山东大学网络空间安全学院

网络空间安全创新创业实践

Project 5 SM3长度扩展攻击

姓名：张麟康

学号：201900301107

1 原理概述

1.1 SM3算法综述

SM3密码杂凑算法适用于商用密码应用中的数字签名和验证、消息认证码的生成和验证以及随机数的生成，可以满足多种密码应用的安全需求。

对于一个长度为l比特的消息m，SM3杂凑算法经过填充、迭代压缩和输出选裁生成杂凑值，杂凑值的输出长度为256比特。

1.2 常数与函数

初始值

IV=7380166f 4914b2b9 172442d7 da8a0600 a96f30bc 163138aa e38dee4d b0fb0e4e

常量

$$T_j=\begin{cases}79cc4519 & 0\leqslant j\leqslant 15\\ 7a879d8a & 16\leqslant j\leqslant 63\end{cases}$$

布尔函数

$$FF_j(X,Y,Z)=\begin{cases}X\oplus Y\oplus Z & 0\leqslant j\leqslant 15\\ (X\wedge Y)\vee(X\wedge Z)\vee(Y\wedge Z) & 16\leqslant j\leqslant 63\end{cases}$$

$$GG_j(X,Y,Z)=\begin{cases}X\oplus Y\oplus Z & 0\leqslant j\leqslant 15\\ (X\wedge Y)\vee(\neg X\wedge Z) & 16\leqslant j\leqslant 63\end{cases}$$

式中 X,Y,Z 为字。

置换函数

$P_0(X)=X\oplus(X<<<9)\oplus(X<<<17)$
$P_1(X)=X\oplus(X<<<15)\oplus(X<<<23)$
式中 X 为字。

1.3 填充

假设消息 m 的长度为 l 比特，则首先将比特“1”添加到消息的末尾，再添加 k 个“0”，k 是满足 $l+1+k\equiv 448 \pmod{512}$ 的最小的非负整数。然后再添加一个 64 位比特串，该比特串是长度 l 的二进制表示。填充后的消息 m' 的比特长度为 512 的倍数。

例如：对消息：01100001 01100010 01100011，其长度 $l=24$，经填充得到比特串：

$$01100001\ \ 01100010\ \ 01100011\ \ 1\ \overbrace{00\cdots00}^{423\text{比特}}\ \overbrace{\underbrace{00\cdots011000}_{l\text{的二进制表示}}}^{64\text{比特}}$$

1.4 迭代压缩

1.4.1 迭代过程

将填充后的消息 m' 按 512 比特进行分组：

$$m' = B^{(0)} B^{(1)} \cdots B^{(n-1)}$$

其中 $n=(l+k+65)/512$。

对 m' 按下列方式迭代：

FOR $i=0$ **TO** $n-1$

$V^{(i+1)} = CF(V^{(i)}, B^{(i)})$

ENDFOR

其中 CF 是压缩函数，$V^{(0)}$ 为 256 比特初始值 IV，$B^{(i)}$ 为填充后的消息分组，迭代压缩的结果为 $V^{(n)}$。

1.4.2 消息扩展

将消息分组 $B^{(i)}$ 按以下方法扩展生成 132 个消息字 $W_0, W_1, \cdots W_{67}, W'_0, W'_1, \cdots W'_{63}$，用于压缩函数 CF：

a) 将消息分组 $B^{(i)}$ 划分为 16 个字 $W_0, W_1, \cdots W_{15}$。

b) **FOR** $j=16$ **TO** 67

$W_j \leftarrow P_1(W_{j-16} \oplus W_{j-9} \oplus (W_{j-3} <<< 15)) \oplus (W_{j-13} <<< 7) \oplus W_{j-6}$

ENDFOR

c) **FOR** $j=0$ **TO** 63

$W'_j = W_j \oplus W_{j+4}$

ENDFOR

1.4.3 压缩函数

令 A, B, C, D, E, F, G, H 为字寄存器，$SS1, SS2, TT1, TT2$ 为中间变量，压缩函数 $V^{(i+1)} = CF(V^{(i)}, B^{(i)})$，$0 \leqslant i \leqslant n-1$。计算过程描述如下：

$ABCDEFGH \leftarrow V^{(i)}$

FOR $j=0$ **TO** 63

$SS1 \leftarrow ((A <<< 12) + E + (T_j <<< (j \bmod 32))) <<< 7$

$SS2 \leftarrow SS1 \oplus (A <<< 12)$

$TT1 \leftarrow FF_j(A, B, C) + D + SS2 + W'_j$

$TT2 \leftarrow GG_j(E, F, G) + H + SS1 + W_j$

$D \leftarrow C$

$C \leftarrow B <<< 9$

$B \leftarrow A$

$A \leftarrow TT1$

$H \leftarrow G$

$G \leftarrow F <<< 19$

$F \leftarrow E$

$E \leftarrow P_0(TT2)$

ENDFOR

$V^{(i+1)} \leftarrow ABCDEFGH \oplus V^{(i)}$

其中，字的存储为 big-endian 格式，左边为高有效位，右边为低有效位。

1.5 杂凑值

$ABCDEFGH \leftarrow V^{(n)}$

输出 256 比特的杂凑值 $y = ABCDEFGH$。

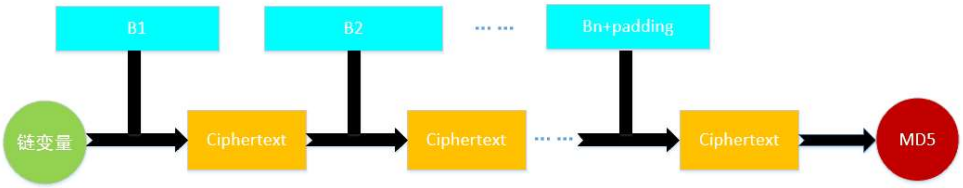
1.6 长度扩展攻击原理

长度扩展攻击是指对于某些允许包含额外信息的密码杂凑函数的攻击手段，对于满足

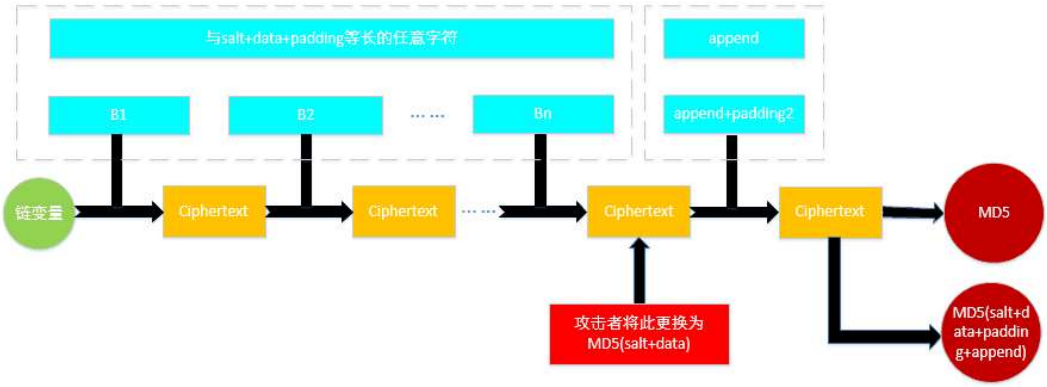
- 加密前将待加密的明文按照一定规则填充到固定长度的倍数。
- 按照该固定长度将明文分块散列并用前一个块的散列结果作为下一个块的初始向量。

的散列函数都可以作为长度扩展攻击的攻击对象。

根据上文中SM3算法具体细节的阐述可以看出SM3满足上述两个特征，为MD结构的散列函数。



通常，对于某一特定满足MD结构的密码杂凑函数H，如果攻击者知道$H(IV||m)$的值并且可以控制m的值，那么攻击者可以设定m为与$m||padding||append$登场的任意字符串m'，然后计算$H(m'||append)$，示意图如下



对于SM3我们已知其初始向量长度和消息拓展方案，因此可以很容易地得到对于某一特定消息m的$padding$和$append$并随即伪造另一个合法的哈希值。

SM3的消息长度是64字节或者它的倍数，如果消息的长度不足则需要进行填充。在填充时，根据算法标准，首先填充一个1，随后填充0，直到消息长度为56(或者再加整数倍的64)字节，最后8字节用来填充消息的长度。

在SM3算法计算杂凑值时，首先对消息进行分组，每组64字节，每一次加密一组，并更新8个初始向量（其初始值在算法中已经标明），下一次用新的初始向量去加密下一个分块。利用这一特性，当我们得到第一次经过杂凑函数的初始向量值时，再构造一组消息用于下一次杂凑，就可以在不知道m的情况下得到合法的杂凑值，这是因为8个初始向量中的值即可表示第一轮的杂凑结果。

2 具体实现

在本项目中，所有程序均采用Python语言编写，具体实现方案如下。

2.1 SM3算法的具体实现

SM3算法完全依据国密算法标准实现，未经任何优化，具体实现如下。

2.1.1 常量和函数

```
IV = [
    1937774191, 1226093241, 388252375, 3666478592,
    2842636476, 372324522, 3817729613, 2969243214,
]

T_j = [
    2043430169, 2043430169, 2043430169, 2043430169, 2043430169, 2043430169,
    2043430169, 2043430169, 2043430169, 2043430169, 2043430169, 2043430169,
    2043430169, 2043430169, 2043430169, 2043430169, 2055708042, 2055708042,
    2055708042, 2055708042, 2055708042, 2055708042, 2055708042, 2055708042,
    2055708042, 2055708042, 2055708042, 2055708042, 2055708042, 2055708042,
    2055708042, 2055708042, 2055708042, 2055708042, 2055708042, 2055708042,
    2055708042, 2055708042, 2055708042, 2055708042, 2055708042, 2055708042,
    2055708042, 2055708042, 2055708042, 2055708042, 2055708042, 2055708042,
    2055708042, 2055708042, 2055708042, 2055708042, 2055708042, 2055708042,
    2055708042, 2055708042, 2055708042, 2055708042
]
```

```python
def sm3_ff_j(x, y, z, j):
    if 0 <= j and j < 16:
        ret = x ^ y ^ z
    elif 16 <= j and j < 64:
        ret = (x & y) | (x & z) | (y & z)
    return ret

def sm3_gg_j(x, y, z, j):
    if 0 <= j and j < 16:
        ret = x ^ y ^ z
    elif 16 <= j and j < 64:
        #ret = (X | Y) & ((2 ** 32 - 1 - X) | Z)
        ret = (x & y) | ((~ x) & z)
    return ret

def sm3_p_0(x):
    return x ^ (rotl(x, 9 % 32)) ^ (rotl(x, 17 % 32))

def sm3_p_1(x):
    return x ^ (rotl(x, 15 % 32)) ^ (rotl(x, 23 % 32))
```

```python
from random import choice


xor = lambda a, b:list(map(lambda x, y: x ^ y, a, b))

rotl = lambda x, n:((x << n) & 0xffffffff) | ((x >> (32 - n)) & 0xffffffff)

get_uint32_be = lambda key_data:((key_data[0] << 24) | (key_data[1] << 16) | (key_data[2] << 8) | (key_data[3]))

put_uint32_be = lambda n:[((n>>24)&0xff), ((n>>16)&0xff), ((n>>8)&0xff), ((n)&0xff)]

padding = lambda data, block=16: data + [(16 - len(data) % block)for _ in range(16 - len(data) % block)]

unpadding = lambda data: data[:-data[-1]]

list_to_bytes = lambda data: b''.join([bytes((i,)) for i in data])

bytes_to_list = lambda data: [i for i in data]

random_hex = lambda x: ''.join([choice('0123456789abcdef') for _ in range(x)])
```

2.2 填充

```python
def padding(msg):
    mlen = len(msg)
    msg.append(0x80)
    mlen += 1
    tail = mlen % 64
    range_end = 56
    if tail > range_end:
        range_end = range_end + 64
    for i in range(tail, range_end):
        msg.append(0x00)
    bit_len = (mlen - 1) * 8
    msg.extend([int(x) for x in struct.pack('>q', bit_len)])
    for j in range(int((mlen - 1) / 64) * 64 + (mlen - 1) % 64, len(msg)):
        global pad
        pad.append(msg[j])
        global pad_str
        pad_str += str(hex(msg[j]))
    return msg
```

2.3 迭代压缩

```
def sm3_cf(v_i, b_i):
    w = []
    for i in range(16):
        weight = 0x1000000
        data = 0
        for k in range(i*4,(i+1)*4):
            data = data + b_i[k]*weight
            weight = int(weight/0x100)
        w.append(data)

    for j in range(16, 68):
        w.append(0)
        w[j] = sm3_p_1(w[j-16] ^ w[j-9] ^ (rotl(w[j-3], 15 % 32))) ^ (rotl(w[j-13], 7 % 32)) ^ w[j-6]
        str1 = "%08x" % w[j]
    w_1 = []
    for j in range(0, 64):
        w_1.append(0)
        w_1[j] = w[j] ^ w[j+4]
        str1 = "%08x" % w_1[j]
```

```
    for j in range(0, 64):
        ss_1 = rotl(
            ((rotl(a, 12 % 32)) +
            e +
            (rotl(T_j[j], j % 32))) & 0xffffffff, 7 % 32
        )
        ss_2 = ss_1 ^ (rotl(a, 12 % 32))
        tt_1 = (sm3_ff_j(a, b, c, j) + d + ss_2 + w_1[j]) & 0xffffffff
        tt_2 = (sm3_gg_j(e, f, g, j) + h + ss_1 + w[j]) & 0xffffffff
        d = c
        c = rotl(b, 9 % 32)
        b = a
        a = tt_1
        h = g
        g = rotl(f, 19 % 32)
        f = e
        e = sm3_p_0(tt_2)

        a, b, c, d, e, f, g, h = map(
            lambda x:x & 0xFFFFFFFF ,[a, b, c, d, e, f, g, h])

    v_j = [a, b, c, d, e, f, g, h]
```

```python
def sm3_hash(msg, new_v):
    # print(msg)
    len1 = len(msg)
    reserve1 = len1 % 64
    msg.append(0x80)
    reserve1 = reserve1 + 1
    # 56-64, add 64 byte
    range_end = 56
    if reserve1 > range_end:
        range_end = range_end + 64

    for i in range(reserve1, range_end):
        msg.append(0x00)

    bit_length = (len1) * 8
    bit_length_str = [bit_length % 0x100]
    for i in range(7):
        bit_length = int(bit_length / 0x100)
        bit_length_str.append(bit_length % 0x100)
    for i in range(8):
```

```python
    group_count = round(len(msg) / 64) - 1

    B = []
    for i in range(0, group_count):
        B.append(msg[(i + 1)*64:(i+2)*64])

    V = []
    V.append(new_v)
    for i in range(0, group_count):
        V.append(sm3_cf(V[i], B[i]))

    y = V[i+1]
    result = ""
    for i in y:
        result = '%s%08x' % (result, i)
    return result
```

2.4 长度扩展攻击的具体实现

在SM3密码杂凑算法上实现长度扩展攻击，主要包含如下几个步骤：

- 随机选择一个消息m，计算其过SM3杂凑算法的结果h_1。
- 生成一个附加消息m'，用h_1的结果推断杂凑结束后八个初始向量IV的值，再将其作为初始向量过SM3杂凑密码算法m'得到另一个杂凑值h_2。
- 计算$m||padding||m'$的杂凑值h_3，若攻击成功则h_2与h_3的值相等。

首先随机生成一个浮点数（64字节）作为最初的消息m并计算得到其杂凑值h_1。随后将杂凑值按照8字节为一组进行分组得到第一次加密后8个初始向量的值。

得到初始向量值后开始构造消息，由于已知消息m的长度，所以这一部分可以用等长的字符代替并进行填充从而得到64字节的消息，随后将附加消息m'追加到其后便构造完成新的消息。

随后运行SM3算法，此时将原有SM3算法稍作修改，存储每轮更新之后的初始向量值，具体代码如下：

```
    group_count = round(len(msg) / 64) - 1

    B = []
    for i in range(0, group_count):
        B.append(msg[(i + 1)*64:(i+2)*64])

    V = []
    V.append(new_v)
    for i in range(0, group_count):
        V.append(sm3_cf(V[i], B[i]))

    y = V[i+1]
    result = ""
    for i in y:
        result = '%s%08x' % (result, i)
    return result
```

对于消息的伪造，声明方法forge()，获取初始向量值，根据原始消息的长度构造满足条件的伪造消息长度，在这里用字符'z'填充。

```
def forge(h0, length, appending):
    iv = []
    msg = ""
    # 将原始消息的杂凑值进行分组 得到最新的初始向量值
    for i in range(0, len(h0), 8):
        iv.append(int(h0[i:i + 8], 16))

    # 伪造等长的消息 如果长度大于64 用z填充
    if length > 64:
        for i in range(0, int(length / 64) * 64):
            msg += 'z'
    for i in range(0, length % 64):
        msg += 'z'
    msg = func.bytes_to_list(bytes(msg, encoding='utf-8'))
    msg = padding(msg)
    msg.extend(func.bytes_to_list(bytes(appending, encoding='utf-8')))
    return sm3_modified.sm3_hash(msg, iv)
```

```python
def padding(msg):
    length = len(msg)
    msg.append(0x80)
    length += 1
    tail = length % 64
    range_end = 56
    if tail > range_end:
        range_end = range_end + 64
    for i in range(tail, range_end):
        msg.append(0x00)
    bit_len = (length - 1) * 8
    msg.extend([int(x) for x in struct.pack('>q', bit_len)])
    for j in range(int((length - 1) / 64) * 64 + (length - 1) % 64, len(msg)):
        global pad
        global pad_str
        pad.append(msg[j])
        pad_str += str(hex(msg[j]))
    return msg
```

接下来首先使用random模块中的random()方法生成一个浮点数并计算其哈希值、获取其长度，令附加消息为'201900301107张麟康'并构造待伪造杂凑值h1，接下来生成消息m1，根据上文中的分析可以知道通过填充和附加消息其计算出的哈希值h2必定与h1相等。

```python
m0 = str(random.random())
h0 = sm3.sm3_hash(func.bytes_to_list(bytes(m0, encoding='utf-8')))
len0 = len(m0)
appending = "201900301107张麟康"    # 附加消息
pad_str = ""
pad = []

h1 = forge(h0, len0, appending)
m1 = func.bytes_to_list(bytes(m0, encoding='utf-8'))
m1.extend(pad)
m1.extend(func.bytes_to_list(bytes(appending, encoding='utf-8')))
m1_str = m0 + pad_str + appending

h2 = sm3.sm3_hash(m1)
```

运行上述程序，得到结果如下，可以看出得到一个成功的伪造，针对于SM3密码杂凑算法的长度扩展攻击成功。

```
PS F:\course-project-2022> & D:/Python310/python.exe "f:/course-project-2022/Project 5 SM3 length extension/sm3_length_extension_attack.py"
原始消息m: 0.5877823286290847
原始消息m的长度:18
原始消息杂凑值h:9dd12de94b11b3355b6251df6b81404bb9797bd11c156da8f35a8060a8b6cdcd
附加消息m': 201900301107
待伪造杂凑值h1:1683a5465576cce0a8558e9439c6ad7e1ed77115b67360dcbc8880bd32209c8e
构造新消息:
0.58778232862908470x800x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x90201900301107
新消息杂凑值h2:1683a5465576cce0a8558e9439c6ad7e1ed77115b67360dcbc8880bd32209c8e
```