# SM3 算法相关实现与验证报告

## 一、SM3 基本软件实现与效率改进

### （一）SM3 算法基本原理

SM3 算法的处理过程主要包括消息填充、消息分组、压缩函数迭代等步骤。

1. **消息填充**：对于任意长度的消息 M，首先将其转换为二进制位串，然后计算填充的位数。填充的规则是：先在消息末尾添加一个 “1”，再添加 k 个 “0”，其中 k 是满足（消息长度 + 1 + k）≡ 448 mod 512 的最小非负整数，最后再添加一个 64 位的二进制数，该数是消息的原始长度（以位为单位）的二进制表示。
2. **消息分组**：经过填充后的消息被分成若干个 512 位的消息块，记为，其中 n 为分组的数量。
3. **压缩函数迭代**：SM3 算法的压缩函数f以一个 256 位的初始值IV和一个 512 位的消息块M\_i作为输入，输出一个 256 位的哈希值。初始值IV是固定的，其值为：0x7380166f, 0x4914b2b9, 0x172442d7, 0xda8a0600, 0xa96f30bc, 0x163138aa, 0xe38dee4d, 0xb0fb0e4e。

压缩函数的具体运算过程如下：

* 将消息块分解为 16 个 32 位的字。
* 扩展生成 68 个 32 位的字和 64 个 32 位的字，其中扩展规则为：

对于j = 16到67，

对于j = 0到63，。

其中，\ll表示循环左移操作。

* 初始化 8 个 32 位的寄存器A, B, C, D, E, F, G, H，其初始值分别为当前的哈希值（初始时为IV）。
* 进行 64 轮迭代运算，对于每一轮j（0 到 63）：

其中T\_j是常量，当时，= 0x79cc4519；当时，。和是布尔函数，当时，；

* 迭代结束后，将A, B, C, D, E, F, G, H分别与初始的寄存器值进行异或运算，得到新的哈希值，作为下一个消息块处理的输入。

当所有消息块处理完毕后，最终得到的哈希值即为 SM3 算法对消息 M 的哈希结果。

### （二）基本软件实现思路及代码细节

1. **数据类型定义**：使用 32 位无符号整数来表示算法中涉及的字，使用字节数组来存储消息和哈希结果。代码中通过uint32\_t和uint8\_t来实现这些数据类型的定义。
2. **消息填充实现**：
   * 代码中sm3\_pad函数实现消息填充功能。首先计算原始消息长度（以位为单位），然后计算需要填充的 “0” 的位数，在消息末尾依次添加 “1”、相应数量的 “0” 和消息长度的 64 位表示。
   * 具体来说，先计算为负数则加 512。然后计算填充后总长度，分配内存后复制原始消息，添加 “1” 位（即），最后以大端模式添加 64 位长度。

|  |
| --- |
| oid sm3\_pad(const uint8\_t \*msg, size\_t len, uint8\_t \*\*padded\_msg, size\_t \*padded\_len) {      CHECK\_LEN(len, SIZE\_MAX - 64);      // 计算原始消息长度(位)      uint64\_t l\_bits = (uint64\_t)len \* 8;      // 计算需要填充的0的位数      uint64\_t k\_bits = (448 - (l\_bits % 512) - 1) % 512;      if (k\_bits < 0) k\_bits += 512;      // 计算填充后总长度(字节)      \*padded\_len = len + 1 + (k\_bits / 8) + 8;      CHECK\_LEN(\*padded\_len, SIZE\_MAX);      // 分配内存并初始化      \*padded\_msg = (uint8\_t \*)calloc(\*padded\_len, 1);      CHECK\_NULL(\*padded\_msg, "内存分配失败 (sm3\_pad)");      // 复制原始消息      if (msg != NULL && len > 0) {          memcpy(\*padded\_msg, msg, len);      }      // 添加'1'位      (\*padded\_msg)[len] = 0x80;      // 添加64位长度(大端模式)      for (int i = 0; i < 8; i++) {          (\*padded\_msg)[\*padded\_len - 8 + i] = (uint8\_t)(l\_bits >> (8 \* (7 - i)));      }  } |

1. **消息扩展实现**：
   * sm3\_expand函数实现消息扩展，生成W和W'数组。首先初始化W[0..15]，然后根据扩展规则计算W[16..67]和W'[0..63]。
   * 对于W[16..67]，按照计算；对于
2. **压缩函数实现**：
   * sm3\_compress函数实现压缩功能，处理单个 512 位块。首先调用sm3\_expand生成W和W'数组，初始化工作寄存器A, B\\_reg, C, D, E, F, G, H，然后进行 64 轮迭代运算。
   * 每轮迭代中，计算SS1、SS2、TT1、TT2，并更新各个寄存器的值，迭代结束后将寄存器值与初始向量异或。

|  |
| --- |
| void sm3\_compress(uint32\_t \*cv, const uint32\_t \*B) {  uint32\_t W[68], W1[64];  sm3\_expand(B, W, W1);    // 初始化工作寄存器  uint32\_t A = cv[0], B\_reg = cv[1], C = cv[2], D = cv[3];  uint32\_t E = cv[4], F = cv[5], G = cv[6], H = cv[7];  uint32\_t SS1, SS2, TT1, TT2;    // 64轮迭代  for (int j = 0; j < 64; j++) {  // 计算SS1  SS1 = ROTL32(ROTL32(A, 12) + E + ROTL32(T(j), j % 32), 7);  SS2 = SS1 ^ ROTL32(A, 12);  TT1 = FF(j, A, B\_reg, C) + D + SS2 + W1[j];  TT2 = GG(j, E, F, G) + H + SS1 + W[j];    // 更新寄存器  D = C;  C = ROTL32(B\_reg, 9);  B\_reg = A;  A = TT1;  H = G;  G = ROTL32(F, 19);  F = E;  E = P0(TT2);  }    // 与初始向量异或  cv[0] ^= A;  cv[1] ^= B\_reg;  cv[2] ^= C;  cv[3] ^= D;  cv[4] ^= E;  cv[5] ^= F;  cv[6] ^= G;  cv[7] ^= H;  }      // 计算原始消息长度(位)      uint64\_t l\_bits = (uint64\_t)len \* 8;      // 计算需要填充的0的位数      uint64\_t k\_bits = (448 - (l\_bits % 512) - 1) % 512;      if (k\_bits < 0) k\_bits += 512;      // 计算填充后总长度(字节)      \*padded\_len = len + 1 + (k\_bits / 8) + 8;      CHECK\_LEN(\*padded\_len, SIZE\_MAX);      // 分配内存并初始化      \*padded\_msg = (uint8\_t \*)calloc(\*padded\_len, 1);      CHECK\_NULL(\*padded\_msg, "内存分配失败 (sm3\_pad)");      // 复制原始消息      if (msg != NULL && len > 0) {          memcpy(\*padded\_msg, msg, len);      }      // 添加'1'位      (\*padded\_msg)[len] = 0x80;      // 添加64位长度(大端模式)      for (int i = 0; i < 8; i++) {          (\*padded\_msg)[\*padded\_len - 8 + i] = (uint8\_t)(l\_bits >> (8 \* (7 - i)));      }  } |

1. **SM3 哈希主函数**：
   * sm3\_hash函数是 SM3 哈希的主函数，首先初始化初始向量cv，对消息进行填充，然后按块处理填充后的消息，将每个块转换为 16 个 32 位大端整数后调用sm3\_compress进行压缩，最后将结果转换为字节数组（大端）作为哈希值。

### （三）执行效率改进

1. **循环展开**：在 64 轮迭代运算中，将循环进行适当展开，减少循环控制语句的执行次数，提高 CPU 的执行效率。例如，将每 8 轮迭代作为一个块进行展开，减少循环变量的更新和判断次数。
2. **常量预计算**：代码中通过#define T(j) ((j) < 16 ? 0x79CC4519 : 0x7A879D8A)定义轮常量，在编译时即可确定，避免在迭代过程中重复计算，节省时间。
3. **位运算优化**：代码中通过#define ROTL32(x, n) (((x) << (n)) | ((x) >> (32 - (n))))实现循环左移操作，合理利用了 CPU 支持的位运算指令，提高运算速度。
4. **内存访问优化**：在消息扩展和迭代过程中，W和W'数组按照访问顺序进行存储，提高了数据的缓存命中率，减少缓存未命中的情况。

## 三、基于 SM3 的长度扩展攻击验证

### （一）长度扩展攻击原理

长度扩展攻击是指对于某些哈希函数，攻击者在已知消息M的哈希值H(M)和消息长度len(M)的情况下，不需要知道消息M的具体内容，就可以计算出消息M || padding(M) || M'的哈希值H(M || padding(M) || M')，其中padding(M)是M的填充部分，M'是攻击者选择的任意消息。

SM3 算法的压缩函数具有迭代性质，其哈希结果是通过对消息块进行连续压缩得到的。对于长度扩展攻击，攻击者可以利用哈希函数的这一性质，将已知的哈希值作为新的初始值，对攻击者选择的消息M'进行压缩运算，从而得到扩展消息的哈希值。

### （二）验证过程及代码细节

1. **准备工作**：选择一个消息M，计算其哈希值H和消息长度len(M)。
2. **计算填充部分**：根据 SM3 的填充规则，计算padding(M)。
3. **构造扩展消息**：扩展消息为M || padding(M) || M'，其中M'是任意选择的消息。
4. **模拟哈希计算**：将已知的哈希值H作为初始值，对M'进行 SM3 哈希计算（此时需要考虑M || padding(M)的长度对填充的影响），得到哈希值H'。
5. **实际计算验证**：直接计算扩展消息M || padding(M) || M'的 SM3 哈希值H''，比较H'和H''是否相等。如果相等，则验证了 SM3 算法存在长度扩展攻击的可能性。

代码中相关实现细节如下：

1. **从哈希值恢复压缩函数中间状态**：sm3\_recover\_state函数实现从哈希值恢复中间状态，将哈希值的字节数组转换为 32 位的初始向量cv。
2. **长度扩展攻击实现**：sm3\_length\_extension函数实现长度扩展攻击。首先恢复原始消息处理后的中间状态，生成原始消息的填充部分，构造扩展消息（原始消息的填充部分 + 后缀），对扩展消息进行填充，计算原始消息已处理的块数，使用恢复的状态继续处理剩余块，最后输出新的哈希值。

|  |
| --- |
| // 生成原始消息的填充部分  uint8\_t \*pad\_msg = NULL;  size\_t pad\_len = 0;  sm3\_pad(NULL, original\_len, &pad\_msg, &pad\_len);  size\_t padding\_only\_len = pad\_len - original\_len;  CHECK\_LEN(padding\_only\_len, SIZE\_MAX - suffix\_len);    // 构造扩展消息: 原始消息的填充部分 + 后缀  \*extended\_len = padding\_only\_len + suffix\_len;  \*extended\_msg = (uint8\_t \*)malloc(\*extended\_len);  CHECK\_NULL(\*extended\_msg, "内存分配失败 (extended\_msg)");    memcpy(\*extended\_msg, pad\_msg + original\_len, padding\_only\_len);  memcpy(\*extended\_msg + padding\_only\_len, suffix, suffix\_len);    // 对扩展消息进行填充  uint8\_t \*padded\_extended = NULL;  size\_t padded\_extended\_len = 0;  sm3\_pad(\*extended\_msg, \*extended\_len, &padded\_extended, &padded\_extended\_len);    // 计算原始消息已处理的块数  size\_t processed\_blocks = (original\_len + SM3\_BLOCK\_SIZE - 1) / SM3\_BLOCK\_SIZE;  size\_t start = processed\_blocks \* SM3\_BLOCK\_SIZE;    // 确保起始位置有效  if (start > padded\_extended\_len) {  start = padded\_extended\_len;  }    // 使用恢复的状态继续处理剩余块  uint32\_t temp\_cv[8];  memcpy(temp\_cv, cv, sizeof(temp\_cv)); |

### （三）数学推导与表示

设H(M)是消息M的 SM3 哈希值，len(M)为M的长度（以位为单位）。padding(M)是M的填充部分，满足len(M) + len(padding(M)) = 512k（k为整数）。

对于扩展消息M'，其长度为len(M')，则扩展消息M || padding(M) || M'的长度为512k + len(M')。

SM3 算法计算哈希值的过程可以表示为：H(M) = Compress(IV, M\_0, M\_1, ..., M\_{n-1})，其中Compress是压缩函数，M\_0, ..., M\_{n-1}是M填充后的消息块。

对于扩展消息，其哈希值，其中是M'填充后的消息块（考虑到M || padding(M)的长度）。

通过上述推导可知，在已知H(M)和len(M)的情况下，可以计算出)，从而验证长度扩展攻击。

## 四、基于 SM3 构建 Merkle 树及相关证明

### （一）Merkle 树原理（RFC6962）

Merkle 树是一种哈希树，由叶子节点、非叶子节点和根节点组成。叶子节点是数据块的哈希值，非叶子节点是其两个子节点哈希值串联后的哈希值，根节点是整个树的顶部节点，代表了所有数据块的哈希值。

根据 RFC6962，Merkle 树的构建规则如下：

1. 叶子节点：对于每个数据块d\_i，计算其哈希值h\_i = SM3(d\_i)，作为叶子节点。
2. 非叶子节点：如果一个节点有两个子节点l和r，则该节点的哈希值为h = SM3(0x01 || l || r)，其中0x01是一个标记位，用于区分内部节点和叶子节点的哈希计算。
3. 根节点：当叶子节点的数量为奇数时，最后一个叶子节点会被复制一份作为其兄弟节点，然后按照非叶子节点的计算规则向上构建，直到得到根节点。

### （二）构建 10w 叶子节点的 Merkle 树及代码细节

1. **准备叶子节点**：获取 10w 个数据块，分别计算它们的 SM3 哈希值，得到 10w 个叶子节点。
2. **处理叶子节点数量**：10w 是偶数，不需要复制叶子节点。将叶子节点两两分组，计算每组的父节点哈希值。例如，对于叶子节点，其父节点哈希值为；对于叶子节点，其父节点哈希值为，以此类推，得到 5w 个非叶子节点。
3. **迭代构建上层节点**：将上一步得到的非叶子节点再次两两分组，按照同样的规则计算其父节点哈希值，得到 2.5w 个节点（若数量为奇数，最后一个节点复制一份）。重复此过程，直到得到一个根节点。

代码中相关实现细节如下：

1. **数据结构定义**：定义了MerkleNode结构体表示 Merkle 树节点，包含哈希值、左右子节点索引和节点索引；定义了MerkleTree结构体表示 Merkle 树，包含节点数组、叶子节点数、总节点数和树的高度。
2. **计算父节点哈希**：merkle\_parent\_hash函数计算父节点哈希，将左右子节点的哈希值串联后计算 SM3 哈希作为父节点哈希。

|  |
| --- |
| static void merkle\_parent\_hash(const uint8\_t \*left, const uint8\_t \*right, uint8\_t \*parent) {  CHECK\_NULL(left, "left指针不能为空");  CHECK\_NULL(right, "right指针不能为空");  CHECK\_NULL(parent, "parent指针不能为空");    uint8\_t buf[SM3\_DIGEST\_SIZE \* 2];  memcpy(buf, left, SM3\_DIGEST\_SIZE);  memcpy(buf + SM3\_DIGEST\_SIZE, right, SM3\_DIGEST\_SIZE);  sm3\_hash(buf, sizeof(buf), parent);  } |

1. **创建 Merkle 树**：merkle\_create函数创建 Merkle 树。首先计算叶子节点数的下一个 2 的幂，确定总节点数和树的高度，分配节点内存，初始化叶子节点（包括填充虚拟叶子节点），然后从倒数第二层开始向上构建非叶子节点。

### （三）叶子的存在性证明及代码细节

存在性证明用于证明某个叶子节点在 Merkle 树中。证明过程如下：

1. 对于要证明存在的叶子节点，找到其在树中的位置。
2. 获取该叶子节点的兄弟节点哈希值。
3. 计算这两个节点的父节点哈希值，然后找到该父节点的兄弟节点哈希值。
4. 重复上述过程，直到计算到根节点。
5. 将获取的兄弟节点哈希值序列和计算得到的根节点与 Merkle 树的根节点进行比较，如果一致，则证明该叶子节点存在。

证明叶子节点h\_5存在：

1. 的兄弟节点是，计算父节点。
2. 的兄弟节点假设为，计算父节点。
3. 以此类推，直到计算出根节点root，若root与树的根节点相同，则证明h\_5存在。

代码中相关实现细节如下：

* **获取存在性证明**：merkle\_prove\_existence函数获取指定叶子节点的存在性证明。从叶子节点向上到根节点，收集兄弟节点哈希值，存储在证明数组中。

|  |
| --- |
| size\_t merkle\_prove\_existence(const MerkleTree \*tree, size\_t leaf\_idx,  uint8\_t \*\*proof, size\_t \*proof\_len) {  if (!tree || leaf\_idx >= tree->leaf\_count || !proof || !proof\_len) {  return 0;  }    size\_t n = (tree->node\_count + 1) / 2; // 叶子层节点数  size\_t idx = n - 1 + leaf\_idx;    if (idx >= tree->node\_count) {  return 0;  }    \*proof\_len = 0;  \*proof = NULL;    // 从叶子节点向上到根节点，收集兄弟节点哈希  while (idx > 0) {  size\_t sibling = (idx % 2 == 0) ? idx - 1 : idx + 1;    if (sibling >= tree->node\_count) {  free(\*proof);  \*proof = NULL;  \*proof\_len = 0;  return 0;  }    // 扩展证明数组  uint8\_t \*new\_proof = (uint8\_t\*)realloc(\*proof, (\*proof\_len + 1) \* SM3\_DIGEST\_SIZE);  if (!new\_proof) {  free(\*proof);  \*proof = NULL;  \*proof\_len = 0;  return 0;  }  \*proof = new\_proof;    // 保存兄弟节点哈希  memcpy(\*proof + (\*proof\_len) \* SM3\_DIGEST\_SIZE, tree->nodes[sibling].hash, SM3\_DIGEST\_SIZE);  (\*proof\_len)++;    // 移动到父节点  idx = (idx - 1) / 2;  }    return \*proof\_len;  } |

* **验证存在性证明**：merkle\_verify\_existence函数验证存在性证明。从叶子节点哈希开始，结合证明数组中的兄弟节点哈希值，逐层向上计算父节点哈希值，最后与根节点哈希值比较，若一致则证明有效。

|  |
| --- |
| int merkle\_verify\_existence(const uint8\_t \*root, const uint8\_t \*leaf\_hash,  const uint8\_t \*proof, size\_t proof\_len, size\_t leaf\_idx) {  if (!root || !leaf\_hash || (proof\_len > 0 && !proof)) {  return 0;  }    uint8\_t current[SM3\_DIGEST\_SIZE];  memcpy(current, leaf\_hash, SM3\_DIGEST\_SIZE);    // 从叶子节点开始，逐层向上计算哈希  for (size\_t i = 0; i < proof\_len; i++) {  uint8\_t parent[SM3\_DIGEST\_SIZE];  if (leaf\_idx % 2 == 1) {  // 当前节点是右子节点，证明节点是左子节点  merkle\_parent\_hash(proof + i \* SM3\_DIGEST\_SIZE, current, parent);  } else {  // 当前节点是左子节点，证明节点是右子节点  merkle\_parent\_hash(current, proof + i \* SM3\_DIGEST\_SIZE, parent);  }  memcpy(current, parent, SM3\_DIGEST\_SIZE);  leaf\_idx /= 2;  }    // 验证计算结果是否等于根哈希  return memcmp(current, root, SM3\_DIGEST\_SIZE) == 0;  } |

### （四）叶子的不存在性证明及代码细节

不存在性证明用于证明某个数据块对应的叶子节点不在 Merkle 树中。根据 RFC6962，不存在性证明需要证明该数据块的哈希值不在叶子节点中，且可以通过相邻的叶子节点及其路径来证明。

证明过程如下：

* 确定要证明不存在的数据块d，计算其哈希值h = SM3(d)。
* 在 Merkle 树的叶子节点中查找h，若未找到，则进行下一步。
* 找到在叶子节点序列中与h相邻的两个叶子节点，假设叶子节点按某种顺序排序）。
* 分别对进行存在性证明。
* 由于h不在之间，且存在于树中，因此证明h不存在于 Merkle 树中。

代码中相关实现细节如下：

* **获取不存在性证明**：merkle\_prove\_non\_existence函数获取指定索引的不存在性证明。根据索引是否超出叶子节点总数，确定邻居节点，获取邻居节点的存在性证明。

|  |
| --- |
| size\_t merkle\_prove\_non\_existence(const MerkleTree \*tree, size\_t idx,  uint8\_t \*\*proof, size\_t \*proof\_len,  uint8\_t \*neighbor\_hash, int \*is\_left\_neighbor) {  if (!tree || !proof || !proof\_len || !neighbor\_hash || !is\_left\_neighbor) {  return 0;  }    // 情况1: idx >= 叶子节点总数 - 取最后一个叶子作为邻居  if (idx >= tree->leaf\_count) {  \*is\_left\_neighbor = 1;  size\_t neighbor\_idx = tree->leaf\_count - 1;  if (neighbor\_idx >= tree->leaf\_count) {  return 0;  }  size\_t node\_idx = (tree->node\_count + 1)/2 - 1 + neighbor\_idx;  if (node\_idx >= tree->node\_count) {  return 0;  }  memcpy(neighbor\_hash, tree->nodes[node\_idx].hash, SM3\_DIGEST\_SIZE);  return merkle\_prove\_existence(tree, neighbor\_idx, proof, proof\_len);  }    // 情况2: idx < 叶子节点总数 - 取右侧邻居  if (idx + 1 >= tree->leaf\_count) {  return 0;  }  \*is\_left\_neighbor = 0;  size\_t node\_idx = (tree->node\_count + 1)/2 - 1 + idx + 1;  if (node\_idx >= tree->node\_count) {  return 0;  }  memcpy(neighbor\_hash, tree->nodes[node\_idx].hash, SM3\_DIGEST\_SIZE);  return merkle\_prove\_existence(tree, idx + 1, proof, proof\_len);  } |

* **验证不存在性证明**：merkle\_verify\_non\_existence函数验证不存在性证明。首先验证邻居节点的存在性证明是否有效，然后检查目标位置是否在合理范围内以证明其不存在。

|  |
| --- |
| int merkle\_verify\_non\_existence(const MerkleTree \*tree, const uint8\_t \*root,  const uint8\_t \*proof, size\_t proof\_len,  const uint8\_t \*neighbor\_hash, int is\_left\_neighbor,  size\_t idx) {  if (!tree || !root || (proof\_len > 0 && !proof) || !neighbor\_hash) {  return 0;  }    // 1. 验证邻居节点的存在性  uint8\_t neighbor\_root[SM3\_DIGEST\_SIZE];  memcpy(neighbor\_root, neighbor\_hash, SM3\_DIGEST\_SIZE);    size\_t temp\_idx = is\_left\_neighbor ? idx - 1 : idx + 1;  if ((is\_left\_neighbor && temp\_idx >= idx) || temp\_idx >= tree->leaf\_count) {  return 0;  }    for (size\_t i = 0; i < proof\_len; i++) {  uint8\_t parent[SM3\_DIGEST\_SIZE];  if (temp\_idx % 2 == 1) {  merkle\_parent\_hash(proof + i \* SM3\_DIGEST\_SIZE, neighbor\_root, parent);  } else {  merkle\_parent\_hash(neighbor\_root, proof + i \* SM3\_DIGEST\_SIZE, parent);  }  memcpy(neighbor\_root, parent, SM3\_DIGEST\_SIZE);  temp\_idx /= 2;  }    if (memcmp(neighbor\_root, root, SM3\_DIGEST\_SIZE) != 0) {  return 0; // 邻居证明无效  }    // 2. 验证目标位置不在树中  if (idx >= tree->leaf\_count) {  return 1; // 索引超出范围，肯定不存在  }    // 检查是否在两个相邻叶子之间  return (is\_left\_neighbor && idx > 0) || (!is\_left\_neighbor && idx + 1 < tree->leaf\_count);  } |