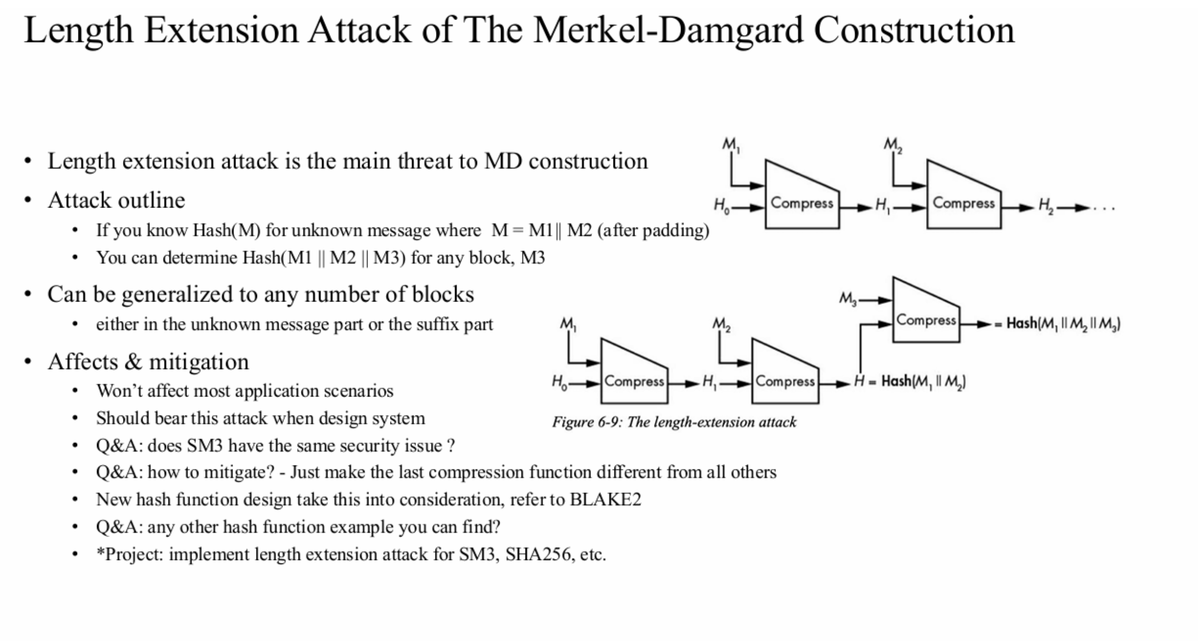
SM3 哈希算法长度扩展攻击实验报告



本实验旨在深入理解 SM3 哈希算法的实现机制，特别是其 Merkle-Damgård 结构的弱点，并通过编程实现和演示长度扩展攻击(Length Extension Attack)。这种攻击利用了哈希函数的状态继承特性，允许攻击者在未知原始消息的情况下扩展消息并计算有效哈希值。

长度扩展攻击原理与实现

长度扩展攻击利用了 SM3 哈希算法的 Merkle-Damgård 结构特性：

哈希函数的输出实际上是处理完所有消息块后的内部状态

这个状态可以作为一个新的初始向量(IV)

攻击者可以在不知道原始消息内容的情况下，基于已知的哈希值继续处理额外消息

具体流程如下：

1、得到某条未知消息 M=M1||padding 的哈希值 H(M)。

2、用该哈希值 H(M)作为初始 IV 向量，与攻击者自选消息 M'共同输入到哈希函数中，得到附加消息之后的哈希值 H'。

3、此时，该未知消息填充之后的内容与攻击者自选消息级联所得到的哈希值 H(M||M')便与此条新哈希值 H'相同，成功实现了碰撞。

代码实现：

```
void sm3_for_length_attack(char plaintext[], unsigned int* hash_val,
                                    int text_len, int total_len) {
    // 使用特定填充函数（考虑整个消息长度）
    int padded_len = bit_stuff_for_length_attack(plaintext, text_len, total_len);
    int block_count = padded_len / 64;

    // 使用给定 IV 处理附加消息
    for (int i = 0; i < block_count; i++) {
        CF(IV, (int*)&plaintext_after_stuffing[i * 64]);
    }

    // 获取结果哈希值
    for (int i = 0; i < 8; i++) {
        hash_val[i] = IV[i];
    }

    // 重置 IV
    memcpy(IV, IV2, 8 * sizeof(unsigned int));
}
```

长度扩展攻击验证函数

```
int sm3_length_attack(char* appended_data, unsigned int* expected_digest,
                            unsigned int* initial_digest, int append_len, int total_len) {
    // 将 IV 设置为初始消息的哈希值
    memcpy(IV, initial_digest, 8 * sizeof(unsigned int));

    // 计算攻击后的哈希值
```

```
    unsigned int attack_digest[8];
    sm3_for_length_attack(appended_data, attack_digest, append_len, total_len);

    // 输出攻击结果
    cout << "The hash obtained by the length extension attack:" << endl;
    dump_buf((char*)attack_digest, 32);

    // 验证攻击是否成功
    if (compare((char*)expected_digest, (char*)attack_digest, 32)) {
        cout << "The length attack succeeded" << endl;
        return 1;
    }
    else {
        cout << "The length attack failed" << endl;
        return 0;
    }
}
```

攻击演示与验证

```
int main() {
    // 原始消息："202"（前 3 字节）
    char m[] = "202200460066";
    unsigned int hash_val[8];
    unsigned int hash_val2[8];

    // 计算"202"的哈希值
    sm3(m, hash_val, 3);
    cout << "Original hash of '202':" << endl;
```

```
    dump_buf((char*)hash_val, 32);

    // 构造扩展消息：原始消息(3B) + 填充(61B) + 附加消息"zzx"(3B)
    bit_stuffing(m, 3); // 获取原始消息的填充版本
    char extended_message[67]; // 3+64=67
    memcpy(extended_message, plaintext_after_stuffing, 64);

    char append_data[] = "zzx";
    memcpy(&extended_message[64], append_data, 3);

    // 计算扩展消息的哈希值
    sm3(extended_message, hash_val2, 67);
    cout << "Manually filled message and its hash:" << endl;
    dump_buf((char*)hash_val2, 32);

    // 执行长度扩展攻击
    cout << "\nLaunching length extension attack...\n";
    sm3_length_attack(append_data, hash_val2, hash_val, 3, 64);

    return 0;
}
```

```
Original hash of '202':
9B 36 68 09 EA 71 DD 9E B8 A4 E8 42 55 A3 66 95 EA CB 52 BB A6 BB 4E B8 5B E3 1B 88 B8 0A C7 5E
Manually filled message and its hash:
73 00 AE 46 0F 3F 7E 09 27 7F D2 C8 87 67 28 25 63 D0 7B 8E 80 11 AF 55 BB 31 2E 26 65 62 9A B1
The hash obtained by the length extension attack:
73 00 AE 46 0F 3F 7E 09 27 7F D2 C8 87 67 28 25 63 D0 7B 8E 80 11 AF 55 BB 31 2E 26 65 62 9A B1
The length attack succeeded
```

成功攻击的关键

填充一致性：

bit_stuff_for_length_attack 函数在填充时考虑了原始消息的总长度

添加的长度值反映了整个消息(原始消息+附加消息)的真实长度

状态继承:

攻击开始时使用原始消息的哈希值作为初始 IV

这模拟了原始消息被完整处理后的状态

消息结构模拟:

攻击正确处理了填充和长度附加

生成了与原始消息||填充||附加消息结构等价的哈希值