# Software Requirements Specification (SRS)

Dhruv Sunil Bhatia
24CSB0A19
CSE - A
Q119 - Secure OTA Update Compiler

## 1 Introduction

### 1.1 Purpose

This document specifies the functional and non-functional requirements for the Secure OTA Update Compiler. The compiler statically analyzes firmware update logic and enforces mandatory security invariants at compile time. Any firmware update code that violates these invariants is rejected, preventing insecure firmware from being generated or deployed.

### 1.2 Scope

The system operates as a compiler extension or static analysis pass integrated into the firmware build process. It focuses exclusively on firmware update logic, ensuring that critical security checks are present and correctly ordered along all execution paths leading to firmware installation.

The compiler does not:

- Perform runtime enforcement
- Manage key distribution or firmware servers
- Replace secure boot or hardware trust anchors

### 1.3 Definitions and Acronyms

- OTA: Over-The-Air firmware update
- Invariant: Mandatory security property that must hold on all valid execution paths
- CFG: Control Flow Graph
- AST: Abstract Syntax Tree

# 2 Overall Description

## 2.1 Product Perspective

The Secure OTA Update Compiler is positioned between source code and firmware binary generation. It analyzes source-level update logic using AST and CFG representations before emitting compiled firmware.

**High-Level Flow:**
Input Source Code → Parsing → AST → CFG → Security Invariant Checks → Accept / Reject → Firmware Binary

## 2.2 User Classes and Characteristics

- **Firmware Developers**: Write update logic and receive compile-time errors on violations
- **Security Auditors**: Review compiler logs and invariant enforcement reports
- **System Integrators**: Integrate the compiler into build pipelines

## 2.3 Operating Environment

- Host OS: Linux
- Target: Embedded / IoT firmware
- Compiler Framework: LLVM/Clang or equivalent static analysis framework

## 2.4 Assumptions and Dependencies

- Firmware update logic is written in C/C++ or a supported language
- Update installation occurs through identifiable function calls
- Cryptographic verification APIs are statically identifiable

# 3 Functional Requirements

- **FR-1: Firmware Update Logic Detection**
  The compiler shall identify firmware update–related functions, including verification, version checks, and installation routines.
- **FR-2: Control Flow Analysis**
  The compiler shall construct a CFG for all update-related functions and identify all execution paths leading to firmware installation.

- **FR-3: Signature Verification Enforcement**
  The compiler shall ensure that cryptographic signature verification dominates firmware installation on all valid paths.
- **FR-4: Rollback Protection Enforcement**
  The compiler shall verify firmware version monotonicity checks prior to installation.
- **FR-5: Trusted Source Validation**
  The compiler shall ensure that firmware originates from a trusted and authenticated source.
- **FR-6: Sensitive Information Leakage Prevention**
  The compiler shall detect and reject update logic that logs or exposes sensitive firmware or cryptographic data.
- **FR-7: Cryptographic Policy Enforcement**
  The compiler shall reject usage of weak or disallowed cryptographic primitives within update logic.
- **FR-8: Compile-Time Rejection**
  The compiler shall terminate compilation with explicit error messages if any security invariant is violated.
- **FR-9: Enforcement Reporting**
  The compiler shall generate logs detailing enforced invariants and detected violations.

## 4 Non-Functional Requirements

- **NFR-1: Security** – Guaranteed sound enforcement
- **NFR-2: Performance** – Acceptable compilation overhead
- **NFR-3: Usability** – Precise and actionable error messages
- **NFR-4: Extensibility** – Easy addition of new invariants
- **NFR-5: Reliability** – Deterministic compilation results

## 5 External Interface Requirements

### 5.1 Compiler Interface

- Input: Firmware source code
- Output: Firmware binary or compilation error report

### 5.2 Logging Interface

Text-based logs describing invariant checks and violations.

## 6 Constraints

- Static analysis only (no runtime instrumentation)
- Language and framework constraints of the compiler infrastructure
- Limited visibility into hardware-enforced security mechanisms

## 7 Acceptance Criteria

The system is considered complete when:

- All defined invariants are enforced on all control-flow paths
- Insecure firmware update samples fail compilation
- Secure firmware update samples compile successfully

## 8 Future Enhancements

- Automated code instrumentation for compliant fixes
- Support for additional programming languages
- Integration with CI/CD pipelines