Cryptographic S/W Modules with C

Design, Implementation, and Integration of Core Crypto Modules

Secure, Efficient, High-Performance Cryptographic Software Modules

Ji, Yong-hyeon
hacker3740@kookmin.ac.kr

Department of Cyber Security
Kookmin University

April 5, 2025

Contents

Chapter 1

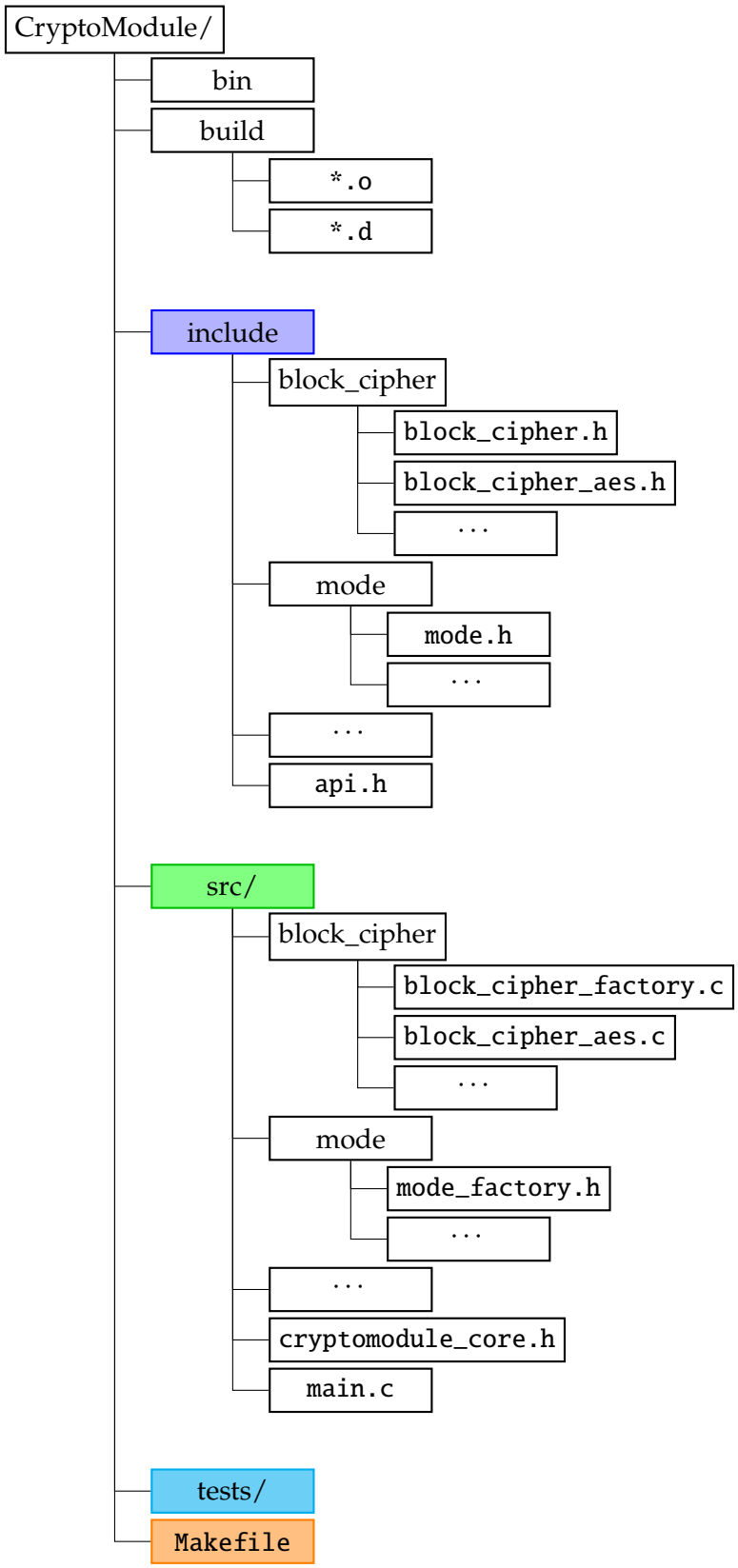
Project Overview

This document provides a comprehensive guide to the design, implementation, and integration of cryptographic modules written in C (sometimes assembly).

Key Objectives:

- Describing the cryptographic primitives and algorithms
  (block ciphers, hash functions, MACs, signature algorithms, etc.).
- Explaining the structure of the source files and headers.
- Providing guidelines for building, testing, and integrating these modules into larger software systems.

1.1 Directory Structure



The code base is organized to reflect modular cryptographic primitives and functionality. The main directories and their purposes are outlined below:

- **include/**: Contains all public headers for cryptographic modules.

  - `cryptomodule/block/`: Headers for block cipher implementations (e.g., AES).
  - `cryptomodule/mode/`: Headers for modes of operation (CBC, CTR, GCM, etc.).
  - `cryptomodule/rng/`: Headers for random number generators.
  - `cryptomodule/hash/`: Headers for hash functions (e.g., SHA-256, SHA-512).
  - `cryptomodule/mac/`: Headers for message authentication codes (e.g., HMAC).
  - `cryptomodule/kdf/`: Headers for key derivation functions (PBKDF, HKDF, etc.).
  - `cryptomodule/keysetup/`: Headers for key exchange primitives (ECDH).
  - `cryptomodule/sign/`: Headers for signature algorithms (ECDSA, RSA, etc.).
- **src/**: Contains the corresponding C/ASM source files for each cryptographic category.
  - `block/`, `mode/`, `rng/`, `hash/`, `mac/`, `kdf/`, `keysetup/`, `sign/`
- **tests/**: Houses unit tests and integration tests for all cryptographic modules.
- **Makefile**: Defines how to build and link the libraries and tests. Contains flags for C and ASM code.
- **README.md**: Provides a high-level overview of the project, including build instructions and usage examples.

1.1.1 Hierarchy and Relationships

Each functional category (block cipher, hash, etc.) is encapsulated in its own subdirectory to keep code organized and maintainable. Corresponding header files in `include/cryptomodule/` expose the public API, while the implementations in `src/` include both C and, where appropriate, ASM files for optimized routines.

1.2 Build Tools and Dependencies

A standard Unix-like build environment is assumed, with the following tools and dependencies required:

- **Compiler** (e.g., `gcc` or `clang`) with support for assembling inline or separate ASM files.
- **Make** (GNU Make) to use the provided `Makefile`.
- **CMake (optional)**: Some teams prefer CMake-based workflows; a `CMakeLists.txt` can also be maintained for cross-platform compatibility.
- **Perl/Python (optional)**: May be required for certain test scripts, code generation, or performance analysis scripts.
- **OpenSSL (optional)**: Useful for comparing test vectors or for using the system's cryptographic library as a reference.

When building the library, you can enable or disable specific optimizations or algorithms by modifying the `Makefile` (or `CMakeLists.txt`, if you choose to add one). For instance, enabling ASM routines for AES might require additional flags like:

```
1       CFLAGS += -march=native -maes
```

depending on your target CPU capabilities.

1.2.1 Environment Configuration

Before compiling, ensure that your development environment is set up with the correct paths. For instance:

```
export CC=gcc
export AS=nasm # or another assembler if preferred
export CFLAGS="-O2 -Wall -Wextra"
```

Adjust these variables as needed based on your local toolchain and performance requirements.

1.3 Coding Guidelines

All C code should follow a consistent style (e.g., `K&R` or `LLVM` style) with adequate comments explaining the purpose and usage of functions. Inline ASM or standalone ASM files should use readable label names, and macros must be well-documented to clarify any platform-specific instructions.

Furthermore, each function in the cryptographic modules should include:

- **Parameter validations**: Ensure pointers are not `NULL`, lengths are within expected ranges, etc.
- **Error handling**: Return clear error codes and avoid silent failures.
- **Security considerations**: Erase sensitive data buffers immediately after use to prevent leakage.

1.4 Security and Maintenance Policies

Because cryptographic libraries are critical to overall system security, the project maintains strict policies regarding:

- **Patch review**: All code changes are peer-reviewed to detect potential vulnerabilities or performance regressions.
- **Regular audits**: Scheduled internal and external audits are conducted to verify compliance with best security practices.
- **Versioning and backward compatibility**: Each stable release is tagged in version control, with major version increments for breaking changes.

Chapter 2

Cryptographic Module

2.1 Block Cipher Modules

2.2 Modes of Operation

2.3 Random Number Generator

2.4 Hash Functions

2.5 Message Authentication Codes

2.6 Key Derivation Functions

2.7 Key Exchange

2.8 Signature Algorithms

Chapter 3

Build and Integration

Chapter 4

Testing

Appendices