

网络空间安全创新创业实践报告



山东大学

SHANDONG UNIVERSITY

姓 名：_____孔白哲_____ 学 号：__202100460094__

学 院：_____网络空间安全学院（研究院）_____

专 业：_____网络空间安全_____

年 级：☐一年级 ☒二年级 ☐三年级 ☐其它_____

山东大学制

Project1

implement the naïve birthday attack of reduced SM3

实验介绍

生日攻击

生日攻击是一种密码学攻击方法，利用生日悖论来寻找两个不同的输入，它们产生相同的哈希值或者其他指纹值。该攻击方法可以用于破解哈希函数的强度或者构造冲突。

实现方式

步骤一:选择哈希算法和哈希值长度

首先，选择要攻击的哈希算法和相应的哈希值长度。通常，生日攻击的难度随着哈希值长度的增加而增加。

步骤二:生成随机字符串

实现一个函数来生成随机字符串，以用作攻击过程中的输入。随机字符串的长度应该足够长，以提高碰撞的概率。

步骤三:构造哈希前缀集合

创建一个空的哈希前缀集合，用于存储已经计算过的哈希前缀。

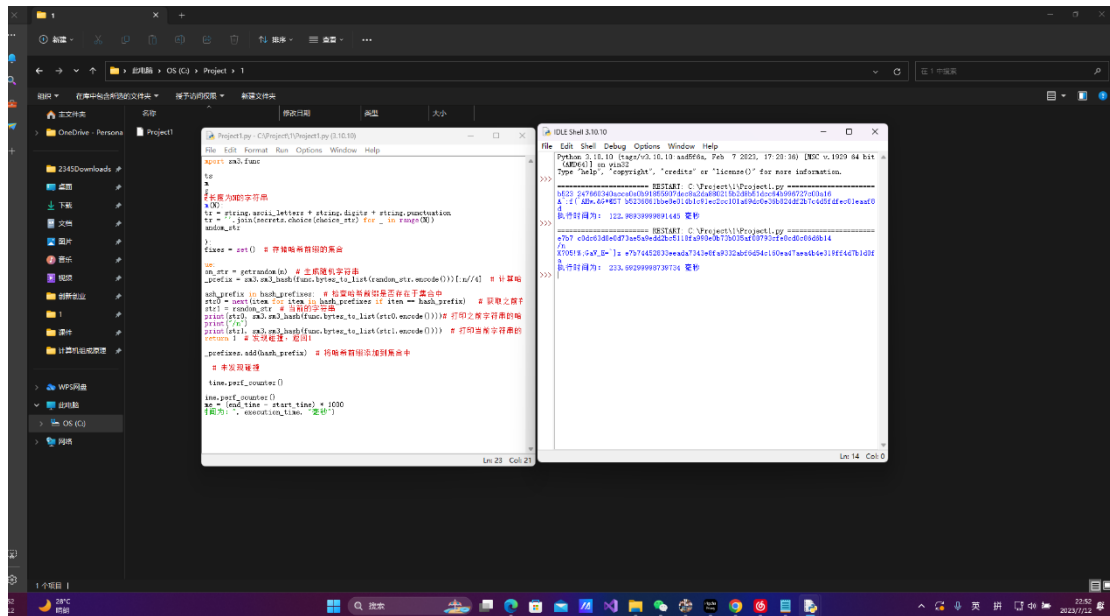
步骤四:进行生日攻击

- 1、生成一个随机字符串。
- 2、对随机字符串进行哈希计算，获取其哈希值。
- 3、提取哈希值的前缀（长度根据选定的哈希算法和长度确定）。
- 4、检查哈希前缀是否存在于哈希前缀集合中。
- 5、如果存在，说明找到了两个不同的字符串，它们的哈希前缀相同，发现了碰撞。
- 6、如果不存在，将哈希前缀添加到哈希前缀集合中，继续下一轮循环。

步骤五:处理碰撞结果

一旦发现碰撞，可以获取碰撞的字符串和对应的哈希值，用于进一步分析和验证攻击的成功性。

运行效果



运行时间: 233ms

CPU: AMD Ryzen 9 5900HX with Radeon Graphics 3.30 GHz

Project2

implement the Rho method of reduced SM3

实验说明

SM3

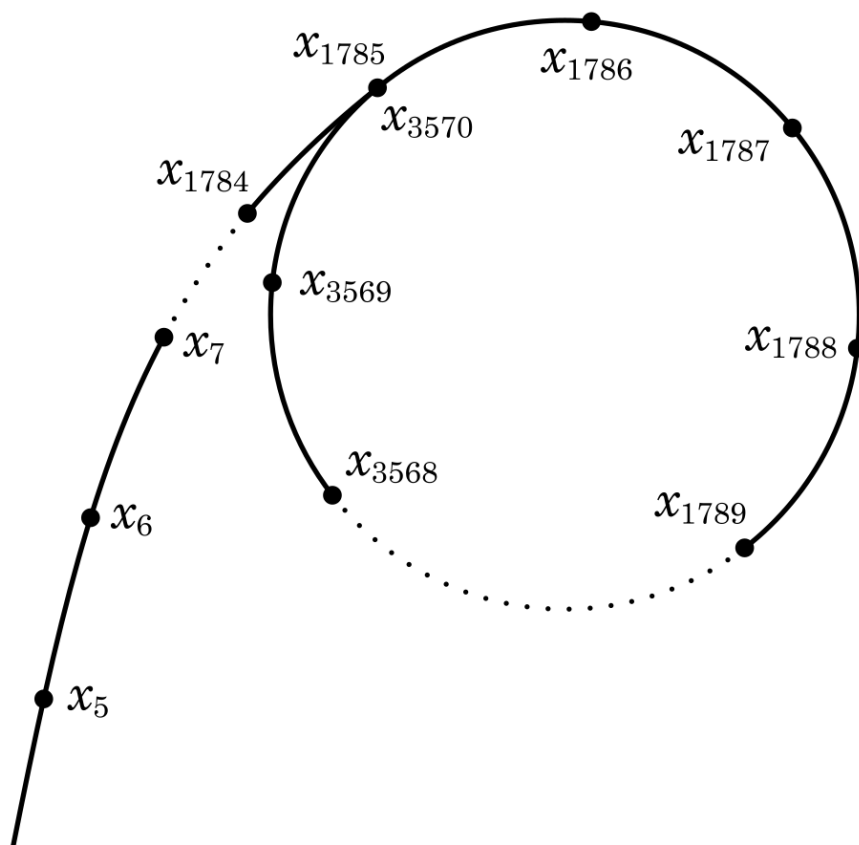
ShangMi 3 (SM3)是中国国家标准中使用的加密散列函数。国家密码管理局于2010-12-17以“GM/T 0004-2012: SM3 密码学哈希算法”的名称发布。

SM3 用于实现数字签名、消息验证码和伪随机数生成器。该算法是公开的，在安全性和效率方面被认为与 SHA-256 相似。SM3 与传输层安全一起使用。

Rho

https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm

Rho 方法是一个简单的碰撞搜索算法，用于在 SM3 哈希函数中找到两个不同的消息 (m_1 和 m_2)，它们的前 exm 位 SM3 哈希值相同。



下面将介绍 Rho 方法的实现过程：

1、输入参数 exm

这个参数指定了要搜索碰撞的位数。在示例代码中，exm 设置为 16，表示搜索前 16 位的碰撞。

2、随机生成初始消息 x

首先，在范围 $[0, 2^{(exm+1)-1}]$ 内随机生成一个整数，然后将其转换为 16 进制字符串作为初始消息 x。

3、对 x 进行迭代

通过调用 SM3.SM3(x)，计算消息 x 的 SM3 哈希值 x_1。然后再次计算 x_1 的 SM3 哈希值，得到 x_2。

4、碰撞搜索循环

在 x_1 和 x_2 的前 num 位 (num 为 exm 的 1/4) 不相同的情况下，对 x_1 和 x_2 进行迭代，直到找到两者前 num 位相同的情况。此时得到的 x_1 和 x_2 就是碰撞消息。

5、返回结果：

找到碰撞后，返回 x_1 和 x_2 的前 num 位 SM3 哈希值（表示为 col），以及 x_1 和 x_2 的原始消息。

在主程序中，使用 Rho 方法搜索前 16 位的碰撞，并计算执行时间。该算法只是一个简单的示例，不适用于实际应用。

运行效果

找到碰撞！

消息1: a1755fb6c2e7a064f7b0bed571ce84fea056751968b920ca5027b3f11f321c79

消息2: 49dbd9c9c3faaadce6e778c78077ee4558257de2bc36273585c705004e029381

两者哈希值的前16bit相同，16进制表示为:c69f

执行时间为: 211297.08079993725 毫秒

找到碰撞！

消息1: 2657b9fb471c49bd87c5ce527a9b601b15ff8aa69400a1a29e417175d9e3b38a

消息2: 286d7715dfce6067692298ffdf3c0969f1ac573e33c71aa44da7e6a37947c02b

两者哈希值的前16bit相同，16进制表示为:9d43

执行时间为: 156273.26859999448 毫秒

找到碰撞！

消息1: 2b0f59bcfbdb6612f7e434120176292ecf6440e38c836a9ba4b3471152639b67d

消息2: ac11ea6af8f4349a490e98efe7c969dbb0ff0a76d96d949839e520af65df3c6f

两者哈希值的前16bit相同，16进制表示为:a2eb

执行时间为: 64214.17739999015 毫秒

16bit 碰撞平均运行时间: 143.928s

CPU: AMD Ryzen 9 5900HX with Radeon Graphics 3.30 GHz

Project3

implement length extension attack for SM3

实验介绍

长度攻击：

Length Extension Attack of The Merkle-Damgård Construction

- Length extension attack is the main threat to MD construction
- Attack outline
 - If you know Hash(M) for unknown message where $M = M_1 || M_2$ (after padding)
 - You can determine Hash($M_1 || M_2 || M_3$) for any block, M_3
- Can be generalized to any number of blocks
 - either in the unknown message part or the suffix part
- Affects & mitigation
 - Won't affect most application scenarios
 - Should bear this attack when design system
 - Q&A: does SM3 have the same security issue ?
 - Q&A: how to mitigate? - Just make the last compression function different from all others
 - New hash function design take this into consideration, refer to BLAKE2
 - Q&A: any other hash function example you can find?
 - *Project: implement length extension attack for SM3, SHA256, etc.

Figure 6-9: The length-extension attack

在密码学和计算机安全中，长度扩展攻击（英语：Length extension attacks）是指一种针对特定加密散列函数的攻击手段，攻击者可以利用 H （消息 1）和消息 1 的长度，不知道消息 1 内容的情形下，将攻击者控制的消息 2 计算出 H （消息 1 \parallel 消息 2）。

该攻击适用于在消息与密钥的长度已知的情形下，所有采取了 H （密钥 \parallel 消息）此类构造的散列函数。MD5 和 SHA-1 等基于 Merkle - Damgård 构造的算法均对此类攻击显示出脆弱性。注意，由于密钥散列消息认证码（HMAC）并未采取 H （密钥 \parallel 消息）的构造方式，因此不会受到此类攻击的影响（如 HMAC-MD5、HMAC-SHA1）。SHA-3 算法对此攻击免疫。

实现方式：

以 SM3 实现代码为基础，对长度攻击进行介绍

- 计算原始消息的 SM3 哈希值。 `h_m = SM3.SM3(msg)`
- 将 SM3 哈希值分割成 8 个 32 位整数。 `Hm = [int(h_m[i * 8:i * 8 + 8], 16) for i in range(8)]`

- 3、计算总消息位长并表示为 16 进制字符串。`len_e = hex((n + len(ext)) * 4)[2:]`。这里的 `n` 是原始消息经过填充后的 16 进制数个数。
- 4、补全表示总消息位长的 16 进制字符串，使其长度为 16 位。`len_e = (16 - len(len_e)) * '0' + len_e`
- 5、根据扩展部分 `ext` 的长度，构造要添加的新数据并附加到 `ext` 后面。具体的添加过程是，如果 `ext` 的长度对 128 取模后大于 112，则在 `ext` 后面添加足够的零字符 `'0'`，使其长度满足特定要求，然后再将表示总消息位长的 16 进制字符串 `len_e` 添加到 `ext` 的末尾。
- 6、将构造的扩展部分 `ext` 进行分组，即将其转换为一系列 16 进制数。
- 7、初始化 `V` 列表为 `Hm`: `V = [Hm]`。`V` 列表用于存储压缩函数输出的中间状态。
- 8、通过循环，将分组后的 `ext` 逐个与 `V` 列表中的最后一个元素（即上一轮的中间状态）一起传入压缩函数 `SM3.CF(V, ext_g, i)` 进行计算，并将计算得到的中间状态添加到 `V` 列表中：`for i in range(len_ext_g): V.append(SM3.CF(V, ext_g, i))`
- 9、最终，`V` 列表中的最后一个元素即为构造的新消息的哈希值，将它转换为十六进制字符串并返回：`res = '' for x in V[len_ext_g]: res += hex(x)[2:] return res`。

实现效果

```
----- test.py: C:\projects\demo.py -----
新消息的哈希值为: 488da9eae30ba06c7bae869d9b99d25077e0a4f8cdd9ba4f32f5ea7bcc5907
aa
长度扩展攻击结果: 488da9eae30ba06c7bae869d9b99d25077e0a4f8cdd9ba4f32f5ea7bcc5907
aa
长度扩展攻击成功!
执行时间为: 42.52379998797551 毫秒
```

运行速度: 42ms

CPU: AMD Ryzen 9 5900HX with Radeon Graphics 3.30 GHz

Project4

do your best to optimize SM3 implementation (software)

实验介绍

SM3

SM3 是一种国密算法，具体是一种分组密码算法，分组长度为 128bit，以字（32 位）为单位进行加密运算。主要分为消息扩展、消息压缩、迭代压缩函数三个部分。

SM3 的具体内容参考：<https://www.doc88.com/p-9347835163554.html?s=rel&id=9>

优化方式：基于 SIMD 的优化技术，主要有循环展开、流水线、分块等

实现方式：主要利用了循环展开的优化方式，将消息扩展、消息压缩部分的循环进行展开，减少循环次数。同时我尝试使用更改函数计算方式，例如将异或函数更改成运用 C++ 自带的异或运算符 \wedge ，但是会报错，报错原因是异或的两个字符串的位数不同，我尝试了一些方法都会报错，所以放弃。

实验效果

优化前加密速度：

```
杂凑值：
66C7F0F4 62EEEDD9 D1F2D46B DC10E4E2 4167C487 5CF2F7A2 297DA02B 8F4BA8E0
time: 144 ms
```

各个函数的 CPU 利用时间：

热路径		
函数名	CPU 总计[单位, %]	自 CPU [单位, %]
外部代码	153 (100.00%)	13 (8.50%)
mainCRTStartup	140 (91.50%)	0 (0.00%)
__srt_common_main	140 (91.50%)	0 (0.00%)
__srt_common_main_seh	140 (91.50%)	0 (0.00%)
invoke_main	139 (90.85%)	0 (0.00%)
main	139 (90.85%)	0 (0.00%)
iteration	137 (89.54%)	0 (0.00%)
compress	112 (73.20%)	0 (0.00%)
ModAdd	47 (30.72%)	0 (0.00%)
FF	18 (11.76%)	0 (0.00%)

iteration	138 (92.00%)	0 (0.00%)	SM3_C++.exe	内核
compress	108 (72.00%)	0 (0.00%)	SM3_C++.exe	内核
extension	30 (20.00%)	0 (0.00%)	SM3_C++.exe	内核

优化后加密速度:

杂凑值:
66C7F0F4 62EEEDD9 D1F2D46B DC10E4E2 4167C487 5CF2F7A2 297DA02B 8F4BA8E0
time: 143 ms

各个函数的 CPU 利用时间:

函数名	CPU 总计[单位, %]	自 CPU [单位, %]
[外部代码]	143 (100.00%)	11 (7.69%)
mainCRTStartup	132 (92.31%)	0 (0.00%)
__srt_common_main	132 (92.31%)	0 (0.00%)
__srt_common_main_seh	132 (92.31%)	0 (0.00%)
invoke_main	130 (90.91%)	0 (0.00%)
main	130 (90.91%)	0 (0.00%)
iteration	129 (90.21%)	0 (0.00%)
compress	102 (71.33%)	0 (0.00%)
ModAdd	37 (25.87%)	0 (0.00%)
GG	17 (11.89%)	0 (0.00%)
FF	16 (11.19%)	0 (0.00%)

参考文献: SM3 的 C++ 实现 https://blog.csdn.net/nicai_hualuo/article/details/121555000

Project5

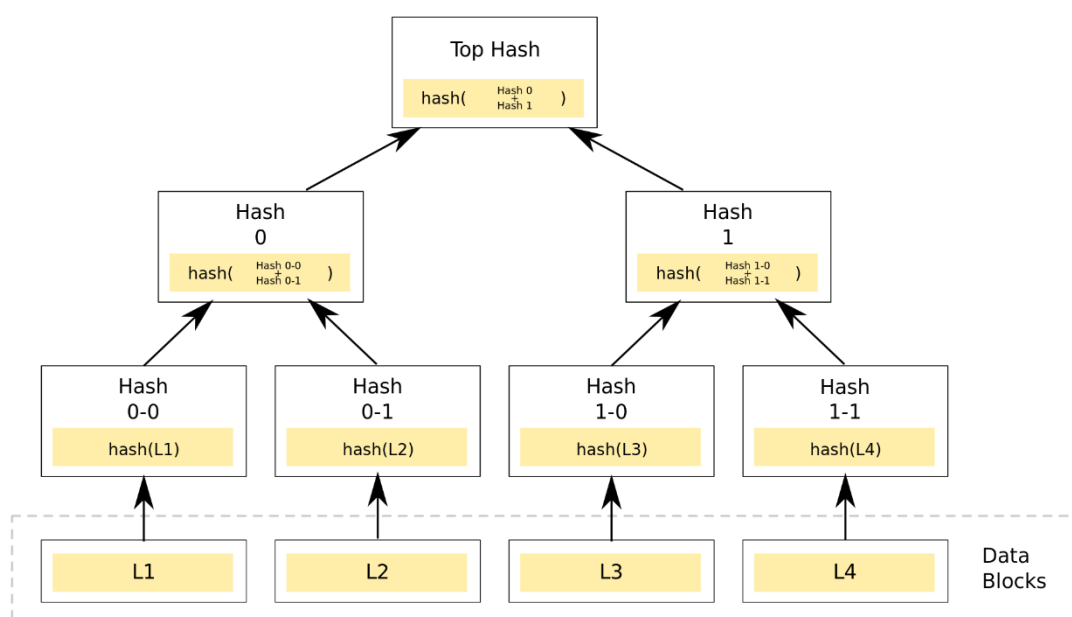
Impl Merkle Tree following RFC6962

实验说明

Merkle Tree

Merkle Tree, 通常也被称作 Hash Tree, 顾名思义, 就是存储 hash 值的一棵树。Merkle 树的叶子是数据块(例如, 文件或者文件的集合)的 hash 值。非叶节点是其对应子节点串联字符串的 hash。

Merkle 树看起来非常像二叉树, 其叶子节点上的值通常为数据块的哈希值, 而非叶子节点上的值, 所以有时候 Merkle tree 也表示为 Hash tree, 如下图所示:



实现方式

默克尔树节点定义:

定义了一个 MerkleTreeNode 类, 其中包含了左子节点、右子节点、父节点以及节点的值(哈希值)等属性。这个类用于构建树的节点。

默克尔树结构定义: 定义了一个 MerkleTree 类, 其中包含了叶子节点列表、根节点以及用于存储整个树中节点值的 mtlist 列表。MerkleTree 类有以下几个重要方法:

sha256_leaf(value):

这个方法用于计算叶子节点的哈希值。它对节点的值进行了级联操作，并使用 SHA-256 哈希算法对级联后的字符串进行哈希计算。

sha256_node(value):

这个方法用于计算中间节点的哈希值。同样，它对节点的值进行了级联操作，并使用 SHA-256 哈希算法对级联后的字符串进行哈希计算。

create_MerkleTree():

这个方法用于构建默克尔树。它先创建叶子节点，并对每个叶子节点计算其哈希值。然后，从叶子节点开始，逐层构建中间节点，直到生成根节点。中间节点的哈希值是由其两个子节点的哈希值计算得到。

Inorder(root):

这个方法用于中序遍历整个默克尔树，得到树中所有节点的值，并存储在 mtlist 列表中。

proof(root, nodevalue):

这个方法用于验证给定的节点值是否存在于默克尔树中。它通过调用 Inorder 方法得到整个树的节点值列表，并检查给定的节点值是否在列表中，从而判断节点是否在树中。

主程序部分:

在主程序中，首先生成 10*5 个随机整数作为叶子节点的值，并将其转换为对应的十六进制表示。然后，利用这些叶子节点值构建默克尔树，并输出树的根节点的值。接着，验证一些随机生成的节点值是否在默克尔树中，并输出验证结果。

运行效果

```
内哈希值
0x00
+ "00" + value[2:]
0x01
+ "01" + value[2:]

self.leaf[i].value)
le(hashvalue)
)

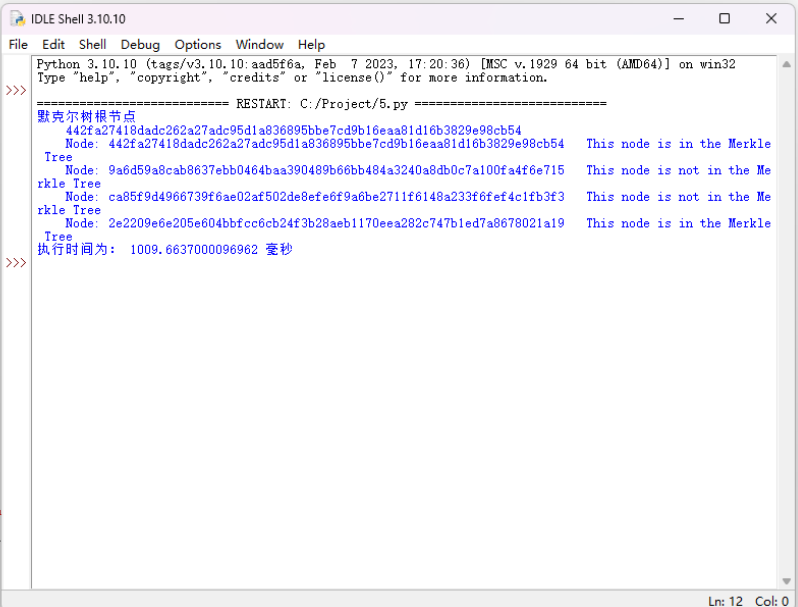
Mhash

子节点的hash

h), 2):

的hash值进行相加再hash
256_node(lchild.val
eeNode(parenthash)
child
child
rtnode

个子节点的值级联后hash作为父节点的值
]
256_node(lchild.value+rchild.value)
```



```
Python 3.10.10 (tags/v3.10.10:aad5f6a, Feb 7 2023, 17:20:36) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Project/5.py =====
默克尔树根节点
442fa27418dad262a27adc95d1a836895bbe7cd9b16eaa81d16b3829e98cb54
Node: 442fa27418dad262a27adc95d1a836895bbe7cd9b16eaa81d16b3829e98cb54 This node is in the Merkle
Tree
Node: 9a6d59a8cab8637ebb0464baa390489b66bb484a3240a8db0c7a100fa4f6e715 This node is not in the Me
rkle Tree
Node: ca85f9d4966739f6ae02af502de8efe6f9a8be2711f6148a233f6fef4c1fb3f3 This node is not in the Me
rkle Tree
Node: 2e2209e6e205e604bbfcc6cb24f3b28aeb1170eea282c747bled7a8678021a19 This node is in the Merkle
Tree
执行时间为: 1009.6637000096962 毫秒
>>>
```

运行速度 1.009s

CPU: AMD Ryzen 9 5900HX with Radeon Graphics 3.30 GHz