Cryptographic-Module Source-Code Development Manual

Design, Implementation, and Integration of Cryptography Modules

Secure, Efficient, High-Performance Cryptographic Software Modules

Ji, Yong-hyeon

`hacker3740@kookmin.ac.kr`

Department of Cyber Security
Kookmin University

May 3, 2025

Contents

Chapter 1

Project Overview

I have developed a cryptographic software module in the **C** language. 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, 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.

Section	Description	Status
1.1	Directory layout & development environment	Drafted
1.2	Development Environment	Drafted
1.3	TBA	TBA

1.1 Directory Structure

```
CryptoModule/
├── bin
├── build
│   ├── *.o
│   └── *.d
├── include
│   ├── block_cipher
│   │   ├── api_block_cipher.h
│   │   ├── block_cipher_aes.h
│   │   └── ...
│   ├── mode
│   │   ├── api_mode.h
│   │   ├── mode_gcm.h
│   │   └── ...
│   ├── ...
│   └── api_cryptomodule.h
├── src/
│   ├── block_cipher
│   │   ├── block_cipher_factory.c
│   │   ├── block_cipher_aes.c
│   │   └── ...
│   ├── mode
│   │   ├── mode_factory.c
│   │   ├── mode_gcm.c
│   │   └── ...
│   ├── ...
│   ├── cryptomodule_core.c
│   └── main.c
├── tests/
└── Makefile
```

1.2 Development Environment

- **Operating System: Linux Mint** (based on Debian and Ubuntu)

```
@>$ cat /etc/os-release
NAME="Linux Mint"
VERSION="21.3 (Virginia)"
ID=linuxmint
ID_LIKE="ubuntu debian"
PRETTY_NAME="Linux Mint 21.3"
VERSION_ID="21.3"
HOME_URL="https://www.linuxmint.com/"
SUPPORT_URL="https://forums.linuxmint.com/"
BUG_REPORT_URL="http://linuxmint-troubleshooting-guide.readthedocs.io/en/latest/"
PRIVACY_POLICY_URL="https://www.linuxmint.com/"
VERSION_CODENAME=virginia
UBUNTU_CODENAME=jammy
```

- **Compiler: GNU Compiler Collection** 11.4.0

```
@>$ gcc --version
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

- **Hardware: AMD Ryzen 7 5800X3D 8-Core Processor**

```
@>$ lscpu
Architecture: x86_64
  CPU op-mode(s): 32-bit, 64-bit
  Address sizes: 48 bits physical, 48 bits virtual
  Byte Order: Little Endian
CPU(s): 16
  On-line CPU(s) list: 0-15
Vendor ID: AuthenticAMD
  Model name: AMD Ryzen 7 5800X3D 8-Core Processor
    CPU family: 25
    Model: 33
    Thread(s) per core: 2
    CPU max MHz: 3400.0000
    CPU min MHz: 2200.0000
...
```

- **Additional Tools:**

  ‘valgrind‘ for memory checks,

```
@>$ \valgrind --version
valgrind-3.18.1
```

  ‘gdb‘ for debugging,

```
@>$ gdb --version
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

  and TBA

Chapter 2

Cryptographic Software Module

2.1 Block Cipher

A **block cipher** is a keyed family of permutations over a fixed-size data block.

- Let k be a fixed key size and n be a fixed block size.
- Let $\mathcal{K} = \{0,1\}^k$ be the set of possible k-bit keys (each key is chosen from this set).
- Let $\mathcal{M} = \{0,1\}^n$ be the set of all n-bit messages (plaintext blocks).
- Let $C = \{0,1\}^n$ be the set of all n-bit ciphertext blocks.

A **block cipher** is have two efficient induced functions:

$$E : \mathcal{K} \times \mathcal{M} \to C \quad \text{and} \quad D : \mathcal{K} \times C \to \mathcal{M},$$

referred to as the **encryption** and **decryption** functions, respectively. These must satisfy:

1. *Invertibility (permutation property)*: For each fixed key $k \in \mathcal{K}$, the encryption function

   $$E_k(\cdot) = E(k, \cdot) : \mathcal{M} \to C \quad \text{is a bijection (i.e., permutation) on } \{0,1\}^n.$$

   In other words, for every key k, there is a unique inverse $D_k(\cdot) = D(k, \cdot) : C \to \mathcal{M}$ s.t.

   $$D_k\big(E_k(m)\big) = m \quad \text{and} \quad E_k\big(D_k(c)\big) = c \quad \text{for every } m \in \mathcal{M} \text{ and } c \in C.$$

2. *Keyed operation*: The cipher's behavior depends on the choice of key k. Changing k results in a different permutation over the n-bit block space.

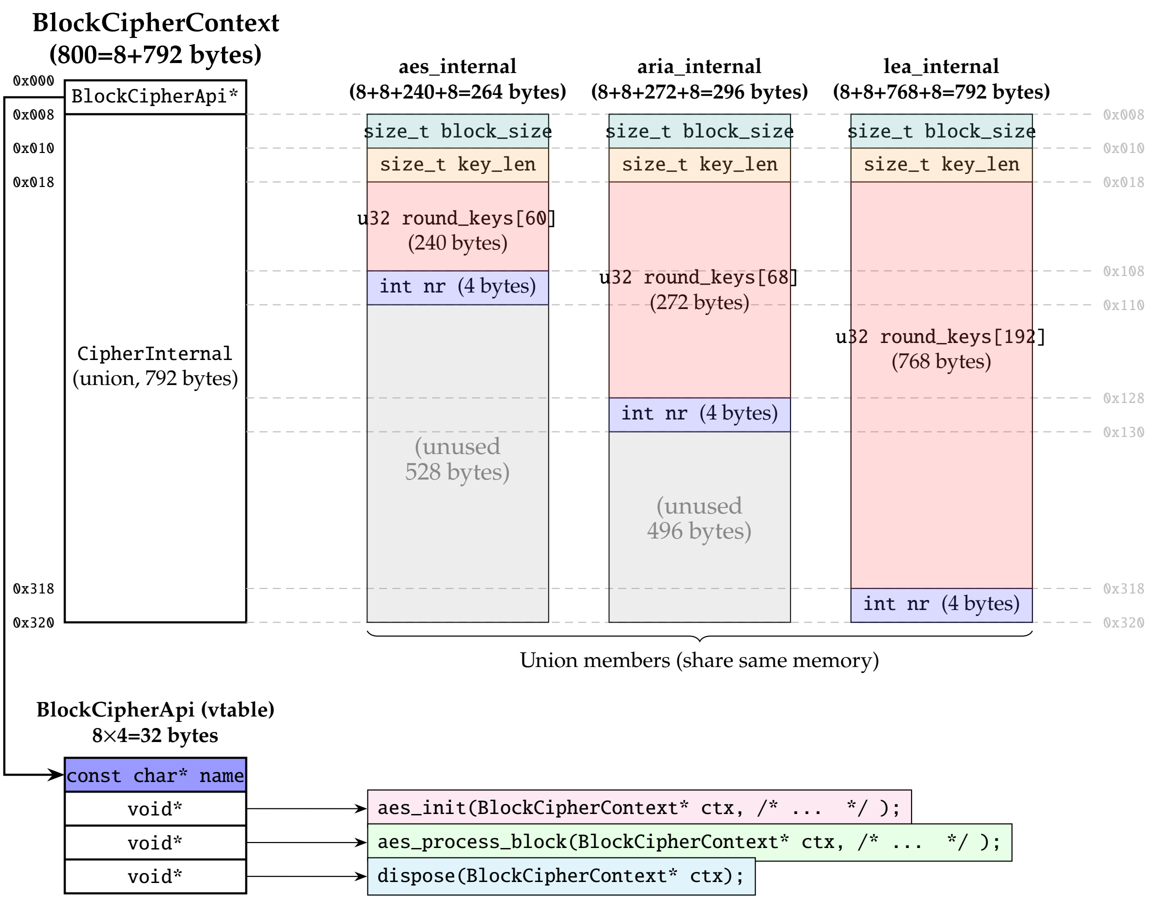
Alg.	$\boldsymbol{n}$ (bit)	$\boldsymbol{k}$ (bit)	**# of Rounds**	**RK Size** (bit)	**# of RKs**	**Total RK Size** (bit)
AES–128	128	128	10	128 (4-word)	11	1408 (44-word)
AES–192	128	192	12	128 (4-word)	13	1664 (52-word)
AES–256	128	256	14	128 (4-word)	15	1920 (60-word)
ARIA-128	128	128	12	128 (4-word)	13	1664 (52-word)
ARIA-192	128	192	14	128 (4-word)	15	1920 (60-word)
ARIA-256	128	256	16	128 (4-word)	17	2176 (68-word)
LEA-128	128	128	24	192 (6-word)	24	4608 (144-word)
LEA-192	128	192	28	192 (6-word)	28	5376 (168-word)
LEA-256	128	256	32	192 (6-word)	32	6144 (192-word)

Table 2.1: Comparison of AES, ARIA, and LEA parameters for 128-, 192-, and 256-bit keys.

```c
typedef struct __BlockCipherApi__ {
   const char *name;
   void (*init)(BlockCipherContext* ctx, /* ... */);
   void (*process_block)(BlockCipherContext* ctx, /* ... */);
   void (*dispose)(BlockCipherContext* ctx);
} BlockCipherApi;

typedef union __CipherInternal__ {
   struct __aes_internal__ {
      /* ... */
   } aes_internal;
   struct __aria_internal__ {
      /* ... */
   } aria_internal;
   struct __lea_internal__ {
      /* ... */
   } lea_internal;
} CipherInternal;

typedef struct __BlockCipherContext__ {
   const BlockCipherApi *api;
   CipherInternal internal_data; /* Generic internal state for any cipher */
} BlockCipherContext;
```



Code 2.1: include/block_cipher/api_block_cipher.h

```c
/* Forward declaration for the context. */
typedef struct __BlockCipherContext__ BlockCipherContext;

typedef struct __BlockCipherApi__ {
  const char *cipher_name; /* e.g. "AES" or "MyCipher" */

  block_cipher_status_t (*cipher_init)(
    BlockCipherContext* cipher_ctx,
    const u8* key,
    size_t key_len,
    size_t block_len,
    BlockCipherDirection dir);
  block_cipher_status_t (*cipher_process)(
    BlockCipherContext* cipher_ctx,
    const u8* in,
    u8* out,
    BlockCipherDirection dir);
  void (*cipher_dispose)(BlockCipherContext* cipher_ctx);
} BlockCipherApi;

typedef union __CipherInternal__ {
  struct __aes_internal__ {
    size_t block_size; /* Typically must be 16 for AES */
    size_t key_len; /* 16, 24, or 32 for AES-128/192/256 */
    /* max 60 for AES-256 */
    u32 round_keys[4 * (AES256_NUM_ROUNDS + 1)];
    int nr; /* e.g., 10 for AES-128, 12, or 14... */
  } aes_internal;
  struct __aria_internal__ {
    size_t block_size; /* Typically must be 16 for ARIA */
    size_t key_len; /* 16, 24, or 32 for ARIA-128/192/256 */
    /* max 68 for ARIA-256 */
    u32 round_keys[4 * (ARIA256_NUM_ROUNDS + 1)];
    int nr; /* e.g., 12 for ARIA-128, 14, or 16... */
  } aria_internal;
  struct __lea_internal__ {
    size_t block_size; /* Typically must be 16 for LEA */
    size_t key_len; /* 16, 24, or 32 for LEA-128/192/256 */
    /* max 192 for LEA-256 */
    u32 round_keys[6 * LEA256_NUM_ROUNDS];
    int nr; /* e.g., 24 for LEA-128, 28, or 32... */
  } lea_internal;
} CipherInternal;

struct __BlockCipherContext__ {
  const BlockCipherApi *cipher_api;
  CipherInternal cipher_state; /* Generic internal state for any cipher */
};
```

Subsection	Description	Status
2.1.1	AES (Advanced Encryption Standard)	Drafted
2.1.2	ARIA (Academy, Research Institute, and Agency)	Drafted
2.1.3	LEA (Lightweight Encryption Algorithm)	Drafted

2.1.1 AES (Advanced Encryption Standard)

Table 2.2: Parameters of the Block Cipher AES (1-word = 32-bit)

Alg.	n (bit)	k (bit)	**# of Rounds**	**RK Size** (bit)	**# of RKs**	**Total RK Size** (bit)
AES–128	128	128	10	128 (4-word)	11	1408 (44-word)
AES–192	128	192	12	128 (4-word)	13	1664 (52-word)
AES–256	128	256	14	128 (4-word)	15	1920 (60-word)

Code 2.2: include/block_cipher/block_cipher_aes.h

```
/* Get the AES block cipher vtable. */
const BlockCipherApi* get_aes_api(void);

void aes_set_encrypt_key(const u8 *key, size_t bytes, u32 *rk);
void aes_set_decrypt_key(const u8 *key, size_t bytes, u32 *rk);
void aes_encrypt(const u8 *in, u8 *out, const u32 *rk, int r);
void aes_decrypt(const u8 *in, u8 *out, const u32 *rk, int r);
```

Code 2.3: src/block_cipher/block_cipher_aes.c

```
/* Forward declarations of static functions. */
static block_cipher_status_t aes_init(
    BlockCipherContext *ctx,
    const u8 *key,
    size_t key_len,
    size_t block_len,
    BlockCipherDirection dir);
static block_cipher_status_t aes_process(
    BlockCipherContext *ctx,
    const u8 *in,
    u8 *out,
    BlockCipherDirection dir);
static void aes_dispose(BlockCipherContext *ctx);

/* The AES block cipher API. */
static const BlockCipherApi AES_API = {
    .cipher_name = "AES",
    .cipher_init = aes_init,
    .cipher_process = aes_process,
    .cipher_dispose = aes_dispose
};

/* Get the AES block cipher API. */
const BlockCipherApi *get_aes_api(void) { return &AES_API; }
```

2.1.2 ARIA (Academy, Research Institute, and Agency)

Table 2.3: Parameters of the Block Cipher ARIA (1-word = 32-bit)

Alg.	n (bit)	k (bit)	**# of Rounds**	**RK Size** (bit)	**# of RKs**	**Total RK Size** (bit)
ARIA–128	128	128	12	128 (4-word)	13	1664 (52-word)
ARIA–192	128	192	14	128 (4-word)	15	1920 (60-word)
ARIA–256	128	256	16	128 (4-word)	17	2176 (68-word)

Code 2.4: include/block_cipher/block_cipher_aria.h

```c
/* Get the ARIA block cipher vtable. */
const BlockCipherApi* get_aria_api(void);

void aria_set_encrypt_key(const u8 *key, size_t bytes, u32 *rk);
void aria_set_decrypt_key(const u8 *key, size_t bytes, u32 *rk);
void aria_encrypt(const u8 *in, u8 *out, const u32 *rk, int r);
void aria_decrypt(const u8 *in, u8 *out, const u32 *rk, int r);
```

Code 2.5: src/block_cipher/block_cipher_aria.c

```c
/* Forward declarations of static functions. */
static block_cipher_status_t aria_init(
   BlockCipherContext *ctx,
   const u8 *key,
   size_t key_len,
   size_t block_len,
BlockCipherDirection dir);
static block_cipher_status_t aria_process(
   BlockCipherContext *ctx,
   const u8 *in,
   u8 *out,
   BlockCipherDirection dir);
static void aria_dispose(BlockCipherContext *ctx);

/* The ARIA block cipher API. */
static const BlockCipherApi ARIA_API = {
   .cipher_name = "ARIA",
   .cipher_init = aria_init,
   .cipher_process = aria_process,
   .cipher_dispose = aria_dispose
};
/* Get the ARIA block cipher API. */
const BlockCipherApi* get_aria_api(void) { return &ARIA_API; }
```

2.1.3 LEA (Lightweight Encryption Algorithm)

Table 2.4: Parameters of the Block Cipher LEA (1-word = 32-bit)

Alg.	n (bit)	k (bit)	**# of Rounds**	**RK Size** (bit)	**# of RKs**	**Total RK Size** (bit)
LEA–128	128	128	24	192 (6-word)	24	4608 (144-word)
LEA–192	128	192	28	192 (6-word)	28	5376 (168-word)
LEA–256	128	256	32	192 (6-word)	32	6144 (192-word)

Code 2.6: include/block_cipher/block_cipher_aria.h

```
/* Get the LEA block cipher vtable. */
const BlockCipherApi* get_lea_api(void);

void lea_set_encrypt_key(const u8 *key, size_t bytes, u32 *rk);
void lea_set_decrypt_key(const u8 *key, size_t bytes, u32 *rk);
void lea_encrypt(const u8 *in, u8 *out, const u32 *rk, int r);
void lea_decrypt(const u8 *in, u8 *out, const u32 *rk, int r);
```

Code 2.7: src/block_cipher/block_cipher_aria.c

```
/* Forward declarations of static functions. */
static block_cipher_status_t lea_init(
   BlockCipherContext *ctx,
   const u8 *key,
   size_t key_len,
   size_t block_len,
   BlockCipherDirection dir);
static block_cipher_status_t lea_process(
   BlockCipherContext *ctx,
   const u8 *in,
   u8 *out,
   BlockCipherDirection dir);
static void lea_dispose(BlockCipherContext *ctx);

/* The LEA block cipher API. */
static const BlockCipherApi LEA_API = {
   .cipher_name = "LEA",
   .cipher_init = lea_init,
   .cipher_process = lea_process,
   .cipher_dispose = lea_dispose
};
/* Get the LEA block cipher API. */
const BlockCipherApi *get_lea_api(void) { return &LEA_API; }
```

2.2 Modes of Operation

```c
typedef struct __ModeOfOperationApi__ {
   const char *name;
   void (*init)( /* ... */ );
   void (*process)( /* ... */ );
   void (*dispose)( /* ... */ );
} ModeOfOperationApi;

typedef union __ModeInternal__ {
   struct __cbc_internal__ {
      /* ... */
   } cbc_internal;
   struct __ctr_internal__ {
      /* ... */
   } ctr_internal;
   struct __gcm_internal__ {
      /* ... */
   } gcm_internal;
   struct __ecb_internal__ {
      /* ... */
   } ecb_internal;

} ModeInternal;

typedef struct __ModeOfOperationContext__ {
   const ModeOfOperationApi *api; // Pointer to the mode API
   BlockCipherContext cipher_ctx; // Block cipher context
   ModeInternal internal_data; // Internal state for the mode
} ModeOfOperationContext;
```

2.2.1 Electronic Codebook (ECB)

TBA

2.2.2 Cipher Block Chaining (CBC)

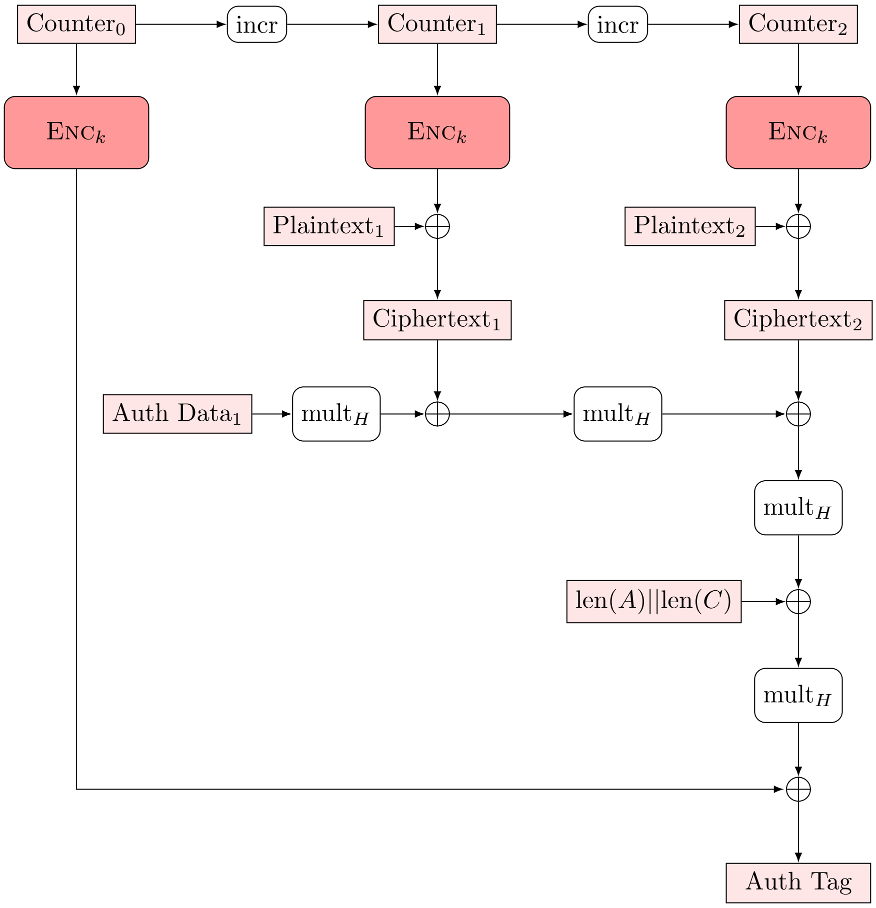
TBA

2.2.3 Counter (CTR)

TBA

2.3 Galois / Counter Mode (GCM)

References: [MV04]



2.3.1 Multiplication in GF(2^{128})

Definition 2.1. Let $\mathbb{F}_2 = \{0, 1\}$ be the field with two elements. Fix an irreducible polynomial

$$f(x) \;=\; x^{128} + x^7 + x^2 + x + 1 \quad \in \; \mathbb{F}_2[x].$$

Then

$$\mathrm{GF}(2^{128}) \;=\; \mathbb{F}_2[x] \,/\, \big(f(x)\big)$$

is the degree-128 binary extension field.

Remark 2.1. Every element $\alpha \in \mathrm{GF}(2^{128})$ can be written uniquely as

$$\alpha \;=\; a_{127}x^{127} + a_{126}x^{126} + \cdots + a_1 x + a_0 \quad (a_i \in \{0, 1\}).$$

We identify α with the 128-bit vector $(a_0, \ldots, a_{127}) \in \mathbb{F}_2$.

Polynomial Representation and Reduction Consider

$$\left(\sum_{i=0}^{127} a_i x^i\right)\left(\sum_{j=0}^{127} b_j x^j\right) \;=\; \sum_{i=0}^{127}\sum_{j=0}^{127}(a_i b_j)\, x^{i+j}, \quad \text{where each } a_i b_j \in \{0, 1\}.$$

The raw product has degree at most 254. To obtain an element of GF(2^{128}), we reduce it modulo $f(x) = x^{128} + x^7 + x^2 + x + 1$. Concretely, whenever a term x^k with $k \geq 128$ appears, one replaces

$$x^{128} \;\mapsto\; x^7 + x^2 + x + 1$$

and iterates until the remainder has degree ≤ 127.

Bit-Level Algorithm We implement multiplication by a simple "shift-and-add" method with reduction on each shift, often called `gf128_xtime`. For $v \in \mathrm{GF}(2^{128})$ represented as a 128-bit, define a function

$$\texttt{gf128\_xtime} : \mathrm{GF}(2^{128}) \to \mathrm{GF}(2^{128}), \quad v \mapsto xv.$$

for each $v \in \mathrm{GF}(2^{128})$. Since

$$\begin{aligned}
\texttt{gf128_xtime}(v) &= x \cdot v = x \cdot (a_0 a_1 \cdots a_7 \parallel a_8 a_9 \cdots a_{15} \parallel \cdots \parallel a_{120} a_{121} \cdots a_{127}) \\
&= x \cdot (a_0 + a_1 x + \cdots + a_{126} x^{126} + a_{127} x^{127}) \\
&= a_0 x + a_1 x^2 + \cdots + a_{126} x^{127} + a_{127} x^{128} \\
&= a_0 x + a_1 x^2 + \cdots + a_{126} x^{127} + a_{127} (x^7 + x^2 + x + 1) \\
&= \begin{cases} (0 a_1 \cdots a_6 \oplus \texttt{00000000}) \parallel a_7 \cdots a_{14} \parallel \cdots \parallel a_{119} a_{120} \cdots a_{126} & \text{if } a_{127} = 0 \\ (0 a_1 \cdots a_6 \oplus \texttt{11100001}) \parallel a_7 \cdots a_{14} \parallel \cdots \parallel a_{119} a_{120} \cdots a_{126} & \text{if } a_{127} = 1 \end{cases},
\end{aligned}$$

we have

$$\texttt{gf128\_xtime}(v) = \begin{cases} v \gg 1, & \text{if the MSB of } v \text{ is } 0, \\ (v \gg 1) \oplus \texttt{0xE1}, & \text{if the MSB of } v \text{ is } 1, \end{cases}$$

where `0xE1` is the bit-vector corresponding to the reduction polynomial $x^7 + x^2 + x + 1$.

```c
/* v <- vx mod (x^128 + x^7 + x^2 + x + 1) */
void gf128_xtime(uint8_t v[16]) {
  uint8_t t = v[15] & 1;    // p[15] = p120...p127; extract p127
  // Shift the 128-bit value right by 1:
  for (int i = 15; i > 0; --i)
    v[i] = (v[i] >> 1) | ((v[i-1] & 1) << 7);
  p[0] >>= 1;        // p0p1...p7 -> 0p0...p6
  if (t) v[0] ^= 0xE1; // // 0p0...p6 ^ 11100001
}
```

```c
/*
* Compute p(x) <- p(x)q(x) over GF(2^128) using "shift-and-add" multiplication.
*/
void gf128_mul(u8 p[16], u8 q[16]) {
  u8 buffer[16] = { 0x00, }; // accumulator for the product
  u8 bit_mask; // mask for each bit of q
  for (int i = 0; i < 16; ++i) { // Loop over each byte of q (Q[0] = q120...q127)
    for (int j = 0; j < 8; ++j) { // Process bits q(128-8i-1) down to q(128-8i-8)
      bit_mask = q[i] & (1 << (7 - j));
      if (bit_mask) {
        // If the current bit of q is 1, XOR the current p(x) into buffer
        for (int k = 0; k < 16; ++k) { buffer[k] ^= p[k]; }
      }
      gf128_xtime(p); // Multiply p(x) by x (i.e. shift-and-reduce) for next bit
    }
  }
  // Write the accumulated product back into p[0...15]
  for (int i = 0; i < 16; ++i) { p[i] = buffer[i]; }
}
```

2.3.2 Efficient Multiplication in $\mathrm{GF}(2^{128})$

TBA

2.4 Random Number Generator

TBA

2.5 Hash Functions

2.5.1 SHA-2 Algorithms

TBA

2.5.2 SHA-3 Algorithms

TBA

2.5.3 Lightweight Secure Hash (LSH)

TBA

2.6 Message Authentication Codes

TBA

2.7 Key Derivation Functions

TBA

2.8 Diffie-Hellman Key Exchange

TBA

2.9 Signature Algorithms

TBA

Chapter 3

Build and Integration

3.1 Makefile Configuration and Overview

This section describes the build system for the CryptoModule demo, driven by a single GNU Makefile. It covers compiler settings, directory layout, source discovery, and all available targets.

3.1.1 Compiler, Flags, and Directories

```
# Compiler and flags
CC := gcc
CFLAGS := -std=c99 -g -O2 -Wall -Wextra -I. -Iinclude -Isrc

# Executable name
TARGET := cryptomodule-demo

# Output directories
OBJ_DIR := build
BIN_DIR := bin
```

- `gcc` in C99 mode, with debug symbols (`-g`) and optimization (`-O2`).
- Warnings enabled (`-Wall -Wextra`), include paths set for project headers.
- Object files placed under `build/`, preserving the `src/` subdirectory structure; final binary in `bin/`.

3.1.2 Automatic Source and Object Discovery

```
# Find all .c files in src/ recursively
SRCS := $(shell find src -name '*.c')

# Map src/foo.c -> build/foo.o
OBJS := $(patsubst src/%.c,$(OBJ_DIR)/%.o,$(SRCS))
```

3.1.3 Usage Examples

```
################################################################################
# 1) build : compile + link
################################################################################
build: $(BIN_DIR)/$(TARGET)

# Link step: gather all objects into a single executable
$(BIN_DIR)/$(TARGET): $(OBJS)
	@echo "[LINK] Linking objects to create $@"
	@mkdir -p $(BIN_DIR)
	$(CC) $(CFLAGS) $^ -o $@
# Compile step: For each .c -> .o
$(OBJ_DIR)/%.o: src/%.c
	@echo "[CC] Compiling $< into $@"
	@mkdir -p $(dir $@)
	$(CC) $(CFLAGS) -c $< -o $@
################################################################################
# 2) run : run the resulting binary
################################################################################
run: build
	@echo "[RUN] Running $(BIN_DIR)/$(TARGET)"
	@./$(BIN_DIR)/$(TARGET)

################################################################################
# 3) clean : remove build artifacts
################################################################################
clean:
@echo "[CLEAN] Removing build artifacts..."
	rm -rf $(OBJ_DIR) $(BIN_DIR)
	@echo "[CLEAN] Removing *.req and *.rsp files in testvectors folder..."
	find testvectors -type f \( -name '*.req' -o -name '*.rsp' \) -delete

################################################################################
# 4) rebuild : clean + build
################################################################################
rebuild: clean build

################################################################################
# 5) valgrind : run the binary under Valgrind for memory checking
################################################################################
valgrind: build
	@echo "[VALGRIND] Running Valgrind..."
	valgrind --leak-check=full ./$(BIN_DIR)/$(TARGET)
```

`make build` Compile (`.c → .o`) and link (`.o → executable`).

`make run` Build if necessary, then execute `bin/cryptomodule-demo`.

`make clean` Remove `build/`, `bin/`, and any `*.req`/`*.rsp` in `testvectors/`.

`make rebuild` Alias for `clean` followed by `build`.

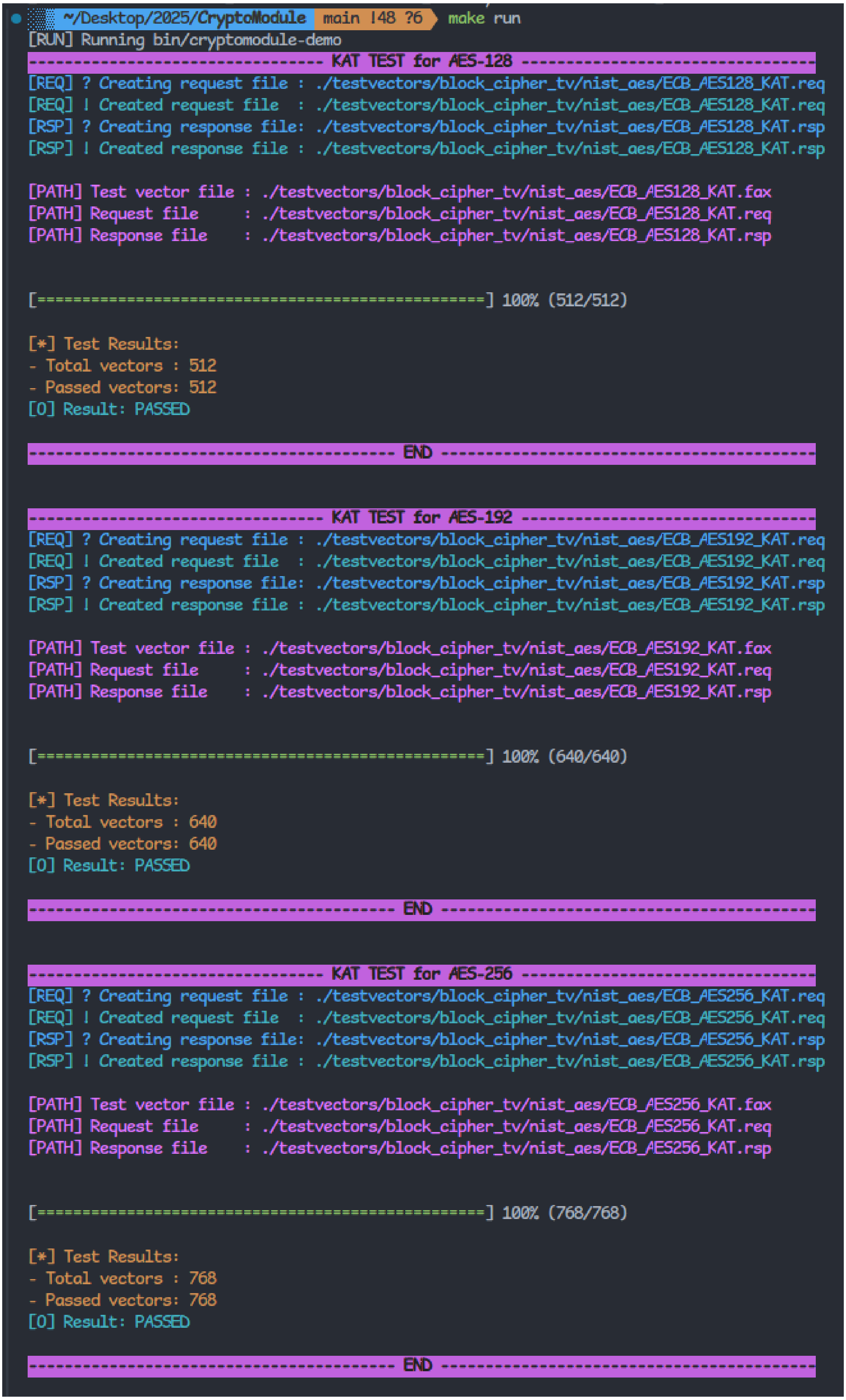
`make valgrind` Build, then run under Valgrind for memory-leak checks.

3.2 Example: Main Function for Block-Cipher KATs

Code 3.1: Invoke known-answer tests for AES block ciphers

```
int main(void) {
	KAT_TEST_BLOCKCIPHER(BLOCK_CIPHER_AES128);
	KAT_TEST_BLOCKCIPHER(BLOCK_CIPHER_AES192);
	KAT_TEST_BLOCKCIPHER(BLOCK_CIPHER_AES256);
	return 0;
}
```

```
@>$ make rebuild
@>$ make run
```



```
~/Desktop/2025/CryptoModule  main !48 ?6  make run
[RUN] Running bin/cryptomodule-demo
--------------------------------- KAT TEST for AES-128 ---------------------------------
[REQ] ? Creating request file : ./testvectors/block_cipher_tv/nist_aes/ECB_AES128_KAT.req
[REQ] ! Created request file  : ./testvectors/block_cipher_tv/nist_aes/ECB_AES128_KAT.req
[RSP] ? Creating response file: ./testvectors/block_cipher_tv/nist_aes/ECB_AES128_KAT.rsp
[RSP] ! Created response file : ./testvectors/block_cipher_tv/nist_aes/ECB_AES128_KAT.rsp

[PATH] Test vector file : ./testvectors/block_cipher_tv/nist_aes/ECB_AES128_KAT.fax
[PATH] Request file     : ./testvectors/block_cipher_tv/nist_aes/ECB_AES128_KAT.req
[PATH] Response file    : ./testvectors/block_cipher_tv/nist_aes/ECB_AES128_KAT.rsp


[==================================================] 100% (512/512)

[*] Test Results:
- Total vectors : 512
- Passed vectors: 512
[O] Result: PASSED

----------------------------------------- END -----------------------------------------


--------------------------------- KAT TEST for AES-192 ---------------------------------
[REQ] ? Creating request file : ./testvectors/block_cipher_tv/nist_aes/ECB_AES192_KAT.req
[REQ] ! Created request file  : ./testvectors/block_cipher_tv/nist_aes/ECB_AES192_KAT.req
[RSP] ? Creating response file: ./testvectors/block_cipher_tv/nist_aes/ECB_AES192_KAT.rsp
[RSP] ! Created response file : ./testvectors/block_cipher_tv/nist_aes/ECB_AES192_KAT.rsp

[PATH] Test vector file : ./testvectors/block_cipher_tv/nist_aes/ECB_AES192_KAT.fax
[PATH] Request file     : ./testvectors/block_cipher_tv/nist_aes/ECB_AES192_KAT.req
[PATH] Response file    : ./testvectors/block_cipher_tv/nist_aes/ECB_AES192_KAT.rsp


[==================================================] 100% (640/640)

[*] Test Results:
- Total vectors : 640
- Passed vectors: 640
[O] Result: PASSED

----------------------------------------- END -----------------------------------------


--------------------------------- KAT TEST for AES-256 ---------------------------------
[REQ] ? Creating request file : ./testvectors/block_cipher_tv/nist_aes/ECB_AES256_KAT.req
[REQ] ! Created request file  : ./testvectors/block_cipher_tv/nist_aes/ECB_AES256_KAT.req
[RSP] ? Creating response file: ./testvectors/block_cipher_tv/nist_aes/ECB_AES256_KAT.rsp
[RSP] ! Created response file : ./testvectors/block_cipher_tv/nist_aes/ECB_AES256_KAT.rsp

[PATH] Test vector file : ./testvectors/block_cipher_tv/nist_aes/ECB_AES256_KAT.fax
[PATH] Request file     : ./testvectors/block_cipher_tv/nist_aes/ECB_AES256_KAT.req
[PATH] Response file    : ./testvectors/block_cipher_tv/nist_aes/ECB_AES256_KAT.rsp


[==================================================] 100% (768/768)

[*] Test Results:
- Total vectors : 768
- Passed vectors: 768
[O] Result: PASSED

----------------------------------------- END -----------------------------------------
```

Bibliography

[MV04] David A. McGrew and John Viega. The galois/counter mode of operation (gcm). Technical report, Submission to NIST Modes of Operation Process, Cisco Systems, Inc. and Secure Software, January 2004. Initial version posted January 15, 2004.

Appendices

TBA