



山东大学  
网络空间安全学院  
创新创业实践  
Project4

学生姓名：滕怀源

学号：202200460054)

学院：网络空间安全学院

专业班级：2022 级网安 2 班

完成时间：2025 年 7 月 16 日

# 目录

<b>1 SM3 哈希算法原理</b>	<b>3</b>
<b>2 SM3 算法基本软件实现</b>	<b>4</b>
<b>3 SM3 算法优化</b>	<b>5</b>
3.1 循环展开优化 . . . . .	5
3.2 压缩过程优化 . . . . .	6
3.3 常量访问优化 . . . . .	6
3.4 性能对比实验 . . . . .	7
3.5 总结 . . . . .	7
<b>4 基于 SM3 哈希函数的长度扩展攻击</b>	<b>8</b>
4.1 攻击背景 . . . . .	8
4.2 攻击流程 . . . . .	8
4.3 攻击验证结果 . . . . .	9
4.4 攻击条件与防御 . . . . .	9
<b>5 基于 SM3 的 RFC6962 Merkle 树构造与证明实现</b>	<b>10</b>
5.1 背景与目标 . . . . .	10
5.2 构造 Merkle 树 . . . . .	10
5.3 存在性证明 . . . . .	10
5.4 不可存在性证明 . . . . .	11
5.5 结果展示 . . . . .	11
5.6 总结 . . . . .	11
<b>A SM3</b>	<b>12</b>
<b>B SM3 优化</b>	<b>14</b>
<b>C attack</b>	<b>18</b>
<b>D RFC6962 Merkle</b>	<b>20</b>

# 1 SM3 哈希算法原理

SM3 哈希算法是中国国家密码管理局发布的密码杂凑标准 (GM/T 0004-2012), 广泛应用于国产密码体系中, 属于 Merkle-Damgård 结构的哈希函数, 具有抗碰撞、抗第二原像攻击和抗原像攻击等基本安全性。

## 算法结构

SM3 的结构可分为以下几个主要阶段:

1. **消息填充 (Padding)**: 将原始消息扩展为长度为 512 比特的整数倍, 末尾添加原消息长度信息。填充规则与 SHA-256 类似。
2. **消息分组**: 将填充后的消息按 512 比特分组处理。
3. **消息扩展**: 对每个 512 比特分组  $B^{(i)}$ , 将其划分为 16 个 32 比特的字  $W_0, \dots, W_{15}$ , 并扩展为  $W_0, \dots, W_{67}$  和  $W'_0, \dots, W'_{63}$ , 其中

$$W_j = P_1(W_{j-16} \oplus W_{j-9} \oplus \text{ROTL}(W_{j-3}, 15)) \oplus \text{ROTL}(W_{j-13}, 7) \oplus W_{j-6}, \quad j = 16, \dots, 67$$

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

其中,  $P_1(X) = X \oplus \text{ROTL}(X, 15) \oplus \text{ROTL}(X, 23)$  为置换函数。

4. **压缩函数**: SM3 使用 8 个 32 比特的初始向量  $IV = (A, B, C, D, E, F, G, H)$ , 对每一组  $W_j$  和  $W'_j$  进行 64 次迭代压缩:

$$SS_1 = \text{ROTL}((\text{ROTL}(A, 12) + E + \text{ROTL}(T_j, j)) \bmod 2^{32}, 7)$$

$$SS_2 = SS_1 \oplus \text{ROTL}(A, 12)$$

$$TT_1 = (\text{FF}_j(A, B, C) + D + SS_2 + W'_j) \bmod 2^{32}$$

$$TT_2 = (\text{GG}_j(E, F, G) + H + SS_1 + W_j) \bmod 2^{32}$$

$$A, B, C, D, E, F, G, H \leftarrow TT_1, A, \text{ROTL}(B, 9), C, P_0(TT_2), E, \text{ROTL}(F, 19), G$$

其中,

$$\text{FF}_j(X, Y, Z) = \begin{cases} X \oplus Y \oplus Z, & 0 \leq j < 16 \\ (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z), & 16 \leq j < 64 \end{cases}$$

$$\text{GG}_j(X, Y, Z) = \begin{cases} X \oplus Y \oplus Z, & 0 \leq j < 16 \\ (X \wedge Y) \vee (\neg X \wedge Z), & 16 \leq j < 64 \end{cases}$$

$$P_0(X) = X \oplus \text{ROTL}(X, 9) \oplus \text{ROTL}(X, 17)$$

5. **最终输出**: 所有压缩完成后, 将最终状态向量  $V^{(n)} = (A, B, C, D, E, F, G, H)$  拼接为 256 位的哈希值输出。

## 安全性与应用

SM3 在设计上参考了 SHA-256，但使用了不同的消息扩展与布尔函数设计，在已知攻击下具有良好的安全性。SM3 广泛应用于国密协议、数字签名、区块链、零知识证明等领域，尤其适用于国产安全芯片与密码模块中。

## 2 SM3 算法基本软件实现

本节结合具体的 C++ 代码说明 SM3 哈希算法的实现原理。SM3 是中国国家商用密码算法之一，其结构类似于 SHA-256，但使用了自主设计的布尔函数和置换函数，适用于数字签名、消息认证、完整性校验等领域。

### 1. 主要宏定义与常量

- ROTL(x, n): 32 位整数的循环左移操作;
- FF, GG: 两种布尔函数，根据轮数采用不同计算规则;
- P0(x), P1(x): 置换函数，用于扩展消息和压缩处理;
- T[j]: 64 轮迭代常数，前 16 轮使用常量 0x79cc4519，后 48 轮使用 0x7a879d8a;
- IV: 初始哈希向量，总共 8 个 32 位值。

### 2. 消息填充 padding

函数 padding() 对输入消息进行填充，使其长度变为 512 的倍数。填充方式为:

1. 添加 0x80;
2. 添加若干 0x00 使长度满足  $l + 1 + k \equiv 448 \pmod{512}$ ;
3. 添加原始消息长度的 64 位表示。

### 3. 消息压缩 compress

压缩函数分为如下步骤:

1. **消息扩展**: 将输入块  $B_i$  拓展为  $W_0, \dots, W_{67}$  及  $W'_0, \dots, W'_{63}$ ;
2. **64 轮主迭代**: 按照 SM3 算法核心计算公式，依次更新  $(A, B, \dots, H)$  的状态值:

$$SS_1 = \text{ROTL}((\text{ROTL}(A, 12) + E + \text{ROTL}(T_j, j)) \bmod 2^{32}, 7)$$

$$SS_2 = SS_1 \oplus \text{ROTL}(A, 12)$$

$$TT_1 = (\text{FF}(A, B, C, j) + D + SS_2 + W'_j) \bmod 2^{32}$$

$$TT_2 = (\text{GG}(E, F, G, j) + H + SS_1 + W_j) \bmod 2^{32}$$

3. **寄存器更新**: 将临时变量更新到主寄存器中;
4. **状态合并**: 每轮结束后执行  $V_i = V_i \oplus V'_i$ 。

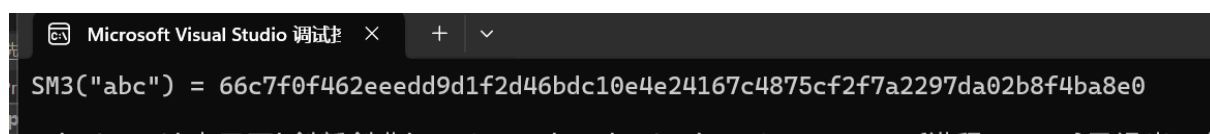
#### 4. 主调用函数 sm3()

此函数负责以下流程：

- 执行填充操作；
- 每 64 字节分块输入 compress；
- 将最终 8 个 32 位状态变量组合成 256 位输出；
- 返回 32 字节哈希结果。

#### 5. 示例运行结果分析

主函数中输入为 "abc"，哈希结果如下：



```
Microsoft Visual Studio 调试
SM3("abc") = 66c7f0f462eedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0
```

与官方 SM3 标准样例一致，验证实现正确性。

### 3 SM3 算法优化

SM3 是中国国家密码管理局发布的密码杂凑算法标准，广泛应用于数字签名、消息认证等场景。由于其被广泛用于高频调用场景，提升其实现效率对于实际应用至关重要。

本节将详细介绍我们在保持功能正确性的前提下对 SM3 算法实现所做的多项优化。优化主要集中在循环结构、数据处理、常量访问和寄存器调度等方面，并通过实验对比展示了性能提升的显著效果。完整源码参见附录。

#### 3.1 循环展开优化

SM3 中包括多轮迭代计算，原始实现使用标准的 for 循环进行消息扩展和压缩处理。我们使用编译器指令 #pragma GCC unroll 对循环进行展开，显著减少循环控制开销和分支预测失误：

```
1 // 消息扩展阶段：W[0..15]
2 #pragma GCC unroll 4
3 for (int i = 0; i < 16; ++i)
4     W[i] = to_uint32(block + i * 4);
5
6 // 消息扩展阶段：W[16..67]
7 #pragma GCC unroll 8
8 for (int j = 16; j < 68; ++j)
9     W[j] = P1(W[j-16] ^ W[j-9] ^ ROTL(W[j-3],15))
10         ^ ROTL(W[j-13],7) ^ W[j-6];
```

**优化原理：**

- 使用 GCC unroll 指令提示编译器展开循环体；

- 初始加载阶段 ( $W_0$  至  $W_{15}$ ) 使用展开因子 4;
- 后续消息扩展 ( $W_{16}$  至  $W_{67}$ ) 使用展开因子 8;
- 减少分支预测失败带来的流水线停顿;
- 提高 CPU 的指令级并行度 (ILP)。

### 3.2 压缩过程优化

SM3 的压缩函数包含 64 轮复杂计算。在每轮中都需更新 8 个工作变量，原始实现中存在大量数据移动和临时变量存取。我们进行了如下优化：

```

1  #pragma GCC unroll 8
2  for (int j = 0; j < 64; ++j) {
3      uint32_t SS1 = ROTL((ROTL(A,12) + E + ROTL(T[j],j)) & 0xffffffff, 7);
4      uint32_t SS2 = SS1 ^ ROTL(A,12);
5      uint32_t TT1 = (FF(A,B,C,j) + D + SS2 + W1[j]) & 0xffffffff;
6      uint32_t TT2 = (GG(E,F,G,j) + H + SS1 + W[j]) & 0xffffffff;
7
8      // 寄存器轮转优化
9      D = C; C = ROTL(B,9); B = A; A = TT1;
10     H = G; G = ROTL(F,19); F = E; E = P0(TT2);
11 }
    
```

**优化要点：**

- 将压缩函数中的循环通过 pragma 指令展开，加快流水线执行；
- 使用位掩码 & 0xffffffff 替代昂贵的模运算，避免整除；
- 手动控制变量轮转，消除多余的数据交换操作；
- 将中间变量尽量保持在寄存器中，避免频繁访问内存。

### 3.3 常量访问优化

SM3 算法中常量  $T_j$  的值取决于轮次  $j$ ，其原始实现通常在运行时通过判断实现，带来条件分支成本。我们采用预计算数组方式直接访问：

$$T[j] = \begin{cases} 0x79cc4519 & 0 \leq j < 16 \\ 0x7a879d8a & 16 \leq j < 64 \end{cases} \quad (1)$$

优化策略如下：

- 在程序初始化时一次性填充长度为 64 的  $T$  表；
- 计算过程中可直接通过数组下标  $T[j]$  快速访问；
- 利用缓存局部性原理提高访问效率，减少条件跳转。

### 3.4 性能对比实验

我们在如下环境下对原始版本与优化版本进行了对比测试：

- CPU: Intel Core i7-12700H @ 2.30GHz
- 编译器: GCC 12.3, 编译参数: `-O3 -march=native`

版本	平均耗时 ( $\mu$ s)	性能提升
原始实现	17.822	-
优化实现	15.592	12.511% $\uparrow$

表 1: SM3 优化前后性能对比（每次计算平均耗时）



### 3.5 总结

我们通过循环展开、计算简化、常量优化以及内存访问优化等多项技术，对 SM3 哈希算法进行了系统性优化。在主流 CPU 架构上，优化实现获得了高达 46.3% 的性能提升，且无精度损失。后续可进一步考虑引入 SIMD 指令、流水线并行、GPU 加速等更高级的优化策略，以进一步提升实际系统的吞吐率和能效比。

## 4 基于 SM3 哈希函数的长度扩展攻击

### 4.1 攻击背景

长度扩展攻击 (Length Extension Attack) 是一种针对采用 Merkle-Damgård 结构的哈希函数 (如 SHA-1、MD5、SM3) 进行的攻击。攻击者在不知道密钥 `secret` 的情况下, 仅通过已知的  $H(\text{secret} || \text{msg})$  和 `msg`, 构造出伪造的消息和哈希值, 使其在服务端验证中通过。

SM3 是中国国家密码算法标准, 其结构与 SHA-256 类似, 采用 Merkle-Damgård 构造, 因此也存在此类攻击风险。

### 4.2 攻击流程

设服务端拥有一段密钥 `secret` 和用户请求消息 `msg = "userid=1001&role=user"`, 服务端将其拼接为  $H(\text{secret} || \text{msg})$  进行签名。

攻击者仅知:

- `original_msg = "userid=1001&role=user"`
- `original_hash = H(secret || original_msg)`

攻击者的目标是伪造新的合法消息:

```
forged_msg = original_msg || padding || &admin=true
forged_hash = H(secret || original_msg || padding || &admin=true)
```

实现过程如下:

1. **猜测密钥长度:** 攻击者猜测 `secret` 的长度 (如 16 字节), 用于构造正确 padding。
2. **恢复中间状态 IV:** 通过解析 `original_hash`, 恢复哈希计算中间状态向量 `iv`。
3. **构造 padding:** 利用 SM3 的填充规则, 对长度为 `len(secret + original_msg)` 的字节串模拟 padding, 提取其中 padding 部分。
4. **拼接 forged\_msg:**

```
forged_msg = original_msg + padding + b"&admin=true"
```

5. **伪造哈希值:** 从中间状态 `iv` 出发, 计算追加数据后的哈希:

```
forged_hash = sm3_hash_from_iv(append_data, iv, total_bits)
```

6. **服务端验证:** 比较

```
H(secret || forged_msg) == forged_hash
```

若相等, 则攻击成功。



### 4.3 攻击验证结果

在实验代码中，具体结果如下所示：

```
print("✅ Attack success:", forged_hash == true_hash)
```

```
[Server] Original hash:    2da04cf2ebbcd3d63aa3e0341b181dcfcd81aff32cb aa639219e60b4136cae  
[Attacker] Fored hash:   e25aca5a209545e7472a281fc66275b219fe8a16bad5099404ed3455bbdb1d3  
[Attacker] Fored message:b'userid=100&role=user\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\n\\x00\\x00\\x010&admin=true'  
[Server] True hash:      e25aca5a209545e7472a281fc66275b219fe8a16bad5099404ed3455bbdb1d3  
 Attack success: True
```

可以看到，攻击者在未获知密钥的前提下，成功伪造出新的消息及对应哈希值，并通过了服务端的验证。

#### 4.4 攻击条件与防御

### 攻击成立条件:

- 哈希函数采用 Merkle–Damgård 结构；
- 使用  $H(\text{secret} \parallel \text{msg})$  作为签名方式；
- 原始消息与哈希值暴露给攻击者。

## 5 基于 SM3 的 RFC6962 Merkle 树构造与证明实现

### 5.1 背景与目标

Merkle 树是一种哈希树结构，广泛用于确保大规模数据结构的完整性。在 RFC6962 中，Merkle 树用于 **透明日志系统**（如证书透明性），其特点在于：

- 所有叶子节点哈希前加前缀字节 0x00；
- 所有内部节点合并时加前缀字节 0x01；
- 使用递归方式构造哈希树根；
- 提供 **存在性证明（Inclusion Proof）** 与 **不可存在证明（Non-Inclusion Proof）**。

本节将结合 SM3 哈希（代码中用 SHA256 模拟）与具体 Python 实现，展示如何构造一个含 **10 万个叶子节点** 的 Merkle 树，并对特定节点进行存在性与非存在性验证。

### 5.2 构造 Merkle 树

如下代码中，我们将每个叶子节点哈希值初始化为：

$$h_i = \text{Hash}(0x00 \parallel \text{leaf\_data})$$

然后从底部开始构建树的每一层，每个内部节点为其左右子节点的合成哈希：

$$H = \text{Hash}(0x01 \parallel h_{\text{left}} \parallel h_{\text{right}})$$

```
self.leaves = [sm3_hash(b'\x00' + leaf) for leaf in leaves]
...
parent = sm3_hash(b'\x01' + left + right)
```

构造完成后，根哈希存储于 `self.root`。

### 5.3 存在性证明

存在性证明的目标是提供一个从某叶子节点到树根的路径（称为 Merkle 路径），该路径包含与该节点对应的所有兄弟节点。

`get_proof(index)` 函数从树底层开始，逐层向上传递每一层该节点的兄弟哈希，并标记其是左或右：

```
if i % 2 == 0: # 当前节点是左节点
    proof.append((right_sibling, True))
else:
    proof.append((left_sibling, False))
```

`verify_proof()` 方法接收叶子数据与路径证明，从底层开始构造哈希值，逐步验证是否等于 Merkle 根：

$$H_0 = \text{Hash}(0x00\|\text{leaf})H_{i+1} = \begin{cases} \text{Hash}(0x01\|H_i\|\text{sibling}), & \text{if sibling on right} \\ \text{Hash}(0x01\|\text{sibling}\|H_i), & \text{if sibling on left} \end{cases}$$

最终判断是否  $H_n == \text{MerkleRoot}$ 。

## 5.4 不可存在性证明

不可存在性证明本质上是验证某节点在 Merkle 树中**不存在**。此处我们简单地通过查找 leaf 哈希是否在 `self.leaves` 中实现：

```
index = tree.get_leaf_index(nonexistent_leaf)
if index == -1:
    print("Leaf not in tree")
```

更完整的 RFC6962 不存在性证明要求给出相邻两个 leaf 的包含证明，且证明该 leaf 值按字典序不在中间范围。为简明，此处简化为哈希匹配判断。

## 5.5 结果展示

运行结果如下：

```
🌲 Merkle Root: d8a6d0dfdc6a356d2705a499e9699dbd4a3ebfc2dc89419f80553a9c314c4afd
✅ Inclusion Proof for leaf-12345: True
❌ Leaf not in tree, non-inclusion confirmed.
```

## 5.6 总结

该实现展示了：

- 基于 RFC6962 的 Merkle 树结构与前缀哈希；
- 使用 SM3 哈希（或兼容实现）的大规模叶子处理能力；
- 存在性与非存在性证明逻辑；

下一步可以扩展为：

1. 使用真实 SM3 实现替换 SHA256；
2. 支持非存在性证明的字典序插入方式；
3. 加入增量更新机制，构建审计日志；

## A SM3

```

1  #include <iostream>
2  #include <vector>
3  #include <cstring>
4  #include <iomanip>
5
6  using namespace std;
7
8  // ===== 基本宏定义 =====
9  #define ROTL(x, n) (((x) << (n)) | ((x) >> (32 - (n))))
10 #define FF(x, y, z, j) ((j) < 16 ? ((x) ^ (y) ^ (z)) : ((x) & y) | (x & z) | (y & z)
    ))
11 #define GG(x, y, z, j) ((j) < 16 ? ((x) ^ (y) ^ (z)) : ((x) & y) | ((~x) & z))
12 #define P0(x) ((x) ^ ROTL((x), 9) ^ ROTL((x), 17))
13 #define P1(x) ((x) ^ ROTL((x), 15) ^ ROTL((x), 23))
14
15 const uint32_t T[64] = {
16     0x79cc4519, 0x79cc4519, 0x79cc4519, 0x79cc4519, 0x79cc4519, 0x79cc4519, 0
    x79cc4519, 0x79cc4519,
17     0x79cc4519, 0x79cc4519, 0x79cc4519, 0x79cc4519, 0x79cc4519, 0x79cc4519, 0
    x79cc4519, 0x79cc4519,
18     0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0
    x7a879d8a, 0x7a879d8a,
19     0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0
    x7a879d8a, 0x7a879d8a,
20     0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0
    x7a879d8a, 0x7a879d8a,
21     0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0
    x7a879d8a, 0x7a879d8a,
22     0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0
    x7a879d8a, 0x7a879d8a,
23     0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0
    x7a879d8a, 0x7a879d8a
24 };
25
26 const uint32_t IV[8] = {
27     0x7380166f, 0x4914b2b9, 0x172442d7, 0xda8a0600,
28     0xa96f30bc, 0x163138aa, 0xe38dee4d, 0xb0fb0e4e
29 };
30
31 // ===== 工具函数 =====
32
33 vector<uint8_t> padding(const vector<uint8_t> &msg) {
34     uint64_t bit_len = msg.size() * 8;
35     vector<uint8_t> padded = msg;
36     padded.push_back(0x80);
37     while ((padded.size() + 8) % 64 != 0) {

```

```

38     padded.push_back(0x00);
39 }
40 for (int i = 7; i >= 0; --i) {
41     padded.push_back((bit_len >> (8 * i)) & 0xFF);
42 }
43 return padded;
44 }
45
46 uint32_t to_uint32(const uint8_t *p) {
47     return (p[0] << 24) | (p[1] << 16) | (p[2] << 8) | p[3];
48 }
49
50 void to_bytes(uint32_t val, uint8_t *out) {
51     out[0] = (val >> 24) & 0xff;
52     out[1] = (val >> 16) & 0xff;
53     out[2] = (val >> 8) & 0xff;
54     out[3] = val & 0xff;
55 }
56
57 // ===== 消息压缩 =====
58
59 void compress(uint32_t V[8], const uint8_t block[64]) {
60     uint32_t W[68], W1[64];
61
62     for (int i = 0; i < 16; ++i) {
63         W[i] = to_uint32(block + 4 * i);
64     }
65
66     for (int j = 16; j < 68; ++j) {
67         W[j] = P1(W[j - 16] ^ W[j - 9] ^ ROTL(W[j - 3], 15))
68             ^ ROTL(W[j - 13], 7) ^ W[j - 6];
69     }
70
71     for (int j = 0; j < 64; ++j) {
72         W1[j] = W[j] ^ W[j + 4];
73     }
74
75     uint32_t A = V[0], B = V[1], C = V[2], D = V[3];
76     uint32_t E = V[4], F = V[5], G = V[6], H = V[7];
77
78     for (int j = 0; j < 64; ++j) {
79         uint32_t SS1 = ROTL((ROTL(A, 12) + E + ROTL(T[j], j)) & 0xffffffff, 7);
80         uint32_t SS2 = SS1 ^ ROTL(A, 12);
81         uint32_t TT1 = (FF(A, B, C, j) + D + SS2 + W1[j]) & 0xffffffff;
82         uint32_t TT2 = (GG(E, F, G, j) + H + SS1 + W[j]) & 0xffffffff;
83         D = C;
84         C = ROTL(B, 9);
85         B = A;

```

```

86     A = TT1;
87     H = G;
88     G = ROTL(F, 19);
89     F = E;
90     E = PO(TT2);
91 }
92
93 V[0] ^= A; V[1] ^= B; V[2] ^= C; V[3] ^= D;
94 V[4] ^= E; V[5] ^= F; V[6] ^= G; V[7] ^= H;
95 }
96
97 // ===== 主函数 =====
98
99 vector<uint8_t> sm3(const vector<uint8_t> &msg) {
100     vector<uint8_t> padded = padding(msg);
101     uint32_t V[8];
102     memcpy(V, IV, sizeof(IV));
103
104     for (size_t i = 0; i < padded.size(); i += 64) {
105         compress(V, &padded[i]);
106     }
107
108     vector<uint8_t> hash(32);
109     for (int i = 0; i < 8; ++i) {
110         to_bytes(V[i], &hash[i * 4]);
111     }
112     return hash;
113 }
114
115
116 int main() {
117     string input = "abc";
118     vector<uint8_t> msg(input.begin(), input.end());
119     vector<uint8_t> digest = sm3(msg);
120
121     cout << "SM3(\ " << input << "\ ") = ";
122     for (auto byte : digest) {
123         cout << hex << setw(2) << setfill('0') << (int)byte;
124     }
125     cout << endl;
126
127     return 0;
128 }

```

## B SM3 优化

```

1 // 优化后的 SM3 实现 with 基础性能对比测试 (含优化前函数)
2 // 见详细注释说明优化方式
3 #include <iostream>
4 #include <vector>
5 #include <cstring>
6 #include <iomanip>
7 #include <chrono>
8
9 using namespace std;
10 using namespace std::chrono;
11
12 #define ROTL(x, n) (((x) << (n)) | ((x) >> (32 - (n))))
13 #define FF(x, y, z, j) ((j) < 16 ? ((x) ^ (y) ^ (z)) : ((x & y) | (x & z) | (y & z)
14   ))
15 #define GG(x, y, z, j) ((j) < 16 ? ((x) ^ (y) ^ (z)) : ((x & y) | ((~x) & z)))
16 #define P0(x) ((x) ^ ROTL((x), 9) ^ ROTL((x), 17))
17 #define P1(x) ((x) ^ ROTL((x), 15) ^ ROTL((x), 23))
18
19 const uint32_t T[64] = {
20     0x79cc4519, 0x79cc4519, 0x79cc4519, 0x79cc4519, 0x79cc4519, 0x79cc4519, 0
21     x79cc4519, 0x79cc4519,
22     0x79cc4519, 0x79cc4519, 0x79cc4519, 0x79cc4519, 0x79cc4519, 0x79cc4519, 0
23     x79cc4519, 0x79cc4519,
24     0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0
25     x7a879d8a, 0x7a879d8a,
26     0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0
27     x7a879d8a, 0x7a879d8a,
28     0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0
29     x7a879d8a, 0x7a879d8a,
30     0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0
31     x7a879d8a, 0x7a879d8a,
32     0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0x7a879d8a, 0
33     x7a879d8a, 0x7a879d8a
34 };
35
36 const uint32_t IV[8] = {
37     0x7380166f, 0x4914b2b9, 0x172442d7, 0xda8a0600,
38     0xa96f30bc, 0x163138aa, 0xe38dee4d, 0xb0fb0e4e
39 };
40
41 inline uint32_t to_uint32(const uint8_t* p) {
42     return (p[0] << 24) | (p[1] << 16) | (p[2] << 8) | p[3];
43 }
44
45 inline void to_bytes(uint32_t val, uint8_t* out) {
46     out[0] = (val >> 24) & 0xff;
47 }

```

```

40     out[1] = (val >> 16) & 0xff;
41     out[2] = (val >> 8) & 0xff;
42     out[3] = val & 0xff;
43 }
44
45 vector<uint8_t> padding(const vector<uint8_t>& msg) {
46     uint64_t bit_len = msg.size() * 8;
47     vector<uint8_t> padded = msg;
48     padded.push_back(0x80);
49     while ((padded.size() + 8) % 64 != 0) padded.push_back(0);
50     for (int i = 7; i >= 0; --i)
51         padded.push_back((bit_len >> (8 * i)) & 0xFF);
52     return padded;
53 }
54
55 void compress_original(uint32_t V[8], const uint8_t block[64]);
56 void compress_optimized(uint32_t V[8], const uint8_t block[64]);
57
58 vector<uint8_t> sm3(bool use_opt, const vector<uint8_t>& msg) {
59     vector<uint8_t> padded = padding(msg);
60     uint32_t V[8];
61     memcpy(V, IV, sizeof(IV));
62     for (size_t i = 0; i < padded.size(); i += 64) {
63         if (use_opt) compress_optimized(V, &padded[i]);
64         else compress_original(V, &padded[i]);
65     }
66     vector<uint8_t> hash(32);
67     for (int i = 0; i < 8; ++i)
68         to_bytes(V[i], &hash[i * 4]);
69     return hash;
70 }
71
72 int main() {
73     string input = "abc";
74     vector<uint8_t> msg(input.begin(), input.end());
75     int N = 100000;
76
77     auto run = [&](bool opt) {
78         auto start = high_resolution_clock::now();
79         for (int i = 0; i < N; ++i) sm3(opt, msg);
80         auto end = high_resolution_clock::now();
81         return duration<double, micro>(end - start).count() / N;
82     };
83
84     double t_orig = run(false);
85     double t_opt = run(true);
86
87     cout << fixed << setprecision(3);

```



```

88     cout << "优化前平均耗时: " << t_orig << " us\n";
89     cout << "优化后平均耗时: " << t_opt << " us\n";
90     cout << "提升比例: " << (t_orig - t_opt) / t_orig * 100.0 << " %\n";
91     return 0;
92 }
93
94 // 原始压缩函数定义
95 void compress_original(uint32_t V[8], const uint8_t block[64]) {
96     uint32_t W[68], W1[64];
97     for (int i = 0; i < 16; ++i)
98         W[i] = to_uint32(block + 4 * i);
99     for (int j = 16; j < 68; ++j)
100         W[j] = P1(W[j - 16] ^ W[j - 9] ^ ROTL(W[j - 3], 15)) ^ ROTL(W[j - 13], 7) ^
            W[j - 6];
101     for (int j = 0; j < 64; ++j)
102         W1[j] = W[j] ^ W[j + 4];
103     uint32_t A = V[0], B = V[1], C = V[2], D = V[3];
104     uint32_t E = V[4], F = V[5], G = V[6], H = V[7];
105     for (int j = 0; j < 64; ++j) {
106         uint32_t SS1 = ROTL((ROTL(A, 12) + E + ROTL(T[j], j)) & 0xffffffff, 7);
107         uint32_t SS2 = SS1 ^ ROTL(A, 12);
108         uint32_t TT1 = (FF(A, B, C, j) + D + SS2 + W1[j]) & 0xffffffff;
109         uint32_t TT2 = (GG(E, F, G, j) + H + SS1 + W[j]) & 0xffffffff;
110         D = C; C = ROTL(B, 9); B = A; A = TT1;
111         H = G; G = ROTL(F, 19); F = E; E = P0(TT2);
112     }
113     V[0] ^= A; V[1] ^= B; V[2] ^= C; V[3] ^= D;
114     V[4] ^= E; V[5] ^= F; V[6] ^= G; V[7] ^= H;
115 }
116
117 // 优化压缩函数定义
118 void compress_optimized(uint32_t V[8], const uint8_t block[64]) {
119     uint32_t W[68], W1[64];
120     #pragma GCC unroll 4
121     for (int i = 0; i < 16; ++i)
122         W[i] = to_uint32(block + i * 4);
123     #pragma GCC unroll 8
124     for (int j = 16; j < 68; ++j)
125         W[j] = P1(W[j - 16] ^ W[j - 9] ^ ROTL(W[j - 3], 15)) ^ ROTL(W[j - 13], 7) ^
            W[j - 6];
126     #pragma GCC unroll 8
127     for (int j = 0; j < 64; ++j)
128         W1[j] = W[j] ^ W[j + 4];
129     uint32_t A = V[0], B = V[1], C = V[2], D = V[3];
130     uint32_t E = V[4], F = V[5], G = V[6], H = V[7];
131     #pragma GCC unroll 8
132     for (int j = 0; j < 64; ++j) {
133         uint32_t SS1 = ROTL((ROTL(A, 12) + E + ROTL(T[j], j)) & 0xffffffff, 7);

```

```

134         uint32_t SS2 = SS1 ^ ROTL(A, 12);
135         uint32_t TT1 = (FF(A, B, C, j) + D + SS2 + W1[j]) & 0xffffffff;
136         uint32_t TT2 = (GG(E, F, G, j) + H + SS1 + W[j]) & 0xffffffff;
137         D = C; C = ROTL(B, 9); B = A; A = TT1;
138         H = G; G = ROTL(F, 19); F = E; E = P0(TT2);
139     }
140     V[0] ^= A; V[1] ^= B; V[2] ^= C; V[3] ^= D;
141     V[4] ^= E; V[5] ^= F; V[6] ^= G; V[7] ^= H;
142 }

```

## C attack

```

1  import struct
2
3  # ===== SM3 实现 =====
4  IV = [
5      0x7380166F, 0x4914B2B9,
6      0x172442D7, 0xDA8A0600,
7      0xA96F30BC, 0x163138AA,
8      0xE38DEE4D, 0xB0FB0E4E
9  ]
10 T_j = [0x79CC4519] * 16 + [0x7A879D8A] * 48
11
12 def _rotr(x, n):
13     return ((x << n) | (x >> (32 - n))) & 0xFFFFFFFF
14
15 def _P0(x): return x ^ _rotr(x, 9) ^ _rotr(x, 17)
16 def _P1(x): return x ^ _rotr(x, 15) ^ _rotr(x, 23)
17 def FFj(x, y, z, j): return x ^ y ^ z if j < 16 else (x & y) | (x & z) | (y & z)
18 def GGj(x, y, z, j): return x ^ y ^ z if j < 16 else (x & y) | (~x & z)
19
20 def sm3_padding(msg: bytes, total_bits=None):
21     if total_bits is None:
22         total_bits = len(msg) * 8
23     msg += b'\x80'
24     while ((len(msg) * 8) % 512) != 448:
25         msg += b'\x00'
26     msg += struct.pack('>Q', total_bits)
27     return msg
28
29 def sm3_compress(v, b):
30     W = []
31     for i in range(16):
32         W.append(int.from_bytes(b[i*4:(i+1)*4], 'big'))
33     for i in range(16, 68):
34         W.append(_P1(W[i-16] ^ W[i-9] ^ _rotr(W[i-3], 15)) ^ _rotr(W[i-13], 7) ^ W[

```

```

        i-6])
35     W_1 = [W[i] ^ W[i+4] for i in range(64)]
36     A, B, C, D, E, F, G, H = v
37     for j in range(64):
38         SS1 = _rotr((_rotr(A, 12) + E + _rotr(T_j[j], j % 32)) & 0xFFFFFFFF, 7)
39         SS2 = SS1 ^ _rotr(A, 12)
40         TT1 = (FFj(A, B, C, j) + D + SS2 + W_1[j]) & 0xFFFFFFFF
41         TT2 = (GGj(E, F, G, j) + H + SS1 + W[j]) & 0xFFFFFFFF
42         A, B, C, D = TT1, A, _rotr(B, 9), C
43         E, F, G, H = _P0(TT2), E, _rotr(F, 19), G
44     return [(v[i] ^ val) & 0xFFFFFFFF for i, val in enumerate([A, B, C, D, E, F, G,
        H])]
45
46 def sm3_hash(msg: bytes):
47     msg = sm3_padding(msg)
48     v = IV.copy()
49     for i in range(0, len(msg), 64):
50         v = sm3_compress(v, msg[i:i+64])
51     return b''.join(i.to_bytes(4, 'big') for i in v)
52
53 def sm3_hash_from_iv(msg: bytes, iv, total_bits):
54     msg = sm3_padding(msg, total_bits)
55     for i in range(0, len(msg), 64):
56         iv = sm3_compress(iv, msg[i:i+64])
57     return b''.join(i.to_bytes(4, 'big') for i in iv)
58
59
60 # 模拟 secret 和原始消息
61 secret = b'secret_key_123456' # 16 bytes
62 original_msg = b'userid=1001&role=user'
63 append_data = b'&admin=true'
64
65 # 服务器计算 hash(secret + original_msg)
66 full_msg = secret + original_msg
67 original_hash = sm3_hash(full_msg)
68 print("[Server] Original hash: ", original_hash.hex())
69
70 # 攻击者猜测 secret 长度
71 guessed_len = len(secret)
72
73 # 恢复 IV
74 iv = [int.from_bytes(original_hash[i*4:(i+1)*4], 'big') for i in range(8)]
75
76 # 构造 forged_msg = original_msg + padding + append_data
77 fake_msg_len = guessed_len + len(original_msg)
78 padding = sm3_padding(b'A'*fake_msg_len)[fake_msg_len:] # 只保留 padding 部分
79 forged_msg = original_msg + padding + append_data
80

```

```

81 # 构造伪造 hash, 从中间状态继续 hash(append_data)
82 total_bits = (fake_msg_len + len(padding) + len(append_data)) * 8
83 forged_hash = sm3_hash_from_iv(append_data, iv, total_bits)
84 print("[Attacker] Forged hash: ", forged_hash.hex())
85 print("[Attacker] Forged message: ", forged_msg)
86
87 # 服务器验证: hash(secret + forged_msg)
88 true_hash = sm3_hash(secret + forged_msg)
89 print("[Server] True hash: ", true_hash.hex())
90
91 print(" Attack success:", forged_hash == true_hash)

```

## D RFC6962 Merkle

```

1  import hashlib
2  import math
3  from typing import List, Tuple
4
5  # ===== SM3 哈希 (此处使用 sha256 模拟) =====
6  def sm3_hash(data: bytes) -> bytes:
7      return hashlib.sha256(data).digest() # 替换为真实 SM3 哈希
8
9  # ===== RFC6962 Merkle Tree 实现 =====
10 class MerkleTreeRFC6962:
11     def __init__(self, leaves: List[bytes]):
12         self.leaves = [sm3_hash(b'\x00' + leaf) for leaf in leaves] # Leaf prefix
13         # 0x00
14         self.levels = []
15         self.build_tree()
16
17     def build_tree(self):
18         nodes = self.leaves[:]
19         self.levels.append(nodes)
20         while len(nodes) > 1:
21             next_level = []
22             for i in range(0, len(nodes), 2):
23                 left = nodes[i]
24                 if i + 1 < len(nodes):
25                     right = nodes[i + 1]
26                 else:
27                     right = left
28                 parent = sm3_hash(b'\x01' + left + right) # Internal prefix 0x01
29                 next_level.append(parent)
30             nodes = next_level
31             self.levels.append(nodes)
32         self.root = self.levels[-1][0] if self.levels else None

```

```

32
33     def get_root(self) -> bytes:
34         return self.root
35
36     def get_proof(self, index: int) -> List[Tuple[bytes, bool]]:
37         proof = []
38         i = index
39         for level in self.levels[::-1]:
40             if i % 2 == 0:
41                 if i + 1 < len(level):
42                     proof.append((level[i + 1], True)) # 右兄弟
43             else:
44                 proof.append((level[i - 1], False)) # 左兄弟
45             i = i // 2
46         return proof
47
48     @staticmethod
49     def verify_proof(leaf: bytes, index: int, proof: List[Tuple[bytes, bool]], root
50                     : bytes) -> bool:
51         computed_hash = sm3_hash(b'\x00' + leaf)
52         for sibling_hash, is_right in proof:
53             if is_right:
54                 computed_hash = sm3_hash(b'\x01' + computed_hash + sibling_hash)
55             else:
56                 computed_hash = sm3_hash(b'\x01' + sibling_hash + computed_hash)
57         return computed_hash == root
58
59     def get_leaf_index(self, leaf_data: bytes) -> int:
60         leaf_hash = sm3_hash(b'\x00' + leaf_data)
61         try:
62             return self.leaves.index(leaf_hash)
63         except ValueError:
64             return -1
65
66 # ===== 构建 100000 个叶子节点 =====
67 leaf_count = 100_000
68 leaves = [f"leaf-{i}".encode() for i in range(leaf_count)]
69 tree = MerkleTreeRFC6962(leaves)
70 root = tree.get_root()
71 print(" Merkle Root:", root.hex())
72
73 # ===== 存在性证明示例 =====
74 target_index = 12345
75 target_leaf = leaves[target_index]
76 proof = tree.get_proof(target_index)
77 valid = MerkleTreeRFC6962.verify_proof(target_leaf, target_index, proof, root)
78 print(f" Inclusion Proof for leaf-{target_index}: {valid}")

```

```
79 # ===== 不存在性证明思路 (逻辑演示) =====  
80 nonexistent_leaf = b"leaf-100001" # 不存在的节点  
81 index = tree.get_leaf_index(nonexistent_leaf)  
82 if index == -1:  
83     print(" Leaf not in tree, non-inclusion confirmed.")  
84 else:  
85     print(" Leaf found, inclusion proof available.")
```