

POLITECNICO DI TORINO

Cybersecurity National Lab – CINI

Master Degree Thesis

Master Degree in Cybersecurity

Design and Implementation of an Interactive Web-Based Cyber Range for Hardware Security Training



Supervisor

Alessandro Savino

Co-supervisor

Samuele Yves Cerini

Candidate

Davide Leone

Academic Year 2025/2026

Abstract

Hardware security is an essential yet underserved area of cybersecurity education. While a mature ecosystem of browser-based Capture The Flag (CTF) platforms exists for software vulnerabilities, hands-on hardware training remains locked behind significant barriers: a basic workstation with multimeter, logic analyser, and serial adapters costs hundreds of euros, expendable target devices must be sourced and replaced, and professional courses charge between 3,000 and 8,000 EUR per participant. Existing intermediate solutions—remote laboratories, episodic hardware CTF competitions, and single-architecture emulators—require dedicated infrastructure, physical kit shipping, or fail to reproduce the board-level workflow that defines real hardware analysis: inspecting a PCB, probing test points, wiring a serial adapter, and dumping firmware.

This thesis addresses that gap by presenting PCB-CTF, a web-based interactive cyber range for hardware security training in CTF format. The platform lets a student interact with a high-resolution photograph of a real printed circuit board, perform simulated electrical measurements, establish serial connections through a virtual UART adapter, extract firmware via a simulated SPI programmer, and explore a fully emulated embedded terminal—entirely within the browser, with no server-side code execution.

The work delivers two contributions. The first is the PCB-CTF platform, built on three subsystems: a client-side instrument simulation that reproduces a digital multimeter, UART adapter, and SPI programmer with realistic electrical characteristics (differential voltages, impedances, crossover-wiring validation); a declarative terminal emulation engine that defines virtual filesystems, multi-stage boot sequences, custom commands, and multi-tab environments entirely through JSON configuration; and a no-code visual authoring interface with drag-and-drop PCB markup, structured exercise editing, and a preset system for versioning and distribution. Confining all execution to the client side eliminates an entire class of server-side vulnerabilities and enables offline operation after the initial page load.

The second contribution is a validation case study based on the TP-Link WR841N wireless router. A complete multi-step exercise reproduces the full hardware-analysis workflow—SoC and flash identification, UART connection with crossover wiring, SPI firmware extraction, and six progressive vulnerability-discovery challenges including boot-misconfiguration analysis, default-credential login, and password-hash cracking across multiple terminal tabs. The case study shows that a faithful, multi-step hardware investigation can be authored and delivered without writing code, and confirms that all twenty-three functional requirements are met in practice.

By removing cost, logistics, and security barriers simultaneously, PCB-CTF occupies a design space that no existing solution covers, offering a zero-cost, zero-install bridge between today’s software-centric training ecosystem and the hardware-aware skills the field increasingly demands. The work was developed within the SERICS programme (Spoke 4), funded under the Italian National Recovery and Resilience Plan (PNRR).

Acknowledgements

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

I would like to thank my supervisor, Alessandro Savino, and my co-supervisor, Samuele Yves Cerini, for giving me the opportunity to work on this thesis and collaborate with the ARTIC SERICS project at CINI, and Nicolò Maunero for his continuous support throughout this experience. This work 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 family. 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.

Finally, thank you to all my friends, for all the beautiful memories we have shared and for those still to come. A special thanks to Francesco and Riccardo, for their constant support throughout my career, and to Mirko, for sharing incredible adventures with me on our travels.

Contents

Abstract	2
Acknowledgements	3
1 Introduction	7
2 Background	9
2.1 Foundations of Hardware Security	9
2.1.1 The Role of Hardware in Cybersecurity	9
2.1.2 Hardware Security Threats and Adversary Motivations	10
2.1.3 Hardware Analysis Methodology	10
2.2 Embedded Systems Fundamentals	11
2.2.1 Architecture, SoC Families, and CPU Cores	11
2.2.2 Non-Volatile Storage and the MTD Layer	12
2.2.3 Boot Chain and Runtime Environment	12
2.3 Debug and Communication Interfaces	13
2.3.1 UART	13
2.3.2 SPI	14
2.4 Hardware and Software Tools	14
2.4.1 Hardware Instrumentation	14
2.4.2 Software Toolchain	14
2.4.3 Exploitability Factors in Consumer IoT Devices	15
2.5 Web Technologies and Frameworks	15
2.5.1 Modern Web Application Architectures	16
2.5.2 Component-Based UI Development with React	16
2.5.3 Static Typing in JavaScript Ecosystems	16
2.5.4 Client-Side State Management	17
2.5.5 Styling and Accessible UI Primitives	17
3 State of the Art	19
3.1 Hardware Security Training Landscape	19
3.1.1 Commercial and Community Training Platforms	19
3.1.2 IoT-Focused Hands-On Resources	20
3.2 Remote Access and Virtualisation Platforms	21
3.2.1 Remote Laboratories and Cyber Ranges	21
3.2.2 Circuit Simulators and Firmware Emulation	22
3.3 Gap Analysis	22
3.4 Positioning of This Work	23

4	Requirements Analysis	26
4.1	Motivations	26
4.2	Operational Context and Actors	27
4.3	Functional Requirements	27
4.3.1	PCB Simulation	28
4.3.2	Measurement and Connection Tools	28
4.3.3	Simulated Terminal	28
4.3.4	Exercise Structure	29
4.3.5	Authoring and Configuration	29
4.4	Non-Functional Requirements	29
4.4.1	Accessibility and Deployment	29
4.4.2	User Experience	30
4.4.3	Architecture and Security	30
4.5	Use Cases	30
4.5.1	Student Use Cases	30
4.5.2	Author Use Cases	31
5	System Architecture	32
5.1	Technology Stack	32
5.2	General System Architecture	33
5.3	Data Model	34
5.3.1	Exercise Configuration	34
5.3.2	Coordinate System	34
5.3.3	Terminal Configuration	34
5.4	State Management	35
5.4.1	Dirty Tracking	35
5.4.2	Persistence Strategy	36
5.5	Terminal Simulation Design	36
5.5.1	Design Principles	37
5.5.2	Command Execution Pipeline	37
5.5.3	Filesystem Abstraction	37
5.5.4	Boot Sequence System	38
5.5.5	Progressive Flag System	38
5.6	Data Flow Between Authoring and Simulation	38
6	Implementation	40
6.1	Interactive PCB Viewer	40
6.1.1	Coordinate Conversion	40
6.1.2	Snap-to-Pin Mechanism	41
6.1.3	Wire Rendering	42
6.1.4	Magnifying Lens	42
6.2	Simulated Hardware Analysis Tools	43
6.2.1	Digital Multimeter	43
6.2.2	UART Adapter	44
6.2.3	SPI Firmware Dumper	44
6.3	Terminal Engine	45
6.3.1	Configuration Loader	45
6.3.2	Command Executor	45
6.3.3	Boot Sequence Animation	47
6.3.4	Tab Completion	48
6.3.5	State Persistence Across Mounts	48

6.4	Exercise Completion System	48
6.5	Authoring Panel	49
6.5.1	Visual Canvas Editor	49
6.5.2	Terminal Configuration Editor	50
6.5.3	Preset Management	50
6.6	API Layer	51
6.7	Security by Design	51
6.7.1	Architectural Mitigation: Client-Side Execution	52
6.7.2	Input Validation and Sanitisation	52
6.7.3	Path Traversal Prevention	52
6.7.4	Access Control	53
6.7.5	Secure Deployment	53
6.7.6	Cross-Site Scripting Prevention	53
6.7.7	Static Analysis and Type Safety	53
6.7.8	Summary of OWASP Top 10 Coverage	54
7	Validation: Case Study	55
7.1	Device Selection	55
7.2	Exercise Structure	55
7.2.1	Step 1: Hardware Identification	55
7.2.2	Step 2: UART Connection and Vulnerability Analysis	56
7.2.3	Step 3: SPI Firmware Dump	56
7.2.4	Terminal Phase: Embedded System Exploitation	57
7.3	Terminal Configuration	58
7.3.1	Terminal Challenge Walkthrough	59
7.4	Preliminary Observations	60
7.5	Authoring Workflow	61
8	Conclusions	64
8.1	Summary of Contributions	64
8.2	Requirements Assessment	65
8.3	Discussion	65
8.4	Limitations	66
8.5	Future Work	67

Chapter 1

Introduction

Consider a cybersecurity student willing to explore the main concepts, vulnerabilities, and attacks in the vast domain of Hardware Security and Penetration Testing. They know that real-world attacks against IoT devices—extracting firmware through debug interfaces, identifying hardcoded credentials on flash storage, exploiting bootloader misconfigurations—require hands-on interaction with physical circuit boards. Yet the path forward is daunting. A basic workstation with a multimeter, logic analyser, UART adapter, and SPI programmer can cost up to several hundred euros. Expendable target devices must be sourced and periodically replaced. Professional training courses at conferences like Black Hat or through SANS carry fees between 3,000 and 8,000 EUR per participant [1]. Unless their university maintains a dedicated hardware security laboratory—or their employer, if any, is willing to invest in hardware-security education—and most do not, they have no affordable way to develop these skills.

This scenario is not hypothetical: it captures a structural problem in cybersecurity education with measurable consequences. Costin et al. [2] analysed over 32,000 firmware images from embedded devices and found that a majority contained at least one serious vulnerability—hardcoded passwords, unencrypted storage, or exposed debug interfaces—all entry-level findings that a trained analyst can identify with a multimeter, a serial adapter, and basic command-line skills. The bottleneck is not the difficulty of the analysis but the scarcity of training environments in which students can develop these skills without a physical laboratory. A mature ecosystem of browser-based software Capture The Flag platforms already exists and confirms that practical, interactive exercises produce measurably better learning outcomes than lecture-only instruction [3]; the same body of work reveals a stark asymmetry, however, in that virtually all existing platforms target software vulnerabilities, while hardware security training remains locked behind equipment costs, logistics, and physical access requirements.

Intermediate solutions exist but leave significant gaps. Remote laboratories such as iLab [4] grant network access to real instruments, and more recent efforts remotise hardware-based CTF challenges by streaming interaction with development boards over the network [5]; both approaches, however, demand dedicated server infrastructure and support only limited concurrency. Hardware CTF competitions like RHme [6] produce excellent pedagogical experiences, but they are episodic events that require physical kits shipped to registered participants. Platforms like Microcorruption [7] focus narrowly on a single microcontroller architecture and do not simulate the physical board-level workflow that defines hardware analysis in practice: visually inspecting a PCB, probing test points with a multimeter, wiring a serial adapter with the correct crossover convention, dumping firmware through an SPI programmer. No existing solution combines these capabilities with a no-code authoring workflow in a zero-cost, browser-based package.

This thesis addresses that gap by presenting PCB-CTF, a web-based interactive cyber range for hardware security training in CTF format. The platform enables a student to interact with a high-resolution photograph of a real printed circuit board, perform simulated electrical measurements, establish serial connections through a virtual UART adapter, extract firmware via a simulated SPI programmer, and explore a fully emulated embedded terminal—all within the browser, with no server-side code execution.

The contributions of this work are twofold. The first is the PCB-CTF platform itself: a zero-install cyber range that reproduces the full PCB hardware-analysis workflow on the client side, integrating a client-side simulation of the hardware analysis instruments (digital multimeter, UART adapter, SPI programmer), a declarative terminal-emulation engine, and a visual no-code authoring interface. The second is a validation through a real-world case study based on the TP-Link WR841N router, building on Genova’s [1] vulnerability assessment to demonstrate that a multi-step hardware investigation can be faithfully reproduced in a fully virtual environment.

The remainder of the thesis is organised as follows. Chapter 2 establishes the technical foundations of hardware security, embedded systems, debug interfaces, and the web technologies underlying the platform. Chapter 3 surveys hardware security training, CTF platforms, and remote laboratories. Chapter 4 formalises the requirements. Chapters 5 and 6 detail the system architecture and its implementation. Chapter 7 presents the TP-Link WR841N case study, and Chapter 8 draws conclusions and identifies directions for future work.

This work has been developed within the SERICS (Security and Rights in CyberSpace) national research programme, funded under the Italian National Recovery and Resilience Plan (PNRR). The project falls under Spoke 4, “Operating Systems and Virtualization Security”, which includes among its objectives the development of accessible training resources for hardware security. The ARTIC (Advanced Research on Testing and Intelligent Cybersecurity) project within Spoke 4 provided the institutional and research context for this thesis.

Chapter 2

Background

Hardware security sits at the intersection of physical engineering and adversarial thinking. While software vulnerabilities dominate public attention, a growing body of research demonstrates that physical access to a device often collapses its entire security model—authentication bypassed through a serial console, encryption rendered irrelevant by reading plaintext flash, firmware integrity undermined by the absence of secure boot. We introduce the foundational concepts that the rest of this thesis builds upon: the threat landscape, the architecture of the embedded systems we target, the protocols through which they can be interrogated, and the tools that make analysis practical. Throughout, we use the TP-Link WR841N router—the case study device of Chapter 7—as a running example.

2.1 Foundations of Hardware Security

2.1.1 The Role of Hardware in Cybersecurity

Software-centric security models rest on an implicit assumption: the underlying hardware operates correctly and cannot be tampered with. Physical access invalidates this assumption. An adversary who can physically access a device can probe electrical signals, read memory chips such as Flash, inject faults, or extract cryptographic keys through side-channel observations [8]. Hardware is therefore both the foundation of all software security guarantees and, when inadequately protected, a direct path to compromise.

The distinctiveness of hardware security lies not only in the attack vectors it exposes but in the nature of the assets at risk and the timescales over which compromise unfolds. A tampered device may circulate in the field for years, carrying extracted cryptographic keys or modified firmware into production environments. The objectives of a hardware attacker—recovering secrets embedded at manufacture, bypassing access controls that rely on device integrity, or injecting persistent modifications into long-lived deployments—therefore demand a correspondingly distinct defensive posture.

Prinetto [9] classifies hardware security threats along three dimensions: abstraction level (circuit, microarchitecture, system), required access (remote, local, physical), and outcome (information leakage, denial of service, control hijack). The classification underscores that hardware security is not a single discipline but a family of concerns—a fact reflected in the breadth of the training objectives we define for the PCB-CTF platform.

2.1.2 Hardware Security Threats and Adversary Motivations

Adversary motivations span intellectual property theft, espionage, sabotage, and financially motivated attacks—most notably against consumer IoT devices, where low cost, minimal security investment, and massive deployment volumes converge into an especially attractive target profile [10, 11].

Any account of hardware threats is incomplete without a parallel account of the adversaries behind them. Van Woudenberg and O’Flynn [12] arrange hardware attackers along a continuum defined by budget, time, and instrumentation. At one end sits the hobbyist whose toolkit seldom exceeds a multimeter, a USB-to-serial adapter, and a few tens of euros of consumables; at the other, state-sponsored teams with access to decapsulation chambers, focused ion beam workstations, and multi-month engagements. Between these extremes lie the laboratory-grade analyst—equipped with oscilloscopes, logic analysers, and rework stations, typical of academic groups and penetration-testing firms—and the dedicated offensive outfit, whose budget permits custom tooling but falls short of full-silicon reverse engineering. This gradient is more than descriptive: each tier enables a different subset of the vectors enumerated below. Debug-interface exploitation and firmware extraction sit within reach of the first tier; side-channel analysis presupposes laboratory instrumentation; laser-based fault injection remains confined to well-funded specialists.

The attack vectors most relevant to this work are:

- **Debug interface exploitation.** Protocols such as UART, JTAG, and SWD left accessible on production boards provide direct console access or memory read-out [13].
- **Firmware extraction and analysis.** Flash contents read via SPI or I²C expose filesystem images, cryptographic keys, and hardcoded credentials [2].
- **Side-channel analysis.** Power consumption, electromagnetic emissions, and timing variations during cryptographic operations can reveal secret key material [14].
- **Fault injection.** Voltage glitching, clock manipulation, or laser illumination can alter execution flow, bypassing security checks or corrupting cryptographic computations [15].

The first two vectors—debug interfaces and firmware extraction—are the focus of the PCB-CTF platform, because they represent the entry-level skills most frequently exercised in professional hardware security assessments and are amenable to faithful browser-based simulation.

2.1.3 Hardware Analysis Methodology

Hardware compromise follows a structured methodology that experienced analysts apply almost instinctively but that students must learn through deliberate practice. The workflow proceeds in seven phases:

1. **Physical reconnaissance.** The analyst opens the device enclosure and inspects the PCB, identifying major ICs (SoC, flash, DRAM), connectors, test points, and component markings.
2. **Electrical characterisation.** The analyst measures voltage levels and impedances at suspected interface pins using a multimeter.

3. **Interface identification.** Based on the electrical measurements, the analyst confirms the protocol type and pin assignments. Three wires at 3.3 V/0 V levels with the described impedance pattern is a strong UART signature.
4. **Communication establishment.** The analyst connects a USB-to-Serial adapter with correct crossover wiring (adapter TX to device RX, adapter RX to device TX, GND to GND) and opens a terminal emulator at 115 200 bps.
5. **Firmware extraction.** For devices with SPI flash, the analyst connects a flash programmer (e.g., CH341A) to the chip's VCC, GND, CS, CLK, MOSI, and MISO pins and reads the full flash contents to a binary file.
6. **Firmware analysis.** The extracted binary is processed with Binwalk to identify and extract filesystem images. The root filesystem is mounted and explored for configuration files, credentials, and executables of interest.
7. **Vulnerability identification.** The analyst searches for hardcoded credentials (in files such as `/etc/shadow` and `/etc/passwd`), sensitive data in configuration partitions (via `strings` on raw partition images), and vulnerable code patterns in binaries (e.g., injection-prone function calls found with `strings`).

These phases differ sharply in cost and required expertise: visual reconnaissance and electrical characterisation can be performed with a multimeter and a few hours of work, whereas firmware analysis and vulnerability identification demand specialist tooling and binary reverse-engineering skills. This gradient informs both the training objectives of the PCB-CTF platform and the ordering of its exercises, where each phase maps to one or more exercise objectives. The platform cannot fully replicate the physical experience of holding a probe against a test pad, but it faithfully reproduces the *reasoning process*: observe, measure, hypothesise, connect, extract, analyse. The case study in Chapter 7 demonstrates that this reasoning process can be taught effectively through simulation.

2.2 Embedded Systems Fundamentals

2.2.1 Architecture, SoC Families, and CPU Cores

Consumer IoT devices overwhelmingly share a common architecture: a System-on-Chip (SoC) integrating CPU core, memory controllers, and peripheral interfaces; a NOR flash chip for non-volatile storage; DRAM for runtime memory; and a PCB that interconnects these components through copper traces. The SoC is typically the largest IC on the board and can be identified visually by package size and pin count. On the WR841N, this role is played by the Qualcomm Atheros QCA9533, a MIPS-based processor clocked at 650 MHz.

Beyond the QCA95xx series found in TP-Link routers, common SoC families in low-cost IoT devices include MediaTek MT76xx (routers and access points) and Realtek RTL8196/RTL8197 (networking equipment). These processors run MIPS or ARM cores at 400 MHz to 1 GHz, with 32–128 MB of DRAM and 4–32 MB of flash. The WR841N sits at the lower end of this spectrum: 4 MB NOR flash (Winbond W25Q32, 8-pin SOIC package) and 32 MB DRAM—enough to run a stripped-down Linux distribution but with no margin for encryption overhead or integrity checking, a trade-off with direct security consequences.

ARM and MIPS dominate the IoT space—ARM in smartphones, wearables, and higher-end embedded devices, MIPS in networking equipment from TP-Link, D-Link, and

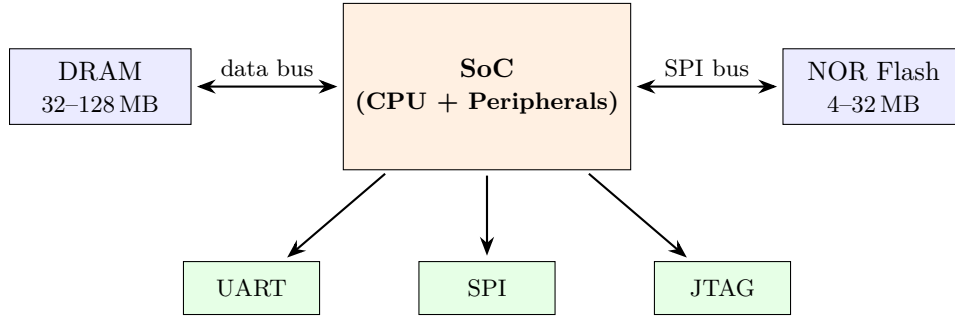


Figure 2.1: Block diagram of a typical consumer-grade embedded system, showing the SoC, flash memory, DRAM, and debug interfaces (UART, SPI, JTAG).

Netgear. Knowing the target ISA determines the choice of disassembler and decompiler (for instance Ghidra, IDA Pro, or Radare2), the calling conventions, and the exploitation techniques available to the analyst.

2.2.2 Non-Volatile Storage and the MTD Layer

Consumer IoT devices overwhelmingly use NOR flash for firmware storage, accessed via SPI. NOR flash provides random read access—essential for execute-in-place bootloaders—and ships in compact 8-pin SOIC packages that are straightforward to probe with clip adapters. Higher-density NAND flash appears in devices with larger storage needs but requires a flash translation layer for wear levelling. EEPROM, with byte-level write granularity, is occasionally used for small configuration blobs.

On the WR841N and similar devices, the entire firmware—bootloader, kernel, and root filesystem—resides on a single NOR flash chip. The typical partition layout, shown in Figure 2.2, dedicates the first sector to U-Boot, followed by the compressed kernel, a read-only SquashFS root filesystem, and a writable JFFS2 configuration partition.

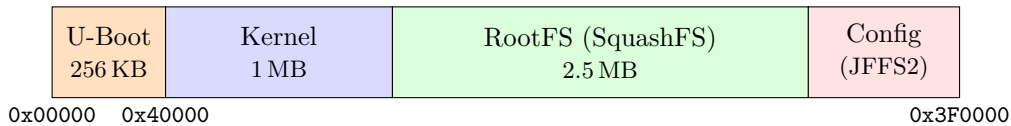


Figure 2.2: Typical flash memory partition layout in a Linux-based IoT device, showing the bootloader, kernel, root filesystem (SquashFS), and configuration (JFFS2) partitions.

Linux-based embedded systems organise flash through the Memory Technology Device (MTD) subsystem, which exposes raw flash partitions as block devices (`/dev/mtdblockN`). The configuration partition in particular is of interest to an attacker: it frequently stores Wi-Fi credentials, PPPoE passwords, and administrative credentials in plaintext or with trivially reversible encoding.

2.2.3 Boot Chain and Runtime Environment

Das U-Boot is the de facto standard bootloader for embedded Linux. It handles hardware initialisation, memory testing, and kernel loading from flash into RAM. A command-line interface, accessible via UART during a configurable timeout window, exposes commands that are invaluable for security analysis: `printenv` reveals boot arguments and memory layout, `md` dumps arbitrary memory regions, and `sf read/write` performs SPI flash operations. On the WR841N, `printenv` discloses the kernel load address, root filesystem

type, and boot arguments—information that the PCB-CTF case study exercise uses as the basis for its first terminal challenge.

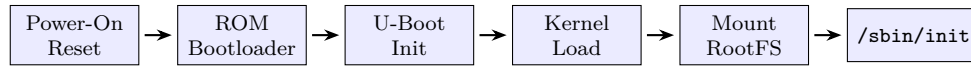


Figure 2.3: Boot chain of a typical embedded Linux device: power-on reset, ROM bootloader, U-Boot initialisation, kernel loading, root filesystem mount, and `init` process.

Embedded devices are usually split into two broad categories: Real-Time Operating Systems (FreeRTOS, Zephyr, proprietary firmware) and Linux-based distributions (typically OpenWrt or vendor-customised BusyBox). The latter—dominant in consumer routers and IP cameras—offers a familiar Unix environment that is more accessible to exploitation and constitutes the target category of the PCB-CTF platform.

2.3 Debug and Communication Interfaces

UART and SPI are the two serial protocols most relevant to this work, as they constitute the primary interfaces through which the PCB-CTF platform simulates hardware interaction.

2.3.1 UART

The Universal Asynchronous Receiver-Transmitter is a point-to-point serial protocol used extensively for debug consoles. Without a shared clock, both endpoints must agree on a baud rate—115 200 bps is standard for embedded consoles. Each frame consists of a start bit (logic low), 8 data bits (LSB first), an optional parity bit, and one or two stop bits (logic high).

A minimal connection requires three wires: TX, RX, and GND. The critical wiring rule is the *crossover convention*: the TX output of one device connects to the RX input of the other. The PCB-CTF platform explicitly validates this rule during the UART connection step (Section 6.2.2).

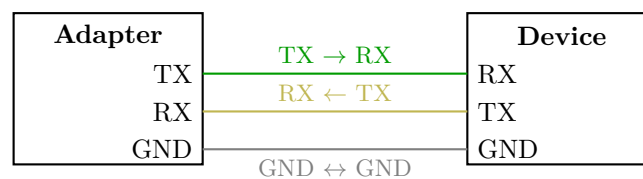


Figure 2.4: UART crossover wiring convention. The adapter’s TX connects to the device’s RX and vice versa. GND is shared directly.

From a security standpoint, UART consoles matter because manufacturers routinely leave them enabled on production boards. On the WR841N, the four-pin header near the SoC provides an unauthenticated root shell at 115 200 bps—no password prompt, no rate limiting. Shwartz et al. [13] found that over 45% of analysed IoT devices exposed an active UART interface.

Identifying UART pins on an unknown board involves two steps: (i) visual inspection for groups of 3–4 aligned pins near the SoC, and (ii) multimeter measurement. TX idles at logic high (3.3 V, low output impedance $\sim 65\ \Omega$), RX also idles high but with high input impedance ($\sim 47\ \text{k}\Omega$ due to a pull-up resistor), and GND reads 0 V. These impedance differences allow an analyst to distinguish TX from RX without an oscilloscope—a technique the PCB-CTF multimeter simulation reproduces.

2.3.2 SPI

The Serial Peripheral Interface is a synchronous, full-duplex protocol used primarily for flash memory access. It uses four signals: CS (Chip Select, active low) selects the target peripheral; CLK (Clock) synchronises transfers; MOSI (Master Out, Slave In) carries data to the peripheral; MISO (Master In, Slave Out) carries data back. VCC and GND complete the connection.

For firmware extraction, a flash programmer such as the CH341A acts as SPI master, issuing read commands over MOSI and capturing responses on MISO. The PCB-CTF platform simulates this by requiring six correct probe connections (VCC, GND, CS, CLK, MOSI, MISO) before initiating the dump.

In idle state (CS deasserted), the SPI signals exhibit characteristic voltage levels: CS is pulled high to 3.3 V via a 10 k Ω pull-up; CLK and MOSI are driven low by the SoC ($\sim 65\ \Omega$ output impedance); MISO floats at ~ 0.15 V with very high impedance (~ 150 k Ω) because the flash output is tri-stated when not selected. These signatures guide pin identification and are reproduced in the platform's multimeter simulation.

2.4 Hardware and Software Tools

2.4.1 Hardware Instrumentation

Embedded device analysis requires a modest toolkit:

- **Digital multimeter.** Identifies voltage levels on PCB test points—3.3 V near the SoC may indicate a UART TX line, 0 V indicates ground.
- **USB-to-Serial adapter.** Converts UART signals to USB for serial console access. FTDI FT232 and CP2102 adapters are standard.
- **Logic analyser.** Captures digital signals on multiple channels for protocol identification and timing analysis.
- **SPI flash programmer.** The CH341A reads and writes flash memory chips directly, enabling firmware extraction without powering the target device.

2.4.2 Software Toolchain

After firmware extraction, analysis proceeds with:

- **Binwalk** [16]: scans binary images for file signatures and entropy patterns, automating filesystem extraction.
- **Flashrom** [17]: reads, writes, and verifies flash chips via SPI, parallel, or FWH interfaces.
- **Ghidra** [18]: supports disassembly and decompilation for ARM, MIPS, and other architectures.
- **Hashcat / John the Ripper:** recover cleartext passwords from hashes extracted from `/etc/shadow` or configuration files.
- **Strings:** extracts printable character sequences from binary files, often revealing hardcoded credentials and debug messages.

2.4.3 Exploitability Factors in Consumer IoT Devices

The vulnerabilities encountered during hardware analysis of consumer IoT devices are not random; they follow recurring patterns rooted in economic and engineering trade-offs. Understanding these patterns is essential for designing training exercises that cover the most practically relevant attack categories. Based on the literature [13, 2, 1] and the device analyses informing the case study in Chapter 7, we identify six key exploitability factors—each of which maps to a specific training objective in the PCB-CTF platform:

1. **Exposed debug interfaces.** UART headers, JTAG test points, and SWD connectors are frequently left accessible on production boards. Manufacturers include them for factory testing and field diagnostics but rarely disable them before shipping [13]. *Training objective:* the platform teaches students to visually locate these headers and verify them with multimeter measurements before establishing a serial connection—the core skill exercised in the UART connection step.
2. **Unencrypted firmware storage.** SPI NOR flash contents are stored in plaintext, with no encryption or integrity verification. The entire firmware of the WR841N can be read with a CH341A programmer in under 30 seconds. *Training objective:* the SPI firmware dump step requires students to correctly identify and connect all six probe wires, reinforcing the physical extraction workflow.
3. **Hardcoded credentials.** Default passwords for administrative access are embedded in the firmware and are often identical across all units of a given model. On the WR841N, the credentials `root/sohoadmin` are widely documented. *Training objective:* a terminal challenge requires the student to gain root access using these default credentials, demonstrating why hardcoded passwords are a systemic risk.
4. **Sensitive data in configuration partitions.** The JFFS2 configuration partition on many routers stores Wi-Fi passwords, PPPoE credentials, and administrative passwords in plaintext or with trivially reversible encoding. Running `strings` on the raw partition image is often sufficient. *Training objective:* a terminal challenge tasks the student with extracting credentials from the configuration partition, reinforcing both the `strings` technique and the MTD partition model.
5. **Absent secure boot.** Most low-cost IoT devices lack a hardware root of trust. The bootloader does not verify kernel integrity or authenticity, enabling firmware modification attacks. *Training objective:* the U-Boot analysis challenge asks the student to identify inconsistencies in boot environment variables that reveal the absence of integrity checking.
6. **Vulnerable network services.** Web servers and UPnP services on IoT devices frequently contain command injection vulnerabilities—functions like `execFormatCmd()` or direct `system()` calls with unsanitised input are common findings. *Training objective:* a binary analysis challenge in the terminal phase requires the student to locate such a vulnerable function in the extracted firmware, connecting the hardware extraction workflow to its ultimate security payoff.

2.5 Web Technologies and Frameworks

The PCB-CTF platform is not only a hardware security training tool: it is also a modern single-page web application whose design decisions—client-side simulation, zero-install deployment, declarative authoring—depend critically on a specific ecosystem of web

technologies. This section introduces, at a foundational level, the families of frameworks and patterns that together constitute that ecosystem. Our goal is to equip the reader with enough vocabulary to follow the architectural discussion in Chapter 5 and the implementation details in Chapter 6, where the specific choices adopted by PCB-CTF are motivated and put into practice.

2.5.1 Modern Web Application Architectures

Web applications have historically oscillated between two rendering models. In the *multi-page* model, every user navigation triggers a full round trip to the server, which returns a complete HTML document; this approach is simple and search-engine friendly but produces perceived latency on every interaction. The *single-page application* (SPA) model, popularised in the early 2010s, loads a JavaScript bundle once and performs subsequent navigation and data fetching entirely on the client, yielding near-instant interactions at the cost of longer initial loads and more complex state management.

Contemporary *meta-frameworks* attempt to combine the strengths of both approaches. They support *server-side rendering* (SSR) or *static site generation* (SSG) to produce pre-rendered HTML for the first paint, then *hydrate* the document on the client so that subsequent interactions behave as in an SPA. `Next.js` [19] is a representative example: built on top of `React` (Section 2.5.2), it introduces a file-system-based routing scheme, hybrid rendering modes selectable per route, and—in recent versions—a *standalone output* mode that produces a self-contained Node.js server image suitable for single-container deployment. These capabilities are what allow a simulator like PCB-CTF to ship as a single Docker artifact while still offering SPA-like interactivity.

2.5.2 Component-Based UI Development with React

`React` [20] is a JavaScript library for building user interfaces based on a *component* abstraction: the interface is decomposed into self-contained, reusable units, each of which declaratively describes its rendered output as a function of its current inputs. Rather than mutating the Document Object Model (DOM) directly, `React` maintains an in-memory representation—the *virtual DOM*—and computes a minimal set of DOM operations through a diffing algorithm called reconciliation. This separation between the declared state and the physical DOM is what allows complex, deeply nested interfaces to update efficiently in response to state changes.

Since the introduction of *hooks* in `React` 16.8, function components (as opposed to the older class-based components) have become the idiomatic unit of UI code. Hooks such as `useState`, `useEffect`, and `useRef` let function components encapsulate local state, side effects, and mutable references without inheritance. The declarative, composable style that hooks encourage maps naturally onto the kind of richly interactive UI required by a hardware simulator, where probes, overlays, and terminal tabs all need to react to a shared underlying state in a predictable way.

2.5.3 Static Typing in JavaScript Ecosystems

JavaScript is a dynamically typed language, which is well suited to rapid prototyping but increasingly fragile as a codebase grows. `TypeScript` [21] addresses this limitation by layering a *structural* type system on top of JavaScript: any valid JavaScript program is also a valid `TypeScript` program, and type annotations are erased at compile time to produce plain JavaScript output. The typing discipline is structural rather than nominal—two types are considered compatible if their shapes match, regardless of their

declared names—which aligns well with the duck-typed idioms of existing JavaScript libraries.

TypeScript supports *gradual* adoption: individual files, or even individual variables within a file, may remain untyped, enabling incremental migration of legacy code. In modern React projects, however, strict compiler settings (`strict`, `noImplicitAny`, `strictNullChecks`) are increasingly the default, and the type system is used to encode invariants such as exhaustive handling of enumerated message types, nullable reference tracking, and discriminated unions for state machines. For a domain with as many moving parts as a hardware simulator, this form of compile-time verification shortens the feedback loop on refactors and reduces the surface for regressions.

2.5.4 Client-Side State Management

As SPAs grew in complexity, the question of how to organise state that is shared between many components became a first-class architectural concern. An early and influential answer was the *Flux* pattern proposed by Facebook, in which UI events are transformed into explicit *actions* that flow through a central dispatcher and produce new immutable state in one or more stores. Redux [22] popularised a refined version of this pattern, combining a single global store, pure reducer functions, and time-travel debugging tools.

Flux/Redux promotes predictability at the cost of considerable boilerplate: defining actions, action creators, reducers, and selectors for every slice of state. In response, a second generation of state libraries has emphasised ergonomics. *Zustand* [23] is representative: it exposes a minimalist API in which a store is a plain function returning state and actions, components subscribe through a single hook, and updates are triggered by direct function calls rather than dispatched actions. This lean approach retains the centralised-store benefits—shared state across components, easy persistence, clear ownership—while drastically reducing the amount of code required. Such lightweight stores are particularly well suited to applications like a PCB simulator, in which multiple specialised stores (e.g. one for exercise progress, one for authoring drafts, one for terminal configuration) must coexist without imposing a single monolithic state tree.

2.5.5 Styling and Accessible UI Primitives

The styling layer of modern web applications has converged on a set of complementary practices in which utility-first CSS, accessible unstyled primitives, and source-level component distribution replace the monolithic design systems of earlier generations.

The *utility-first* approach, popularised by *Tailwind CSS* [24], inverts the traditional separation between semantic markup and external stylesheets: rather than authoring bespoke style rules, developers compose interfaces by applying small, single-purpose utility classes (`flex`, `p-4`, `text-sm`) directly in markup. A build step scans the source code and emits only the utilities actually used, keeping the final CSS bundle small and co-locating the style description with the component it applies to.

Complementary to utility-first styling is the separation between *behaviour* and *appearance* in UI primitives. Libraries such as *Radix UI* [25] provide unstyled, fully accessible implementations of common interactive components—dialogs, dropdown menus, tooltips, tabs—that take care of keyboard navigation, focus management, and WAI-ARIA attributes while leaving all visual decisions to the consumer. The resulting components are correct by construction with respect to accessibility requirements, yet visually indistinguishable from bespoke ones.

A more recent development is the *component distribution* model exemplified by *shadcn/ui* [26], in which the developer copies the source code of each component directly

into the project—typically a combination of Radix primitives and Tailwind utility classes—rather than installing a component library as a binary dependency. This trades the convenience of versioned upgrades for full local control over component behaviour, styling, and API surface, a trade-off particularly attractive to projects that require deeply customised interactive widgets.

Chapter 3

State of the Art

Hardware security training faces a fundamental tension: the skills that matter most—probing a PCB, identifying debug interfaces, extracting firmware—are inherently physical, yet the platforms best positioned to deliver scalable education are entirely digital. We examine the existing landscape of training solutions, from professional courses and CTF competitions to remote laboratories and cyber ranges, evaluating each not only for what it offers but for what it leaves unaddressed. The gaps that emerge from this analysis directly motivate the design of the PCB-CTF platform.

3.1 Hardware Security Training Landscape

3.1.1 Commercial and Community Training Platforms

Hands-on hardware security training is primarily delivered through professional conferences. Black Hat USA and Europe offer multi-day courses on hardware hacking, embedded device exploitation, and IoT penetration testing, with fees typically between 3,000 and 5,000 EUR [1]. The SANS Institute runs specialised courses (e.g., SEC556: IoT Penetration Testing) at 5,000–8,000 EUR. These courses offer excellent depth and direct access to physical equipment, but their cost and limited scheduling restrict participation to sponsored professionals. University students, independent researchers, and self-learners—precisely the populations that would benefit most from structured hardware security training—are largely excluded.

Capture The Flag competitions have become the dominant paradigm for practical cybersecurity training. Research by Švábenský et al. [3] confirms that CTF-based learning environments effectively develop technical security skills, particularly when challenges are structured progressively and accompanied by pedagogical scaffolding. Software-focused platforms illustrate this well: CTFd [27] and PicoCTF [28] provide scalable, browser-accessible challenges spanning web exploitation, binary analysis, cryptography, and network security. Hack The Box [29] and TryHackMe [30] extend the model with guided learning paths and realistic machine-based scenarios. These platforms demonstrate that gamified, self-paced environments can develop security skills at scale—but they operate entirely in software, with no mechanism for simulating physical hardware interaction.

Hardware-oriented CTF competitions exist but remain far less accessible:

- **eCTF** (MITRE): an annual embedded systems CTF requiring teams to develop and attack custom hardware designs on FPGA development boards. The barrier to entry—both in equipment cost and prerequisite expertise—limits participation to well-resourced university teams.

- **RHme** [6] (Riscure): a hardware CTF series shipping Arduino-based target boards to registered teams. The format is innovative but inherently constrained by physical distribution logistics and per-edition hardware availability.
- **CSAW ESC** [31]: an annual academic competition involving embedded security analysis, restricted to university teams with laboratory access.
- **Microcorruption** [7]: a web-based CTF focused on exploiting an MSP430 microcontroller emulator. It provides an outstanding learning experience for low-level exploitation, but targets a single architecture and a single abstraction level—there is no PCB, no multimeter, no physical reconnaissance phase.
- **OWASP IoTGoat** [32]: a deliberately vulnerable firmware image for security testing practice, targeting network-level and application-level vulnerabilities rather than hardware analysis.

A recent proposal by Roascio et al. [5] seeks to overcome the distribution constraints of physical hardware CTFs by streaming networked access to development boards such as the ChipWhisperer, retaining authentic hardware interaction while removing the need to ship per-participant kits; the approach, however, still requires dedicated server resources proportional to concurrent participants.

A common limitation cuts across all of these: none provides pedagogical scaffolding that guides a student through the complete hardware analysis workflow. Physical CTFs assume participants already know how to use the equipment; software CTFs skip the physical layer entirely. The gap between “knowing that UART exists” and “being able to locate, verify, and connect to a UART header on an unfamiliar board” remains unaddressed.

3.1.2 IoT-Focused Hands-On Resources

Several standalone resources target IoT and embedded security education, each covering a different slice of the problem. The Damn Vulnerable Router Firmware (DVRF) [32] provides a deliberately vulnerable MIPS firmware image that can be emulated in QEMU, allowing students to practise binary exploitation on a realistic target. Its pedagogical limitation is fundamental: DVRF assumes the student has already obtained the firmware. The hardware reconnaissance and extraction phases—where most real-world hardware assessments begin—fall outside its scope.

The Attify Badge packages a UART/JTAG adapter with guided exercises for analysing real IoT devices. While pedagogically effective, it requires purchasing a physical kit (~40 EUR) and a target device, reintroducing the cost and logistics barriers that web-based solutions aim to eliminate. Scaling to a classroom of 30 students means 30 kits, 30 target devices, and an instructor who can troubleshoot 30 different wiring mistakes simultaneously.

More recently, Hack The Box [29] and TryHackMe [30] have introduced IoT-themed challenges, but these focus on network-level exploitation—firmware downloaded via HTTP, API vulnerabilities, misconfigured services—rather than physical hardware interaction. The workflow that defines hardware security analysis—probing pins with a multimeter, distinguishing TX from RX by impedance, connecting a serial adapter with correct crossover wiring—remains entirely outside their scope.

The pattern across all these resources is consistent: each excels within its chosen abstraction level but leaves a gap at the boundary between physical hardware and digital analysis. No existing platform bridges this boundary in a browser-accessible format.

3.2 Remote Access and Virtualisation Platforms

Remote laboratories, cyber ranges, circuit simulators, and firmware emulation frameworks represent established approaches to delivering hands-on technical training without shipping physical hardware, each with structural characteristics that are instructive for our design.

3.2.1 Remote Laboratories and Cyber Ranges

Remote laboratories allow students to interact with physical instruments over the internet. iLab (MIT) [4] provides a middleware layer abstracting instrument access behind web service APIs, enabling remote operation of oscilloscopes and signal generators. WebLab-Deusto [33] extends this with FPGA-based experiments, allowing hardware design synthesis on remote boards. VISIR [34] specialises in analog circuit experiments through a virtual breadboard interface connected to real instruments via a relay switching matrix.

These systems deliver authentic experimental experiences, but they share three fundamental limitations for hardware security training. First, infrastructure cost scales linearly with concurrent users—each session requires exclusive access to physical instruments. A university course with 60 students and a two-hour lab window needs either 60 instrument sets or strict time-slotting that limits hands-on time per student. Second, available experiments are fixed by the laboratory administrator; instructors cannot easily reconfigure them for new training scenarios. Third, the focus is electronics and circuit design, not security analysis—the instruments measure signals but do not contextualise them as attack vectors.

Cyber ranges simulate entire network and system infrastructures for security training. KYPO [35] provides a cloud-based platform for multi-stage network security scenarios involving virtual machines, firewalls, and vulnerable services. CyRIS [36] automates scenario instantiation from YAML-based specifications. The DETER testbed [37] at USC/ISI provides shared infrastructure for network security research with real traffic at scale.

These platforms have proven effective for network security and incident response training, and they represent the closest existing analogue to what we propose—both architecturally (web-based scenario delivery) and pedagogically (multi-step guided exercises). KYPO in particular shares several design principles with the PCB-CTF platform: declarative scenario specification, automated environment provisioning, and progressive difficulty. The key difference is one of abstraction level. A KYPO scenario can simulate a compromised IoT device at the network layer—an SSH session into a vulnerable router, a network capture showing anomalous traffic—but it cannot simulate the physical process of discovering that the device has an exposed UART header in the first place. The analyst’s workflow begins well before the first SSH session: it begins when they open the device enclosure and examine the PCB. This is precisely the layer that cyber ranges leave unaddressed.

It is worth noting that the technologies underlying cyber ranges—containerised scenario deployment, YAML-based configuration, web-based frontends—are mature and well-documented. The PCB-CTF platform benefits indirectly from this maturity: the Next.js framework, Docker containerisation, and JSON-based configuration patterns we adopt are proven in similar contexts, reducing implementation risk.

3.2.2 Circuit Simulators and Firmware Emulation

Circuit simulation tools approach the problem from the electronics education side. Falstad’s simulator [38] provides real-time interactive simulation of analog and digital circuits in the browser, with animated current flow. Tinkercad Circuits extends this with Arduino simulation. Wokwi [39] supports a broader range of microcontrollers (ESP32, STM32, Raspberry Pi Pico) with realistic peripheral simulation.

These tools are valuable for understanding how embedded hardware works at the electrical level, but they incorporate no security concepts, no CTF mechanics, and no vulnerability analysis workflows. A student can simulate an SPI bus in Wokwi but cannot practise the skill of identifying an SPI flash chip on an unfamiliar PCB and extracting its contents.

Firmware emulation operates at yet another abstraction level. Firmadyne [40] automatically unpacks Linux-based firmware images, configures a QEMU virtual machine with the appropriate kernel, and starts network services for dynamic analysis. FirmAE [41] improves emulation success rates through better kernel matching and NVRAM handling. The Avatar framework bridges emulation and real hardware by forwarding I/O from QEMU to a physical device.

These tools are powerful for advanced firmware analysis, but they assume the firmware has already been obtained. The earlier phases—hardware identification, electrical measurement, interface connection—that are central to a complete hardware security workflow fall outside their design.

3.3 Gap Analysis

The survey above reveals that existing solutions cluster around distinct abstraction levels, each leaving characteristic gaps. Professional conferences offer depth but not scale. Software CTF platforms offer scale but not hardware. Physical CTFs offer hardware but not accessibility. Remote laboratories offer real instruments but not security context. Cyber ranges offer security scenarios but not physical interaction. Firmware emulation frameworks offer analysis depth but skip the extraction workflow.

From this analysis, five structural gaps emerge:

1. **No browser-based PCB-level simulation.** No existing platform provides interactive, browser-accessible simulation of hardware analysis at the PCB level—visual inspection, electrical measurement, protocol-level instrument interaction—in a unified environment.
2. **High cost and logistical barriers.** Professional courses require budgets of thousands of euros. Physical CTFs require hardware distribution. Remote laboratories require server infrastructure with limited concurrent capacity. The cost barrier is particularly acute for the student and self-learner populations that the exploitability factors identified in Section 2.4.3 suggest would benefit most from hands-on training.
3. **Architecture-specific scope.** Platforms like Microcorruption [7] are tied to a single processor architecture (MSP430), limiting the breadth of skills developed. Modern IoT devices span ARM, MIPS, RISC-V, and proprietary architectures. A training platform should be device-agnostic, configurable for any target.
4. **Absence of no-code exercise authoring.** Existing hardware CTF platforms offer no visual, code-free tools for instructors to create new exercises. Content

creation requires programming expertise and familiarity with platform internals—a bottleneck that limits the supply of training material.

5. **No integration across abstraction levels.** The most significant gap is not within any single category but between them. No existing solution connects physical board inspection to serial communication to firmware analysis to vulnerability discovery in a single, coherent learning experience. Students must assemble their own toolchain across multiple disconnected platforms, losing the pedagogical thread that ties the hardware analysis methodology together.

3.4 Positioning of This Work

Table 3.1 positions the PCB-CTF platform relative to existing solutions across the dimensions identified above.

Table 3.1: Comparison of existing platforms and the proposed PCB-CTF system.

Feature	CTFd	RHme	Microcorr.	Rem. Labs	Cyber R.	PCB-CTF
HW simulation	No	No (real)	MCU emu.	Partial	No	Yes
PCB interaction	No	Physical	No	Webcam	No	Virtual
Instruments	No	No (real)	No	Real	No	Simulated
Terminal	No	No	Yes	SSH	SSH	Simulated
Browser access	Yes	No	Yes	Partial	Partial	Yes
Cost / student	0	50+ EUR	0	Infra.	Infra.	0
Scalability	High	Low	High	Low	Medium	High
No-code authoring	Partial	No	No	No	Partial	Yes
Deterministic	N/A	No	Yes	No	No	Yes
Server risk	N/A	N/A	N/A	Yes	Yes	None
Target audience	Broad	Expert	Interm.	Students	Prof.	All
Pedagogical support	Minimal	None	Minimal	Guides	Scenario	Guided

Several patterns emerge from the comparison. First, no existing platform combines PCB-level visual interaction with simulated instruments and an embedded terminal in a single browser-based environment. CTFd and PicoCTF—the most mature and widely deployed platforms—are content-management frameworks for software CTFs; they provide flag submission and scoring infrastructure but no hardware simulation capability whatsoever. Adding hardware challenges to these platforms would require building the simulation layer from scratch, which is precisely what the PCB-CTF platform provides.

Second, the platforms that do involve real hardware face inherent scalability limitations. RHme ships custom Arduino-based boards to registered participants, limiting each competition edition to a few hundred teams. The CSAW Embedded Security Challenge [31] follows a similar model. Remote laboratories (iLab, WebLab-Deusto, VISIR) provide genuine instrument access but require one physical setup per concurrent session—a cost that scales linearly with class size. These are excellent tools for institutions that can afford the infrastructure investment, but they are not viable paths toward democratising hardware security education.

Third, the existing browser-accessible options leave critical phases of the hardware analysis workflow unaddressed. Microcorruption [7] provides an elegant web-based debugger for MSP430 exploitation, but its scope begins after firmware has been loaded into the emulator. There is no physical board, no pin probing, no serial connection, and no firmware extraction step. The student learns to exploit a running binary but not how to obtain it in the first place—a skill that, in practice, constitutes the larger part of a hardware security engagement. Cyber ranges like KYPO [35] and CyRIS [36] operate

at the network layer: students interact via SSH with virtual machines, not with PCB images or instrument widgets.

Fourth, pedagogical support varies dramatically across platforms. Most CTF frameworks provide only a challenge description and a flag submission box; the student receives no guidance on methodology, no intermediate feedback, and no progressive disclosure of concepts. The PCB-CTF platform takes a different approach: exercises are structured as guided sequences of steps with contextual instructions, hint systems, and progressive flag fragments that reward each intermediate discovery. This design is informed by evidence that scaffolded learning environments produce better outcomes than unstructured exploration, particularly for novice learners [3].

Finally, none of the surveyed platforms offers a no-code authoring interface for hardware security exercises. CTFd supports challenge creation through a web form, but this covers only flag submission metadata—there is no tool for configuring instrument simulations, virtual filesystems, or boot sequences. The content creation barrier is therefore a fundamental bottleneck: even when the platform infrastructure exists, the difficulty of producing new exercises limits the breadth and freshness of available training material.

The design of the PCB-CTF platform navigates a three-way trade-off between *accessibility*, *realism*, and *reproducibility*:

- **Accessibility vs. realism.** Physical laboratories maximise realism—real signals, real noise, real failure modes—but restrict access to those who can be physically present with funded equipment. We sacrifice electrical fidelity (no capacitance, inductance, or signal integrity simulation) in exchange for zero-cost, zero-installation browser access. The exploitability factors described in Section 2.4.3—exposed debug interfaces, unencrypted flash, hardcoded credentials—can be faithfully represented without analogue simulation, because they are fundamentally digital properties of the device’s configuration and firmware.
- **Realism vs. reproducibility.** Physical devices exhibit unit-to-unit variation, component aging, and wiring mistakes that make each student’s experience slightly different. For assessment and demonstration, this is a liability. The PCB-CTF platform is fully deterministic: every student encounters identical pin voltages, identical firmware contents, and identical terminal responses, enabling fair grading and reproducible classroom demonstrations.
- **Reproducibility vs. content creation.** Deterministic, configurable exercises require an authoring workflow. Existing platforms either hard-code their content or require programming expertise to extend. The PCB-CTF no-code configurator allows instructors with hardware security expertise but without web development skills to create, modify, and share exercises—addressing the content bottleneck that limits hardware security education at scale.

It is important to state clearly what the platform does *not* claim to provide. It does not simulate full electrical complexity. It does not support arbitrary code execution on a real embedded target. It does not cover advanced techniques such as side-channel analysis, fault injection, or hardware trojan detection, which require specialised instrumentation beyond what browser-based simulation can meaningfully reproduce. These boundaries are deliberate: the platform targets the foundational skills that constitute the bulk of a hardware security analyst’s daily practice—board inspection, interface identification, serial communication, firmware analysis, and common vulnerability patterns—skills that

the exploitability factors of Section [2.4.3](#) confirm are both the most prevalent in real devices and the most underserved by existing training solutions.

Chapter 4

Requirements Analysis

This chapter formalises the requirements that drive the design of the PCB-CTF platform. We begin by outlining the motivations for an accessible and reproducible hardware security training environment, then describe the system’s actors and define the functional and non-functional requirements that shape the architecture presented in Chapter 5.

4.1 Motivations

The gaps identified in Chapter 3 point to a clear need for a training solution that satisfies three properties simultaneously: *realism* (the training experience should approximate real hardware analysis), *accessibility* (the platform should be usable by anyone with a web browser, at no cost), and *reproducibility* (exercises should produce deterministic, repeatable outcomes). Existing solutions optimise for at most two of these three properties.

The present work aims to demonstrate that a web-based simulation can achieve a useful degree of realism—sufficient for developing foundational hardware security skills—while fully satisfying the accessibility and reproducibility requirements. The platform does not claim to replace physical laboratories for advanced training. Rather, it provides an accessible entry point that lowers the barrier to a field traditionally gated by equipment costs and institutional resources. We believe this positioning is both realistic and useful: a student who has practised component identification, multimeter measurements, UART crossover wiring, and embedded system navigation in the simulator approaches a physical laboratory session with better conceptual preparation and greater confidence, reducing the time and supervision required for hands-on training.

The choice of a Capture The Flag format is motivated by its proven effectiveness in cybersecurity education. Gamified learning structures—progressive challenges, flag-based scoring, immediate feedback—have been shown to increase student engagement and knowledge retention compared to traditional laboratory exercises [27, 42]. Hardware security CTFs such as RHme [6] and CSAW ESC [31] have demonstrated this motivational effect in practice, but their physical nature limits participation to students with access to dedicated hardware kits. By bringing CTF mechanics into a browser-based simulation, we combine the engagement benefits of gamification with the reach of web-based learning.

An equally important motivation is the need to empower content creation by educators who may lack web development skills. Creating interactive educational content typically demands both domain expertise and software engineering proficiency—a combination that is rare and creates a significant bottleneck in pedagogical innovation. The vulnerability analysis conducted by Genova [1] on three consumer IoT devices (TP-Link WR841N router, Ezviz C6N camera, Xiaomi Mi Router 4C) produced a rich corpus of documented findings—boot logs, UART transcripts, firmware dumps, filesystem structures, hardcoded

credentials—that can serve as raw material for training exercises. However, transforming this raw material into structured CTF challenges has historically required programming effort that domain experts cannot justify. The knowledge of what to teach exists in the hardware security community, but the means to package it into interactive experiences does not. The no-code authoring interface proposed in this thesis addresses this bottleneck directly, enabling an instructor to compose a complete exercise—from PCB layout to terminal simulation—through visual editors and structured forms, without writing a single line of code.

4.2 Operational Context and Actors

The platform serves two distinct user roles:

- **Student.** The primary user of the simulation environment. The student interacts with the virtual PCB, uses the simulated instruments, and progresses through the exercise by completing objectives and discovering flag fragments. No prior experience with the platform is assumed; contextual instructions and hints guide the student through each step.
- **Author (Instructor).** The user responsible for designing and configuring exercises. The author uses the settings interface to upload a PCB image, place components and measurement pins, define exercise steps and objectives, configure the terminal environment, and manage exercise presets. The author is assumed to have domain expertise in hardware security but not necessarily in web development.

In the current version, the platform operates in a single-user mode: each instance serves one student at a time, with no shared state between sessions. This design choice simplifies the architecture and eliminates the need for user authentication and access control, which are deferred to future work (Section 8.5).

The separation between these two roles has an important architectural consequence: the entire authoring subsystem can be disabled in production without affecting the student experience. A production deployment exposes only the simulator route (`/`), while the authoring panel (`/settings`) is accessible exclusively in development builds. This separation ensures that the exercise configuration is immutable from the student’s perspective, preventing accidental or malicious modification of the training content during a session. It also simplifies the security model: since students interact only with client-side simulation logic and never with the authoring API, the attack surface of a production deployment is minimal.

The design also anticipates a future multi-instance deployment model in which a single author prepares exercises on a development instance and exports them as portable presets, which are then deployed across multiple read-only student instances. This workflow is already supported by the preset system (Section 6.5.3): the author creates a preset, downloads the JSON file and the associated PCB image, and distributes them to any number of platform instances. No database synchronisation, shared storage, or network connectivity between instances is required.

4.3 Functional Requirements

The functional requirements are organised by subsystem and expressed below as prose paragraphs. Each requirement retains a unique identifier (RF-*nn*) for traceability against the architecture and implementation chapters.

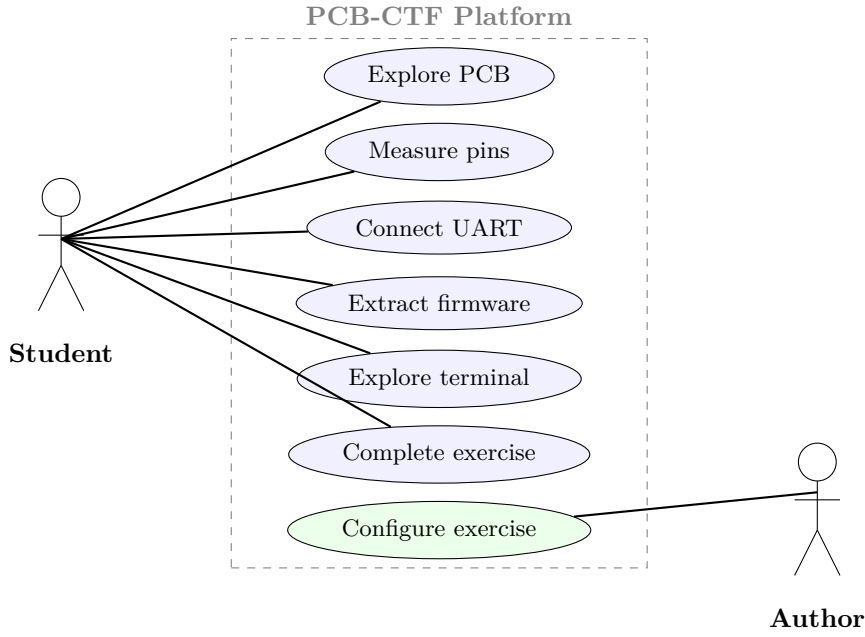


Figure 4.1: Use case diagram showing the interactions of the two actors (Student and Author) with the PCB-CTF platform.

4.3.1 PCB Simulation

The simulator displays a high-resolution image of a printed circuit board as the central interactive canvas (RF-01). Students identify hardware components by clicking on predefined regions of this image, with the system highlighting the selected component and providing contextual information (RF-02). To support detailed inspection of densely populated boards, the platform provides a magnifying lens tool that renders a zoomed view of the area under the cursor, with configurable radius and zoom level (RF-03).

4.3.2 Measurement and Connection Tools

The platform simulates the physical instruments commonly used in hardware security assessments. A digital multimeter with two probes (red and black) supports voltage and resistance measurement modes, allowing students to practice identifying active pins and characterising electrical properties (RF-04). A USB-to-Serial adapter with TX, RX, and GND pins simulates UART connectivity, enforcing the crossover convention—adapter TX connects to device RX and vice versa—so that students internalise correct wiring practice (RF-05). For firmware extraction scenarios, an SPI flash programmer with six colour-coded probes (VCC, GND, CS, CLK, MOSI, MISO) validates role-based connections against the target device’s pinout (RF-06). All probe-based tools benefit from a snap-to-pin mechanism that automatically aligns probes to nearby measurement points within a configurable radius, reducing frustration from imprecise mouse placement (RF-07).

4.3.3 Simulated Terminal

The terminal subsystem provides an in-browser terminal emulator that displays command output and accepts user input (RF-08). It operates on a virtual filesystem composed of directories and files, navigable through standard commands such as `ls`, `cd`, `cat`, `pwd`, `grep`, and `find` (RF-09). To recreate the experience of interacting with an embedded

device, the terminal supports configurable boot sequences with animated stage transitions that simulate bootloader and kernel startup (RF-10). Flag discovery is woven into the terminal interaction through a progressive flag system, where individual flag fragments are unlocked as side effects of specific commands (RF-11). A single exercise can include multiple terminal tabs—for instance, a UART console and a local analysis workstation—each with its own independent environment (RF-12). The system enforces command-level constraints based on the current working directory, permission context, prerequisite conditions, argument patterns, and boot stage, ensuring that students follow a realistic sequence of operations (RF-13).

4.3.4 Exercise Structure

Exercises are organised as an ordered sequence of steps, each containing one or more objectives that the student must complete before advancing (RF-14). The platform supports five distinct objective types: component identification, pin measurement, UART connection, terminal exploration, and firmware extraction (RF-15). As objectives are completed, their associated flag fragments are concatenated to build the exercise flag progressively, giving students continuous feedback on their advancement (RF-16). Upon completion of all steps, the platform displays a configurable completion dialog that can provide a download link, a redirect URL, or a copy-to-clipboard action for the assembled flag (RF-17).

4.3.5 Authoring and Configuration

The authoring subsystem enables instructors to create exercises without writing code. A visual editor supports drag-and-drop placement of components and measurement pins on the PCB image (RF-18). A dedicated terminal configuration editor allows the definition of commands, virtual filesystems, boot sequences, and flag systems through structured forms (RF-19), complemented by a terminal preview panel where authors can verify terminal behaviour without leaving the editor (RF-20). The preset system supports saving, loading, updating, and deleting complete exercise configurations—each preset bundles an exercise definition together with its terminal configuration—enabling rapid iteration and sharing of exercises across deployments (RF-21). Authors can also fine-tune built-in tool parameters such as magnifier radius and zoom level, UART connector persistence, and firmware dump probe layout (RF-22). Finally, tool groups determine which instruments are available during each exercise step, allowing instructors to introduce tools progressively and avoid overwhelming students with options irrelevant to the current objective (RF-23).

4.4 Non-Functional Requirements

Beyond its functional capabilities, the platform is subject to a set of quality and architectural constraints that shape design decisions throughout. These non-functional requirements fall into three broad categories: accessibility and deployment, user experience, and maintainability and security.

4.4.1 Accessibility and Deployment

The platform is accessible through a standard web browser without requiring any plugin, extension, or local software installation (RNF-01). Deployment is equally straightfor-

ward: the entire application runs as a single Docker container or a standalone Node.js process (RNF-02).

4.4.2 User Experience

Each objective is accompanied by textual instructions and an optional hint, supporting self-paced learning without instructor intervention (RNF-03). Interactions with probes, connections, and the PCB provide immediate visual feedback—colour changes, animations, snap indicators—so that students always understand the effect of their actions (RNF-04). The interface adapts to different screen sizes, with a minimum supported resolution of 1280×720 pixels (RNF-05).

4.4.3 Architecture and Security

All simulation logic—instrument readings, terminal command execution, flag validation—executes in the browser; the server is required only for initial page load and configuration persistence (RNF-06). Once the exercise configuration has been loaded, the simulation operates without further server communication, ensuring resilience to network interruptions (RNF-07). The codebase is written in **TypeScript** with strict compiler settings to enable compile-time error detection and improve long-term maintainability (RNF-08). The terminal system is configurable through a declarative schema, without requiring code changes for new exercises (RNF-09). Critically, the platform never executes user-supplied commands on the server, eliminating an entire class of remote code execution vulnerabilities (RNF-10). Each browser session operates independently, with no shared mutable state between concurrent users (RNF-11).

4.5 Use Cases

The principal use cases for each actor are summarised below and correspond to the diagram in Figure 4.1.

4.5.1 Student Use Cases

The student begins by exploring the PCB image using the magnifying lens to identify components and read markings (CU-01). This is a deliberate pedagogical choice: before reaching for instruments, the student develops visual literacy—learning to distinguish SoC packages from flash chips, to recognise UART headers by their characteristic 3–4 pin arrangement, and to identify SPI flash by package type. Using the digital multimeter, the student then measures voltage and resistance on PCB pins to locate active interfaces (CU-02). The progression from visual to electrical analysis mirrors professional practice: an experienced analyst does not randomly probe pins but uses visual cues to narrow the search space before taking measurements. Once candidate UART pins have been identified, the student establishes a serial connection using the USB-to-Serial adapter, respecting the TX/RX crossover convention (CU-03). In firmware extraction exercises, the student connects the SPI flash programmer probes to the appropriate pins and initiates a dump (CU-04). With a connection established or firmware extracted, the student explores the embedded system through the simulated terminal, navigating the filesystem, inspecting configuration files, and discovering flag fragments (CU-05). The exercise concludes when the student has progressed through all steps and assembled the complete flag (CU-06). The ordering of these use cases is intentional: it follows the

hardware analysis methodology described in Section 2.1.3, ensuring that the student develops skills in the same sequence they would apply in a real assessment.

4.5.2 Author Use Cases

The instructor uploads a PCB image and configures the visual layout by placing components, measurement pins, and connection points on the canvas (CU-07). This visual approach to exercise authoring is a distinguishing feature: rather than writing JSON or YAML by hand, the instructor sees the PCB image and interacts with it directly, placing elements where they physically exist on the real board. Exercise steps, objectives, and their associated flag fragments are then defined through the structured editor (CU-08). The step editor enforces structural constraints—each step must have at least one objective, each objective must have a type and a flag fragment—that prevent malformed exercises from being saved. The terminal environment—filesystem, commands, boot sequence, and flag system—is configured using the dedicated terminal editor and verified through the preview panel (CU-09). The preview capability is critical for iterative development: the author can modify a command’s output, click preview, and immediately see the rendered terminal response without saving or switching to the simulator. Finally, the instructor saves the complete exercise as a preset for distribution or later refinement (CU-10).

Chapter 5

System Architecture

This chapter describes the architectural design of the PCB-CTF platform. We present the technology stack and the rationale behind each choice, then examine the overall system structure, the data model, the state management strategy, and the design of the terminal simulation subsystem. The focus throughout is on *design decisions*—the *what* and the *why*—while Chapter 6 addresses the concrete implementation of each component.

5.1 Technology Stack

Two constraints dominate the technology selection: all simulation logic must execute in the browser (RNF-06), and the platform must be deployable as a single artifact (RNF-02). Together, these constraints favour a unified JavaScript/TypeScript stack that serves both the frontend application and the backend API from a single process.

- **Next.js 15** (App Router, standalone output mode). Next.js provides server-side rendering, file-system-based routing, and built-in API route handlers within a single project. The App Router architecture organises pages and API endpoints as filesystem conventions (`src/app/page.tsx` for pages, `src/app/api/*/route.ts` for API handlers), eliminating the need for manual routing configuration. The standalone output mode produces a self-contained Node.js server that includes only the runtime dependencies needed for production, making it directly suitable for Docker deployment without a separate backend service or a `node_modules` directory.
- **React 19**. The component model of React enables a declarative, composable UI architecture well-suited to the interactive nature of the platform. React 19 introduces performance improvements relevant to this project, including automatic batching of state updates and concurrent rendering features that help maintain smooth interaction during complex operations such as probe dragging and lens rendering.
- **TypeScript 5**. Static typing enforces data model consistency across the codebase, which is particularly valuable given the complexity of the terminal configuration schema (over 30 interface definitions) and the exercise data structures. The `strict: true` compiler flag activates all strictness checks, including null safety and implicit type prevention, serving as a form of static application security testing (Section 6.7.7).
- **Tailwind CSS 4** and **Radix UI**. Tailwind CSS provides utility-first styling without a component library lock-in. Radix UI supplies accessible, unstyled primitive

components (dialogs, tooltips, alert dialogs) that comply with WAI-ARIA patterns and integrate with Tailwind through the `class-variance-authority` composition library.

- **Zustand 5.** A lightweight state management library chosen over Redux for its minimal boilerplate and direct API. Zustand stores are plain JavaScript objects with actions, avoiding the ceremony of reducers, action creators, and middleware configuration. The library’s subscription mechanism supports inter-store communication (dirty tracking in the preset store) and its `persist` middleware provides automatic `localStorage` synchronisation for the terminal settings store.
- **JSON file persistence.** Exercise and terminal configurations are stored as JSON files on the server filesystem (`src/data/`), avoiding the operational complexity of a database. This is appropriate given the single-user deployment model and has the additional benefit of making exercise configurations human-readable and version-controllable through Git.

5.2 General System Architecture

The platform exposes two routes, corresponding to its two user roles. The root path (`/`) serves the *simulator*, where students interact with the exercise (satisfies RF-01 through RF-17). The `/settings` path serves the *authoring panel*, where instructors configure exercises (satisfies RF-18 through RF-23). The authoring panel is accessible only when the environment variable `NEXT_PUBLIC_DEV_MODE` is set to `true`; in production deployments it returns a 404, ensuring that students cannot access configuration internals.

The simulator presents a three-panel layout: a sidebar for tool selection, a central canvas for PCB or terminal interaction, and a top bar for instructions and flag progress. The PCB viewer and the terminal are mutually exclusive—when a terminal-type objective is active, the central panel switches from the PCB image to the terminal emulator.

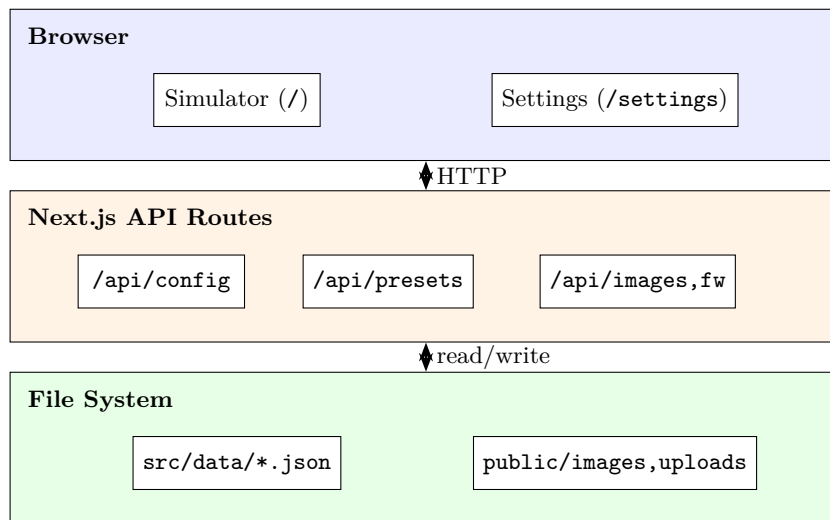


Figure 5.1: High-level system architecture. The browser hosts the simulator and the authoring panel; the Next.js server provides API routes for configuration persistence. All simulation logic executes client-side.

The backend consists exclusively of Next.js API routes whose sole responsibility is configuration persistence. These routes handle exercise and terminal configuration read/write operations (`/api/config/*`, `/api/terminal-config/*`), CRUD operations on

exercise presets (`/api/presets/*`), and file upload and deletion for PCB images, firmware binaries, and hint attachments (`/api/images/*`, `/api/firmware/*`, `/api/hints/*`). No simulation logic resides on the server (RNF-06), and no user-supplied input is ever executed as code (RNF-10).

5.3 Data Model

The platform operates on two principal data structures: the *Exercise* configuration and the *TerminalConfig* configuration. A *Preset* bundles both into a single distributable unit. This section describes the conceptual model; the concrete type definitions and their validation logic are detailed in Chapter 6.

5.3.1 Exercise Configuration

The **Exercise** type captures the complete structure of a training scenario. At its core, the configuration specifies a PCB image path and an ordered sequence of exercise steps, where each step contains a title, a description, the expected flag string, the set of available tools, and one or more objectives. Surrounding this step sequence are the spatial definitions that make the simulation interactive: hardware components visible on the PCB (with names, coordinates, and flag fragments), measurement pins with voltage and resistance values, UART interface pins with role assignments (TX, RX, GND, VCC), and SPI interface pins for firmware extraction scenarios (VCC, GND, CS, CLK, MOSI, MISO). The configuration also includes tool-level parameters (magnifier settings, UART connector behaviour, firmware dumper layout) and an optional completion dialog specification.

Each objective carries a **type** field that determines its completion semantics. A **component** objective is completed when the student clicks the correct PCB region. A **pin** objective requires probes to be connected to pins satisfying specified electrical conditions. A **uart** objective is satisfied when the adapter is correctly wired with the TX/RX crossover. A **terminal** objective requires that all specified terminal flags have been discovered. A **firmware-dump** objective completes when the SPI probes are correctly placed and the dump file is downloaded.

5.3.2 Coordinate System

All spatial coordinates are stored as normalised percentages: a four-element tuple [**left%**, **top%**, **width%**, **height%**] relative to the PCB image dimensions. This percentage-based representation makes the overlay system intrinsically responsive—coordinates remain valid regardless of the rendering resolution or the image file’s native dimensions, and the same snap-detection algorithm operates uniformly across measurement, UART, and SPI pin types. The conversion between percentages and pixel coordinates occurs at render time using the image container’s bounding rectangle.

Objectives that are not spatially bound to the PCB—such as UART connection objectives (which depend on probe-to-pin wiring) or terminal objectives (which take place entirely in the terminal view)—use the placeholder coordinates [0, 0, 0, 0] and are exempt from spatial validation.

5.3.3 Terminal Configuration

The **TerminalConfig** type defines the behaviour of the simulated terminal. It contains metadata (version, name, description), an array of tab configurations, a progressive flag system, and an optional set of global commands shared across all tabs. Each tab

configuration specifies its own virtual filesystem, command definitions, and optional boot sequence, making tabs fully independent environments within the same exercise. This multi-tab design supports scenarios such as a UART console running alongside a local analysis workstation (satisfies RF-12).

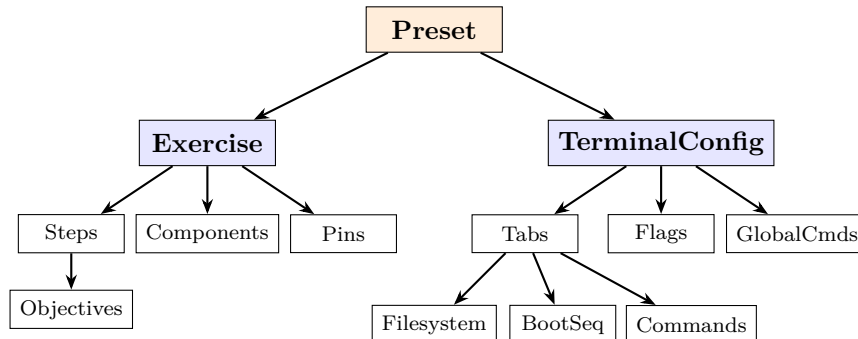


Figure 5.2: Entity-relationship diagram of the main data structures: **Exercise** (steps, objectives, pins, components) and **TerminalConfig** (tabs, commands, filesystem, boot stages, flags). A **Preset** bundles both configurations.

5.4 State Management

The application state is distributed across four **Zustand** stores, each responsible for a distinct concern. This separation reflects the platform’s dual-mode nature: the simulator and the authoring panel operate on different state shapes and have different persistence needs.

The **exerciseStore** holds the runtime simulator state: current step and objective indices, tool selection, probe connections, flag progress, terminal discovery tracking, and firmware dump status. This store is transient—its state is not persisted across page reloads, since each simulation session starts fresh. The **settingsStore** manages the authoring panel state: draft components, pins, steps, objectives, canvas transforms (zoom, pan, rotation), and image references. Its state is persisted both to **localStorage** and to the server filesystem via API calls. The **terminalSettingsStore** holds the terminal configuration being edited, including tabs, commands, filesystem entries, boot stages, and flag parts; it uses **Zustand**’s **persist** middleware for automatic **localStorage** synchronisation (satisfies RNF-09). Finally, the **presetStore** manages the list of available presets, the active preset identifier, loading state, and dirty tracking. It coordinates with the other three stores to capture and restore complete configuration snapshots (satisfies RF-21).

5.4.1 Dirty Tracking

The preset store implements automatic dirty tracking through **Zustand** subscriptions. It monitors a predefined set of content-significant fields in both the settings store and the terminal settings store, using reference comparison—guaranteed by **Zustand**’s immutability contract—rather than deep object comparison. This approach is both efficient and reliable: reference equality holds as long as no mutation has occurred, and fails the instant any content field is updated through a store action. UI-only fields such as canvas zoom, active selection, and drag state are explicitly excluded from the dirty check, so that navigating the canvas does not mark a preset as modified. A guard flag (**isLoading**) prevents the preset load operation itself from triggering a false dirty state.

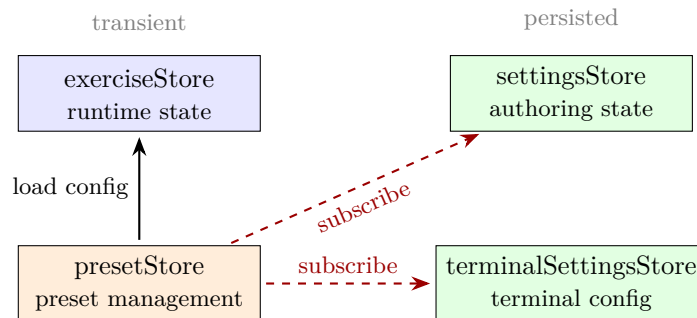


Figure 5.3: Zustand store architecture and inter-store relationships. Dashed arrows indicate subscriptions for dirty tracking; solid arrows indicate data flow.

5.4.2 Persistence Strategy

The platform employs a three-level persistence strategy. The in-memory **Zustand** store is the authoritative source during a session. The **localStorage** layer acts as a client-side cache, ensuring that state survives page reloads without a server round-trip. The server filesystem, accessed through JSON file operations via the API routes, provides canonical storage that persists across browsers and machines.

The terminal configuration follows an additional synchronisation path that merits detailed explanation, as it is the mechanism that enables the live preview workflow central to exercise authoring. When a preset is loaded or the terminal settings are applied, the terminal configuration is serialised to JSON and written to a well-known **localStorage** key (`pcb-ctf-terminal-config`). The simulator page reads this key through a custom React hook (`useTerminalConfig`) that listens for two distinct event types: the browser-native **StorageEvent**, which fires when **localStorage** is modified from a *different* tab, and a custom DOM event (`terminal-config-updated`), which fires when the modification happens in the *same* tab. This dual-listener pattern is necessary because the **StorageEvent** specification explicitly excludes the originating tab—a nuance that, if overlooked, would break same-tab live preview.

The practical benefit is that an instructor can open the authoring panel in one browser tab and the simulator in another, modify a command’s output or add a new boot stage, click “Apply”, and immediately see the change reflected in the simulator without reloading either page. This tight feedback loop reduces the iteration time for exercise authoring from minutes (save, reload, navigate to the relevant step) to seconds (apply, observe), a difference that proved significant during the case study exercise development (Section 7.5).

The three-level architecture also provides a natural recovery mechanism. If the server becomes temporarily unavailable (e.g., during a Docker restart), the **localStorage** layer ensures that the most recent configuration remains accessible. When the server recovers, the next explicit save operation reconciles the local and server-side state. No conflict resolution is needed because the platform operates in a single-author model: there is at most one active editor at any time.

5.5 Terminal Simulation Design

The terminal subsystem is the most architecturally complex component of the platform. Its design is driven by two competing requirements: the simulation must feel sufficiently realistic to be pedagogically useful—boot sequences, login prompts, filesystem navigation—

yet it must execute entirely in the browser without any server-side process (satisfies RF-08 through RF-13, RNF-06, RNF-10).

5.5.1 Design Principles

Three principles guide the terminal’s architecture. First, *client-side execution*: all command parsing, constraint validation, output generation, and side effects run synchronously in the browser, eliminating server security concerns and ensuring deterministic behaviour. Second, *configuration-driven behaviour*: the terminal’s behaviour is entirely determined by its declarative configuration, with no game logic hardcoded in the rendering layer. A new exercise can introduce new commands, new filesystem structures, and new flag unlock conditions without modifying platform code. Third, *modularity*: each exercise can define multiple independent terminal environments (e.g., a UART console and a local analysis workstation), each with its own filesystem, command set, and boot sequence, composed as tabs within the same interface.

5.5.2 Command Execution Pipeline

Command execution follows a four-stage synchronous pipeline. The input string is first *parsed* by splitting on whitespace into a command name and an argument array. The system then performs *constraint validation*, evaluating the command’s constraints in a fixed order—path, permissions, prerequisites, arguments, state—and terminating with an error message at the first unmet constraint. If all constraints are satisfied, the *output generation* stage produces the command’s response according to one of six supported output types: *static* (a fixed array of lines), *conditional* (an if-then-else chain evaluated against the execution context), *template* (a string with `{{variable}}` placeholders resolved at execution time), *lookup* (a table indexed by a command argument, supporting substring, exact, or regex matching), *dynamic* (a call to a registered function), and *script* (a variant of dynamic with a different invocation signature). Finally, the *side effect* stage processes any consequences of the command: flag unlocks (unconditional or condition-gated), state variable changes, and boot stage transitions.

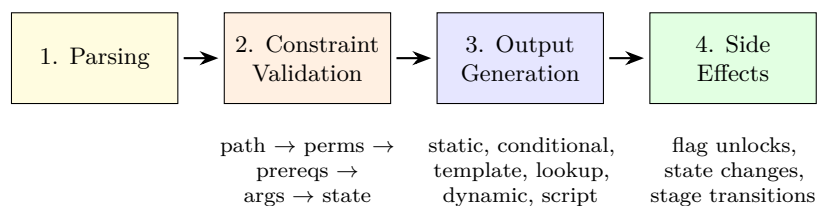


Figure 5.4: The four-stage command execution pipeline. Input is parsed, constraints are validated in fixed order, output is generated according to the command’s type, and side effects are applied.

5.5.3 Filesystem Abstraction

The virtual filesystem is represented conceptually as two parallel dictionaries: a directory map (path → list of entry names) and a file map (path → content or metadata). Authors specify the filesystem in a nested tree notation that mirrors a real directory hierarchy. The configuration loader then flattens this tree into the two-dictionary representation, computing directory entries from file paths. This dual representation—intuitive tree for authoring, flat dictionaries for runtime—balances ergonomics with lookup efficiency.

5.5.4 Boot Sequence System

Boot sequences model the startup behaviour of embedded devices, a critical aspect of realistic hardware security training (satisfies RF-10). Each boot sequence is defined as a chain of stages. A stage specifies output lines, a display duration, an optional prompt, and a reference to the next stage. Special stage types handle U-Boot-style countdown interrupts, username prompts, and password entry with masked input.

Stage transitions can be automatic (after a timeout or after all lines have been displayed) or user-triggered (by entering a command whose side effects include a stage change). An auto-progress timeout allows stages to advance after a configurable period of inactivity, simulating the behaviour of a bootloader that proceeds to kernel boot if no key is pressed.

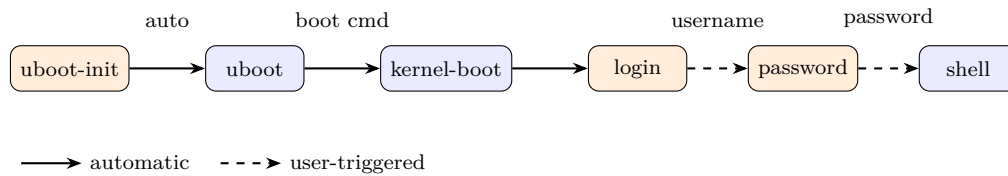


Figure 5.5: Boot sequence state machine for the case study exercise. Solid arrows indicate automatic transitions; dashed arrows indicate user-triggered transitions.

5.5.5 Progressive Flag System

The flag system divides the complete exercise flag into independently discoverable parts, each with an identifier, a text fragment, a description, and an optional hint (satisfies RF-11, RF-16). Parts are unlocked through command-level side effects: when a command executes successfully, its configuration can specify which flag identifiers are revealed, optionally gated by a runtime condition. This design ties flag discovery directly to the learning objectives—a student discovers flag fragments not by guessing, but by performing the correct analysis steps in the terminal. The terminal UI displays a progress indicator showing discovered and pending parts, providing continuous feedback as the student advances.

5.6 Data Flow Between Authoring and Simulation

The configuration lifecycle follows three phases. During *authoring*, the instructor manipulates draft objects in the settings and terminal settings stores through the visual editor. On *persistence*, the stores export their state as JSON objects, which are written to server-side files via API calls and simultaneously cached in `localStorage`. During *simulation*, the simulator page loads the exercise configuration through the `useExerciseConfig` hook (which fetches from the API) and the terminal configuration through the `useTerminalConfig` hook (which reads from `localStorage`). Changes propagated via storage events enable live preview without page reload.

The preset system adds a snapshot mechanism on top of this flow: saving a preset captures the current state of both the exercise and terminal configurations as an immutable bundle. Each preset maintains an independent copy of the PCB image, ensuring that subsequent uploads or transformations do not affect previously saved configurations.

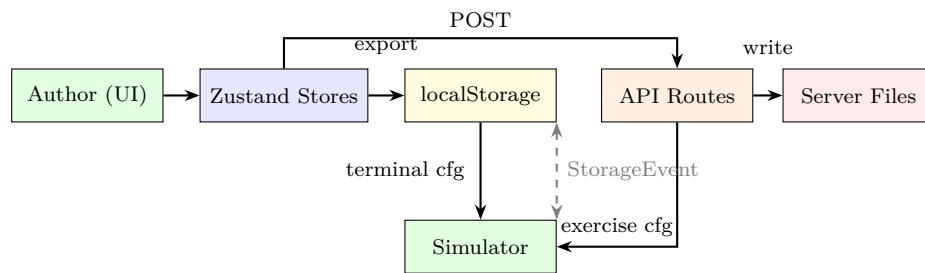


Figure 5.6: Data flow between the authoring panel and the simulator. Configuration changes propagate through three persistence levels and are synchronised via storage events.

Chapter 6

Implementation

We now turn from design to implementation. The sections that follow detail how each subsystem of the PCB-CTF platform is realised in code, with emphasis on the non-obvious engineering decisions, the algorithms behind the interactive tools, and the security measures baked into the platform. Cross-references to Chapter 5 are provided where a design concept is implemented without modification; the focus here is on the concrete techniques, data structures, and code patterns that bring the architecture to life.

6.1 Interactive PCB Viewer

The PCB viewer renders the uploaded board image as a background and overlays interactive elements—component highlight regions, measurement pins, probe wires, and snap indicators—using absolutely positioned HTML elements combined with SVG paths (RF-01).

6.1.1 Coordinate Conversion

As described in Section 5.3.2, all persistent coordinates are stored as percentages of the image dimensions. At render time, the conversion to pixel positions uses the container’s bounding rectangle obtained via `getBoundingClientRect()`:

$$x_{\text{px}} = \frac{x\%}{100} \times w_{\text{container}}, \quad y_{\text{px}} = \frac{y\%}{100} \times h_{\text{container}} \quad (6.1)$$

The conversion runs on every render cycle to account for window resizes and layout changes. The inverse transform (pixel to percentage) is used by the authoring panel when the instructor places elements by clicking on the canvas (RF-18).

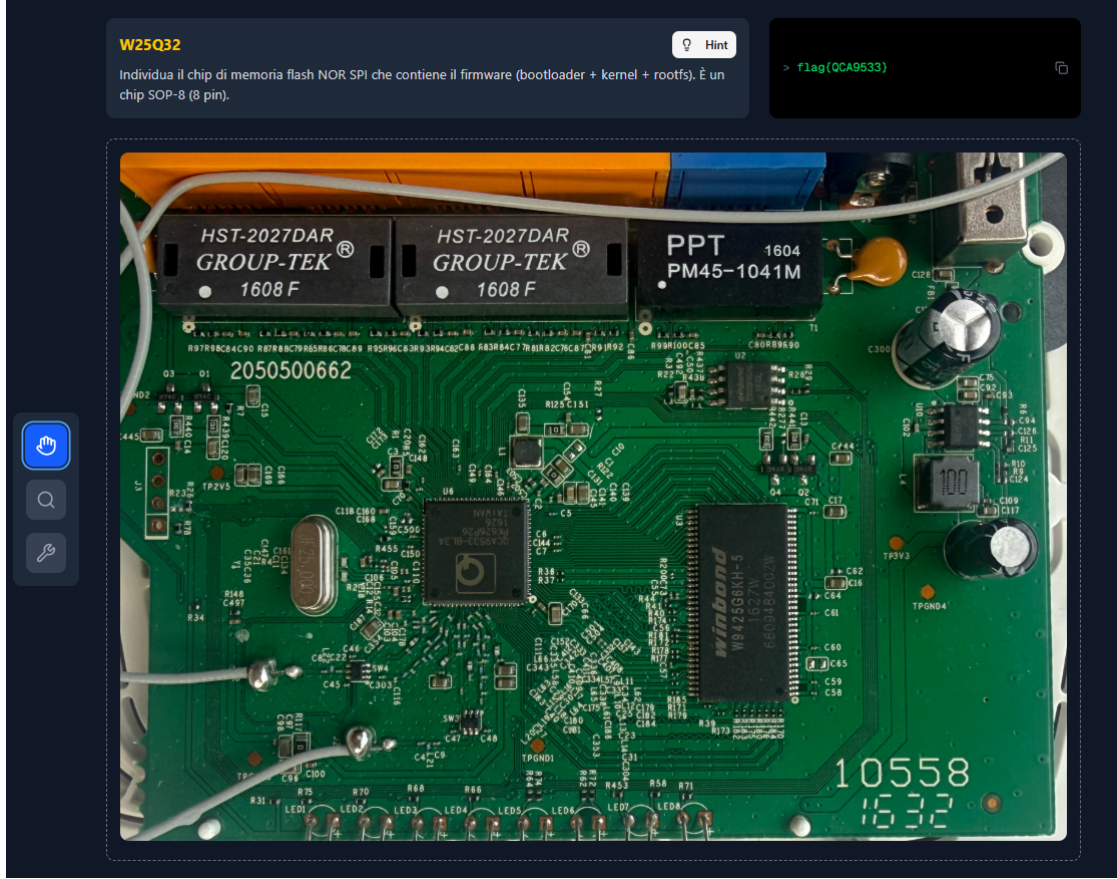


Figure 6.1: The PCB viewer during an active exercise. The sidebar (left) provides tool selection; the central canvas displays the PCB image with overlaid component regions, measurement pins, and probe wires; the top bar shows the current objective and flag progress.

6.1.2 Snap-to-Pin Mechanism

When a probe is dragged, the viewer continuously evaluates the Euclidean distance between the cursor position and every visible pin (RF-07). A pin becomes a snap target when the distance to its centre falls below a dynamic snap radius:

$$r_{\text{snap}} = \max\left(r_{\text{min}}, \frac{d_{\text{pin}}}{2}\right) \quad (6.2)$$

where $r_{\text{min}} = 15 \text{ px}$ is a global minimum and d_{pin} is the rendered diameter of the pin. Larger pins are therefore easier to target, while small pins retain a usable interaction area. Among all candidates within the snap radius, the one with the smallest distance wins; ties between overlapping pins are broken by proximity.

Pin filtering is tool-dependent: the multimeter exposes all measurement pins, the UART adapter shows only UART-role pins, and the firmware dumper restricts candidates to SPI-role pins. Pins that belong to multiple categories (e.g., a shared GND pin used by both UART and SPI) are deduplicated by identifier and flagged with multiple type markers.

Figure 6.4 shows the snap mechanism in action: the red probe connects to a PCB pin after being guided by the snap indicator within the dynamic radius.

6.1.3 Wire Rendering

Probe-to-pin connections are rendered as quadratic Bézier curves in SVG. The control point sits below both endpoints to produce a natural “hanging wire” appearance (RNF-04):

$$P_{\text{control}} = \left(\frac{x_1 + x_2}{2}, \max(y_1, y_2) + 60 \right) \quad (6.3)$$

Wires pass through three visual states: dashed and semi-transparent while a probe is being dragged, solid upon connection, and colour-coded by probe identity (red/black for multimeter, green/yellow/grey for UART, role-specific colours for SPI).

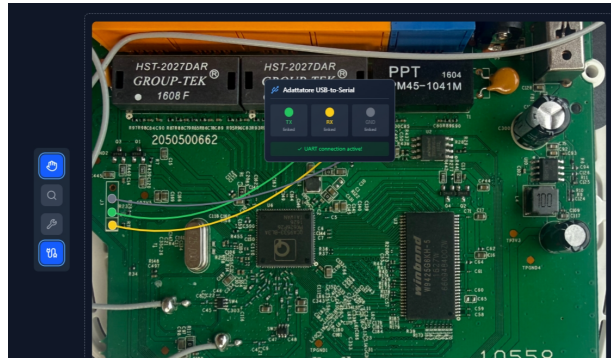


Figure 6.2: Bézier wire rendering illustrated through a UART connection. Solid wires connect the adapter pins (TX, RX, GND) to PCB pins, with the control point positioned below the endpoints to produce a natural curve.

6.1.4 Magnifying Lens

The magnifying lens (RF-03) provides a zoomed view of the PCB region under the cursor. It is implemented as a circular CSS element whose `background-image` property references the same PCB image at a scaled position:

$$\text{bg-pos}_x = -(x_{\text{cursor}} \times z - r), \quad \text{bg-pos}_y = -(y_{\text{cursor}} \times z - r) \quad (6.4)$$

where z is the zoom level and r is the lens radius. A dedicated content sublayer renders wires, probe endpoints, and snap indicators within the zoomed viewport. Visibility testing relies on Bézier curve intersection against the circular lens boundary, approximated with 10 samples along the quadratic curve.

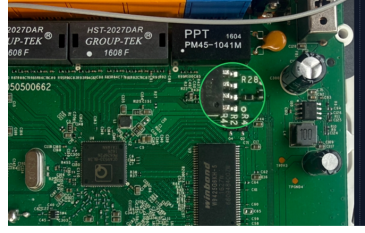


Figure 6.3: The magnifying lens overlaid on the PCB image, showing a zoomed view of a component region. Wires and snap indicators are correctly rendered within the lens viewport.

6.2 Simulated Hardware Analysis Tools

6.2.1 Digital Multimeter

The multimeter widget (RF-04) is a draggable panel displaying a simulated LCD reading. Two measurement modes are supported:

- **Voltage mode (V):** With a single probe connected, the display shows the absolute voltage of the contacted pin. With both probes connected, the display shows the differential voltage $V_{\text{red}} - V_{\text{black}}$, preserving sign.
- **Resistance mode (Ω):** With a single probe, the display shows “OL” (open circuit). With both probes connected, the display shows $R_1 + R_2$, where R_1 and R_2 are the resistance values of the contacted pins. Values exceeding $99,999 \Omega$ are displayed as “OL”.

A digital noise component of $\pm 0.02 \text{ V}$ (or $\pm 0.02 \Omega$), refreshed every 400 ms, simulates the last-digit instability characteristic of real digital multimeters. Pin electrical characteristics follow role-based models: UART TX pins idle at 3.3 V with approximately 65Ω output impedance (CMOS driver characteristic), while UART RX pins idle at 3.3 V with approximately $47 \text{ k}\Omega$ input impedance (pull-up resistor). SPI pins follow analogous role-based electrical models. These values produce pedagogically meaningful readings that guide the student toward correct interface identification.

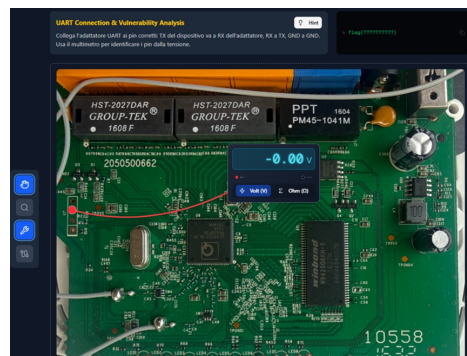


Figure 6.4: The simulated digital multimeter displaying a voltage reading with realistic last-digit noise. The red and black probe indicators show connection status.

6.2.2 UART Adapter

The UART adapter (RF-05) simulates a USB-to-Serial converter with three active pins: TX, RX, and GND. The student connects each adapter pin to the corresponding PCB pin, respecting the crossover convention: adapter TX to device RX, adapter RX to device TX, GND directly to GND.

The validation logic checks that all three connections are established and that the role mapping is correct. Upon successful connection, the platform optionally auto-launches a terminal component, simulating the moment when the analyst gains serial console access to the target device.

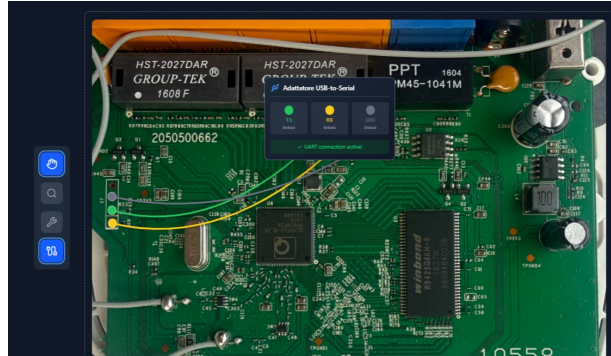


Figure 6.5: The UART adapter widget with three probes (TX, RX, GND) connected to the PCB. The status bar shows “Connected” after successful crossover validation.

6.2.3 SPI Firmware Dumper

The firmware dumper (RF-06) simulates a flash memory programmer with six probes, one for each SPI signal: VCC, GND, CS (Chip Select), CLK (Clock), MOSI (Master Out, Slave In), and MISO (Master In, Slave Out). Each probe is colour-coded and must be connected to the corresponding role on the PCB.

Validation supports two strategies: role-based matching (the probe’s role must match the pin’s role) and explicit mapping (a predefined list of required probe-to-pin connections). The role-based approach is the default in the current implementation for its flexibility.

Upon successful connection and initiation, the dumper displays an animated progress bar with a simulated hex address counter, advancing from 0x000000 to 0xFFFFFFFF over a configurable duration. Upon completion, a download button offers the pre-uploaded firmware binary file.



Figure 6.6: The SPI firmware dumper during a dump operation. Six colour-coded probes are connected to the flash chip. The progress bar shows the simulated read address advancing.

6.3 Terminal Engine

The terminal subsystem implements the architecture described in Section 5.5. We concentrate here on the concrete classes, their internal logic, and implementation-specific details not covered by the design chapter.

6.3.1 Configuration Loader

The `TerminalConfigLoader` class normalises and validates the terminal configuration at load time through a four-phase process:

1. **Tree flattening.** If the filesystem uses nested tree notation, the loader recursively traverses it to produce flat `directories` and `files` records.
2. **File normalisation.** String-valued file entries (a convenient authoring shorthand) are expanded to full `FileNode` objects with default permissions (`-rw-r--r--`) and owner (`root`).
3. **Directory computation.** Directory entries are auto-generated by scanning the keys of both dictionaries and extracting direct children.
4. **Command resolution.** Global commands and tab-specific commands are merged, with tab-level default constraints applied through a shallow merge—command-level fields take precedence.

6.3.2 Command Executor

The `CommandExecutor` class implements the four-stage pipeline described in Section 5.5.2. Constraint validation proceeds in a fixed order, failing fast on the first unmet constraint (RF-13). The output generation stage dispatches to type-specific generators, each receiving the full execution context: command arguments, current path, filesystem state, environment variables, boot stage, discovered flags, command history, and state variables.

Seven built-in command handlers provide standard filesystem operations (RF-09): `ls` (with `-l` and `-a` flags), `cd`, `cat`, `pwd`, `grep` (with regex and case-insensitive support),

`find` (with `-name` and `-type` filters), and `clear`. These handlers operate on the virtual filesystem abstraction and are registered at executor construction time.

Listing 6.1 shows an example command definition that illustrates the interplay between constraints, output generation, and side effects (RF-11). The `strings` command is restricted to the `shell` boot stage, uses a lookup table to produce context-dependent output based on the file path argument, and conditionally unlocks a flag when the student analyses the correct file.

```

1 {
2   "strings": {
3     "description": "Extract printable strings from binary",
4     "handler": "custom",
5     "constraints": {
6       "state": { "bootStage": "shell" }
7     },
8     "output": {
9       "type": "lookup",
10      "argIndex": 0,
11      "matchType": "contains",
12      "table": {
13        "mtdblock3": [
14          "admin_pass=soheadadmin",
15          "wifi_password=MyW1F1P@ssw0rd!"
16        ],
17        "httpd": [
18          "execFormatCmd(%s)",
19          "system(cmd_buffer)"
20        ]
21      }
22    },
23    "sideEffects": {
24      "unlockFlags": [{
25        "id": "leak",
26        "condition": {
27          "type": "argument",
28          "contains": "mtdblock3"
29        }
30      }]
31    }
32  }
33 }

```

Listing 6.1: Example command definition with lookup output and conditional flag unlock.

6.3.4 Tab Completion

The terminal supports basic tab completion for filesystem paths (RF-09). When the user presses Tab, the component extracts the last whitespace-delimited token from the input buffer, determines the target directory and filename prefix, and queries the virtual filesystem for matching entries. A single match completes the token in-place, with a trailing slash appended for directories. Multiple matches are displayed as a list below the prompt.

6.3.5 State Persistence Across Mounts

The terminal React component may be mounted and unmounted multiple times during an exercise—for example, when the student switches between PCB view and terminal view. To preserve the session, the component uses a module-level `Map` outside the React component tree, keyed by the terminal component identifier. This map stores a `PersistedTerminalState` object containing the output history, current working directory, command history (for arrow-key navigation), and current boot stage for each tab. On mount, the component reads from this map; on each state update, it writes back. Because the map lives outside React’s lifecycle, the session survives unmount-remount cycles without requiring a global state management library for this specific purpose.

Listing 6.2 shows the structure of the persisted state object.

```
1 interface PersistedTerminalState {
2   primaryHistory: HistoryLine[];      // UART tab output
3   stage: string;                      // current boot stage
4   primaryPath: string;                // current working directory
5   primaryCmdHistory: string[];        // command history (arrow keys)
6   secondaryHistory: HistoryLine[];    // Local Machine tab output
7   secondaryPath: string;              // secondary tab working dir
8   secondaryCmdHistory: string[];      // secondary tab cmd history
9 }
```

Listing 6.2: Terminal state persisted across component mount cycles.

6.4 Exercise Completion System

The exercise progression is governed by the `exerciseStore` (Section 5.4), which maintains the current step index, objective index, and a step mode: `education` (introductory text displayed), `active` (simulator enabled), or `completed` (all objectives fulfilled). Each objective type has a distinct completion path (RF-14, RF-15):

- **Component (RF-02).** The student clicks on the correct PCB region. The store compares the click coordinates against the objective’s bounding box and marks the objective as complete on a match.
- **Pin.** The store evaluates current probe connections against the objective’s pin condition array. Conditions can be combined with AND or OR logic—for example, requiring that both probes are connected to specific pins, or that either connection is established.
- **UART (RF-05).** The store verifies that all three UART connections (TX, RX, GND) are established with the correct crossover mapping. The terminal auto-launches if configured.

- **Terminal (RF-11).** The store monitors the `terminalDiscoveries` array, populated as the student unlocks flags through terminal commands. The objective completes when all specified flag identifiers have been discovered.
- **Firmware dump (RF-06).** The store confirms correct SPI role mapping, tracks dump progress, and marks the objective complete only after the student dismisses the download dialog—ensuring they have had the opportunity to save the firmware file.

When all objectives in a step are complete, the step transitions to **completed** mode. A flag validation dialog prompts the student to enter the expected flag for the step (RF-16). If the entered flag matches the `expectedFlag` in the configuration, the next step begins in **education** mode.

The flag displayed to the student is built progressively by concatenating the `flagPart` values of completed objectives. For steps with a terminal phase, the terminal’s progressive flag—discovered parts shown, pending parts masked with `?` characters—is appended to the objective-derived flag parts, providing continuous visual feedback on progress.

Between steps, the exercise store performs a full state reset: all probe connections are cleared, the UART adapter is disconnected, the terminal component is unmounted, and the active tool reverts to the pointer. Each step therefore begins from a clean slate (RNF-11).

6.5 Authoring Panel

The authoring panel (`/settings`) provides a visual interface for exercise configuration (RF-18, RF-19, RF-22). It is organised into a sidebar with tab-based navigation and a central canvas.

6.5.1 Visual Canvas Editor

The canvas displays the uploaded PCB image and lets the instructor place and resize components and pins through mouse interactions. Components are created by dragging a rectangle; pins are placed by clicking. Both element types support repositioning by drag and resizing through handles.

The canvas supports zoom (`Ctrl+scroll`), pan (middle-click drag), and rotation (manual adjustment). All transforms are applied to a CSS wrapper element, preserving the coordinate calculation logic. A minimum component size of $2\% \times 2\%$ prevents accidentally invisible elements.

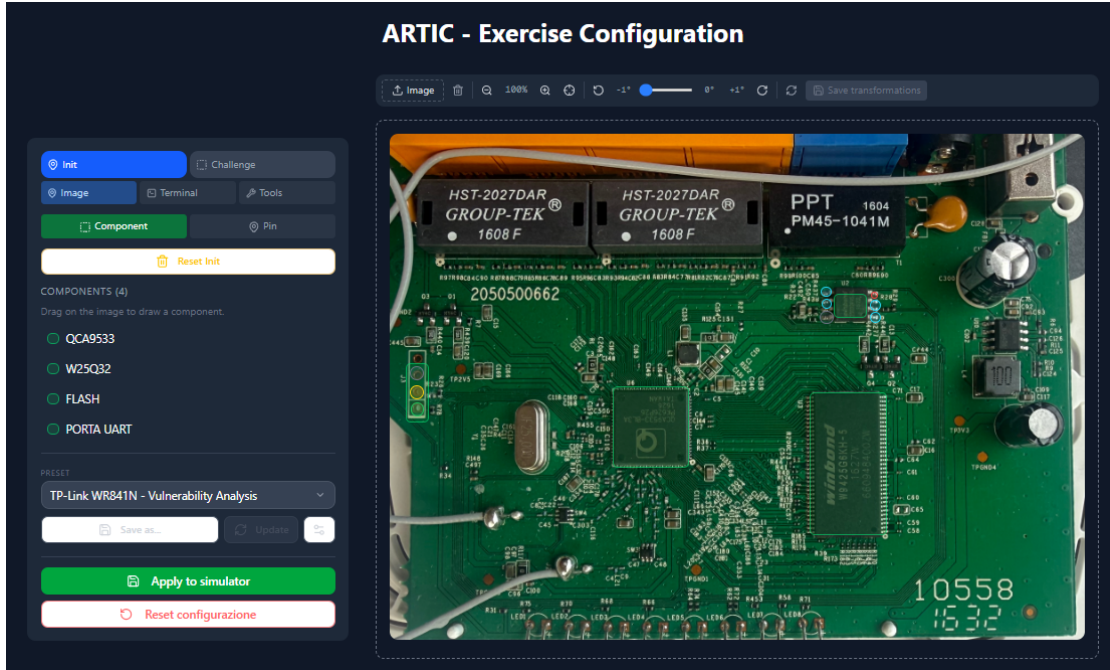


Figure 6.8: The authoring panel canvas editor. Components (blue rectangles) and pins (coloured circles) are positioned on the PCB image via drag-and-drop. The sidebar provides configuration controls for the selected element.

6.5.2 Terminal Configuration Editor

The terminal configuration section of the sidebar provides editors for commands (output type selection, constraint configuration, flag unlock definitions), filesystem entries (tree-view interface), boot sequence stages (duration and prompt configuration), and flag parts (RF-19). An integrated preview panel renders the current terminal configuration in YAML format for verification (RF-20).

6.5.3 Preset Management

The preset system (RF-21) lets instructors save complete exercise snapshots—exercise configuration plus terminal configuration—with a name and description. Presets are stored as JSON files in the `src/data/presets/` directory, with an `index.json` manifest tracking metadata (name, description, creation and modification timestamps). The active preset identifier is persisted in `localStorage` for continuity across page reloads.

When a preset is created or updated, the current PCB image is copied to a preset-specific file (`public/images/preset-{id}.{ext}`), giving each preset an independent copy of its associated image. Without this isolation, uploading a new image for one exercise would retroactively alter previously saved presets. The copy operation is skipped for protected images—default PCB images shipped with the platform (`pcb.jpg`, `pcb_v1.png`, etc.)—which are never modified or deleted.

The preset identifier is generated from the preset name through a slug transformation (lowercase, non-alphanumeric characters replaced with hyphens, truncated to 40 characters) followed by a timestamp-based hash suffix. This guarantees global uniqueness without requiring a database sequence.

Listing 6.3 shows the structure of a preset JSON file.

```

1 {
2   "metadata": {
3     "id": "vuln-wr841n-mmq0k8fjp2z9w",
4     "name": "TP-Link WR841N - Vulnerability Analysis",
5     "description": "Complete walkthrough: HW identification,
6                     UART, vulnerability analysis, SPI dump",
7     "createdAt": "2025-01-15T10:30:00.000Z",
8     "updatedAt": "2025-01-20T14:22:00.000Z"
9   },
10  "exerciseConfig": { /* Exercise object */ },
11  "terminalConfig": [
12    { "id": "tc-uart", "name": "UART Console",
13      "config": { /* TerminalConfig */ } },
14    { "id": "tc-local", "name": "Local Machine",
15      "config": { /* TerminalConfig */ } }
16  ]
17 }

```

Listing 6.3: Preset file structure bundling exercise and terminal configurations.

The terminal configuration field supports both a legacy single-object format and a newer bundle format (an array of `{id, name, config}` entries), providing backward compatibility as the data model evolved from single-terminal to multi-terminal exercises (RF-12).

6.6 API Layer

The backend consists of Next.js API route handlers organised into four groups:

- **Configuration API** (`/api/config/load`, `/api/config/save`): reads and writes the exercise configuration override file (`src/data/exercise.override.json`). The save endpoint auto-generates a step structure from the component list if the input lacks explicit steps.
- **Terminal Configuration API** (`/api/terminal-config/*`): reads and writes the terminal configuration file (`src/data/terminal.override.json`).
- **Presets API** (`/api/presets/*`): full CRUD with index management. Handles the image lifecycle (copy on create/update, delete on remove) and supports both single-config and bundle formats for backward compatibility.
- **Media APIs** (`/api/images/*`, `/api/firmware/*`, `/api/hints/*`): file upload and deletion with filename sanitisation, size limits (20 MB for images, 50 MB for firmware and hints), and path traversal protection.

All upload endpoints prefix filenames with timestamps for uniqueness. All delete endpoints validate that the resolved filesystem path remains within the expected directory, preventing path traversal attacks. No authentication is implemented in the current version, as the authoring panel is intended for local development use and is disabled in production builds.

6.7 Security by Design

Given that the thesis falls within the domain of cybersecurity, we devote particular attention to the security posture of the platform itself. The design and implementation adopt a defence-in-depth approach, addressing the relevant categories of the OWASP Top 10 Web Application Security Risks (2021 edition) [43] through architectural choices, input validation, and secure deployment practices (RNF-10).

6.7.1 Architectural Mitigation: Client-Side Execution

The most consequential security decision is the confinement of all simulation logic—including terminal command execution—to the client side. The server never interprets, compiles, or executes user-supplied commands. This eliminates entire categories of vulnerabilities that plague platforms where user input reaches a shell or interpreter:

- **OS Command Injection (A03:2021 – Injection).** No API route invokes `eval()`, `exec()`, `child_process`, or any other code execution primitive; there is simply no attack surface for command injection. A comprehensive audit of the server-side codebase confirmed zero instances of these functions.
- **Server-Side Request Forgery (A10:2021 – SSRF).** The server makes no outbound HTTP requests on behalf of user input, eliminating SSRF entirely.

The terminal’s `CommandExecutor` operates on pre-registered handler functions and declarative output definitions. Command output is generated through type-safe template substitution and lookup tables—never through interpretation of arbitrary code. The constraint validation pipeline (path, permissions, prerequisites, arguments, state) further restricts command behaviour to the boundaries defined by the exercise author.

6.7.2 Input Validation and Sanitisation

All file upload endpoints implement defence-in-depth input validation:

- **MIME type whitelisting.** The image upload endpoint accepts only `image/jpeg`, `image/png`, and `image/webp`. Requests with other content types are rejected with HTTP 400.
- **File size limits.** Images are capped at 20 MB; firmware and hint files at 50 MB. Oversized uploads are rejected before being written to disk.
- **Filename sanitisation.** All uploaded filenames are processed through a strict regular expression (`/[^\a-zA-Z0-9_-]/g`) that replaces any character outside the alphanumeric, underscore, and hyphen set. A unique timestamp prefix is prepended to prevent collisions and eliminate predictable filenames. This mitigates risks related to special characters, null bytes, and double-extension attacks.

6.7.3 Path Traversal Prevention

All file deletion endpoints enforce a two-layer defence against path traversal attacks:

1. **Path canonicalisation.** The user-supplied path is resolved to an absolute path using `path.resolve()`, which eliminates `..`, symbolic links, and relative path components.
2. **Directory boundary enforcement.** The resolved path is checked against the expected base directory using a `startsWith()` comparison. Requests that resolve outside the boundary (e.g., `../../etc/passwd`) are rejected with HTTP 400.

Additionally, a set of protected image filenames (`pcb.jpg`, `pcb_v1.png`, `pcb_v2.jpg`, `pcb_v3.png`, `pcb_v4.png`) is maintained as a whitelist. Deletion requests targeting these files are unconditionally rejected. Preset-owned images are identified through a strict regex pattern (`/\/images\/preset-[^\s]+\.[^\s]+$/`) and are only deleted as part of preset removal operations.

6.7.4 Access Control

The authoring panel (`/settings`) is protected by a build-time environment flag. The `NEXT_PUBLIC_DEV_MODE` variable is injected at build time and cannot be modified at runtime. In production builds (`.env.production` sets the flag to `false`), the settings page returns HTTP 404 via Next.js's `notFound()` function, making the route effectively nonexistent. This approach is more robust than runtime middleware checks: the flag is embedded in the compiled JavaScript bundle and cannot be tampered with through request manipulation.

6.7.5 Secure Deployment

The Docker deployment configuration implements several hardening measures:

- **Multi-stage build.** The production image is built from a clean `node:20-alpine` base, excluding development dependencies, source code, and build artifacts. This reduces both the attack surface and the image size.
- **Non-root execution.** A dedicated system user (`nextjs`, UID 1001) is created, and all application files are owned by this user. The container runs as this non-root user (`USER nextjs`), limiting the impact of a potential container escape.
- **Environment isolation.** The `.dockerignore` file excludes `.env`, `.env.local`, log files, and editor configuration from the build context, preventing accidental inclusion of sensitive data in the image.
- **Production mode.** The `NODE_ENV=production` variable is set explicitly, disabling development-only features and enabling Next.js production optimisations.

6.7.6 Cross-Site Scripting Prevention

The platform relies on React 19's built-in XSS protection: all JSX expressions are automatically escaped before rendering into the DOM. A codebase audit confirmed zero instances of `dangerouslySetInnerHTML`, the React API that bypasses this protection. Terminal output lines are rendered as plain text within `<div>` elements, and template-based output generation in the `CommandExecutor` uses string replacement (`String.prototype.replace`) rather than HTML parsing, ensuring that no user-controlled content is interpreted as markup.

6.7.7 Static Analysis and Type Safety

The codebase enforces TypeScript strict mode (`"strict": true` in `tsconfig.json`), activating the full suite of compile-time checks: `noImplicitAny`, `strictNullChecks`, `strictFunctionTypes`, `strictPropertyInitialization`, and `noImplicitThis`. These checks serve as a form of Static Application Security Testing (SAST), catching at compile time several classes of bugs that could lead to security vulnerabilities at runtime—null pointer dereferences, type confusion, and implicit `any` escapes that bypass the type system.

Dependency versions are locked through `package-lock.json`, ensuring reproducible builds and preventing supply chain attacks through transitive dependency updates. ESLint with the `next/core-web-vitals` and `next/typescript` rulesets provides an additional layer of static analysis.

6.7.8 Summary of OWASP Top 10 Coverage

Table 6.1 summarises the mapping between OWASP Top 10 categories and the mitigation strategies adopted in the platform.

Table 6.1: OWASP Top 10 (2021) coverage and mitigation strategies.

ID	Category	Mitigation
A01	Broken Access Control	Settings page returns 404 in production via build-time flag
A02	Cryptographic Failures	N/A — no sensitive user data stored
A03	Injection	Client-side terminal; filename regex sanitisation; no <code>eval/exec</code>
A04	Insecure Design	Defence-in-depth: client-only execution, path boundary checks, file type whitelisting
A05	Security Misconfiguration	TypeScript strict mode, non-root Docker user, minimal <code>.env</code>
A06	Vulnerable Components	<code>package-lock.json</code> for reproducible builds
A07	Auth. Failures	Dev mode flag prevents production access to authoring
A08	Integrity Failures	Multi-stage Docker build; no external scripts during build
A09	Logging Failures	Errors logged server-side; no stack traces in API responses
A10	SSRF	No server-initiated outbound requests

Chapter 7

Validation: Case Study

We validate the PCB-CTF platform through a complete exercise based on the hardware analysis of a TP-Link WR841N wireless router. The exercise covers the full spectrum of the platform’s capabilities—visual board inspection, instrument-based measurements, serial interface connection, terminal-based vulnerability analysis, and firmware extraction—and serves as a concrete demonstration that each functional requirement is met in practice. We describe the device selection rationale, the exercise structure, the terminal configuration, a detailed walkthrough of the terminal challenges, and the authoring workflow that produced the exercise.

7.1 Device Selection

The TP-Link WR841N was selected for several converging reasons. First, it is one of the most widely deployed consumer routers globally, with millions of units in service, making it a realistic and relatable training target. Second, it has been extensively analysed in the hardware security literature and was among the devices investigated by Genova [1], providing a well-documented reference for the exercise content. Third, the device exhibits a representative set of vulnerabilities: exposed UART and SPI interfaces, default credentials, unencrypted configuration partitions, and a web server binary susceptible to command injection. These vulnerabilities span a broad range of the attack patterns discussed in Chapter 2.

The device is based on the Qualcomm Atheros QCA9533 SoC (MIPS architecture, 650 MHz), 4 MB NOR flash (Winbond W25Q32), and 32 MB DRAM. It runs a Linux-based firmware with a SquashFS root filesystem and a JFFS2 configuration partition.

7.2 Exercise Structure

The exercise is organised into three formal steps followed by an auto-launched terminal phase (RF-14).

7.2.1 Step 1: Hardware Identification

The first step introduced the student to physical board reconnaissance. Three objectives had to be completed in sequence:

1. **Identify the SoC.** The student used the magnifying lens to inspect the PCB and clicked on the Qualcomm Atheros QCA9533 chip. Upon correct identification, the flag fragment `QCA9533` was revealed. This step validates RF-02 (component identification) and RF-03 (magnification).

2. **Identify the flash memory.** The student located the Winbond W25Q32 flash chip. The flag fragment `_W25Q32` was revealed.
3. **Identify the UART header.** The student located the UART debug header on the board. The flag fragment `PORTA_UART` was revealed.

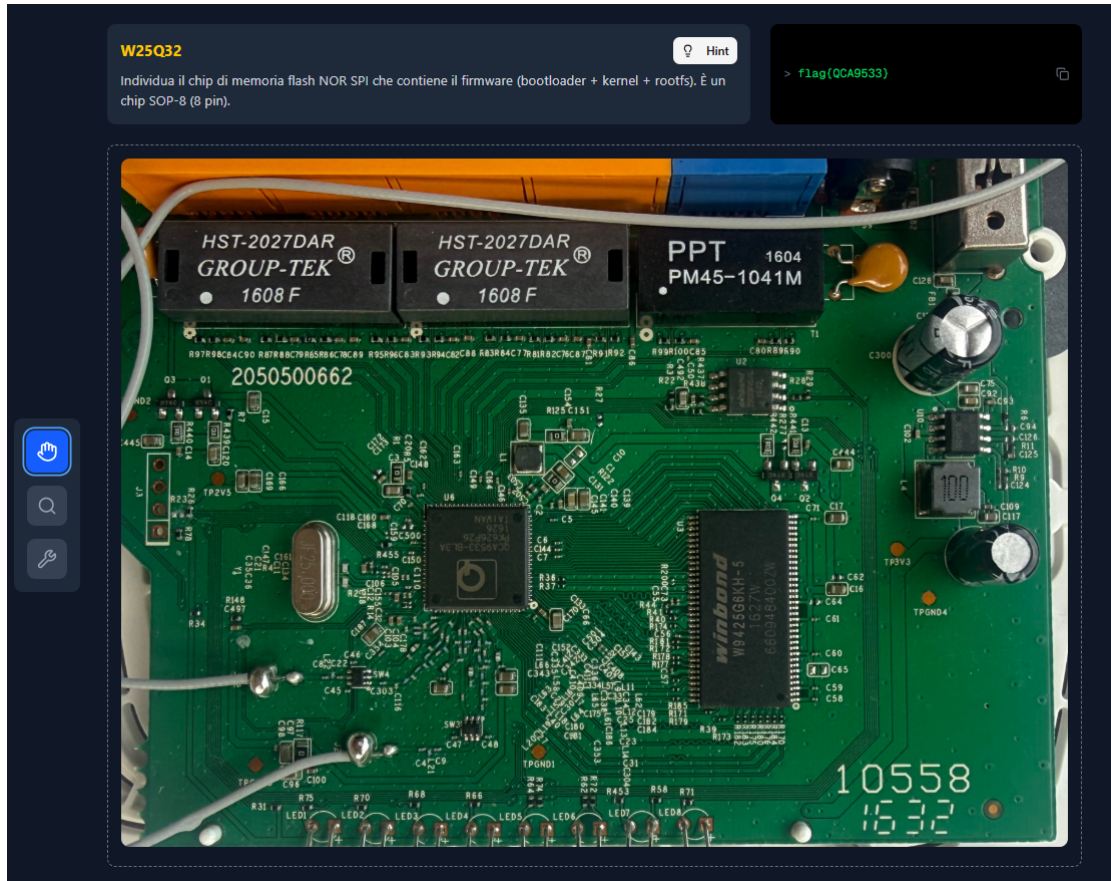


Figure 7.1: Step 1 of the case study exercise: hardware analysis. The student interacts with the PCB image, identifying components and connecting instrument probes to measurement points.

7.2.2 Step 2: UART Connection and Vulnerability Analysis

The second step required the student to establish a serial connection with the device and begin vulnerability analysis. The student selected the UART adapter tool, identified the TX, RX, and GND pins on the PCB (guided by multimeter readings from Step 1), and connected the adapter pins with the correct crossover wiring. This step validates RF-04 (multimeter usage for pin identification), RF-05 (UART crossover wiring), and RF-07 (snap-to-pin mechanism). Upon successful connection, the platform automatically launched the terminal interface, transitioning to the terminal exploration phase where the student analysed the embedded system for vulnerabilities.

7.2.3 Step 3: SPI Firmware Dump

The third step required the student to extract the firmware from the flash chip using the SPI programmer. The student connected the six SPI probes (VCC, GND, CS, CLK, MOSI, MISO) to the corresponding pins on the Winbond W25Q32 chip. Upon correct

role-based connection and successful dump completion, the firmware binary was made available for download. This step validates RF-06 (SPI firmware extraction) and RF-15 (firmware-dump objective type).

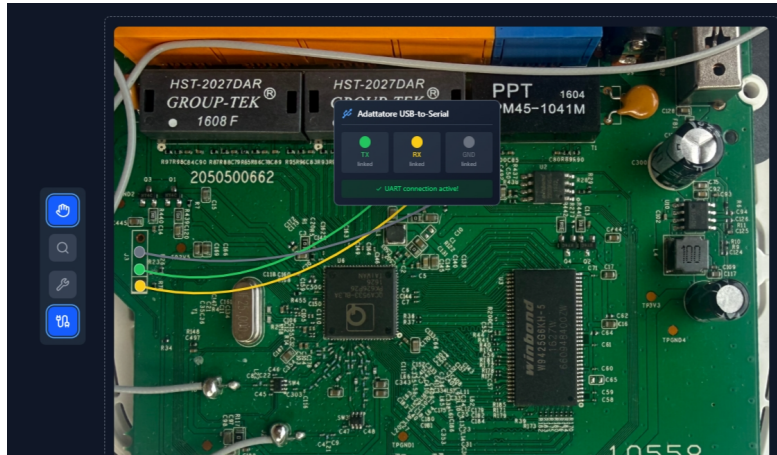


Figure 7.2: Step 2: UART connection. The student has connected the adapter probes with correct crossover wiring (TX→RX, RX→TX, GND→GND). The status bar confirms a successful connection.

7.2.4 Terminal Phase: Embedded System Exploitation

The terminal phase presented six progressive challenges, each revealing a flag fragment (RF-11). The challenges mirror the actual vulnerabilities documented in the WR841N analysis:

1. **Boot Analysis** (b00t). The student executed `printenv` in the U-Boot shell and identified inconsistencies between the configured `rootfstype` variable and the actual filesystem mounted by the kernel.
2. **Root Access** (r00t). After the simulated kernel boot and login prompt, the student gained shell access using the device's default credentials (`root/sohoadmin`), hardcoded in the firmware.
3. **Hash Cracking** (h4sh). The student extracted password hashes from `/etc/shadow` in the UART console tab and cracked the MD5 hash using `hashcat` in the Local Machine tab. This step validates RF-12 (multi-tab terminal support).
4. **Data Leak** (l34k). The student used `strings /dev/mtdblock3` to extract plain-text data from the configuration partition, revealing Wi-Fi credentials and administrative passwords stored without encryption.
5. **Command Injection** (1nj3ct). The student analysed the web server binary using `strings /usr/bin/httpd` and identified calls to functions like `execFormatCmd` that are vulnerable to command injection.
6. **Backdoor Discovery** (sh311). The student examined running processes with `ps` and discovered an anomalous process. Investigation with `strings` on the suspicious binary revealed a planted backdoor.

The complete terminal flag is `flag{b00t_r00t_h4sh_l34k_1nj3ct_sh311}`.

```

UART Console Local Machine
Ram size passed from bootloader =32M
flash_size passed from bootloader = 4
CPU revision is: 00019374 (MIPS 24Kc)
Determined physical RAM map:
 memory: 02000000 @ 00000000 (usable)
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
Memory: 25844k/32768k available (1868k kernel code, 6924k reserved, 448k data, 120k init)
NET: Registered protocol family 16
squashfs: version 4.0 (2009/01/31) Phillip Lougher
Serial: 8250/16550 driver, 1 ports, IRQ sharing disabled
serial8250.0: ttyS0 at MMIO 0xb8020000 (irq = 19) is a 16550A
console [ttyS0] enabled
5 cmdlinepart partitions found on MTD device ath-nor0
Creating 5 MTD partitions on "ath-nor0":
0x000000000000-0x0000000020000 : "u-boot"
0x0000000020000-0x0000000120000 : "kernel"
0x0000000120000-0x00000003e0000 : "rootfs"
0x00000003e0000-0x00000003f0000 : "config"
0x00000003f0000-0x0000000400000 : "art"
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
DES SSID SET=TP-LINK_087C
====>>>wlanBootupAll ok
(none) login:

```

Figure 7.3: The simulated UART console during the boot sequence. U-Boot output is displayed with an autoboot countdown, followed by kernel boot messages and the login prompt.

```

Terminal Challenge Hint
U-Boot bootargs inconsistency
> flag(UART_????_r00
t?????????????????)

UART Console Local Machine
NET: Registered protocol family 16
squashfs: version 4.0 (2009/01/31) Phillip Lougher
Serial: 8250/16550 driver, 1 ports, IRQ sharing disabled
serial8250.0: ttyS0 at MMIO 0xb8020000 (irq = 19) is a 16550A
console [ttyS0] enabled
5 cmdlinepart partitions found on MTD device ath-nor0
Creating 5 MTD partitions on "ath-nor0":
0x000000000000-0x0000000020000 : "u-boot"
0x0000000020000-0x0000000120000 : "kernel"
0x0000000120000-0x00000003e0000 : "rootfs"
0x00000003e0000-0x00000003f0000 : "config"
0x00000003f0000-0x0000000400000 : "art"
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
DES SSID SET=TP-LINK_087C
====>>>wlanBootupAll ok
(none) login: root
Password:
BusyBox v1.01 (2015.06.16-06:24+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.
/ # ls
bin dev etc lib proc root sbin tmp usr var web
/ # cat /etc/shadow
root:$1$GTN.gpr1$D1SykVZKWRGASUj9e9wR3/:1502:0:99999:7:::
/ #

```

Figure 7.4: The terminal shell after successful login. The student executes commands to explore the embedded filesystem. The flag progress bar at the top tracks discovery progress.

7.3 Terminal Configuration

The exercise defines two terminal tabs (RF-12):

- **UART Console.** Simulates the serial console of the WR841N, with a boot sequence progressing through U-Boot initialisation (`uboot-init`), U-Boot console (`uboot`), kernel boot (`kernel-boot`), login prompt, password entry, and root shell (RF-10). The virtual filesystem mirrors the device's actual directory structure (`/etc/passwd`, `/etc/shadow`, `/proc/cpuinfo`, `/dev/mtdblock*`, `/usr/bin/httpd`).

- **Local Machine.** Simulates the analyst’s workstation, providing access to password cracking tools (**hashcat**) and file analysis utilities. This tab starts directly at a shell prompt without a boot sequence.

Custom commands are defined with conditional outputs to simulate context-dependent behaviour (RF-13). The **strings** command, for example, uses a lookup output type: when invoked with **/dev/mtdblock3**, it returns simulated partition content including cleartext credentials; when invoked with **/usr/bin/httpd**, it returns strings suggesting command injection vulnerabilities. Each successful discovery triggers a flag unlock side effect.

Boot stage constraints restrict certain commands to specific stages. The **printenv** command is available only during the **uboot** stage, while filesystem commands (**ls**, **cat**, **cd**) are restricted to the **shell** stage. This enforces a realistic temporal progression through the device’s boot process.

7.3.1 Terminal Challenge Walkthrough

To illustrate the interaction between the terminal configuration and the student experience, we describe the expected progression through the six terminal challenges in detail.

The exercise began with the simulated UART console displaying U-Boot initialisation output: the SoC identification banner (“Booting QCA953x”), memory detection (“DRAM: 32 MB”), and flash chip identification (“flash size 4MB, sector count = 64”). After the initialisation stage, the U-Boot prompt appeared, and the student could execute **printenv** to examine the bootloader environment variables. The output revealed a **rootfstype=ext4** variable that contradicted the SquashFS filesystem actually mounted by the kernel—a deliberate configuration anomaly that mirrors what is observed on the real device. Recognising this inconsistency unlocked the **b00t** flag.

After the simulated kernel boot (auto-progression), a login prompt appeared. The student had to enter the username **root** and password **sohoadmin**—the factory default credentials for TP-Link WR841N devices. These credentials are defined as commands in the terminal configuration: entering the correct username triggered a stage transition from **login** to **password**, and entering the correct password transitioned to the **shell** stage, unlocking the **_r00t** flag.

Once in the shell, the student navigated the virtual filesystem. The **/etc/shadow** file contained a realistic shadow entry with an MD5 password hash (**\$1\$** prefix). The student was expected to copy this hash and switch to the Local Machine tab, where the **hashcat** command was available. Running **hashcat** with the hash as argument produced simulated cracking output and unlocked the **_h4sh** flag. This cross-tab workflow validates RF-12 in a practical scenario.

The data leak challenge required the student to run **strings /dev/mtdblock3** in the UART Console tab. The output included plaintext Wi-Fi credentials and the web administration password—data that on the real device is stored unencrypted in the JFFS2 configuration partition. This discovery unlocked the **_134k** flag.

The command injection challenge was discovered by running **strings** on the web server binary (**httpd**), which revealed function calls such as **execFormatCmd** and direct **system()** invocations with unsanitised input—patterns indicative of command injection. On the real WR841N, this corresponds to CVE-2023-33538. Identifying these patterns unlocked the **_1nj3ct** flag.

Finally, the student ran **ps** to inspect running processes, where an unfamiliar **backdoorTest** process was visible. Running **strings** and **file** on the suspicious binary

revealed that it was a planted backdoor, simulating a supply-chain attack scenario documented by Genova [1]. This discovery unlocked the final `_sh311` flag, completing the terminal challenge with the full flag:

```
flag{b00t_r00t_h4sh_134k_1nj3ct_sh311}
```

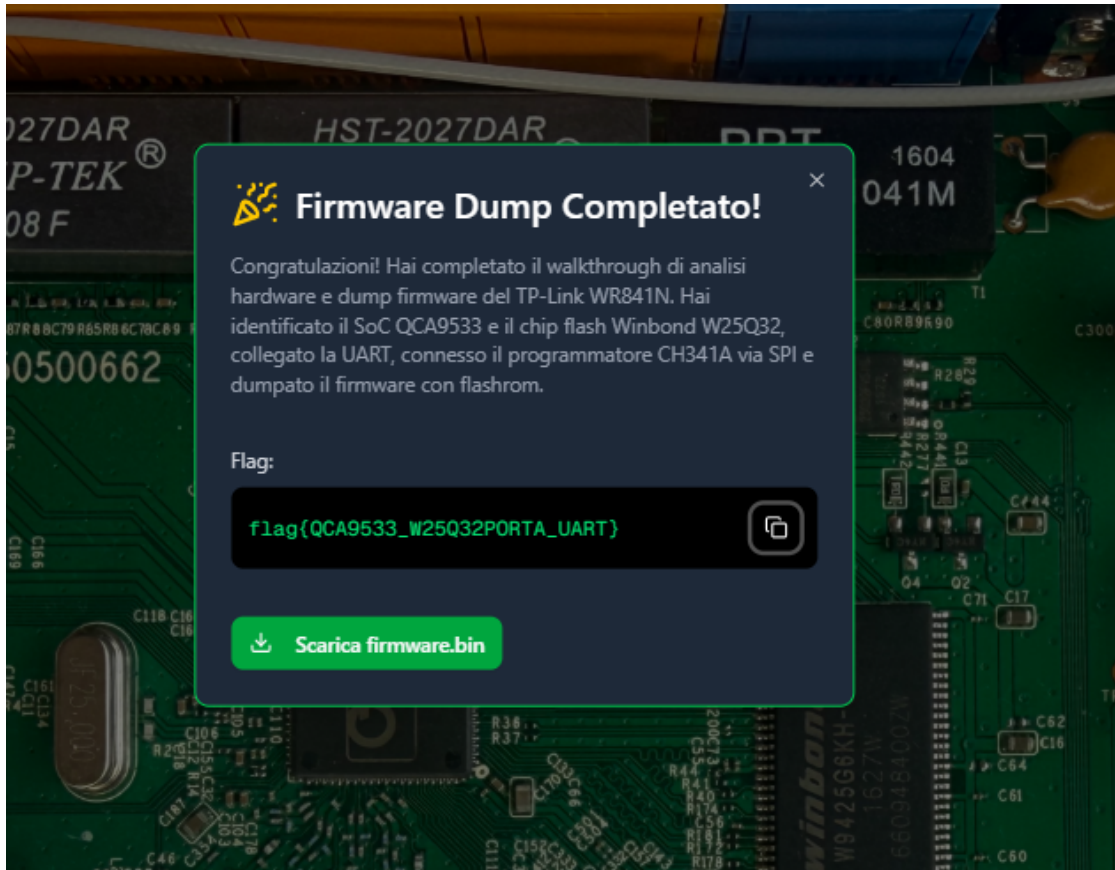


Figure 7.5: The exercise completion dialog, displayed after all six terminal flags have been discovered. The dialog shows the complete flag, a copy-to-clipboard button, and an optional download link for the exercise report.

7.4 Preliminary Observations

Although a formal user study is deferred to future work (Section 8.5), several preliminary observations emerged during the development and internal testing of the case study exercise.

The exercise successfully exercises the full pipeline from physical board inspection to embedded system exploitation, confirming that the platform can host a complete, multi-step hardware security CTF without requiring physical equipment or server-side infrastructure. Each of the six terminal challenges maps to a documented vulnerability on the real WR841N device, demonstrating that the simulation is grounded in authentic attack scenarios rather than artificial puzzles.

The boot sequence simulation proved particularly effective at conveying the temporal structure of embedded device interaction. Testers reported that the U-Boot countdown, kernel boot messages, and login prompt sequence created a convincing sense of interacting with a real device—even though all output was generated client-side from a declarative configuration. The auto-progression mechanism (Section 6.3.3) was essential to this effect:

it reproduced the characteristic “press any key to interrupt” behaviour of U-Boot without hardcoding bootloader-specific logic.

The multi-tab terminal design (RF-12) demonstrated its value in the hash cracking challenge, which required the student to extract a hash from one environment (UART console) and crack it in another (local workstation). Without the tab abstraction, this cross-environment workflow could not have been modelled within a single exercise.

The progressive flag system proved pedagogically effective in ways we had not fully anticipated. Because each flag fragment is named to reflect the underlying vulnerability category (`b00t` for bootloader analysis, `l34k` for data leakage, `1nj3ct` for injection), the partial flag string itself serves as a summary of the student’s progress through the vulnerability taxonomy. A student who sees `flag{b00t_r00t_????_????_????_????}` knows immediately which categories remain to be explored. This is a form of progressive disclosure: the flag does not merely confirm correctness but scaffolds the student’s mental model of the device’s attack surface.

The snap-to-pin mechanism and colour-coded probe wiring also contributed to usability. During internal testing, the visual feedback—probes snapping to valid pins, wires changing from dashed to solid on connection, colour coding matching UART and SPI roles—reduced the number of incorrect connections compared to an earlier prototype that lacked visual affordances. The magnifying lens was particularly useful for identifying small pin groups on the high-resolution PCB photograph, where unaided visual inspection would have been insufficient.

A notable limitation surfaced in the terminal challenges: students who were unfamiliar with the `strings` command or the concept of raw partition inspection struggled with challenges 4 and 5 (data leak and command injection). The hint system partially addressed this gap, but a future iteration could benefit from contextual tutorials that introduce unfamiliar commands before requiring their use.

From a requirements perspective, the case study exercise validates all functional requirements from RF-01 through RF-17. The authoring workflow, described in the next section, validates RF-18 through RF-23.

7.5 Authoring Workflow

The entire exercise was configured through the authoring panel without manual JSON editing (RF-18, RF-19). The workflow proceeded through the following stages:

1. **Image upload.** A high-resolution photograph of the actual WR841N board was uploaded through the media management interface.
2. **Component placement.** Regions were drawn around the SoC and EEPROM using the drag-and-drop canvas editor (RF-18). Each region was assigned a name, a flag fragment, and a description.
3. **Pin placement.** UART and SPI pins were placed at their correct physical positions, with roles (TX, RX, GND for UART; VCC, GND, CS, CLK, MOSI, MISO for SPI) and electrical characteristics (voltage, resistance) assigned through the sidebar form (RF-22).
4. **Step definition.** Three exercise steps were defined, each with its objectives, expected flag, flag fragments, and the set of tools available during that step (RF-23). The first step enabled only the pointer and magnifier; the second added the multimeter and UART adapter; the third added the firmware dumper.

5. **Terminal configuration.** Two tabs were created (UART Console and Local Machine). For each tab, the virtual filesystem was populated through the tree-view editor, custom commands were defined with their output types and constraints, the boot sequence was configured as a chain of stages with durations and prompts, and the six flag parts were registered with their identifiers and text fragments (RF-19, RF-10, RF-11).
6. **Preset save.** The complete configuration was saved as a named preset, bundling the exercise definition, both terminal configurations, and an independent copy of the PCB image into a self-contained package (RF-21).

The complete exercise was configured in approximately five hours, including iterative testing of the boot sequence timing, command output correctness, and flag unlock conditions. The bulk of the effort—roughly three hours—went into the terminal configuration: defining the virtual filesystem to match the real device’s directory structure, crafting realistic command outputs, and tuning the boot sequence durations to feel natural. The remaining two hours covered image upload, component and pin placement, step definition, and the overall exercise structure.

Several aspects of the authoring workflow deserve specific comment. The component and pin placement stage was straightforward: the drag-and-drop interface maps naturally to the task of annotating a PCB image, and the snap-to-grid feedback helped align elements precisely. The step definition stage required more thought, as the author had to decide which tools to make available at each stage and how to sequence the objectives to build a coherent pedagogical narrative—moving from passive observation (magnifier) to active measurement (multimeter) to protocol-level interaction (UART adapter, SPI dumper).

The terminal configuration stage was the most demanding, not because of the interface complexity but because of the content design effort. Crafting realistic command outputs required consulting the actual device analysis data from Genova’s work [1]: boot logs, filesystem listings, `/etc/shadow` entries, and `strings` output had to be adapted from real device dumps into the declarative configuration format. The boot sequence timing required multiple iterations—initial durations were either too fast (breaking the illusion of a real boot) or too slow (frustrating for testing). The final values (3 seconds for U-Boot initialisation, 2 seconds for autoboot countdown, 4 seconds for kernel boot) represent a compromise between realism and usability.

The dirty tracking mechanism in the preset store proved valuable during this iterative process: it alerted the author to unsaved changes when switching between presets, preventing accidental data loss during the frequent save-test-modify cycles that characterise exercise development.

The resulting preset is fully portable: it can be loaded on any instance of the platform to reproduce the exercise identically. No server-side setup, database migration, or manual file copying is required—the preset JSON file and its associated image are the only artifacts needed for deployment (RNF-02).

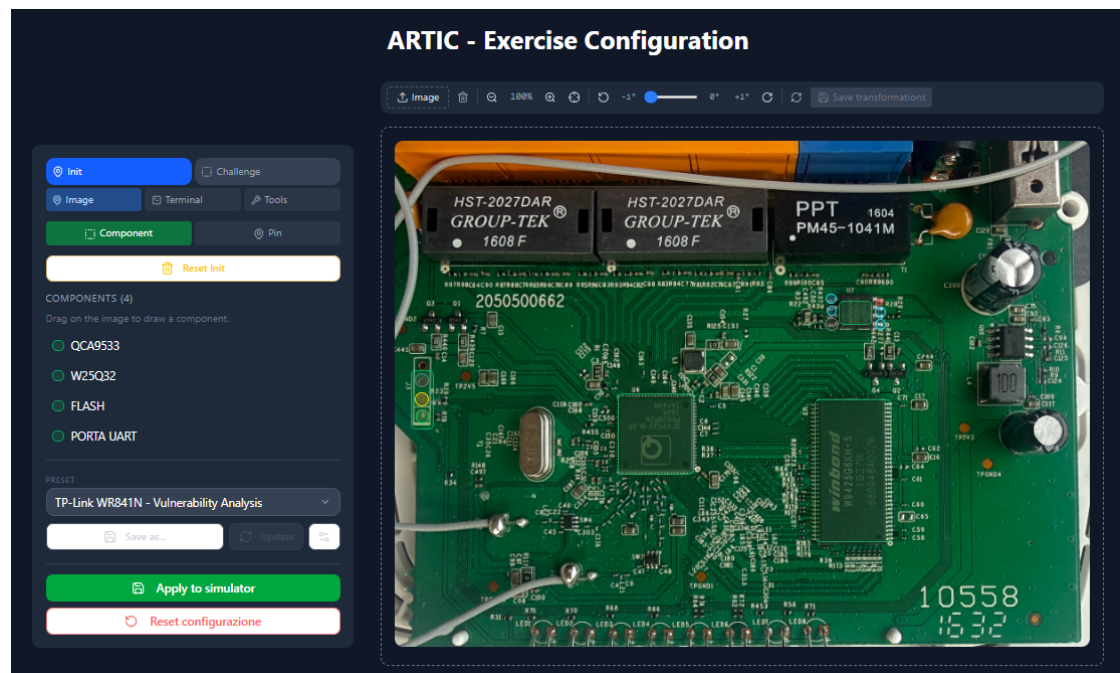


Figure 7.6: The authoring panel during exercise configuration. The canvas shows the PCB image with component regions and pins placed via drag-and-drop. The sidebar displays the step and objective editor.

Chapter 8

Conclusions

8.1 Summary of Contributions

This thesis set out to close a specific gap in cybersecurity education: the absence of an accessible, browser-based environment that simulates the complete workflow of hardware analysis, from physical board inspection to embedded system exploitation. We proposed and implemented PCB-CTF, a web-based interactive cyber range for hardware security training in Capture The Flag format. The design was guided by two overarching principles—*interaction fidelity* and *declarative configurability*—and yielded two contributions that we now revisit against the goals stated in Chapter 1. The first is the PCB-CTF platform itself, whose internal structure rests on three interlocking subsystems described below; the second is the validation case study built on top of it.

The platform’s first subsystem, the client-side instrument simulation architecture, delivers realistic hardware analysis tools—digital multimeter, UART adapter, SPI programmer—directly in the browser. Differential voltage measurements, crossover wiring validation, and role-based probe connections reproduce the diagnostic reasoning a student would exercise on a physical workstation. A normalised coordinate system ensures that interactive overlays scale correctly across display sizes, and the absence of server-side execution eliminates an entire class of security concerns while enabling offline operation after initial page load.

The second subsystem, the declarative terminal emulation engine, provides a configuration language expressive enough to define virtual filesystems, multi-stage boot sequences, custom commands with six output types, five constraint categories, conditional flag unlock side effects, and multi-tab environments with independent boot sequences. This engine reproduces the experience of interacting with a real embedded device through a synchronous command pipeline. Exercise authors shape terminal behaviour entirely through JSON configuration, never through code changes.

The third subsystem, the no-code authoring interface, provides a visual exercise editor with drag-and-drop PCB markup, structured step and objective editing, integrated terminal configuration, and a preset system for exercise versioning and distribution. The four-store state management architecture (`exerciseStore`, `settingsStore`, `terminalSettingsStore`, `presetStore`) supports automatic dirty tracking, three-level persistence, and live synchronisation between the authoring panel and the simulator.

The second contribution, the validation case study, demonstrates that a complete multi-step exercise—covering component identification, UART connection, firmware extraction, and six distinct vulnerability discovery tasks on a TP-Link WR841N router—can be fully configured and delivered through the proposed system.

8.2 Requirements Assessment

The functional and non-functional requirements defined in Chapter 4 provide a concrete basis for evaluating the platform. All twenty-three functional requirements (RF-01 through RF-23) are satisfied by the current implementation: the PCB simulation subsystem (RF-01 through RF-03) is validated by the interactive viewer described in Chapter 6; the measurement and connection tools (RF-04 through RF-07) are exercised in the case study’s multimeter, UART, and SPI tasks; the terminal subsystem (RF-08 through RF-13) supports the full range of commands, constraints, and boot sequences required by the WR841N exercise; the exercise structure (RF-14 through RF-17) governs the nine-step learning path; and the authoring subsystem (RF-18 through RF-23) enables the complete exercise to be created, modified, and distributed without code changes. Among the non-functional requirements, zero installation (RNF-01), client-side execution (RNF-06), server independence (RNF-07), and the prohibition of server-side code execution (RNF-10) are architectural invariants enforced by the implementation’s design. The remaining non-functional requirements—simplified deployment (RNF-02), guided interaction (RNF-03), visual feedback (RNF-04), responsive layout (RNF-05), static typing (RNF-08), modular terminal design (RNF-09), and session isolation (RNF-11)—are met as documented in Chapters 5 and 6.

8.3 Discussion

The comparative analysis in Chapter 3 positions PCB-CTF in a space that existing solutions do not occupy: a zero-cost, fully browser-based platform offering PCB-level interaction, instrument simulation, and terminal emulation with no server-side code execution. This combination addresses the three main barriers to hardware security training—cost (no equipment required), logistics (no physical setup or shipping), and security (no risk of server compromise through student inputs).

The pedagogical design of the platform draws on well-established principles. Svabensky et al.’s survey of hands-on cybersecurity training confirms that practical, interactive exercises produce measurably better learning outcomes than passive instruction [3]. The progressive structure of PCB-CTF exercises—where each step builds on skills acquired in previous steps, new tools become available only when the student has demonstrated competence with existing ones, and hints are available but not forced—embodies the concept of *scaffolding*, in which instructional support is calibrated to the learner’s current level and gradually withdrawn as competence grows. Similarly, the progressive flag system implements *progressive disclosure*: flag fragments are revealed only when the student demonstrates understanding of the underlying concept (recognising a boot configuration anomaly, cracking a password hash, identifying a command injection pattern), reinforcing the connection between analytical reasoning and tangible outcomes. The student cannot guess the flag; each fragment requires completing a specific analytical step, and the partial flag string provides continuous feedback on progress. Hamari et al. established that gamification elements—points, progressive unlocks, visible achievement markers—significantly increase engagement in learning contexts [42], and the CTF format exploits precisely these mechanisms.

The pedagogical coverage of the platform spans the core phases of hardware analysis: visual reconnaissance, electrical measurement, protocol identification, serial connection, firmware extraction, bootloader analysis, credential extraction, and vulnerability discovery. This progression aligns with the methodology described in standard references such as the *Hardware Hacking Handbook* [12] and *Practical IoT Hacking* [44]. The nine-step learning

path of the case study exercise maps directly to the procedural phases described in Section 2.1.3, demonstrating that the platform supports a structurally complete training experience within a single exercise.

The choice to simulate instrument readings with realistic electrical characteristics—voltage levels, impedances, noise—rather than presenting idealised values reflects a commitment to developing transferable diagnostic reasoning. When a student measures 3.3 V with low impedance on a UART TX line and 3.3 V with high impedance on the RX line, they are practising the same inference they would apply on real hardware. The simulated digital noise on the multimeter display, while cosmetic, reinforces the perception of working with a physical instrument.

The deterministic nature of the simulation is both a strength and a limitation. It is a strength because identical conditions across all sessions make the platform suitable for graded assessments and reproducible demonstrations. It is a limitation because real hardware analysis involves noise, timing variations, and unexpected device behaviour that a configuration-driven simulation cannot fully capture. We therefore position the platform as a preparatory tool that develops foundational reasoning skills, not as a replacement for physical laboratory experience.

The security-by-design approach documented in Section 6.7 demonstrates that the platform practises what it teaches: by confining all execution to the client side, sanitising inputs, and enforcing path boundaries, the implementation avoids the very classes of vulnerabilities—*injection*, *path traversal*, *insecure defaults*—that its exercises train students to identify in IoT devices.

8.4 Limitations

Five limitations bound the current implementation, each representing a conscious design trade-off rather than an oversight.

First, the terminal supports only pre-configured commands. A student who types a command absent from the exercise configuration receives a “command not found” error, even when the command would be meaningful in a real environment—`ifconfig`, `dmesg`, or `mount`, for example. This restricts the sense of open-ended exploration that characterises interaction with a real device. The trade-off is deliberate: supporting only declared commands guarantees deterministic behaviour and eliminates the risk of students encountering undefined states, both of which are essential for graded assessments. In practice, the limitation can be mitigated by the exercise author, who can pre-configure any command that is pedagogically relevant. The case study exercise includes over twenty custom commands, which proved sufficient to sustain a convincing exploration experience across six challenges.

Second, protocol coverage is limited to UART and SPI. JTAG, I²C, and CAN—all common in embedded devices—are not yet supported. Each new protocol would require a dedicated tool widget, a pin type definition with role-based validation, and a set of corresponding exercise objective types. The current architecture is designed to accommodate such extensions: the modular tool system, the generic pin model, and the objective type registry are all extensible without structural changes. We prioritised UART and SPI because they cover the two most prevalent attack vectors in consumer IoT devices (serial console access and firmware extraction, respectively), as documented in Section 2.4.3.

Third, the platform operates in a single-user architecture without scoring, leaderboards, or persistent progress tracking. Each browser session is independent, and there is no concept of user identity or session history. This limits the platform’s applicability to

competitive CTF events, where simultaneous participation and ranking are expected. The single-user model was chosen to minimise architectural complexity and avoid the security implications of user authentication in an educational tool. Multi-user support is the most frequently cited extension in our future work plan.

Fourth, no empirical user evaluation has been conducted. The validation presented in Chapter 7 is based on feature coverage, structural alignment with real-world methodology, and informal observation during development. A controlled study with a representative student cohort—measuring learning outcomes through pre- and post-assessments, comparing against traditional laboratory instruction, and evaluating usability through standardised questionnaires—remains necessary to quantify pedagogical effectiveness. We acknowledge this as the most important outstanding task.

Fifth, the authoring panel is protected only by a build-time environment flag and lacks authentication or authorisation layers. While this is sufficient for the intended deployment model (single-user, local development), it makes the platform unsuitable for multi-tenant production environments where multiple instructors might share a single instance. Adding role-based access control to the authoring API is a straightforward extension that we defer to a future version.

8.5 Future Work

Several directions for future development follow naturally from these limitations and from the broader design goals of the platform.

Multi-user support and scoring. Adding user authentication, session management, database-backed progress persistence, and a timing-based leaderboard would enable the platform to host live CTF events with concurrent participants. A scoring system could incorporate partial credit for discovered flag fragments, time penalties for hint usage, and bonus points for optional objectives. This extension would also enable instructors to collect anonymised aggregate data on exercise difficulty and common stumbling points, supporting evidence-based refinement of the learning path. The existing four-store architecture provides a natural integration point: per-session state would migrate from local persistence to a server-backed identity, while exercise definitions and authoring artefacts would remain shared resources.

Hybrid simulation–hardware integration. A natural evolution connects the platform to remote laboratory infrastructure, enabling a seamless transition from simulated exercises to interaction with real hardware. A student might begin with the simulated WR841N exercise to develop foundational skills, then switch to a remote lab session with web-controlled instruments targeting a physical device. The platform’s clean separation between simulation frontend and configuration backend is amenable to this extension: instrument widgets could relay commands to remote hardware rather than to local state, while the exercise definition language could be augmented with an indirection layer that maps virtual pins to physical test points on a paired board.

Additional protocol support. Implementing JTAG would enable exercises focused on boundary scan, debug port exploitation, and memory read-out. I²C support would extend coverage to EEPROM-based devices. CAN bus support would open the door to automotive security scenarios, a rapidly growing area of concern. Each new protocol requires a pin type definition, a tool widget, and a validation strategy in the exercise store.

Dynamic terminal responses. The restriction to pre-configured commands could be relaxed by generating plausible output for unexpected commands based on the current context—filesystem state, boot stage, previous command history. A pragmatic

intermediate step would be to ship a library of common embedded-Linux command stubs (`ifconfig`, `dmesg`, `mount`, `ps`, `netstat`) that exercise authors can opt into and override on a per-exercise basis. Such a mechanism would improve the sense of open-ended exploration while maintaining the security guarantee of client-side-only execution and the determinism required for graded assessments.

Localisation and accessibility. Broader adoption beyond English-language curricula calls for a translation layer covering the user interface strings, the in-exercise hints, and the authored content of community-contributed exercises. The configuration-driven nature of the platform already isolates pedagogical text from code, simplifying the introduction of a string-table mechanism. Accessibility improvements—keyboard-only navigation of the PCB viewer, screen-reader-friendly descriptions of board components, high-contrast and colour-blind-safe palettes for the instrument widgets—are equally important to lower barriers for students with sensory or motor impairments.

Student analytics and adaptive difficulty. Tracking completion times, hint usage frequency, error patterns, and command sequences would provide instructors with quantitative data to calibrate exercise difficulty. An adaptive system could dynamically adjust hint visibility, present additional guidance for struggling students, or unlock bonus challenges for advanced learners.

Open exercise library and community contributions. A natural enabler of broader adoption is the curation of a publicly accessible library of exercises spanning a representative set of consumer and industrial devices. The preset-based authoring model is well suited to community contributions, since each exercise is fully described by a self-contained JSON document that can be reviewed, versioned through standard source-control tools, and redistributed without runtime side effects on the host platform. Combined with a lightweight contribution workflow—template repositories, automated schema validation against the exercise data model, and a minimal peer-review process for content quality and pedagogical soundness—such a library would amortise the authoring effort across institutions and progressively cover the long tail of hardware platforms that no single course could reasonably maintain in-house. The same infrastructure would also support targeted thematic collections (e.g., consumer routers, industrial control boards, automotive electronic control units), allowing instructors to compose course-specific learning paths from a shared pool of vetted material.

Empirical validation. The most important outstanding task is a rigorous empirical evaluation. A controlled study with a representative student cohort—ideally within a university course—would quantify pedagogical effectiveness through pre- and post-assessments, measure usability through standardised questionnaires such as the System Usability Scale, and compare learning outcomes against traditional laboratory instruction. A complementary line of investigation could examine how well skills acquired in the simulated environment transfer to interaction with physical hardware, for instance by having one cohort complete the simulated WR841N exercise and a parallel cohort work on the same device through a remote laboratory or in a supervised laboratory setting, then comparing performance on a follow-up task that requires unaided diagnostic reasoning on a different device. Such a study would provide the evidence base needed to justify broader adoption and would also help calibrate the boundary between what can be effectively taught in a fully simulated setting and what continues to require time on real hardware.

Hardware security knowledge should not be gated by the price of a logic analyser or the availability of a supervised laboratory. The barriers that make hardware skills scarce are primarily logistical, not intellectual, and a well-designed simulation platform can lower them dramatically. PCB-CTF demonstrates that the core workflow of hardware analysis—from visual board inspection through protocol-level probing to embedded

system exploitation—can be faithfully reproduced in a standard web browser at zero marginal cost per student. If the cybersecurity community is serious about closing the hardware skills gap, tools like this one are not optional; they are the necessary bridge between the software-centric training ecosystem that exists today and the hardware-aware workforce that tomorrow’s threat landscape demands.

Bibliography

- [1] Alessandro Genova. Vulnerability assessment of low-cost IoT devices: Towards a virtual hardware security training environment. Master’s thesis, Politecnico di Torino, 2025.
- [2] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *USENIX Security Symposium*, pages 95–110. USENIX Association, 2014.
- [3] Valdemar Švábenský, Jan Vykopal, and Pavel Čeleda. What are cybersecurity education papers about? a systematic literature review of SIGCSE and ITiCSE conferences. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 2–8. ACM, 2020.
- [4] V. Judson Harward, Jesús A. del Alamo, Steven R. Lerman, Philip H. Bailey, Joel Carpenter, Kirky K. DeLong, Chris Felknor, James L. Hardison, Bryant Harrison, Imad Jabbour, Phillip D. Long, Tingting Mao, Loai Naamani, Jedidiah Northridge, Mark Schulz, Daniel Talavera, Charuleka Varadharajan, Shaomin Wang, Karim Yehia, Rabih Zbib, and David Zych. The iLab shared architecture: A web services infrastructure to build communities of internet accessible laboratories. *Proceedings of the IEEE*, 96(6):931–950, 2008.
- [5] Gianluca Roascio, Samuele Yves Cerini, and Paolo Prinetto. Remotizing and virtualizing chips and circuits for hardware-based capture-the-flag challenges. In *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 477–485. IEEE, 2022.
- [6] Riscure. RHme: Riscure hack me, 2016.
- [7] Trail of Bits. Microcorruption: Embedded security ctf, 2023.
- [8] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, Indianapolis, Indiana, 3rd edition, 2020.
- [9] Paolo Prinetto. Hardware security, 2019. Lecture Notes, Politecnico di Torino.
- [10] Georg T. Becker, Francesco Regazzoni, Christof Paar, and Wayne P. Burleson. Stealthy dopant-level hardware trojans. In *Cryptographic Hardware and Embedded Systems (CHES)*, volume 8086 of *Lecture Notes in Computer Science*, pages 197–214. Springer, 2013.
- [11] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai

- botnet. In *USENIX Security Symposium*, pages 1093–1110. USENIX Association, 2017.
- [12] Jasper van Woudenberg and Colin O’Flynn. *The Hardware Hacking Handbook: Breaking Embedded Security with Hardware Attacks*. No Starch Press, San Francisco, CA, 2021.
 - [13] Omer Schwartz, Yael Mathov, Michael Bohadana, Yuval Elovici, and Yossi Oren. Opening pandora’s box: Effective techniques for reverse engineering IoT devices. In *Smart Card Research and Advanced Applications (CARDIS)*, volume 10728 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2017.
 - [14] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology – CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
 - [15] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.
 - [16] Craig Heffner. Binwalk: Firmware analysis tool, 2023.
 - [17] flashrom contributors. flashrom: Universal flash programming utility, 2023.
 - [18] National Security Agency. Ghidra: Software reverse engineering framework, 2023.
 - [19] Vercel. Next.js: The React framework for the web, 2024.
 - [20] Meta Open Source. React: The library for web and native user interfaces, 2024.
 - [21] Microsoft. The TypeScript handbook, 2024.
 - [22] Redux Maintainers. Redux: A predictable state container for JS apps, 2024.
 - [23] Poimandres. Zustand: A small, fast, and scalable state-management solution, 2024.
 - [24] Tailwind Labs. Tailwind CSS: A utility-first CSS framework, 2024.
 - [25] WorkOS. Radix UI: Unstyled, accessible components for building high-quality design systems, 2024.
 - [26] shadcn. shadcn/ui: Beautifully designed components built with Radix UI and Tailwind CSS, 2024.
 - [27] CTFd. CTFd: Capture the flag framework, 2023.
 - [28] Carnegie Mellon University. PicoCTF: Free computer security education, 2023.
 - [29] Hack The Box. Hack the box: Cyber security training platform, 2024.
 - [30] TryHackMe. Tryhackme: Cyber security training, 2024.
 - [31] NYU Center for Cyber Security. CSAW embedded security challenge, 2023.
 - [32] OWASP. IoTGoat: A deliberately insecure firmware, 2023.
 - [33] Javier García-Zubia, Diego López de Ipiña, and Pablo Ordu na. Evolving towards better architectures for remote laboratories: A practical case. *International Journal of Online Engineering*, 1(2), 2005.

- [34] Ingvar Gustavsson, Johan Zackrisson, Lars Håkansson, Ingvar Claesson, and Thomas Lagö. The VISIR project: An open source software initiative for distributed online laboratories. In *International Conference on Remote Engineering and Virtual Instrumentation (REV)*, pages 1–6, 2007.
- [35] Jan Vykopal, Radek Ošlejšek, Pavel Čeleda, Martin Vizáry, and Daniel Tovarňák. KYPO cyber range: Design and use cases. In *International Conference on Software Technologies (ICSOFT)*, pages 310–321. SciTePress, 2017.
- [36] Cuong Pham, Dat Tran, Ken ichi Chinen, and Razvan Beuran. CyRIS: A cyber range instantiation system for facilitating security training. In *International Symposium on Information and Communication Technology (SoICT)*, pages 251–258. ACM, 2016.
- [37] Terry Benzel, Robert Braden, Dongho Kim, Clifford Neuman, Anthony D. Joseph, Keith Sklower, Ron Ostrenga, and Stephen Schwab. Experience with DETER: A testbed for security research. In *International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TRIDENTCOM)*. IEEE, 2006.
- [38] Paul Falstad. Circuit simulator applet, 2023.
- [39] Wokwi. Wokwi: Online electronics simulator, 2023.
- [40] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for Linux-based embedded firmware. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2016.
- [41] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. FirmAE: Towards large-scale emulation of IoT firmware for dynamic analysis. In *Annual Computer Security Applications Conference (ACSAC)*, pages 733–745. ACM, 2020.
- [42] Juho Hamari, Jonna Koivisto, and Harri Sarsa. Does gamification work? — a literature review of empirical studies on gamification. In *Hawaii International Conference on System Sciences (HICSS)*, pages 3025–3034. IEEE, 2014.
- [43] OWASP Foundation. OWASP top 10:2021 – the ten most critical web application security risks, 2021.
- [44] Fotios Chantzis, Ioannis Stais, Paulino Calderon, Evangelos Deirmentzoglou, and Beau Woods. *Practical IoT Hacking: The Definitive Guide to Attacking the Internet of Things*. No Starch Press, San Francisco, CA, 2021.