Week 6 Deliverables Report

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

---

1 Identifiable IR Generation

In this phase, the source C programs are translated into an intermediate representation (IR) that preserves both structural and semantic information of the original code. The generated IR provides a uniform and low-level view of program constructs such as functions, variables, control flow, and function calls, making them easier to analyze systematically. Each IR instruction is identifiable and traceable back to its corresponding source-level construct, which enables precise inspection of program behavior. This representation serves as a crucial bridge between source code parsing and later analysis stages, allowing subsequent modules to reliably traverse, inspect, and reason about security-relevant operations.

2 Logs Showing Detected Update-Related Code

The traversal module produces diagnostic logs during compilation to demonstrate successful detection of update-related logic. When the compiler pass encounters the firmware update routine, it prints structured messages indicating the presence of relevant function calls.

Example log output includes detection of:

- Cryptographic verification functions
- Trusted source validation checks
- Logging or debug operations
- Firmware installation calls

These logs provide verifiable evidence that the compiler is able to traverse the program structure and accurately identify security-critical operations associated with the OTA update process.

3 Working AST

The Week 6 milestone implements a working traversal module as part of the Secure OTA Update Compiler. The compiler operates on LLVM Intermediate Representation (IR) generated from C source code using the Clang frontend.

The traversal module is implemented as a function-level compiler pass that systematically visits all functions in the IR. When the firmware update routine (e.g., `updateFirmware`) is encountered, the pass traverses all basic blocks and instructions within the function. This enables identification of security-relevant operations such as verification routines, firmware installation calls, logging functions, and cryptographic API usage.

At this stage, the traversal is read-only and does not enforce security policies. It serves as a foundational component for later semantic analysis and compile-time enforcement.