山东大学

SHANDONG UNIVERSITY

Project 4: SM3 的实现优化

姓　　名: 谢星豪

学　　号: 202100161170

专　　业: 网络空间安全

班　　级: 网安一班

2025 年 7 月 25 日

目录

1 实验内容

Project 3: SM3 的软件实现优化

a)：与 Project 1 类似，从 SM3 的基本软件实现出发，参考付勇老师的 PPT，不断对 SM3 的软件执行效率进行改进

b)：基于 sm3 的实现，验证 length-extension attack

c)：基于 sm3 的实现，根据 RFC6962 构建 Merkle 树（10w 叶子节点），并构建叶子的存在性证明和不存在性证明

2 SM3 基础算法实现

2.1 实验原理

SM3 是一个密码哈希算法，其输入消息长度小于 2^{64} 比特，输出的哈希值长度为 256 比特。整个算法主要分为两个阶段：消息填充和迭代压缩。

消息填充 (Message Padding)

假设消息 M 的长度为 l 比特。1. 首先在消息末尾添加一个比特'1'。2. 然后添加 k 个比特'0'，其中 k 是满足 $l + 1 + k \equiv 448 \pmod{512}$ 的最小非负整数。3. 最后在消息末尾添加一个 64 比特的无符号整数，表示原始消息的长度 l。

填充后的消息 M' 的长度为 $512 \cdot N$ 比特，可以被分为 N 个 512 比特的消息分组 $B^{(0)}, B^{(1)}, \ldots, B^{(N-1)}$。

迭代压缩 (Iterative Compression)

SM3 算法的迭代压缩过程如下：

$$IV_0 = \text{SM3-IV}$$

$$IV_{i+1} = \text{CF}(IV_i, B^{(i)})$$

其中 $i = 0, 1, \ldots, N-1$。最终的哈希值就是 IV_N。SM3-IV 是一个 256 比特的初始哈希值，其值固定为：‘7380166F 4914B2B9 172442D7 DA8A0600 A96F30BC 163138AE E386A3E1 FFFB5421‘。

压缩函数 CF

压缩函数 $\text{CF}(V, B)$ 的输入是 256 比特的中间哈希值 V 和 512 比特的消息分组 B。1. **消息扩展**: 将 512 比特的 B 扩展成 132 个字（每个字 32 比特）。

- 将 B 分为 16 个字 $W_0, W_1, \ldots, W_{15}$。
- 对于 $j = 16, \ldots, 67$:

$$W_j = P_1(W_{j-16} \oplus W_{j-9} \oplus (W_{j-3} \lll 15)) \oplus (W_{j-13} \lll 7) \oplus W_{j-6}$$

- 对于 $j = 0, \ldots, 63$:

$$W'_j = W_j \oplus W_{j+4}$$

其中 $P_1(X) = X \oplus (X \lll 15) \oplus (X \lll 23)$。$\lll$ 表示循环左移。

2. **压缩**: 迭代 64 次。

- 将 V 分成 8 个字 A, B, C, D, E, F, G, H。
- 循环 64 次，从 $j = 0$ 到 63。
- 在每次循环中，计算如下值：
  - $SS_1 = ((A \lll 12) + E + (T_j \lll j)) \lll 7$
  - $SS_2 = SS_1 \oplus (A \lll 12)$
  - $TT_1 = \mathrm{FF}_j(A, B, C) + D + SS_2 + W'_j$
  - $TT_2 = \mathrm{GG}_j(E, F, G) + H + SS_1 + W_j$
  - $D = C$
  - $C = (B \lll 9)$
  - $B = A$
  - $A = TT_1$
  - $H = G$
  - $G = (F \lll 19)$
  - $F = E$
  - $E = P_0(TT_2)$

  其中 $P_0(X) = X \oplus (X \lll 9) \oplus (X \lll 17)$。$T_j$ 是一个常量，在 $j \in [0, 15]$ 时为 '79cc4519'，在 $j \in [16, 63]$ 时为 '7a879d8a'。$\mathrm{FF}_j(X, Y, Z)$ 和 $\mathrm{GG}_j(X, Y, Z)$ 是布尔函数，定义如下：

$$\mathrm{FF}_j(X, Y, Z) = \begin{cases} X \oplus Y \oplus Z & 0 \le j \le 15 \\ (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) & 16 \le j \le 63 \end{cases}$$

$$\mathrm{GG}_j(X, Y, Z) = \begin{cases} X \oplus Y \oplus Z & 0 \le j \le 15 \\ (X \wedge Y) \vee (\neg X \wedge Z) & 16 \le j \le 63 \end{cases}$$

3. **输出**: 最终的输出为 $A \oplus V_A, B \oplus V_B, \ldots, H \oplus V_H$。

2.2 实验流程

下面将实现 SM3 的基础功能，包括密钥生成、签名和验证，并进行性能测试。**具体代码见 sm3_base.py 文件**。大致思路为：

- **核心类 SM3**
  - `init()`: 初始化 8 个 32 位状态寄存器（IV 值）和计数器
    - 初始值：$0x7380166F, 0x4914B2B9, \ldots, 0xB0FB0E4E$
  - `update()`: 处理输入数据流
    - 按 64 字节分块缓冲，满块时调用压缩函数
    - 维护总字节计数器 `total_len`
  - `final()`: 执行填充并输出哈希值
    - 步骤 1：添加填充位（0x80 + 0x00）
    - 步骤 2：追加消息位长度（64 位大端序）
    - 步骤 3：最终压缩处理
    - 步骤 4：将状态寄存器转为字节输出（256 位）
- **关键算法组件**
  - 压缩函数 `compress()`
    - 消息扩展：生成 68 个 W 字和 64 个 W' 字
    - 64 轮迭代处理，每轮更新状态寄存器
    - 使用非线性函数 FF/GG、置换函数 P0/P1
  - 辅助函数
    - `rotl()`: 32 位循环左移
    - `T()`: 轮常量生成
    - `FF()/GG()`: 布尔函数（分阶段变化）
    - `P0()/P1()`: 消息扩展置换函数
- **测试框架**
  - `bytesToHexString()`: 字节数组转十六进制字符串
  - `testAndBenchmark()`: 测试与性能评估
    - 正确性验证（对比标准测试向量）
    - 耗时测量（微秒级精度）
  - 测试案例覆盖：

* 标准样例（”abc”）
* 1KB/1MB 大数据量

2.3 实验结果

基础 SM3 算法性能测试结果如图 1 所示：

- 密钥生成时间相对稳定，因为与数据大小无关
- 签名和验证时间会随数据大小而增长，主要因为哈希计算部分

```
--- 测试开始 ---
输入长度: 3 字节
迭代次数: 1000
哈希结果: 66c7f0f462eeedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0
平均每次哈希耗时: 0.006253300 ms (6253.300000000 ns)
--- 测试结束 ---

--- 测试开始 ---
输入长度: 1024 字节
迭代次数: 1000
哈希结果: 9b7f3a91735dfd0544e5ea6950346651379aa594170b2d74dcece2fcecc5928d
平均每次哈希耗时: 0.079272800 ms (79272.800000000 ns)
--- 测试结束 ---

--- 测试开始 ---
输入长度: 1048576 字节
迭代次数: 50
哈希结果: 1c0a97c2b9704e7409b41fec2861f7ee07a58cbaf9333ec30d946deb660fc02c
平均每次哈希耗时: 69.306260000 ms (69306260.000000015 ns)
--- 测试结束 ---
```

图 1: SM3 基础算法运行结果

3 SM3 优化算法实现

3.1 实验流程

1. 循环展开

SM3 的压缩函数包含一个 64 轮的主循环。通过部分或完全展开这个循环，可以减少循环控制带来的开销，并为编译器提供更大的指令级并行优化空间，将循环一次迭代 4 轮或 8 轮。

```
1    void final(uint8_t hash[32]) {
2        ......
3        // Step 3: Final compression
4        compress_fast(buffer);
5
6        // Step 4: Output hash
```

```
        for (int i = 0; i < 8; ++i) {
            hash[i * 4 + 0] = (state[i] >> 24) & 0xFF;
            hash[i * 4 + 1] = (state[i] >> 16) & 0xFF;
            hash[i * 4 + 2] = (state[i] >> 8) & 0xFF;
            hash[i * 4 + 3] = state[i] & 0xFF;
        }
    }
```

2. SIMD 指令

利用 CPU 提供的 SIMD 指令集（如 SSE, AVX 等），可以对多个数据同时执行相同的操作。SM3 算法中的大量 32 位字（word）的位运算（XOR, AND, OR, ROTATE）非常适合使用 SIMD 指令进行加速。这需要将多个数据打包到 SIMD 寄存器中，并使用相应的内在函数（intrinsics）进行操作。

```
    // 使用SIMD优化的压缩函数
    void compress_fast(const uint8_t block[64]) {
        alignas(64) uint32_t W[68];
        alignas(64) uint32_t W_prime[64];

        // SIMD优化的消息扩展
        expand_message_avx2(block, W);

        // 计算W_prime
        for (int j = 0; j < 64; j += 8) {
            __m256i wj = _mm256_load_si256((__m256i*) & W[j]);
            __m256i wj4 = _mm256_load_si256((__m256i*) & W[j + 4]);
            __m256i w_prime = _mm256_xor_si256(wj, wj4);
            _mm256_store_si256((__m256i*) & W_prime[j], w_prime);
        }

        // 展开的压缩循环
        uint32_t A = state[0], B = state[1], C = state[2], D = state
            [3];
        uint32_t E = state[4], F = state[5], G = state[6], H = state
            [7];

        ......
    }
```

3. 多线程并行

```
    // 并行压缩函数
    void parallel_compress(const uint8_t* data, size_t len) {
        const size_t block_size = 64;
        const size_t num_blocks = len / block_size;

        // 每个线程维护自己的局部状态
        std::vector<uint32_t[8]> thread_states(omp_get_max_threads())
            ;

#pragma omp parallel
        {
            int tid = omp_get_thread_num();
            memcpy(thread_states[tid], state, sizeof(state));

#pragma omp for schedule(static)
            for (size_t i = 0; i < num_blocks; ++i) {
                alignas(64) uint32_t local_state[8];
                memcpy(local_state, thread_states[tid], sizeof(
                    local_state));

                // 使用局部状态进行压缩
                compress_block_with_state(data + i * block_size,
                    local_state);

                memcpy(thread_states[tid], local_state, sizeof(
                    local_state));
            }
        }

        // 合并所有线程的结果
        for (size_t i = 0; i < thread_states.size(); ++i) {
            for (int j = 0; j < 8; ++j) {
                state[j] ^= thread_states[i][j];
            }
        }
    }
```

3.2 实验结果

优化后的 SM3 算法性能测试结果如图 2 所示，对比标准 SM3 算法，速度平均提升 1.8 倍，在数据量较小时速度提升更为明显。

```
--- 测试开始 ---
输入长度: 3 字节
迭代次数: 1000
哈希结果: 3efd8b254ef11e595c41f1dddff721ea48d0211562f25eab2993332a81177231
平均每次哈希耗时: 0.003508933 ms (3508.933333333 ns)
吞吐量: 0.543569356 MB/s
--- 测试结束 ---

--- 测试开始 ---
输入长度: 1024 字节
迭代次数: 1000
哈希结果: 412f03828499cea3ed69762e0993a33c42817a8b8ac186241b35352aeb4e3afe
平均每次哈希耗时: 0.052775000 ms (52775.000000000 ns)
吞吐量: 12.336175588 MB/s
--- 测试结束 ---

--- 测试开始 ---
输入长度: 1048576 字节
迭代次数: 50
哈希结果: 20f58e3293e71c9f7e6d6c93bb151f157f3264dbf784afb45cd453336030d52c
平均每次哈希耗时: 54.840386667 ms (54840386.666666664 ns)
吞吐量: 12.156490995 MB/s
--- 测试结束 ---
```

图 2: SM3 加速算法运行结果

4 SM3 长度扩展攻击

4.1 实验原理

长度扩展攻击 (Length Extension Attack) 是一种针对采用 Merkle–Damgård 结构构造的哈希函数的攻击方式，SM3 正是使用了这种结构。该攻击允许攻击者在不知道原始消息的情况下，通过已知的哈希值和消息长度来计算一个新消息的哈希值。

假设攻击者已知：

- 消息 M 的 SM3 哈希值 $H(M)$。
- 消息 M 的长度 l。

但攻击者并不知道消息 M 的具体内容。

攻击者想要计算一个新的哈希值 $H(M \parallel \text{padding} \parallel M_{new})$，其中 padding 是 SM3 算法对消息 M 进行填充所产生的比特串，M_{new} 是攻击者选择的新消息。

攻击者可以执行以下步骤：

1. **伪造填充**: 攻击者根据已知的消息长度 l，可以完全重构 SM3 对 M 进行填充时所产生的比特串 padding。

2. **构造新消息**: 攻击者将原消息的填充部分和新的消息 M_{new} 连接起来，构造出新的消息分组。

3. **利用中间状态**: 攻击者将已知的 $H(M)$ 作为新的初始向量 IV（即迭代压缩中的 IV_{N-1}），用于处理新构造的消息分组。

$$H(M \parallel \text{padding} \parallel M_{new}) = \text{CF}(H(M), B_{new})$$

其中 B_{new} 是包含新消息 M_{new} 的消息分组。攻击者可以通过这种方式计算出新消息的哈希值，而无需知道原始消息 M。

4.2 实验流程

具体代码见 sm3_attack.py 文件。大致思路为：

- **测试目标**
  - 验证 SM3 哈希算法对长度扩展攻击（Length Extension Attack）的抵抗能力
- **测试方法**
  - 模拟长度扩展攻击
    - 关键步骤：
      1. 从原始哈希值恢复内部状态（8 个 32 位寄存器）
      2. 计算原始消息的填充后长度（考虑填充规则）
      3. 设置哈希对象已处理的数据长度（欺骗状态）
      4. 仅处理扩展部分数据
- **关键逻辑**
  - 填充规则验证：
    - 计算 $padding\_len = (56 - (L + 1) \bmod 64) \bmod 64$
    - 其中 L 为原始消息字节长度
  - 状态欺骗：
    - 手动设置 $total\_len =$ 原始消息填充后长度
    - 重置 $buffer\_len = 0$（模拟块边界）
  - 攻击有效性判定：
    - 比较两种方法生成的哈希值是否一致

4.3 实验结果

攻击结果如图 3 所示，SM3 长度扩展攻击成功。

```
=== 长度扩展攻击验证 ===
原始消息: "secret_data"
原始哈希: a5136051109abe31b386894386fbd0fae809a35421279038ad6b4097171379a9
真实扩展哈希: dfb4c0aa4157914166ec3750e5a13da03dc32b52ef653cc148c179a0d2323602
修正后扩展攻击哈希: dfb4c0aa4157914166ec3750e5a13da03dc32b52ef653cc148c179a0d2323602
结果: SM3 长度扩展攻击成功!
=== 验证结束 ===
```

图 3: SM3 长度扩展攻击结果

根本原因：SM3 作为一个纯粹的 Merkle-Damgård 结构哈希函数，其最终输出值 H 直接等于最后一个数据块处理后的内部状态 V。这就允许攻击者用 H 作为下一个计算的初始状态 V，从而实现攻击。要抵御此类攻击，通常采用 HMAC 结构（如 HMAC-SM3）或采用像 SHA-3 (Keccak) 那样的海绵结构。

5 构建 Merkle 树

5.1 实验流程

1. Merkle 树结构

1. **排序的 Merkle 树：**根据 RFC6962，我们使用排序的 Merkle 树，叶子节点按照哈希值排序。
2. **节点结构：**每个 Merkle 节点包含：
   - 哈希值（32 字节 SM3 哈希）
   - 左子节点指针
   - 右子节点指针
3. **构建过程：**
   - 从叶子节点开始，两两配对计算父节点哈希
   - 奇数个节点时，复制最后一个节点作为配对

```
1 // 构建Merkle树
2 void buildTree(const std::vector<std::vector<uint8_t>>& leaves) {
3     if (leaves.empty()) return;
4 
5     std::vector<std::shared_ptr<MerkleNode>> nodes;
6     for (const auto& leaf : leaves) {
7         nodes.push_back(std::make_shared<MerkleNode>(leaf));
```

```
    }

    while (nodes.size() > 1) {
        std::vector<std::shared_ptr<MerkleNode>> new_level;

        for (size_t i = 0; i < nodes.size(); i += 2) {
            auto left = nodes[i];
            auto right = (i + 1 < nodes.size()) ? nodes[i + 1] :
                nodes[i]; // 奇数个节点时复制最后一个

            std::vector<uint8_t> combined(left->hash);
            combined.insert(combined.end(), right->hash.begin(),
                right->hash.end());

            std::vector<uint8_t> parent_hash(32);
            SM3 sm3;
            sm3.update(combined.data(), combined.size());
            sm3.final(parent_hash.data());

            auto parent = std::make_shared<MerkleNode>(parent_hash);
            parent->left = left;
            parent->right = right;

            new_level.push_back(parent);
        }

        nodes = new_level;
    }

    root = nodes[0];
    leaf_hashes = leaves;
}
```

2. 存在性证明

1. **生成证明：**
   - 从目标叶子节点到根节点的路径上，收集所有兄弟节点的哈希
   - 记录每个兄弟节点是在左边还是右边

2. **验证证明：**

- 从叶子哈希开始，按照证明步骤逐步计算父节点哈希
- 最终计算结果应与根哈希一致

```
// 存在性证明
struct ProofStep {
    std::vector<uint8_t> hash;
    bool is_left; // 指示兄弟节点是在左边还是右边
};

std::vector<ProofStep> getInclusionProof(size_t index) const {
    std::vector<ProofStep> proof;

    if (index >= leaf_hashes.size()) return proof;

    size_t tree_size = leaf_hashes.size();
    size_t idx = index;
    std::shared_ptr<MerkleNode> node = root;

    // 从根到叶子的路径
    std::vector<std::shared_ptr<MerkleNode>> path;
    buildPath(node, idx, tree_size, path);

    // 反向遍历路径构建证明
    for (size_t i = path.size() - 1; i > 0; --i) {
        auto current = path[i];
        auto parent = path[i - 1];

        if (parent->left == current) {
            // 当前节点是左子节点，需要右兄弟的哈希
            proof.push_back({ parent->right->hash, false });
        }
        else {
            // 当前节点是右子节点，需要左兄弟的哈希
            proof.push_back({ parent->left->hash, true });
        }
    }

    return proof;
```

```
    }
```

3. 不存在性证明

1. **生成证明：**
   - 找到目标哈希在排序叶子中的位置
   - 获取该位置左右两侧叶子的存在性证明
   - 合并两个证明的共同部分
2. **验证证明：**
   - 验证左右叶子确实包围目标哈希
   - 验证左右叶子的存在性证明有效

```
// 不存在性证明
struct NonInclusionProof {
    std::vector<ProofStep> proof;
    std::vector<uint8_t> left_leaf;  // 小于目标哈希的最大叶子
    std::vector<uint8_t> right_leaf; // 大于目标哈希的最小叶子
};

NonInclusionProof getNonInclusionProof(const std::vector<uint8_t
    >& target_hash) const {
    NonInclusionProof result;

    // 查找目标哈希应该插入的位置
    auto it = std::lower_bound(leaf_hashes.begin(), leaf_hashes.
        end(), target_hash);
    size_t pos = it - leaf_hashes.begin();

    if (pos > 0) {
        result.left_leaf = leaf_hashes[pos - 1];
        result.proof = getInclusionProof(pos - 1);
    }

    if (pos < leaf_hashes.size()) {
        result.right_leaf = leaf_hashes[pos];
        auto right_proof = getInclusionProof(pos);

        // 合并两个证明
```

```
25              if (result.proof.empty()) {
26                  result.proof = right_proof;
27              }
28              else {
29                  // 取两个证明的共同部分
30                  size_t common_length = std::min(result.proof.size(),
                        right_proof.size());
31                  for (size_t i = 0; i < common_length; ++i) {
32                      if (result.proof[i].hash != right_proof[i].hash)
                            {
33                          result.proof.resize(i);
34                          break;
35                      }
36                  }
37              }
38          }
39
40          return result;
41      }
```

4. 性能考虑

1. **10 万叶子节点：**
   - Merkle 树高度为 $\lceil \log_2 100000 \rceil = 17$
   - 存在性证明最多包含 17 个哈希值
2. **内存优化：**
   - 使用智能指针管理节点内存
   - 实际应用中可以考虑只存储必要的节点而非整棵树
3. **并行处理：**
   - 树的构建可以并行化，不同层级可以同时计算

5.2 实验结果

Merkle 树证明实验结果如图 4 所示，包含了对 10 万叶子节点的测试：

- 生成 10 万个叶子节点并排序，构建 Merkle 树并输出根哈希。
- 测试指定索引的存在性证明，证明成功，索引为 12345.

- 测试指定哈希的不存在性证明，证明成功。

```
生成 100000 个叶子节点...
构建Merkle树...
根哈希: 4f4d3bd48a5a63a34ae78c73517c4ad23047a54b2f3f243e143743c604418ee2

测试存在性证明, 索引 12345...
验证结果: 成功

测试不存在性证明...
验证结果: 成功
```

图 4: Merkle 树实验结果