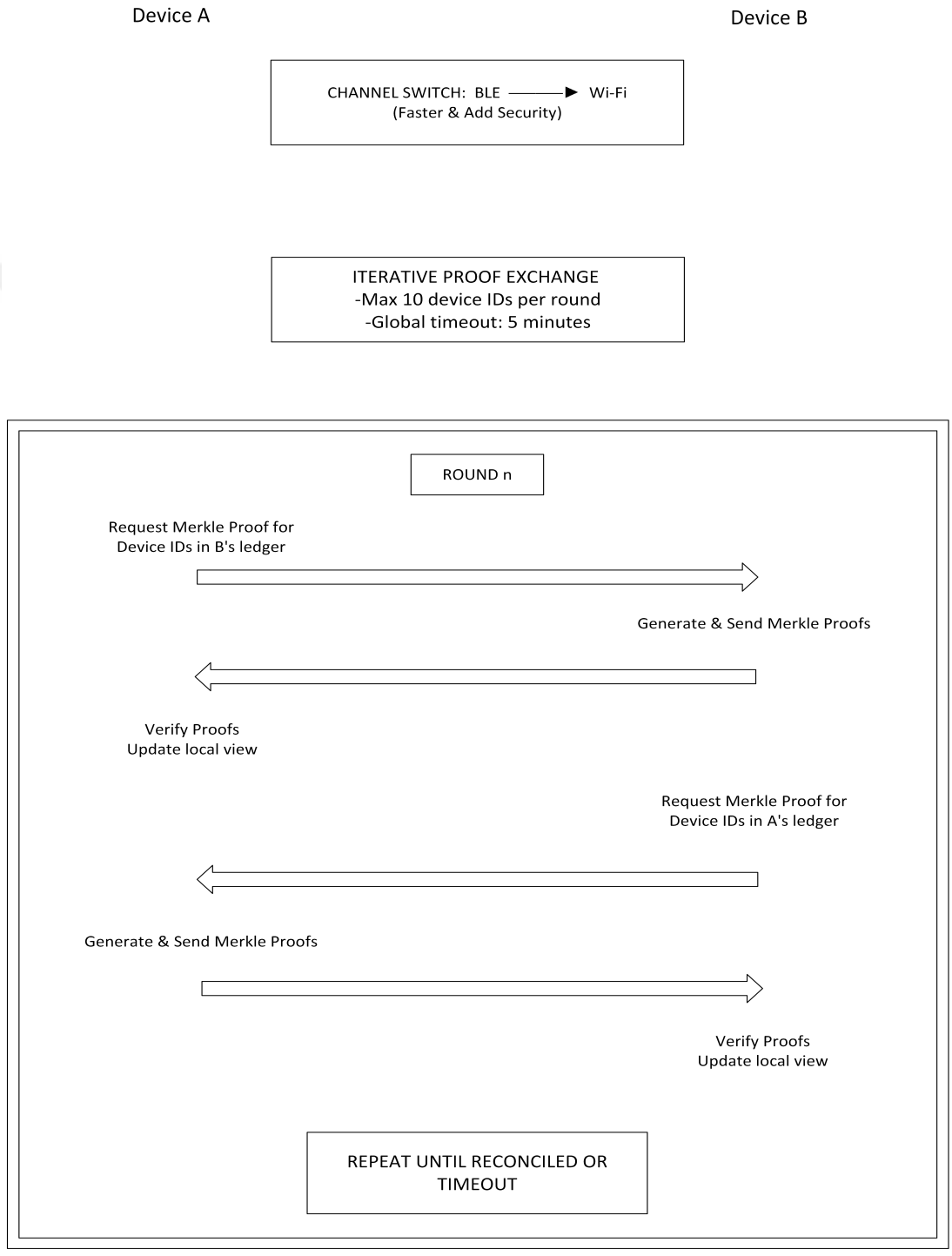


Figure 3.15 Wi-Fi channel switch and iterative proof exchange

3.7.2 Step 4.2: Trust Threshold Evaluation

After the iterative proof exchange, shown in Figure 3.16, both devices calculate the percentage of their own trust threshold. The 60% threshold is an assumed limit that balances confidence and practicality. It would be unreasonable to necessitate 100% overlap, but lowering the threshold to less than 60% would allow devices with little to no shared ledgers to authenticate. If the overlap is 60If it is less, the connection is terminated, meaning devices don't share the necessary common trusted device IDs in their local ledgers.

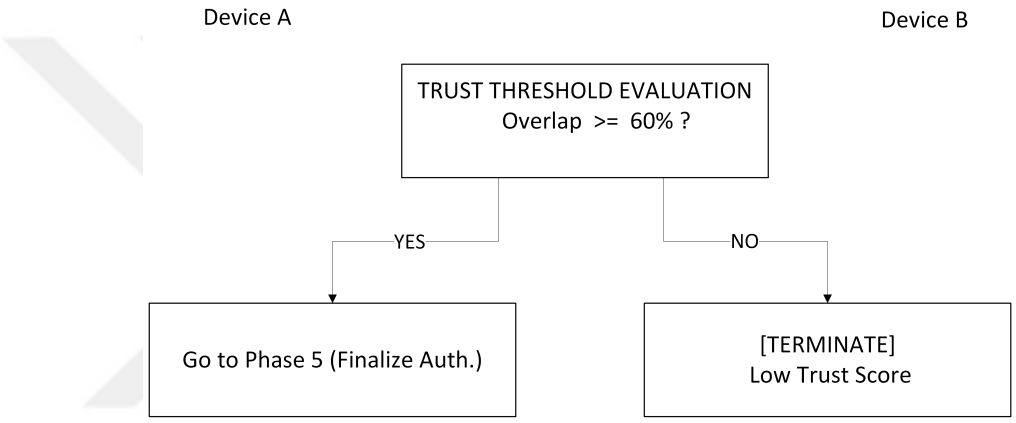


Figure 3.16 Trust threshold evaluation

3.7.3 Step 4.3: Timeout Check

The standard timeout check, which is shown in Figure 3.17, is performed. To keep security assurances, this pattern repeats at the end of each phase. Before moving on to the next phase, both devices check if the time passed since T_{start} is longer than the maximum allowed time T_{max}, which is $(CurrentTime - T_{start} > T_{max})$, as shown in Figure 3.7. At the end of each phase, this timeout mechanism kicks in. If the timeout limit is reached, the connection is immediately terminated, and all session data is erased. If not reached, both devices move on to Phase 5.

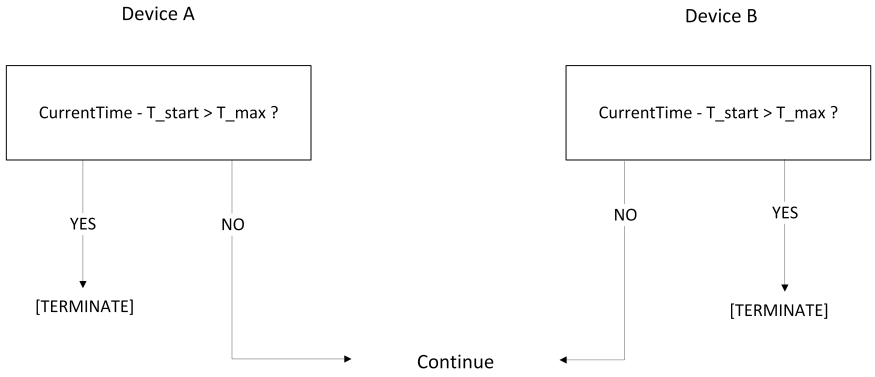


Figure 3.17 Timeout check

3.8 Phase 5: Finalizing Authentication and New Session Key Derivation

Phase Overview: This final phase establishes a new session key. Devices perform a fresh nonce exchange and derive the new session key. The new session key will encrypt all subsequent communication. To establish the new session key, the scheme proceeds through these steps:

3.8.1 Step 5.1: New Session Key Derivation

As shown in Figure 3.18, each device generates a fresh nonce n'_A and n'_B. These nonces are new and different from the nonces generated in Phase 1, confirming that the new session key will be different from the temporary session key. The new nonces are exchanged encrypted with K_{temp}. Both devices derive the new session key K_{new} using the $HKDF$ with the new nonces and the K_{DH} function: $K_{new} = HKDF(n'_A||n'_B||K_{DH})$.

The result is a new session key that delivers the forward secrecy with K_{temp}. If K_{temp} is compromised, K_{new} cannot be discovered, or the other way around. K_{new} encrypts all subsequent messages.

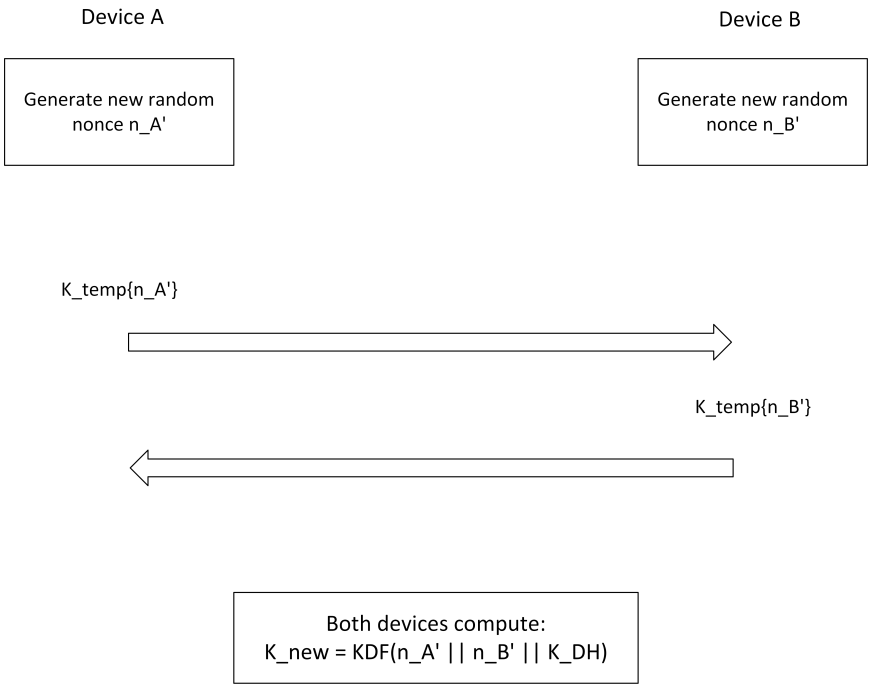


Figure 3.18 New session key derivation

3.8.2 Step 5.2: Final Timeout Check

One last timeout check makes sure that the whole scheme is completed within the timeout limit of 10 minutes, $(CurrentTime - T_{start} > T_{max})$, as shown in Figure 3.19. Even though each phase had its own timeout checks, this last timeout check can detect any edge cases where the total delays are too long.

If the timeout limit is reached, the new session key K_{new} must be erased; using a key from a session that has timed out could potentially yield an attack on the system. If the timeout limit is not reached, authentication is now complete.

Both devices now have a new session key, K_{new}, that they can use to encrypt and verify all future communications. The scheme has established a secure channel, a secure session key, and mutual authentication. Devices can share any private information, since the mutual trust is set.

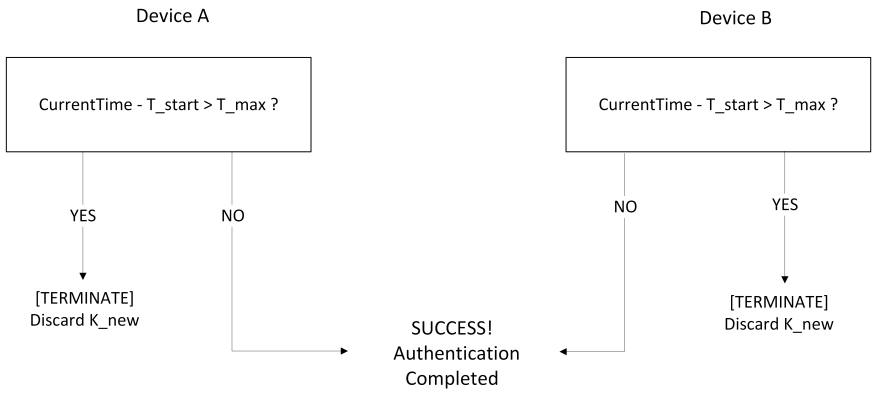


Figure 3.19 Final timeout check

3.9 BAN Logic Analysis

This section presents a formal BAN logic validation that proves the scheme's properties of mutual authentication, key freshness, forward secrecy, and secure session key establishment.

3.9.1 Notations

- A, B: Communicating IoT devices
- PK_X, SK_X: Public and private ECC key pair of device X
- PSK: 256-bit Pre-shared key
- n_X, n'_X: Nonces generated by device X during auth session
- K_{DH}: ECDH shared secret derived from key exchange
- K_{temp}: Temporary session key derived in Phase 1 (AES-256, 32 bytes)
- K_{new}: Final session key derived in Phase 5 (AES-256, 32 bytes)
- $H(ID_X)$: SHA-256 Hash of device X's ID
- $Root_X$: Merkle tree root of device X's trusted device ledger
- *MerkleProof*(LR): Merkle leaf proof of device or leaf denoted as LR

3.9.2 Initial State Assumptions

The security analysis starts with the following initial assumptions, established by the gateway device during setup:

$$A \mid\equiv AK_{PSK \leftrightarrow B} \quad \text{(A believes PSK is shared with B)} \quad (3.1)$$

$$B \mid\equiv AK_{PSK \leftrightarrow B} \quad \text{(B believes PSK is shared with A)} \quad (3.2)$$

$$A \mid\equiv \#(n_A) \quad \text{(A believes its nonce is fresh)} \quad (3.3)$$

$$B \mid\equiv \#(n_B) \quad \text{(B believes its nonce is fresh)} \quad (3.4)$$

$$A \mid\equiv (B \Rightarrow PK_B) \quad \text{(A trusts B's public key ownership)} \quad (3.5)$$

$$B \mid\equiv (A \Rightarrow PK_A) \quad \text{(B trusts A's public key ownership)} \quad (3.6)$$

$$A \mid\equiv \text{Owns}(SK_A, PK_A) \quad \text{(A owns its ECC key pair)} \quad (3.7)$$

$$B \mid\equiv \text{Owns}(SK_B, PK_B) \quad \text{(B owns its ECC key pair)} \quad (3.8)$$

$$A \mid\equiv \text{Authentic}(Root_A) \quad \text{(A trusts its Merkle root)} \quad (3.9)$$

$$B \mid\equiv \text{Authentic}(Root_B) \quad \text{(B trusts its Merkle root)} \quad (3.10)$$

$$\forall X : X \mid\equiv \text{Secure}(\text{AES-GCM}) \quad \text{(Trust in authenticated encryption)} \quad (3.11)$$

$$\forall X : X \mid\equiv \text{Secure}(\text{HMAC-SHA256}) \quad \text{(Trust in MAC algorithm)} \quad (3.12)$$

$$\forall X : X \mid\equiv \text{Secure}(\text{ECDH}) \quad \text{(Trust in key agreement)} \quad (3.13)$$

$$\forall X : X \mid\equiv \text{Secure}(\text{HKDF}) \quad \text{(Trust in key derivation)} \quad (3.14)$$

3.9.3 Implementation Assumptions

- There is a 1-byte type tag τ in each message to regulate synchronization
- A restart mechanism ($\tau_{restart} = 0xFF$) allows restart from desynchronization or an error during the session
- The ESP32's hardware RNG generates nonces with enough entropy
- AES-GCM's authentication tags deliver confidentiality and integrity
- Bloom filters can give false positives, but never give false negatives

- T_{max}: Session timeout limit of 10 minutes

3.9.4 Scheme Message Formalization

There are five phases in the scheme, and it has fourteen distinct messages. Each message has a formal description of its cryptographic construction.

3.9.5 Phase 1: Initial Handshake (Key Agreement)

Purpose: Create a temporary session key K_{temp} using ECDH with PSK.
Message Construction: Device A generates a temporary ECC key pair (SK_A, PK_A) and fresh nonce n_A, then sends:

$$\textbf{Msg 1:} \quad A \rightarrow B : \left\{PK_A \parallel n_A \parallel \text{HMAC}_{PSK}(PK_A \parallel n_A)\right\}_{PSK} \tag{3.15}$$

The message contains:

- Temporary ECC public key PK_A
- Fresh nonce n_A
- HMAC tag for integrity confirmation
- The entire message encrypted with AES-256-GCM based PSK

When device B gets Msg 1:

1. Decrypts with PSK which verifies the GCM authentication tag
2. Validates $\text{HMAC}_{PSK}(PK_A \parallel n_A)$
3. If both successfull: $B \mid\equiv A \mid\sim (PK_A, n_A)$

Device B responds symmetrically:

$$\textbf{Msg 2:} \quad B \rightarrow A : \left\{PK_B \parallel n_B \parallel \text{HMAC}_{PSK}(PK_B \parallel n_B)\right\}_{PSK} \tag{3.16}$$

Upon receiving Msg 2, device A:

1. Decrypts and verifies HMAC
2. If valid: $A \mid\equiv B \mid\sim (PK_B, n_B)$

Key Derivation: Both devices compute the shared secret and temporary key:

$$K_{DH} = \text{ECDH}(SK_A, PK_B) = \text{ECDH}(SK_B, PK_A) \tag{3.17}$$

$$K_{temp} = \text{HKDF}(n_A \parallel n_B \parallel K_{DH}) \tag{3.18}$$

Goals Achieved:

$$A \mid\equiv (AK_{K_{temp} \leftrightarrow B}) \quad \text{(Shared temporary key)} \tag{3.19}$$

$$B \mid\equiv (AK_{K_{temp} \leftrightarrow B}) \tag{3.20}$$

$$A \mid\equiv \#(K_{temp}) \quad \text{(Key freshness from nonces)} \tag{3.21}$$

$$B \mid\equiv \#(K_{temp}) \tag{3.22}$$

3.9.6 Phase 2: Ledger Verification (Identity Exchange)

Purpose: Exchange hashed device IDs to authenticate corresponding existence in both device ledgers.

$$\textbf{Msg 3:} \quad A \rightarrow B : \left\{ H(ID_A) \parallel n_B \right\}_{K_{temp}} \tag{3.23}$$

$$\textbf{Msg 4:} \quad B \rightarrow A : \left\{ H(ID_B) \parallel n_A \right\}_{K_{temp}} \tag{3.24}$$

Construction Details:

- $H(ID_X)$: SHA-256 hash of device's ID
- Peer's nonce is included for to protect against replay attacks
- Encrypted with AES-256-GCM based K_{temp}

Verification Process:

1. Decrypt the message and verify GCM tag
2. Confirm the nonce corresponds to the expected value
3. Verify the Bloom filter for $H(ID_{peer})$
4. If found, conduct a precise ledger search
5. Exchange lookup outcomes

$$\textbf{Msg 5:} \quad A \rightarrow B : \left\{ result_A \parallel n_B \right\}_{K_{temp}} \tag{3.25}$$

$$\textbf{Msg 6:} \quad B \rightarrow A : \left\{ result_B \parallel n_A \right\}_{K_{temp}} \tag{3.26}$$

Fast-Path Optimization: If both devices identify each other in their opposing ledgers ($result_A = result_B = \top$), Phases 3 and 4 are skipped, directly to Phase 5:

$$(result_A \wedge result_B) \implies \text{Skip to Phase 5}$$

3.9.7 Phase 3: Merkle Root Exchange

Purpose: Compare the Merkle tree roots to detect differences in the ledgers.

$$\textbf{Msg 7:} \quad A \rightarrow B : \left\{ Root_A \right\}_{K_{temp}} \tag{3.27}$$

$$\textbf{Msg 8:} \quad B \rightarrow A : \left\{ Root_B \right\}_{K_{temp}} \tag{3.28}$$

Construction:

- $Root_X$: Merkle tree root hash
- Built from SHA-256 hashes of all trusted device IDs Made from SHA-256 hashes of all device IDs that are trusted

Decision Logic:

$$Root_A = Root_B \implies \text{Skip the Phase 4, proceed directly to Phase 5}$$

$$Root_A \neq Root_B \implies \text{Proceed to Phase 4, reconciliation step}$$

3.9.8 Phase 4: Merkle Tree Reconciliation

Purpose: When Merkle tree roots do not match, find and verify ledger entries with Merkle tree leaf proofs. **Iterative Proof Exchange:** For each device or leaf LR that necessitates verification:

$$\textbf{Msg 9:} \quad A \leftrightarrow B : \left\{H(MerkleProof(LR))\right\}_{K_{temp}} \quad \text{(Request)} \tag{3.29}$$

$$\textbf{Msg 10:} \quad B \leftrightarrow A : \left\{\textit{MerkleProof}(LR)\right\}_{K_{temp}} \quad \text{(Response)} \tag{3.30}$$

Privacy Feature: Hashing ($H(MerkleProof(LR))$) blocks plaintext identity exposure.

Trust Threshold:

$$\frac{\textit{VerifiedCommon}}{\textit{TotalSentProofs}} \geq 60\% \implies \text{Proceed to Phase 5}$$

Timeout: Phase 4 has a 5-minute session timeout deadline to protect against resource exhaustion attacks.

Message Quota: Phase 4 has a 10-request limit per device during reconciliation to protect against multiple attacks.

3.9.9 Phase 5: Final Key Derivation and Confirmation

Purpose: Mutually create a shared fresh new session key and explicitly confirm it.

Fresh Nonce Exchange: Both devices generate new nonces n'_A, n'_B for the final phase:

$$A \mid\equiv \#(n'_A), \quad B \mid\equiv \#(n'_B)$$

$$\textbf{Msg 11:} \quad A \rightarrow B : \left\{n'_A\right\}_{K_{temp}} \tag{3.31}$$

$$\textbf{Msg 12:} \quad B \rightarrow A : \left\{n'_B\right\}_{K_{temp}} \tag{3.32}$$

New Session Key Derivation:

$$K_{new} = \text{HKDF}(n'_A \parallel n'_B \parallel K_{DH}) \tag{3.33}$$

The new session key K_{new} incorporates:

- Fresh nonces (n'_A, n'_B) : ensures key freshness
- Original ECDH secret (K_{DH}) : provides forward secrecy

Explicit Key Confirmation: This step delivers cryptographic proof that both parties derived the same key:

$$\textbf{Msg 13:} \quad A \rightarrow B : \left\{\text{“AUTH_COMPLETE”}\right\}_{K_{new}} \tag{3.34}$$

$$\textbf{Msg 14:} \quad B \rightarrow A : \left\{\text{“AUTH_COMPLETE”}\right\}_{K_{new}} \tag{3.35}$$

Verification:

1. Decrypt received message using derived K_{new}
2. Verify plaintext equals “AUTH_COMPLETE”
3. If verification fails terminate the session and delete all keys

Final Goals Achieved:

$$A \mid\equiv (AK_{K \leftrightarrow B}) \qquad \text{(Mutual key belief)} \tag{3.36}$$

$$B \mid\equiv (AK_{K \leftrightarrow B}) \tag{3.37}$$

$$A \mid\equiv B \mid\equiv (AK_{K_{new} \leftrightarrow B}) \qquad \text{(Key confirmation)} \tag{3.38}$$

$$B \mid\equiv A \mid\equiv (AK_{K_{new} \leftrightarrow B}) \tag{3.39}$$

$$A \mid\equiv \#(K_{new}) \qquad \text{(Key freshness)} \tag{3.40}$$

$$B \mid\equiv \#(K_{new}) \tag{3.41}$$

Secure Cleanup: After successful completion:

- K_{temp} is securely zeroed
- K_{DH} is securely zeroed
- All nonces are securely zeroed
- Private key SK_X is cleared from memory

3.9.10 Security Properties Proven

3.9.10.1 Mutual Authentication

Both parties can verify each other's identity through:

1. PSK-authenticated handshake; only valid devices possess PSK
2. HMAC verification proves knowledge of PSK
3. Successful decryption of AUTH_COMPLETE message proves key derivation succeeded

3.9.10.2 Key Freshness

Session key K_{new} is fresh because:

1. It incorporates n'_A (fresh, generated by A)
2. It incorporates n'_B (fresh, generated by B)
3. Neither party can predict the other's nonce

3.9.10.3 Forward Secrecy

Retrieval of a session key does not reveal past session keys because:

1. Session keys depend on ephemeral ECDH (K_{DH})
2. Temporary ECC private keys (SK_A, SK_B) are deleted after use
3. K_{DH} cannot be recovered without temporary private keys

3.9.10.4 Replay Protection

Messages cannot be replayed because:

1. Each message includes a nonce from the peer
2. Nonces are verified to match expected values
3. Scheme has a session timeout limit ($T_{max} = 10$ minutes)

3.9.11 Message Summary

Table 3.2 provides a summary of exchanged messages our scheme's BAN logic representations during a complete authentication session.

Table 3.2 Complete scheme message summary

Msg	Phase	Content	Encryption Key
1	1	$\{PK_A \parallel n_A \parallel \text{HMAC}_{PSK}(PK_A \parallel n_A)\}_{PSK}$	PSK
2	1	$\{PK_B \parallel n_B \parallel \text{HMAC}_{PSK}(PK_B \parallel n_B)\}_{PSK}$	PSK
3	2	$\{H(ID_A) \parallel n_B\}_{K_{temp}}$	K_{temp}
4	2	$\{H(ID_B) \parallel n_A\}_{K_{temp}}$	K_{temp}
5	2	$\{result_A \parallel n_B\}_{K_{temp}}$	K_{temp}
6	2	$\{result_B \parallel n_A\}_{K_{temp}}$	K_{temp}
7	3	$\{Root_A\}_{K_{temp}}$	K_{temp}
8	3	$\{Root_B\}_{K_{temp}}$	K_{temp}
9	4	$\{H(MerkleProof(LR))\}_{K_{temp}}$	K_{temp}
10	4	$\{MerkleProof(LR)\}_{K_{temp}}$	K_{temp}
11	5	$\{n'_A\}_{K_{temp}}$	K_{temp}
12	5	$\{n'_B\}_{K_{temp}}$	K_{temp}
13	5	$\{\text{“AUTH_COMPLETE”}\}_{K_{new}}$	K_{new}
14	5	$\{\text{“AUTH_COMPLETE”}\}_{K_{new}}$	K_{new}

CHAPTER FOUR
SCHEME IMPLEMENTATION AND TESTING

Moving from theoretical design to conducting firmware testing is an important step in the development of any IoT security framework. This process ensures that proposed schemes do work within the strict constraints of embedded hardware. This chapter describes the methodical implementation and testing process run on the ESP32 microcontroller, particularly the ESP32-WROOM-32 model. We chose the ESP32 device because it is widely available, low cost and used extensively in IoT settings.

Our scheme implementation uses the open source mbedtls library to find a good balance between security and efficiency. We measured the scheme performance by session and phase execution times, energy consumption and transferred bandwidth. This was done to detect any performance issues in the authentication process and to analyze the possible improvements. The Scyther formal security tool was used to verify the scheme's security claims, which the details will be discussed in the next chapter.

Espressif Systems developed and produced the ESP32-WROOM-32, a low-cost, low-power system-on-chip microcontroller that is frequently used in IoT, telemetry, and industrial settings. It has built-in wireless connectivity and hardware security features. The main features of the ESP32-WROOM-32 module are as follows: The module is powered by the Xtensa dual-core 32-bit LX6 microprocessor, which can run at up to 240 MHz. The dual-core design makes it effective to run separate tasks: one core handles application logic and the other handles wireless communication protocols. The LX6 is based on TSMC's 40nm ultra-low-power processor and is developed to work efficiently in battery-operated systems.

During runtime, the module provides about 520 KB of SRAM for data and instructions. It has 448 KB of ROM for boot and core functions. It has an external SPI flash memory of 4 MB for storing firmwares. It has 1 KB of eFuse memory for storing system settings, MAC addresses, and encryption keys. This eFuse memory

stores the flash encryption and secure boot private keys. The module has built-in 2.4 GHz Wi-Fi and supports 802.11 b/g/n/e/i protocols. It also supports Classic Bluetooth and Bluetooth Low Energy version 4.2. It has a dedicated hardware accelerator that works with AES, SHA-2, RSA, and Elliptic Curve Cryptography. It supports hardware-verified secure boot, which means that only verified and signed code can run on the device. It supports flash encryption, the module encrypts external SPI flash using AES-256, which keeps firmware and other private data secure. Lastly, it has a real hardware random number generator that can be used to generate unique nonces and cryptographic keys (Espressif Systems, 2025).

4.1 Implementation Overview

The scheme was implemented on the ESP32 device using the Arduino IDE and the Arduino C++ programming language. Arduino was chosen because it is easy to run, has a large community support, and integrates open-source libraries directly in the IDE. Because the source code is very large and complex, this section provides only a selected number of code snippets . The scheme's more detailed implementation code snippets can be found in Appendix 2.

4.1.1 Message Type Classification

```
#define MSG_TYPE_HANDSHAKE     0x01
#define MSG_TYPE_ID_HASH       0x02
#define MSG_TYPE_LOOKUP_RESULT 0x03
#define MSG_TYPE_MERKLE_ROOT   0x04
#define MSG_TYPE_PROOF_REQ     0x05
#define MSG_TYPE_PROOF_RESP    0x06
#define MSG_TYPE_NONCE         0x07
#define MSG_TYPE_AUTH_COMPLETE 0x08
#define MSG_TYPE_WIFI_INFO     0x09
#define MSG_TYPE_RESTART       0xFF
```

Figure 4.1 Message type classification

Figure 4.1 shows that each message in the scheme has a 1-byte tag that is sent in plain

text. These tags let the receiver device identify what is the current step of the sender device, without decrypting any message, which makes it quick to detect if there is a desynchronization between devices. (`MSG_TYPE_RESTART (0xFF)`) restarts the authentication process without terminating the device connection.

4.1.2 SHA-256 Hash Function

```
void calculateSHA256(const uint8_t *data, size_t dataLen, uint8_t *output) {
  mbedtls_sha256_context ctx;
  mbedtls_sha256_init(&ctx);
  mbedtls_sha256_starts(&ctx, 0);
  mbedtls_sha256_update(&ctx, data, dataLen);
  mbedtls_sha256_finish(&ctx, output);
  mbedtls_sha256_free(&ctx);
}
```

Figure 4.2 SHA-256 hash function

Figure 4.2 shows that SHA-256 generates a 256-bit standard hash from any length input, which provides collision-resistance. It is the core hash function that the entire scheme is built on. It is used for HMAC authentication, HKDF key derivation, building Merkle trees, computing Bloom filter positions, and hashing device IDs. The mbedtls library is used with ESP32's hardware cryptographic accelerator to improve performance.

4.1.3 Shared ECDH Computation

```
int generateKeyPair(ECCKeyPair *kp) {
  int ret = mbedtls_ecp_group_load(&kp->grp, MBEDTLS_ECP_DP_SECP256R1);
  if (ret != 0) return ret;
  ret = mbedtls_ecp_gen_keypair(&kp->grp, &kp->d, &kp->Q,mbedtls_ctr_drbg_random,
  &ctr_drbg);
  return ret; }
```

Figure 4.3 Key pair generation

```
int computeECDH(ECCKeyPair *myKey, const uint8_t *peerPubKey, size_t peerKeyLen,
uint8_t *sharedSecret, size_t *secretLen) {
  int ret;
  mbedtls_ecp_point peerQ;
  mbedtls_mpi z;
  mbedtls_ecp_point_init(&peerQ);
  mbedtls_mpi_init(&z);
  ret = mbedtls_ecp_point_read_binary(&myKey->grp, &peerQ, peerPubKey,
peerKeyLen);
  if (ret != 0) {
    mbedtls_ecp_point_free(&peerQ);
    mbedtls_mpi_free(&z);
    return ret; }
  ret = mbedtls_ecdh_compute_shared(&myKey->grp, &z, &peerQ, &myKey->d,
  ↪ mbedtls_ctr_drbg_random, &ctr_drbg);
  if (ret == 0) {
    *secretLen = mbedtls_mpi_size(&z);
    ret = mbedtls_mpi_write_binary(&z, sharedSecret, *secretLen);
  }
  mbedtls_mpi_free(&z);
  mbedtls_ecp_point_free(&peerQ);
  return ret; }
```

Figure 4.4 Shared ECDH computation

Figures 4.3 and 4.4 show that ECDH key agreement uses the secp256r1 curve, which gives it 128-bit security. Each device creates a temporary public-private key pair, and both devices use the ECDH algorithm to compute the same shared secret. This shared secret is never transmitted between devices and is used only as input to derive the session key.

4.1.4 AES-256-GCM

```
int aesGcmEncrypt(const uint8_t *key, const uint8_t *iv,
const uint8_t *plaintext, size_t ptLen, const uint8_t *aad,
size_t aadLen, uint8_t *ciphertext, uint8_t *tag) {
  mbedtls_gcm_context ctx;
  mbedtls_gcm_init(&ctx);
  int ret = mbedtls_gcm_setkey(&ctx, MBEDTLS_CIPHER_ID_AES, key, AES_KEY_SIZE * 8);
  if (ret == 0) {
    ret = mbedtls_gcm_crypt_and_tag(&ctx, MBEDTLS_GCM_ENCRYPT, ptLen, iv, GCM_IV_SIZE, aad,
  ↪ aadLen, plaintext, ciphertext, GCM_TAG_SIZE, tag); }
  mbedtls_gcm_free(&ctx);
  return ret; }
```

Figure 4.5 AES-256-GCM encryption

```
int aesGcmDecrypt(const uint8_t *key, const uint8_t *iv, const uint8_t *ciphertext,
  size_t ctLen, const uint8_t *aad, size_t aadLen, const uint8_t *tag, uint8_t *plaintext) {
  mbedtls_gcm_context ctx;
  mbedtls_gcm_init(&ctx);
  int ret = mbedtls_gcm_setkey(&ctx,MBEDTLS_CIPHER_ID_AES, key, AES_KEY_SIZE * 8);
  if (ret == 0) { ret = mbedtls_gcm_auth_decrypt(&ctx, ctLen, iv,   GCM_IV_SIZE, aad, aadLen
  ↪ , tag, GCM_TAG_SIZE, ciphertext, plaintext); }
  mbedtls_gcm_free(&ctx);
  return ret; }
```

Figure 4.6 AES-256-GCM decryption

As shown in 4.5 and 4.6 AES-256-GCM is the encryption technique that provides both privacy and integrity to our scheme. It is used to encrypt all messages during the authentication session. A randomly generated 96-bit IV makes sure that the ciphertext is unique, and the 16-byte authentication tag finds any changes to the ciphertext. If tag verification fails, decryption fails as well. Tag verification prevents unauthenticated data from being accepted by the devices.

4.1.5 HMAC-Based Message Authentication

```
int calculateHMAC(const uint8_t *key, size_t keyLen, const uint8_t *data, size_t dataLen,
  ↪ uint8_t *output) {
  mbedtls_md_context_t ctx;
  mbedtls_md_type_t md_type = MBEDTLS_MD_SHA256;
  mbedtls_md_init(&ctx);
  int ret = mbedtls_md_setup(&ctx, mbedtls_md_info_from_type(md_type), 1);
  if (ret == 0)
    ret = mbedtls_md_hmac_starts(&ctx, key, keyLen);
  if (ret == 0)
    ret = mbedtls_md_hmac_update(&ctx, data, dataLen);
  if (ret == 0)
    ret = mbedtls_md_hmac_finish(&ctx, output);
  mbedtls_md_free(&ctx);
  return ret; }
  [...]
  if (calculateHMAC(key, keyLen, data, dataLen, calculated) != 0) {
    return false; }
  return (memcmp(calculated, expectedHMAC, HASH_SIZE) == 0); }
```

Figure 4.7 HMAC functions

Shown in Figure 4.7, HMAC-SHA256 is a way to verify the authenticity of a message while making it tamper-proof. It accomplishes by using the nested construction H(K $\oplus$ opad || H(K $\oplus$ ipad || message), which stops length extension attacks. HMAC verification uses constant time comparison to stop any side-channel attacks.

4.1.6 HMAC-Based Key Derivation (HKDF)

```
int deriveKey(const uint8_t *ikm, size_t ikmLen, const uint8_t *salt,size_t saltLen, const
  ↪ uint8_t *info, size_t infoLen, uint8_t *okm,size_t okmLen) {
  const mbedtls_md_info_t *md_info =
  mbedtls_md_info_from_type(MBEDTLS_MD_SHA256);
  return mbedtls_hkdf(md_info, salt, saltLen, ikm, ikmLen, info, infoLen, okm, okmLen); }
```

Figure 4.8 HKDF implementation

```
uint8_t ikm[NONCE_SIZE * 2 + ECC_KEY_SIZE];
int cmp = memcmp(state.myNonce, state.peerNonce, NONCE_SIZE);
const uint8_t *nonce1 = (cmp <= 0) ? state.myNonce : state.peerNonce;
const uint8_t *nonce2 = (cmp <= 0) ? state.peerNonce : state.myNonce;
memcpy(ikm, nonce1, NONCE_SIZE);
memcpy(ikm + NONCE_SIZE, nonce2, NONCE_SIZE);
memcpy(ikm + NONCE_SIZE * 2, state.sharedSecret, secretLen);
const char *info = "temp-session-key";
ret = deriveKey(ikm, sizeof(ikm), NULL, 0, (const uint8_t *)info,strlen(info), state.
  ↪ tempSessionKey, AES_KEY_SIZE);
```

Figure 4.9 HKDF usage in phase 1

Figure 4.8 shows that HKDF-SHA256 function derives the session key by going through extract-and-expand stages. Figure 4.9 show its usage in Phase 1. The two session keys generated in Phase 1 and Phase 5 are kept separate by session key tags: `temp-session-key` and `final-session-key`. A key part of the HKDF is the deterministic nonce ordering. Before HKDF, nonces are sorted. If the nonce order is different, the session keys would be different.

4.1.7 Bloom Filter Device Lookup

```
void bloomFilterGetPositions(const uint8_t *data,uint16_t positions[BLOOM_HASH_COUNT]) {
  for (int h = 0; h < BLOOM_HASH_COUNT; h++) {
    uint8_t hashInput[HASH_SIZE + 1];
    memcpy(hashInput, data, HASH_SIZE);
    hashInput[HASH_SIZE] = (uint8_t)h; // Salt with hash index
    uint8_t hashOut[HASH_SIZE];
    calculateSHA256(hashInput, HASH_SIZE + 1, hashOut);
    positions[h] = ((hashOut[0] << 8) | hashOut[1]) % BLOOM_SIZE_BITS; }}
```

Figure 4.10 Bloom filter positions

```
void bloomFilterAdd(const uint8_t *hash) {
  uint16_t positions[BLOOM_HASH_COUNT];
  bloomFilterGetPositions(hash, positions);
  for (int i = 0; i < BLOOM_HASH_COUNT; i++) {
    bloomFilter[positions[i] / 8] |= (1 << (positions[i] % 8)); } }
bool bloomFilterCheck(const uint8_t *hash) {
  uint16_t positions[BLOOM_HASH_COUNT];
  bloomFilterGetPositions(hash, positions);
  for (int i = 0; i < BLOOM_HASH_COUNT; i++) {
    if ((bloomFilter[positions[i] / 8] & (1 << (positions[i] % 8))) == 0) {return false; } }
  ↪ return true; }
```

Figure 4.11 Bloom filter operations

The Bloom filter provides a local ledger with an O(1) probabilistic existence lookup, as shown in Figures 4.10 and 4.11. Any negative result is definitive; there could be a small chance of false positives, which are fixed by a subsequent ledger lookup process.

4.1.8 Merkle Tree Construction

```
void merkleTreeBuild(MerkleTree *tree) {
  if (tree->leafCount == 0) {
    memset(tree->root, 0, HASH_SIZE);
    tree->isBuilt = true;
    return; }
  uint16_t nodeCount = tree->leafCount;
  for (uint16_t i = 0; i < nodeCount; i++) {
    memcpy(tree->nodes[i], tree->leaves[i], HASH_SIZE); }
  uint16_t levelStart = 0;
  uint16_t nextLevelStart = nodeCount;
  while (nodeCount > 1) {
    uint16_t newNodeCount = 0;
    [...]
    levelStart = nextLevelStart;
    nextLevelStart += newNodeCount;
    nodeCount = newNodeCount; }
  memcpy(tree->root, tree->nodes[levelStart], HASH_SIZE);
  tree->isBuilt = true; }
```

Figure 4.12 Merkle tree build

Figure 4.12 shows how to build a binary Merkle tree from the bottom up using SHA-

256 hashes of trusted device IDs in the local ledger. The root hash, which is 32 bytes long, is a single cryptographic summary of the whole ledger. Any change to a leaf changes every hash above it, which changes the root hash.

4.1.9 Merkle Tree Proof Generation and Verification

```
bool merkleTreeGenerateProof(MerkleTree *tree, uint16_t leafIndex,
  uint8_t proof[][HASH_SIZE], int *proofLen) {
  if (!tree->isBuilt || leafIndex >= tree->leafCount) {
    *proofLen = 0;
    return false; }
  *proofLen = 0;
  uint16_t nodeCount = tree->leafCount;
  uint16_t levelStart = 0;
  uint16_t currentIndex = leafIndex;
  while (nodeCount > 1) {
    uint16_t siblingIndex;
    if (currentIndex % 2 == 0) { siblingIndex = (currentIndex + 1 < nodeCount) ?
                                 currentIndex + 1 : currentIndex; }
    else { siblingIndex = currentIndex - 1; }
    memcpy(proof[*proofLen], tree->nodes[levelStart + siblingIndex], HASH_SIZE);
    (*proofLen)++;
    currentIndex = currentIndex / 2;
    levelStart += nodeCount;
    nodeCount = (nodeCount + 1) / 2; }
  return true; }
```

Figure 4.13 Merkle tree proof generation

```
bool merkleTreeVerifyProof(const uint8_t *root, const uint8_t *leaf,
  uint8_t proof[][HASH_SIZE], int proofLen,uint16_t leafIndex) {
  uint8_t current[HASH_SIZE];
  memcpy(current, leaf, HASH_SIZE);
  uint16_t index = leafIndex;
  for (int i = 0; i < proofLen; i++) {
    uint8_t combined[HASH_SIZE * 2];
    if (index % 2 == 0) { memcpy(combined, current, HASH_SIZE);
                          memcpy(combined + HASH_SIZE, proof[i], HASH_SIZE); }
    else { memcpy(combined, proof[i], HASH_SIZE);
           memcpy(combined + HASH_SIZE, current, HASH_SIZE); }
    calculateSHA256(combined, HASH_SIZE * 2, current);
    index = index / 2; }
  return memcmp(current, root, HASH_SIZE) == 0; }
```

Figure 4.14 Merkle proof verification

Figures 4.13 and 4.14 show that a Merkle tree is constructed of sibling hashes along the path from a leaf up to the root. The receiver must send the Merkle tree proof by comparing the received hash proof with its Merkle tree, calculating each leaf switching between left and right children. If the matching leaf or node is found, it is sent plainly to the other device. In Phase 4, devices share inclusion proofs to check for shared ledger entries and set a trust level based on a threshold overlap, without sharing the entire ledger.

4.1.10 BLE Communication Layer

```
void initBLEServer() {
  String deviceName = "ESP32-" + String((uint32_t)ESP.getEfuseMac(), HEX);
  BLEDevice::init(deviceName.c_str()); BLEDevice::setMTU(185);
  pServer = BLEDevice::createServer();
  pServer->setCallbacks(new ServerCallbacks());
  BLEService *pService = pServer->createService(SERVICE_UUID);
  pCharacteristic = pService->createCharacteristic(
        CHARACTERISTIC_UUID,BLECharacteristic::PROPERTY_READ |
  ↪ BLECharacteristic::PROPERTY_WRITE | BLECharacteristic::PROPERTY_NOTIFY);
  pCharacteristic->setCallbacks(new CharacteristicCallbacks());
  pCharacteristic->addDescriptor(new BLE2902());
  pService->start();
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  pAdvertising->addServiceUUID(SERVICE_UUID);
  pAdvertising->setScanResponse(true);
  pAdvertising->setMinPreferred(0x06);
  pAdvertising->setMinPreferred(0x12);
  BLEDevice::startAdvertising();
  bleServerMode = true; }
```

Figure 4.15 BLE server initialization

```
int bleReceiveMessage(uint8_t *buffer, size_t maxLen, uint32_t timeoutMs) {
  if (bleMessageReady && bleBufferLen > 0) {
    [...]
    size_t copyLen = (bleBufferLen < maxLen) ? bleBufferLen : maxLen;
    memcpy(buffer, bleBuffer, copyLen);
    int len = bleBufferLen;
    bleBufferLen = 0;
    bleMessageReady = false;
    return len; }
  unsigned long start = millis();
  [...]
  if (bleMessageReady && bleBufferLen > 0) {
    if (bleBuffer[0] == MSG_TYPE_RESTART) {
      bleBufferLen = 0;
      bleMessageReady = false;
      return -1; }
    size_t copyLen = (bleBufferLen < maxLen) ? bleBufferLen : maxLen;
    memcpy(buffer, bleBuffer, copyLen);
    int len = bleBufferLen;
    bleBufferLen = 0;
    bleMessageReady = false;
    return len; }
  return 0; }
```

Figure 4.16 BLE message reception

Figures 4.15 and 4.16 show that the BLE communication is the handshake and the communication channel for Phases 1 through 3. It uses GATT with a custom service and characteristics that support read, write, and notify operations. BLE communication keeps earlier phase message exchanges lightweight while maintaining two directional communication between devices.

4.1.11 WiFi Channel Switching

```
if (bleServerMode) {
  Serial.println("Role: SERVER (SoftAP)");
  state.isServer = true;
  uint8_t iv[GCM_IV_SIZE];
  generateNonce(iv, GCM_IV_SIZE);
  uint8_t plaintext[128];
  memset(plaintext, 0, sizeof(plaintext));
  char ssid[64];
  uint64_t mac = ESP.getEfuseMac();
  uint32_t mac32 = (mac & 0xFFFFFFFF);
  snprintf(ssid, sizeof(ssid), "%s%08X", WIFI_SSID_PREFIX, mac32);
  IPAddress IP(192, 168, 4, 1);
  char ipStr[16];
  snprintf(ipStr, sizeof(ipStr), "%d.%d.%d.%d", IP[0], IP[1], IP[2], IP[3]);
  int infoLen = snprintf((char *)plaintext, sizeof(plaintext),
                "%s|%s|%s", ssid, WIFI_PASSWORD, ipStr);
  uint8_t ciphertext[128];
  uint8_t tag[GCM_TAG_SIZE];
  aesGcmEncrypt(state.tempSessionKey, iv, plaintext, infoLen, NULL, 0, ciphertext, tag);
  size_t msgLen = 1 + GCM_IV_SIZE + infoLen + GCM_TAG_SIZE;
  uint8_t message[1 + GCM_IV_SIZE + 128 + GCM_TAG_SIZE];
  message[0] = MSG_TYPE_WIFI_INFO;
  memcpy(message + 1, iv, GCM_IV_SIZE);
  memcpy(message + 1 + GCM_IV_SIZE, ciphertext, infoLen);
  memcpy(message + 1 + GCM_IV_SIZE + infoLen, tag, GCM_TAG_SIZE);
  bleSendMessage(message, msgLen);
  if (pServer != nullptr) { pServer->disconnect(0); }
  BLEDevice::deinit(false);
  startWiFiServer();
  unsigned long waitStart = millis();
  while (!wifiServer->hasClient() && (millis() - waitStart) < 15000) { delay(100); }
  if (wifiServer->hasClient()) {
    serverClient = wifiServer->available();
    if (serverClient && serverClient.connected()) {
      wifiConnected = true; }}
} else {
  const size_t wifiInfoMsgLen = 1 + GCM_IV_SIZE + 128 + GCM_TAG_SIZE;
  uint8_t wifiInfoBuffer[wifiInfoMsgLen];
  int receivedLen = bleReceiveMessage(wifiInfoBuffer, wifiInfoMsgLen, 10000);
  if (receivedLen > 0) { connectToWiFiServer(wifiInfoBuffer, receivedLen); }}
```

Figure 4.17 WiFi channel switching

Figure 4.17 shows that Phase 4 switches the communication channel from BLE to

Wi-Fi so that the reconciliation phase can use more bandwidth. Wi-Fi roles are given out in the same way that BLE server client relationships are. The BLE server turns into a Wi-Fi server, and the BLE client turns into a Wi-Fi client. Before the channel switch, the server sends encrypted Wi-Fi credentials (`MSG_TYPE_WIFI_INFO, 0x09`) over BLE. After the TCP connection is made, BLE is completely shut down using `BLEDevice::deinit()` to avoid problems with Wi-Fi.

4.1.12 State Management

```
struct SchemeState {
  uint8_t tempSessionKey[AES_KEY_SIZE];
  uint8_t finalSessionKey[AES_KEY_SIZE];
  uint8_t myNonce[NONCE_SIZE];
  uint8_t peerNonce[NONCE_SIZE];
  uint8_t myMerkleRoot[HASH_SIZE];
  uint8_t peerMerkleRoot[HASH_SIZE];
  uint8_t sharedSecret[ECC_KEY_SIZE];
  unsigned long schemeStartTime;
  bool phase1Complete;
  bool phase2Complete;
  bool phase3Complete;
  bool phase4Complete;
  bool phase5Complete;
  bool myLedgerLookupResult;
  bool peerLedgerLookupResult;
  int skipToPhase;
  int verifiedCommonEntries;
  int totalUniqueEntries;
  bool isServer;
  String peerIP;
  uint64_t peerMAC;
  int restartCount; }
```

Figure 4.18 State structure

The `SchemeState` structure, shown in Figure 4.18, holds all the data structures and the step tracking values that our scheme needs, such as session keys, nonces, shared secret, Merkle tree root, phase completion booleans, reconciliation quota counter, and the network configuration. The `skipToPhase` field lets devices that mutually recognize each other skip the phases 3 and 4 to the final phase. A restart counter with a maximum

of 3 stops infinite loops to mitigate against DoS attacks.

4.1.13 Restart and Recovery Mechanism

```
void handleRestart() {
  if (state.restartCount >= MAX_RESTART_COUNT) {
    secureZero(&state, sizeof(state));
    bleBufferLen = 0;
    bleMessageReady = false;
    return; }
[...]
  bool wifiWasUsed = wifiConnected || state.phase4Complete;
  if (wifiWasUsed) {
    if (wifiConnected) {
      if (state.isServer) {
        if (serverClient)
          serverClient.stop();
        if (wifiServer) {
          wifiServer->close();
          delete wifiServer;
          wifiServer = nullptr; }
        WiFi.softAPdisconnect(true);
      } else {
        wifiClient.stop();
        WiFi.disconnect(true); }
      wifiConnected = false; }
    secureZero(&state, sizeof(state));
    bleBufferLen = 0;
    bleMessageReady = false;
    return; }
  int restartCount = state.restartCount + 1;
  secureZero(&state, sizeof(state));
  state.restartCount = restartCount;
  bleBufferLen = 0;
  bleMessageReady = false;
  runFullScheme(); }
```

Figure 4.19 Restart handler

Figure 4.19 shows that the restart mechanism makes it possible to recover from scheme steps and phase desynchronization without having to restart the devices or require outside input. If Phase 1 is successfull, the restart command is encrypted with the available key K_{temp}; if it isn't, the PSK is used. The 1-byte tag field (`0xFF`) stays

unencrypted so that messages can be sorted quickly.

4.1.14 Metrics Collection and Energy Estimation

```
struct PhaseMetrics {
  unsigned long computeTimeUs;
  unsigned long startTimeMs;
  unsigned long endTimeMs;
  uint32_t messagesSent;
  uint32_t messagesReceived;
  uint32_t bytesSent;
  uint32_t bytesReceived;
  uint16_t hashOperations;
  uint16_t hmacOperations;
  uint16_t encryptOperations;
  uint16_t decryptOperations;
  uint16_t ecdhOperations;
  uint16_t hkdfOperations;
  float energyMJ;
  uint8_t successCount;
  uint8_t failCount;
  bool completed; }
```

Figure 4.20 Phase metrics

Figure 4.20 shows that every phase keeps track of the number of cryptographic operations, such as hash, HMAC, encrypt, decrypt, ECDH, HKDF, message size statistics, and timing.

4.1.15 Secure Memory Handling

```
void secureZero(void *ptr, size_t len) {
  volatile uint8_t *p = (volatile uint8_t *)ptr;
  while (len--) { *p++ = 0; }
}
```

Figure 4.21 Secure zero function

```
secureZero(state.tempSessionKey, AES_KEY_SIZE);
secureZero(state.sharedSecret, ECC_KEY_SIZE);
secureZero(ikm, sizeof(ikm));
secureZero(myNewNonce, NONCE_SIZE);
secureZero(peerNewNonce, NONCE_SIZE);
mbedtls_mpi_free(&myKeyPair.d);
mbedtls_mpi_init(&myKeyPair.d);
```

Figure 4.22 Secure cleanup usage

Figures 4.21 and 4.22 show how to erase private cryptographic materials like private keys, session keys, nonces, and shared secrets using a `volatile` pointer to prevent a compiler optimization. This mitigates a possible memory disclosure vulnerability. Whether the authentication is successful or terminated, all session materials and private keys are erased from the device's memory.

4.1.16 Device Ledger Management

```
struct LedgerEntry {
  uint8_t idHash[HASH_SIZE];
  char friendlyName[32];
  bool isActive;
};
```

Figure 4.23 Ledger entry

```
bool addDeviceToLedger(const uint8_t *deviceHash, const char *friendlyName) {
  if (ledgerCount >= MAX_LEDGER_ENTRIES) {
    return false; }
  for (uint16_t i = 0; i < ledgerCount; i++) {
    if (memcmp(ledger[i].idHash, deviceHash, HASH_SIZE) == 0) {
      return false; }}
  memcpy(ledger[ledgerCount].idHash, deviceHash, HASH_SIZE);
  strncpy(ledger[ledgerCount].friendlyName, friendlyName, 31);
  ledger[ledgerCount].friendlyName[31] = '\0';
  ledger[ledgerCount].isActive = true;
  ledgerCount++;
  saveLedger();
  return true; }
```

Figure 4.24 Add device to ledger

```
bool lookupDeviceInLedger(const uint8_t *deviceHash) {
  for (uint16_t i = 0; i < ledgerCount; i++) {
    if (ledger[i].isActive && memcmp(ledger[i].idHash, deviceHash, HASH_SIZE) == 0) {
      return true;
    }
  }
  return false;
}
```

Figure 4.25 Lookup devices in ledger

Figures 4.23, 4.24, and 4.25 show that the device's local ledger is consist of trusted device hashed ID entries stored in ESP32's non-volatile flash. Each entry has a SHA-256 hash of the device ID, a descriptive optional name, and an activation tag. Duplicate detection helps keep duplicate entries from happening, and the `isActive` tag allows devices to temporarily revoke trust without deleting the entry.

4.2 Testing

Our scheme was first tested and simulated in Wokwi. It is a powerful, browser-based electronics simulator for prototyping IoT projects.

4.2.1 Wokwi Simulation

In contrast to conventional simulators that require heavy software installations, Wokwi runs entirely in the web browser. It allows writing code, modeling sophisticated circuits involving microcontrollers such as Arduino and ESP32, and connecting them to a wide range of circuit components, sensors, and displays.

The following Figure 4.26 is a snippet of simulated standalone code output, run on Wokwi. A more detailed output is in Appendix 3.

```
Step 1: Generating new nonces...
New nonce (mine): 97F71CCFA13701C1E722AA348953D1C4
New nonce (peer): 7F02CA6A7A549F380B95BCBF9D3D3AA1
Step 2: Encrypting nonce exchange...
Step 3: Deriving K_new...
K_new: 0BD9FB312529145F4C094A98E57B03E3...
Step 4: Sending AUTH_COMPLETE...
[OK] AUTH_COMPLETE exchanged
Total scheme time: 4571 ms
Phase 5: 31 ms | Energy: 6.01 mJ | Bytes: 88 sent, 88 recv
|====================================|
|  [OK] AUTHENTICATION COMPLETE!     |
|====================================|
```

Figure 4.26 Wokwi simulation output

4.2.2 Real World Standalone Testing

The following 4.27 is a snippet of our scheme's standalone code output run on the ESP32-WROOM-32 device. A more detailed output is in Appendix 4.

```
Step 1: Generating new nonces...
New nonce (mine): 28E060D8FE8322FE18D2963683245A31
New nonce (peer): C3167F0F2D233D9DDF582660F7C2F71A
Step 2: Encrypting nonce exchange...
Step 3: Deriving K_new...
K_new: 4BAB12BD0A996C0E9295B875FAAB5EBC...
Step 4: Sending AUTH_COMPLETE...
[OK] AUTH_COMPLETE exchanged
Total scheme time: 643 ms
Phase 5: 32 ms | Energy: 6.17 mJ | Bytes: 88 sent, 88 recv
|====================================|
|  [OK] AUTHENTICATION COMPLETE!     |
|====================================|
```

Figure 4.27 ESP32 standalone output

4.2.3 Real World Two-Device Testing

The following Figure 4.28 is a snippet of real output, run on ESP32-WROOM-32 device to test out scheme. A more detailed output is in Appendix 5.

```
Device ID: A7B3...F8C2 (truncated)
Step 1: Generating ECC key pair and nonce...
My public key: 04A8C7E2B1D9F3...
K_final: 83D7344DFB83C5503519731DB58E5865...
Step 4: Exchanging AUTH_COMPLETE...
WiFi: Sent 42 bytes
[OK] Sent AUTH_COMPLETE [STEP=08]
WiFi: Received 42 bytes
Received AUTH_COMPLETE [STEP=08]
[OK] Peer AUTH_COMPLETE verified
|==================================|
|  [OK] AUTHENTICATION COMPLETE!   |
|==================================|
Secure channel established
Total scheme time: 7911 ms
```

Figure 4.28 ESP32 two device testing output

4.3 Metrics and Analysis

In this section, we analyze our scheme test result in three different types: Wokwi simulator, standalone ESP32 test, two device ESP32 test. For data storage, each trusted device ID stored in a local ledger is approximately 72 bytes. Devices can have up to 100 entries meaning 7.2 kB. For private keys, only PSK is stored hard coded, which is 32 bytes. The device's ID is also 32 bytes. Session keys are generated at runtime, deleted after the authentication process. WiFi Credentials are partly hardcoded and partly received via BLE, not stored on device, about 100 bytes.

As we can see the averaged test results in Table 4.1, Wokwi sim results yield very closely to the two device real life test. The standalone test result is different from the other two because in standalone, we don't simulate the BLE and Wi-Fi message delays.

Table 4.1 Test average metrics

Metric type	Sim Result	Standalone Result	Two Device Result
Total Scheme Time	4506 ms	566.17 ms	5988.4 ms
Average Phase Time	901.2 ms	113.23 ms	1197.68 ms
Total Bytes Sent	590 bytes	590 bytes	3515 bytes
Total Bytes Received	1262 bytes	1262 bytes	2281 bytes
Total Bandwidth	1852 bytes	1852 bytes	5796 bytes
Total Energy	751.7 mJ	101.63 mJ	1007.49 mJ
Average per Phase	150.34 mJ	20.33 mJ	201.5 mJ

ESP32 Authentication Scheme was tested using two identical ESP32-WROOM-32 microcontrollers running the complete scheme implementation. After booting, both devices automatically initialize as BLE servers, advertising under unique device names such as ”ESP32-abc123”. Device A was chosen as the client by pressing ’A’ in the menu, which initiated BLE server scan. Device A successfully discovered Device B’s BLE signal within 5 seconds and established a BLE connection.

Then, both devices execute Phase 1 at the same time. Each device generates a new ECC public-private key pair and fresh nonce, then using PSK to calculate the HMAC-SHA256 hash using its PSK, ECC public key, nonce, then send this hash message encrypted with PSK. After both devices mutually receive and decrypting the messages, they verify both devices have the same HMAC function, calculate the ECDH shared secret, and use HKDF-SHA256 to derive the temporary session key K_{temp}.

After Phase 1 was done, both devices moved on to Phase 2 automatically. Using AES-GCM with K_{temp} (`MSG_TYPE_ID_HASH`, 0x02), each device calculate the hash of its own device ID and sent this to the other device along with the peer’s nonce, encrypted with K_{temp}. After decryption, each device uses a Bloom filter lookup and if necessary an exact ledger search to see if the peer’s hashed ID is in their local ledger. The results of the lookup were encrypted and sent to each other with a tag

(`MSG_TYPE_LOOKUP_RESULT, 0x03`). If both devices found the peer's hashed ID in their ledgers, they establish mutual trust and go directly to the final phase, else the move to phase 3.

Then in Phase 3, each device has a Merkle tree built from its local ledger entries and calculates the Merkle tree root. They send each other and compare the encrypted Merkle roots with the tag (`MSG_TYPE_MERKLE_ROOT, 0x04`). For the scheme to run all phases, the ledgers were kept not identical, so the roots didn't match. This proved that the ledgers were exactly the same and that a reconciliation phase, Phase 4, is needed.

Phase 4 starts with a communication channel switch from BLE to Wi-Fi to allow for quick and secure reconciliation. The BLE server device sends its Wi-Fi server credentials (`MSG_TYPE_WIFI_INFO, 0x09`) then starts the Wi-Fi SoftAP. Our test devices can't run BLE and Wi-Fi at the same time, so BLE must be turned off before the Wi-Fi connection can be established. The client connects to the Wi-Fi network and makes a TCP connection. After a successfull Wi-Fi connection, the two devices exchange iterative Merkle tree proofs over Wi-Fi TCP. First device sends a proof request that is the hash of its Merkle tree leaf or node with the tag`MSG_TYPE_PROOF_REQ, 0x05`, and the other device check its own Merkle tree, iterates leaf by leaf, and send the plain Merkle tree leaf proof that proves it possesses the actual leaf or node (`MSG_TYPE_PROOF_RESP, 0x06`). This goes on until either the verified common entries went over the 60% trust threshold, or 5-minute timeout limit or message quota is exceeded. If the trust threshold is met, they mutually trust each other and move on the finalize authentication. If the timeout limit or the message quota is exceeded in the reconciliation phase, they terminate the authentication process.

In Phase 5. Each device generates a new nonce (n'_A and n'_B), send it to the peer device, encrypted it with K_{temp}, along with the tag (`MSG_TYPE_NONCE, 0x07`). When each device receives the peer's nonce, they derive the new session key using $K = \text{HKDF}\big(\text{sort}(n'_A, n'_B) \parallel K_{DH}, \text{“final-session-key”}\big)$.

Finally, both devices sent each other an encrypted AUTH_COMPLETE confirmation message with the tag (`MSG_TYPE_AUTH_COMPLETE, 0x08`) using K_{new}. After the authentication session, all private information, like K_{temp}, K_{DH}, ECC public-private keys, and nonces, was safely erased from memory.

CHAPTER FIVE
SECURITY ANALYSIS AND PERFORMANCE EVALUATION

This chapter gives an in-depth look at the security and performance of our shceme. First, we address a few essential security terms; confidentiality, integrity, mutual authentication, secure session key agreement, forward/backward secrecy, and formal verification. Next, we'll discuss the types of security attacks that may occur over wireless communication; man-in-the-middle (MitM), replay, impersonation, key escrow, side-channel attacks, eavesdropping, denial-of-service, and physical tampering. To protect against these threats, we then clarify the specific countermeasures we used in our scheme. Furthermore, our approach is a lightweight design that works well on resource-constrained IoT devices. The design uses very low memory, processing power, and energy, which makes battery-powered devices last longer without making them unsafe. We utilize the Scyther formal verification tool to check the security claims of our scheme. This formal verification, which is based on the Dolev-Yao model, shows that our scheme is secure from both logical errors and complex attacks.

5.1 Security Terms

The initial step in the development of a secure IoT security framework is to determine the critical security features. In advance of focusing on particular attacks, we define our main security objectives: confidentiality, integrity, availability, mutual authentication, secure session key agreement, and forward/backward secrecy. We define and explain each term, and then we show how our scheme accomplishes that goal.

5.1.1 Confidentiality

Confidentiality keeps data safe from unauthorized access during its being sent and stored, which prevents attacks such as eavesdropping. During the first phase, devices

share public keys and nonces to set up a secure channel without revealing any secret keys. Then, using the ECDH key exchange function, they derive a temporary shared session key and use it to encrypt all remaining messages to keep the data secure. This encryption prevents attackers from intercepting message traffic, even if they can eavesdrop on the physical link. The scheme can make sure that the data stays inconceivable to any third-party attacker who doesn't possess the correct session credentials by implementing strong encryption algorithms (Wen et al., 2024).

5.1.2 Integrity

Data integrity makes sure that the information sent and received is identical, which means that no adversary corrupted it without permission during transmission. The scheme uses nonce exchanges, HMAC verifications, and public-private ECC key pairs to validate the integrity of message contents. The system can detect even single-bit changes made with MitM attacks by utilizing hash functions. Also, the Merkle tree structure makes sure that any change to a device's local ledger can be detected immediately, taking advantage of the Merkle tree's multi-level hashing structure (Öksüzer, 2022).

5.1.3 Mutual Authentication

Mutual authentication is a vital security feature in which both devices must prove their identities to the other party before sending any private information. It constructs trust between devices, so they can communicate with each other in insecure and unknown places. Both devices contribute by sending their unique identifier's hash, fresh nonces, and HMAC. This guarantees that only the correct devices are authenticated. This peer-to-peer authentication prevents impersonation attacks, where an attacker pretends to be a node to steal credentials. The addition of Merkle trees helps build trust between devices by maintaining a record of previously trusted devices and making sure that both devices settle (Mansour, 2021).

5.1.4 Secure Session Key Agreement

Secure session key agreement is the process which two devices create a one-time shared key that only they know and can use. In our scheme, the key is generated using ECDH and HKDF, so every session has a fresh, strong, and unique key (Kumar et al., 2024).

5.1.5 Forward/Backward Secrecy

In IoT environments where devices may be physically captured, forward and backward secrecy are essential for protecting data over time. Forward secrecy ensures that even if one session key is exposed, previously exchanged messages remain confidential. Backward secrecy similarly ensures that compromise of a current key does not reveal future communications. In our scheme, ECDH, HKDF, HMAC, and fresh nonces are used to support both Forward/Backward Secrecy. These design choices isolate each session, so any secuirty breach is contained in that session rather than spreading across past or future sessions (Babu et al., 2022).

5.1.6 Formal Verification

Formal verification uses formal logic and automated tools to prove that a scheme meets its security requirements and doesn't have any logical errors. This detailed process goes beyond empirical testing to give a theoretical assurance of security against specific threat models, like the Dolev–Yao model. We formally verify the scheme using BAN logic, which models the beliefs and trust relationships between communicating devices during a session. By clearly defining assumptions and message flows, we can identify potential weaknesses before implementation, such as replay attacks and key-freshness issues. We also use the Scyther formal tool for automated verification: it explores possible attack traces and checks whether the scheme's security claims hold under the modeled adversary (Öksüzer, 2022).

5.2 Security Attack Types

We have defined the core concepts, now we will briefly explain common security attacks on IoT environments and how to prevent them, based on previously discussed security terms.

5.2.1 Replay Attacks

Replay attacks occur when an adversary captures a valid message exchanged between an IoT device and another node,such as a gateway, and retransmits the message later to trigger unauthorized actions. Because the replayed message was originally legitimate, the attacker may bypass weak authentication checks without decrypting the payload. To prevent this, our scheme uses fresh nonces so each message can be checked if it belong to the current session or not, allowing duplicate transmissions to be detected and rejected. Additional defenses, such as HMAC-based one-time password mechanisms (HOTP), can further ensure that credentials change across sessions, making intercepted messages useless in future attempts (Gupta et al., 2025a).

5.2.2 Impersonation Attacks

Impersonation attacks occur when an attacker attempts to impersonate a trusted device in order to trick other devices on the network and gain unauthorized access. In these situations, attackers leverage weak authentication systems to pretend to be a trusted node, which lets them inject false information and execute malicious commands. To defend against these threats, robust mutual authentication is necessary. This implies that both devices must use cryptographic techniques to confirm and trust each other before sharing any private data. Additionally, PUF offers robust hardware-based protection by creating unique device fingerprints that cannot be duplicated or reproduced, preventing adversaries from impersonating a device. Using dynamic alias identities that update with each session can also stop opponents from

keeping track of or profiling certain devices, which makes it difficult to impersonate a certain node (Hasan et al., 2024).

5.2.3 Man-in-the-Middle Attacks

A Man-in-the-Middle (MitM) attack happens when a malicious party actively intercepts, relays, or changes messages between two devices that believe they are communicating only with each other. Adversaries exploit insecure public channels to eavesdrop on private conversations or modify messages, which could lead to unauthorized device control or data theft. To defend against MitM attacks, schemes usually use robust encryption standards like AES and ECC and impose mutual authentication to make sure that both devices are trusted entities. Also, systems can prevent attackers from getting the credentials they require to read and modify messages by adding an extra protection layer with hash functions and also by making sure that secret keys are created uniquely for each session. Furthermore, incorporating unique digital signatures like nonces and PUFs guarantees the integrity and freshness of messages, rendering it virtually impossible for a malicious party to modify the data without detection (Wen et al., 2024).

5.2.4 Key Escrow

Key escrow vulnerability occurs when a gateway server, when it is responsible for generating private keys, keeps copies of the nodes' private keys or can reproduce them. This creates a single point of failure and makes insider threats likely. This vulnerability indicates that the gateway servers are the main targets of key escrow attacks, not the resource-constrained IoT devices. In conventional cryptographic schemes, this vulnerability puts the entire system at risk, since if the gateway device is captured, attackers could decrypt all communication using that scheme. Systems can minimize the risk of key escrow attacks by not storing the full private keys of nodes. Mitigating against key escrow attacks, researchers have suggested certificateless public key cryptography and other decentralized approaches. In these

suggested methods, a node's private key is generated by combining a partial key from the gateway server with a secret value generated by the node itself, such as the output of a private hash function of its MAC address. This indicates that the gateway server doesn't and can't store the private key of the node. Using PUFs also allows devices to generate keys internally from unique physical functions without any gateway server input, which can mitigate the key escrow attack (Gupta and Varshney, 2023).

5.2.5 Denial of Service (DoS) Attacks

Denial-of-service (DoS) attacks attempt to make IoT devices or networks unusable for authenticated devices by sending excessive traffic and forcing them to process and memory overhead, which leads to resource exhaustion. Resource-constrained IoT devices are especially vulnerable to resource-exhaustion attacks, like battery-draining attacks. In these attacks, hackers prevent devices from going into low-power or energy-saving mode. To reduce the DoS attacks, schemes can adopt puzzle-like difficult computational problems that must be solved to continue communications, which might result in an attacker device process overhead. This solution can deter the attacker from launching large-scale attacks. Lightweight cryptographic operations don't consume much energy on IoT devices. Utilizing lightweight cryptographic operations to quickly detect and reject any illegal request with no processor and energy overhead. Message rate limiting and message filtering implementation are also useful solutions to mitigate DoS attacks. Developing simple but effective DoS attack recognition patterns can help detect and deter unusual traffic surges before any device failure (Gupta et al., 2025a).

5.2.6 Side-channel Attacks

Side-channel attacks use leaked information during the execution of cryptographic algorithms. This information is usually any irregularity measured outside of the normal range, like differences in timing, power usage patterns, or electromagnetic signal emissions. Adversaries can use these non-invasive, discreet methods to deduce

the scheme's framework without requiring access to internal components or damaging the hardware. For instance, consistent stream ciphers are easier to analyze for timing and power differences compared to dynamically changing algorithms. Lightweight architectures need to be enhanced to strengthen against any leaked information. Countermeasures like masking techniques and operand permutations can be used to mitigate side channel attacks (Wen et al., 2024).

5.2.7 Eavesdropping

Eavesdropping, or sniffing attack, is a type of passive attack in which an adversary intercepts messages over an insecure channel. In IoT environments, unencrypted communications can expose sensitive data such as authentication tokens, encryption keys, and device IDs. Even without modifying any message traffic, an attacker can analyze intercepted packets to analyze the authentication scheme's behavior and plan a more targeted attack later. Because this attack is passive, it is often difficult to detect through routine traffic monitoring using on device sensors. To mitigate this risk, communication must be protected with strong end-to-end encryption so that captured data remains unreadable. In addition, modern authentication designs combine cryptographic hashing with lightweight operations such as XOR to further reduce the usefulness of intercepted packets to an attacker (Wen et al., 2024).

5.2.8 Physical Tampering

Physical tampering occurs when an adversary gains a direct access to an IoT device in order to extract private and sensitive data or modify its firmware for more advance attacks. In practice, an attacker may capture an edge node device and use techniques such as differential power analysis and firmware binary reverse engineering to recover private credentials or analyze core details of cryptographic scheme design. Once decrypted, these credentials can be reused to impersonate the compromised device and support further attacks on the network. Unlike destructive attacks like DoS, physical tampering is intended to take control of a device and imitate a legitimate

node by obtaining its secret keys. The risk is highest when devices are deployed in unattended or weakly secured environments. Modern authentication schemes rely on hardware-based security primitives like a PUF; since a PUF's actions are managed by random physical properties, any attempt to explore, clone, or modify the PUF alters its functionality, rendering previously obtained credentials invalid and useless. This critically deters the attacker's efforts to exploit any captured device. Additionally, some IoT devices like ESP32, possess device flash encryption, which adds another protection layer by encrypting their firmware binaries (Gupta et al., 2025a).

5.3 Our Countermeasures Against Specific Attacks

In this section, we briefly explain how our proposed scheme mitigates against these common attacks in resource-constrained IoT systems.

5.3.1 Replay Attack

Every session includes newly generated nonces. Usage of fresh nonces (n_X) for each session prevents replay attacks. Because nonces are distinctive in each session, old nonces fail validation processes. Utilizing a session timeout T_{max} also prevents attacks based on repeated replay attacks.

5.3.2 Impersonation Attack

An attacker that doesn't possess the correct PSK can't decrypt initial messages from the start of the authentication scheme. Furthermore, it can't generate a valid $HMAC$, all of which prevent an impersonation attack. An attacker also needs to possess the correct public-private key pair (PK_X, SK_X) to generate the shared session key K_{temp}.

5.3.3 Man-in-the-Middle (MitM) Attack

MitM attacks are firstly prevented by public key encryptions and the ledger verification process. A hacker can't pretend to be a real device. The attacker must have the PSK in order to pass the $HMAC$ check. They can't change the PK or n values without making the $HMAC$ invalid, and they can't make valid HMACs for their own injected keys or nonces. After K_{temp} is set up with verified public keys, all future messages are encrypted. This stops a MitM who wasn't part of the first key exchange from listening in or changing the messages.

5.3.4 Key Escrow

The scheme eliminates the vulnerabilities related to key escrow attacks by employing ECDH to generate the shared session key by the IoT devices during the session, avoiding the dependence on a gateway server. Each device independently generates a temporary ECC key pair (PK_X, SK_X), shares its own public key and nonce (PK_X, n_X) with the other devices, receives the corresponding pair (PK_Y, n_Y) from the opposing device, and both of them compute the ECDH shared secret $K_{DH} = ECDH(SK_X, PK_Y)$ directly on the edge device, without any private keys being shared. Then the temporary shared session key $K_{new} = KDF(n'_X|n'_Y|K_{DH})$ is simultaneously generated by both devices, with nonces from both devices, ensuring that the gateway device, which distributes the PSK, cannot determine the shared session key established between devices.

5.3.5 DoS Attack Resistance

No authentication scheme can completely prevent denial-of-service attacks at the network layer. Our proposed scheme tries to mitigate by making resource-exhaustion attempts less desired by restricting resource-costly operations until a certain trust is established after Phase 1 and by enforcing a session timeout and message rate limits and quotas for each authentication session. The session timeout T_{max} prevents an

attacker from keeping the connection with the device open endlessly to prevent battery drainage attacks. Message rate limiting and quotas are other additional solutions to prevent resource exhaustion attacks. In the initialization phase, $HMAC$ verification can add considerable processing load; however, taking advantage of ESP32's hardware and firmware, we can minimize the processing overhead. If the device can't be authenticated and trusted, the scheme prevents them from exhausting the device's resources. Our scheme does not claim to mitigate all DoS vectors. Additionally, for future improvements, we can decrease the protocol-level DoS surface by ensuring that the authentication session can have a worst-case cost threshold and banning unauthenticated devices when that threshold is passed. When a device detects a malicious attack, it can add the attacker's device identifiers to a blacklist and share it with other trusted nodes and gateway server.

5.3.6 Side-Channel Attack

The scheme uses the mbedTLS library to perform cryptographic operations and the secureZero() function to securely erase all private session information from its memory. After each auth session, private information, such as temporary session keys, mutual secrets, and nonces, is replaced with zeros at the core level. This prevents timing-based attacks and cold-boot attacks. Using hardware-backed random number generation to generate nonces makes it trickier for an attacker to perform a power analysis to find predictable patterns.

5.3.7 Eavesdropping

AES-256-GCM encrypts all scheme messages, delivering discretion and reliability. (PSK) encrypts the initial messages in Phase 1 until a shared temporary session key (K_{temp}) is established, then that session key is used to encrypt the subsequent communications. The GCM authentication tag in AES-256-GCM ensures that any malicious attempt to capture the ciphertext can't change it without being caught. Temporary ECDH key exchange generated for each session provides forward and

backward secrecy. Even if K_{temp} is captured, the session keys for the past and the future are still safe because each session uses temporary session keys.

5.3.8 Physical Tampering

The scheme utilizes the ESP32's hardware and firmware security features and cryptographic verification to prevent physical tampering. The flash encryption feature makes sure that scheme firmware and private data stored in flash memory are kept encrypted with a unique 256-bit key that is built into eFuses of ESP32s. This blocks the analysis of the flash chip directly.The Secure Boot feature of ESP32 checks the firmware signature chain at the boot, guaranteeing only verified firmware runs the device. Any firmware that has been altered fails this cryptographic verification process and automatically rejected. Bloom filters and $Hash(ID)$ lookups in Merkle trees searches for the identity of a device. If an attacker alters the device's local ledger, the Merkle root comparison will no longer be valid. The two-step ledger verification, which consists of a Bloom filter check and a Merkle tree root comparison, ensures that a cloned or an altered device can't be authenticated unless it possesses the correct PSK and valid local ledger. BLE to WiFi channel switching also helps to shorten the time window for a physical proximity-based interception attack. These hardware and software methods work together coherently to create an in-depth defense for the resource-constrained IoT devices.

5.4 Lightweight Design

This section demonstrates that our scheme comprises a lightweight design by employing methods such as elliptic-curve primitives, probabilistic lightweight data structures, and communication channel switching. These methods collectively guarantee robust security while keeping computational, communication, and storage overhead within the limitations of standard IoT devices.

5.4.1 Efficient Cryptography

The scheme uses ECC-based cryptography techniques and HMAC-SHA256, which is more efficient than other cryptographic algorithms on ESP32 devices.

5.4.2 Optimized Data Structures

Bloom filters are quick and consume very low memory for ledger lookups. Merkle trees provide an efficient solution to validate the larger datasets.

5.4.3 Communication Efficiency

BLE is used for sending small-sized communications. The switch to Wi-Fi only occurs if devices proceed to the reconciliation phase. Shifting the communication channel is a necessary step to make quick communication and increase the security.

5.4.4 Computational Efficiency

The scheme prioritizes easily computable operations. HMAC verification is very quick. Ledger checking with a bloom filter is rapid. The complete Merkle reconciliation phase is implemented as a fallback option. The ECDH calculation does put a considerable processing load on the devices, but it is necessary for secure key establishment.

5.4.5 Energy Efficiency

The scheme's goal is to consume very little energy by using lightweight operations for computation and communication, which is important for battery-operated IoT devices. Using a DLT and a Bloom filter in the initial trusted device verification stage lowers energy use with minimal processor load.

5.4.6 Reduced Storage Overhead

Bloom filters help minimize devices' local ledger sizes by only storing the necessary information.

5.4.7 Hardware Acceleration

The scheme uses algorithms like ECC, AES-GCM, and SHA256 that utilize the hardware acceleration feature of ESP32 devices to speed up cryptographic operations.

5.5 Formal Security Tool Analysis

We use the Scyther formal security tool to formally verify our scheme against the threats outlined in previous sections and to validate our claims that such as mutual authentication and the secrecy of the session key are true.

Our Scyther testing environment consists of running Ubuntu 22.04 LTS on the UTM Virtual Machine App running on macOS. The Scyther code was written and tested on the scyther-linux-v1.2.1 version.

5.5.1 Scyther Tool Features

The Scyther tool is an automated formal verification tool for analyzing, verifying, and falsifying security schemes. It was developed by Cas Cremers initially around 2007-2008 (Cremers, 2008).

5.5.1.1 Verification and Falsification

The Scyther tool can verify whether a scheme fulfills its security properties or falsifies them by finding potential attacks.

5.5.1.2 Problem Identification

It is aimed to precisely and quickly identify problems in the scheme's construction through model checking.

5.5.1.3 Model Checking

It uses a custom specification language, SPDL, that allows users to easily describe cryptographic schemes. This description is then used to represent traces, verifying behavior against various attacks, and automatically analyzing the authentication scheme.

5.5.2 Security Properties

It verifies numerous security properties, often denoted to as claim events, such as:

5.5.2.1 Alive

Confirms that communication partners are active and that all events are executed in the planned sequence.

5.5.2.2 Nisynch (Non-injective Synchronization)

Ensures the receiver successfully receives all messages from the sender and complies with the scheme requirements.

5.5.2.3 Niagree (Non-injective Agreement)

Focuses on both parties consistently and accurately understanding the exchanged data to guarantee mutual authentication.

5.5.2.4 Weak-agree

Provides protection against identity spoofing attacks.

5.5.2.5 Secret (Confidentiality)

Confirms that certain information, like keys, nonces, and random values, remains unknown to an attacker.

5.5.2.6 SKR (Session Key Reveal/Secrecy of Session Key)

Relates to the secrecy and establishment of session keys, incorporating protection against known session-key attacks and forward secrecy.

5.5.2.7 Support for Sessions

It can handle an unlimited number of sessions and run multiple schemes in parallel. Scyther delivers attack graphs for counterexamples and a summary report. Each attack graph shows a single claim. If it is indicated ”SAFE” and ”No attacks,” it ensures that the scheme is secure and that both parties have validated the security claims. Scyther uses the Dolev-Yao threat model. In this model, an enemy has complete control over the entire network, and it can intercept, delete, insert, eavesdrop, and replay messages. It can also capture devices to retrieve private data.

5.5.3 Usage and Workflow

5.5.3.1 Protocol Description Language (SPDL)

Scyther’s own SPDL language is used to define security protocols. The roles defined in the protocol are client and server, which are a set of events that show how information is sent and received.

5.5.3.2 Claim Events

Claim events are used to model the security properties. Scyther can either verify claims that the user makes or automatically generate and verify them.

5.5.3.3 Graphical User Interface and Command-Line

Scyther has a graphical user interface, a command-line tool, and additionally a Python scripting interface.

5.5.3.4 Analysis Modes

There are three main ways to use Scyther: to check a claim, to make automatic claims, and for characterization.

5.5.4 Limitations of SPDL

Scyther is a powerful and well-known security tool in cybersecurity research for methodically verifying the security claims of encryption schemes. It provides security assurances and points out any major weaknesses. When we tried to define our scheme in Scyther's SPDL code, we ran into the following problems and limitations:

- Scyther doesn't support ECDH key agreement, so ephemeral public keys (PK_A^{temp} and PK_B^{temp}) are modeled as new nonces instead of elliptic curve points. Instead of modeling the authentic ECDH computation, in the process of shared secret key derivation of K_{temp}, it uses these symbolic values along with the PSK.
- Key Derivation Functions (KDF) are not natively supported; thus, they are represented as uninterpreted hash functions.

- For HMAC operations, we use ”hashfunction HMAC” instead of the actual keyed-hash message authentication construction function.
- Scyther doesn’t support conditional logic or compare variables. It can’t run the logic, such as ”check if ‘RootA == RootB‘ then skip Phase 4,” so all scheme phases are executed one after the other, even if the Merkle roots are the exact same.
- The reconciliation phase is modeled as a single request-response exchange, leaving out iterative rounds, request quotas, acceptance thresholds, and dynamic termination conditions.
- Merkle tree operations, like tree building,proof generating and checking, are represented as opaque nonce values ($Root_A$, $Root_B$, $Proof_A$) without modeling how trees, hash chains, or proof paths actually work.
- The operations of the bloom filter and the logic for reconciliation steps are also abstracted; only the final roots and proofs are shown symbolically.
- Scyther cannot model and verify timing restrictions and timeout mechanisms, and it also cannot detect any timing-based attacks.
- There is no ability of modeling of communication layer differences, such as BLE and Wi-Fi, signal strength, message latency, or connection reliability.
- Entity identifiers are modeled solely as hashed values ($Hash(ID(A))$, $Hash(ID(B))$) employing an uninterpreted hash function, rather than as actual identity data and certificates containing the authentication logic of the scheme.

5.5.5 Our Scheme's SPDL Representation

We now divide our scheme's Scyther code into three snippets: declarations, role definitions, and claims. The complete SPDL code can be found in Appendix 1.

```
usertype SessionKey;
usertype String;
hashfunction HMAC; // models HMAC(PSK, ...) in a -DolevYao way
hashfunction KDF; // models KDF(...) as a one-way derivation
hashfunction HID; // models Hash(ID_X) (hashed identifier)
const M1,M2,M3,M4,M5,M6,M7,M8,M9,M10,M11,M12: String;
const tKtemp,tKnew: String;
```

Figure 5.1 Declarations

The code in Figure 5.1 starts by defining the custom types and cryptographic functions that the scheme's Scyther modeling requires. "usertype SessionKey and usertype String" define abstract data types for session keys and string labels. In the Dolev-Yao model, there are three hash functions: HMAC models message authentication codes, KDF represents key derivation operations, and H(ID) handles identifier hashing. All three are treated as uninterpreted irreversible functions. Twelve message tag constants (M1 through M12) are used as domain separation labels to stop type confusion attacks, in which an attacker might send a message from one phase to another. When deriving the temporary and final session keys, two more constants (tKtemp and tKnew) separate the KDF domains. This makes sure that these keys are cryptographically independent.

```
role A {
fresh PKAtemp: Nonce;
fresh nA: Nonce;
var PKBtemp: Nonce;
var nB: Nonce;
fresh RootA: Nonce;
var RootB: Nonce;
fresh ReqA: Nonce;
var ProofB: Nonce;
fresh nA2: Nonce;
var nB2: Nonce;

macro mac1 = HMAC(k(A,B), M1, A, B, PKAtemp, nA);
macro mac2 = HMAC(k(A,B), M2, A, B, PKAtemp, nA, PKBtemp, nB);
macro Ktemp = KDF(k(A,B), tKtemp, A, B, PKAtemp, nA, PKBtemp, nB);
macro Knew = KDF(Ktemp, tKnew, A, B, nA2, nB2);

send_1(A,B, { M1, A, B, PKAtemp, nA, mac1 }k(A,B) );
recv_2(B,A, { M2, A, B, PKBtemp, nB, mac2 }k(A,B) );
send_3(A,B, { M3, HID(A), nB }Ktemp );
recv_4(B,A, { M4, HID(B), nA }Ktemp );
send_5(A,B, { M5, RootA }Ktemp );
recv_6(B,A, { M6, RootB }Ktemp );
send_7(A,B, { M7, ReqA }Ktemp );
recv_8(B,A, { M8, ReqA, ProofB }Ktemp );
send_9(A,B, { M9, nA2 }Ktemp );
recv_10(B,A, { M10, nB2 }Ktemp );
send_11(A,B, { M11, nB2 }Knew );
recv_12(B,A, { M12, nA2 }Knew ); }
```

Figure 5.2 Role definitions

In Figure 5.2; Role A (Initiator): Role A starts by generating new values required for Phase 1, PKAtemp and nA, representing a temporary contribution and a nonce. In the real scheme, PKAtemp would be a public key for ECC. The var keyword creates variables for the values that A will get from B. The scheme uses macro definitions to connect complex expressions. For example, mac1 and mac2 use the pre-shared key k(A,B) to verify the handshake messages, and mac2 connects both parties' contributions. The Ktemp macro creates the temporary session key with the PSK and all the necessary values. The Knew macro gets the final key from Ktemp and the Phase 5 nonces.

The message flow sends and receives operations in a sequence of numbers. Messages 1–2 (Phase 1) send temporary contributions that are encrypted with k(A, B) with MAC tags. Messages 3 and 4 (Phase 2) send hashed identities back and forth with nonce confirmation under Ktemp. Messages 5 and 6 (Phase 3) send Merkle tree roots back and forth, and messages 7 and 8 (Phase 4) model the reconciliation request-response mechanism. Finally, messages 9–12 (Phase 5) set up Knew by exchanging nonces and mutual key confirmation.

Role B (Responder): Role B has the same structure as A, but it switches the send and receive operations. B gets A's first contribution in `recv_1`, then generates its own fresh values (PKBtemp, nB) before sending back in `send_2`. The macro definitions are the same to make sure that both roles get the same keys from the same inputs. If there is any difference here, the verification will fail. B follows the same five-phase structure and responds each of A's messages correctly.

```
protocol LedgerAuth(A,B) {
role A {
claim_A1(A, Secret, Ktemp);
claim_A2(A, Secret, Knew);
claim_A3(A, Nisynch);
claim_A4(A, Niagree); }
role B {
claim_B1(B, Secret, Ktemp);
claim_B2(B, Secret, Knew);
claim_B3(B, Nisynch);
claim_B4(B, Niagree); } }
```

Figure 5.3 Claims

As shown in Figure 5.3, both roles asserts four security claims after the scheme is finished. Secret claims (`claim_A1, claim_A2, claim_B1, claim_B2`) assert that the attacker can't discover Ktemp or Knew, which confirms that the information is private. Claims A3 and B3 from Nisynch check that when a role concludes, there is a corresponding execution of the partner role with identical action following the same sequence. Niagree (`claim_A4 and claim_B4`) makes sure that both sides agree on all the values that were exchanged during the scheme execution. Together, these

claims ensure mutual authentication, key confidentiality, and session integrity. They also validate that participants agree on the scheme's execution and the generated session keys.

Lastly, the verified claim proof screenshots are shown below in Figure 5.4 and Figure 5.5.

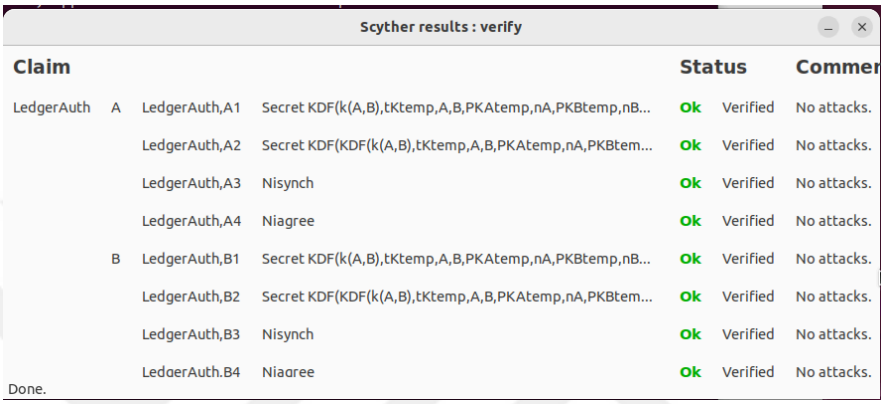


Figure 5.4 Verified scheme

Claim				Status		Comments
LedgerAuth	A	LedgerAuth,A5	Secret nA2	Ok		No attacks within bounds.
		LedgerAuth,A6	Secret ReqA	Ok		No attacks within bounds.
		LedgerAuth,A7	Secret RootA	Ok		No attacks within bounds.
		LedgerAuth,A8	Secret nA	Ok		No attacks within bounds.
		LedgerAuth,A9	Secret PKAtemp	Ok		No attacks within bounds.
		LedgerAuth,A10	Secret nB2	Ok		No attacks within bounds.
		LedgerAuth,A11	Secret ProofB	Ok		No attacks within bounds.
		LedgerAuth,A12	Secret RootB	Ok		No attacks within bounds.
		LedgerAuth,A13	Secret nB	Ok		No attacks within bounds.
		LedgerAuth,A14	Secret PKBtemp	Ok		No attacks within bounds.
		LedgerAuth,A15	Alive	Ok	Verified	No attacks.
		LedgerAuth,A16	Weakagree	Ok	Verified	No attacks.
		LedgerAuth,A17	Niagree	Ok	Verified	No attacks.
		LedgerAuth,A18	Nisynch	Ok	Verified	No attacks.
	B	LedgerAuth,B5	Secret nB2	Ok		No attacks within bounds.
		LedgerAuth,B6	Secret ProofB	Ok		No attacks within bounds.
		LedgerAuth,B7	Secret RootB	Ok		No attacks within bounds.
		LedgerAuth,B8	Secret nB	Ok		No attacks within bounds.
		LedgerAuth,B9	Secret PKBtemp	Ok		No attacks within bounds.
		LedgerAuth,B10	Secret nA2	Ok		No attacks within bounds.
		LedgerAuth,B11	Secret ReqA	Ok		No attacks within bounds.
		LedgerAuth,B12	Secret RootA	Ok		No attacks within bounds.
		LedgerAuth,B13	Secret nA	Ok		No attacks within bounds.
		LedgerAuth,B14	Secret PKAtemp	Ok		No attacks within bounds.
		LedgerAuth,B15	Alive	Ok	Verified	No attacks.
		LedgerAuth,B16	Weakagree	Ok	Verified	No attacks.
		LedgerAuth,B17	Niagree	Ok	Verified	No attacks.

Done.

Figure 5.5 Verified automatic claims

CHAPTER SIX
CONCLUSION AND FUTURE WORK

This thesis introduces a robust and lightweight mutual authentication scheme with session key agreement, designed for deployment in peer-to-peer resource-limited IoT devices. Its efficiency is validated on the ESP32-WROOM-32 device test runs. The proposed scheme removes the need for a gateway server during the authentication process by using distributed ledgers for a trusted mutual base. Distributed ledgers increase resilience against physical tampering and side-channel attacks. A global timeout for the authentication session shields against replay attacks. The proposed scheme combines AES-GCM, HMAC, HKDF, ECDH, fresh nonces, DLT, Merkle Trees, Bloom filters, and session timeout checks.

Additionally, the scheme's security claims are validated with BAN logic proof and the Scyther formal tool. It has met with the important Scyther requirements of aliveness, secrecy, and non-injective synchronization. It also mitigated the following MitM, replay, key escrow, DoS, eavesdropping, and impersonation attacks. Using both BLE and Wi-Fi in the scheme demonstrates that it could work on a multi-node network. We have tested the scheme in real life, and it has operated as expected without any complications. Our test runs showed our scheme implementation has a near-perfect 99% authentication success rate, proving that it is suitable for real-time IoT applications that require a mutual authentication feature.

In future work, the scheme may be enhanced with grouping proofs to enable a secure group formation and perform a cooperative authentication with multiple IoT devices. A group of adjacent devices could work together to create a cryptographic proof that they are all nearby and collectively authenticate a new device, rather than authenticating one at a time. This feature would make resistance against relay-based impersonation stronger and allow for scalable trust control for large IoT deployments. Combining grouping proofs with a distributed ledger trust model could make it efficient to notice and update the status of ledger entries while keeping the computation overhead low. Continuous vulnerability testing and developing and

embedding defensive algorithms and techniques to defend against malicious attacks should also be considered. Lastly, focusing on the growing threat of machine learning-based attacks on IoT devices should be considered. To overcome these attacks, the scheme may integrate dynamically changing PUFs without additional computing overhead.

REFERENCES

Abdellatif, K. M., Hériveaux, O., and Thillard, A. (2023). Unlimited results: Breaking firmware encryption of ESP32-v3. Cryptology ePrint Archive, Paper 2023/090.

Alghamdi, A. M. (2025). Design and analysis of lightweight and robust authentication protocol for securing the resource constrained iiot environment. *PloS one*, 20(2):e0318064.

Ali, S. and Anwer, F. (2025). An iot-enabled cloud computing model for authentication and data confidentiality using lightweight cryptography. *Arabian Journal for Science and Engineering*, pages 1–23.

Almuhaideb, A. and Al-Khulaifi, D. (2024). An efficient authentication and key agreement scheme for the internet of underwater things (iout) environment. *IEEE Access*, PP:1–1.

Altaş, H. (2022). Internet of things security with blockchain. Master's thesis, Dokuz Eylul University, Graduate School of Natural and Applied Sciences.

Arshad, Q.-u.-A., Khan, W. Z., Azam, F., Khan, M. K., Yu, H., and Zikria, Y. B. (2023). Blockchain-based decentralized trust management in iot: systems, requirements and challenges. *Complex & Intelligent Systems*, 9(6):6155–6176.

Asif, M., Abrar, M., Salam, A., Amin, F., Ullah, F., Shah, S., and AlSalman, H. (2025). Intelligent two-phase dual authentication framework for internet of medical things. *Scientific Reports*, 15.

Babu, P. R., Reddy, A. G., Palaniswamy, B., and Kommuri, S. K. (2022). Ev-auth: Lightweight authentication protocol suite for dynamic charging system of electric vehicles with seamless handover. *IEEE Transactions on Intelligent Vehicles*, 7:734–747.

Burgos, J. B. and Pustišek, M. (2024). Decentralized iot data authentication with signature aggregation. *Sensors*, 24.

Cremers, C. J. (2008). The scyther tool: Verification, falsification, and analysis of security protocols: Tool paper. In *International conference on computer aided verification*, pages 414–418. Springer.

El Hanine, M., El-Yahyaoui, A., and Es-Sadaoui, R. (2025). Design and assessment of lightweight cryptographic algorithms on esp32 and stm32 for iod security. *Egyptian Informatics Journal*, 32:100818.

Espressif Systems (2025). *ESP32-WROOM-32 Datasheet*. Espressif Systems. Retrieved from https://documentation.espressif.com.

Fei, H., Millwood, O., Prosanta, G., Miskelly, J., and Sikdar, B. (2024). Phenoauth: A novel puf-phenotype-based authentication protocol for iot devices.

Google DeepMind (2025). Introducing nano banana pro. `https://deepmind.google/models/nano-banana-pro`. Accessed: 2026-01-14.

Gupta, A., Surpur, A., Das, B. P., and Manhas, S. (2025a). A unified approach to a secure and lightweight mutual authentication protocol using pre-characterized cots sram ics for iot applications. *ACM Trans. Embed. Comput. Syst.*, 24(5).

Gupta, C. and Varshney, G. (2023). A lightweight and secure puf-based authentication and key-exchange protocol for iot devices.

Gupta, K., Kumar, V., Prakash, R., and Pal, O. (2025b). Slmas: a secure and lightweight mutual authentication scheme for telecare medical information system based on ecc. *The Journal of Supercomputing*, 81(5):639.

Hasan, M. K., Weichen, Z., Safie, N., Ahmed, F. R. A., and Ghazal, T. M. (2024). A survey on key agreement and authentication protocol for internet of things application. *IEEE Access*, 12:61642–61666.

Hu, D., Xiong, X., Zhang, H., Zhang, J., Zheng, J., and Xiao, L. (2024). A novel blockchain-based anonymous mutual authentication method for iot security. In *Proceedings - 2024 IEEE International Conference on Smart Internet of Things, SmartIoT 2024*, pages 123–128. Institute of Electrical and Electronics Engineers Inc.

Jain, U., Tripathi, A., Kumar, S., and Kumar, G. (2025). Simple, secure and lightweight authentication protocol with session-key generation for iiot device in iiot networks. *Microsystem Technologies*, 31(2):299–311.

Kaygısız, S. Y. and Dalkılıç, G. (2025). Merkle tree-enhanced ecdh mutual authentication for secure and lightweight iot communication. In *2025 9th International Symposium on Innovative Approaches in Smart Technologies (ISAS)*, pages 1–8. IEEE.

Kumar, V., Hassan, F., Nishad, A. K., Prasad, R., Gupta, P., Amir, M., and Ahmad, F. (2024). Design of secure and efficient framework for vehicular digital twin networks using ecc. *IEEE Access*, 12:194352–194366.

Lo, N.-W., Huang, J.-J., and Yang, T.-C. (2025). A lightweight mutual authentication mechanism for applications utilizing low-power iot devices. *Electronics*, 14(21):4178.

Mansour, S. (2021). *Lightweight mutual authentication and privacy preservation schemes for IOT systems.* PhD thesis, University of Louisville.

Mershad, K. and Cheikhrouhou, O. (2023). Lightweight blockchain solutions: Taxonomy, research progress, and comprehensive review.

Mo, S., Feng, W., Huang, M., Feng, S., Wang, Z., and Li, Y. (2025). Two-factor authentication for intellectual property transactions based on improved zero-knowledge proof. *Scientific Reports*, 15.

Naser, H. Y., Mattar, A. K., Saare, M. A., Almaiah, M. A., and Shehab, R. (2025). A comparison of lightweight cryptographic protocols for energy-efficient and sustainable iomt authentication. *Engineering, Technology Applied Science Research*, 15(4):25746–25756.

Nkenyereye, L., Thakare, A., Khataniar, P., Imandi, R., and BN, P. K. (2025). Lightweight authentication protocol for smart grids: An energy-efficient authentication scheme for resource-limited smart meters. *Mathematics*, 13(4):580.

Öksüzer, Ş. (2022). Privacy in blockchain based internet of things network. Master's thesis, Dokuz Eylül University, Graduate School of Natural and Applied Sciences.

Qiu, S., Li, J., Di, X., Li, X., Wu, Y., and Ibrahim, M. (2025). Lightweight mutual authentication scheme based on blockchain for internet of medical things. *IEEE Internet of Things Journal*, 12:8848–8861.

Rzepecki, W. and Ryba, P. (2019). Iotsp: Thread mesh vs other widely used wireless protocols–comparison and use cases study. In *2019 7th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 291–295. IEEE.

Satpathy, S. P., Mohanty, S., and Pradhan, M. (2025). A sustainable mutual authentication protocol for iot-fog-cloud environment. *Peer-to-Peer Networking and Applications*, 18(1):35.

Şeker, Ö. (2024). *Application of Lightweight Cryptography Techniques in Internet of Things Scenarios*. Phd thesis, Dokuz Eüylül University, Graduate School of Natural and Applied Sciences.

Şeker, Ö., Dalkılıç, G., and Çabuk, U. C. (2023). Maras: Mutual authentication and role-based authorization scheme for lightweight internet of things applications. *Sensors*, 23(12):5674.

Soni, M. and Singh, D. K. (2022). Laka: Lightweight authentication and key agreement protocol for internet of things based wireless body area network. *Wireless Personal Communications*, 127:1067–1084.

Syam, S. R., Nedungadi, N., and Sankaran, S. (2023). A novel lightweight group authentication protocol for internet of things. In *ISED 2023 - International Conference on Intelligent Systems and Embedded Design*. Institute of Electrical and Electronics Engineers Inc.

Tanveer, M., El-Latif, A. A. A., Khan, A. U., Ahmad, M., and Ateya, A. A. (2024). Leaf-iiot: Lightweight and efficient authentication framework for the industrial internet of things. *IEEE Access*, 12:31771–31787.

Ukpebor, A., Addy, J., Ali, K., and Humos, A. A.-E. (2023). Secure end-to-end communications with lightweight cryptographic algorithm.

Wen, F. E., Ohno, H., and Sampalli, S. (2024). A systematic review of iot security: Research potential, challenges, and future directions. *ACM Computing Surveys*, 56.

Yu, S., Park, K., and Park, Y. (2025). A machine learning attack-resistant puf-based robust and efficient mutual authentication scheme in fog-enabled iot environments. *IEEE Internet of Things Journal*, 12:20652–20669.

APPENDICES

Appendix 1: Scyther Complete Code

```
usertype SessionKey;
usertype String;
hashfunction HMAC;
hashfunction KDF;
hashfunction HID;

const M1,M2,M3,M4,M5,M6,M7,M8,M9,M10,M11,M12: String;
const tKtemp,tKnew: String;

protocol LedgerAuth(A,B)
{
  role A
  {
    fresh PKAtemp: Nonce;
    fresh nA: Nonce;
    var PKBtemp: Nonce;
    var nB: Nonce;
    fresh RootA: Nonce;
    var RootB: Nonce;
    fresh ReqA: Nonce;
    var ProofB: Nonce;
    fresh nA2: Nonce;
    var nB2: Nonce;

    macro mac1  = HMAC(k(A,B), M1, A, B, PKAtemp, nA);
    macro mac2  = HMAC(k(A,B), M2, A, B, PKAtemp, nA, PKBtemp, nB);
    macro Ktemp = KDF(k(A,B), tKtemp, A, B, PKAtemp, nA, PKBtemp, nB);
    macro Knew  = KDF(Ktemp, tKnew, A, B, nA2, nB2);

    send_1(A,B, { M1, A, B, PKAtemp, nA, mac1 }k(A,B) );
    recv_2(B,A, { M2, A, B, PKBtemp, nB, mac2 }k(A,B) );
    send_3(A,B, { M3, HID(A), nB }Ktemp );
    recv_4(B,A, { M4, HID(B), nA }Ktemp );
    send_5(A,B, { M5, RootA }Ktemp );
    recv_6(B,A, { M6, RootB }Ktemp );
    send_7(A,B, { M7, ReqA }Ktemp );
    recv_8(B,A, { M8, ReqA, ProofB }Ktemp );
    send_9(A,B,  { M9,  nA2 }Ktemp );
    recv_10(B,A, { M10, nB2 }Ktemp );
    send_11(A,B, { M11, nB2 }Knew );
    recv_12(B,A, { M12, nA2 }Knew );

    claim_A1(A, Secret, Ktemp);
```

Appendix 1: Continues

```
    claim_A2(A, Secret, Knew);
    claim_A3(A, Nisynch);
    claim_A4(A, Niagree);
  }

  role B
  {
    var PKAtemp: Nonce;
    var nA: Nonce;
    fresh PKBtemp: Nonce;
    fresh nB: Nonce;
    var RootA: Nonce;
    fresh RootB: Nonce;
    var ReqA: Nonce;
    fresh ProofB: Nonce;
    var nA2: Nonce;
    fresh nB2: Nonce;

    macro mac1  = HMAC(k(A,B), M1, A, B, PKAtemp, nA);
    macro mac2  = HMAC(k(A,B), M2, A, B, PKAtemp, nA, PKBtemp, nB);
    macro Ktemp = KDF(k(A,B), tKtemp, A, B, PKAtemp, nA, PKBtemp, nB);
    macro Knew  = KDF(Ktemp, tKnew, A, B, nA2, nB2);

    recv_1(A,B, { M1, A, B, PKAtemp, nA, mac1 }k(A,B) );
    send_2(B,A, { M2, A, B, PKBtemp, nB, mac2 }k(A,B) );
    recv_3(A,B, { M3, HID(A), nB }Ktemp );
    send_4(B,A, { M4, HID(B), nA }Ktemp );
    recv_5(A,B, { M5, RootA }Ktemp );
    send_6(B,A, { M6, RootB }Ktemp );
    recv_7(A,B, { M7, ReqA }Ktemp );
    send_8(B,A, { M8, ReqA, ProofB }Ktemp );
    recv_9(A,B,  { M9,  nA2 }Ktemp );
    send_10(B,A, { M10, nB2 }Ktemp );
    recv_11(A,B, { M11, nB2 }Knew );
    send_12(B,A, { M12, nA2 }Knew );

    claim_B1(B, Secret, Ktemp);
    claim_B2(B, Secret, Knew);
    claim_B3(B, Nisynch);
    claim_B4(B, Niagree);
  }
}
```

Figure A.1 Scyther Complete Code

Appendix 2: ESP32-WROOM-32 Detailed Code Snippets

```
[...]
#define NONCE_SIZE 16
#define HASH_SIZE 32
#define PSK_SIZE 32
#define ECC_KEY_SIZE 32
#define PUBLIC_KEY_SIZE 65
#define AES_KEY_SIZE 32
#define GCM_TAG_SIZE 16
#define GCM_IV_SIZE 12
#define T_MAX 600000
#define PHASE4_TIMEOUT 300000
#define MAX_PROOF_REQUESTS 10
#define TRUST_THRESHOLD 60
#define MSG_TYPE_HANDSHAKE 0x01
#define MSG_TYPE_ID_HASH 0x02
#define MSG_TYPE_LOOKUP_RESULT 0x03
#define MSG_TYPE_MERKLE_ROOT 0x04
#define MSG_TYPE_PROOF_REQ 0x05
#define MSG_TYPE_PROOF_RESP 0x06
#define MSG_TYPE_NONCE 0x07
#define MSG_TYPE_AUTH_COMPLETE 0x08
#define MSG_TYPE_WIFI_INFO 0x09
#define MSG_TYPE_RESTART 0xFF
#define MAX_RESTART_COUNT 3
#define BLOOM_SIZE_BYTES 256
#define BLOOM_SIZE_BITS (BLOOM_SIZE_BYTES * 8)
#define BLOOM_HASH_COUNT 3
#define MAX_MERKLE_LEAVES 128
#define MAX_MERKLE_PROOF_LEN 8
#define MAX_LEDGER_ENTRIES 100
#define LEDGER_NAMESPACE "deviceLedger"

struct LedgerEntry {
  uint8_t idHash[HASH_SIZE];
  char friendlyName[32];
  bool isActive;
  uint32_t addedTimestamp;
};

LedgerEntry ledger[MAX_LEDGER_ENTRIES];
uint16_t ledgerCount = 0;
Preferences preferences;
struct ECCKeyPair {
  mbedtls_ecp_group grp;
```

Appendix 2: Continues

```
  mbedtls_mpi d;
  mbedtls_ecp_point Q;
};

struct PhaseMetrics {
  unsigned long computeTimeUs;
  unsigned long startTimeMs;
  unsigned long endTimeMs;
  uint32_t messagesSent;
  uint32_t messagesReceived;
  uint32_t bytesSent;
  uint32_t bytesReceived;
  uint16_t hashOperations;
  uint16_t hmacOperations;
  uint16_t encryptOperations;
  uint16_t decryptOperations;
  uint16_t ecdhOperations;
  uint16_t hkdfOperations;
  float energyMJ;
  uint8_t successCount;
  uint8_t failCount;
  bool completed;
};

struct ProtocolMetrics {
  PhaseMetrics phase[5];
  unsigned long totalTimeMs;
  uint32_t totalBytesSent;
  uint32_t totalBytesReceived;
  float totalEnergyMJ;
  uint16_t totalCryptoOps;
  uint32_t peakHeapUsage;
  uint32_t freeHeapAtStart;
  uint32_t freeHeapAtEnd;
  bool usedFastPath;
  uint8_t phasesCompleted;
  uint8_t phasesSkipped;
};

struct ProtocolState {
  uint8_t tempSessionKey[AES_KEY_SIZE];
  uint8_t finalSessionKey[AES_KEY_SIZE];
  uint8_t myNonce[NONCE_SIZE];
  uint8_t peerNonce[NONCE_SIZE];
  uint8_t myMerkleRoot[HASH_SIZE];
```

Appendix 2: Continues

```
  uint8_t peerMerkleRoot[HASH_SIZE];
  uint8_t sharedSecret[ECC_KEY_SIZE];
  unsigned long protocolStartTime;
  bool phase1Complete;
  bool phase2Complete;
  bool phase3Complete;
  bool phase4Complete;
  bool phase5Complete;
  bool myLedgerLookupResult;
  bool peerLedgerLookupResult;
  int skipToPhase;
  int verifiedCommonEntries;
  int totalUniqueEntries;
  bool isServer;
  String peerIP;
  uint64_t peerMAC;
  int restartCount;
};

ProtocolMetrics metrics;
PhaseMetrics phase1Metrics = {0};
PhaseMetrics phase2Metrics = {0};
PhaseMetrics phase3Metrics = {0};
PhaseMetrics phase4Metrics = {0};
PhaseMetrics phase5Metrics = {0};
PhaseMetrics cryptoTestMetrics = {0};
ProtocolState state;
ECCKeyPair myKeyPair;
bool verboseOutput = true;
mbedtls_entropy_context entropy;
mbedtls_ctr_drbg_context ctr_drbg;
bool rngInitialized = false;
BLEServer *pServer = NULL;
BLECharacteristic *pCharacteristic = NULL;
BLERemoteCharacteristic *pRemoteCharacteristic = NULL;
bool deviceConnected = false;
bool bleServerMode = false;
#define BLE_BUFFER_SIZE 256
uint8_t bleBuffer[BLE_BUFFER_SIZE];
size_t bleBufferLen = 0;
volatile bool bleMessageReady = false;
WiFiServer *wifiServer = NULL;
WiFiClient wifiClient;
WiFiClient serverClient;
bool wifiConnected = false;
```

Appendix 2: Continues

```
bool pendingProtocolRun = false;
bool protocolJustCompleted = false;
uint8_t bloomFilter[BLOOM_SIZE_BYTES];
struct MerkleTree {
  uint8_t leaves[MAX_MERKLE_LEAVES][HASH_SIZE];
  uint8_t nodes[MAX_MERKLE_LEAVES * 2][HASH_SIZE];
  uint16_t leafCount;
  uint8_t root[HASH_SIZE];
  bool isBuilt;
};

MerkleTree distributedLedgerTree;
MerkleTree localLedgerTree;

void generateNonce(uint8_t *nonce, size_t len) {
  for (size_t i = 0; i < len; i++) {
    nonce[i] = esp_random() & 0xFF;
  }
}

int initializeRNG() {
  if (rngInitialized)
    return 0;
  mbedtls_entropy_init(&entropy);
  mbedtls_ctr_drbg_init(&ctr_drbg);
  const char *pers = "esp32_protocol";
  int ret = mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy,
                                  (const unsigned char *)pers, strlen(pers));
  if (ret == 0) {
    rngInitialized = true;
  }
  return ret;
}

void initMetrics() {
  memset(&metrics, 0, sizeof(ProtocolMetrics));
  metrics.freeHeapAtStart = ESP.getFreeHeap();
}

void startPhaseMetrics(int phaseIndex) {
  PhaseMetrics *pm = &metrics.phase[phaseIndex];
  memset(pm, 0, sizeof(PhaseMetrics));
  pm->startTimeMs = millis();
}

```

Appendix 2: Continues

```
void endPhaseMetrics(int phaseIndex, bool success) {
  PhaseMetrics *pm = &metrics.phase[phaseIndex];
  pm->endTimeMs = millis();
  pm->computeTimeUs = (pm->endTimeMs - pm->startTimeMs) * 1000;
  pm->completed = success;
  if (success) pm->successCount++;
  else pm->failCount++;
  float basePowerMW = 165.0;
  float cryptoPowerMW = 99.0;
  float durationMs = pm->endTimeMs - pm->startTimeMs;
  float cryptoTimeMs = (pm->ecdhOperations * 50.0) + (pm->hkdfOperations * 5.0)
                       + ((pm->hashOperations + pm->hmacOperations) * 0.5) +
                      ((pm->encryptOperations + pm->decryptOperations) * 1.0);

  pm->energyMJ = (durationMs * basePowerMW / 1000.0) +
                 (cryptoTimeMs * cryptoPowerMW / 1000.0);
}

void aggregateMetrics() {
  metrics.totalTimeMs = 0;
  metrics.totalBytesSent = 0;
  metrics.totalBytesReceived = 0;
  metrics.totalEnergyMJ = 0;
  metrics.totalCryptoOps = 0;
  metrics.phasesCompleted = 0;
  metrics.phasesSkipped = 0;
  for (int i = 0; i < 5; i++) {
    PhaseMetrics *pm = &metrics.phase[i];
    if (pm->completed) {
      metrics.phasesCompleted++;
      metrics.totalTimeMs += (pm->endTimeMs - pm->startTimeMs);
      metrics.totalBytesSent += pm->bytesSent;
      metrics.totalBytesReceived += pm->bytesReceived;
      metrics.totalEnergyMJ += pm->energyMJ;
      metrics.totalCryptoOps += pm->hashOperations + pm->hmacOperations +
                                pm->encryptOperations + pm->decryptOperations +
                                pm->ecdhOperations + pm->hkdfOperations;
    } else if (state.skipToPhase == 5 && (i == 2 || i == 3)) {
      metrics.phasesSkipped++;
    } }
  metrics.freeHeapAtEnd = ESP.getFreeHeap();
  if (metrics.freeHeapAtStart > metrics.freeHeapAtEnd) {
    metrics.peakHeapUsage = metrics.freeHeapAtStart - metrics.freeHeapAtEnd;
  } else {
    metrics.peakHeapUsage = 0;
```

Appendix 2: Continues

```
  }
  metrics.usedFastPath = (state.skipToPhase == 5);
}

void printFullMetrics() {
  Serial.println("\n|================================|");
  Serial.println("|      COMPREHENSIVE METRICS         |");
  Serial.println("|================================|");
  aggregateMetrics();
  const char *phaseNames[] = {"Phase 1: Handshake", "Phase 2: Ledger Check",
                              "Phase 3: Merkle Root", "Phase 4: Reconcile",
                              "Phase 5: New Key"};
  for (int i = 0; i < 5; i++) {
    if (metrics.phase[i].completed || metrics.phase[i].startTimeMs > 0) {
      printPhaseMetrics(i, phaseNames[i]);
    }
  }
  Serial.println("\n|================================|");
  Serial.println("|        AGGREGATE SUMMARY           |");
  Serial.println("|================================|");
  Serial.printf("Phases Completed: %u/5\n", metrics.phasesCompleted);
  Serial.printf("Phases Skipped: %u (fast path: %s)\n", metrics.phasesSkipped,
                metrics.usedFastPath ? "YES" : "NO");
  Serial.printf("\  n Timing  \n");
  Serial.printf("Total Scheme Time: %lu ms\n", metrics.totalTimeMs);
  Serial.printf("Average Phase Time: %.1f ms\n",
                metrics.phasesCompleted > 0
                    ? (float)metrics.totalTimeMs / metrics.phasesCompleted
                    : 0);

  Serial.printf("\  n Communication  \n");
  Serial.printf("Total Bytes Sent: %lu bytes\n", metrics.totalBytesSent);
  Serial.printf("Total Bytes Received: %lu bytes\n",
                metrics.totalBytesReceived);
  Serial.printf("Total Bandwidth: %lu bytes\n",
                metrics.totalBytesSent + metrics.totalBytesReceived);
  Serial.printf("\  n Cryptographic Operations  \n");
  Serial.printf("Total Crypto Operations: %u\n", metrics.totalCryptoOps);
  Serial.printf("\  n Energy Consumption  \n");
  Serial.printf("Total Energy: %.3f mJ\n", metrics.totalEnergyMJ);
  Serial.printf("Average per Phase: %.3f mJ\n",
                metrics.phasesCompleted > 0
                    ? metrics.totalEnergyMJ / metrics.phasesCompleted
                    : 0);
  Serial.printf("\  n Memory Usage  \n");
```

Appendix 2: Continues

```
  Serial.printf("Heap at Start: %lu bytes\n", metrics.freeHeapAtStart);
  Serial.printf("Heap at End: %lu bytes\n", metrics.freeHeapAtEnd);
  Serial.printf("Peak Usage: %lu bytes\n", metrics.peakHeapUsage);
}

void initKeyPair(ECCKeyPair *kp) {
  mbedtls_ecp_group_init(&kp->grp);
  mbedtls_mpi_init(&kp->d);
  mbedtls_ecp_point_init(&kp->Q);
}
void freeKeyPair(ECCKeyPair *kp) {
  mbedtls_ecp_group_free(&kp->grp);
  mbedtls_mpi_free(&kp->d);
  mbedtls_ecp_point_free(&kp->Q);
}

int calculateHMAC(const uint8_t *key, size_t keyLen, const uint8_t *data,
                  size_t dataLen, uint8_t *output) {
  mbedtls_md_context_t ctx;
  mbedtls_md_type_t md_type = MBEDTLS_MD_SHA256;
  mbedtls_md_init(&ctx);
  int ret = mbedtls_md_setup(&ctx, mbedtls_md_info_from_type(md_type), 1);
  if (ret == 0)
    ret = mbedtls_md_hmac_starts(&ctx, key, keyLen);
  if (ret == 0)
    ret = mbedtls_md_hmac_update(&ctx, data, dataLen);
  if (ret == 0)
    ret = mbedtls_md_hmac_finish(&ctx, output);
  mbedtls_md_free(&ctx);
  return ret;
}

bool verifyHMAC(const uint8_t *key, size_t keyLen, const uint8_t *data,
                size_t dataLen, const uint8_t *expectedHMAC) {
  uint8_t calculated[HASH_SIZE];
  if (calculateHMAC(key, keyLen, data, dataLen, calculated) != 0) {
    return false;
  }
  return (memcmp(calculated, expectedHMAC, HASH_SIZE) == 0);
}

void calculateSHA256(const uint8_t *data, size_t dataLen, uint8_t *output) {
  mbedtls_sha256_context ctx;
  mbedtls_sha256_init(&ctx);
  mbedtls_sha256_starts(&ctx, 0);
```

Appendix 2: Continues

```
315   mbedtls_sha256_update(&ctx, data, dataLen);
316   mbedtls_sha256_finish(&ctx, output);
317   mbedtls_sha256_free(&ctx);
318 }
319
320 int generateKeyPair(ECCKeyPair *kp) {
321   int ret = mbedtls_ecp_group_load(&kp->grp, MBEDTLS_ECP_DP_SECP256R1);
322   if (ret != 0)
323     return ret;
324   ret = mbedtls_ecp_gen_keypair(&kp->grp, &kp->d, &kp->Q,
325                                 mbedtls_ctr_drbg_random, &ctr_drbg);
326   return ret;
327 }
328
329 int serializePublicKey(const mbedtls_ecp_group *grp, const mbedtls_ecp_point *Q, uint8_t *
    ↪ output, size_t *len) {
330   return mbedtls_ecp_point_write_binary(grp, Q, MBEDTLS_ECP_PF_UNCOMPRESSED,
331                                         len, output, PUBLIC_KEY_SIZE);
332 }
333
334 int computeECDH(ECCKeyPair *myKey, const uint8_t *peerPubKey, size_t peerKeyLen, uint8_t *
    ↪ sharedSecret, size_t *secretLen) {
335   int ret;
336   mbedtls_ecp_point peerQ;
337   mbedtls_mpi z;
338   mbedtls_ecp_point_init(&peerQ);
339   mbedtls_mpi_init(&z);
340   ret = mbedtls_ecp_point_read_binary(&myKey->grp, &peerQ, peerPubKey,
341                                       peerKeyLen);
342   if (ret != 0) {
343     mbedtls_ecp_point_free(&peerQ);
344     mbedtls_mpi_free(&z);
345     return ret;
346   }
347   ret = mbedtls_ecdh_compute_shared(&myKey->grp, &z, &peerQ, &myKey->d,
348                                     mbedtls_ctr_drbg_random, &ctr_drbg);
349   if (ret == 0) {
350     *secretLen = mbedtls_mpi_size(&z);
351     ret = mbedtls_mpi_write_binary(&z, sharedSecret, *secretLen);
352   }
353   mbedtls_mpi_free(&z);
354   mbedtls_ecp_point_free(&peerQ);
355   return ret;
356 }
357
```

Appendix 2: Continues

```
int deriveKey(const uint8_t *ikm, size_t ikmLen, const uint8_t *salt,
              size_t saltLen, const uint8_t *info, size_t infoLen, uint8_t *okm,
              size_t okmLen) {
  const mbedtls_md_info_t *md_info = mbedtls_md_info_from_type(MBEDTLS_MD_SHA256);
  return mbedtls_hkdf(md_info, salt, saltLen, ikm, ikmLen, info, infoLen, okm,
                      okmLen);
}

int aesGcmEncrypt(const uint8_t *key, const uint8_t *iv,
                  const uint8_t *plaintext, size_t ptLen, const uint8_t *aad,
                  size_t aadLen, uint8_t *ciphertext, uint8_t *tag) {
  mbedtls_gcm_context ctx;
  mbedtls_gcm_init(&ctx);
  int ret = mbedtls_gcm_setkey(&ctx, MBEDTLS_CIPHER_ID_AES, key, AES_KEY_SIZE * 8);
  if (ret == 0) {
    ret = mbedtls_gcm_crypt_and_tag(&ctx, MBEDTLS_GCM_ENCRYPT, ptLen, iv,
                                    GCM_IV_SIZE, aad, aadLen, plaintext,
                                    ciphertext, GCM_TAG_SIZE, tag);
  }
  mbedtls_gcm_free(&ctx);
  return ret;
}

int aesGcmDecrypt(const uint8_t *key, const uint8_t *iv,
                  const uint8_t *ciphertext, size_t ctLen, const uint8_t *aad,
                  size_t aadLen, const uint8_t *tag, uint8_t *plaintext) {
  mbedtls_gcm_context ctx;
  mbedtls_gcm_init(&ctx);
  int ret = mbedtls_gcm_setkey(&ctx, MBEDTLS_CIPHER_ID_AES, key, AES_KEY_SIZE * 8);
  if (ret == 0) {
    ret = mbedtls_gcm_auth_decrypt(&ctx, ctLen, iv, GCM_IV_SIZE, aad, aadLen,  tag,
  ↪ GCM_TAG_SIZE, ciphertext, plaintext);
  }
  mbedtls_gcm_free(&ctx);
  return ret;
}

void secureZero(void *ptr, size_t len) {
  volatile uint8_t *p = (volatile uint8_t *)ptr;
  while (len--) {
    *p++ = 0;
  }
}

void bloomFilterInit() { memset(bloomFilter, 0, BLOOM_SIZE_BYTES); }
```

Appendix 2: Continues

```
402
403 void bloomFilterAdd(const uint8_t *hash) {
404   uint16_t positions[BLOOM_HASH_COUNT];
405   bloomFilterGetPositions(hash, positions);
406   for (int i = 0; i < BLOOM_HASH_COUNT; i++) {
407     bloomFilter[positions[i] / 8] |= (1 << (positions[i] % 8));
408   }
409 }
410
411 bool bloomFilterCheck(const uint8_t *hash) {
412   uint16_t positions[BLOOM_HASH_COUNT];
413   bloomFilterGetPositions(hash, positions);
414   for (int i = 0; i < BLOOM_HASH_COUNT; i++) {
415     if ((bloomFilter[positions[i] / 8] & (1 << (positions[i] % 8))) == 0) {
416       return false;
417     }
418   }
419   return true;
420 }
421
422 void bloomFilterRebuildFromLedger() {
423   bloomFilterInit();
424   for (uint16_t i = 0; i < ledgerCount; i++) {
425     if (ledger[i].isActive) {
426       bloomFilterAdd(ledger[i].idHash);
427     }
428   }
429 }
430
431 void merkleTreeInit(MerkleTree *tree) {
432   memset(tree, 0, sizeof(MerkleTree));
433   tree->leafCount = 0;
434   tree->isBuilt = false;
435 }
436
437 void merkleTreeAddLeaf(MerkleTree *tree, const uint8_t *hash) {
438   if (tree->leafCount >= MAX_MERKLE_LEAVES) {
439     return;
440   }
441   memcpy(tree->leaves[tree->leafCount], hash, HASH_SIZE);
442   tree->leafCount++;
443   tree->isBuilt = false;
444 }
445
446 void merkleTreeBuild(MerkleTree *tree) {
```

Appendix 2: Continues

```
  if (tree->leafCount == 0) {
    memset(tree->root, 0, HASH_SIZE);
    tree->isBuilt = true;
    return;
  }
  uint16_t nodeCount = tree->leafCount;
  for (uint16_t i = 0; i < nodeCount; i++) {
    memcpy(tree->nodes[i], tree->leaves[i], HASH_SIZE);
  }
  uint16_t levelStart = 0;
  uint16_t nextLevelStart = nodeCount;
  while (nodeCount > 1) {
    uint16_t newNodeCount = 0;
    for (uint16_t i = 0; i < nodeCount; i += 2) {
      uint8_t combined[HASH_SIZE * 2];
      memcpy(combined, tree->nodes[levelStart + i], HASH_SIZE);

      if (i + 1 < nodeCount) {
        memcpy(combined + HASH_SIZE, tree->nodes[levelStart + i + 1],
               HASH_SIZE);
      } else {
        memcpy(combined + HASH_SIZE, tree->nodes[levelStart + i], HASH_SIZE);
      }
      calculateSHA256(combined, HASH_SIZE * 2,
                      tree->nodes[nextLevelStart + newNodeCount]);
      newNodeCount++;
    }
    levelStart = nextLevelStart;
    nextLevelStart += newNodeCount;
    nodeCount = newNodeCount;
  }
  memcpy(tree->root, tree->nodes[levelStart], HASH_SIZE);
  tree->isBuilt = true;
}

bool merkleTreeGenerateProof(MerkleTree *tree, uint16_t leafIndex,
                             uint8_t proof[][HASH_SIZE], int *proofLen) {
  if (!tree->isBuilt || leafIndex >= tree->leafCount) {
    *proofLen = 0;
    return false;
  }
  *proofLen = 0;
  uint16_t nodeCount = tree->leafCount;
  uint16_t levelStart = 0;
  uint16_t currentIndex = leafIndex;
```

Appendix 2: Continues

```
  while (nodeCount > 1) {
    uint16_t siblingIndex;
    if (currentIndex % 2 == 0) {
      siblingIndex = (currentIndex + 1 < nodeCount) ? currentIndex + 1 : currentIndex;
    } else {
      siblingIndex = currentIndex - 1;
    }
    memcpy(proof[*proofLen], tree->nodes[levelStart + siblingIndex], HASH_SIZE);
    (*proofLen)++;
    currentIndex = currentIndex / 2;
    levelStart += nodeCount;
    nodeCount = (nodeCount + 1) / 2;
  }
  return true;
}

bool merkleTreeVerifyProof(const uint8_t *root, const uint8_t *leaf,
                           uint8_t proof[][HASH_SIZE], int proofLen,
                           uint16_t leafIndex) {
  uint8_t current[HASH_SIZE];
  memcpy(current, leaf, HASH_SIZE);
  uint16_t index = leafIndex;
  for (int i = 0; i < proofLen; i++) {
    uint8_t combined[HASH_SIZE * 2];
    if (index % 2 == 0) {
      memcpy(combined, current, HASH_SIZE);
      memcpy(combined + HASH_SIZE, proof[i], HASH_SIZE);
    } else {
      memcpy(combined, proof[i], HASH_SIZE);
      memcpy(combined + HASH_SIZE, current, HASH_SIZE);
    }
    calculateSHA256(combined, HASH_SIZE * 2, current);
    index = index / 2;
  }
  return memcmp(current, root, HASH_SIZE) == 0;
}

void merkleTreeBuildFromLedger(MerkleTree *tree) {
  merkleTreeInit(tree);
  for (uint16_t i = 0; i < ledgerCount; i++) {
    if (ledger[i].isActive) {
      merkleTreeAddLeaf(tree, ledger[i].idHash);
    } }
  merkleTreeBuild(tree);
}
```

Appendix 2: Continues

```
int merkleTreeFindLeaf(MerkleTree *tree, const uint8_t *hash) {
  for (uint16_t i = 0; i < tree->leafCount; i++) {
    if (memcmp(tree->leaves[i], hash, HASH_SIZE) == 0) {
      return i;
    }}
  return -1;
}

void initBLEServer() {
  String deviceName = "ESP32-" + String((uint32_t)ESP.getEfuseMac(), HEX);
  BLEDevice::init(deviceName.c_str());
  BLEDevice::setMTU(185);
  pServer = BLEDevice::createServer();
  pServer->setCallbacks(new ServerCallbacks());
  BLEService *pService = pServer->createService(SERVICE_UUID);
  pCharacteristic = pService->createCharacteristic(
      CHARACTERISTIC_UUID, BLECharacteristic::PROPERTY_READ |
                               BLECharacteristic::PROPERTY_WRITE |
                               BLECharacteristic::PROPERTY_NOTIFY);

  pCharacteristic->setCallbacks(new CharacteristicCallbacks());
  pCharacteristic->addDescriptor(new BLE2902());
  pService->start();
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  pAdvertising->addServiceUUID(SERVICE_UUID);
  pAdvertising->setScanResponse(true);
  pAdvertising->setMinPreferred(0x06);
  pAdvertising->setMinPreferred(0x12);
  BLEDevice::startAdvertising();
  Serial.println("[OK] BLE Server started");
  Serial.printf("Device name: %s\n", deviceName.c_str());
  bleServerMode = true;
}
void bleSendMessage(const uint8_t *data, size_t len) {
  if (pCharacteristic && deviceConnected && bleServerMode) {
    pCharacteristic->setValue((uint8_t *)data, len);
    pCharacteristic->notify();
    if (verboseOutput) {
      Serial.printf("BLE: Sent %d bytes (server)\n", len);
    }
  }
  else if (pRemoteCharacteristic && deviceConnected && !bleServerMode) {
    pRemoteCharacteristic->writeValue((uint8_t *)data, len);
    if (verboseOutput) {
      Serial.printf("BLE: Sent %d bytes (client)\n", len);
```

Appendix 2: Continues

```
    }
  }
}

void sendRestartCommand() {
  Serial.println("[!] Sending RESTART command to peer...");
  uint8_t *key = state.phase1Complete ? state.tempSessionKey : const_cast<uint8_t *>(PSK);
  uint8_t plaintext[8] = {MSG_TYPE_RESTART, 'R', 'E', 'S', 'T', 'A', 'R', 'T'};
  uint8_t iv[GCM_IV_SIZE];
  generateNonce(iv, GCM_IV_SIZE);
  uint8_t ciphertext[sizeof(plaintext)];
  uint8_t tag[GCM_TAG_SIZE];
  aesGcmEncrypt(key, iv, plaintext, sizeof(plaintext), NULL, 0, ciphertext,
                tag);
  uint8_t message[1 + GCM_IV_SIZE + sizeof(ciphertext) + GCM_TAG_SIZE];
  message[0] = MSG_TYPE_RESTART;
  memcpy(message + 1, iv, GCM_IV_SIZE);
  memcpy(message + 1 + GCM_IV_SIZE, ciphertext, sizeof(ciphertext));
  memcpy(message + 1 + GCM_IV_SIZE + sizeof(ciphertext), tag, GCM_TAG_SIZE);
  bleSendMessage(message, sizeof(message));
}

void handleProtocolRestart() {
  if (state.restartCount >= MAX_RESTART_COUNT) {
    Serial.println("[X] Too many restarts - aborting protocol");
    secureZero(&state, sizeof(state));
    bleBufferLen = 0;
    bleMessageReady = false;
    return; }
  Serial.printf("[!] Scheme restart (%d/%d)\n", state.restartCount + 1,MAX_RESTART_COUNT);
  bool wifiWasUsed = wifiConnected || state.phase4Complete;
  if (wifiWasUsed) {
    if (wifiConnected) {
      if (state.isServer) {
        if (serverClient)
          serverClient.stop();
        if (wifiServer) {
          wifiServer->close();
          delete wifiServer;
          wifiServer = nullptr; }
        WiFi.softAPdisconnect(true);
      } else {
        wifiClient.stop();
        WiFi.disconnect(true);
      }
```

Appendix 2: Continues

```
627       wifiConnected = false;
628     }
629     secureZero(&state, sizeof(state));
630     bleBufferLen = 0;
631     bleMessageReady = false;
632     Serial.println(
633         "[!] WiFi was used - returning to menu. Please restart manually.");
634     return;
635   }
636   int restartCount = state.restartCount + 1;
637   secureZero(&state, sizeof(state));
638   state.restartCount = restartCount;
639   bleBufferLen = 0;
640   bleMessageReady = false;
641   delay(500);
642   runFullProtocol();
643 }
644 [...]
645 void scanAndConnectBLE() {
646   if (bleServerMode) {
647     Serial.println("Stopping BLE Server to become Client...");
648     BLEDevice::getAdvertising()->stop();
649     bleServerMode = false;
650     delay(100); }
651   Serial.println("\n--- Scanning for BLE devices ---");
652   BLEScan *pBLEScan = BLEDevice::getScan();
653   pBLEScan->setActiveScan(true);
654   BLEScanResults *foundDevices = pBLEScan->start(5, false);
655   Serial.printf("Found %d devices\n", foundDevices->getCount());
656   for (int i = 0; i < foundDevices->getCount(); i++) {
657     BLEAdvertisedDevice device = foundDevices->getDevice(i);
658     if (device.haveServiceUUID() &&
659         device.isAdvertisingService(BLEUUID(SERVICE_UUID))) {
660       Serial.printf("Found target device: %s\n",
661                     device.getAddress().toString().c_str());
662       BLEClient *pClient = BLEDevice::createClient();
663       if (pClient->connect(&device)) {
664         Serial.println("[OK] Connected to BLE server");
665         deviceConnected = true;
666         pClient->setMTU(185);
667         Serial.printf("MTU negotiated: %d\n", pClient->getMTU());
668         BLERemoteService *pRemoteService = pClient->getService(SERVICE_UUID);
669         if (pRemoteService) {
670           pRemoteCharacteristic =
671               pRemoteService->getCharacteristic(CHARACTERISTIC_UUID);
```

Appendix 2: Continues

```
672           if (pRemoteCharacteristic) {
673             Serial.println("[OK] Found characteristic");
674             if (pRemoteCharacteristic->canNotify()) {
675               pRemoteCharacteristic->registerForNotify(
676                   [](BLERemoteCharacteristic *pChar, uint8_t *pData,
677                      size_t length, bool isNotify) {
678                     if (length > 0 && length <= BLE_BUFFER_SIZE) {
679                       memcpy(bleBuffer, pData, length);
680                       bleBufferLen = length;
681                       bleMessageReady = true;
682                     }
683                   });
684             }
685             Serial.println("\n*** Starting scheme automatically ***");
686             delay(500);
687             runFullProtocol();
688             return; }}}}}
689   Serial.println("No compatible devices found");
690 }
691
692 void startWiFiServer() {
693   Serial.println("--- Starting WiFi Server (Phase 4) ---");
694   WiFi.mode(WIFI_AP);
695   delay(100);
696   char ssid[64];
697   uint64_t mac = ESP.getEfuseMac();
698   uint32_t mac32 = (mac & 0xFFFFFFFF);
699   snprintf(ssid, sizeof(ssid), "%s%08X", WIFI_SSID_PREFIX, mac32);
700   Serial.printf("[DEBUG] SSID: %s\n", ssid);
701   Serial.printf("[DEBUG] Free heap before SoftAP: %u bytes\n",
702                 ESP.getFreeHeap());
703   if (wifiServer != NULL) {
704     Serial.println("[DEBUG] Cleaning up existing WiFi server instance");
705     wifiServer->stop();
706     delete wifiServer;
707     wifiServer = NULL;
708     delay(100);
709   }
710   Serial.println("[DEBUG] Starting SoftAP...");
711   unsigned long t1 = millis();
712   if (!WiFi.softAP(ssid, WIFI_PASSWORD, WIFI_CHANNEL)) {
713     Serial.println("[X] SoftAP failed - authentication will fail");
714     state.phase4Complete = false;
715     endPhaseMetrics(3, false);
716     return;
```

Appendix 2: Continues

```
  }
  Serial.printf("[DEBUG] SoftAP started in %lu ms\n", millis() - t1);
  delay(100);
  IPAddress IP = WiFi.softAPIP();
  Serial.printf("AP started - IP: %s\n", IP.toString().c_str());
  Serial.println("[DEBUG] Creating WiFiServer instance...");
  wifiServer = new WiFiServer(SERVER_PORT);
  if (wifiServer == NULL) {
    Serial.println(
        "[X] Failed to allocate WiFiServer - authentication will fail");
    state.phase4Complete = false;
    endPhaseMetrics(3, false);
    return;
  }
  wifiServer->begin();
  Serial.println("[OK] WiFi Server listening");
  Serial.printf("WiFi Server ready on port %d\n", SERVER_PORT);
}

void connectToWiFiServer(const uint8_t *wifiInfo, size_t len) {
  Serial.printf("[DEBUG] connectToWiFiServer: received %d bytes\n", len);
  Serial.printf("[DEBUG] Step tag: 0x%02X (expected 0x%02X)\n", wifiInfo[0],
                MSG_TYPE_WIFI_INFO);
  if (wifiInfo[0] != MSG_TYPE_WIFI_INFO) {
    Serial.printf("[X] Wrong step tag: expected 0x%02X, got 0x%02X\n",
                  MSG_TYPE_WIFI_INFO, wifiInfo[0]);
    state.phase4Complete = false;
    return;
  }
  uint8_t iv[GCM_IV_SIZE];
  uint8_t ciphertext[128];
  uint8_t tag[GCM_TAG_SIZE];
  uint8_t plaintext[128];
  size_t payloadLen = len - 1;
  size_t cipherLen = payloadLen - GCM_IV_SIZE - GCM_TAG_SIZE;
  Serial.printf("[DEBUG] payloadLen=%d, cipherLen=%d\n", payloadLen, cipherLen);
  Serial.printf("[DEBUG] IV offset=1, cipher offset=%d, tag offset=%d\n",
                1 + GCM_IV_SIZE, (int)(len - GCM_TAG_SIZE));
  memcpy(iv, wifiInfo + 1, GCM_IV_SIZE);
  memcpy(ciphertext, wifiInfo + 1 + GCM_IV_SIZE, cipherLen);
  memcpy(tag, wifiInfo + len - GCM_TAG_SIZE, GCM_TAG_SIZE);
  Serial.printf("[DEBUG] Attempting AES-GCM decrypt with %d bytes ciphertext\n",
                cipherLen);
  int ret = aesGcmDecrypt(state.tempSessionKey, iv, ciphertext, cipherLen, NULL,
                          0, tag, plaintext);
```

Appendix 2: Continues

```
762   PhaseMetrics *pm = &metrics.phase[3];
763   pm->decryptOperations++;
764   Serial.printf("[DEBUG] Decryption result: %d (0=success)\n", ret);
765   if (ret != 0) {
766     Serial.println(
767         "[X] Failed to decrypt WiFi info - authentication will fail");
768     Serial.printf("[DEBUG] This usually means: wrong key, corrupted data, or "
769                   "tag mismatch\n");
770     state.phase4Complete = false;
771     return;
772   }
773   Serial.printf("[DEBUG] Decrypted WiFi info: %s\n", (char *)plaintext);
774   char *token = strtok((char *)plaintext, "|");
775   if (token == NULL) {
776     Serial.println("[X] Invalid WiFi info format - authentication will fail");
777     state.phase4Complete = false;
778     return;
779   }
780   String ssid = String(token);
781   token = strtok(NULL, "|");
782   if (token == NULL) {
783     Serial.println("[X] Invalid WiFi info format");
784     state.skipToPhase = 5;
785     state.phase4Complete = true;
786     return;
787   }
788   String password = String(token);
789   token = strtok(NULL, "|");
790   if (token == NULL) {
791     Serial.println("[X] Invalid WiFi info format");
792     state.skipToPhase = 5;
793     state.phase4Complete = true;
794     return;
795   }
796   String serverIP = String(token);
797   Serial.printf("Connecting to WiFi: %s (server: %s)\n", ssid.c_str(),
798                 serverIP.c_str());
799   BLEDevice::deinit(false);
800   Serial.println("[DEBUG] BLE stopped");
801   WiFi.disconnect(true);
802   WiFi.mode(WIFI_STA);
803   delay(100);
804   Serial.println("[DEBUG] Starting WiFi.begin()...");
805   WiFi.begin(ssid.c_str(), password.c_str());
806   int attempts = 0;
```

Appendix 2: Continues

```
807   const int maxAttempts = 40;
808   while (WiFi.status() != WL_CONNECTED && attempts < maxAttempts) {
809     delay(500);
810     int status = WiFi.status();
811     if (attempts % 5 == 0) {
812       Serial.printf("\n[DEBUG] WiFi status: %d (", status);
813       switch (status) {
814       case WL_IDLE_STATUS:
815         Serial.print("IDLE");
816         break;
817       case WL_NO_SSID_AVAIL:
818         Serial.print("NO_SSID_AVAIL");
819         break;
820       case WL_SCAN_COMPLETED:
821         Serial.print("SCAN_COMPLETED");
822         break;
823       case WL_CONNECTED:
824         Serial.print("CONNECTED");
825         break;
826       case WL_CONNECT_FAILED:
827         Serial.print("CONNECT_FAILED");
828         break;
829       case WL_CONNECTION_LOST:
830         Serial.print("CONNECTION_LOST");
831         break;
832       case WL_DISCONNECTED:
833         Serial.print("DISCONNECTED");
834         break;
835       default:
836         Serial.print("UNKNOWN");
837         break;
838       }
839       Serial.print(") ");
840     }
841     Serial.print(".");
842     attempts++;
843   }
844   Serial.printf("\n[DEBUG] Final WiFi status: %d after %d attempts\n",
845                 WiFi.status(), attempts);
846   if (WiFi.status() == WL_CONNECTED) {
847     Serial.println("[OK] WiFi connected");
848     Serial.printf("IP: %s\n", WiFi.localIP().toString().c_str());
849     Serial.printf("[DEBUG] Connecting TCP to %s:%d\n", serverIP.c_str(),
850                   SERVER_PORT);
851     wifiClient.connect(serverIP.c_str(), SERVER_PORT);
```

Appendix 2: Continues

```
    if (wifiClient.connected()) {
      Serial.println("[OK] Connected to server");
      wifiConnected = true;
      state.isServer = false;
      state.peerIP = serverIP;
    } else {
      Serial.println(
          "[X] TCP connection to server failed - authentication will fail");
      state.phase4Complete = false;
    }
  } else {
    Serial.println("[X] WiFi connection failed - authentication will fail");
    Serial.printf("[DEBUG] Check: Is server SoftAP running? SSID: %s\n",
                  ssid.c_str());
    state.phase4Complete = false;
  }
}
[...]
void viewLedger() {
  if (ledgerCount == 0) {
    Serial.println("Ledger is empty");
    return;
  }
  Serial.printf("Total entries: %d/%d\n\n", ledgerCount, MAX_LEDGER_ENTRIES);

  for (uint16_t i = 0; i < ledgerCount; i++) {
    Serial.printf("--- Entry %d ---\n", i);
    Serial.printf("Name: %s\n", ledger[i].friendlyName);
    Serial.printf("Status: %s\n", ledger[i].isActive ? "ACTIVE" : "INACTIVE");
    Serial.print("ID Hash: ");
    for (int j = 0; j < 16; j++) {
      if (ledger[i].idHash[j] < 0x10)
        Serial.print("0");
      Serial.print(ledger[i].idHash[j], HEX);
    }
    Serial.printf("Added: %lu seconds ago\n\n",
                  (millis() / 1000) - ledger[i].addedTimestamp);
  }
}

bool lookupDeviceInLedger(const uint8_t *deviceHash) {
  for (uint16_t i = 0; i < ledgerCount; i++) {
    if (ledger[i].isActive &&
        memcmp(ledger[i].idHash, deviceHash, HASH_SIZE) == 0) {
      return true;
```

Appendix 2: Continues

```
    }
  }
  return false;
}
[...]
void runFullProtocol() {
  Serial.println("\n|================================|");
  Serial.println("|  RUNNING FULL SCHEME FLOW      |");
  Serial.println("|================================|");
  memset(&state, 0, sizeof(ProtocolState));
  initMetrics();
  bloomFilterRebuildFromLedger();
  merkleTreeBuildFromLedger(&distributedLedgerTree);
  merkleTreeBuildFromLedger(&localLedgerTree);
  runPhase1_Handshake();
  if (!state.phase1Complete) {
    Serial.println("\n[X] Scheme failed at Phase 1");
    return;
  }
  delay(500);
  runPhase2_LedgerVerification();
  if (!state.phase2Complete) {
    Serial.println("\n[X] Scheme failed at Phase 2");
    return;
  }
  if (state.skipToPhase == 5) {
    Serial.println("\n* Fast path: Skipping to Phase 5 (mutual recognition)");
    state.phase3Complete = true;
    state.phase4Complete = true;
  } else {
    delay(500);
    runPhase3_MerkleRootExchange();
    if (!state.phase3Complete) {
      Serial.println("\n[X] Scheme failed at Phase 3");
      return;
    }
    if (state.skipToPhase == 5) {
      Serial.println("\n* Fast path: Skipping to Phase 5 (identical ledgers)");
      state.phase4Complete = true;
    } else {
      delay(500);
      runPhase4_Reconciliation();
      if (!state.phase4Complete) {
        Serial.println("\n[X] Scheme failed at Phase 4");
        return;
```

Appendix 2: Continues

```
    }
  }
}
runPhase5_FinalKeyDerivation();
Serial.println("\n|================================|");
Serial.println("|      SCHEME SUMMARY            |");
Serial.println("|================================|");
Serial.printf("Phase 1 (Handshake): %s\n", state.phase1Complete ? "[OK]" : "[X]");
Serial.printf("Phase 2 (Ledger): %s\n",state.phase2Complete ? "[OK]" : "[X]");
Serial.printf("Phase 3 (Merkle): %s%s\n", state.phase3Complete ? "[OK]" : "[X]
               ", (state.skipToPhase == 5 && state.phase2Complete && !state.
↪ phase3Complete) ? " (skipped)" : "");
Serial.printf("Phase 4 (Reconciliation): %s%s\n", state.phase4Complete ? "[OK]
             " : "[X]", (state.skipToPhase == 5) ? " (skipped)" : "");
Serial.printf("Phase 5 (New Key): %s\n", state.phase5Complete ? "[OK]" : "[X]");
Serial.println("\nMode: TWO-DEVICE");
bool allSuccess = state.phase1Complete && state.phase2Complete &&
                  state.phase3Complete && state.phase4Complete &&
                  state.phase5Complete;
if (allSuccess) {
  Serial.println("\n[OK][OK][OK] FULL SCHEME SUCCESS [OK][OK][OK]");
} else {
  Serial.println("\n[X] Scheme incomplete");
}
printFullMetrics();
if (wifiConnected) {
  Serial.println("\n[INFO] Cleaning up WiFi connection...");
  if (state.isServer) {
    if (serverClient)
      serverClient.stop();
    if (wifiServer) {
      wifiServer->close();
      delete wifiServer;
      wifiServer = nullptr;
    }
    WiFi.softAPdisconnect(true);
  } else {
    wifiClient.stop();
    WiFi.disconnect(true);
  }
  wifiConnected = false;
}
bool wasSuccess = allSuccess;
secureZero(&state, sizeof(state));
bleBufferLen = 0;
```

Appendix 2: Continues

```
  bleMessageReady = false;
  pendingProtocolRun = false;
  protocolJustCompleted = true;
  if (wasSuccess) {
    Serial.println(
        "\n[INFO] Scheme completed successfully. Returning to menu.");
  } else {
    Serial.println("\n[INFO] Scheme failed. Returning to menu.");
  }
  displayMenu();
}
void viewAllMetrics() { printFullMetrics(); }
void resetMetrics() {
  initMetrics();
  memset(&cryptoTestMetrics, 0, sizeof(PhaseMetrics));
  memset(&state, 0, sizeof(ProtocolState));
  Serial.println("\n[OK] All metrics reset");
}

void displayMenu() {
  Serial.println("\n--- Scheme Execution ---");
  Serial.println("1. Run Phase 1 (Handshake)");
  Serial.println("2. Run Phase 2 (Ledger Verification)");
  Serial.println("3. Run Phase 3 (Merkle Root Exchange)");
  Serial.println("4. Phase 4 (Reconciliation)");
  Serial.println("5. Run Phase 5 (New Key Derivation)");
  Serial.println("6. Run FULL Scheme flow");
  Serial.println("\n--- Ledger Management ---");
  Serial.println("L. Ledger management menu");
  Serial.println("V. Quick view ledger");
  Serial.println("\n--- Two-Device Communication ---");
  Serial.println("9. Start BLE server");
  Serial.println("A. Scan and connect BLE");
  Serial.println("\n--- Settings & Info ---");
  Serial.println("7. Toggle verbose output");
  Serial.println("8. View all metrics");
  Serial.println("0. Reset metrics");
  Serial.printf("Mode: TWO-DEVICE | Verbose: %s | Ledger: %d\n",
                verboseOutput ? "ON" : "OFF", ledgerCount);
  Serial.printf("BLE: %s | WiFi: %s\n",
                deviceConnected ? "CONNECTED" : "DISCONNECTED",
                wifiConnected ? "CONNECTED" : "DISCONNECTED");
  Serial.print("Enter choice: ");
}

```

Appendix 2: Continues

```
void setup() {
  Serial.begin(115200);
  while (!Serial) {
    delay(10);
  }
  [...]
  initMetrics();
  initLedger();
  merkleTreeInit(&distributedLedgerTree);
  merkleTreeInit(&localLedgerTree);
  merkleTreeBuildFromLedger(&distributedLedgerTree);
  Serial.printf("[OK] (%d leaves)\n", distributedLedgerTree.leafCount);
  initKeyPair(&myKeyPair);
  memset(&state, 0, sizeof(ProtocolState));
  displayMenu();
  initBLEServer();
}

void loop() {
  [...]
  if (pendingProtocolRun) {
    pendingProtocolRun = false;
    if (protocolJustCompleted) {
      protocolJustCompleted = false;
      return;
    }
    if (!bleServerMode || pServer == nullptr) {
      Serial.println("\n[!] BLE not properly initialized - reinitializing...");
      initBLEServer();
      delay(500);
      return;
    }
    runFullProtocol();
  }
    displayMenu();
  }
}
```

Figure A.2 ESP32 Detailed Code Snippets

Appendix 3: Wokwi Simulation Output

```
[...]
|====================================|
|  FULL PROTOCOL (STANDALONE)       |
|====================================|
|====================================|
|   PHASE 1: Initial Handshake      |
|====================================|
Step 1: Generating ECC key pair...
[OK] Key pair generated
My nonce: 925F9286455E676AF8B93D5AF1E4938B
My public key: 04F7C1F74C436FBDCB75DB505F2ED017...
My HMAC: 82A0DF663F99692B892B4141CA7E5950...
Step 2: Encrypting handshake with PSK...
[OK] Encrypted handshake (141 bytes)
Step 3: Simulating peer handshake...
Step 4: Computing ECDH shared secret...
Shared secret: 8F677D5A02BA4BD83AC74EF504222E6B...
Step 5: Deriving K_temp...
K_temp: 804A7847BF1733B6034B9E6BFD7015A4...
[OK] PHASE 1 COMPLETE
Time: 4390 ms | Energy: 729.94 mJ | Bytes: 141 sent, 141 recv
|====================================|
|  PHASE 2: Ledger Verification     |
|====================================|
Step 1: Computing ID hash...
My ID hash: 40C7052E70F4C266251ABD998E7900A7...
Step 2: Encrypting ID hash...
[OK] ID hash encrypted
Step 3: Simulating peer ID...
Peer ID hash: D6C3D5BC63E01578E40BD3F0D118ECEE...
Step 4: Checking Bloom filter...
Bloom filter: [X] NOT FOUND
Peer lookup: [OK] FOUND
Devices not mutually recognized
[OK] PHASE 2 COMPLETE
Time: 38 ms | Energy: 6.67 mJ | Bytes: 121 sent, 121 recv
|====================================|
| PHASE 3: Merkle Root Exchange     |
|====================================|
Step 1: Building Merkle tree from ledger...
My Merkle root: 993103BB7EDD7B4BCB5A728F9194D6EB...
Tree has 5 leaves
Step 2: Encrypting Merkle root...
[OK] Root encrypted
```

Appendix 3: Continues

```
Step 3: Simulating peer root...
[Simulated different root]
Peer Merkle root: DB461B550C85319BF383B2F0B6FA9BA0...
Root comparison: [X] MISMATCH
[OK] PHASE 3 COMPLETE
Time: 29 ms | Energy: 5.23 mJ | Bytes: 60 sent, 60 recv
|====================================|
|  PHASE 4: Reconciliation          |
|====================================|
Step 1: Simulating proof exchange...
Round 1: +1 verified
Round 2: +3 verified
Round 3: +3 verified
Verified: 7/5 (140.0%)
Threshold: 60%
[OK] PHASE 4 COMPLETE
Time: 12 ms | Energy: 2.87 mJ | Bytes: 180 sent, 852 recv
|====================================|
| PHASE 5: Final Key Derivation     |
|====================================|
Step 1: Generating new nonces...
New nonce (mine): C8D8D48400FA30CE918B3A7B34746BBF
New nonce (peer): 7D4BC62C16B2FB1402ADF0B71A0252B5
Step 2: Encrypting nonce exchange...
[OK] Nonce encrypted
Step 3: Deriving K_final...
K_final: 4BC848DB48F3281B97A91749080141E2...
Step 4: Sending AUTH_COMPLETE...
[OK] AUTH_COMPLETE exchanged
Total protocol time: 4555 ms
Phase 5: 31 ms | Energy: 6.01 mJ | Bytes: 88 sent, 88 recv
|====================================|
|  [OK] AUTHENTICATION COMPLETE!       |
|====================================|
|====================================|
|      COMPREHENSIVE METRICS         |
|====================================|
|- Phase 1: Handshake ------------------------
| Status: [OK] Complete
| Time: 4390 ms
| Bytes: Sent=141, Recv=141
| Crypto Ops:
|   Hash: 0, HMAC: 1
|   Encrypt: 1, Decrypt: 0
|   ECDH: 1, HKDF: 1
```

Appendix 3: Continues

```
| Energy: 729.943 mJ
|- Phase 2: Ledger -------------------------
| Status: [OK] Complete
| Time: 38 ms
| Bytes: Sent=121, Recv=121
| Crypto Ops:
|   Hash: 4, HMAC: 0
|   Encrypt: 2, Decrypt: 0
|   ECDH: 0, HKDF: 0
| Energy: 6.666 mJ
|- Phase 3: Merkle -------------------------
| Status: [OK] Complete
| Time: 29 ms
| Bytes: Sent=60, Recv=60
| Crypto Ops:
|   Hash: 5, HMAC: 0
|   Encrypt: 1, Decrypt: 1
|   ECDH: 0, HKDF: 0
| Energy: 5.231 mJ
|- Phase 4: Reconcile -------------------------
| Status: [OK] Complete
| Time: 12 ms
| Bytes: Sent=180, Recv=852
| Crypto Ops:
|   Hash: 6, HMAC: 0
|   Encrypt: 3, Decrypt: 3
|   ECDH: 0, HKDF: 0
| Energy: 2.871 mJ
|- Phase 5: Final Key -------------------------
| Status: [OK] Complete
| Time: 31 ms
| Bytes: Sent=88, Recv=88
| Crypto Ops:
|   Hash: 0, HMAC: 0
|   Encrypt: 2, Decrypt: 2
|   ECDH: 0, HKDF: 1
| Energy: 6.006 mJ
|==================================|
|        AGGREGATE SUMMARY          |
|==================================|
Phases Completed: 5/5
Phases Skipped: 0 (fast path: NO)
--- Timing ---
Total Protocol Time: 4500 ms
Average Phase Time: 900.0 ms
```

Appendix 3: Continues

```
--- Communication ---
Total Bytes Sent: 590 bytes
Total Bytes Received: 1262 bytes
Total Bandwidth: 1852 bytes
--- Cryptographic Operations ---
Total Crypto Operations: 34
--- Energy Consumption ---
Total Energy: 750.717 mJ
Average per Phase: 150.143 mJ
--- Memory Usage ---
Heap at Start: 317532 bytes
Heap at End: 316848 bytes
Peak Usage: 684 bytes
======================================
Final Result: [OK] SUCCESS
======================================
```

Figure A.3 Wokwi simulation output

Appendix 4: ESP32-WROOM-32 Standalone Output

```
[...]

|====================================|
|  FULL PROTOCOL (STANDALONE)       |
|====================================|
|====================================|
|   PHASE 1: Initial Handshake      |
|====================================|
Step 1: Generating ECC key pair...
[OK] Key pair generated
My nonce: 49F408FD722CA1C7466C89DEB5B49CAC
My public key: 04DA03BA5209173DB5CF03863CD2CCAC...
My HMAC: 0D165C66F6DEB915FE8B74AA09C48EED...
Step 2: Encrypting handshake with PSK...
[OK] Encrypted handshake (141 bytes)
Step 3: Simulating peer handshake...
Step 4: Computing ECDH shared secret...
Shared secret: 5C631D7B6E33ED12E58ED7BA7B611182...
Step 5: Deriving K_temp...
K_temp: C228B62A496A56DC18E0D57AB846C7EB...
[OK] PHASE 1 COMPLETE
Time: 464 ms | Energy: 82.15 mJ | Bytes: 141 sent, 141 recv
|====================================|
|  PHASE 2: Ledger Verification     |
|====================================|
Step 1: Computing ID hash...
My ID hash: 40C7052E70F4C266251ABD998E7900A7...
Step 2: Encrypting ID hash...
[OK] ID hash encrypted
Step 3: Simulating peer ID...
Peer ID hash: 78E14C63EFA0D7E68DFC1EE4434F874A...
Step 4: Checking Bloom filter...
Bloom filter: [X] NOT FOUND
Peer lookup: [OK] FOUND
Devices not mutually recognized
[OK] PHASE 2 COMPLETE
Time: 32 ms | Energy: 5.68 mJ | Bytes: 121 sent, 121 recv
|====================================|
| PHASE 3: Merkle Root Exchange     |
|====================================|
Step 1: Building Merkle tree from ledger...
My Merkle root: F1306EC007706B4C43F5910ACE26E9E2...
Tree has 5 leaves
Step 2: Encrypting Merkle root...
```

Appendix 4: Continues

```
[OK] Root encrypted
Step 3: Simulating peer root...
[Simulated matching root]
Peer Merkle root: F1306EC007706B4C43F5910ACE26E9E2...
Root comparison: [OK] MATCH
* Identical ledgers - Fast Path possible...
* STANDALONE: Ignoring Fast Path to force Phase 4 testing
[OK] PHASE 3 COMPLETE
Time: 42 ms | Energy: 7.38 mJ | Bytes: 60 sent, 60 recv
|====================================|
|  PHASE 4: Reconciliation          |
|====================================|
Step 1: Simulating proof exchange...
Round 1: +4 verified
Round 2: +3 verified
Round 3: +3 verified
Verified: 10/5 (200.0%)
Threshold: 60%
[OK] PHASE 4 COMPLETE
Time: 21 ms | Energy: 4.36 mJ | Bytes: 180 sent, 852 recv
|====================================|
| PHASE 5: Final Key Derivation     |
|====================================|
Step 1: Generating new nonces...
New nonce (mine): 98A475A8D705344543F82C7D130E853E
New nonce (peer): 269A8EEFCB8F214323406917664AA6B2
Step 2: Encrypting nonce exchange...
[OK] Nonce encrypted
Step 3: Deriving K_final...
K_final: BC671FFB679B81C70941EAF6AF55165D...
Step 4: Sending AUTH_COMPLETE...
[OK] AUTH_COMPLETE exchanged
Total protocol time: 644 ms
Phase 5: 33 ms | Energy: 6.34 mJ | Bytes: 88 sent, 88 recv
|====================================|
|  [OK] AUTHENTICATION COMPLETE!       |
|====================================|
|====================================|
|      COMPREHENSIVE METRICS         |
|====================================|
|- Phase 1: Handshake ------------------------
| Status: [OK] Complete
| Time: 464 ms
| Bytes: Sent=141, Recv=141
| Crypto Ops:
```

Appendix 4: Continues

```
|   Hash: 0, HMAC: 1
|   Encrypt: 1, Decrypt: 0
|   ECDH: 1, HKDF: 1
| Energy: 82.154 mJ
|----------------------------------
|- Phase 2: Ledger -------------------------
| Status: [OK] Complete
| Time: 32 ms
| Bytes: Sent=121, Recv=121
| Crypto Ops:
|   Hash: 4, HMAC: 0
|   Encrypt: 2, Decrypt: 0
|   ECDH: 0, HKDF: 0
| Energy: 5.676 mJ
|----------------------------------
|- Phase 3: Merkle -------------------------
| Status: [OK] Complete
| Time: 42 ms
| Bytes: Sent=60, Recv=60
| Crypto Ops:
|   Hash: 5, HMAC: 0
|   Encrypt: 1, Decrypt: 1
|   ECDH: 0, HKDF: 0
| Energy: 7.376 mJ
|----------------------------------
|- Phase 4: Reconcile -------------------------
| Status: [OK] Complete
| Time: 21 ms
| Bytes: Sent=180, Recv=852
| Crypto Ops:
|   Hash: 6, HMAC: 0
|   Encrypt: 3, Decrypt: 3
|   ECDH: 0, HKDF: 0
| Energy: 4.356 mJ
|----------------------------------
|- Phase 5: Final Key -------------------------
| Status: [OK] Complete
| Time: 33 ms
| Bytes: Sent=88, Recv=88
| Crypto Ops:
|   Hash: 0, HMAC: 0
|   Encrypt: 2, Decrypt: 2
|   ECDH: 0, HKDF: 1
| Energy: 6.336 mJ
|----------------------------------
```

Appendix 4: Continues

```
|====================================|
|        AGGREGATE SUMMARY           |
|====================================|
Phases Completed: 5/5
Phases Skipped: 0 (fast path: NO)

--- Timing ---
Total Protocol Time: 592 ms
Average Phase Time: 118.4 ms

--- Communication ---
Total Bytes Sent: 590 bytes
Total Bytes Received: 1262 bytes
Total Bandwidth: 1852 bytes

--- Cryptographic Operations ---
Total Crypto Operations: 34

--- Energy Consumption ---
Total Energy: 105.897 mJ
Average per Phase: 21.179 mJ

--- Memory Usage ---
Heap at Start: 313952 bytes
Heap at End: 313280 bytes
Peak Usage: 672 bytes

====================================
Final Result: [OK] SUCCESS
====================================
```

Figure A.4 ESP32 standalone test output

Appendix 5: ESP32-WROOM-32 Two-device Testing Output

```
[...]

--- Scanning for BLE devices ---
Found 9 devices
Found target device: 7c:9e:bd:f6:c2:f6
BLE: Device connected
[OK] Connected to BLE server
MTU negotiated: 185
[OK] Found characteristic
*** Starting protocol automatically ***
|==================================|
|  RUNNING FULL PROTOCOL FLOW      |
|==================================|
|==================================|
|   PHASE 1: Initial Handshake     |
|==================================|
Step 1: Generating ECC key pair and nonce...
[OK] Key pair generated
My public key: 04D72E584FE3C44860D3AB44189F598D...
My nonce: C2914E34E8044AB425E5F4D7E8EBEDC9
My HMAC: A418D8E05D3CCEDFE5427381225D216C...
Step 2: Encrypting and sending handshake...
BLE: Sent 150 bytes (client)
[OK] Sent encrypted handshake [STEP=01] (150 bytes)
Step 3: Waiting for peer handshake...
[DEBUG] BLE: Received 77 bytes, tag=0x02
[X] Wrong step tag: expected 0x01, got 0x02
[!] Sending RESTART command to peer...
BLE: Sent 37 bytes (client)
[!] Protocol restart (1/3)
|==================================|
|  RUNNING FULL PROTOCOL FLOW      |
|==================================|
|==================================|
|   PHASE 1: Initial Handshake     |
|==================================|
Step 1: Generating ECC key pair and nonce...
[OK] Key pair generated
My public key: 0481A40682131F8C5022449D9BC9DD87...
My nonce: B82520F0F2DB819BF72CD1A1568CD8F2
My HMAC: 637FF8D4EF2A5F30590026000941A58A...
Step 2: Encrypting and sending handshake...
BLE: Sent 150 bytes (client)
[OK] Sent encrypted handshake [STEP=01] (150 bytes)
```

Appendix 5: Continues

```
Step 3: Waiting for peer handshake...
[DEBUG] BLE: Received 150 bytes, tag=0x01
Received 150 bytes [STEP=01]
Peer public key: 046398AA206E7BCA8B8E761B5F47C0A1...
Peer nonce: 039391B21A90406E87988D5384C672C5
Peer MAC: F4C2F6BD9E7C
Step 4: Verifying peer HMAC...
[OK] Peer HMAC verified
Step 5: Computing ECDH shared secret...
Shared secret: C7A079B7747E4C6399B693B93581DBC2...
Step 6: Deriving K_temp...
K_temp: 80A9F2D8F22DB3636B0ABCB63B493228...
[OK] PHASE 1 COMPLETE
Time: 360 ms (timeout: 30000 ms)
|==================================|
|  PHASE 2: Ledger Verification     |
|==================================|
Step 1: Computing ID hash...
My ID hash: 40C7052E70F4C266251ABD998E7900A7...
Step 2: Encrypting and sending ID hash...
BLE: Sent 77 bytes (client)
[OK] Sent ID hash [STEP=02] (77 bytes)
Step 3: Receiving peer ID hash...
[DEBUG] BLE: Received 77 bytes, tag=0x02
Received ID hash [STEP=02] (77 bytes)
Peer ID hash: 40C7052E70F4C266251ABD998E7900A7...
Step 4: Checking Bloom filter...
Bloom filter check: [X] NOT FOUND
Step 5: Exchanging lookup results...
BLE: Sent 46 bytes (client)
[OK] Sent lookup result [STEP=03]
[DEBUG] BLE: Received 46 bytes, tag=0x03
Peer's lookup result: [X] They didn't find us
Devices not mutually recognized - proceeding to Phase 3
[OK] PHASE 2 COMPLETE
Time: 947 ms
|==================================|
| PHASE 3: Merkle Root Exchange     |
|==================================|
Step 1: Building Merkle tree from ledger...
My Merkle root: 5D6FC3A87C36A0422182B24BC4A0CB23...
Tree has 20 leaves
Step 2: Sending Merkle root...
BLE: Sent 61 bytes (client)
[OK] Sent Merkle root [STEP=04]
```

Appendix 5: Continues

```
Step 3: Receiving peer Merkle root...
[DEBUG] BLE: Already buffered 61 bytes, tag=0x04
[DEBUG] BLE: Returning buffered message, 61 bytes
Received Merkle root [STEP=04]
Peer Merkle root: 9CA2762227853A6BBFF0A456841D222C...
Step 4: Comparing Merkle roots...
Root comparison: [X] MISMATCH
Ledgers differ - proceeding to Phase 4 reconciliation
[OK] PHASE 3 COMPLETE
Time: 1501 ms
|==================================|
|   PHASE 4: Channel Switch &       |
|        Reconciliation              |
|==================================|
Step 1: Negotiating WiFi roles...
My MAC: 0095F7BD9E7C, Peer MAC: F4C2F6BD9E7C
Role: CLIENT
Waiting for WiFi credentials from server...
[DEBUG] BLE: Received 74 bytes, tag=0x09
[DEBUG] connectToWiFiServer: received 74 bytes
[DEBUG] Step tag: 0x09 (expected 0x09)
[DEBUG] payloadLen=73, cipherLen=45
[DEBUG] IV offset=1, cipher offset=13, tag offset=58
[DEBUG] Attempting AES-GCM decrypt with 45 bytes ciphertext
[DEBUG] Decryption result: 0 (0=success)
[DEBUG] Decrypted WiFi info: ESP32-AUTH-F6BD9E7C|SecureAuth123|192.168.4.1
Connecting to WiFi: ESP32-AUTH-F6BD9E7C (server: 192.168.4.1)
[DEBUG] Password length: 13
[DEBUG] Stopping BLE before WiFi connection...
[DEBUG] BLE stopped
[DEBUG] Starting WiFi.begin()...
[DEBUG] WiFi status: 6 (DISCONNECTED) .....
[DEBUG] WiFi status: 0 (IDLE) ..
[DEBUG] Final WiFi status: 3 after 7 attempts
[OK] WiFi connected
IP: 192.168.4.2
[DEBUG] Connecting TCP to 192.168.4.1:8080
[OK] Connected to server
[OK] WiFi established for Phase 4
Step 2: Iterative Merkle proof exchange...
Round 1: Processing 10 proofs
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 1
WiFi: Sent 61 bytes
```

Appendix 5: Continues

```
WiFi: Received 93 bytes
  [OK] Verified proof 2
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 3
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 4
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 5
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 6
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 7
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 8
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 9
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 10
Current overlap: 50.0% (10/20)
Round 2: Processing 10 proofs
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 1
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 2
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 3
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 4
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 5
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
```

Appendix 5: Continues

```
  [OK] Verified proof 6
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 7
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 8
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 9
WiFi: Sent 61 bytes
WiFi: Received 93 bytes
  [OK] Verified proof 10
Current overlap: 100.0% (20/20)
Step 3: Evaluating trust threshold...
=== Reconciliation Summary ===
Verified entries: 20
Total entries: 20
Overlap: 100.0%
Threshold: 60%
[OK] PHASE 4 COMPLETE - Trust threshold met
|==================================|
| PHASE 5: Final Key Derivation     |
|==================================|
Step 1: Generating new nonces...
New nonce (mine): C82734AF27571F31DBA55217E50EE5D5
Step 2: Exchanging nonces...
WiFi: Sent 45 bytes
[OK] Sent nonce [STEP=07]
WiFi: Received 45 bytes
Received nonce [STEP=07]
New nonce (peer): A653C6703EB3F39C5F1FEDC81BC06ABE
Step 3: Deriving K_new...
[DEBUG] Nonce comparison result: 34
[DEBUG] Sorted nonce1: A653C6703EB3F39C5F1FEDC81BC06ABE
[DEBUG] Sorted nonce2: C82734AF27571F31DBA55217E50EE5D5
[DEBUG] Shared secret: C7A079B7747E4C6399B693B93581DBC2...
K_new: 6DCA28EA8B45FB3C2F40E51489997E2C...
Step 4: Exchanging AUTH_COMPLETE...
WiFi: Sent 42 bytes
[OK] Sent AUTH_COMPLETE [STEP=08]
WiFi: Received 42 bytes
Received AUTH_COMPLETE [STEP=08]
[OK] Peer AUTH_COMPLETE verified
|==================================|
```

Appendix 5: Continues

```
|  [OK] AUTHENTICATION COMPLETE!       |
|==================================|
Secure channel established
Total protocol time: 8096 ms
|==================================|
|      PROTOCOL SUMMARY              |
|==================================|
Phase 1 (Handshake):      [OK]
Phase 2 (Ledger):         [OK]
Phase 3 (Merkle):         [OK]
Phase 4 (Reconciliation): [OK]
Phase 5 (Final Key):      [OK]
[OK][OK][OK] FULL PROTOCOL SUCCESS [OK][OK][OK]
|==================================|
|      COMPREHENSIVE METRICS         |
|==================================|
 Phase 1: Handshake
| Status: [OK] Complete
| Time: 370 ms
| Messages: Sent=0, Recv=0
| Bytes: Sent=150, Recv=150
| Crypto Ops:
|   Hash: 0, HMAC: 2
|   Encrypt: 1, Decrypt: 1
|   ECDH: 2, HKDF: 1
| Energy: 71.742 mJ
 Phase 2: Ledger
| Status: [OK] Complete
| Time: 78 ms
| Messages: Sent=0, Recv=0
| Bytes: Sent=77, Recv=123
| Crypto Ops:
|   Hash: 4, HMAC: 0
|   Encrypt: 2, Decrypt: 2
|   ECDH: 0, HKDF: 0
| Energy: 13.464 mJ
Phase 3: Merkle
| Status: [OK] Complete
| Time: 55 ms
| Messages: Sent=0, Recv=0
| Bytes: Sent=61, Recv=61
| Crypto Ops:
|   Hash: 40, HMAC: 0
|   Encrypt: 1, Decrypt: 1
|   ECDH: 0, HKDF: 0
```

Appendix 5: Continues

```
| Energy: 11.253 mJ
Phase 4: Reconcile
| Status: [OK] Complete
| Time: 5489 ms
| Messages: Sent=0, Recv=0
| Bytes: Sent=3140, Recv=1860
| Crypto Ops:
|   Hash: 20, HMAC: 0
|   Encrypt: 20, Decrypt: 21
|   ECDH: 0, HKDF: 0
| Energy: 910.734 mJ
Phase 5: Final Key
| Status: [OK] Complete
| Time: 107 ms
| Messages: Sent=0, Recv=0
| Bytes: Sent=87, Recv=87
| Crypto Ops:
|   Hash: 0, HMAC: 0
|   Encrypt: 2, Decrypt: 2
|   ECDH: 0, HKDF: 1
| Energy: 18.546 mJ
|==================================|
|        AGGREGATE SUMMARY          |
|==================================|
Phases Completed: 5/5
Phases Skipped: 0 (fast path: NO)
-----Timing-----
Total Protocol Time: 6099 ms
Average Phase Time: 1219.8 ms
-----Communication-----
Total Bytes Sent: 3515 bytes
Total Bytes Received: 2281 bytes
Total Bandwidth: 5796 bytes
-----Cryptographic Operations-----
Total Crypto Operations: 123
-----Energy Consumption-----
Total Energy: 1025.739 mJ
Average per Phase: 205.148 mJ
-----Memory Usage-----
Heap at Start: 60004 bytes
Heap at End: 107728 bytes
Peak Usage: 0 bytes
```

Figure A.5 ESP32 two device test output