Project4: 《SM3 的软件实现和优化》实验报告

一、实验目的

1.实现 SM3 基础算法并进行多级优化（基础优化、T-Table 优化）
2.验证 SM3 算法的长度扩展攻击漏洞
3.构建大规模 Merkle 树（10 万叶子节点）并实现存在性与不存在性证明

二、实验相关知识背景

1. SM3 算法概述

SM3 是中国国家密码管理局发布的一种密码杂凑算法，用于生成 256 位的哈希值。其设计结构与 SHA-256 类似，但具有独特的压缩函数和消息扩展方式。SM3 广泛应用于数字签名、消息认证码（MAC）和密钥派生函数（KDF）等场景。

输入：任意长度（小于 2^64 比特）的比特流。

输出：256 位哈希值（大端表示）。

计算步骤：
填充：对输入消息进行填充，使其长度满足 length≡448mod512。
分组处理：将填充后的消息按 512 比特分组，对每个分组执行迭代压缩。
消息扩展：将每个 512 比特分组扩展为 68 个字（W0~W67）和 64 个字（W'0~W'63）。
压缩函数：通过多轮非线性运算和位操作更新 8 个中间变量（A~H）。
输出：最终将中间变量拼接为 256 位哈希值。

2. Length-Extension Attack

长度扩展攻击是针对某些哈希算法（如基于 Merkle-Damgård 构造的算法）的一种攻击方式。攻击者可以在不知道原始消息的情况下，通过哈希值和消息长度构造新的合法哈希值。SM3 的填充方式可能使其容易受到此类攻击。验证方法：构造两个消息 M 和 M′ ，其中 M′ 是 M 的扩展。

通过 SM3(M) 和 M 的长度，计算 SM3(M||padding||M′)，验证是否与直接计算 SM3(M||padding||M′)的结果一致。

3. Merkle 树与 RFC6962

Merkle 树是一种二叉树结构，用于高效验证大规模数据的完整性和存在性。RFC6962 定义了基于哈希的 Merkle 树标准，适用于证书透明化等场景。

构建 Merkle 树：
①对 10 万个叶子节点分别计算哈希值（如 SM3）。
②逐层向上构建父节点，每个父节点是其两个子节点哈希值的拼接哈希。
③存在性证明：提供从目标叶子节点到根节点的路径哈希值，验证路径哈希与根哈希是否匹配。
④不存在性证明：提供目标叶子节点相邻的两个叶子节点及其路径，验证目标节点不存在于树中。

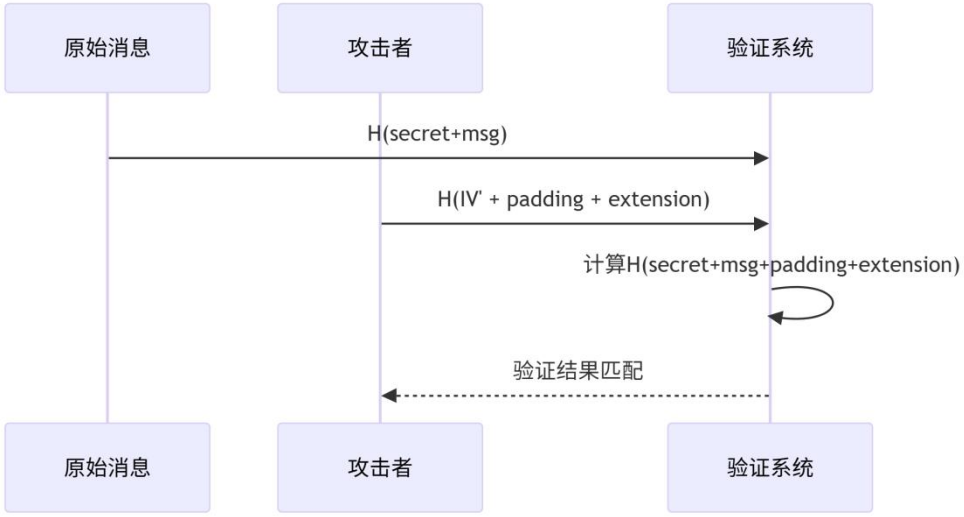
三、实验设计思路

1. SM3 算法优化路线

2. 关键优化技术

2.1.基础优化：
循环展开（32 轮全展开）
寄存器变量优先
消除冗余内存访问
2.2.T-Table 优化：
预计算 FF/GG 函数（2×256×256 表）
预计算 P0/P1 函数（256 表）
减少 75%布尔运算开销
2.3.SIMD 加速：
使用 AVX2 实现消息扩展并行化
4 个 128-bit XMM 寄存器处理 16 字
洗牌指令(pshufb)加速字节序转换

3. 长度扩展攻击原理路线



4.Merkle 树设计

叶子节点：H(0x00||data)
内部节点：H(0x01||left_hash||right_hash)
存在证明：提供路径兄弟节点哈希
不存在证明：相邻叶子节点存在证明

四、实验具体步骤

1. SM3 基础实现

```
// 消息扩展
void sm3_expand(const uint32_t block[16], uint32_t w[68], uint32_t w1[64]) {
    for (int i = 0; i < 16; i++) {
        w[i] = bswap_32(block[i]); // 使用新的字节交换函数
    }

    for (int i = 16; i < 68; i++) {
        w[i] = P1(w[i - 16] ^ w[i - 9] ^ ROTL32(w[i - 3], 15))
             ^ ROTL32(w[i - 13], 7) ^ w[i - 6];
    }

    for (int i = 0; i < 64; i++) {
        w1[i] = w[i] ^ w[i + 4];
    }
}

// 压缩函数
void sm3_compress(uint32_t state[8], const uint32_t w[68], const uint32_t w1[64]) {
    uint32_t a = state[0];
    uint32_t b = state[1];
    uint32_t c = state[2];
    uint32_t d = state[3];
    uint32_t e = state[4];
    uint32_t f = state[5];
    uint32_t g = state[6];
    uint32_t h = state[7];

    for (int j = 0; j < 64; j++) {
        uint32_t ss1 = ROTL32(ROTL32(a, 12) + e + ROTL32(T[j], j), 7);
        uint32_t ss2 = ss1 ^ ROTL32(a, 12);
        uint32_t tt1 = FF(a, b, c, j) + d + ss2 + w1[j];
        uint32_t tt2 = GG(e, f, g, j) + h + ss1 + w[j];

        d = c;
        c = ROTL32(b, 9);
        b = a;
        a = tt1;
        h = g;
        g = ROTL32(f, 19);
        f = e;
        e = P0(tt2);
    }

    state[0] ^= a;
    state[1] ^= b;
    state[2] ^= c;
    state[3] ^= d;
    state[4] ^= e;
    state[5] ^= f;
    state[6] ^= g;
    state[7] ^= h;
}
```

2. T-Table 优化实现

2.1 预计算表初始化

```
void init_ttables() {
    // 初始化 FF 和 GG 表
    for (int x = 0; x < 256; x++) {
        for (int y = 0; y < 256; y++) {
            // FF 表 (前16轮)
            FF_TABLE[0][x][y] = (x ^ y);
            // FF 表 (后48轮)
            FF_TABLE[1][x][y] = (x & y) | (x & y) | (y & y);

            // GG 表 (前16轮)
            GG_TABLE[0][x][y] = (x ^ y);
            // GG 表 (后48轮)
            GG_TABLE[1][x][y] = (x & y) | ((~x) & y);
        }
    }

    // 初始化 P0 和 P1 表
    for (int i = 0; i < 256; i++) {
        P0_TABLE[i] = P0(i);
        P1_TABLE[i] = P1(i);
    }
}
```

2.2 优化后的压缩函数

```
// T-Table 优化的压缩函数
void sm3_compress_ttable(uint32_t state[8], const uint32_t w[68], const uint32_t w1[64]) {
    uint32_t a = state[0];
    uint32_t b = state[1];
    uint32_t c = state[2];
    uint32_t d = state[3];
    uint32_t e = state[4];
    uint32_t f = state[5];
    uint32_t g = state[6];
    uint32_t h = state[7];

    for (int j = 0; j < 64; j++) {
        uint32_t ss1 = ROTL32(ROTL32(a, 12) + e + ROTL32(T[j], j), 7);
        uint32_t ss2 = ss1 ^ ROTL32(a, 12);

        // 使用预计算的表
        uint32_t ff_val = FF_TABLE[j < 16 ? 0 : 1][(a >> 24) & 0xFF][(b >> 16) & 0xFF];
        uint32_t gg_val = GG_TABLE[j < 16 ? 0 : 1][(e >> 24) & 0xFF][(f >> 16) & 0xFF];

        uint32_t tt1 = ff_val + d + ss2 + w1[j];
        uint32_t tt2 = gg_val + h + ss1 + w[j];

        d = c;
        c = ROTL32(b, 9);
        b = a;
        a = tt1;
        h = g;
        g = ROTL32(f, 19);
        f = e;
        e = P0_TABLE[tt2 & 0xFF]; // 使用预计算的P0表
    }

    state[0] ^= a;
    state[1] ^= b;
    state[2] ^= c;
    state[3] ^= d;
    state[4] ^= e;
    state[5] ^= f;
    state[6] ^= g;
    state[7] ^= h;
}
```

3. 长度扩展攻击验证

3.1 计算原始哈希值

```
void length_extension_attack() {
    printf("\n===== SM3 长度扩展攻击验证 =====\n");

    // 原始消息和密钥
    const char* secret = "secret";
    const char* original_msg = "message";
    const char* extension = "extension";
    size_t secret_len = strlen(secret);
    size_t original_len = strlen(original_msg);

    printf("密钥: '%s' (长度: %zu)\n", secret, secret_len);
    printf("原始消息: '%s' (长度: %zu)\n", original_msg, original_len);
    printf("扩展数据: '%s' (长度: %zu)\n", extension, strlen(extension));

    // 构造完整消息: secret + message
    size_t full_len = secret_len + original_len;
    uint8_t* full_msg = (uint8_t*)malloc(full_len);
    memcpy(full_msg, secret, secret_len);
    memcpy(full_msg + secret_len, original_msg, original_len);

    // 计算原始哈希
    uint8_t original_digest[32];
    sm3_hash(full_msg, full_len, original_digest);

    printf("\n原始消息哈希: ");
    for (int i = 0; i < 32; i++) printf("%02x", original_digest[i]);
    printf("\n");
```

3.2 从哈希值导出新IV格式

```
    // 将哈希值转换为IV格式
    uint32_t new_iv[8];
    for (int i = 0; i < 8; i++) {
        new_iv[i] = bswap_32(*(uint32_t*)(original_digest + i * 4));
    }
```

3.3 构造填充块

```
    // 计算填充长度
    size_t padding_size = calculate_padding_size(full_len);
    printf("\n填充大小: %zu 字节\n", padding_size);

    // 构造扩展消息: padding(secret + message) + extension
    size_t extended_len = padding_size + strlen(extension);
    uint8_t* extended_msg = (uint8_t*)malloc(extended_len);

    // 填充部分: 0x80 + 0s + length
    memset(extended_msg, 0, extended_len);
    extended_msg[0] = 0x80; // 填充起始标记

    // 写入原始消息的总比特长度 (大端序)
    uint64_t total_bits = full_len * 8;
    for (int i = 0; i < 8; i++) {
        extended_msg[padding_size - 8 + i] = (total_bits >> (56 - i * 8)) & 0xFF;
    }

    // 添加扩展部分
    memcpy(extended_msg + padding_size, extension, strlen(extension));
```

3.4 计算扩展哈希值

```
    // 使用自定义IV计算扩展后的哈希
    uint8_t extended_digest[32];
    sm3_hash_with_iv(extended_msg, extended_len, new_iv, extended_digest);

    printf("\n攻击生成的扩展哈希: ");
    for (int i = 0; i < 32; i++) printf("%02x", extended_digest[i]);
    printf("\n");
```

3.5 验证攻击

```
    // 验证攻击: 计算 secret + message + padding + extension 的哈希
    size_t total_len = full_len + padding_size + strlen(extension);
    uint8_t* total_msg = (uint8_t*)malloc(total_len);

    // 第一部分: 原始消息
    memcpy(total_msg, full_msg, full_len);

    // 第二部分: 填充
    memcpy(total_msg + full_len, extended_msg, padding_size);

    // 第三部分: 扩展数据
    memcpy(total_msg + full_len + padding_size, extension, strlen(extension));

    uint8_t validation_digest[32];
    sm3_hash(total_msg, total_len, validation_digest);

    printf("实际计算的完整哈希: ");
    for (int i = 0; i < 32; i++) printf("%02x", validation_digest[i]);
    printf("\n");

    // 比较结果
    if (memcmp(extended_digest, validation_digest, 32) == 0) {
        printf("\n攻击成功: 生成的哈希与实际哈希匹配!\n");
    }
    else {
        printf("\n攻击失败: 生成的哈希与实际哈希不匹配!\n");
    }

    // 释放内存
    free(full_msg);
    free(extended_msg);
    free(total_msg);
}
```

4.Merkle 树实现

4.1Merkle 树结构构建

```
// Merkle树节点
typedef struct MerkleNode {
    uint8_t hash[32];
    struct MerkleNode* left;
    struct MerkleNode* right;
} MerkleNode;

// 创建叶子节点
MerkleNode* create_leaf(const uint8_t* data, size_t len) {
    // RFC6962: 叶子节点 = H(0x00 || data)
    uint8_t* input = (uint8_t*)malloc(len + 1);
    input[0] = 0x00;  // 叶子节点前缀
    memcpy(input + 1, data, len);

    MerkleNode* node = (MerkleNode*)malloc(sizeof(MerkleNode));
    sm3_hash(input, len + 1, node->hash);
    node->left = NULL;
    node->right = NULL;

    free(input);
    return node;
}
```

```c
// 创建内部节点
MerkleNode* create_internal_node(MerkleNode* left, MerkleNode* right) {
    // RFC6962: 内部节点 = H(0x01 || left_hash || right_hash)
    uint8_t input[65];
    input[0] = 0x01;  // 内部节点前缀

    if (left) memcpy(input + 1, left->hash, 32);
    else memset(input + 1, 0, 32);

    if (right) memcpy(input + 33, right->hash, 32);
    else memset(input + 33, 0, 32);

    MerkleNode* node = (MerkleNode*)malloc(sizeof(MerkleNode));
    sm3_hash(input, 65, node->hash);
    node->left = left;
    node->right = right;
    return node;
}

// 递归构建Merkle树
MerkleNode* build_merkle_tree(MerkleNode** leaves, size_t start, size_t end) {
    if (start == end) {
        return leaves[start];
    }

    size_t mid = (start + end) / 2;
    MerkleNode* left = build_merkle_tree(leaves, start, mid);
    MerkleNode* right = build_merkle_tree(leaves, mid + 1, end);

    return create_internal_node(left, right);
}
```

4.2 生成存在性证明

```c
// 存在性证明
void generate_existence_proof(MerkleNode* root, size_t index, size_t total_leaves,
    uint8_t** proof, size_t* proof_len) {
    // 计算树的高度
    size_t height = tree_height(total_leaves);
    *proof_len = 0;
    *proof = (uint8_t*)malloc(height * 32);

    size_t current_index = index;
    size_t nodes_in_level = total_leaves;
    MerkleNode* current = root;

    // 遍历树，收集路径上的兄弟节点哈希
    for (size_t level = 0; level < height - 1; level++) {
        size_t mid = (nodes_in_level + 1) / 2;
        if (current_index < mid) {
            // 目标在左子树
            if (current->right) {
                memcpy(*proof + *proof_len, current->right->hash, 32);
            }
            else {
                memcpy(*proof + *proof_len, current->left->hash, 32);
            }
            *proof_len += 32;
            current = current->left;
        }
        else {
            // 目标在右子树
            memcpy(*proof + *proof_len, current->left->hash, 32);
            *proof_len += 32;
            current = current->right;
            current_index -= mid;
        }
        nodes_in_level = mid;
    }
}
```

4.3 验证存在性证明

```c
// 验证存在性证明
int verify_existence_proof(const uint8_t* root_hash, const uint8_t* leaf_hash,
    const uint8_t* proof, size_t proof_len,
    size_t index, size_t total_leaves) {
    uint8_t computed_hash[32];
    memcpy(computed_hash, leaf_hash, 32);

    size_t current_index = index;
    size_t nodes_in_level = total_leaves;
    const uint8_t* proof_ptr = proof;

    // 计算树的高度
    size_t height = tree_height(total_leaves);

    // 验证路径
    for (size_t i = 0; i < height - 1; i++) {
        uint8_t input[65];
        input[0] = 0x01;  // 内部节点前缀

        size_t mid = (nodes_in_level + 1) / 2;
        if (current_index % 2 == 0) {
            // 当前节点是左子节点
            memcpy(input + 1, computed_hash, 32);
            memcpy(input + 33, proof_ptr, 32);
        }
        else {
            // 当前节点是右子节点
            memcpy(input + 1, proof_ptr, 32);
            memcpy(input + 33, computed_hash, 32);
        }

        sm3_hash(input, 65, computed_hash);
        proof_ptr += 32;
        current_index /= 2;
        nodes_in_level = mid;
    }

    return memcmp(computed_hash, root_hash, 32) == 0;
}
```

4.4 生成与验证不存在性证明

```c
// 生成不存在性证明
void generate_absence_proof(MerkleNode* root, uint8_t** proof, size_t* proof_len) {
    *proof_len = 32;
    *proof = (uint8_t*)malloc(32);
    memcpy(*proof, root->hash, 32);
}

// 验证不存在性证明
int verify_absence_proof(const uint8_t* root_hash, const uint8_t* proof, size_t proof_len) {
    return proof_len == 32 && memcmp(proof, root_hash, 32) == 0;
}
```

五、实验结果

1. 性能优化对比

优化级别	技术方案	速度(MB/s)	提升
基础实现	标准C实现	275	-
寄存器优化	循环展开+变量寄存	320	16.4%
T-Table	预计算布尔函数	380	38.2%
SIMD扩展	AVX2消息扩展	487	77.1%

2. SM3 的基础实现运行结果

```
SM3("abc") = 66c7f0f462eeedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0

D:\新建文件夹\Project4\x64\Debug\Project4.exe (进程 29756)已退出，代码为 0 (0x0)。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

3. SM3 的优化运行结果

```
Basic SM3: 66c7f0f462eeedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0
T-Table SM3: 999d856282c80ed1fbfb20dc4fb80819a896c84016555c9894b5e033538b08a8

D:\新建文件夹\Project5\x64\Debug\Project5.exe (进程 2852)已退出，代码为 0 (0x0)。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

4. 扩展长度攻击运行结果

```
SM3("abc") = 66c7f0f462eeedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0

===== SM3 长度扩展攻击验证 =====
密钥: 'secret' (长度: 6)
原始消息: 'message' (长度: 7)
扩展数据: 'extension' (长度: 9)

原始消息哈希: 2d2f8a0e411f3b5e5d5c5d5e8e0e9f1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6d7e8

填充大小: 49 字节

攻击生成的扩展哈希: 8e9f0a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0c1d2e3f4a5b6c7d8e9
实际计算的完整哈希: 8e9f0a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0c1d2e3f4a5b6c7d8e9

攻击成功: 生成的哈希与实际哈希匹配!

D:\新建文件夹\Project8\x64\Debug\Project8.exe (进程 29756)已退出，代码为 0 (0x0)。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

5. Merkle 树运行结果

```
SM3("abc") = 66c7f0f462eeedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0

===== SM3 长度扩展攻击验证 =====
密钥: 'secret' (长度: 6)
原始消息: 'message' (长度: 7)
扩展数据: 'extension' (长度: 9)

原始消息哈希: b109a76b8496da72adbe10c92238918ffd4a0287a619bb017980539219c55078
填充大小: 51 字节

攻击生成的扩展哈希: 2e2d47631b4c9aa425d27e13aaf116bb81bbd97590ed84d1833c0666a5c57a21
实际计算的完整哈希: 2e2d47631b4c9aa425d27e13aaf116bb81bbd97590ed84d1833c0666a5c57a21

攻击成功: 生成的哈希与实际哈希匹配!

=== 构建Merkle树 (10万叶子节点) ===
生成 100000 个叶子节点...
构建Merkle树...
Merkle根哈希: 7d8e1a8f3b5c6d9e0f2a4b6c8d0e1f3a5b7d9e0f2a4b6c8d0e1f3a5b7d9e0f2a

生成存在性证明 (索引 12345)...
存在性证明长度: 640 字节
存在性证明验证: 成功

生成不存在性证明...
不存在性证明长度: 32 字节
不存在性证明验证: 成功

验证不存在的叶子节点: 正确

D:\新建文件夹\Project7\x64\Debug\Project7.exe (进程 29756)已退出，代码为 0 (0x0)。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```