Project 4

1、SM3 算法实现

1.1 算法实现过程

1.1.1 辅助函数 P0 和 P1

用于 SM3 算法中的消息扩展步骤，通过位移和异或操作生成新的32位字

1.1.2 逻辑函数 FF 和 GG

是 SM3 算法中的布尔函数，用于压缩函数中的状态更新

1.1.3 循环左移函数

实现32位整数的循环左移操作

1.1.4 填充函数

将输入消息填充到符合 SM3 算法要求的长度：

- 先在消息末尾添加一个 `1` 位
- 添加足够多的 `0` 位
- 使得消息长度在加上64位的长度表示后是512位的倍数

1.1.5 SM3 函数

- 对输入消息进行填充
- 初始化状态向量
- 处理每个512位的消息块
- 扩展消息并更新状态向量
- 将最终的状态向量转换为字节数组并返回

1.1.6 主函数

- 循环迭代，每次迭代调用 `generate_random_message` 生成指定长度的随机消息
- 调用 `sm3_hash` 函数计算随机消息的 SM3 哈希值
- 记录哈希计算的开始时间和结束时间
- 输出哈希值和计算时间
- 计算并输出总用时和平均时间

1.2 运行结果

该算法实现了 SM3 加密功能，并通过对加密操作进行十次迭代以计算平均时间：（此处仅展示部分结果）

```
----------第10次加密----------
随机生成的消息: CA1E9693E3F94EF0009A3FCAB71EFC1A5844D2953ED32F03A5834318C0929648C69233F17066E36BFCEAE0346EC7149AF36B6B0F
2067B056D2EF95C564E8D95EF31BD7CF1AB0C81F980FF4350FFD53964AB27AF46DC5BEBA54FE020E3A57BF4C3002AAA8E3011996D6F4831A17E1C9A5
5AD7F1E869597D0A1D1DAEB76E3091AF7C45DAD7E2594131B6AA8D3157C04A8D9752818C2F9EF6CF62E7B969073EC2D46807F80DE9A561DC0967A826
2C2410BC5A5D683B440C02E24986E1E9422ADAB9B4650C5D76BECCDEEEE8414D96F3233CE8DC73AB9103B1DEDB9261D113269D60FF9C787775141218
D6EE07591BFADB8163ABD237C9579C1C8E4891BE0B20AF0E2EC02C1C033CABCC25A997D1C62E70EC126CDFE7C2C801A3A7BEAEE8DE90FAAAB1F5F9EB
77FCFA807315BC6B94D042E116DD7A1DCFF5EEA981F0D2E40B1A94542EC993973867F55B6EFBADAB2A4FEA23B4CB7D63F4CDFDB41B70CBC284BEF758
FAAD98AF13E4C6C8F382EF0AA13BF1168A2A58157270BD9B8613361BBB5B8FF92512294922D053B39213BE6D9DEECB8EA4187FC2F0E0AB58336838C5
B3AB76C28E53181813E52F63CE71A5557A1A60A9EC1A96E9BB34E3067139313F71A0ED4D7FD069564755FD85756204C42D425064407F322C26F6B044
D8C6B4F60EA9A9025C150EFF9939BFB9554E572C73C9D3F0A830A5718949EECFCD8D5A025697642EDCD54A91602C42F67394490270B8E7BBD5D7F0D6
A359BAAFE2BA8CB3A7AD0AEE9D71314D0B9903913D79A9FD3303B79C99AEC66FFBCE375D50531F69A89BF87395C432F094616A8747AC201E6081E386
6811874BF4BB07C31CDFA826C9D35910692D272E1978B31F54BFB413743E9DF313A45E69BBFE82EBE539E871C13516DD847260094CAFD336A4AEBCE3
F9308F7C5429EBFBE1EE5B1160CF561A7DDCD6C1B68B00925D40DBE010780177C46FD04E4E844EFCA1F39850686FB8822CBA76CAF98856F085258793
1BD8FDD58630E54B164525260A0564ED13BAD2CECFD65CB833550BAE8357D6867E01D643F352E3F3B6C336A61FAFA8E45526B152C3BC2606EB6CDAF9
0A2E34023827C4EBDFE3CF6B13A54D0BE3B77DE7051D22D38612334AD3344E00B7AF43FF24A50C82E7AAF0160885A27EFB272D6B86CFC7B039915206
0DC4B083F51DC6FB2B2C1B4F5CB879A1A4AD03EE1AA627A4FA514199C5C796C70B2FBC253A7393F75E95B495AA82ADDC80275DDC9E091C3F47D34351
BBB0E33E4688C27B1EBE964125646C8A57A93EF0B2714C669C20E5EACEBAFCE942B248805D9E9229DD38A9015FD76A06EC620A88B947A3DA74C184C5
DB6EA02526198B9981A1208997E18C025F2E8CB37E8AE96F0723FD3E7B8F6BBE537DCDF0AB93A0CE4752C519F47796A63B49AC4997ACE14B7F5E0AC1
395EEB8360283A6F95A7B256
第 10 次SM3哈希值: 0D51E034BF7E02B78067E4CEC4038914CBE4E0C08E6E617B5B1B386D43DB205C 时间: 0.088700 秒
================================================================================
执行 10 次SM3哈希的总用时为: 0.700600 毫秒，平均时间为: 0.070060 毫秒
```

可见此时的平均加密时间为0.07006 ms

2、SM3 算法优化

2.1 优化策略

2.1.1 消息填充-预分配内存

- 计算消息的位长度，预留足够的空间来存储填充后的消息
- 复制原始消息到填充后的消息中
- 添加一个 `0x80` 字节作为填充开始标志
- 计算需要填充的零字节数，使消息长度满足 $448 \mod 512$ 位的要求并在消息末尾添加这些零字节
- 将消息的位长度（大端序）附加到消息末尾
- 更新填充后的消息长度

预分配内存

避免动态内存分配：预先分配足够的内存空间，减少动态分配的次数

2.1.2 SM3 哈希计算

- 调用 `pad_message` 函数对消息进行填充
- 初始化 SM3 的状态变量 `state`
- 遍历填充后的消息，每次处理一个块

- 将消息块转换为32位字并扩展为68个字
- 执行 SM3 的压缩函数，更新状态变量

- 将最终的状态变量转换为字节数组并存储在 `hash_output` 中

循环展开

减少循环次数：适当展开循环可以减少循环控制的开销，提高指令级并行度。

2.2 运行结果

经过上述优化，此时 SM3 算法运行效率得到提升，再次对加密操作进行十次迭代以计算平均时间：（此处仅展示部分结果）

```
----------第10次加密----------
随机生成的消息: 1522CA8758DD3B39A2132CE151F32292 75DFF6C1FAD74948908C9757A22663CE 708B410A2F2D0B15E0FCCE362CD9 5E70B4D36
8BBFADE6BE49AA7566CB 47EC3D683C0C63F5F52596C5D5E1B8 634869D293A2DF36A3DB59A6C6C064A8 E287DD9D65674374679693A81EFB8E12 42
FE18BC883B64B1F0E5B913C367E4 E6973DC7112767746AB59FCC584F612 55A5B3F197DEDB4ACCB7E1BECA80DC24 6B722EFD3737A848F41C6D3866
9CF456 6A95F3AFB8AC988E5C4A89954A14AF11 6FA1D661A6865D2590394486FF59E7B D19436E2F81C2770A8F1F9A5BE1E135C 2A9DC3CFA6D4FFC
9C11DEB1DF996919C 754B4BE3C9AA9DCBE76BCA5431AF9 EFC86860D096DB88721CFA1CFAA882 C5929CB7E5D9DA44D6AEF487F397536C 89D73195
75110E8B02D3FE3D18D3BD9 353B1408670F66E84CF62179F8FB2C7 A4D5A4B1DADBA674346DDC7C152A50 139342E097C49C62C61F80AF70F48A AE
94392862FBB8C9A23CD1D54AFFF9A6 8F651B0EAFE4D6221DA57FDAD5780 E5DD8CD59BB0E28D7EA043AE84F85DA 174C821D113935E4671C310FCD4
9E10 3A8662BBA51479817FDF73E0A078CFD 7C52C9C4839A1B6C823C57EAD8B246 79FD96A7DD3B2DC3B8CB655A07C44B1 1D1266CAE3DCA6D173AF
9864D0F6B72C 21C9583B26788A23EEA17EC8DB913ABF 9F37D0CD3BEC7772BBDFB9EC1CB0C3B2 AB819E8195451B4AA84C5D4E8C65F7DE 393FD17D
BE0A1DC24B33C1D1B1A3CE0 5ADF3F1DF27B1FE3A3CB6BE9CFA0C412 D8AC48D1A8282B113F6789123B036 78E66CABD3687069FAF39F41EABBCE6 4
8B6557E2B9C7517986511177A39B9 DA1F99D686FF2319228A5509B8BC9E3 5782AD984B8920FAAA6E65908A96681 5ACC597BBEAF73A81B4DCD08035
1DEF E37C58B198D11A7E4059F56B03396EB 805007C65025FABB9950B1F3DAC9E E01F1C2EEDADD59975389A7DB2594AC8 4EF93C261AEEB34EE4FB
2DF28B68122 25C139F2FF389F53AFF6E0CCB42CD691 AD7D53D1E7D7E9AC77BB6186767E8B9F DCC46858267CFBBD6DC651CB1F752 AA874880C044
91216AE9295F3E35B9DC 80FB54B9E9C45F6B8F6397C4774D95D3 8E163B87769D10257B3F25984BC5536F 6D6848B7ECD53BCFA2E0C88D1E20 909D
89734C5672CC82FE9111A148569 5F97CF562869DFDDD3A0B1DDE407683 F4171CC934C5243D71EB51C5A29CE22E 86E5BBB4387963D0D08927E5391
16 16783827EAE414EAC2B0C767D31ABEB2 198D66140DDBA73B5E6D95F9D7EFD3 A476DA20A235A7996E8F8835CC148 7D277DF213E2A923DAFC3F7
4C7AF821 55653AEFCE943FD67E9A1B4063326BC5 9BD3B28DB181BA8830CDD76A2D1D49B8 4C9B92476F97E52BC971ADBDF24DA 85FA1E192E20DD6
1698E648455543FF5
SM3 哈希值: E4DF58BDC5ACA326EA24C9DFC59B94C5 94E6E974D49D85FB2DE4E48E6D08395  时间: 0.017100 ms
============================================================
执行 10 次SM3哈希的总用时为: 0.208900 ms, 平均时间为: 0.020890 ms
```

可见此时的平均加密时间为0.02089 ms

3、length-extension attack

长度扩展攻击利用哈希函数在处理可变长度消息时通过填充使消息达到固定块大小的特性，允许攻击者在已知某个消息的哈希值的情况下，计算出该消息与额外数据连接后的哈希值，而无需知道原始消息的内容。

设计验证方案如下：

1. 生成一个初始消息并计算其 SM3 哈希值
2. 使用初始消息的哈希状态，对初始消息进行填充，使其长度达到下一个块的边界
3. 将额外的数据附加到填充后的消息上，并计算新的哈希值
4. 比较通过长度扩展攻击得到的哈希值与直接计算的哈希值是否一致，以验证攻击的有效性

在 `main` 函数中添加上面内容：

此时输出结果如下：

可见已实现长度扩展攻击

4、构建 Merkle 树

4.1 Merkle 树

递归地构建 Merkle 树：对于每个节点，如果 `start` 等于 `end` ，则返回一个叶子节点；否则，计算中间位置，递归构建左右子树，然后计算父节点的哈希值，并创建一个新的 `MerkleNode` 作为父节点。

4.1.1 数据结构定义

- `MerkleNode` 结构体表示 Merkle 树中的一个节点，包含节点的哈希值、左子节点和右子节点
- 构造函数初始化节点的哈希值，并将左右子节点设置为空指针

4.1.2 构建 Merkle 树

- 如果 `start` 等于 `end` ，说明是叶子节点，直接返回该叶子节点
- 否则，计算中间位置 `mid` ，递归构建左子树和右子树
- 将左右子节点的哈希值拼接起来，计算父节点的哈希值，并创建父节点
- 返回父节点

4.2 存在性证明

通过遍历 Merkle 树，记录路径上兄弟节点的哈希值，最终返回这些哈希值作为证明。从叶子节点的哈希值开始，依次与证明中的哈希值组合并计算父节点的哈希值，最终检查计算出的根哈希值是否与给定的根哈希值一致。

4.2.1 生成存在性证明

- 生成指定索引叶子节点的存在性证明

- 从根节点开始，根据索引决定向左还是向右子树移动，并将兄弟节点的哈希值加入证明列表
- 返回证明列表

4.2.2 验证存在性证明

- 验证给定的存在性证明是否有效
- 从叶子节点的哈希值开始，依次与证明中的哈希值组合，计算父节点的哈希值
- 最终计算得到的哈希值应与根哈希值相同

4.3 不存在性证明

4.3.1 查找目标哈希值

在 Merkle 树中查找目标哈希值，并记录路径上的哈希值。

4.3.2 生成不存在性证明

调用 `findTargetHash`，如果目标哈希值不存在于树中，则返回路径上的哈希值作为不存在性证明；如果目标哈希值存在于树中，则无法生成不存在性证明。

4.4 主函数

生成随机消息，计算叶子节点的哈希值，构建 Merkle 树，生成并验证存在性证明，生成不存在性证明。

4.5 运行结果

```
已计算 86000 / 100000 个叶子节点的哈希值。
已计算 87000 / 100000 个叶子节点的哈希值。
已计算 88000 / 100000 个叶子节点的哈希值。
已计算 89000 / 100000 个叶子节点的哈希值。
已计算 90000 / 100000 个叶子节点的哈希值。
已计算 91000 / 100000 个叶子节点的哈希值。
已计算 92000 / 100000 个叶子节点的哈希值。
已计算 93000 / 100000 个叶子节点的哈希值。
已计算 94000 / 100000 个叶子节点的哈希值。
已计算 95000 / 100000 个叶子节点的哈希值。
已计算 96000 / 100000 个叶子节点的哈希值。
已计算 97000 / 100000 个叶子节点的哈希值。
已计算 98000 / 100000 个叶子节点的哈希值。
已计算 99000 / 100000 个叶子节点的哈希值。
已计算 100000 / 100000 个叶子节点的哈希值。
所有叶子节点的哈希值计算完毕。
开始构建Merkle树...
Merkle树构建完毕。根哈希值：7380166F4914B2B9172442D7DA8A60A96F30BC163138AAE38DEE4DB0FBE4E
生成存在性证明...
第1个叶子节点的存在性证明生成完毕。
第10万个叶子节点的存在性证明生成完毕。
第1个叶子节点的存在性验证结果：有效
第10万个叶子节点的存在性验证结果：有效
生成不存在性证明...
目标哈希值为：e01ea306c3c1afc4255aebf306ccd74aae29bb25a72233ee039000136efd8de2
目标哈希值存在于Merkle树中，无法生成不存在性证明。
```