Embedded Firmware Security Tester

An Embedded Systems QA and Fuzz Testing Tool

John Gahagan
john.gahagan3@gmail.com

May 7, 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.

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" or "READ TEMP".
- **Security Tester:** A Python script that sends both valid and malicious inputs over UART to evaluate the firmware's robustness.
- **Logger:** Captures command/response pairs, timeouts, and system resets. Outputs results to a structured log file.
- **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 and appending checksums to all outgoing responses.
- **Security Tester:** A Python script that sends both valid and malicious inputs over UART. It now also verifies checksums on responses from the Arduino to detect unexpected corruption or unauthorized response changes.

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.

This protects the system from malformed messages caused by signal noise, buffer overruns, or unauthorized message manipulation.

2.3 Architecture Diagram

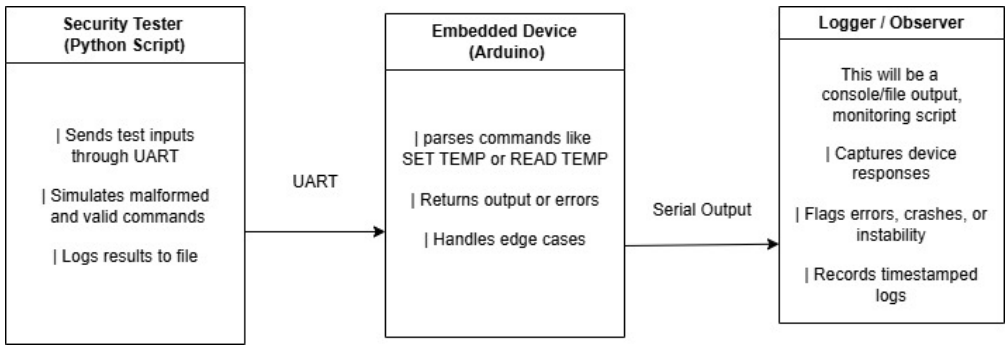

Figure 1: System Architecture: Tester injects serial data, firmware responds, logs are recorded

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

This section outlines both the test levels and specific test cases used to evaluate firmware behavior under abnormal and malicious input conditions.

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

The table below outlines the fuzz test cases applied across testing levels. Each test maps to a common embedded vulnerability class.

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.

Table 1: Defined fuzz test cases and expected outcomes

5 Evaluation Criteria

To determine whether the firmware responds appropriately to various types of malformed or malicious input, each test will be evaluated using the following quality assurance and safety metrics. These criteria reflect industry standards for embedded system reliability, particularly in environments where input cannot always be trusted.

- **Stability:** The firmware must remain operational regardless of the input it receives. A test case passes this criterion if the device does not hang, reset, or enter an unrecoverable state following an input. Stability is assessed by monitoring the device's responsiveness and checking for any unexpected resets, crashes, or watchdog triggers.
- **Accuracy:** Responses to invalid input should be precise, predictable, and informative. For example, receiving 'SET TEMP -999' should result in a specific and well-structured error message (e.g., 'ERROR: Out of range'). Accuracy ensures that the firmware properly parses inputs, identifies violations, and reports issues clearly without ambiguity.
- **Consistency:** Across repeated runs and different test inputs, the device should behave in a consistent manner. The same command must always yield the same response under the same conditions. This criterion is validated by running identical tests multiple times and comparing outputs. Consistency is key to debugging, system predictability, and ensuring reproducibility in both testing and deployment.
- **Safety:** The firmware must not allow malformed or malicious input to cause unintended side effects. This includes preventing overwrites of protected variables, avoiding command injection, and ensuring that critical system states (such as thresholds or configurations) are not altered without proper validation. Tests in this category validate that the firmware adheres to secure input-handling best practices.
- **Performance (Optional):** While not a primary focus, performance may be observed to ensure that repeated testing or high-volume input does not degrade system responsiveness. Excessive latency, jitter, or memory consumption under stress conditions could indicate vulnerabilities. This may be tracked using response timing logs or resource usage statistics during prolonged fuzz testing.

Each test case will be marked as *Pass*, *Fail*, or *Warning* based on these criteria. Results will be documented in structured log files and summarized in a test results report for final evaluation.

Test results will be documented in a log file for review, and a pass/fail summary will be compiled based on these criteria.

6 Tools and Technologies

A combination of software development tools, embedded programming environments, and testing utilities were used to design, implement, and validate the Embedded Firmware Security Tester system:

- **Arduino IDE:** The Arduino Integrated Development Environment (IDE) was used to write, compile, and upload the embedded C++ firmware to the Arduino Uno R4 board. The IDE provides built-in support for serial monitoring, which was helpful during early stages of manual testing and debugging. The open-source toolchain simplifies microcontroller programming and hardware interfacing.
- **Python 3 + PySerial:** Python was selected as the scripting language for the security tester due to its readability, ease of string manipulation, and wide availability of hardware-interfacing libraries. The `pyserial` module allows seamless communication with the microcontroller over UART, enabling scripted test case injection and response logging. Python's flexibility also enables integration with logging tools and CI workflows.

- **UART / USB Serial Communication:** The core interface between the host system and the embedded device is UART over USB. This provides a simple and reliable method for transmitting ASCII command strings and receiving responses. UART enables real-time testing without requiring additional hardware protocols such as I2C or SPI, while remaining extensible to them in the future.
- **Visual Studio Code:** VS Code served as the primary development environment for editing Python scripts and reviewing log files. Its extensions for linting, syntax highlighting, Git integration, and Markdown preview accelerated development and documentation workflows. The built-in terminal also facilitated interaction with Git and Python environments.
- **Git + GitHub:** Git was used for local version control of firmware and tester scripts, enabling incremental development and experimentation with test logic. GitHub hosted the centralized project repository, providing remote backup, issue tracking, pull request reviews, and CI integration. The public repository also facilitates collaboration, reproducibility, and visibility of the project.
- **(Optional) PlatformIO CLI:** While not required, PlatformIO can serve as an advanced alternative to the Arduino IDE for compiling and uploading firmware. Its command-line interface integrates well with CI pipelines and supports unit testing frameworks like Unity or Ceedling for embedded C code.
- **(Optional) Logic Analyzer / Serial Monitor Tools:** During debugging and timing verification, a USB logic analyzer or serial sniffing tool (e.g., Saleae Logic or RealTerm) can be used to verify exact message timing, framing issues, and protocol integrity on the UART line.

7 Continuous Integration and Testing Pipeline (CI/CD)

A CI/CD pipeline will help automate software testing and code validation, especially for the Python-based tester. While hardware tests require physical interaction, the software components will benefit from automation:

- **Unit Testing via GitHub Actions:** Python unit tests will automatically run on every push or pull request. Checksum verification logic is included in Python test cases to simulate command parsing and response validation.
- **Firmware Build Validation:** The firmware will be compiled using PlatformIO CLI or Arduino CLI in CI to check for build errors.
- **Static Analysis:** Tools like `pylint` or `cppcheck` will validate code quality.
- **Manual HIL Test Logging:** Hardware-in-the-loop tests will be manually run, with logs uploaded to the repository.

This setup ensures rapid feedback and quality control for software layers, while supporting structured QA for the embedded firmware.

8 Planned Deliverables

The following deliverables will be produced to support development, validation, and documentation of the Embedded Firmware Security Tester project:

- **Firmware and Tester Codebase:** The complete source code for the Arduino firmware (C/C++) and the Python-based fuzz testing tool will be provided. All files will include inline comments explaining function logic, test routines, and communication structure. The codebase will be organized in a GitHub repository with version history, build instructions, and a project README file.
- **Documentation Package:** This includes the project report in LaTeX format, covering architecture, test strategy, implementation details, and evaluation criteria. Supporting documents such as the test plan, logs of test executions, and a vulnerability report will be bundled as supplementary PDF files or within a 'docs/' folder in the repository.
- **Demo Video:** A short recorded demonstration showing the fuzzer in action. This will cover interactions between the Python tester and the firmware, how inputs are injected, how responses are received, and how results are logged and interpreted. This will help non-technical reviewers or collaborators understand the testing flow.
- **Final Evaluation Report:** A concluding report summarizing test outcomes, identifying any vulnerabilities or failure points discovered, and describing firmware updates or design changes made in response to test findings. This report will include an overview of testing coverage, performance metrics, and known limitations.
- **Optional CI/CD Pipeline Files:** If CI is configured, GitHub Actions workflows or shell scripts will be included in the repository to demonstrate automatic test/lint/build execution. These will validate Python scripts, build the firmware, and document results in logs or badges.
- **Log Archives:** A collection of timestamped run logs from fuzz test sessions, including both passing and failing cases. These logs will provide traceability and support regression testing.