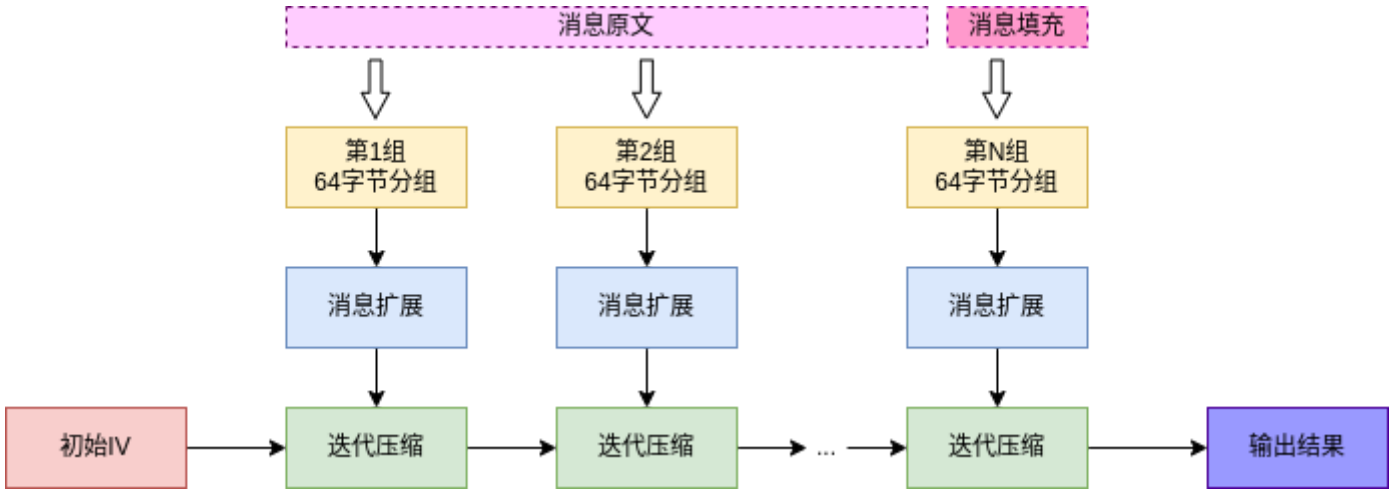
SM3的软件实现与优化

SM3实现和优化

1. SM3算法是一种消息摘要算法，一般用于生成消息以及文件的数字签名，以保证信息的完整性和不可否认性。SM3算法的执行过程共有五部分，分别是：消息填充、消息分组、消息扩展、迭代压缩、输出结果。



2. 消息填充

- 将原始消息填充至长度模512等于448
- 添加64位长度信息

消息扩展

- 将512位的消息分组扩展为132个32位字(W0-W67, W'0-W'63)

压缩函数

- 使用8个初始向量(A-H)与扩展消息进行64轮迭代计算
- 每轮使用不同的布尔函数和常量

输出结果

- 最终将A-H连接起来形成256位杂凑值

3. 优化策略

查表法优化：预计算布尔函数和置换函数的输出，使用查找表代替实时计算

循环展开：展开压缩函数中的64轮循环，减少循环控制开销

SIMD指令优化：使用SSE/AVX等SIMD指令并行处理多个数据，特别适用于消息扩展和压缩函数中的批量操作

消息调度优化：优化消息扩展过程，减少中间变量，利用寄存器重用技术

验证length-extension attack

长度扩展攻击的核心：如果知道 `Hash(M)` 和 `M` 的长度（但不知道 `M` 本身），攻击者可以构造 `M' = M || Pad(M) || NewData`，并计算 `Hash(M')` 而无需知道原始消息 `M`。
这要求算法满足：

- 未对填充后的消息进行保护（如 HMAC 的密钥混淆）。
- 内部状态直接暴露（SM3 的最终哈希值是最后的状态值）。

1. 攻击流程：
    - 从 `Hash(M)` 中提取 SM3 的 内部状态（即 8 个 32 位变量 `A-H`）。
    - 将 `A-H` 作为初始状态，继续处理 `Pad(M) || NewData`。
    - 输出结果应与直接计算 `Hash(M || Pad || NewData)` 一致。
2. 攻击代码

需要注意的是在**消息末尾**总要加一次 `0x80 || 0...0 || len64` 的填充，用长度扩展构造 `M' = M || pad(M) || append` 时，真正参与计算的消息其实是 `M'` 再做一次 padding，其中结尾的 `len64` 是 `|M'|`（padding 之前）的比特长度，否则伪造哈希和真实哈希必然不一致。

代码块

```
// 从哈希值恢复内部状态
void hash_to_state(const uint8_t hash[32], uint32_t state[8]) {
    for (int i = 0; i < 8; i++) {
        state[i] = (hash[i*4]<<24) | (hash[i*4+1]<<16) |
                   (hash[i*4+2]<<8) | hash[i*4+3];
    }
}

// 长度扩展攻击核心函数
void length_extension_attack(
    const uint8_t original_hash[32],
    size_t orig_len,
    const uint8_t *append_data,
    size_t append_len,
    uint8_t forged_hash[32]
) {
    uint32_t state[8];
    hash_to_state(original_hash, state);

    // 原消息做完 padding 后一定在块边界上
    size_t orig_pad_len = (orig_len % 64 < 56) ? (56 - orig_len % 64) : (120 -
orig_len % 64);
```

```
22
23      // 新消息（padding 之前）的总长度: |M'| = |M| + |pad(M)| + |append|
24      size_t new_pre_len = orig_len + orig_pad_len + 8 + append_len;
25
26      // 现在只构造: append || pad_for_M'
27      size_t tail_pad_len = (new_pre_len % 64 < 56) ? (56 - new_pre_len % 64) :
    (120 - new_pre_len % 64);
28      size_t ext_len = append_len + tail_pad_len + 8;
29
30      uint8_t *buf = (uint8_t *)malloc(ext_len);
31      if (!buf) { perror("malloc"); exit(EXIT_FAILURE); }
32
33      // 先放入 append
34      memcpy(buf, append_data, append_len);
35
36      // 然后是对整条新消息 M' 的最终 padding
37      buf[append_len] = 0x80;
38      memset(buf + append_len + 1, 0, tail_pad_len - 1);
39
40      uint64_t new_bit_len = (uint64_t)new_pre_len * 8ULL;
41      for (int i = 0; i < 8; i++) {
42          buf[append_len + tail_pad_len + i] = (new_bit_len >> (56 - 8*i)) &
    0xFF;
43      }
44
45      // 用已知的内部状态继续压缩
46      for (size_t i = 0; i < ext_len; i += 64) {
47          sm3_compress(state, buf + i);
48      }
49
50      // 输出伪造哈希
51      for (int i = 0; i < 8; i++) {
52          forged_hash[i*4+0] = (state[i] >> 24) & 0xFF;
53          forged_hash[i*4+1] = (state[i] >> 16) & 0xFF;
54          forged_hash[i*4+2] = (state[i] >>  8) & 0xFF;
55          forged_hash[i*4+3] =  state[i]        & 0xFF;
56      }
57
58      free(buf);
59  }
```

Merkle树

1. Merkle树(又称哈希树）为典型的二叉树结构，每个非叶子节点都是其子节点哈希值的组合，节点类型包括叶子节点（存储数据块的哈希值）、非叶子节点（存储子节点哈希值的组合哈希）、根节点（树的顶部节点，代表整个数据集的"指纹"）。
2. Merkle树的构建是一个自底向上的递归过程，首先，将待验证的数据分成固定大小的块，如果数据块数量不是2的幂次方，需要复制最后一个块使其补全

分层构建过程：

叶子层构建：计算每个数据块的SM3哈希值（使用0x00前缀）

中间层构建：两两组合叶子哈希值计算SM3哈希（使用0x01前缀）

根节点构建：组合中间层哈希值计算SM3哈希

存在性证明

验证特定数据是否包含在树中：

- 需要提供：目标数据、从叶子到根的路径上的兄弟节点哈希
- 验证过程：从叶子哈希开始，逐步计算到根哈希，与已知根比较

不存在性证明

验证数据不在树中：

- 对于排序Merkle树，证明目标值的前驱和后继的存在性
- 验证前驱<目标<后继