

## SM3 算法的基本步骤：

填充消息：将消息扩展为适当的长度（即长度为 512 的整数倍）。在原始消息末尾添加比特'1'，添加 k 个比特'0'，k 满足  $(l + 1 + k) \equiv 448 \pmod{512}$ ，添加 64 比特的无符号整数（大端序），值为原始消息比特长度 l。

```
1 个用法
def message_padding(self, message: bytes) -> bytes:
    """ 消息填充函数 """
    original_length = len(message)
    bit_length = original_length * 8

    # 添加填充位
    padded_message = bytearray(message)
    padded_message.append(0x80) # 添加1

    # 添加0直到长度满足要求
    while (len(padded_message) % 64) != 56:
        padded_message.append(0x00)

    # 添加原始消息长度（64位）
    padded_message.extend(struct.pack(_fmt: '>Q', *v: bit_length))

    return bytes(padded_message)
```

SM3 算法使用 8 个 32 位的变量来存储状态信息。使用固定的 256 比特初始值 IV=7380166F 4914B2B9 172442D7 DA8A0600 A96F30BC 163138AA E38DEE4D B0FB0E4E 进行消息拓展

```

def message_expansion(self, block: bytes) -> Tuple[List[int], List[int]]:
    """消息扩展函数"""
    # 将512位块分解为16个32位字
    words = []
    for i in range(16):
        word = struct.unpack(_format: '>I', block[i * 4:(i + 1) * 4])[0]
        words.append(word)

    # 扩展生成68个字
    for j in range(16, 68):
        temp = (words[j - 16] ^ words[j - 9] ^
                self.circular_left_shift(words[j - 3], shift: 15))
        word = (self.permutation_p1(temp) ^
                self.circular_left_shift(words[j - 13], shift: 7) ^
                words[j - 6])
        words.append(word & 0xFFFFFFFF)

    # 生成64个W'字
    words_prime = []
    for j in range(64):
        word_prime = words[j] ^ words[j + 4]
        words_prime.append(word_prime)

    return words, words_prime

```

消息迭代处理：分块处理消息，每次处理 512 位的消息块。

```

def compression_function(self, block: bytes) -> None:
    """压缩函数"""
    # 消息扩展
    words, words_prime = self.message_expansion(block)

    # 初始化工作变量
    a, b, c, d, e, f, g, h = self.state

    # 64轮迭代
    for j in range(64):
        # 计算中间变量
        ss1 = self.circular_left_shift(
            (self.circular_left_shift(a, shift: 12) + e +
             self.circular_left_shift(self.ROUND_CONSTANTS[j], j % 32)) & 0xFFFFFFFF,
            shift: 7
        )
        ss2 = ss1 ^ self.circular_left_shift(a, shift: 12)

        tt1 = (self.boolean_function_f(a, b, c, j) + d + ss2 + words_prime[j]) & 0xFFFFFFFF
        tt2 = (self.boolean_function_g(e, f, g, j) + h + ss1 + words[j]) & 0xFFFFFFFF

        # 更新工作变量
        d = c
        c = self.circular_left_shift(b, shift: 9)
        b = a
        a = tt1

        h = g
        g = self.circular_left_shift(f, shift: 19)
        f = e
        e = self.permutation_p0(tt2)

```

最终输出：经过多次迭代处理后，得到 256 位的哈希值作为结果。

```

def compute_hash(self, message: bytes) -> str:
    """计算SM3哈希值"""
    # 重置状态
    self.state = self.INITIAL_VECTOR.copy()

    # 消息填充
    padded_message = self.message_padding(message)

    # 分块处理
    block_size = 64
    for i in range(0, len(padded_message), block_size):
        block = padded_message[i:i + block_size]
        self.compression_function(block)

    # 返回十六进制字符串
    return ''.join(f'{x:08x}' for x in self.state)

```

### 三、长度拓展攻击的原理

Merkle-Damgård 结构的迭代特性导致了长度拓展攻击的可能性：

哈希值  $H(m)$  本质是消息  $m$  经过填充和分块后，最后一次压缩得到的链接变量  $CV_n$ ；若攻击者已知  $H(m)$  和  $m$  的长度，则可推算出  $m$  的填充内容（因填充仅依赖原始长度）；攻击者可将  $H(m)$  作为新的“初始向量”，对“填充内容 + 新消息”组成的新块进行压缩，直接得到拓展消息的哈希值，无需知晓  $m$  的具体内容。

```
步骤1: 从哈希值恢复内部状态
恢复的状态: 55e12e91650d2fec56ec74e1d3e4ddbfc2ef3a65890c2a19ecf88a307e76a23

步骤2: 计算原始消息填充
原始消息长度: 4 字节
填充大小: 60 字节

步骤3: 计算伪造消息总长度
伪造消息总长度: 544 位

步骤4: 执行长度扩展攻击
攻击者伪造的哈希: b781e69f20ecce36ee833f178c4ec631eac50724342dcb55f0b2ad32b3fd6b43

步骤5: 验证攻击结果
真实扩展消息哈希: b781e69f20ecce36ee833f178c4ec631eac50724342dcb55f0b2ad32b3fd6b43

=== 攻击结果 ===
攻击成功!
哈希值匹配: 是
```

长度拓展攻击具体步骤

SM3 长度拓展攻击的具体步骤

假设攻击者已知：

原始消息  $m$  的哈希值  $h = H(m)$ ；

原始消息  $m$  的长度  $\text{len}(m)$ （以字节或比特为单位）。

攻击步骤如下：

计算原始消息的填充内容

根据  $\text{len}(m)$ ，按 SM3 填充规则计算  $m$  的填充数据  $\text{pad}$ ，使得  $m \parallel \text{pad}$  的总长度为 512 位的整数倍（ $\parallel$  表示拼接）。

构造拓展消息

攻击者选择任意新内容  $m''$ ，构造拓展消息  $m' = m \parallel \text{pad} \parallel m''$ 。

计算拓展消息的哈希值

将  $m'$  按 512 位分块，其中前  $n$  块为  $m \parallel \text{pad}$  对应的块（与原始消息分块一致），剩余块为  $m''$  对应的块  $M_{\{n+1\}}, \dots, M_{\{n+k\}}$ ；

以  $h$ （即原始哈希值，等价于  $CV_n$ ）作为初始链接变量，对  $M_{\{n+1\}}, \dots, M_{\{n+k\}}$  依次应用压缩函数  $CF$ ，得到最终链接变量  $CV_{\{n+k\}}$ ，即  $H(m') = CV_{\{n+k\}}$ 。

```

// 执行长度扩展攻击
static void perform_attack(const string& original_message,
    const string& additional_data,
    const string& original_hash) {
    cout << "=== SM3长度扩展攻击演示 ===" << endl;
    cout << "原始消息: " << original_message << endl;
    cout << "原始哈希: " << original_hash << endl;
    cout << "附加数据: " << additional_data << endl;
    cout << endl;

    // 步骤1: 从哈希值恢复状态
    auto recovered_state = hash_to_state(original_hash);
    cout << "步骤1: 从哈希值恢复内部状态" << endl;
    cout << "恢复的状态: " << state_to_hex(recovered_state) << endl;
    cout << endl;

    // 步骤2: 计算原始消息的填充
    auto original_padding = create_padding(original_message, original_message.length() * 8);
    size_t padding_size = original_padding.size() - original_message.length();

    cout << "步骤2: 计算原始消息填充" << endl;
    cout << "原始消息长度: " << original_message.length() << " 字节" << endl;
    cout << "填充大小: " << padding_size << " 字节" << endl;
    cout << endl;

    // 步骤3: 计算伪造消息的总长度
    uint64_t forged_total_length = (original_message.length() + padding_size + additional_data.length()) * 8;

    cout << "步骤3: 计算伪造消息总长度" << endl;
    cout << "伪造消息总长度: " << forged_total_length << " 位" << endl;
    cout << endl;
}

```

## 基于 SM3 与 RFC6962 的 Merkle 树实现及证明

### 核心定义:

**叶子节点哈希:** 对原始数据前加前缀 0x00 后计算 SM3 哈希 (leaf\_hash 方法);

**内部节点哈希:** 对左右子节点哈希拼接后加前缀 0x01 计算 SM3 哈希 (internal\_hash 方法)。

### 树构建流程:

1. 计算所有叶子节点的哈希值, 作为树的第 0 层 (叶子层);
2. 逐层向上计算内部节点: 每 2 个相邻节点的哈希值合并为父节点哈希 (若节点数为奇数, 最后一个节点与自身合并);
3. 重复步骤 2, 直至顶层仅剩 1 个节点 (根哈希), 树的层次结构存储在 tree\_levels\_中。

### 存在性证明:

**生成** (generate\_inclusion\_proof): 对目标叶子, 从叶子层到根层, 收集每一层中目标节点的“兄弟节点哈希”, 组成证明路径;

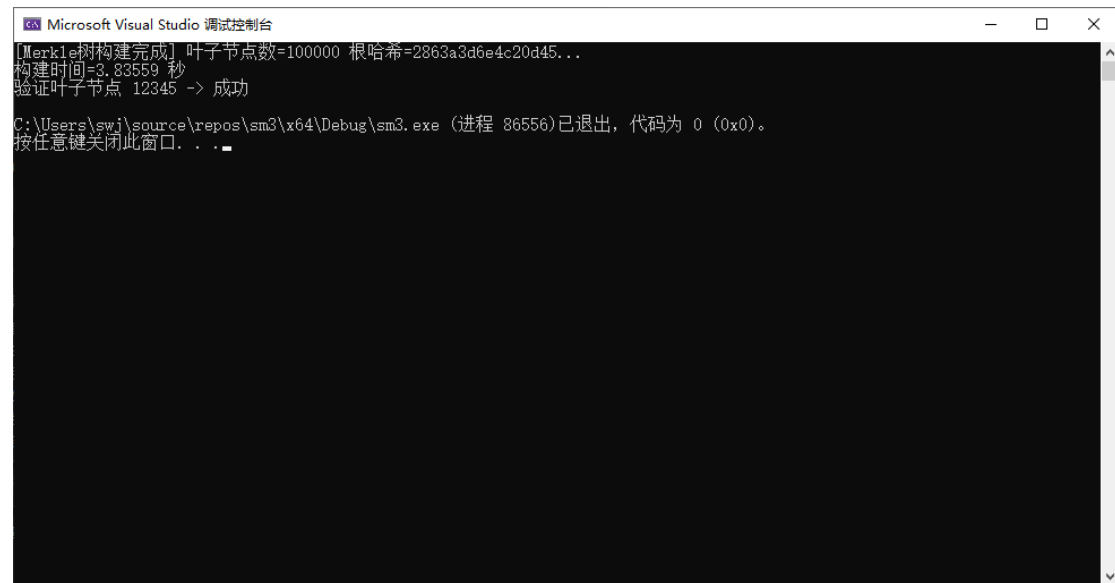
**验证** (verify\_inclusion\_proof): 以目标叶子哈希为起点, 结合证明路径中的兄弟节点, 逐层向上计算哈希, 最终若与根哈希一致, 则证明有效。

### 不存在性证明:

**生成** (generate\_non\_inclusion\_proof): 通过二分查找确定目标数据应插入的位置, 获取该位置左右相邻的叶子节点 (若存在), 并生成这两个叶子的存在性证明;

**验证** (verify\_non\_inclusion\_proof): 验证左右相邻叶子的存在性证明, 同时检查目标数据的哈希值确实在左右相邻叶子的哈希之间 (或超出边界), 从而证明其不存在。

## 验证结果

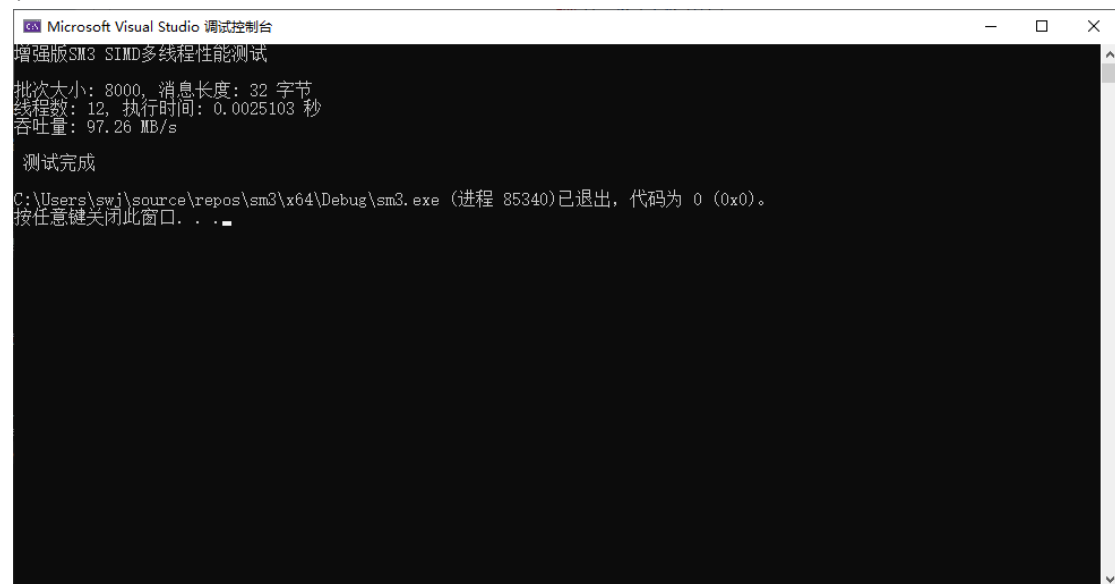


```
Microsoft Visual Studio 调试控制台
[Merkle树构建完成] 叶子节点数=100000 根哈希=2863a3d6e4c20d45...
构建时间=3.83559 秒
验证叶子节点 12345 -> 成功

C:\Users\swj\source\repos\sm3\x64\Debug\sm3.exe (进程 86556)已退出, 代码为 0 (0x0)。
按任意键关闭此窗口. . .
```

## SM3 优化

### 优化结果



```
Microsoft Visual Studio 调试控制台
增强版SM3 SIMD多线程性能测试

批次大小: 8000, 消息长度: 32 字节
线程数: 12, 执行时间: 0.0025103 秒
吞吐量: 97.26 MB/s

测试完成

C:\Users\swj\source\repos\sm3\x64\Debug\sm3.exe (进程 85340)已退出, 代码为 0 (0x0)。
按任意键关闭此窗口. . .
```