Embedded Firmware Security Tester

An Embedded Systems QA and Fuzz Testing Tool

John Gahagan
john.gahagan3@gmail.com

May 11, 2025

Abstract

This project aims to design and evaluate an embedded firmware security testing framework, focusing on identifying input vulnerabilities in microcontroller-based applications. It combines embedded programming with automated fuzz testing to simulate malformed input, boundary violations, and unauthorized command sequences over serial communication. This tool enables quality assurance (QA) validation and basic security auditing for systems deployed in resource-constrained environments. The system now includes checksum validation on both input and output messages to detect message tampering and improve integrity verification. Additionally, EEPROM is used to persist temperature settings across device reboots for configuration retention testing.

1 Project Overview

Microcontrollers often serve as the backbone of embedded systems in critical domains such as healthcare, automotive, and industrial control. These devices must handle untrusted input robustly, especially when exposed to external systems via UART, I2C, or SPI. This project builds a two-part test harness: a firmware target and a serial-based security testing tool. The goal is to simulate and detect vulnerabilities stemming from insufficient input validation, overflows, protocol misuse, or denial of service patterns.

2 System Architecture

2.1 Component Breakdown

- **Firmware Target:** A command-based Arduino firmware that responds to structured serial input, such as "SET TEMP 25—201" or "READ TEMP—114", verifying incoming checksums before processing, appending checksums to all outgoing responses, and persisting temperature settings using EEPROM across reboots.
- **Security Tester:** A Python script that sends both valid and malicious inputs over UART. It verifies checksums on responses from the Arduino and includes a manual pause-based reboot test to verify EEPROM-based persistence.
- **Logger:** Captures command/response pairs, timeouts, and system resets. Outputs results to a structured log file.

2.2 Checksum-Based Integrity Validation

To protect against data corruption or tampering during serial communication, the system implements checksum validation in both directions:

- **Input Validation:** Each incoming command from the tester must include a checksum suffix in the format `COMMAND|CHECKSUM`. The firmware computes the checksum by summing the ASCII values of all characters in the command and taking modulo 256. Mismatched checksums result in a rejection response.
- **Output Protection:** The firmware appends a similar checksum to all outgoing messages (e.g., `TEMP=25|157`). The Python tester then verifies the integrity of the received response using the same checksum algorithm.

2.3 Architecture Diagram

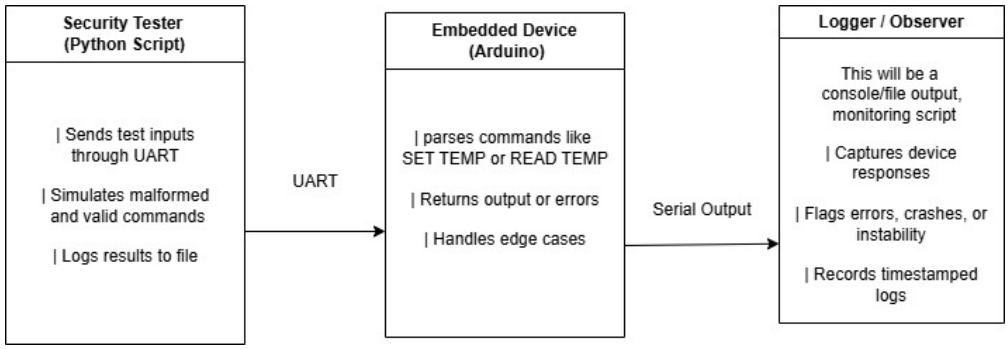

Figure 1: System Architecture: Tester injects serial data, firmware responds, logs are recorded

2.4 EEPROM Persistence Flow

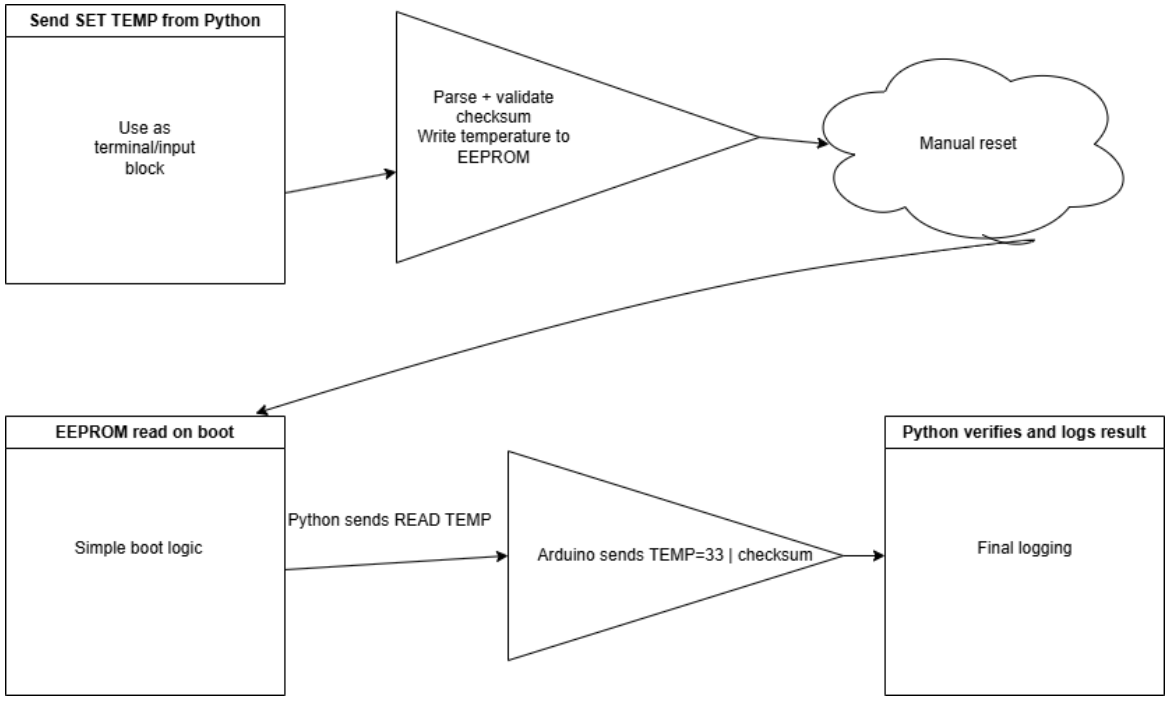

Figure 2: EEPROM Persistence Flow: SET TEMP command is saved, survives reboot, and returns same value on READ TEMP.

2.5 Firmware Input Handling Flow

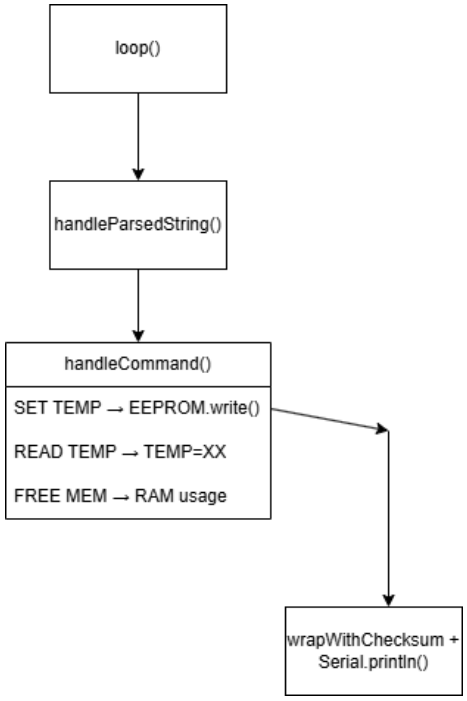

Figure 3: Arduino firmware logic from non-blocking input buffer to command parsing, EEPROM handling, and structured serial output.

2.6 Python Fuzzer Structure

The Python-based security tester follows a layered architecture. Test cases are generated, checksummed, transmitted over UART, and validated through a sequence of functions. Each component in this flow contributes to the input validation and response auditing pipeline.

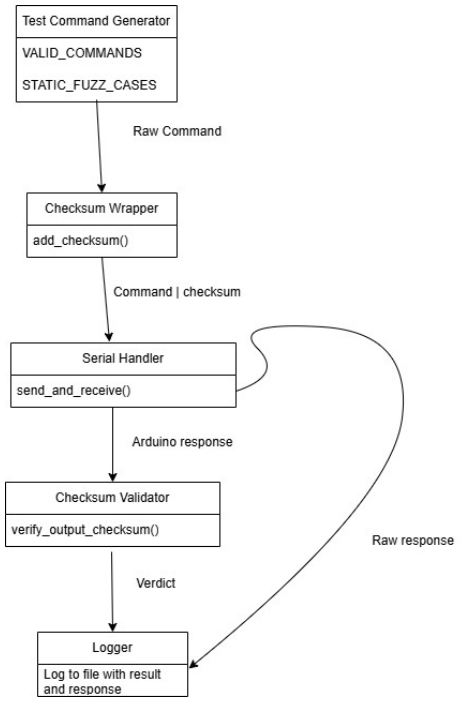

Figure 4: Python fuzzer architecture: commands are generated, checksummed, transmitted, validated, and logged.

3 Use Case Scenario

Consider a low-power environmental sensor node deployed in the field, accepting serial commands from a host. Without robust input handling, such devices may crash or behave unpredictably when presented with malformed commands. This testing framework replicates these input conditions to evaluate real-world resilience. It serves as both a QA tool and a foundational security audit.

4 Test Plan

4.1 Testing Levels

- **Unit Testing:** Validates small, isolated firmware functions such as parsing ‘SET TEMP‘ and ensuring correct extraction of numeric values.
- **Integration Testing:** Tests complete firmware paths from input to response—ensuring serial input is processed correctly and output is returned through the serial interface.
- **System Testing:** Involves full end-to-end runs between the Python fuzzing script and the firmware, verifying correct logging and fault handling.
- **Acceptance Testing:** Evaluates whether the firmware meets project goals, remains stable, handles edge cases, and responds to malformed input appropriately.

4.2 Defined Fuzz Tests

Test Type	Description	Expected Result
BufferOverflow	Send a string longer than the firmware buffer (e.g., 100+ characters).	Firmware rejects input or returns error message. System remains responsive and does not crash.
CommandInjection	Send combined command such as ‘SET TEMP 25;DELETE ALL‘.	Firmware ignores invalid syntax after the first command. Only valid input is executed.
Invalid Range	Send values outside allowed bounds, e.g., ‘SET TEMP -999‘ or ‘999‘.	Firmware returns an error message without applying the value.
ProtocolViolation	Send incomplete or malformed commands like ‘SET TEMP‘ without a value.	Firmware responds with a syntax error or prompts for correct input.
ReplayAttack	Rapidly repeat ‘READ TEMP‘ or similar valid command multiple times.	Device handles requests consistently and remains stable with no delay buildup.
NoiseInjection	Send garbage characters like ‘!@#^&*‘.	Device responds with a clear error or ignores the input without crashing.
ChecksumMismatch	Provide a valid command with a tampered checksum (e.g., `SET TEMP 25|999`).	Firmware rejects input with `ERROR: Invalid Checksum`. Tester detects bad response if output is corrupted.
EEPROM Retention	Send ‘SET TEMP 33‘, manually reboot device, then issue ‘READ TEMP‘.	Firmware correctly returns `TEMP=33`, confirming configuration persisted across reset.

Table 1: Defined fuzz test cases and expected outcomes

5 Evaluation Criteria

- **Stability:** The firmware must remain operational regardless of input. This includes not crashing or becoming unresponsive during fuzzing or reboot cycles.
- **Accuracy:** Output responses must be well-formed and match expected formats, including when checksum or EEPROM values are used.
- **Consistency:** Commands must yield the same result on each run. EEPROM values are expected to persist across power cycles.
- **Safety:** Malformed inputs must not alter persistent configuration or crash the system.

- **Performance (Optional):** Not the main goal, but logs may reflect system response under load.

6 Tools and Technologies

- **Arduino IDE:** Used for firmware development and deployment.
- **EEPROM Library:** Used to persist temperature values across device resets.
- **Python 3 + PySerial:** For sending test input and logging responses.
- **Visual Studio Code:** Editor and Git frontend.
- **Git + GitHub:** For version control and documentation publishing.

7 CI/CD Considerations

- **Manual EEPROM Test:** A reboot test in ‘fuzzer.py‘ prompts the user to reset the Arduino and verifies that a previously set value was retained.