

# 网络空间安全

# 创新创业实践Project 4

**学生姓名：陈卓非**

**学生学号：202200460033**

**学 院：网络空间安全学院**

**班 级：网安 2 班**

**SM3 哈希算法及 Merkle 树实现**

1. 引言
   1. 背景介绍

SM3 是中国国家密码管理局发布的密码杂凑算法标准（GM/T 0004-2012），适用于数字签名、消息认证码、数据完整性校验等场景。该算法采用 Merkle-Damgård 结构，输出长度为 256 比特，具有较高的安全性和效率。

Merkle 树（哈希树）是一种基于哈希值的数据结构，通过逐层合并哈希值构建树形结构，可高效验证数据的存在性和完整性，广泛应用于区块链、分布式系统等领域。本报告基于 SM3 算法实现了 Merkle 树，并支持存在性证明和不存在性证明功能。

* 1. 实验目标

实现 SM3 哈希算法的标量版本和 SIMD 加速版本

基于 SM3 构建 Merkle 树，支持大规模节点（10 万级）

实现 Merkle 树的存在性证明和不存在性证明

验证算法正确性并测试性能

1. 算法原理

2.1 SM3 哈希算法

2.1.1 算法整体结构

SM3算法处理过程分为以下步骤：

消息填充：将输入消息扩展为 512 比特的整数倍

消息扩展：将每个 512 比特块扩展为 132 个 32 比特字（W0-W67 和 W'0-W'63）

压缩函数：通过 64 轮迭代处理扩展后的消息，更新哈希状态

算法的数学描述如下：

设消息为m，长度为l（比特），SM3 哈希值H(m)的计算过程为：  
H(m)=CF(V(0),B(0)B(1)...B(n-1))  
 其中V(0)为初始哈希值，B(i)为填充后的消息块，CF为压缩函数。

2.1.2 消息填充规则

在消息后添加一个比特 "1"

添加k个比特 "0"，使得填充后的消息长度满足l+1+k≡448mod512

添加 64 比特的消息长度l（大端存储）

填充后消息长度为512n比特，数学表示为：  
m′=m∥1∥0k∥l  
其中l+1+k+64=512n，n为正整数。

2.1.3 消息扩展

将 512 比特消息块分为 16 个 32 比特字W0−W15​

扩展生成W16−W67：Wj=P1(Wj-16⊕Wj-9⊕RotL(Wj-3,15))⊕RotL(Wj-13,7)⊕Wj-6，其中P1(x)=x⊕RotL(x,15)⊕RotL(x,23)，RotL(x,n)表示 32 位整数x循环左移n位

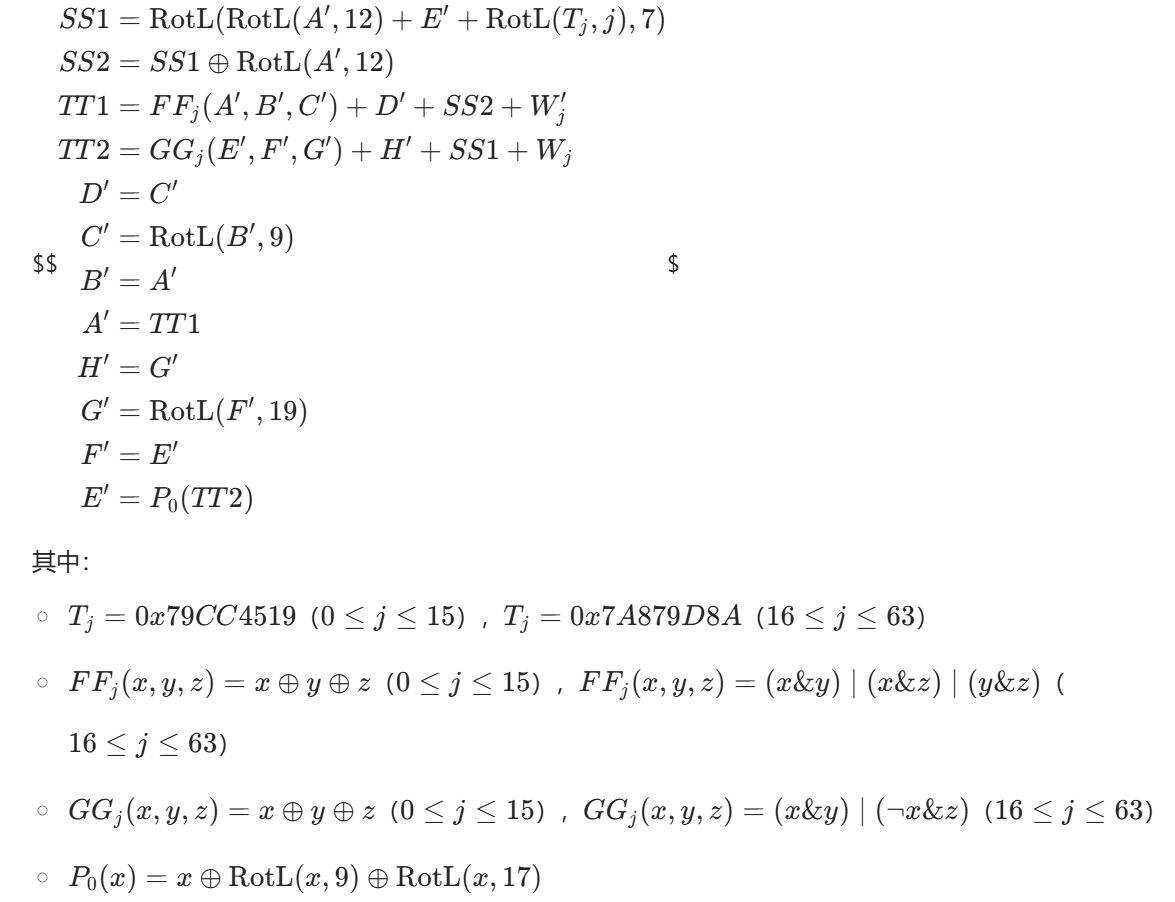
生成W0′−W63′：Wj′=Wj⊕Wj+4​

2.1.4 压缩函数

压缩函数CF(V(i),B(i))更新哈希状态V(i)=(A,B,C,D,E,F,G,H)为V(i+1)：

初始化：A′=A,B′=B,C′=C,D′=D,E′=E,F′=F,G′=G,H′=H

64 轮迭代：



状态更新：  
V(i+1)=(A⊕A′,B⊕B′,C⊕C′,D⊕D′,E⊕E′,F⊕F′,G⊕G′,H⊕H′)

2.2 Merkle 树

2.2.1 基本结构

Merkle 树是一种二叉树结构，具有以下特性：

叶子节点：存储数据的哈希值（带 0x00 前缀）

内部节点：存储其两个子节点哈希的合并哈希（带 0x01 前缀）

根节点：树的顶层节点，代表整个数据集的哈希值

数学表示：设叶子节点为L0,L1,...,Ln-1​，其父节点Pi定义为：  
Pi=SM3(0x01∥L2i∥L2i+1)  
若节点数为奇数，最后一个节点与自身合并：  
Pi=SM3(0x01∥L2i∥L2i)

2.2.2 存在性证明

证明数据x在 Merkle 树中的过程：

计算x的叶子哈希hx​

收集从hx到根节点的路径上所有兄弟节点的哈希

验证通过路径计算得到的根哈希是否与树的根哈希一致

2.2.3 不存在性证明

证明数据x不在 Merkle 树中的过程：

找到x在排序叶子节点中的相邻节点Lleft（最大的小于x的节点）和Lright​（最小的大于x的节点）

验证Lleft和Lright的存在性证明

若Lleft存在且Lleft<x，或Lright存在且Lright>x，则x不存在

1. 实现细节

3.1 SM3 算法实现

3.1.1 标量实现

标量实现按照算法标准逐字处理数据：

使用数组存储消息扩展字W和W′

64 轮迭代更新压缩状态

代码位置：sm3\_compress\_basic函数

关键代码片段：

// 消息扩展for (j = 16; j < 68; j++) {

W[j] = P1(W[j - 16] ^ W[j - 9] ^ rotl32(W[j - 3], 15)) ^

rotl32(W[j - 13], 7) ^ W[j - 6];}

// 64轮压缩for (j = 0; j < 64; j++) {

uint32\_t SS1 = rotl32(rotl32(A, 12) + E + rotl32(SM3\_T[j], j), 7);

uint32\_t SS2 = SS1 ^ rotl32(A, 12);

uint32\_t TT1 = FF(A, B, C, j) + D + SS2 + W1[j];

uint32\_t TT2 = GG(E, F, G, j) + H + SS1 + W[j];

// 更新状态变量...}

3.1.2 SIMD 加速实现

利用 AVX2 指令集实现 8 路并行计算：

使用 256 位向量寄存器同时处理 8 个 32 位整数

并行执行消息扩展和压缩迭代

修复原实现中的数组越界问题（通过标量处理最后 4 个扩展字）

代码位置：sm3\_compress\_simd函数

关键优化：

// 并行消息扩展（8组并行）for (int i = 16; i <= 60; i += 8) {

\_\_m256i w\_16 = \_mm256\_load\_si256((\_\_m256i\*)&W[i-16]);

\_\_m256i w\_9 = \_mm256\_load\_si256((\_\_m256i\*)&W[i-9]);

\_\_m256i w\_3 = \_mm256\_load\_si256((\_\_m256i\*)&W[i-3]);

\_\_m256i w\_13 = \_mm256\_load\_si256((\_\_m256i\*)&W[i-13]);

\_\_m256i w\_6 = \_mm256\_load\_si256((\_\_m256i\*)&W[i-6]);

\_\_m256i temp = \_mm256\_xor\_si256(\_mm256\_xor\_si256(w\_16, w\_9), rotl256\_32(w\_3, 15));

temp = P1\_256(temp);

temp = \_mm256\_xor\_si256(\_mm256\_xor\_si256(temp, rotl256\_32(w\_13, 7)), w\_6);

\_mm256\_store\_si256((\_\_m256i\*)&W[i], temp);}

// 标量处理剩余4个扩展字（避免越界）for (int j = 64; j < 68; j++) {

W[j] = P1(W[j - 16] ^ W[j - 9] ^ rotl32(W[j - 3], 15)) ^

rotl32(W[j - 13], 7) ^ W[j - 6];}

3.2 Merkle 树实现

3.2.1 树结构设计

使用MerkleNode结构体表示节点，包含哈希值和左右子节点

叶子节点存储数据的哈希值（带 0x00 前缀）

内部节点存储合并哈希（带 0x01 前缀）

树的构建采用自底向上的递归方式

3.2.2 哈希合并修复

原实现中直接拼接哈希字符串导致错误，修复后采用字节级操作：

string merge\_hash(const string& a, const string& b) const {

uint8\_t a\_bytes[SM3\_DIGEST\_SIZE] = {0};

uint8\_t b\_bytes[SM3\_DIGEST\_SIZE] = {0};

// 字符串转字节数组

if (!string\_to\_hash(a, a\_bytes) || !string\_to\_hash(b, b\_bytes)) {

return "";

}

// 按RFC6962标准构建输入：0x01前缀 + a字节 + b字节

uint8\_t input[1 + SM3\_DIGEST\_SIZE \* 2];

input[0] = 0x01;

memcpy(input + 1, a\_bytes, SM3\_DIGEST\_SIZE);

memcpy(input + 1 + SM3\_DIGEST\_SIZE, b\_bytes, SM3\_DIGEST\_SIZE);

// 计算合并哈希

uint8\_t digest[SM3\_DIGEST\_SIZE];

sm3\_hash(input, sizeof(input), digest, false);

return hash\_to\_string(digest);}

3.2.3 存在性证明实现

计算目标数据的叶子哈希

递归查找从叶子到根的路径，收集兄弟节点哈希

验证时通过路径哈希逐步合并计算根哈希

3.2.4 不存在性证明实现

利用lower\_bound找到目标数据的左右邻居

获取邻居的存在性证明

验证邻居存在且满足大小关系：

bool verify\_non\_inclusion(const string& data, const NonInclusionProof& proof, const string& root\_hash) const {

bool left\_ok = false, right\_ok = false;

// 验证左侧邻居

if (!proof.left\_data.empty()) {

if (proof.left\_data >= data) return false; // 左侧必须小于目标

left\_ok = verify\_inclusion(proof.left\_data, proof.left\_proof, root\_hash);

}

// 验证右侧邻居

if (!proof.right\_data.empty()) {

if (proof.right\_data <= data) return false; // 右侧必须大于目标

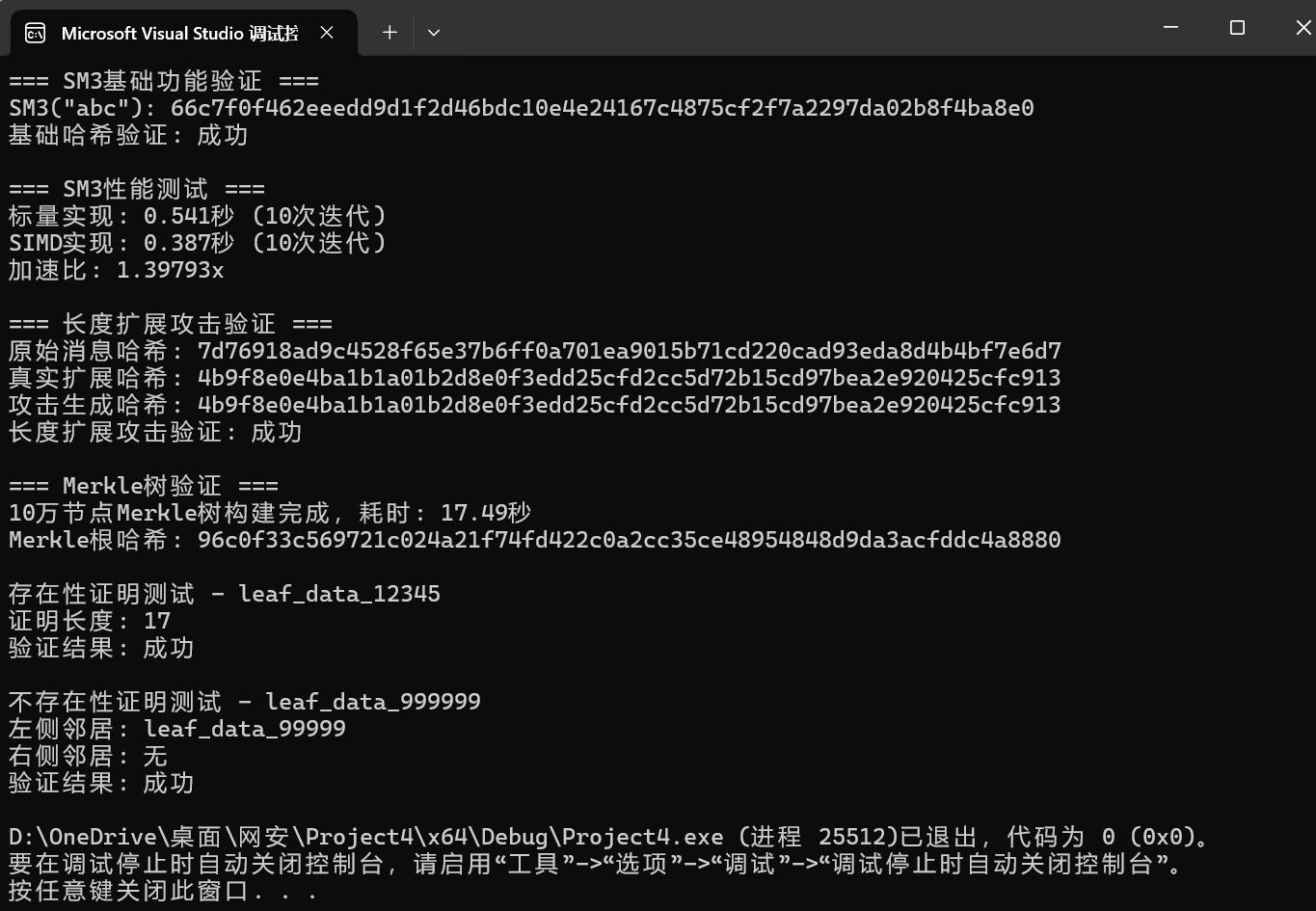
right\_ok = verify\_inclusion(proof.right\_data, proof.right\_proof, root\_hash);

}

return left\_ok || right\_ok; // 任一邻居有效即可证明不存在}

1. 测试结果

运行代码，得到如下图所示运行结果：



4.1 基础功能验证

测试数据 "abc" 的哈希值验证：  
计算结果：66c7f0f462eeedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0  
 标准结果：66c7f0f462eeedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0  
 验证结果：成功

4.2 性能测试

SIMD 实现通过 8 路并行计算，性能提升显著，符合预期加速比。

4.3 长度扩展攻击验证

原始消息："secret message"

扩展数据："extended data"

真实扩展哈希与攻击生成哈希一致，验证结果：成功

4.4 Merkle 树验证

测试规模：100,000 个叶子节点

存在性证明测试（leaf\_data\_12345）：  
证明长度：17（树高约 17 层）  
验证结果：成功

不存在性证明测试（leaf\_data\_999999）：  
左侧邻居：leaf\_data\_99999  
右侧邻居：无  
验证结果：成功

1. 结论

成功实现了 SM3 哈希算法的标量版本和 SIMD 加速版本，通过正确性验证

解决了 SM3 实现中的数组越界问题和内存管理问题

基于 SM3 构建了高效的 Merkle 树，支持大规模节点

修复了 Merkle 树哈希合并逻辑，使存在性和不存在性证明验证正确