



Project 4: SM3 的软件实现与优化

学院: 网络空间安全学院

专业: 密码科学与技术

姓名: 李双平

学号: 202200180026

2025 年 8 月 15 日



目录

1 SM3 基本实现	2
1.1 实现原理	2
1.2 代码解释	2
1.3 实验结果	5
2 SM3 优化	5
2.1 优化原理	5
2.2 代码解释	5
2.3 实验结果	11
3 SM3 长度扩展攻击	12
3.1 实现原理	12
3.2 代码解释	12
3.3 实验结果	17
4 SM3 构建 Merkle 子树	18
4.1 实现原理	18
4.2 代码解释	18
4.3 实验结果	24



1 SM3 基本实现

1.1 实现原理

SM3 哈希算法是我国商用密码标准 GM/T 0004-2012 中规定的密码杂凑算法，其输入为任意长度的消息，输出为长度为 256 位的杂凑值。SM3 在结构上与 SHA-256 类似，采用 Merkle-Damgård 结构，通过消息填充、消息扩展、迭代压缩三个阶段实现。其主要步骤如下：

1. **消息填充：**将原始消息补充至长度模 512 比特后等于 448，比特流末尾追加 64 比特的消息长度。填充规则为先追加一个 1 比特，再追加若干个 0 比特，最后追加原消息长度的 64 位大端表示。
2. **消息扩展：**将每个 512 比特分组划分为 16 个 32 位字 W_0, W_1, \dots, W_{15} ，并利用非线性函数 P_1 扩展生成 $W_{16} \dots W_{67}$ ，再计算 $W'_j = W_j \oplus W_{j+4}$ ，得到 64 个 W'_j 。
3. **迭代压缩：**初始向量 V_0 为标准定义的 256 位常量。对于每个分组，使用布尔函数 FF_j 和 GG_j 、置换函数 P_0 ，结合消息调度和轮常量 T_j ，进行 64 步迭代运算，得到新的向量 V_i 。迭代公式如下：

$$SS1 = \text{ROTL}((\text{ROTL}(A, 12) + E + \text{ROTL}(T_j, j)), 7)$$

$$SS2 = SS1 \oplus \text{ROTL}(A, 12)$$

$$TT1 = FF_j(A, B, C) + D + SS2 + W'_j$$

$$TT2 = GG_j(E, F, G) + H + SS1 + W_j$$

最终 $V_i = V_{i-1} \oplus ABCDEFGH$ 。

4. **输出结果：**将最后一轮迭代结果拼接为 256 位输出，得到消息的哈希值。

1.2 代码解释

1. 常量与基础函数定义

```

1 // 循环左移
2 static inline u32 ROTL(u32 x, unsigned n) {
3     return (x << n) | (x >> (32 - n));
4 }
5
6 // 置换函数
7 static inline u32 P0(u32 x) {
8     return x ^ ROTL(x, 9) ^ ROTL(x, 17);
9 }
10
11 static inline u32 P1(u32 x) {
12     return x ^ ROTL(x, 15) ^ ROTL(x, 23);
13 }
14
15 // 布尔函数

```



```

16 static inline u32 FF(u32 x, u32 y, u32 z, int j) {
17     if (j >= 0 && j <= 15) return x ^ y ^ z;
18     return (x & y) | (x & z) | (y & z);
19 }
20
21 static inline u32 GG(u32 x, u32 y, u32 z, int j) {
22     if (j >= 0 && j <= 15) return x ^ y ^ z;
23     return (x & y) | ((~x) & z);
24 }
```

这些函数实现了 SM3 算法中的基本运算：ROTL 为循环左移， P_0 与 P_1 为消息扩展和压缩中使用的线性置换， FF 与 GG 是布尔函数，在不同轮次 ($j \leq 15$ 或 $j > 15$) 时取值不同。

2. 消息填充与扩展

```

1 vector<u8> sm3(const vector<u8>& msg) {
2     // 初始向量
3     u32 V[8] = {0x7380166f, 0x4914b2b9, 0x172442d7, 0xda8a0600,
4                 0xa96f30bc, 0x163138aa, 0xe38dee4d, 0xb0fb0e4e};
5
6     // 填充
7     vector<u8> M = msg;
8     u64 bitlen = (u64)msg.size() * 8;
9     M.push_back(0x80);
10    while ((M.size() % 64) != 56) M.push_back(0x00);
11    for (int i = 7; i >= 0; --i) {
12        M.push_back((u8)((bitlen >> (i * 8)) & 0xFF));
13    }
}
```

这里先保存初始向量 V ，然后按 SM3 规则进行消息填充：追加一个 $0x80$ （二进制 10000000），再补 0，使长度模 64 余 56，最后附加 64 位原消息长度。

3. 消息扩展与轮函数计算

```

1 size_t nblocks = M.size() / 64;
2 for (size_t bi = 0; bi < nblocks; ++bi) {
3     u32 W[68], W1[64];
4
5     for (int i = 0; i < 16; ++i) {
6         size_t off = bi * 64 + i * 4;
7         W[i] = ((u32)M[off] << 24) | ((u32)M[off + 1] << 16) |
8                 ((u32)M[off + 2] << 8) | ((u32)M[off + 3]);
9     }
10    for (int j = 16; j <= 67; ++j) {
11        u32 x = W[j - 16] ^ W[j - 9] ^ ROTL(W[j - 3], 15);
```

```

12         W[j] = P1(x) ^ ROTL(W[j - 13], 7) ^ W[j - 6];
13     }
14     for (int j = 0; j <= 63; ++j) {
15         W1[j] = W[j] ^ W[j + 4];
16     }

```

此处先将每个 512 位分组拆分为 $W_0 \dots W_{15}$, 然后通过 P_1 扩展到 W_{67} , 再计算 $W'_j = W_j \oplus W_{j+4}$ 。

4. 压缩过程

```

1      u32 A = V[0], B = V[1], C = V[2], D = V[3];
2      u32 E = V[4], F = V[5], G = V[6], H = V[7];
3
4      for (int j = 0; j <= 63; ++j) {
5          u32 Tj = (j <= 15) ? 0x79cc4519u : 0x7a879d8au;
6          u32 SS1 = ROTL((ROTL(A, 12) + E + ROTL(Tj, j)), 7);
7          u32 SS2 = SS1 ^ ROTL(A, 12);
8          u32 TT1 = FF(A, B, C, j) + D + SS2 + W1[j];
9          u32 TT2 = GG(E, F, G, j) + H + SS1 + W[j];
10         D = C;
11         C = ROTL(B, 9);
12         B = A;
13         A = TT1;
14         H = G;
15         G = ROTL(F, 19);
16         F = E;
17         E = PO(TT2);
18     }

```

该循环完成 64 步压缩运算, T_j 为轮常量, 不同轮取值不同。每步更新 A 到 H 八个寄存器的值。

5. 更新向量并输出结果

```

1      V[0] ^= A; V[1] ^= B; V[2] ^= C; V[3] ^= D;
2      V[4] ^= E; V[5] ^= F; V[6] ^= G; V[7] ^= H;
3
4
5      vector<u8> digest(32);
6      for (int i = 0; i < 8; ++i) {
7          digest[i * 4 + 0] = (u8)((V[i] >> 24) & 0xFF);
8          digest[i * 4 + 1] = (u8)((V[i] >> 16) & 0xFF);
9          digest[i * 4 + 2] = (u8)((V[i] >> 8) & 0xFF);
10         digest[i * 4 + 3] = (u8)((V[i] >> 0) & 0xFF);

```

```

11     }
12     return digest;
13 }
```

迭代完成后，将 V 与本轮输出按位异或，得到新的链值，最终将 256 位结果转换为字节数组输出。

1.3 实验结果

实验结果如下：

```

SM3("abcdefg") = 08b7ee8f741fbfb63907fc0029ae3fd6403e6927b50ed9f04665b22eab81e9b7
SM3("") = 1ab21d8355cfa17f8e61194831e81a8f22bec8c728fefb747ed035eb5082aa2b

C:\Users\lenovo\source\repos\ConsoleApplication1\x64\Debug\ConsoleApplication1.exe (进程 7480)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...
```

图 1：实验结果一

2 SM3 优化

2.1 优化原理

- 消息级并行**: 一次性并行处理 4 份独立的单块消息（最多 64 字节），利用 AVX2 的 256-bit 寄存器承载 4 个 32-bit 通道（lane），在压缩函数的 64 步中同步推进，显著提升单位时间内能处理的消息条数。
- SIMD 向量化布尔/算术/循环移位**: 将 SM3 中频繁出现的 XOR、AND、OR、加法、以及 32-bit 旋转操作统一改写为 AVX2 intrinsic (`_mm256_*`)，在 4 个 lane 上并行执行。
- 常量预处理 & 减少分支开销**: 将 T_j 先在标量上做 $\text{ROTL}(T_j, j)$ 再广播成向量，避免在 SIMD 域内做不必要的控制分支和重复计算。
- 内存布局友好**: 输入 4 组块时，先逐字（word）按 lane 打包到向量寄存器，再进行消息扩展和压缩运算，减少跨 lane 的洗牌（shuffle）成本。
- 通用性回退**: 当输入不是“全部为单块消息”时，自动回退到标量实现，保证通用正确性与简洁性。

综合效果：在典型 x86-64/AVX2 平台上，对大量短消息（≤64 字节）的哈希吞吐通常可获得明显加速。

2.2 代码解释

- 标量循环左移（掩码 & 早返回）**



```

1 static inline u32 ROTL(u32 x, unsigned n) {
2     n &= 31u;
3     if (n == 0) return x;
4     return (x << n) | (x >> (32 - n));
5 }
```

相比基础版本增加了 `n &= 31u` 与 `n==0` 的早返回，避免未定义行为与冗余移位，便于在标量路径、常量预处理中复用。

2. AVX2 辅助装载/运算封装

```

1 static inline __m256i set4_u32(u32 a, u32 b, u32 c, u32 d) {
2     return _mm256_setr_epi32((int)a, (int)b, (int)c, (int)d, 0, 0, 0, 0);
3 }
4 static inline __m256i set1_u32(u32 x) { return set4_u32(x, x, x, x); }
5
6 static inline u32 extract_lane_u32(__m256i v, int lane) {
7     alignas(32) u32 tmp[8];
8     _mm256_storeu_si256((__m256i*)tmp, v);
9     return tmp[lane];
10 }
11
12 static inline __m256i xor4(__m256i a, __m256i b) { return
13     _mm256_xor_si256(a, b); }
14 static inline __m256i and4(__m256i a, __m256i b) { return
15     _mm256_and_si256(a, b); }
16 static inline __m256i or4 (__m256i a, __m256i b) { return _mm256_or_si256
17     (a, b); }
18 static inline __m256i not4(__m256i a) { return _mm256_xor_si256(a,
19     _mm256_set1_epi32(-1)); }
20 static inline __m256i add4(__m256i a, __m256i b) { return
21     _mm256_add_epi32(a, b); }
```

这些封装将 AVX2 intrinsic 抽象为“4-lane 32-bit 并行运算”，降低主流程可读性负担，同时避免重复样板代码。

3. 向量化 32-bit 循环左移 & P0/P1/FF/GG

```

1 static inline __m256i rol4(__m256i x, int n) {
2     int r = n & 31;
3     if (r == 0) return x;
4     return _mm256_or_si256(_mm256_slli_epi32(x, r), _mm256_srl_epi32(x,
5         32 - r)); }
```

```

6 static inline __m256i P0_4(__m256i x) { return xor4(x, xor4(rol4(x, 9),
7     rol4(x, 17))); }
8
9 static inline __m256i P1_4(__m256i x) { return xor4(x, xor4(rol4(x, 15),
10    rol4(x, 23))); }
11
12 }
13 static inline __m256i FF_4(__m256i x, __m256i y, __m256i z, int j) {
14     if (j <= 15) return xor4(x, xor4(y, z));
15     return or4(or4(and4(x, y), and4(x, z)), and4(y, z));
16 }
```

将 SM3 的关键置换与布尔函数搬到 SIMD 域中：一次计算 4 个 32-bit lane，完全贴合 SM3 的“按字运算”特性。

4. 装入 4 份消息首 16 字 (W[0..15])

```

1 static inline __m256i load_w4(const array<u32, 4>& w) {
2     return set4_u32(w[0], w[1], w[2], w[3]);
3 }
```

该帮助函数把 4 条消息对应的同一索引的 32-bit 字打包为一个 `__m256i`，后续扩展和压缩在向量域统一进行。

5. 四路单块并行压缩（核心）

```

1 array<array<u32, 8>, 4>
2 sm3_compress_4way_single_block(
3     const array<array<u8, 64>, 4>& blocks,
4     const array<array<u32, 8>, 4>& initialVs) {
5
6     __m256i Wv[68], W1v[64];
7
8     // W[0..15]: 4 路并行装载
9     for (int j = 0; j < 16; ++j) {
10         array<u32, 4> tmp{};
11         for (int lane = 0; lane < 4; ++lane) {
12             size_t off = j * 4;
13             tmp[lane] = ((u32)blocks[lane][off] << 24) |
14                 ((u32)blocks[lane][off+1] << 16) |
15                 ((u32)blocks[lane][off+2] << 8) |
16                 ((u32)blocks[lane][off+3]);
17     }
```

```

18     Wv[j] = load_w4(tmp);
19 }
20
21 // W[16..67] 扩展 (向量化 P1/ROTL)
22 for (int j = 16; j <= 67; ++j) {
23     __m256i x = xor4(xor4(Wv[j-16], Wv[j-9]), rol4(Wv[j-3], 15));
24     Wv[j] = xor4(xor4(P1_4(x), rol4(Wv[j-13], 7)), Wv[j-6]);
25 }
26 for (int j = 0; j <= 63; ++j) W1v[j] = xor4(Wv[j], Wv[j+4]);
27
28 // A..H 初始化为 4 路 IV
29 __m256i A = set4_u32(initialVs[0][0], initialVs[1][0], initialVs
30 [2][0], initialVs[3][0]);
31 __m256i B = set4_u32(initialVs[0][1], initialVs[1][1], initialVs
32 [2][1], initialVs[3][1]);
33 __m256i C = set4_u32(initialVs[0][2], initialVs[1][2], initialVs
34 [2][2], initialVs[3][2]);
35 __m256i D = set4_u32(initialVs[0][3], initialVs[1][3], initialVs
36 [2][3], initialVs[3][3]);
37 __m256i E = set4_u32(initialVs[0][4], initialVs[1][4], initialVs
38 [2][4], initialVs[3][4]);
39 __m256i F = set4_u32(initialVs[0][5], initialVs[1][5], initialVs
40 [2][5], initialVs[3][5]);
41 __m256i G = set4_u32(initialVs[0][6], initialVs[1][6], initialVs
42 [2][6], initialVs[3][6]);
43 __m256i H = set4_u32(initialVs[0][7], initialVs[1][7], initialVs
44 [2][7], initialVs[3][7]);
45
46 // 64 步向量化压缩
47 for (int j = 0; j <= 63; ++j) {
48     u32 Tj_scalar = (j <= 15) ? 0x79cc4519u : 0x7a879d8au;
49     u32 Tj_rot = ROTL(Tj_scalar, (unsigned)j);
50     __m256i Tjv = set1_u32(Tj_rot);
51
52     __m256i SS1 = rol4(add4(add4(rol4(A, 12), E), Tjv), 7);
53     __m256i SS2 = xor4(SS1, rol4(A, 12));
54     __m256i TT1 = add4(add4(add4(FF_4(A, B, C, j), D), SS2), W1v[j]);
55     __m256i TT2 = add4(add4(add4(GG_4(E, F, G, j), H), SS1), Wv[j]);
56
57     D = C; C = rol4(B, 9); B = A; A = TT1;
58     H = G; G = rol4(F, 19); F = E; E = P0_4(TT2);
59 }

```

```

52
53 // 写回各 lane 的 V ^= (A..H)
54 array<array<u32, 8>, 4> outVs;
55 for (int lane = 0; lane < 4; ++lane) {
56     u32 a = extract_lane_u32(A, lane), b = extract_lane_u32(B, lane);
57     u32 c = extract_lane_u32(C, lane), d = extract_lane_u32(D, lane);
58     u32 e = extract_lane_u32(E, lane), f = extract_lane_u32(F, lane);
59     u32 g = extract_lane_u32(G, lane), h = extract_lane_u32(H, lane);
60     for (int i = 0; i < 8; ++i) outVs[lane][i] = initialVs[lane][i];
61     outVs[lane][0] ^= a; outVs[lane][1] ^= b; outVs[lane][2] ^= c;
62     outVs[lane][3] ^= d;
63     outVs[lane][4] ^= e; outVs[lane][5] ^= f; outVs[lane][6] ^= g;
64     outVs[lane][7] ^= h;
65 }
66 return outVs;
67 }
```

这是优化的“压缩核”，完成 4 条单块消息的 **W/W'** 扩展与 A..H 64 步迭代。 T_j 在标量域预旋转为 T_{j_rot} 并广播到向量，以减少 SIMD 内重复工作。

6. 并行接口 sm3_4way

```

1 array<vector<u8>, 4> sm3_4way(const array<vector<u8>, 4>& msgs) {
2     // 1) 各自填充
3     array<vector<u8>, 4> Ms; array<size_t, 4> nblocks{};
4     for (int i = 0; i < 4; ++i) {
5         Ms[i] = msgs[i];
6         u64 bitlen = (u64)msgs[i].size() * 8;
7         Ms[i].push_back(0x80);
8         while ((Ms[i].size() % 64) != 56) Ms[i].push_back(0x00);
9         for (int b = 7; b >= 0; --b) Ms[i].push_back((u8)((bitlen >> (b *
10            8)) & 0xFF));
11        nblocks[i] = Ms[i].size() / 64;
12    }
13
14    // 2) 如果 4 条都是单块，走 AVX2 快路径
15    bool all_single_block = true;
16    for (int i = 0; i < 4; ++i) if (nblocks[i] != 1) { all_single_block =
17        false; break; }
18
19    array<vector<u8>, 4> digests;
20    if (all_single_block) {
21        array<array<u8, 64>, 4> blocks;
22        for (int i = 0; i < 4; ++i)
```

```

1   for (int j = 0; j < 64; ++j) blocks[i][j] = Ms[i][j];

2

3   array<array<u32, 8>, 4> Vs_init;
4   for (int lane = 0; lane < 4; ++lane)
5     Vs_init[lane] = { 0x7380166f, 0x4914b2b9, 0x172442d7, 0xda8a0600,
6                     0xa96f30bc, 0x163138aa, 0xe38dee4d, 0xb0fb0e4e };

7

8   auto outVs = sm3_compress_4way_single_block(blocks, Vs_init);

9

10  // 输出 32 字节摘要
11  for (int i = 0; i < 4; ++i) {
12    vector<u8> dg(32);
13    for (int k = 0; k < 8; ++k) {
14      dg[k*4+0] = (u8)((outVs[i][k] >> 24) & 0xFF);
15      dg[k*4+1] = (u8)((outVs[i][k] >> 16) & 0xFF);
16      dg[k*4+2] = (u8)((outVs[i][k] >> 8) & 0xFF);
17      dg[k*4+3] = (u8)((outVs[i][k] >> 0) & 0xFF);
18    }
19    digests[i] = std::move(dg);
20  }
21  return digests;
22}

23

24  // 3) 否则逐条回退标量实现，保证通用正确性
25  for (int i = 0; i < 4; ++i) digests[i] = sm3_scalar(msgs[i]);
26  return digests;
27}

```

关键点：172 每条消息按 SM3 规则独立填充；173 检查是否都是单块，是则进入 SIMD 快路径；174 否则回退标量版本，兼顾性能与易用性。

7. 主程序

```

1 int main() {
2   // 标量校验
3   string test = "abc";
4   vector<u8> data(test.begin(), test.end());
5   auto dg = sm3_scalar(data);
6   cout << "SM3(\"" << test << "\") = " << to_hex(dg) << "\n";
7
8   // 4-way 单块并行测试
9   array<vector<u8>, 4> msgs;
10  msgs[0] = vector<u8>({'a'});
11  msgs[1] = vector<u8>({'a', 'b', 'c'});

```

```

12     msgs[2] = vector<u8>({'t','e','s','t'});
13     msgs[3] = vector<u8>{};
14
15     auto dgs = sm3_4way(msgs);
16     cout << "4-way SM3 结果:\n";
17     for (int i = 0; i < 4; ++i) {
18         cout << "msg[" << i << "] = \""
19             << string(msgs[i].begin(), msgs[i].end())
20             << "\" -> " << to_hex(dgs[i]) << "\n";
21     }
22
23 // 与标量 abc 对比
24 auto dg_scalar_abc = sm3_scalar(vector<u8>{'a','b','c'});
25 cout << "\n与SM3(\"abc\")结果是否相等: "
26     << (to_hex(dgs[1]) == to_hex(dg_scalar_abc) ? "相等" : "不相等")
27     << "\n";
28 return 0;
29 }
```

通过对 "abc" 的 SIMD 与标量结果对比，进行基本正确性检查。

该优化将 SM3 的按字运算映射到 AVX2 的 4-lane 32-bit SIMD 上，实现“4 消息单块并行”。同时通过常量预旋转、向量化 P0/P1/FF/GG 与循环移位，大幅降低循环体中的标量开销。在满足输入为短消息的典型场景下获得较高吞吐；对一般场景提供自动回退，保持实现的稳健性与易用性。

2.3 实验结果

实验结果如下：

```

SM3("abc") = 66c7f0f462eedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0
4-way SM3 结果：
msg[0] = "a" -> 623476ac18f65a2909e43c7fec61b49c7e764a91a18ccb82f1917a29c86c5e88
msg[1] = "abc" -> 66c7f0f462eedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0
msg[2] = "test" -> 55e12e91650d2fec56ec74e1d3e4ddbfcce2ef3a65890c2a19ecf88a307e76a23
msg[3] = "" -> 1ab21d8355cf17f8e61194831e81a8f22bec8c728fefb747ed035eb5082aa2b

与SM3("abc")结果是否相等：相等

C:\Users\lenovo\source\repos\ConsoleApplication1\x64\Debug\ConsoleApplication1.exe (进程 36448)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...]
```

图 2: 实验结果二



3 SM3 长度扩展攻击

3.1 实现原理

SM3 长度扩展攻击的核心思想是：在不知道原消息 `secret` 的情况下，如果攻击者知道

- 原消息的长度；
- 哈希函数的内部状态（即原消息的摘要值）；

就可以利用哈希的迭代结构（Merkle–Damgård）从中间状态继续计算，从而伪造出

`hash(secret || original || padding || append)`

而不需要知道 `secret` 的具体内容。

其步骤如下：

1. 服务器原本计算 `hash(secret || original)`。
2. 攻击者拦截该摘要，推测 `secret` 长度。
3. 根据 SM3 填充规则构造填充 `padding`。
4. 使用原摘要作为新的 IV，从新消息 `append` 开始继续压缩计算。
5. 得到的结果与服务器重新计算的哈希值一致，从而实现伪造。

3.2 代码解释

1. 位运算与布尔函数

首先定义了 SM3 的基本移位与布尔运算：

```

1 static inline u32 ROTL(u32 x, unsigned n) {
2     return (x << n) | (x >> (32 - n));
3 }
4 static inline u32 P0(u32 x) { return x ^ ROL(x, 9) ^ ROL(x, 17); }
5 static inline u32 P1(u32 x) { return x ^ ROL(x, 15) ^ ROL(x, 23); }
6 static inline u32 FF(u32 x, u32 y, u32 z, int j) {
7     if (j >= 0 && j <= 15) return x ^ y ^ z;
8     return (x & y) | (x & z) | (y & z);
9 }
10 static inline u32 GG(u32 x, u32 y, u32 z, int j) {
11     if (j >= 0 && j <= 15) return x ^ y ^ z;
12     return (x & y) | ((~x) & z);
13 }
```

- `ROTL`: 循环左移，用于比特级旋转操作。
- `P0`、`P1`: SM3 中的置换函数，在压缩与消息扩展中分别使用。

- FF、GG: 布尔函数, 根据轮数 j 选择不同逻辑组合。

2. SM3 主哈希函数

```

1  vector<u8> sm3(const vector<u8>& msg) {
2      u32 V[8] = {
3          0x7380166f, 0x4914b2b9, 0x172442d7, 0xda8a0600,
4          0xa96f30bc, 0x163138aa, 0xe38dee4d, 0xb0fb0e4e
5      };
6
7      vector<u8> M = msg;
8      u64 bitlen = (u64)msg.size() * 8;
9      M.push_back(0x80);
10     while ((M.size() % 64) != 56) M.push_back(0x00);
11     for (int i = 7; i >= 0; --i) M.push_back((u8)((bitlen >> (i * 8)) & 0
12           xFF));
13
14     size_t nblocks = M.size() / 64;
15     for (size_t bi = 0; bi < nblocks; ++bi) {
16         u32 W[68], W1[64];
17         for (int i = 0; i < 16; ++i) {
18             size_t off = bi * 64 + i * 4;
19             W[i] = ((u32)M[off] << 24) | ((u32)M[off + 1] << 16) |
20                   ((u32)M[off + 2] << 8) | ((u32)M[off + 3]);
21         }
22         for (int j = 16; j <= 67; ++j) {
23             u32 x = W[j - 16] ^ W[j - 9] ^ ROTL(W[j - 3], 15);
24             W[j] = P1(x) ^ ROTL(W[j - 13], 7) ^ W[j - 6];
25         }
26         for (int j = 0; j <= 63; ++j) W1[j] = W[j] ^ W[j + 4];
27
28         u32 A = V[0], B = V[1], C = V[2], D = V[3];
29         u32 E = V[4], F = V[5], G = V[6], H = V[7];
30
31         for (int j = 0; j <= 63; ++j) {
32             u32 Tj = (j <= 15) ? 0x79cc4519u : 0x7a879d8au;
33             u32 SS1 = ROTL((ROTL(A, 12) + E + ROTL(Tj, j)), 7);
34             u32 SS2 = SS1 ^ ROTL(A, 12);
35             u32 TT1 = FF(A, B, C, j) + D + SS2 + W1[j];
36             u32 TT2 = GG(E, F, G, j) + H + SS1 + W[j];
37             D = C; C = ROTL(B, 9); B = A; A = TT1;
38             H = G; G = ROTL(F, 19); F = E; E = PO(TT2);
39         }
40     }
41 }
```



```

39
40     V[0] ^= A; V[1] ^= B; V[2] ^= C; V[3] ^= D;
41     V[4] ^= E; V[5] ^= F; V[6] ^= G; V[7] ^= H;
42 }
43
44 vector<u8> digest(32);
45 for (int i = 0; i < 8; ++i) {
46     digest[i * 4 + 0] = (u8)((V[i] >> 24) & 0xFF);
47     digest[i * 4 + 1] = (u8)((V[i] >> 16) & 0xFF);
48     digest[i * 4 + 2] = (u8)((V[i] >> 8) & 0xFF);
49     digest[i * 4 + 3] = (u8)((V[i] >> 0) & 0xFF);
50 }
51 return digest;
52 }
```

该函数完成 SM3 哈希的完整流程：

- (a) 初始化向量 V 为标准常量。
- (b) 对输入进行填充，使消息长度满足 512 位分组的要求。
- (c) 分组进行消息扩展与压缩运算。
- (d) 最终拼接 256 位摘要输出。

3. 单块压缩函数

```

1 void sm3_compress_block(u32 V[8], const u8 block[64]) {
2     u32 W[68], W1[64];
3     for (int i = 0; i < 16; ++i) {
4         size_t off = i * 4;
5         W[i] = ((u32)block[off] << 24) | ((u32)block[off + 1] << 16) |
6             ((u32)block[off + 2] << 8) | ((u32)block[off + 3]);
7     }
8     for (int j = 16; j <= 67; ++j) {
9         u32 x = W[j - 16] ^ W[j - 9] ^ ROTL(W[j - 3], 15);
10        W[j] = P1(x) ^ ROTL(W[j - 13], 7) ^ W[j - 6];
11    }
12    for (int j = 0; j <= 63; ++j) W1[j] = W[j] ^ W[j + 4];
13
14    u32 A = V[0], B = V[1], C = V[2], D = V[3];
15    u32 E = V[4], F = V[5], G = V[6], H = V[7];
16
17    for (int j = 0; j <= 63; ++j) {
18        u32 Tj = (j <= 15) ? 0x79cc4519u : 0x7a879d8au;
19        u32 SS1 = ROTL((ROTL(A, 12) + E + ROTL(Tj, j)), 7);
20        u32 SS2 = SS1 ^ ROTL(A, 12);
```

```

21     u32 TT1 = FF(A, B, C, j) + D + SS2 + W1[j];
22     u32 TT2 = GG(E, F, G, j) + H + SS1 + W[j];
23     D = C; C = ROTL(B, 9); B = A; A = TT1;
24     H = G; G = ROTL(F, 19); F = E; E = PO(TT2);
25   }
26
27   V[0] ^= A; V[1] ^= B; V[2] ^= C; V[3] ^= D;
28   V[4] ^= E; V[5] ^= F; V[6] ^= G; V[7] ^= H;
29 }
```

该函数以当前 V 向量和 512 位消息块为输入：

- 先生成 W 与 $W1$ 两个消息调度数组。
- 再执行 64 轮迭代压缩，更新临时变量 $A \sim H$ 。
- 最后将结果异或回 V ，作为下一块的初始值。

4. SM3 填充函数

```

1 vector<u8> sm3_padding(u64 message) {
2   vector<u8> pad;
3   pad.push_back(0x80);
4   while (((message + pad.size()) % 64) != 56) pad.push_back(0x00);
5   u64 bitlen = message * 8;
6   for (int i = 7; i >= 0; --i) pad.push_back((u8)((bitlen >> (i * 8)) &
7     0xFF));
8 }
```

根据 SM3 规范，构造出满足长度要求的填充数据。

5. 从摘要恢复 IV

```

1 array<u32, 8> iv_digest(const vector<u8>& digest) {
2   array<u32, 8> iv;
3   for (int i = 0; i < 8; ++i) {
4     iv[i] = ((u32)digest[i * 4] << 24) | ((u32)digest[i * 4 + 1] <<
5       16) |
6       ((u32)digest[i * 4 + 2] << 8) | ((u32)digest[i * 4 + 3]);
7   }
8 }
```

SM3 是基于 Merkle–Damgård 结构的，因此摘要本身就是下一轮的初始向量 IV 。

6. 长度扩展攻击函数



```

1  vector<u8> forge_sm3_iv(const array<u32, 8>& IV_arr,
2                           size_t original_len_bytes,
3                           const vector<u8>& appended) {
4
5     vector<u8> glue = sm3_padding((u64)original_len_bytes);
6     size_t glue_len = glue.size();
7     u64 total_len = (u64)original_len_bytes + glue_len + appended.size();
8
9
10    vector<u8> suffix = appended;
11    vector<u8> final_pad = sm3_padding(total_len);
12    suffix.insert(suffix.end(), final_pad.begin(), final_pad.end());
13
14
15    u32 V[8];
16    for (int i = 0; i < 8; ++i) V[i] = IV_arr[i];
17
18
19    // 输出伪造摘要
20    vector<u8> digest(32);
21    for (int i = 0; i < 8; ++i) {
22        digest[i*4+0] = (u8)((V[i] >> 24) & 0xFF);
23        ...
24    }
25
26    return digest;
}

```

- (a) 根据原消息长度生成“粘合填充”(glue padding)。
- (b) 拼接要追加的数据并进行必要的最终填充。
- (c) 以原摘要恢复的 *IV* 作为压缩起点，直接对追加数据进行压缩。
- (d) 得到伪造的最终摘要。

7. 主函数

```

1  int main() {
2
3      string secret = "topsecret_";
4      string original_known = "userid=1001";
5
6      vector<u8> real_msg;
7      real_msg.insert(real_msg.end(), secret.begin(), secret.end());
8      real_msg.insert(real_msg.end(), original_known.begin(), original_known
9                      .end());
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
287
288
289
289
290
291
292
293
294
295
296
297
297
298
299
299
300
301
302
303
304
305
305
306
307
307
308
309
309
310
311
311
312
312
313
313
314
314
315
315
316
316
317
317
318
318
319
319
320
320
321
321
322
322
323
323
324
324
325
325
326
326
327
327
328
328
329
329
330
330
331
331
332
332
333
333
334
334
335
335
336
336
337
337
338
338
339
339
340
340
341
341
342
342
343
343
344
344
345
345
346
346
347
347
348
348
349
349
350
350
351
351
352
352
353
353
354
354
355
355
356
356
357
357
358
358
359
359
360
360
361
361
362
362
363
363
364
364
365
365
366
366
367
367
368
368
369
369
370
370
371
371
372
372
373
373
374
374
375
375
376
376
377
377
378
378
379
379
380
380
381
381
382
382
383
383
384
384
385
385
386
386
387
387
388
388
389
389
390
390
391
391
392
392
393
393
394
394
395
395
396
396
397
397
398
398
399
399
400
400
401
401
402
402
403
403
404
404
405
405
406
406
407
407
408
408
409
409
410
410
411
411
412
412
413
413
414
414
415
415
416
416
417
417
418
418
419
419
420
420
421
421
422
422
423
423
424
424
425
425
426
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
441
441
442
442
443
443
444
444
445
445
446
446
447
447
448
448
449
449
450
450
451
451
452
452
453
453
454
454
455
455
456
456
457
457
458
458
459
459
460
460
461
461
462
462
463
463
464
464
465
465
466
466
467
467
468
468
469
469
470
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
```

```

8     vector<u8> server_digest = sm3(real_msg);
9     cout << "服务器哈希值: " << to_hex(server_digest) << "\n";
10
11    size_t attacker_known_original_len = secret.size() + original_known.
12      size();
13
14    string append_str = ";admin=true";
15    vector<u8> appended(append_str.begin(), append_str.end());
16
17    array<u32, 8> IV = iv_digest(server_digest);
18    vector<u8> forged_digest = forge_sm3_iv(IV,
19      attacker_known_original_len, appended);
20    cout << "伪造哈希值: " << to_hex(forged_digest) << "\n";
21
22    vector<u8> glue = sm3_padding(attacker_known_original_len);
23    vector<u8> forged_message;
24    forged_message.insert(forged_message.end(), real_msg.begin(), real_msg
25      .end());
26    forged_message.insert(forged_message.end(), glue.begin(), glue.end());
27    forged_message.insert(forged_message.end(), appended.begin(), appended
28      .end());
29
30    vector<u8> server_check = sm3(forged_message);
31    cout << "服务器校验哈希值: " << to_hex(server_check) << "\n";
32
33    if (server_check == forged_digest) cout << "长度扩展攻击成功。\\n";
34    else cout << "失败，不匹配。\\n";
35
36    return 0;
37
38  }

```

- 模拟服务器对原始消息进行 SM3 计算。
- 攻击者只知道摘要与已知部分消息，构造长度扩展攻击数据。
- 生成伪造消息与对应摘要，实现哈希碰撞式伪造。

3.3 实验结果

实验结果如下：



```

服务器哈希值： b162118a1d3856b55ce095ef7c3ac632808a165c8fe0f6322e5677c47e80e002
伪造哈希值： 1fe00f07ad7c2d3b442457f1233d9d6815f3eb6b63f6e5d8cd1df31cc9204cd
服务器校验哈希值： 1fe00f07ad7c2d3b442457f1233d9d6815f3eb6b63f6e5d8cd1df31cc9204cd
长度扩展攻击成功。

C:\Users\lenovo\source\repos\ConsoleApplication1\x64\Debug\ConsoleApplication1.exe (进程 30120)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...|
```

图 3: 实验结果三

4 SM3 构建 Merkle 子树

4.1 实现原理

Merkle 树是一种二叉树结构，用于高效验证数据的完整性与一致性。通过将数据块分别计算哈希值作为叶子节点，再将相邻节点的哈希拼接并计算新哈希作为父节点，递归直到得到一个根哈希 (Root Hash)。在基于 RFC6962 的实现中：

- 叶子节点的哈希计算公式为： $LeafHash = SM3(0x00||Data)$ 。
- 非叶子节点的哈希计算公式为： $NodeHash = SM3(0x01||LeftHash||RightHash)$ 。
- 通过递归构建子树，并缓存中间结果，可以提高重复计算时的效率。

本项目使用 SM3 作为底层哈希函数，构建规模为 10^5 叶子节点的 Merkle 树，并实现两类证明：

1. 存在性证明 (Inclusion Proof)：证明某个数据块确实在树中；
2. 不存在性证明 (Non-inclusion Proof)：证明某个数据块不在树中。

证明过程基于根哈希与路径哈希值的验证。

4.2 代码解释

下面给出 SM3 构建 Merkle 子树的完整 C++ 代码及详细解释。

1. 位运算与布尔函数

```

1 static inline u32 R0TL(u32 x, unsigned n) {
2     return (x << n) | (x >> (32 - n));
3 }
4 static inline u32 P0(u32 x) {
5     return x ^ R0TL(x, 9) ^ R0TL(x, 17);
6 }
7 static inline u32 P1(u32 x) {
```



```

8     return x ^ ROTL(x, 15) ^ ROTL(x, 23);
9 }
10 static inline u32 FF(u32 x, u32 y, u32 z, int j) {
11     if (j <= 15) return x ^ y ^ z;
12     return (x & y) | (x & z) | (y & z);
13 }
14 static inline u32 GG(u32 x, u32 y, u32 z, int j) {
15     if (j <= 15) return x ^ y ^ z;
16     return (x & y) | ((~x) & z);
17 }
```

功能说明：

- ROTL：循环左移。
- P0, P1：SM3 定义的置换函数。
- FF, GG：SM3 压缩函数的布尔运算。

```

1 vector<u8> sm3(const vector<u8>& msg) {
2     u32 V[8] = { 0x7380166f, 0x4914b2b9, 0x172442d7, 0xda8a0600,
3                 0xa96f30bc, 0x163138aa, 0xe38dee4d, 0xb0fb0e4e };
4     vector<u8> M = msg;
5     u64 bitlen = (u64)msg.size() * 8;
6     M.push_back(0x80);
7     while ((M.size() % 64) != 56) M.push_back(0x00);
8     for (int i = 7; i >= 0; --i) {
9         M.push_back((u8)((bitlen >> (i * 8)) & 0xFF));
10    }
11    size_t nblocks = M.size() / 64;
12    for (size_t bi = 0; bi < nblocks; ++bi) {
13        u32 W[68], W1[64];
14        for (int i = 0; i < 16; ++i) {
15            size_t off = bi * 64 + i * 4;
16            W[i] = ((u32)M[off] << 24) | ((u32)M[off + 1] << 16) |
17                  ((u32)M[off + 2] << 8) | ((u32)M[off + 3]);
18        }
19        for (int j = 16; j <= 67; ++j) {
20            u32 x = W[j - 16] ^ W[j - 9] ^ ROTL(W[j - 3], 15);
21            W[j] = P1(x) ^ ROTL(W[j - 13], 7) ^ W[j - 6];
22        }
23        for (int j = 0; j <= 63; ++j) {
24            W1[j] = W[j] ^ W[j + 4];
25        }
26        u32 A = V[0], B = V[1], C = V[2], D = V[3];
```

```

27     u32 E = V[4], F = V[5], G = V[6], H = V[7];
28     for (int j = 0; j <= 63; ++j) {
29         u32 Tj = (j <= 15) ? 0x79cc4519u : 0x7a879d8au;
30         u32 SS1 = ROTL((ROTL(A, 12) + E + ROTL(Tj, j)), 7);
31         u32 SS2 = SS1 ^ ROTL(A, 12);
32         u32 TT1 = FF(A, B, C, j) + D + SS2 + W1[j];
33         u32 TT2 = GG(E, F, G, j) + H + SS1 + W[j];
34         D = C;
35         C = ROTL(B, 9);
36         B = A;
37         A = TT1;
38         H = G;
39         G = ROTL(F, 19);
40         F = E;
41         E = P0(TT2);
42     }
43     V[0] ^= A; V[1] ^= B; V[2] ^= C; V[3] ^= D;
44     V[4] ^= E; V[5] ^= F; V[6] ^= G; V[7] ^= H;
45 }
46 vector<u8> digest(32);
47 for (int i = 0; i < 8; ++i) {
48     digest[i * 4 + 0] = (u8)((V[i] >> 24) & 0xFF);
49     digest[i * 4 + 1] = (u8)((V[i] >> 16) & 0xFF);
50     digest[i * 4 + 2] = (u8)((V[i] >> 8) & 0xFF);
51     digest[i * 4 + 3] = (u8)((V[i] >> 0) & 0xFF);
52 }
53 return digest;
54 }
```

功能说明：

- 消息填充（添加 0x80、补 0、追加长度）。
- 消息扩展生成 $W[0..67]$ 和 $W1[0..63]$ 。
- 主压缩循环 64 轮，更新寄存器 A-H。
- 输出 256 位哈希值。

2. Merkle 树构建与子树哈希计算

```

1 vector<u8> leaf_hash(const vector<u8>& data) {
2     vector<u8> prefix = {0x00};
3     vector<u8> input = prefix;
4     input.insert(input.end(), data.begin(), data.end());
5     return sm3(input);
6 }
```



```

7  vector<u8> node_hash(const vector<u8>& left, const vector<u8>& right) {
8      vector<u8> prefix = {0x01};
9      vector<u8> input = prefix;
10     input.insert(input.end(), left.begin(), left.end());
11     input.insert(input.end(), right.begin(), right.end());
12     return sm3(input);
13 }
14 size_t largest_pow2(size_t n) {
15     size_t p = 1;
16     while (p << 1 < n) p <<= 1;
17     return p;
18 }
```

功能说明：

- `leaf_hash`: 生成叶子节点哈希（前缀 0x00）。
- `node_hash`: 生成内部节点哈希（前缀 0x01）。
- `largest_pow2`: 返回小于 n 的最大 2 的幂，用于分割子树。

```

1  class MerkleTree {
2      vector<vector<u8>> leaves;
3      map<string, vector<u8>> cache;
4  public:
5      MerkleTree(const vector<vector<u8>>& leaves_in) : leaves(leaves_in) {}
6      string cache_key(size_t start, size_t size) {
7          return to_string(start) + ":" + to_string(size);
8      }
9      vector<u8> subtree_hash(size_t start, size_t size) {
10         string key = cache_key(start, size);
11         if (cache.count(key)) return cache[key];
12         vector<u8> result;
13         if (size == 1) {
14             result = leaf_hash(leaves[start]);
15         } else {
16             size_t k = largest_pow2(size);
17             vector<u8> left = subtree_hash(start, k);
18             vector<u8> right = subtree_hash(start + k, size - k);
19             result = node_hash(left, right);
20         }
21         cache[key] = result;
22         return result;
23     }
24     vector<u8> root_hash() {
```



```

25     if (leaves.empty()) return sm3({});  

26     return subtree_hash(0, leaves.size());  

27 }  

28 };

```

功能说明：

- 构造函数：初始化叶子节点数组。
 - `cache_key`：生成缓存索引（“起始位置: 大小”）。
 - `subtree_hash`：
 - (a) 检查缓存，若存在直接返回。
 - (b) 若子树只有一个节点，计算叶子哈希。
 - (c) 否则，根据最大 2 次幂分割成左右子树，递归计算并生成内部节点哈希。
 - (d) 将结果存入缓存并返回。
 - `root_hash`：返回整棵树的根哈希。
3. Merkle 证明生成与验证在 Merkle 树中，证明（Proof）用于验证某个叶子节点是否属于给定的 Merkle 根。证明由一系列相邻节点哈希和它们的方向（左/右）组成，通过递归哈希重构出根哈希。

```

1 struct ProofNode {  

2     vector<u8> hash;  

3     bool is_left;  

4 };

```

功能说明：

- `ProofNode`：存储证明路径中的一个节点，包括节点哈希和它在路径中的方向标记（`is_left` 为真表示该节点位于左侧）。

```

1 class MerkleTreeWithProof : public MerkleTree {  

2 public:  

3     using MerkleTree::MerkleTree;  

4  

5     void build_proof(size_t start, size_t size, size_t target,  

6                         vector<ProofNode>& proof) {  

7         if (size == 1) return;  

8         size_t k = largest_pow2(size);  

9         if (target < start + k) {  

10             vector<u8> right_hash = subtree_hash(start + k, size - k);  

11             proof.push_back({right_hash, false});  

12             build_proof(start, k, target, proof);  

13         } else {  

14             vector<u8> left_hash = subtree_hash(start, k);

```



```

15         proof.push_back({left_hash, true});
16         build_proof(start + k, size - k, target, proof);
17     }
18 }
19
20 vector<ProofNode> get_proof(size_t index) {
21     vector<ProofNode> proof;
22     if (index < leaves_count()) {
23         build_proof(0, leaves_count(), index, proof);
24     }
25     return proof;
26 }
27
28 size_t leaves_count() const {
29     return this->leaves.size();
30 }
31 };

```

功能说明：

- `build_proof`:
 - (a) 若当前子树大小为 1，递归结束。
 - (b) 根据最大 2 次幂分割子树，判断目标索引在左子树还是右子树。
 - (c) 如果在左子树，将右子树哈希加入证明，标记 `is_left=false`，并递归进入左子树。
 - (d) 如果在右子树，将左子树哈希加入证明，标记 `is_left=true`，并递归进入右子树。
- `get_proof`: 返回给定叶子索引的完整证明路径。
- `leaves_count`: 返回叶子节点数量。

```

1 bool verify_proof(const vector<u8>& leaf_data,
2                     const vector<ProofNode>& proof,
3                     const vector<u8>& expected_root) {
4     vector<u8> hash = leaf_hash(leaf_data);
5     for (auto& node : proof) {
6         if (node.is_left) {
7             hash = node_hash(node.hash, hash);
8         } else {
9             hash = node_hash(hash, node.hash);
10        }
11    }
12    return hash == expected_root;
13 }

```

功能说明：



- `verify_proof`:
 - (a) 计算叶子数据的哈希值。
 - (b) 遍历证明路径，根据方向依次组合哈希。
 - (c) 最终得到的哈希与期望根哈希进行比较，相等则证明成立。

4. 测试

```

1 int main() {
2     vector<vector<u8>> leaves_data = {
3         {'a'}, {'b'}, {'c'}, {'d'}, {'e'}
4     };
5     MerkleTreeWithProof tree(leaves_data);
6     vector<u8> root = tree.root_hash();
7
8     // 获取第 2 个叶子的证明
9     auto proof = tree.get_proof(2);
10
11    // 验证证明
12    bool ok = verify_proof(leaves_data[2], proof, root);
13    cout << "Proof verification: " << (ok ? "PASS" : "FAIL") << endl;
14    return 0;
15 }
```

功能说明：

- 构造一个包含 5 个叶子节点的 Merkle 树。
- 计算树根哈希。
- 获取第 2 个叶子的证明路径。
- 调用 `verify_proof` 验证该证明的有效性。

4.3 实验结果

实验结果如下：



```
根哈希 = 1138915f5e0418519271da1ec5967898fe42bfa3c6f6034126542155582c0353
索引 12345 的存在性证明包含 17 个兄弟哈希
兄弟 [0] = d317f36099ed2f9e88c327ef03ff95d9c2557c5b13035a6b895c6c131363d3a2
兄弟 [1] = 51992a4594481e7a4c0c7c8ccdc7801cca60cd852a68448e7b5aedd3f6f0761b
兄弟 [2] = a18754e45be267af0e816383f14093adbf93670f94a2d3411a47a95f5af5b98d
兄弟 [3] = ac7cc03639156441d048c3a2d66b4672aaae36be12e7994d6369385c1d95a366
验证存在性证明 -> 成功
值不存在；左邻索引 = 1, 右邻索引 = 2
左邻证明大小 = 17
左邻叶子索引 1 值 = leaf-1
右邻证明大小 = 17
右邻叶子索引 2 值 = leaf-2

C:\Users\lenovo\source\repos\ConsoleApplication1\x64\Debug\ConsoleApplication1.exe (进程 8112)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。 . .
```

图 4: 实验结果四