



网络空间安全创新创业实践 Project4

吴扬

2025 年 7 月 29 日

目录

1 实验环境	2
2 问题重述	2
3 实验原理	2
3.1 SM3 算法原理	2
3.2 长度扩展攻击原理	2
3.3 Merkle 树与存在性/不存在性证明	3
4 实现过程详细说明	3
4.1 SM3 算法的基础与优化实现	3
4.1.1 基础实现	3
4.1.2 优化实现	4
4.2 长度扩展攻击验证	7
4.3 Merkle 树构建与证明	8
5 实验结果展示	10
5.1 SM3 实现与优化	10
5.2 长度扩展攻击	10
5.3 Merkle 树与证明	11
6 实验结论与心得	11

1 实验环境

处理器	Intel(R) Core(TM) i9-14900HX 2.20 GHz
机带 RAM	16.0 GB (15.6 GB 可用)
Windows 版本	Windows 11 家庭中文版
操作系统版本	22631.4169

2 问题重述

SM3 的软件实现与优化。

1. 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。
2. 基于 SM3 的实现，验证 length-extension attack。
3. 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 实验原理

3.1 SM3 算法原理

SM3 是一种中国商用密码标准杂凑函数，输出长度为 256 位（32 字节），与 SHA-256 设计类似但结构与参数均不同。其主要流程如下：

- 消息填充：按照特定规则填充消息至长度为 512 位（64 字节）的整数倍。
- 消息分组：每 512 位为一组，输入压缩函数。
- 消息扩展：对每组消息生成 68 个字（W₀...W₆₇），并通过线性变换生成 W'。
- 压缩函数：迭代 64 轮，每轮引入布尔函数、置换函数与常数 T_j，最终得到输出。
- 输出：将最终的 8 个 32 位状态拼接成 256 位哈希输出。

3.2 长度扩展攻击原理

长度扩展攻击是对基于 Merkle–Damgård 结构的哈希算法的典型攻击方式。攻击者知道 $H(secret||message)$ 和 $message$ ，可通过已知的中间状态伪造 $H(secret||message||pad||append)$ ，即无需 $secret$ 即可构造合法 MAC。

3.3 Merkle 树与存在性/不存在性证明

Merkle 树是一种二叉哈希树，每个叶子节点存储数据的哈希值，父节点存储其左右孩子哈希的组合。通过 Merkle Tree，可以高效地证明某数据是否存在于树中（存在性证明），或者证明某数据不在某个特定位置（不存在性证明）。RFC6962 (Certificate Transparency) 采用了特殊的节点前缀 (0x00/0x01) 加 SM3 计算节点哈希。

4 实现过程详细说明

4.1 SM3 算法的基础与优化实现

4.1.1 基础实现

基础实现严格按照 SM3 标准流程，包括消息填充、扩展和 64 轮压缩。核心代码如下：

Listing 1: SM3 基础压缩函数主要流程

```
1 void SM3::compress_basic(const uint8_t block[SM3_BLOCK_SIZE]) {
2     uint32_t W[68], W_prime[64];
3     uint32_t A, B, C, D, E, F, G, H;
4     uint32_t SS1, SS2, TT1, TT2;
5
6     // 1. 消息扩展 (Message Expansion)
7     for (int j = 0; j < 16; j++) {
8         GET_UINT32_BE(W[j], block, j * 4);
9     }
10    for (int j = 16; j < 68; j++) {
11        W[j] = P1(W[j - 16] ^ W[j - 9] ^ ROTL(W[j - 3], 15)) ^ ROTL(
12            W[j - 13], 7) ^ W[j - 6];
13    }
14    for (int j = 0; j < 64; j++) {
15        W_prime[j] = W[j] ^ W[j + 4];
16    }
17
18    // 2. 迭代压缩 (Iterative Compression)
19    A = state[0]; B = state[1]; C = state[2]; D = state[3];
20    E = state[4]; F = state[5]; G = state[6]; H = state[7];
21
22    for (int j = 0; j < 64; j++) {
23        SS1 = ROTL(ROTL(A, 12) + E + ROTL((j < 16) ? T_00_15 :
24            T_16_63, j), 7);
```

```

23     SS2 = SS1 ^ ROTL(A, 12);

24

25     TT1 = (j < 16) ? FF_00_15(A, B, C) + D + SS2 + W_prime[j] :
26         FF_16_63(A, B, C) + D + SS2 + W_prime[j];
27     TT2 = (j < 16) ? GG_00_15(E, F, G) + H + SS1 + W[j] :
28         GG_16_63(E, F, G) + H + SS1 + W[j];

29     D = C;
30     C = ROTL(B, 9);
31     B = A;
32     A = TT1;
33     H = G;
34     G = ROTL(F, 19);
35     F = E;
36     E = PO(TT2);
37 }

38 // 3. 更新状态
39 state[0] ^= A; state[1] ^= B; state[2] ^= C; state[3] ^= D;
40 state[4] ^= E; state[5] ^= F; state[6] ^= G; state[7] ^= H;
41 }

```

4.1.2 优化实现

优化主要思路为：

- 全部预先展开消息扩展，减少循环和分支判断
- 主压缩循环采用部分循环展开，减少分支和变量赋值
- 常量 T_j 提前左移

代码片段如下：

Listing 2: SM3 优化压缩函数主要流程

```

42 void SM3::compress_optimized(const uint8_t block[SM3_BLOCK_SIZE]) {
43     uint32_t W[68], W_prime[64];
44     uint32_t A, B, C, D, E, F, G, H;
45     uint32_t T_j, SS1, SS2, TT1, TT2;
46
47 // 1. 消息扩展 - 处理前16个字

```

```
48     for (int j = 0; j < 16; j++) {
49         GET_UINT32_BE(W[j], block, j * 4);
50     }
51
52     // 批量计算消息扩展值
53     for (int j = 16; j < 68; j++) {
54         W[j] = P1(W[j - 16] ^ W[j - 9] ^ ROTL(W[j - 3], 15)) ^ ROTL(
55             W[j - 13], 7) ^ W[j - 6];
56     }
57
58     // 预计算W'值
59     for (int j = 0; j < 64; j++) {
60         W_prime[j] = W[j] ^ W[j + 4];
61     }
62
63     // 2. 初始化工作变量
64     A = state[0]; B = state[1]; C = state[2]; D = state[3];
65     E = state[4]; F = state[5]; G = state[6]; H = state[7];
66
67     // 3. 压缩函数主循环 - 分为前16轮和后48轮
68     // 前16轮
69     for (int j = 0; j < 16; j++) {
70         T_j = ROTL(T_00_15, j);
71         SS1 = ROTL(ROTL(A, 12) + E + T_j, 7);
72         SS2 = SS1 ^ ROTL(A, 12);
73
74         TT1 = FF_00_15(A, B, C) + D + SS2 + W_prime[j];
75         TT2 = GG_00_15(E, F, G) + H + SS1 + W[j];
76
77         D = C;
78         C = ROTL(B, 9);
79         B = A;
80         A = TT1;
81         H = G;
82         G = ROTL(F, 19);
83         F = E;
84         E = P0(TT2);
85     }
```

```
86 // 后48轮 - 使用4轮为一组的循环展开
87 for (int j = 16; j < 64; j += 4) {
88     // 第一轮
89     T_j = ROTL(T_16_63, j);
90     SS1 = ROTL(ROTL(A, 12) + E + T_j, 7);
91     SS2 = SS1 ^ ROTL(A, 12);
92     TT1 = FF_16_63(A, B, C) + D + SS2 + W_prime[j];
93     TT2 = GG_16_63(E, F, G) + H + SS1 + W[j];
94     D = C;
95     C = ROTL(B, 9);
96     B = A;
97     A = TT1;
98     H = G;
99     G = ROTL(F, 19);
100    F = E;
101    E = PO(TT2);

102
103    // 第二轮
104    T_j = ROTL(T_16_63, j+1);
105    SS1 = ROTL(ROTL(A, 12) + E + T_j, 7);
106    SS2 = SS1 ^ ROTL(A, 12);
107    TT1 = FF_16_63(A, B, C) + D + SS2 + W_prime[j+1];
108    TT2 = GG_16_63(E, F, G) + H + SS1 + W[j+1];
109    D = C;
110    C = ROTL(B, 9);
111    B = A;
112    A = TT1;
113    H = G;
114    G = ROTL(F, 19);
115    F = E;
116    E = PO(TT2);

117
118    // 第三轮
119    T_j = ROTL(T_16_63, j+2);
120    SS1 = ROTL(ROTL(A, 12) + E + T_j, 7);
121    SS2 = SS1 ^ ROTL(A, 12);
122    TT1 = FF_16_63(A, B, C) + D + SS2 + W_prime[j+2];
123    TT2 = GG_16_63(E, F, G) + H + SS1 + W[j+2];
124    D = C;
```

```

125     C = ROTP(B, 9);
126     B = A;
127     A = TT1;
128     H = G;
129     G = ROTP(F, 19);
130     F = E;
131     E = PO(TT2);

132
133 // 第四轮
134 T_j = ROTP(T_16_63, j+3);
135 SS1 = ROTP(ROTL(A, 12) + E + T_j, 7);
136 SS2 = SS1 ^ ROTP(A, 12);
137 TT1 = FF_16_63(A, B, C) + D + SS2 + W_prime[j+3];
138 TT2 = GG_16_63(E, F, G) + H + SS1 + W[j+3];
139 D = C;
140 C = ROTP(B, 9);
141 B = A;
142 A = TT1;
143 H = G;
144 G = ROTP(F, 19);
145 F = E;
146 E = PO(TT2);
147 }

148
149 // 4. 更新状态
150 state[0] ^= A; state[1] ^= B; state[2] ^= C; state[3] ^= D;
151 state[4] ^= E; state[5] ^= F; state[6] ^= G; state[7] ^= H;
152 }

```

注意：优化实现与基础实现的核心数学流程必须完全一致，否则结果不正确。

4.2 长度扩展攻击验证

长度扩展攻击的主要流程：

1. 攻击者已知 $H(secret||data)$ 和 $data$ ，猜测 $secret$ 长度
2. 根据 SM3 填充规则构造伪造消息长度
3. 以 $H(secret||data)$ 作为初始中间态，追加数据继续压缩
4. 得到 $H(secret||data||pad||append)$

代码片段如下：

Listing 3: SM3 长度扩展攻击代码核心

```
153 std::vector<uint8_t> SM3::length_extension_attack(
154     const std::vector<uint8_t>& original_hash,
155     uint64_t original_len,
156     const std::vector<uint8_t>& extra_data)
157 {
158     if (original_hash.size() != SM3_DIGEST_LENGTH) {
159         throw std::invalid_argument("Original hash must be 32 bytes.
160                                     ");
161     }
162
163     SM3 attacker_sm3;
164     uint32_t internal_state[8];
165     for(int i = 0; i < 8; ++i) {
166         GET_UINT32_BE(internal_state[i], original_hash.data(), i *
167                     4);
168     }
169
170     // 1. 用已知的哈希值作为内部状态来初始化
171     // 2. 伪造消息总长度。长度是原始消息长度加上其填充的长度
172     uint64_t padded_original_len = ((original_len + 8) /
173                                     SM3_BLOCK_SIZE + 1) * SM3_BLOCK_SIZE;
174     attacker_sm3.init_with_state(internal_state, padded_original_len
175                                   );
176
177     // 3. 用新数据更新
178     attacker_sm3.update(extra_data);
179
180     // 4. 计算出最终的伪造哈希
181     return attacker_sm3.final();
182 }
```

4.3 Merkle 树构建与证明

- 叶子节点哈希: $H(0x00||leaf_data)$
- 内部节点哈希: $H(0x01||left_hash||right_hash)$
- 构建过程: 按层自底向上, 两两组合构造父节点

- **存在性证明:** 记录从叶到根的兄弟节点 hash 与左右关系
- **不存在性证明:** 指定索引处的实际数据存在性证明即可

核心代码如下：

Listing 4: Merkle 树构建与证明主要流程

```
179 for each leaf in leaves:  
180     leaf.hash = hash_leaf(leaf.data)  
181 while current_level.size() > 1:  
182     for each pair (L, R):  
183         parent.hash = hash_internal_node(L.hash, R.hash)  
184         ...  
185 proof = []  
186 while current->parent:  
187     proof.append(sibling_hash and position)
```

5 实验结果展示

```
sherlock@ubuntu:~/study/sm3_project$ g++ -std=c++14 -o sm3_project main.cpp sm3.cpp merkle_tree.cpp -O2 -Wall
sherlock@ubuntu:~/study/sm3_project$ ./sm3_project
--- a部分: SM3实现与优化 ---
输入消息: "abc"
期望哈希值: 66c7f0f462eedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0
基础实现哈希0000000000000000: 66c7f0f462eedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0
基础实现是否正确: 是
优化实现哈希0000000000000000: 66c7f0f462eedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0
优化实现是否正确: 是

开始进行效率测试 (处理100MB数据)...
基础实现耗时: 901.316 ms, 速度: 110.949 MB/s
优化实现耗时: 893.784 ms, 速度: 111.884 MB/s
优化提升比例: 1.00843倍

--- b部分: 长度扩展攻击验证 ---
原始MAC H(secret || data)00000000: 8b1d403030804ed3a810a7a25d677fb44270489681e2e0f025ae4d68205d2024
伪造的MAC H(secret||pad||append): f654fc48ce248cda91dccffe941b40d1939184ad722cbde0c3f0ec3b33f0c0bf
合法的扩展MAC0000000000000000: f654fc48ce248cda91dccffe941b40d1939184ad722cbde0c3f0ec3b33f0c0bf
成功: 伪造的MAC与合法的扩展MAC匹配。攻击得到验证。

--- c部分: Merkle树 (RFC6962, 10万叶子节点) ---
正在生成 100000 个叶子节点数据...
正在构建默克尔树...
默克尔树构建完成, 耗时: 246.821 ms.
默克尔树根哈希0000000000000000: 48bd806c4cd1ecfd8217b2ec6e783ad025b995b912e5adceafa1b5c078ddb59a

--- 存在性证明演示 ---
正在为第 77777 个叶子生成存在性证明...
正在验证证明...
成功: 第 77777 个叶子的存在性证明有效。

--- 不存在性证明演示 ---
正在证明数据在索引 88888 处不存在...
(通过证明该索引处的实际数据来间接证明)
正在验证不存在性证明...
成功: 不存在性证明有效。数据确认不在索引 88888 处。
```

图 1: 实验主要结果运行终端截图

5.1 SM3 实现与优化

- 输入消息”abc”，基础和优化两版 SM3 哈希均正确输出
- 基础实现耗时: 901.316ms，速度 110.949MB/s
- 优化实现耗时: 893.784ms，速度 111.884MB/s
- 优化提升比例: 1.00843 倍

5.2 长度扩展攻击

- 原始 MAC 和伪造 MAC 完全一致，证明攻击成立

5.3 Merkle 树与证明

- 10 万叶子节点 Merkle 树构建耗时 246.821ms
- 存在性与不存在性证明均能正确验证

6 实验结论与心得

- SM3 的实现需严格按照标准流程，优化时应保证数学等价性。
- 软件优化主要通过减少分支、循环展开、内存读写优化等实现。
- Merkle 树结合 SM3 可高效完成大规模数据一致性与安全性证明。
- 实践过程中，理论与工程实现需相互印证，调试和单元测试必不可少。