**Integrating LLM-Driven Firmware Generation with Security Validation for FreeRTOS on QEMU**

**Author(s):**  
*Your Name, Department/Institution*  
*Email Address*

**Abstract**

We present a framework that integrates Large Language Model (LLM)-based firmware generation with an automated security testing pipeline for FreeRTOS-based embedded systems running on QEMU. Building on prior works that explore LLM-generated code, we address key security threats—buffer overflows, race conditions, denial of service (DoS), and real-time deadline violations—by integrating fuzz testing, static analysis, and iterative patch refinement. Our approach demonstrates **concrete** threat modeling, **real-time performance checks**, and an end-to-end feedback loop where the LLM suggests targeted fixes upon detecting vulnerabilities. We show that this pipeline systematically reduces common flaws in LLM-generated code and improves real-time performance.

**1. Introduction**

**1.1 Motivation.**  
Embedded systems increasingly rely on network-enabled firmware that demands high reliability, especially when auto-generated by Large Language Models (LLMs) like GPT-4. Recent research suggests that while LLMs excel at quick prototyping, they often introduce subtle security flaws—improper boundary checks, concurrency errors, or logic oversights—potentially jeopardizing real-time performance (Engelhardt et al., 2024). Such vulnerabilities can be disastrous in critical domains (automotive, aerospace, etc.), motivating a secure-by-design approach.

**1.2 Related Challenges.**  
Traditional fuzz testing (AFL++, Syzkaller) and static analysis tools (Clang, Cppcheck) effectively detect general software issues but are not fine-tuned to handle the unique quirks of LLM-generated code—e.g., partial API hallucinations or incomplete protocol handling. Researchers (Dunne et al., 2024) note that LLM-based networking code often fails advanced boundary checks under stress conditions. Our method extends standard fuzzing and static analysis to systematically uncover LLM-specific anomalies, then loops that feedback into the LLM for iterative fixes.

**1.3 Contributions.**

1. **Inlined Threat Model** that explicitly addresses buffer overflows, race conditions, denial of service, and unauthorized access.
2. **LLM + QEMU Integration** for easy build, run, fuzz, and static analysis.
3. **Real-Time Performance Checks** embedded in tasks, logging missed deadlines.
4. **Iterative Patch Refinement** where the LLM receives vulnerabilities and proposes code fixes, forming a closed feedback loop.

**2. Methodology & Differences From Prior Work**

**2.1 Overall Approach.**  
Our methodology follows a five-step cycle (expanded from the “Methodology and Experiment Plan” previously proposed):

1. **Threat Modeling**: Identify specific embedded threats (e.g., buffer overflow, concurrency hazards).
2. **LLM-Driven Code Generation**: Use GPT-4 or Code Llama to produce or refine FreeRTOS tasks (via structured prompts).
3. **Integration & Build**: Compile and run tasks on QEMU with minimal overhead.
4. **Security & Reliability Testing**: Perform fuzz testing (malformed data injection) and static analysis to detect flaws.
5. **Analysis & Patch**: Map flaws to CWE references, then feed them back into the LLM for automated or semi-automated fixes.

**2.2 Differences From Prior Work.**

* **Extended Real-Time Checks**: Beyond standard fuzz or static analysis, we measure execution time each iteration to detect missed deadlines, bridging the gap between “syntactic correctness” and real-time constraints.
* **Inlined Threat Model**: We embed threat modeling comments directly in code, ensuring each design/mitigation step references a known threat (CWE-120, CWE-362, etc.).
* **Iterative LLM Refinement**: Some frameworks (e.g., SecRT-LLM) focus on SoC design. Ours specifically merges *run-time* fuzz logs with an LLM to produce targeted patches for *firmware-level* concurrency, buffer checks, and scheduling deadlines.

**3. Implementation**

**3.1 Development Environment.**

* **FreeRTOS** with an official QEMU Cortex-MPS2 target.
* **GCC** or IAR toolchains for cross-compiling.
* **Python** scripts to automate build, run, fuzz, static analysis, and post-processing.

**3.2 Project File Overview.**

1. **main.c**
   * Contains two tasks: vSensorTask() and vSecureNetworkTask().
   * Demonstrates concurrency (mutex for sSensorData) and boundary checks (packet parsing).
   * Logs real-time performance (“MISSED DEADLINE” if execution surpasses threshold).
   * Inlined threat model (buffer overflow, race conditions, DoS, unauthorized access).
2. **build\_and\_run.py (Step 3)**
   * Automates make -C build\_dir and then launches QEMU with the compiled ELF.
3. **fuzz\_test.py (Step 4)**
   * Generates random or oversized input, pipes it into QEMU, captures console logs for anomalies (“overflow,” “hard fault,” “stack overflow,” etc.).
4. **analyze\_results.py (Step 5)**
   * Parses logs for known vulnerabilities, including “missed deadline” for real-time.
   * Maps each discovered flaw to a threat category / CWE ID (e.g., CWE-120).
5. **static\_analysis.py**
   * Runs Cppcheck and clang scan-build on your project, saving logs in test\_artifacts/static\_analysis/.
6. **generate\_freertos\_task.py**
   * Example for using GPT-4 to produce a “secure network task” with boundary checks, concurrency.
7. **llm\_refine.py (Step 6)**
   * Feeds analysis\_report.json to GPT-4 to get fix suggestions for each discovered vulnerability.

**3.3 Real-Time Performance Check.**  
Within main.c, each task uses vTaskDelayUntil() to maintain a fixed period (e.g., 100 ms for SensorTask). We measure start and end xTaskGetTickCount() to see how many ticks the iteration took. If it exceeds 5 ticks, we log “MISSED DEADLINE.” This clarifies potential DoS or scheduling issues—a unique extension that ensures LLM code meets real-time constraints.

**4. Results & Discussion**

**4.1 Fuzz & Static Analysis Findings.**  
Running fuzz\_test.py across 10–50 iterations typically yields:

* **No anomalies** if tasks handle inputs robustly, or
* Potential “overflow” or “stack overflow” logs if boundary checks are missing.
* **Timeout** messages if the firmware stalls, indicating a possible DoS.

Meanwhile, static\_analysis.py (Cppcheck, clang scan-build) warns of concurrency (“race condition”), unused variables, or pointer hazards. The summary is consolidated by analyze\_results.py, which outputs a JSON file mapping each error line to a known CWE (e.g., “overflow” → CWE-120).

**4.2 Real-Time Checks.**  
When tasks intentionally do large computations or wait incorrectly, the logs show:

makefile

CopyEdit

SensorTask: took 2 ticks

SensorTask: MISSED DEADLINE (took 7 ticks)

NetTask: took 1 ticks

analyze\_results.py sees “missed deadline,” maps it to “Real-Time Violation (CWE-400),” enabling you to fix logic or scheduling priorities.

**4.3 Automated Patch Suggestions.**  
Using llm\_refine.py, each discovered flaw is passed to GPT-4 with a snippet of code around the vulnerable line. GPT-4 proposes boundary checks or concurrency fixes. This approach frequently reduces iterative overhead, as it suggests targeted changes rather than rewriting entire modules.

**5. Comparison to Other Works**

* **SecRT-LLM & SPELL**: Prior frameworks focus on hardware design or SoC security. In contrast, we target **firmware-level vulnerabilities** (network parsing, concurrency) and explicitly measure **run-time** scheduling issues.
* **Zhang et al.** showcased coverage-based fuzz corpus generation for embedded systems. We add a post-fuzz LLM refinement step, generating code patches automatically.
* **Engelhardt et al.** identified that LLM-generated code often fails advanced real-time constraints. Our integrated real-time checks plus iterative LLM patching is a direct step forward, bridging that gap.

**6. Conclusion & Future Work**

**6.1 Conclusion.**  
We have presented an end-to-end pipeline merging LLM-based firmware creation with iterative security testing, fuzzing, static analysis, and real-time performance checks in a QEMU-based FreeRTOS environment. By embedding a clear threat model (buffer overflow, race condition, DoS, etc.) and referencing each discovered issue to CWE IDs, we streamline the process of automatically patching or refining vulnerabilities via GPT-4. Our approach goes beyond basic fuzz testing to incorporate real-time constraints—ensuring that LLM-generated firmware meets both functional and security requirements.

**6.2 Future Directions.**

1. **Advanced Coverage-Guided Fuzzing**: Integrating tools like AFL++ with QEMU to further expand test coverage.
2. **Memory Protection Unit (MPU) Usage**: To address “unauthorized access” threats more deeply, e.g., restricting tasks from each other’s memory.
3. **Public Dataset**: We plan to host an open repository of discovered LLM-induced firmware flaws (including fuzz inputs, logs, and patches) for broader academic and industrial use.

**6.3 Closing Remarks.**  
This project demonstrates that LLM-generated embedded firmware can be systematically secured through an iterative loop of testing, analysis, and code refinement. By addressing concurrency, boundary checks, and real-time scheduling, we significantly reduce the risk of catastrophic failures in networked embedded systems. Our methodology offers a foundation for future enhancements and wide-scale adoption of AI-driven firmware development.

**References (Representative)**

1. Engelhardt, Z. et al. *Exploring and characterizing large language models for embedded system development and debugging.* Extended Abstracts CHI, 2024.
2. Paria, S. et al. *Spell: An end-to-end tool flow for llm-guided secure soc design for embedded systems.* IEEE Embedded Systems Letters, 2024.
3. Zhang, Q. et al. *ECG: Augmenting embedded operating system fuzzing via llm-based corpus generation.* IEEE Transactions on CAD, 2024.
4. Dunne, M. et al. *Weaknesses in llm-generated code for embedded systems networking.* 2024 IEEE QRS.