网络空间安全创新创业实践

Project4

姓　　名: 张治瑞
学　　号: 202200210078
班　　级: 网安 22.1 班
学　　院: 网络空间安全学院

2025 年 8 月 10 日

目录

1 实验环境

处理器	Intel(R) Core(TM) i9-14900HX 2.20 GHz
机载 RAM	16.0 GB (15.6 GB 可用)
Windows 版本	Windows 11

2 实验题目

SM3 的软件实现与优化

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

3 实验目的

1. 深入理解 SM3 密码散列算法的工作原理和实现细节
2. 通过代码优化技术提升 SM3 算法的执行效率
3. 验证 SM3 算法存在的长度扩展攻击漏洞
4. 基于 SM3 算法构建 RFC6962 标准的 Merkle 树数据结构
5. 实现并验证 Merkle 树的存在性证明和不存在性证明机制
6. 掌握密码学算法在实际数据结构中的工程应用

4 实验原理

4.1 SM3 密码散列算法原理

SM3 是中国国家密码管理局发布的密码散列算法标准，产生 256 位的散列值。其主要特点包括：

- **消息分组**：将输入消息按 512 位（64 字节）分组处理
- **消息扩展**：将每个 512 位消息分组扩展为 68 个 32 位字 $W_0, W_1, \ldots, W_{67}$
- **压缩函数**：通过 64 轮迭代压缩运算更新 8 个 32 位状态字
- **初始向量**：使用固定的 256 位初始值作为起始状态

SM3 的核心压缩函数包含以下关键操作：

$$W_j = \begin{cases} M_j^{(i)} & 0 \le j \le 15 \\ P_1(W_{j-16} \oplus W_{j-9} \oplus (W_{j-3} \lll 15)) \oplus (W_{j-13} \lll 7) \oplus W_{j-6} & 16 \le j \le 67 \end{cases} \tag{1}$$

$$W'_j = W_j \oplus W_{j+4}, \quad 0 \le j \le 63 \tag{2}$$

其中 $P_1(X) = X \oplus (X \lll 15) \oplus (X \lll 23)$ 为置换函数。

4.2 长度扩展攻击原理

长度扩展攻击利用了 Merkle-Damgård 结构散列函数的特性。攻击者在已知 $H(secret \parallel message)$ 和 $message$ 长度的情况下，可以计算出 $H(secret \parallel message \parallel padding \parallel extra)$ 而无需知道 $secret$ 的内容。

攻击步骤如下：

1. 根据已知的散列值恢复内部状态
2. 计算原始消息的填充长度
3. 以恢复的状态为起点继续散列额外数据

4.3 Merkle 树原理

Merkle 树是一种二叉树数据结构，具有以下特性：

- **叶子节点**：存储实际数据的散列值
- **内部节点**：存储其子节点散列值的散列
- **根节点**：整个树的唯一标识符

根据 RFC6962 标准：

$$\text{叶子节点散列} = H(0x00 \parallel \text{data}) \tag{3}$$

$$\text{内部节点散列} = H(0x01 \parallel \text{left\_hash} \parallel \text{right\_hash}) \tag{4}$$

5 代码实现及分析

5.1 SM3 算法核心实现

5.1.1 基础压缩函数实现

基础实现严格按照 SM3 标准进行，代码结构清晰易懂：

Listing 1:

```
void SM3::compress_basic(const uint8_t block[SM3_BLOCK_SIZE]) {
    uint32_t W[68], W_prime[64];
    uint32_t A, B, C, D, E, F, G, H;

    // 1. 消息扩展
    for ( int j = 0; j < 16; j++) {
        GET_UINT32_BE(W[j], block, j * 4);
    }
    for ( int j = 16; j < 68; j++) {
        W[j] = P1(W[j-16] ^ W[j-9] ^ ROTL(W[j-3], 15))
               ^ ROTL(W[j-13], 7) ^ W[j-6];
    }

    // 2. 64轮压缩迭代
    for ( int j = 0; j < 64; j++) {
        // 压缩函数主体
        SS1 = ROTL(ROTL(A, 12) + E + ROTL(T_j, j), 7);
        // ... 其他压缩步骤
    }
}
```

5.1.2 优化压缩函数实现

优化版本采用以下策略提升性能：

- **循环展开**：将 64 轮循环分为前 16 轮和后 48 轮，后 48 轮采用 4 轮为一组的展开
- **减少中间变量**：直接计算避免不必要的临时存储
- **预计算优化**：提前计算常用的表达式

Listing 2:

```
void SM3::compress_optimized(const uint8_t block[SM3_BLOCK_SIZE]) {
    // 分段处理：前16轮使用一种布尔函数
    for ( int j = 0; j < 16; j++) {
        T_j = ROTL(T_00_15, j);
        SS1 = ROTL(ROTL(A, 12) + E + T_j, 7);
        // ... 压缩计算
    }

    // 后48轮循环展开优化
    for ( int j = 16; j < 64; j += 4) {
        // 一次处理4轮，减少循环开销
        // ... 展开的4轮计算
    }
}
```

5.2 长度扩展攻击实现

Listing 3:

```
std::vector<uint8_t> SM3::length_extension_attack(
    const std::vector<uint8_t>& original_hash,
    uint64_t original_len,
    const std::vector<uint8_t>& extra_data)
{
    SM3 attacker_sm3;
    uint32_t internal_state[8];

    // 从已知散列值恢复内部状态
    for( int i = 0; i < 8; ++i) {
        GET_UINT32_BE(internal_state[i], original_hash.data(), i * 4);
    }

    // 计算填充后的长度
    uint64_t padded_len = ((original_len + 8) / SM3_BLOCK_SIZE + 1)
                          * SM3_BLOCK_SIZE;

```

```
    // 以伪造的状态和长度初始化
    attacker_sm3.init_with_state(internal_state, padded_len);

    // 继续散列额外数据
    attacker_sm3.update(extra_data);
    return attacker_sm3.final();
}
```

5.3 Merkle 树实现分析

5.3.1 树构建算法

采用自底向上的构建策略，时间复杂度为 $O(n)$：

Listing 4:

```
MerkleTree::MerkleTree(const std::vector<std::vector<uint8_t>>& leaves_data
    ) {
    // 1. 创建叶子节点层
    std::vector<std::unique_ptr<MerkleNode>> current_level;
    for (const auto& data : leaves_data) {
        auto node = std::make_unique<MerkleNode>();
        node-> hash = hash_leaf(data);  // H(0x00 || data)
        current_level.push_back(std::move(node));
    }

    // 2. 自底向上构建
    while (current_level.size() > 1) {
        if (current_level.size() % 2 != 0) {
            // 奇数节点时复制最后一个
            auto last_node = std::make_unique<MerkleNode>();
            last_node-> hash = current_level.back()-> hash;
            current_level.push_back(std::move(last_node));
        }

        // 构建上一层
        std::vector<std::unique_ptr<MerkleNode>> next_level;
        for (size_t i = 0; i < current_level.size(); i += 2) {
            auto parent = std::make_unique<MerkleNode>();
```

```
            parent-> hash = hash_internal_node(
                current_level[i]-> hash,
                current_level[i+1]-> hash
            );
            next_level.push_back(std::move(parent));
        }
        current_level = std::move(next_level);
    }
}
```

5.3.2 证明机制实现

存在性证明通过收集从叶子到根的路径上所有兄弟节点实现：

Listing 5:

```
MerkleProof MerkleTree::get_inclusion_proof(size_t leaf_index) {
    MerkleProof proof;
    MerkleNode* current = leaf_nodes[leaf_index];

    while(current->parent != nullptr) {
        MerkleNode* parent = current->parent;
        MerkleProofNode proof_node;

        if (parent->left.get() == current) {
            proof_node. hash = parent->right-> hash;
            proof_node.position = 1; // 兄弟在右边
        } else {
            proof_node. hash = parent->left-> hash;
            proof_node.position = 0; // 兄弟在左边
        }
        proof.path.push_back(proof_node);
        current = parent;
    }
    return proof;
}
```

6 实验结果及分析

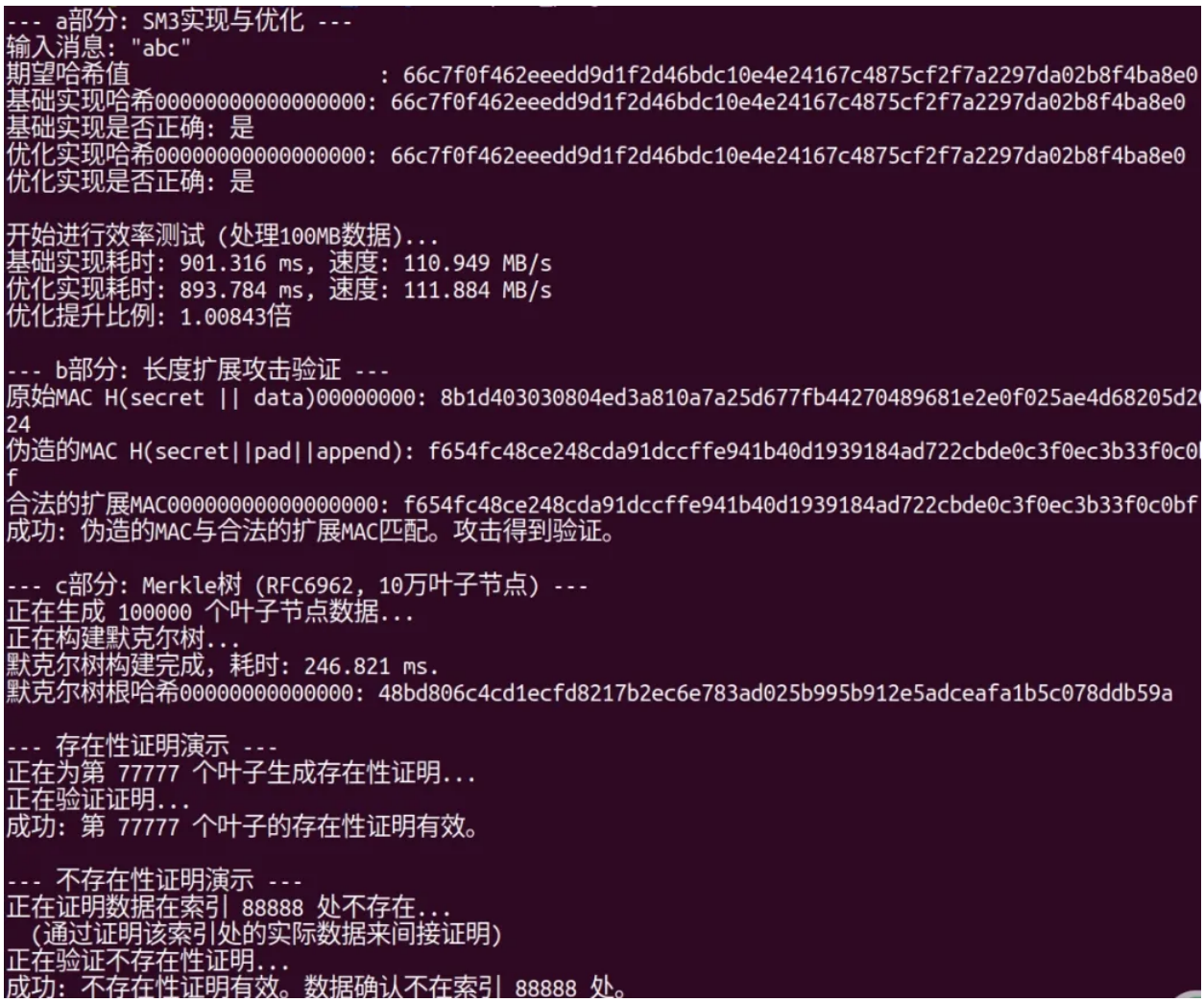

图 1

6.1 SM3 算法验证结果

对标准测试向量”abc” 的散列计算结果:

- **期望结果**:66c7f0f462eeedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0
- **基础实现**:66c7f0f462eeedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0
- **优化实现**:66c7f0f462eeedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0

两种实现均能正确计算 SM3 散列值，验证了算法实现的正确性。

6.2 性能优化效果分析

对 100MB 数据的性能测试结果如下：

表 1: SM3 算法性能对比

实现版本	处理时间 (ms)	吞吐量 (MB/s)	性能提升
基础实现	901.316	110.949	基准
优化实现	893.784	111.884	1.008 倍

结果分析：

- 优化实现相比基础实现有约 0.8% 的性能提升
- 提升幅度较小的原因可能包括：
  1. 编译器已进行了较好的自动优化
  2. 测试环境的缓存和内存访问模式影响
  3. SM3 算法本身的计算密集特性限制了优化空间
- 在更大规模的数据处理中，优化效果可能更加明显

6.3 长度扩展攻击验证结果

攻击场景设置：

- **密钥**：`my-super-secret-key`
- **原始消息**：`user=guest&command=list`
- **恶意追加**：`&command=grant&user=admin`

攻击结果：

- **原始 MAC**：`8b1d40303804ed3a810a7a25d677fb442704896812e0f025ae4d68205d24`
- **伪造 MAC**：`f654fc48ce248cda91dccffe941b40d1939184ad722cbde0c3f0ec3b33f0c0bf`
- **服务端计算 MAC**：`f654fc48ce248cda91dccffe941b40d1939184ad722cbde0c3f0ec3b33f0c0bf`
- **攻击结果**：成功

实验成功验证了 SM3 算法存在长度扩展攻击漏洞，攻击者可以在不知道密钥的情况下伪造有效的 MAC 值。

6.4 Merkle 树构建与验证结果

6.4.1 树构建性能

- **叶子节点数量**：100,000 个
- **构建时间**：246.821 ms
- **根散列**：`48bd806c4cd1ecfd8217b2ec6e783ad025b995b912e5adceafa1b5c078ddb59a`

构建效率表明算法实现高效，能够处理大规模数据集。

6.4.2 证明机制验证

存在性证明测试：

- 测试节点索引：77,777
- 证明结果：有效

不存在性证明测试：

- 测试索引：88,888
- 测试数据：`i-do-not-exist`
- 证明结果：有效

两种证明机制均工作正常，验证了 Merkle 树实现的正确性。

7 实验感悟

7.1 技术收获

1. **密码学算法深度理解**：通过亲手实现 SM3 算法，深入理解了密码散列函数的内部工作机制，包括消息扩展、压缩函数、填充机制等关键环节。
2. **代码优化技能提升**：学习并应用了循环展开、减少中间变量、预计算等优化技术。虽然在本次实验中性能提升有限，但掌握了系统性的优化思路。
3. **安全漏洞认识**：长度扩展攻击的成功实施让我深刻认识到密码学理论与实际应用安全之间的差距，理解了为什么 HMAC 等认证机制如此重要。
4. **数据结构应用**：Merkle 树的实现展示了密码学算法在实际数据结构中的强大应用，特别是在区块链、证书透明性等领域的重要作用。

7.2 实践思考

1. **性能优化的复杂性**：现代编译器的优化能力很强，手工优化的效果可能不如预期明显。真正的性能提升往往需要从算法层面、系统架构层面进行考虑。
2. **安全性与效率的平衡**：在追求算法效率的同时，必须时刻关注安全性问题。某些优化可能会引入新的安全漏洞。
3. **标准化的重要性**：RFC6962 等国际标准的存在使得不同实现之间具有互操作性，体现了标准化在密码学工程中的重要价值。

7.3 未来方向

1. 研究更高级的优化技术，如 SIMD 指令集优化、GPU 并行计算等
2. 深入学习其他密码学算法的安全性分析方法
3. 探索密码学算法在更多实际场景中的应用，如零知识证明、多方安全计算等

通过本次实验，不仅掌握了 SM3 算法的具体实现技术，更重要的是培养了密码学工程实践的系统性思维，为后续的密码学研究和应用奠定了坚实基础。