小组分工表

	姓名	学号	分工
组员 1	邵夕雅	202100460087	负责所有项目

已完成项目：

Project1：https://github.com/sxyyly/Group102/tree/main/Project1

Project2：https://github.com/sxyyly/Group102/tree/main/Project2

Project3：https://github.com/sxyyly/Group102/tree/main/Project3

Project4：https://github.com/sxyyly/Group102/tree/main/Project4

Project5：https://github.com/sxyyly/Group102/tree/main/Project5

Project8：https://github.com/sxyyly/Group102/tree/main/Project8

Project9：https://github.com/sxyyly/Group102/tree/main/Project9

Project10：https://github.com/sxyyly/Group102/tree/main/Project10

Project11：https://github.com/sxyyly/Group102/tree/main/Project11

Project12：https://github.com/sxyyly/Group102/tree/main/Project12

Project13：https://github.com/sxyyly/Group102/tree/main/Project13

Project14：https://github.com/sxyyly/Group102/tree/main/Project14

Project18：https://github.com/sxyyly/Group102/tree/main/Project18

Project19：https://github.com/sxyyly/Group102/tree/main/Project19

Project22：https://github.com/sxyyly/Group102/tree/main/Project22

汇总报告

邵夕雅

2023 年 7 月 13 日

目录

1 Project 1：implement the naïve birthday attack of reduced SM3

1.1 实现方式

1. 简化版 SM3 的实现首先定义简化版的 sm3 ( )，然后定义了填充函数 padding()、消息分块函数 divide_blocks()、压缩函数 compress()、消息扩展函数 expand ( )、循环左移函数 rotate_left()、置换函数等。

2. 寻找碰撞 generate_collisions() 函数用于生成碰撞。它随机生成消息，计算其 SM3 哈希值，将哈希值缩小到指定的位数，并检查是否存在碰撞。如果发现碰撞，它将打印碰撞的消息并返回它们。其中，使用字典的方法检测碰撞，该方法的好处是不需要提前创建一个很大的空间，而是每次都在已有的信息中比对碰撞，这样可以对更长的串进行攻击，其缺点是运行时间较慢，攻击较长的串会消耗大量时间。

num_bits 表示生成 hash 值的位数，原象空间大小为 2**num_bits/2。根据生日攻击，搜索 2**（num_bits/2）便有 1/2 的概率找到一对碰撞。因此在 generate_collisions() 中生成 2**（num_bits/2）个消息便有很大概率寻找到一对碰撞。

1.2 实现效果

```
消息 20246 和 28353 的哈希值发生碰撞
totally cost 0.06052041053771973 s
```

运行速度：0.061s

1.3 实验环境

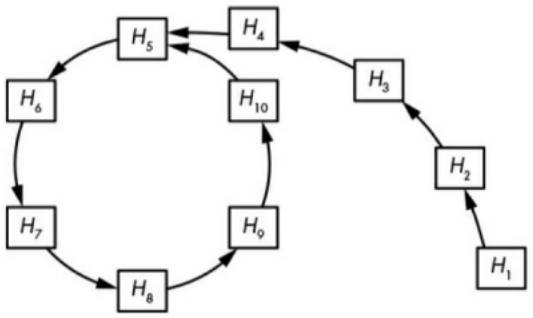
设备名称 DESKTOP-O3UGS4A
处理器 AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
机带 RAM 16.0 GB (13.9 GB 可用)
设备 ID D232D87C-8384-4711-AF60-22AEEE338FF3
产品 ID 00326-10000-00000-AA108
系统类型 64 位操作系统, 基于 x64 的处理器
笔和触控没有可用于此显示器的笔或触控输入

2 Project 2：implement the Rho method of reduced SM3

2.1 实验原理

第二原像攻击：即给定消息 M1 时，攻击者能够找到另一条消息 M2, 其哈希值与 M1 的哈希值相同。理论上复杂度为 O(2n)

Hash 函数的　方法是用来寻找散列碰撞而不需要大量的内存。它的想法是不断地对前一个哈希运算的结果进行哈希运算。当我们对一个起始值进行散列，然后对该散列进行散列，并重复这个过程，不会永远得到不同的数字。最终我们会得到一个重复的值，之后所有进一步的值都是之前值的重复，在一个循环中进行，如下图所示。Rho 攻击流程：



(1) 给定具有 n 比特哈希值的哈希函数,选择一些随机哈希值 H1, 设 H1'=H1

(2) 计算 H2=Hash(H1),H2'=Hash(Hash(H1'))

(3) 迭代该过程并计算 Hi+1=Hash(Hi)，Hi+1'=Hash(Hash(Hi'))，直到有一个 i 可以满足 Hi+1=Hi+1'

更进一步的寻找碰撞的技术的工作原理是：

首先检测循环的开始，然后查找碰撞。既不需要在内存中存储大量的值，也不需要对长的列表进行排序。 方法大约需要 2n/2 次操作才能成功。平均来说，循环和尾部（图中从 H1 延伸到 H5 的部分）各包括约 2n/2 个哈希值，其中 n 是哈希值的长度。因此，我们需要至少 2n/2+ 2n/2 次计算来发现一个碰撞。

2.2 实现方式

(1) 函数 rho_method 接受一个参数 exm，表示需要生成的 16 进制位数。将 exm 除以 4 得到位数的数量（每个十六进制位表示 4 位二进制数。

(2) 生成一个随机的十六进制数 x，位数为 exm，并计算其哈希值作为初始值 x_a。将初始值 x_a 进行一次哈希运算，得到新的值 x_b。

(3) 使用循环进行以下步骤，直到满足终止条件：检查 x_a 和 x_b 的前 num 位是否相等。

(4) 在每一轮中，将 x_a 更新为其哈希值，并将 x_b 更新为经过两次哈希运算的结果。

(5) 循环结束后，将 x_b 的值赋给 x_a，即 x_a = x_b。将 x_a 的哈希值和原始的十六进制值 x_a、x_b 作为结果返回。如果在循环中没有找到满足条件的结果，则返回 None 或者适当的错误处理。

2.3 实现效果

运行速度：2.345s

2.4 实验环境

设备名称 DESKTOP-O3UGS4A
处理器 AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
机带 RAM 16.0 GB (13.9 GB 可用)
设备 ID D232D87C-8384-4711-AF60-22AEEE338FF3
产品 ID 00326-10000-00000-AA108
系统类型 64 位操作系统, 基于 x64 的处理器
笔和触控没有可用于此显示器的笔或触控输入

3 Project3：implement length extension attack for SM3, SHA256, etc.

3.1 实验原理

长度扩展攻击含义：长度扩展攻击是一种针对哈希函数的攻击方法，长度扩展攻击通常利用了哈希函数在计算哈希值时未完全使用消息的长度作为输入。攻击者可以在已知消息哈希值以及原始消息长度的情况下，伪造一个扩展消息，该扩展消息由与原始消息等长的伪造消息、填充和额外的数据组成，该消息的哈希值与原始消息的哈希值相等。

注意：因为 SM3 是一个针对长度扩展攻击具有强安全性的哈希函数。因此我们对简化版的 SM3 展开长度扩展攻击。

长度扩展攻击原理：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 padding。

在 SM3 函数计算时，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 (初始值已经确定)，下一次用新向量去加密下一组，以此类推。

在对 secret + padding + m' 进行加密时，secret + padding 会被划分为一组（无需填充），m' 会划分为另一组。为了使 secret + padding + m' 的加密结果和 secret' + padding + m' 相等，只需要得到 secret + padding 加密后的向量值，然后用来更新哈希函数来加密 m'。这样加密得到的结果便是相同的。

3.2 实现方式

(1) 随机生成一个消息 (secret)，用 SM3 函数算出 hash 值 (hash1)。

(2) 生成与 secret 等长的新消息 secret'，并进行消息扩展。

(3) 将上述消息与附加消息 m' 级联，并计算 hash 值 (hash2)。

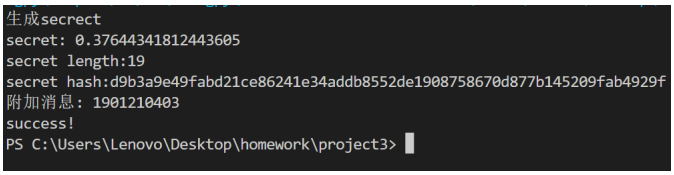
(4) 计算 secret + padding + m' 的 hash 值 (hash3)，如果 hash2 和 hash3 相等，攻击成功。

实验细节：(1) 随机生成了一个浮点数作为 secret，并计算得到了 hash 值。要得到第一次加密之后 8 个向量的值，只需要将 hash 值按 8 字节分组，并把每组的值转换成 int 类型。

(2) 得到了向量值后，便可以开始构造消息。由于我们不需要知道 secret 的值，只知道 secret 的长度，所以 secret 部分可以用等长的任意字符代替 (我这里用的是' a')。随后进行 padding，得到 64 字节的消息，再将附加信息放在后面，消息就构造完成了。

(3) 接着进行加密，由于此时只需要对附加的消息进行加密，所以我修改了一下 sm3 的函数实现，增加了一个 new_v 参数，表示更新之后的向量值。此外这次加密的次数要比之前少一次，从消息的第 64 字节开始加密，即可得到 hash 值。

3.3 实现效果



运行速度：0.0052s

3.4 实验环境

设备名称 DESKTOP-O3UGS4A
处理器 AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
机带 RAM 16.0 GB (13.9 GB 可用)
设备 ID D232D87C-8384-4711-AF60-22AEEE338FF3
产品 ID 00326-10000-00000-AA108
系统类型 64 位操作系统, 基于 x64 的处理器
笔和触控没有可用于此显示器的笔或触控输入

4 Project4:do your best to optimize SM3 implementation (software)

4.1 优化过程

(1) 使用字节操作：将 divide_blocks() 函数中的字节拆分操作修改为使用 np.frombuffer() 函数将字节数据转换为 NumPy 数组，以利用 NumPy 的向量化操作。

```
1.  ····num_blocks·=·len(message)·*·8·//·512
2.  ····blocks·=·np.frombuffer(message,·dtype=np.uint32).reshape(num_blocks,·16).copy()
```

(2) 向量化操作：通过使用 numba 库的 @nb.njit(parallel=True) 装饰器将 compress() 函数进行并行化编译，以提高计算速度。

```
@nb.njit(parallel=True)
def compress(hash_value, blocks):
```

(3) 并行计算：使用 parallel=True 参数并结合 nb.prange() 函数，将循环中的迭代操作进行并行计算，以加速哈希计算。

(4) 缓存优化：对于使用 NumPy 数组进行计算的部分，利用 NumPy 的数据缓存机制，减少内存访问和缓存未命中。

(5) 据规模的不同而有所差异。此外，代码中的优化措施可能并不全面，根据实际需求和环境，还可以进行更多的优化操作。在进行优化时，请进行适当的测试和验证，以确保优化的有效性和正确性。

4.2 运行结果

4.3 实现效果

对于加密 512bit 明文，实现效果如下：

	sm3	new_sm3
time	1.24s	0.07s

4.4 实验环境

设备名称 DESKTOP-O3UGS4A
处理器 AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
机带 RAM 16.0 GB (13.9 GB 可用)
设备 ID D232D87C-8384-4711-AF60-22AEEE338FF3
产品 ID 00326-10000-00000-AA108
系统类型 64 位操作系统, 基于 x64 的处理器
笔和触控没有可用于此显示器的笔或触控输入

5 Project5:Impl Merkle Tree following RFC6962

5.1 基本概念

Hash list

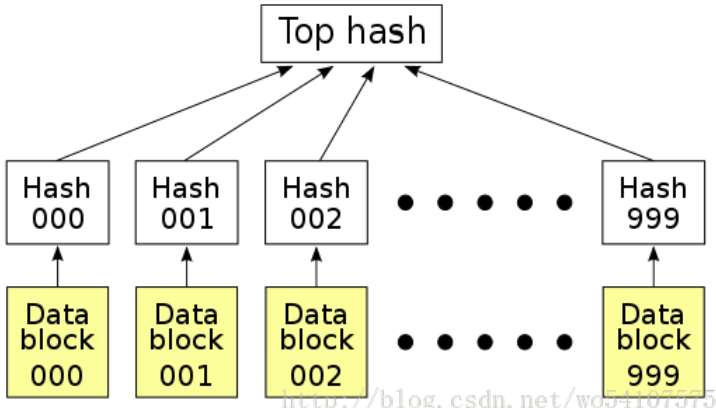
```
for block in nb.prange(blocks.shape[0]):
    w = expand(blocks[block])
    v = hash_value.copy()

    for i in range(64):
        if i < 16:
            ss1 = (rotate_left(v[0], 12) + v[4] + rotate_left(T[0], i)) & 0xFFFFFFFF
        else:
            ss1 = (rotate_left(v[0], 12) + v[4] + rotate_left(T[1], i)) & 0xFFFFFFFF
        ss1 = rotate_left(ss1, 7) & 0xFFFFFFFF
        ss2 = ss1 ^ rotate_left(v[0], 12) & 0xFFFFFFFF
        tt1 = (ff(v[0], v[1], v[2], i) + v[3] + ss2 + w[i]) & 0xFFFFFFFF
        tt2 = (gg(v[4], v[5], v[6], i) + v[7] + ss1 + w[i]) & 0xFFFFFFFF
        v[3] = v[2]
        v[2] = rotate_left(v[1], 9) & 0xFFFFFFFF
        v[1] = v[0]
        v[0] = tt1
        v[7] = v[6]
        v[6] = rotate_left(v[5], 19) & 0xFFFFFFFF
        v[5] = v[4]
        v[4] = p0(tt2)

    for i in range(8):
        hash_value[i] ^= v[i]
```

```
sm3加密后得到的密文为：
ccbd1b23934a785d7bb16b9fbbb93742379de2057ff0edaed250cf053969c75a
sm3运行时间： 1.2462151050567627
new_sm3加密后得到的密文为：
ccbd1b23934a785d7bb16b9fbbb93742379de2057ff0edaed250cf053969c75a
new_sm3运行时间： 0.07395458221435547
```

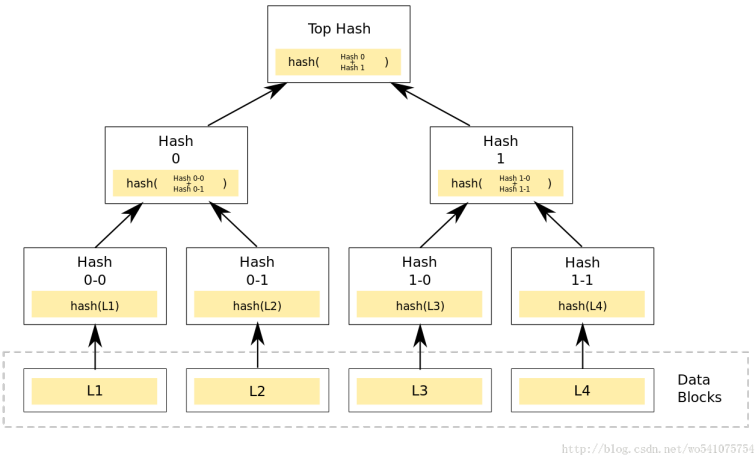
在下载数据时，为了校验数据的完整性，更好的办法是把大的文件分割成小的数据块。为每个数据块做 Hash，所有的 Hash 构成 Hash list。下载真正数据之前，先下载一个 Hash 列表。为确保 Hash list 的正确性，把每个小块数据的 Hash 值拼到一起，然后对这个长字符串在作一次 Hash 运算，这样就得到 Hash 列表的根 Hash(Top Hash or Root Hash)。下载数据的时候，首先从可信的数据源得到正确的根 Hash，就可以用它来校验 Hash 列表了，然后通过校验后的 Hash 列表校验数据块。



Merkle Tree

Merkle Tree 可以看做 Hash List 的泛化（Hash List 可以看作一种特殊的 Merkle Tree，即树高为 2 的多叉 Merkle Tree）。在最底层，和哈希列表一样，我们把数据分成小的数据块，有相应地哈希和它对应。但是往上走，并不是直接去运算根哈希，而是把相邻的两个哈希合并成一个字符串，然后运算这个字符串的哈希，这样每两个哈希就结婚生子，得到了一个”子哈希“。如果最底层的哈希总数是单数，那到最后必然出现一个单身哈希，这种情况就直接对它进行哈希运算，所以也能得到它的子哈希。于是往上推，依然是一样的方式，可以得

到数目更少的新一级哈希，最终必然形成一棵倒挂的树，到了树根的这个位置，这一代就剩下一个根哈希了，我们把它叫做 Merkle Root。如图所示。



Merkle Tree 的特点

(1) MT 是一种树，大多数是二叉树，也可以多叉树，无论是几叉树，它都具有树结构的所有特点。

(2) Merkle Tree 的叶子节点的 value 是数据集合的单元数据或者单元数据 HASH。

(3) 非叶子节点的 value 是根据它下面所有的叶子节点值，然后按照 Hash 算法计算而得出的。

5.2 实现方式

5.2.1 前期准备

代码定义了一个名为 Block 的结构体，表示 Merkle 树中的节点。结构体 Block 包含以下成员变量：

int data：用于存储节点的数据。

Block* prev：指向前一个节点的指针。

Block* leftChild：指向左子节点的指针。

Block* rightChild：指向右子节点的指针。

Block* bro：指向兄弟节点的指针。

结构体 Block 还定义了两个构造函数：

(1) 默认构造函数 Block()：用于创建一个空的节点，将所有指针成员初始化为 NULL。

(2) 参数化构造函数 Block(int data)：用于创建一个具有指定数据的节点，同时将所有指针成员初始化为 NULL。主要函数 (1)randomHexString( ) 用于生成一个长度为 16 的随机字符串（由小写字母组成）

(2)heightOfTree( ) 计算给定叶子节点数的 Merkle 树的高度）

5.3 构建过程

(1) 创建一个名为 leafBlocks 的 Block 数组，大小为 MAX_LEAF_SIZE，用于存储叶子结点。

(2) 创建一个指针数组 Block **nodeList,其大小为树的高度 height。nodeList[i][j] 表示第 i 层第 j 个结点。

(3) 使用两层嵌套循环,遍历所有层的所有结点。外层循环迭代从 0 到 height - 1，内层循环迭代从 0 到 num - 1。在每一次迭代中根据 Merkle 树的高度和节点之间的关系进行节点数据的哈希计算和指针设置：

当结点是最底层时，执行以下操作：

设置 num 表示第 i 层的结点数。

如果结点数为奇数且为当前为最右侧的结点 (j==num-1) 时，则当前叶子节点的数据为哈希函数应用于两个相同数据的哈希值。否则，将当前节点的数据设置为哈希函数应用于两个不同数据的哈希值。

此外，设置第 j 个和第 j-1 个叶子结点的 prev 指针指向当前节点。设置叶子节点的 bro 指针指向兄弟节点。设置当前节点的左子节点指针指向左侧叶子节点，当前节点的右子节点指针指向右侧叶子节点。

如果 i 不是 0，执行以下操作：

同上确定结点数据值。

设置当前节点的左子节点指针指向上一层的左侧节点。设置当前节点的右子节点指针指向上一层的右侧节点。设置上一层的左侧节点的 prev 指针指向当前节点。设置上一层的右侧节点的 prev 指针指向当前节点。设置上一层的左侧节点的 bro 指针指向上一层的右侧节点。设置上一层的右侧节点的 bro 指针指向上一层的左侧节点。

(4)Merkle Tree 验证

细节：heightOfTree( ) 计算 Merkle Tree 的高度不包括叶子结点那一层。

5.4 实现效果

运行速度：3ms

5.5 实验环境

设备名称 DESKTOP-O3UGS4A
处理器 AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
机带 RAM 16.0 GB (13.9 GB 可用)
设备 ID D232D87C-8384-4711-AF60-22AEEE338FF3
产品 ID 00326-10000-00000-AA108
系统类型 64 位操作系统, 基于 x64 的处理器
笔和触控没有可用于此显示器的笔或触控输入

6 Project8: AES impl with ARM instruction

6.1 算法实现

AES 加密过程是在一个 4×4 的字节矩阵上运作，这个矩阵又称为“体（state）”，其初值就是一个明文区块（矩阵中一个元素大小就是明文区块中的一个 Byte）。（Rijndael 加密法因支持更大的区块，其矩阵行数可视情况增加）加密时，各轮 AES 加密循环（除最后一轮外）均包含 4 个步骤：

(1)AddRoundKey—矩阵中的每一个字节都与该次回合密钥（round key）做 XOR 运算；每个子密钥由密钥生成方案产生。

(2)SubBytes—通过一个非线性的替换函数，用查找表的方式把每个字节替换成对应的字节。

(3)ShiftRows—将矩阵中的每个横列进行循环式移位。

(4)MixColumns—为了充分混合矩阵中各个直行的操作。这个步骤使用线性转换来混合每内联的四个字节。最后一个加密循环中省略 MixColumns 步骤，而以另一个 AddRoundKey 替换。

6.2 文件说明

AES.cpp 是 AES 的 C 语言实现算法（以 10 轮 AES 为例）
AES_ARM.txt 是 AES.cpp 的 ARM 汇编指令

6.3 实验环境

设备名称 DESKTOP-O3UGS4A
处理器 AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
机带 RAM 16.0 GB (13.9 GB 可用)
设备 ID D232D87C-8384-4711-AF60-22AEEE338FF3
产品 ID 00326-10000-00000-AA108
系统类型 64 位操作系统, 基于 x64 的处理器
笔和触控没有可用于此显示器的笔或触控输入

7 Project9: AES / SM4 software implementation

算法步骤入 project8 所示。

7.1 实现方式

字节代替变换：
(1) 首先将 S 盒存入数组；
(2) 将输入明文的每一位转为 8 位二进制数，并用 x 表示高四位的值（即 S 盒行标），y 表示低四位的值（即 S 盒列值）；
(3) 以 x,y 作为索引从 S 盒的对应位置取出元素作为输出，放入 cs 列表中。cs 作为明文字节代替变换的结果被返回。
行移位变换：
行移位变换的基本规则是第一行保持不变，第二行循环左移一位，第三行循环左移两位，第四行循环左移三位。数据移位也可以理解为数据原始索引移位。
(1) 定义一个列表 index，表示行变化后每个元素的位置；

(2) 对于输入 c 的每一行，索引来自行变化每个元素的位置，并索引到的元素放入 cs 列表中；

(3) 返回的 cs 列表即行移位变换后的输出。

列混淆变换：

(1) 首先定义 8-bit 二进制数的乘积函数：

8-bit 二进制数与 0x01 相乘的结果就是本身。

8-bit 二进制数 c 与 0x02 相乘，先判断 c 最高位是否为 1。如果是 c[0]==1, 则取 c 的低 7 位，最后一位补零，得到的结果与 00011011 异或。如果 c[0]==0，则取 c 的低 7 位，最后一位补零便是最终结果。

8-bit 二进制数 c 与 0x03 相乘，可以转换为 c 先与 0x02 相乘，得到的结果再与 c 异或，便是最终结果。

(2) 对于输入的每一列，分别左乘矩阵中的每一行，然后将得到的结果放入 cs 列表的对应列对应行中；

(3) 得到的 cs 矩阵即为列混淆变换后的结果。

注意：输入为 c，c[i] 表示输入状态的第 i 列，输出为 cs，cs[i], 表示输出状态的第 i 列。

轮密钥加变换：

轮密钥加操作可以看成是状态一列的四个字节与与轮密钥的一个字进行列间操作。

(1) 首先将第 time 轮的轮密钥取出，放入到 key 中；

(2) 将 c 与 key 按比特异或得出的结果放到 c_new 中，c_new 便是轮密钥加的输出结果。

7.2 实现效果

```
202100460087加密结果是: [['168', '138', '108', '123'], ['26', '115', '114', '72'], ['58', '143', '26', '167'], ['175', '223', '100', '143']]
运行时间为:  0.01432180404663086 s
PS C:\Users\Lenovo\Desktop\homework\project9>
```

运行时间：0.01432180404663086s

7.3 实验环境

设备名称 DESKTOP-O3UGS4A

处理器 AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz

机带 RAM 16.0 GB (13.9 GB 可用)

设备 ID D232D87C-8384-4711-AF60-22AEEE338FF3

产品 ID 00326-10000-00000-AA108
系统类型 64 位操作系统, 基于 x64 的处理器
笔和触控没有可用于此显示器的笔或触控输入

8 Project10: report on the application of this deduce technique in Ethereum with ECDSA

在以太坊中，ECDSA 算法可以用于从签名中推导出公钥。这个功能在以太坊中的身份验证和签名验证过程中非常重要。当一个交易被广播到以太坊网络时，它必须被验证，以确保它是由合法的发送者发送的。ECDSA 算法可以用于验证交易的签名，并从签名中推导出发送者的公钥，从而验证发送者的身份。

1 推导技术概述

推导技术是一种利用已知信息和算法进行推理的方法，可用于从相关数据中推断出隐藏的信息或密钥。

2 ECDSA 算法描述

在以太坊（Ethereum）网络中，ECDSA（椭圆曲线数字签名算法）是一种常用的签名算法，用于验证交易和身份认证。

具体算法如下：

签名过程

(1) 选择一条椭圆曲线 Ep(a,b)，和基点 G；

(2) 选择私有密钥 k（k<n，n 为 G 的阶），利用基点 G 计算公开密钥 K=kG；

(3) 产生一个随机整数 r（r<n），计算点 R=rG；

(4) 将原数据和点 R 的坐标值 x,y 作为参数，计算 SHA1 做为 hash，即 Hash=SHA1(原数据,x,y)；

(5) 计算 s r - Hash * k (mod n)

(6) r 和 s 做为签名值，如果 r 和 s 其中一个为 0，重新从第 3 步开始执行

验证过程

接受方在收到消息 (m) 和签名值 (r,s) 后，进行以下运算:

(1) 计算：sG+H(m)P=(x1,y1), r1 x1 mod p。

(2) 验证等式：r1 r mod p。

(3) 如果等式成立，接受签名，否则签名无效。

(4) 从签名中恢复公钥

这个算法的思路可以简要概括如下：

(1) 使用签名中的 r 值，通过取模运算得到 x 坐标：x = r

(2) 根据椭圆曲线方程 y^2 = x^3 + ax + b，计算 y 坐标。

(3) 将消息 m 进行哈希运算，得到哈希值 e。

(4) 构造两个椭圆曲线点 P1 和 P2，分别为 (x, y) 和 (x, p-y)。利用签名中的 s 值和对应的点 P1 或 P2，计算临时私钥 sk1 或 sk2。

(5) 计算临时点 tmp，为消息哈希值 e 乘以椭圆曲线基点 G。计算 tmp_i，为临时点 tmp 的 y 轴取负。

(6) 计算临时点 tmp_1，为临时私钥 sk1 或 sk2 与 tmp_i 的加法。

(7) 使用临时私钥和 gcd(r, n) 乘法运算，计算出推导的公钥 pk1 或 pk2。

算法可行性验证

```
==========
公钥是 (87066217303934462390729999955761363776392898798643509190668330748562221973116, 735735309159154099630931613228474725011255069934408748259280702257871805636 05)
m是:sdueducn
从签名恢复公钥
(47144453094230725863561477180600593286062760550648075718147774669825751157564, 100428865485808453229861563994879380008412204968731707522792436628183420356668)
(87066217303934462390729999955761363776392898798643509190668330748562221973116, 735735309159154099630931613228474725011255069934408748259280702257871805636 05)
>>
```

代码主要函数

def gcd: 求 gcd

def Euc: 扩展欧几里得求模逆

def add: 椭圆曲线加法

def mul: 椭圆曲线乘法

def isQR: 判断二次剩余

def QR: 求二次剩余

def keygen: 密钥生成

def Signiture_sk: 私钥签名

def Deduce_signiture: 恢复密钥

参考文献

[1]https://github.com/orthokikanium/Innovation-and-Entrepreneurship-Practice/blob/

[2]https://learnblockchain.cn/article/5012

[3] 课程 sm2PPT

9 Project11: impl sm2 with RFC6979

9.1 基本概念

SM2 算法简介:

SM2 算法是基于椭圆曲线的非对称算法，相对于 RSA 算法，SM2 具有密钥更小，运算速度更快，相同密钥长度下具有更高安全性等优势。

SM2 密钥结构定义:

私钥：d 符合要求的 32byte(256bit) 随机数

公钥：(x,y), 实际一个坐标点，依据私钥 d 计算所得

SM2 加密数据:

SM2 加密数据使用公钥 (x,y) 进行加密，加密结果为 c1c3c2 部分算法中定义为 c1c2c3, 下面介绍密文中各个结构实际含义：

c1: 随机数 K 与 G(x,y) 的多倍点运算结果,结果也是一个点,记录为 (kx,ky)

c2: 实际密文值

c3：使用 SM3 对于 kx||data||ky 的 hash 值，在解密时校验解密结果是否正确

c1 的运算逻辑:

c1 = k • G(x,y)(k 是一个随机数，k 的取值范围为 [1,n-1])

c2 计算逻辑:

c2 = 密钥流（异或）data

密钥流的计算逻辑如下：

(1) 复用计算 c1 时产生的随机数 k。

(2) 计算 kpk(x,y), 得到 kpx，kpy。

(3) 根据 data 的长度与 kpx,kpy 生成与 data 等长的密钥流，用于 c2 的最终计算。

整体密钥流计算采用 KDF 方法计算，KDF 可以理解成根据输入的因子，产生期望长度的数据流，目前直流的 KDF 计算采用 HASH 方法。SM2 中的 KDF 使用的是 SM3 摘要。

c3 计算逻辑:

c3 = HASH(kpx,data,kpy)

在解密时需要在解密出原文后计算 HASH 值做最后确认，确认一致后认定解密成功，不一致则解密失败。

SM2 解密数据:

SM2 的解密流程实际是根据 c1 计算出加密时使用的密钥流，使用密文数据与密钥流进行异或得到数据明文，后续在确认计算出的摘要值与密文中 c3 是

否一致。

(1) 密钥流计算逻辑

1. 从密文中分离出 c1(x,y)。

2. 使用私钥 d 与 c1 进行多倍点运算，得到计算结果 (cx,cy)，d • c1(cx,cy)。

3. 根据 c2 长度计算密钥流，KDF(cipherlen,cx,cy)。

(2) 解密运算

上面我们得到了密钥流的中间结果，我们根据密钥流来计算明文：

原文 = 密钥流 (异或) 密文

(3) 解密校验

接下来我们计算解密后的摘要值与 c3 是否相等，计算流程如下：

1. 基于以上的运算结果的 cx,cy 与得到的原文。

2. SM3(cx, 计算得到的明文,cy)。

3. 对比计算结果与 c1，一致解密成功，不一致则解密失败。

9.2 实现方式

9.2.1 前期准备

创建 SM2KeyPair，用于生成一个密钥对对象。提供验证公钥有效性的功能。

创建一个 User 类，将用户 ID 与密钥对对象绑定。提供用于计算用户的标识 ID 相关的哈希值。获取用户的公钥 P，并使用 SM3 哈希算法对一系列数据进行哈希计算等功能。

创建 ECPoint 类，用于创建一个点对象。提供判断点是否为单位元，判断点是否在椭圆曲线上有效，获取点的 x,y 坐标，点相加，点相乘，点取反等功能。

创建 ECCurve 类，用于创建一个椭圆曲线对象。提供用于比较两个椭圆曲线对象是否相等，用于创建一个椭圆曲线上的点对象，于验证椭圆曲线的有效性，实现点的倍乘运算中的双倍运算，实现点的加法运算，实现点的倍乘运算等。

9.2.2 实现过程

(1) 选取私钥和公钥

(2) create_point( ) 利用 ECPoint 创建了一个表示公钥的点 P。

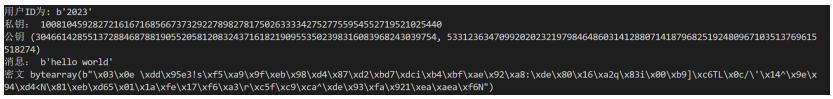
(3) 利用 SM2KeyPair(d, P) 创建一个密钥对对象。

(4) 利用 User(ID, SM2KeyPair(d, P)) 将用户 ID 与密钥对对象绑定。

(5) 计算 C1:

curve.get_g() 通过 ECPoint 返回曲线的生成点 G，然后计算 C1 = G* k。
(6) 计算 C2：
1 计算曲线参数和点 PB 的乘积，并将结果赋值给变量 S。
2 检查 S 是否为单位元，如果是则引发异常。
3 计算点 PB 乘以随机数 k 的结果，并将结果赋值给变量 kPB。
4 将 kPB 的 x 坐标和 y 坐标转换为字节数组的表示形式。
5 执行密钥派生函数（KDF），生成指定长度的密钥或伪随机数。
6 检查派生密钥是否全为零，如果不是则中断循环。
(7) 计算 C3:
利用 SM3 的 hash() 函数计算哈希值。

9.3 实现效果



运行速度：0.012001752853393555s

9.4 代码说明

数学计算相关函数：MATH.py
哈希：SM3.py
SM2 相关参数：ECC.py SM2.py
密钥相关函数：SM2KeyPair.py
SM2 加密：SM2GenEnc.py

9.5 实验环境

设备名称 DESKTOP-O3UGS4A
处理器 AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
机带 RAM 16.0 GB (13.9 GB 可用)
设备 ID D232D87C-8384-4711-AF60-22AEEE338FF3
产品 ID 00326-10000-00000-AA108
系统类型 64 位操作系统, 基于 x64 的处理器
笔和触控没有可用于此显示器的笔或触控输入

9.6 参考文献

[1](142 条消息) SM2 算法加密与解密过程 _sm2 加密 _ 清涵的博客-CSDN 博客

10 Project12: verify the above pitfalls with proof-of-concept code

10.1 存在问题

(1) 泄漏 k，可计算出私钥 d。

(2)k 相同，可计算出私钥 d。

(3)k 重用，可计算出私钥 d。

10.2 实现方式

SM2 签名算法：ECDSA_sig(m, n, G, d,k)

(1) 函数首先计算消息的哈希值 e = hash(m)。

(2) 使用生成点 G 和随机数 k 执行点的倍乘运算得到曲线上的一个点 R。

(3) 提取点 R 的 x 坐标，并计算 r = x % n。

(4) 函数计算签名参数 s=(Euc(k, n) * (e + d * r)) % n，其中 Euc(k, n) 表示随机数 k 的模 n 的逆元。

(5) 函数返回计算得到的签名值 (r, s)。

SM2 验证算法：ECDSA_ver(m, n, G, r, s, P)

(1) 计算消息的哈希值 e = hash(m)。

(2) 计算 s 模 n 的逆元 w = Euc(s, n)。

(3) 计算验证参数 v1 = (e * w)% n , v2 = (r * w) % n。

(4) 函数使用倍乘运算 mul(v1, G) 和 mul(v2, P) 分别得到曲线上的两个点，并将它们相加，得到新的点 w。

(5) 如果点 w 为零点（无穷远点），验证失败。否则，如果点 w 的 x 坐标模 n 的结果等于签名的 r 值，则验证通过；否则验证不通过。

验证潜在问题 (1):Leaking_K_of_Leaking_d(r,n,k,s,m)

计算了 r 对 n 的逆元 r_reverse，已知 e = hash(m)、签名参数 k、s 和计算的逆元 r_reverse ，则 d 可由 r_reverse * (k*s-e)

验证潜在问题 (2):Using_Same_K(s1,m1,s2,m2,r,d1,d2,n)

已知两个消息 m1,m2 的 hash 值 e1,e2，则可根据

d2_1 = ((s2 * e1 - s1 * e2 + s2 * r * d1) * Euc(s1 * r, n))

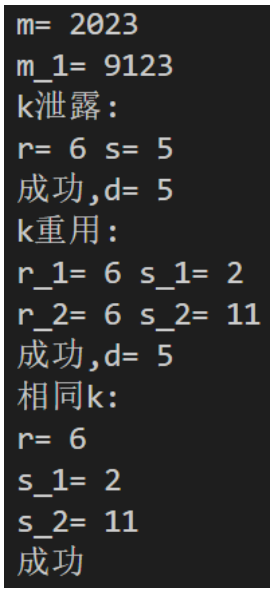
d1_1 = ((s1 * e2 - s2 * e1 + s1 * r * d2) * Euc(s2 * r, n))

计算出两次签名的私钥 d_1,d2_1。

验证潜在问题 (3)：Reuse_K_of_Leaking_d(r1,s1,m1,r2,s2,m2,n)

已知两个消息 m1,m2 的 hash 值 e1,e2，则可根据 d=((s1 * e2 - s2 * e1) * Euc((s2 * r1 - s1 * r1), n))

10.3 实现效果



10.4 实验环境

设备名称 DESKTOP-O3UGS4A

处理器 AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz

机带 RAM 16.0 GB (13.9 GB 可用)

设备 ID D232D87C-8384-4711-AF60-22AEEE338FF3

产品 ID 00326-10000-00000-AA108

系统类型 64 位操作系统, 基于 x64 的处理器

笔和触控没有可用于此显示器的笔或触控输入

10.5 参考文献

[1]Innovation-and-Entrepreneurship-Practice/SM2_project/SM2_pit/pitfalls_attack_sm2.py at main · Hyan1ce/Innovation-and-Entrepreneurship-Practice · GitHub

11 Project13: Implement the above ECMH scheme

11.1 实现方式

11.1.1 主要函数

(1)ext_gcd(a,b)。扩展欧几里得算法，即 a mod b 的逆元，返回的第一个值为结果。

(2) sameAdd(xl,y1)。计算椭圆曲线上 2*(x1,y1)。

(3) NsameA dd(x1,y1,x_G,y_G)。返回 (x3,y3)=(x1,y1)+(x_G,y_G)。

(4) multiply(x1,y1,k,x_G,y_G)。调用函数 (2)(3)，使用类似快速模指数算法，计算 (x1,y1)=k(x_G,y_G)。

(5) Point_bit(x1,y1)。选用未压缩格式，将 (x1,y1) 化为比特串。

(6) SHA256(str)。利用 hashlib.sha256() 将 str 加密。

(7) KDF(x,klen)。调用 SHA256 函数进行密钥扩展。

(8) xor(bit_M,t)。进行逐比特异或。

(9) bit_Point(hexC1)。将 16 进制字符串转为点坐标。

(10) Point_Cl_in(Point_C1)。判断这个点是否在方程上。

11.2 实现过程

加密过程

(1) 利用 multiply() 计算椭圆曲线点 C1=[k]G=(x1,y1)，并利用 Point_bit(x1,y1) 将 C1 数据类型转换为二进制字符串。

(2) 由于所给公钥一定满足条件，因此未验证。

(3) 利用 multiply( ) 计算椭圆曲线点 [k]PB=(x2,y2)。

(4) 将消息 M 转成二进制并进行位填充生成 bit_M，对 x2,y2 转成二进制并填充至 256 位。

(5) 调用密钥派生函数（KDF）来生成长度与消息 bit_M 相同的位串 t。

(6) 利用 xor( ) 计算 C2 = M t。

(7) 利用 SHA256( ) 计算 C3=Hash(x2||M||y2)。

(8) 将 C1 和 C2 转成 16 进制字符串并填充至 130 位。

(9) 利用公式 C = hexC1 + hexC2 + C3[1] 生成密文。

解密过程

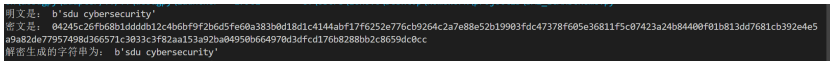
(1) 利用 bit_Point( ) 将 C1 转化为椭圆曲线上的点并判断其是否在椭圆曲线上。

(2) 私钥与 C1 进行多倍点运算，得到计算结果 (x2,y2)，并将 x2 和 y2 的值转换为二进制字符串，并填充到长度为 256 位。

(3) 使用位串 x2 和 y2 作为输入，调用密钥派生函数（KDF）来生成与密文 C2 长度相同的位串 t。对密文 C2 和生成的位串 t 进行异或运算，得到解密后的结果 M。

(4) 遍历 M 中的位串，从第一个出现的"1" 开始提取位串中的有效部分，得到消息的二进制表示 MM_ming_bin。MM_ming_bin 转换为字节串：将 MM_ming_bin 转换为整数，然后转换为十六进制字符串，并最后将其转换为字节串 MM_ming。

11.3 实现效果



加密运行时间:0.1393899917602539s

解密运行时间：0.0687577244567871s

11.4 实验环境

设备名称 DESKTOP-O3UGS4A

处理器 AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz

机带 RAM 16.0 GB (13.9 GB 可用)

设备 ID D232D87C-8384-4711-AF60-22AEEE338FF3

产品 ID 00326-10000-00000-AA108

系统类型 64 位操作系统, 基于 x64 的处理器

笔和触控没有可用于此显示器的笔或触控输入

11.5 参考文献

[1](142 条消息) SM2 椭圆曲线 _zevsee 的博客-CSDN 博客

12 Project14: Implement a PGP scheme with SM2

12.1 基本概念

PGP 定义:Pretty Good Privacy（PGP）是一个加密程序，为数据通信提供加密隐私和身份验证。PGP 用于对文本、电子邮件、文件、目录和整个磁盘分区进行签名、加密和解密，并提高电子邮件通信的安全性。PGP 加密使用散列，数据压缩，对称密钥加密，最后是公钥加密的串行组合。其中最关键的是两种形式的加密的组合：对称密钥加密 (Symmetric Cryptography) 和非对称密钥加密。

12.2 实现方式

本次实验旨在实现一个简易 PGP，调用 GMSSL 库中封装好的 SM2 加解密函数。加密时使用对称加密算法 AES 加密消息，非对称加密算法 SM2 加密会话密钥；解密时先使用 SM2 解密求得会话密钥，再通过 AES 和会话密钥求解原消息。

PGP_Encrypt( ) 的实现

(1) PGP_Encrypt 函数接受两个参数：mes（消息）和 k（密钥）。加密是在 OFB 模式下执行 AES 加密。初始化向量 iv 被设置为 b'0000000000000000'。

(2) 使用 Crypto.Cipher 模块的 AES.new 函数创建了 cryptor 对象。它以 k 密钥作为输入（编码为 UTF-8），以及 mode 和 iv 参数。

(3) 如果消息的长度不能被块大小（16）整除，add 变量被设置为块大小减去消息中剩余的字符数。然后，消息 mes 被使用空字符填充，以确保其长度是块大小的倍数。

(4) 变量 ciphertext1 存储使用 cryptor.encrypt 方法对填充后的消息进行加密的结果。在加密之前，消息被编码为 UTF-8。

(5) 变量 plaintext_bytes 存储密钥 k 的 UTF-8 编码版本。

(6) 变量 ciphertext2 被赋值为 SM2_enc 函数的结果，该函数使用 SM2 算法加密 plaintext_bytes。

(7) 最后，函数打印出 ciphertext1 和 ciphertext2 的值，并将它们作为元组返回。

PGP_Decrypt( ) 实现

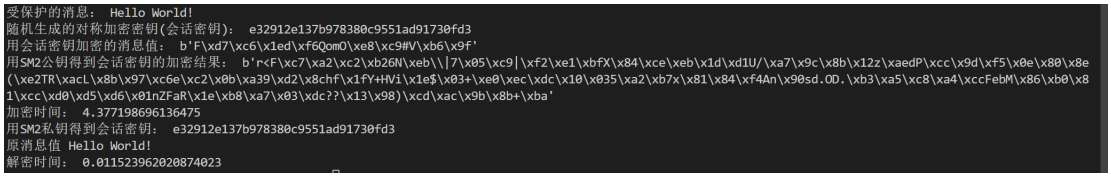
(1) 调用 SM2_dec 函数使用 mes2 参数作为输入，通过 SM2 算法使用私钥解密获得会话密钥。

(2) 使用获取到的会话密钥、AES 工作模式和初始向量创建 AES 密码器（AES.new(get_key, mode, iv)）。

(3) 调用密码器的 decrypt 方法对 mes1 进行解密，得到明文。

(4) 打印输出解密后的原始消息值。

12.3 实现效果



加密时间：4.377s 解密时间：0.011s

12.4 实验环境

设备名称 DESKTOP-O3UGS4A
处理器 AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
机带 RAM 16.0 GB (13.9 GB 可用)
设备 ID D232D87C-8384-4711-AF60-22AEEE338FF3
产品 ID 00326-10000-00000-AA108
系统类型 64 位操作系统, 基于 x64 的处理器
笔和触控没有可用于此显示器的笔或触控输入

12.5 参考文献

[1] PGP 工作原理详解 - 知乎 (zhihu.com)

13 Project18: send a tx on Bitcoin testnet, and parse the tx data down to every bit, better write script yourself

13.1 实现方式

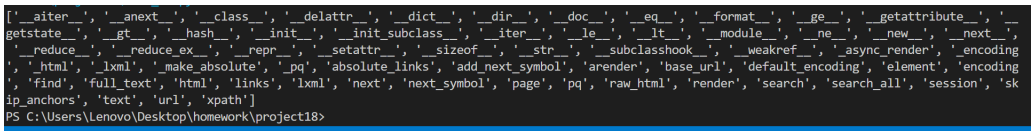
代码片段使用了 requests_html 库，它提供了一种方便的方法来与网站交互并解析 HTML 内容。

首先从 requests_html 库导入 HTMLSession 类，然后将其实例化之后，调用其 get 方法，发送请求，并将响应存储在变量 r 中。

要访问响应的 HTML 内容，使用 Response 对象的 html 属性。html 属性提供了一组用于与解析后的 HTML 交互的方法和属性。

通过 dir 函数查阅解析后的内容。

13.2 实现效果



进行时间：1.6379799842834473s

13.3 运行指导

使用 pip install requests-htmlrequests-html 模块安装，安装后直接运行。

13.4 实验环境

设备名称 DESKTOP-O3UGS4A
处理器 AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
机带 RAM 16.0 GB (13.9 GB 可用)
设备 ID D232D87C-8384-4711-AF60-22AEEE338FF3
产品 ID 00326-10000-00000-AA108
系统类型 64 位操作系统, 基于 x64 的处理器
笔和触控没有可用于此显示器的笔或触控输入

13.5 参考文献

[1]https://blog.csdn.net

14 Project19: forge a signature to pretend that you are Satoshi

14.1 实验思路

概率性算法，通过 random.randrange(1, n - 1) 随机选择 a、b，计算签名 r1、s1。

a= random.randrange(1, n - 1)

b = random.randrange(1, n - 1)

r1 = add(mul(a, G), mul(b, P))[0]

e1 = (r1 * a * Euc(b, n)) % n

s1 = (r1 * Euc(b, n)) % n

14.2 实现方式

(1) 函数开始时计算 s 模 n 的乘法逆元 w。

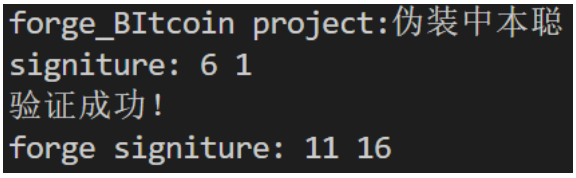
(2) 然后，计算了两个值 v1 和 v2。v1 = (e1 * w) % n ;v2 = (r1 * w) % n。

(3) 接下来，通过使用点 G 和点 P 进行加法和乘法运算，计算了一个新的点 w。

(4) 如果计算得到的点 w 的值为 0，那么它返回，表示验证失败。

(5) 否则，它检查计算得到的点 w 的 x 坐标是否与原始的签名值 r1 对 n 取模后的结果相等。如果相等，则打印" 验证成功！" 的消息，并返回 1，表示验证成功。

14.3 实现效果



运行时间：0.002s

14.4 实验环境

设备名称 DESKTOP-O3UGS4A

处理器 AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz

机带 RAM 16.0 GB (13.9 GB 可用)
设备 ID D232D87C-8384-4711-AF60-22AEEE338FF3
产品 ID 00326-10000-00000-AA108
系统类型 64 位操作系统, 基于 x64 的处理器
笔和触控没有可用于此显示器的笔或触控输入

15 Project22: research report on MPT

研究报告：MPT（默克尔帕特里夏树）的研究

介绍：MPT（Merkle Patricia Tree）是一种用于存储和验证数据的树状数据结构，广泛应用于区块链技术和分布式系统中。MPT 结合了默克尔树（Merkle Tree）和帕特里夏树（Patricia Tree）的特点，提供了高效的数据存储、检索和验证功能。它的设计目标是在保持数据完整性的同时提供快速的数据访问和验证能力，它通过使用哈希函数对数据进行哈希运算，并使用哈希值来验证数据的完整性。

MPT 的特点和优势：

数据完整性验证：MPT 使用哈希函数对数据进行哈希运算，通过比较哈希值来验证数据的完整性。这使得在分布式系统中验证数据的一致性变得高效和可靠。

空间和时间效率：MPT 使用前缀压缩和路径压缩的技术，将相似的键值对合并并共享前缀，从而减少存储空间的使用和提高数据访问的效率。

动态更新支持：MPT 允许在树中插入、删除和更新数据，而不需要重新构建整个树结构。这使得 MPT 适用于存储大规模、频繁变化的数据集。

轻量级验证：MPT 的叶子节点保存了数据的哈希值，通过验证叶子节点的哈希值和根节点的哈希值，可以轻松验证整个树的完整性。

MPT 的应用领域：

区块链技术：MPT 是以太坊（Ethereum）智能合约中状态存储的基础数据结构。通过 MPT，可以高效地存储和验证区块链上的交易、账户状态和合约代码等数据。

数据库系统：MPT 可以作为一种高效的索引结构，用于支持数据库的快速查询和数据完整性验证。它可以减少数据库的存储空间和提高查询性能。

文件系统：MPT 可以用于存储和验证分布式文件系统中的文件块，确保文件的完整性和可靠性。

分布式存储系统：MPT 可以作为分布式存储系统中数据的索引结构，支持高效的数据检索和验证。

MPT 的研究方向：

性能优化：研究者可以探索改进 MPT 的性能，包括存储空间的优化、查询速度的提升和更新操作的效率改进。

安全性研究：针对 MPT 的安全性进行研究，包括对哈希函数的攻击、数据完整性的保护和防止篡改等方面。

分布式环境下的应用：研究者可以研究如何在分布式环境中使用 MPT 进行数据存储和验证，以解决分布式系统中的数据一致性和安全性问题。

高效存储与检索：研究者可以研究如何结合 MPT 和其他数据结构，实现高效的数据存储和检索功能。

总结：

MPT 作为一种树状数据结构，在区块链技术和分布式系统中发挥着重要作用。它通过结合默克尔树和帕特里夏树的特点，提供了高效的数据存储和验证功能。MPT 具有广泛的应用领域，并且在未来还有很大的研究潜力，可以通过性能优化、安全性研究和分布式环境下的应用来进一步完善和拓展。

给定的代码实现了 Merkle Tree 的构建和打印功能。

MerkleTreeNode 类：

初始化函数（___init___）：接收左子节点、右子节点、值和内容，并将其保存在类的实例变量中。

hash 函数：接收一个值，使用 SHA-256 哈希算法对其进行哈希运算，返回哈希值。

___str___ 函数：返回节点的值的字符串表示形式。

MerkleTree 类：

初始化函数（___init___）：接收值列表，并调用 buildTree 方法构建 Merkle 树。

buildTree 函数：接收叶子节点列表，将其转换为包含 MerkleTreeNode 对象的列表。然后通过迭代将叶子节点合并成更高层级的节点，直到只剩下根节点。在每次合并过程中，使用节点值的哈希值来计算父节点的值，并将左右子节点和内容信息传递给父节点。最后，将根节点保存在类的实例变量 root 中。

printTree 函数：打印 Merkle 树的结构，包括每个节点的左子节点、右子节点、值和内容信息。该函数使用递归方式遍历树。

getRootHash 函数：返回根节点的哈希值。

示例用法：

创建一个 Merkle 树对象（mtree = MerkleTree(elems)），其中 elems 是一个字符串列表，表示树的叶子节点。

打印根节点的哈希值（print("Root Hash: " + mtree.getRootHash() + "+"n")）。

打印整个 Merkle 树的结构（mtree.printTree(mtree.root)）。

这段代码展示了如何使用给定的叶子节点构建 Merkle 树，并打印树的结构以及根节点的哈希值。