《网络空间安全创新创业实践》

实验报告

(2022~2023 学年第 2 学期)

学院： 网络空间安全学院

班级： 21 级密码 1 班

黄仲禹 202100460026

张艺腾 202100460001

张昊宇 202100460084

Project1 implement the naïve birthday attack of reduced SM3

运行结果:

```
C:\Users\zyt\AppData\Local\Programs\Python\Python36-32\python.exe D:\SDU\创新创业实践\Wenlong\1\1.py
153f3cb9
98ddff9f
successful birthday!
50.1085658

Process finished with exit code 0
```

运行时间: 50.1085658s

CPU 型号: 12th Gen Intel(R) Core(TM) i5-1240P 1.70 GHz

生日攻击是利用概率论中的生日问题，找到冲突的 Hash 值，伪造报文，使身份验证算法失效。如果输出是 256 位，我们随机地选择输入，并计算哈希值，在检验第 2^256+1 个输入之前便很可能找到碰撞。仅仅通过检验可能输出数量的平方根次数，便大体能找到碰撞。

实验中我们利用字典遍历搜索与目标 hash 值相同的值来构造碰撞。

```python
def birthAttack():
    list_r_value = []
    list_r = RandomList(pow(2,16))
    for i in range(pow(2,16)):
        m = padding(str(list_r[i]))
        M = Group(m)
        Vn=SM3(M)
        aa=""
        for x in Vn:
            aa += hex(x)[2:]
        list_r_value.append(aa[:8])

    print(list_r_value[0])
    coincide = dict(Counter(list_r_value))
    keyList = [key for key,value in coincide.items()if value>1]
    if len(keyList)==0:
        print('terrible birthday!')
    else:
        print(keyList[0])
        print('successful birthday!')
start=time.clock()
birthAttack()
end=time.clock()
print(str(end-start))
```

Project2 implement the Rho method of reduced SM3

代码结果:

```
C:\Users\zyt\AppData\Local\Programs\Python\Python36-32\python.exe D:\SDU\创新创业实践\Wenlong\2\2.py
攻击成功
509bc91f725655f08faf1302f4eff8e5d6cb98a8b9e079da2274500a9ecc1e83
5
0.0015719

Process finished with exit code 0
```

运行时间:0.0015719s

CPU 型号: 12th Gen Intel(R) Core(TM) i5-1240P 1.70 GHz

Rho attack 将每次 hash 的结果都放入一个数组中，之后的每次 hash 都遍历数组, 如果结果在数组中能够找到, 说明攻击成功, 如果没在数组中, 则将此次 hash 结果放入数组并继续循环直到能在数组中找到结果.

```python
def RhoAttack():
    list_r_value = []
    for i in range(pow(2,32)):
        list_r = random.randint(0, pow(2,64))
        m = padding(str(list_r))
        M = Group(m)
        Vn=SM3(M)
        aa=""
        for x in Vn:
            aa += hex(x)[2:]
        bb=aa[:1]
        if(bb in list_r_value):
            print("攻击成功")
            print(aa)
            print(bb)
            break
        else:
            list_r_value.append(bb)
```

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

实验结果:

```
C:\Users\zyt\AppData\Local\Programs\Python\Python36-32\python.exe D:\SDU\创新创业实践\Wenlong\3\sm3le.py
生成secret
secret: 0.18138089443589844
secret length:19
secret hash:62fb485f270df2b0cece55908f3b63c32763cd79d77103abd09641b82d077bfd
附加消息: 202100460001
----------------------------------------------------
计算人为构造的消息的hash值
hash_guess:1bb1fe7c177c4170e531eab7cef1013293703c89edca3430acdc088f8dc05d56
----------------------------------------------------
验证攻击是否成功
计算hash(secret+padding+m')
new message:
0
 .18138089443589844 0x800x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x
 98202100460001
hash(new message):1bb1fe7c177c4170e531eab7cef1013293703c89edca3430acdc088f8dc05d56
success!
0.0017409

Process finished with exit code 0
```

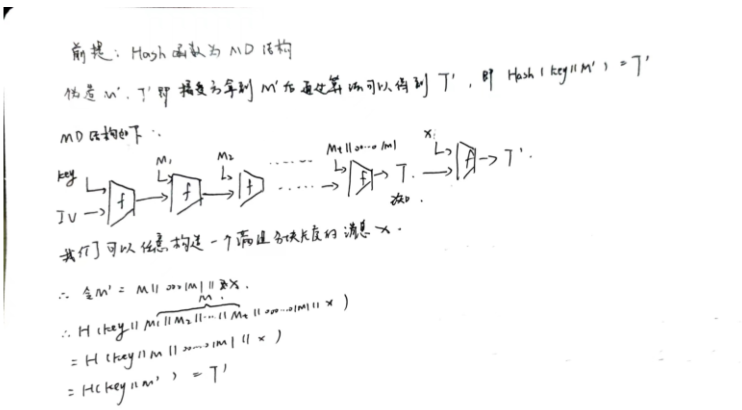
运行时间:0.0017409s

CPU 型号: 12th Gen Intel(R) Core(TM) i5-1240P 1.70 GHz

长度扩展攻击（length extension attack），是指针对某些允许包含额外信息的加密散列函数的攻击手段。对于满足以下条件的散列函数，都可以作为攻击对象：

1、加密前将待加密的明文按一定规则填充到固定长度（例如 512 或 1024 比特）的倍数；

2、 按照该固定长度，将明文分块加密，并用前一个块的加密结果，作为下一块加密的初始向量（IV）。



Project4 do your best to optimize SM3 implementation (software)

运行结果：

```
对比结果为：0
运行时间为：0.4317ms
D:\SDU\创新创业实践\Wenlong\4\Project4\x64\Debug\Project4.exe (进程 24548)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

运行时间：0.4317ms

CPU 型号: 12th Gen Intel(R) Core(TM) i5-1240P 1.70 GHz

结果说明：输出结果为 0 则代表该代码加密的密文和标准密文相同，表明这两个消息的加密值均正确。

```
int main() {
    QueryPerformanceFrequency(&nFreq);
    QueryPerformanceCounter(&nBeginTime);
    cout << "对比结果为：" << SM3_SelfTest() << endl;
    //这里输出0，说明利用该代码加密的密文和标准密文相同，表明这两个消息的加密值均正确。
    QueryPerformanceCounter(&nEndTime);
    time1 = (double)(nEndTime.QuadPart - nBeginTime.QuadPart) / (double)nFreq.QuadPart;
    cout << "运行时间为："<<1000*time1<<"ms";
    return 0;
}
```

Project5: Impl Merkle Tree following RFC6962

运行结果：

```
 '398c2611e085df40b7818e1319c75647bca2013b87122aa5333cf6f9438ab16d', 'fbd71524bc39a156fb75261d9c51fbdc6585d450ad18d06b75d4a67f6fc541db',
 '6dafce999233addc517e2ebc15142d12b5ba9e1eda3354dab155acdcaea62b30'], ['248f00f039da3064e5228f8309e546edb7dc1cca44e334ffe5e165536ae57c5a',
 '77338af147d4196be2033e9c3699ad2e0b6540030fb04d4f8406a26028ee3bb0', '18b441f8e162dc1955fafc8b69c328c07c55528d7518d20b6aa09170550bf3c7',
 'aa928348bc6a553d6b1f3b00e63fea9504a0e05ddd3a43c2194fc6eb21953953', '34f014cd97e94319636e7ec2495249c00dd96e80df8120fe67ae909f76b03d69',
 '618f23ee61192b69a76fafdeceb53f47759efd6f812a9e0632d5d1184207d774', '6dafce999233addc517e2ebc15142d12b5ba9e1eda3354dab155acdcaea62b30'],
 ['d6c7b83abea532ffdf8baeb8baa6b7322284a3956d0c42356fd128425cdaec5f', 'a849a257f547ad917a117bf62f7ffd507e6570e918b07207096e7498105153eb',
 'f186770a624b939782cd5a0c8b47411f089cbef6684beaddae5e33473f094c11', '6dafce999233addc517e2ebc15142d12b5ba9e1eda3354dab155acdcaea62b30'],
 ['6ac8167fa36721c360953cc204b353f41f88c70b642bfaad58e52729580d49a0', 'e96b263efd335d2853b829fe2b83fd3be04da72fcb0325be11bfb846a9b20d09'],
 ['e29ad6826b975c2e705c05380a88ccfcc66f9949959bb97b4413d3a05c576d0c']]
指定元素n包含于Merkel Tree的证据：
[['ec4d02d16fb6f19491ad8fa71e10356c67b052d393d7d116f167086c7968ce3b', '0be82e77700661c1143b1ed7d358d91702e46a88f46c0fab8a2e1af922e1958e'],
 ['74ad7b95dbc4012c3105a2988e46c9fb52919f7b7b0c927d4f242bb3c2c20d65', '23df5fbbafa468e5e3a58d942e2ec007badad9c04078cd090890b811b1b82deb'],
 ['fe0f98370c33b7cd70d14697ad1f7b17e9721f53998a917159bfb8885212e726', 'dc6e75489ab063b891d2536bd07ddace46cb0515351dc24a9b7d42010624e639'],
 ['af6954181ce560a2433bd568f70e2d0d241539e0654801369a76317ca02b1a71', 'f6d36e6961db64f17ad6f8579c11f2b3f0afb48565b5f6f700bfd655f494bb71'],
 ['2ed506dc3982d842f9e90e138d89fcae531c5699cff7917623cf4a71dd0ff6cb', 'c4e6e35ccc94d55f90e03cda563cc36ded056905c7bedc720fa6de9ea82cc693'],
 ['b94f7ab825a388973031cf2941607e763636d151cb5eb9f456a71f4b128ec2f9', '5cdb4be1fd39fedeea7150d0a51b78223e38046eeb584ba8cf64331c6825f40c'],
 ['07c61c2c22ad641904e6fe5d01ebb6ee4514ec3482fbe004bb044d23328e3295', '5e76ac1e6cb90ca8418081cb7c3ef5ed768f26f8163a1ef140d1eff4eeca85e5'],
 ['93d25c8ee0f572c6d34e28e42e2649fcba56a8238511ae807999c26e7dfd2a42', '0d6b116cc0b3cbc1d994898369044b8cc5af1128d1ef56bf10b7b2ac8c259fc5'],
 ['98b8d9e8be52e0b28293184e8e71081476c2ec2a6ea5a81ab45f608786e0d918', 'd4deedb6cf2ab3e833b46b496c216b83a731f5af02f4537cdcbbe34c1b295df0'],
 ['6bc2750c04355c46f384f483250e171260dd547a275ff74eec2d95a8b16bba4d', 'c16992f679c17f7c66093a3ef6122e363e5b1c94cfd1d50e4ea2383a5fd8f46b'],
 ['80df7a495f31ef437e3faa7c783194602c1595892170416f0c232fb0194c4c52', 'ce816a75e42af6525f4da8c76a95e88f7c9236ceab523579adc76cca89d6b19f'],
 ['560974cb774231a02d12ddbcb3466e6fea2285dafce24633b21dc57033498c73', 'e291af65a2da7bb5cc4f528f87962f4db075547622483f9d285b6e3a2138adde'],
 ['b4318b5f31c9122d52a96bd48ae3c21f78288b3179a3d86d39a225b07c804af1', 'bd0edb246312cd4f22d5226d86a99e502276e7545fc37c8df54b804c576e2f99'],
 ['398c2611e085df40b7818e1319c75647bca2013b87122aa5333cf6f9438ab16d', 'fbd71524bc39a156fb75261d9c51fbdc6585d450ad18d06b75d4a67f6fc541db'],
 ['34f014cd97e94319636e7ec2495249c00dd96e80df8120fe67ae909f76b03d69', '618f23ee61192b69a76fafdeceb53f47759efd6f812a9e0632d5d1184207d774'],
 ['f186770a624b939782cd5a0c8b47411f089cbef6684beaddae5e33473f094c11', '6dafce999233addc517e2ebc15142d12b5ba9e1eda3354dab155acdcaea62b30'],
 ['6ac8167fa36721c360953cc204b353f41f88c70b642bfaad58e52729580d49a0', 'e96b263efd335d2853b829fe2b83fd3be04da72fcb0325be11bfb846a9b20d09'],
 ['e29ad6826b975c2e705c05380a88ccfcc66f9949959bb97b4413d3a05c576d0c']]
验证证明正确性的依据：：
True
0.8644504
```

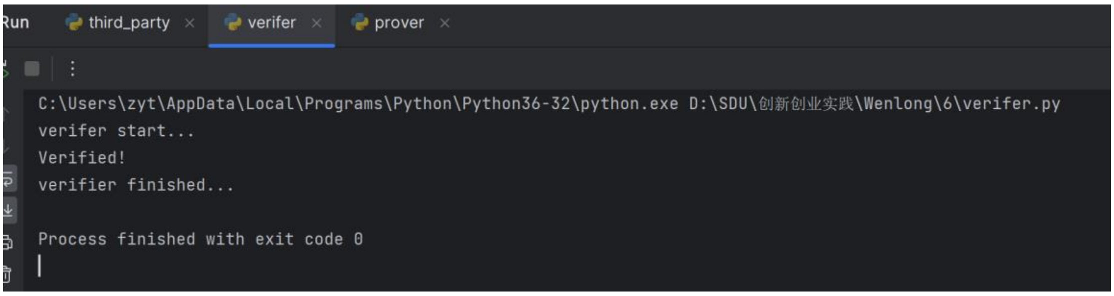
运行时间：0.8644504s

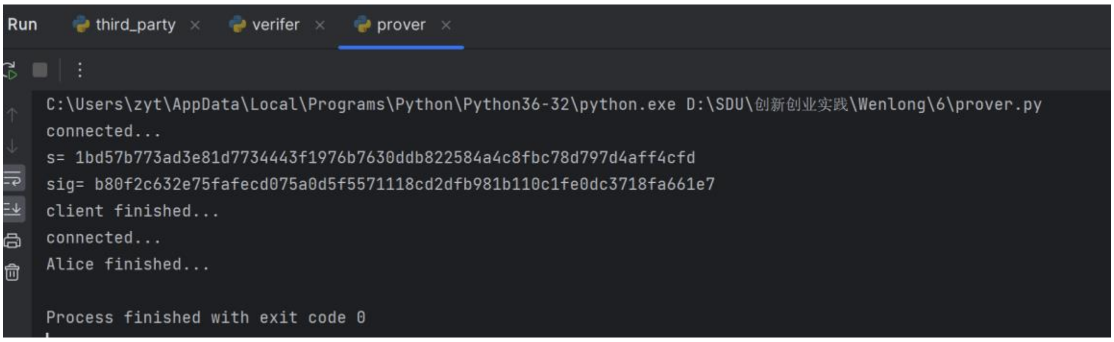
CPU 型号: 12th Gen Intel(R) Core(TM) i5-1240P 1.70 GHz

思路：首先初始化一个二维列表用于存放我们的 Merkel tree，计算树的深度和叶子节点的个数，接着计算数据哈希值并写入叶子节点，每两个子节点计算相加后的哈希值并写入父节点列表。 而对于同一层的节点可以重复调用这个过程，生成下一层（父节点层）Merkle 树的节点，每层向上生成父节点的时候，需要讨论对于节点数为奇数的层的最后一个节点，直接写入下一层（父节点层）；节点数为偶数则正好配对完全，进行递归步骤(3)和(4)的过程，循环步骤(1)计算的树的深度，完成 Merkle 树的生成过程；进行实验测试：输入测试数据，调用 Tree_Generate()函数将整个 Merkle 树 printf 出来，相同深度的 node 位于同一个列表中。

Project6: impl this protocol with actual network communication

运行结果：





CPU 型号: 12th Gen Intel(R) Core(TM) i5-1240P 1.70 GHz

运行说明：依次运行 third_party,verifier,prover。

Project7: Try to Implement this scheme

运行结果：

```
C:\Users\zyt\AppData\Local\Programs\Python\Python36-32\python.exe D:\SDU\创新创业实践\Wenlong\7\7.py
valid proof!
0.0374445

Process finished with exit code 0
```

运行时间：0.0374445s

CPU 型号: 12th Gen Intel(R) Core(TM) i5-1240P 1.70 GHz

```python
start=time.clock()
#_______main________
Max = 4
number_=[2,4,3,4]
number =[3,1,4,1]
s,numList,List = chain(number,Max)
j = compare(number_,number)
prove(number_,s,j,numList,List)
end=time.clock()
print(str(end-start))
```

Project9: AES / SM4 software implementation

运行结果：

```
Run    9_AES-128 ×    verifer ×    prover ×
[41, 245, 214, 150], [161, 77, 206, 238], [24, 174, 155, 245]], [[244, 248, 104, 160], [144, 104, 150, 215], [225, 206, 153, 116], [96, 146, 197, 197]],
[[245, 46, 38, 89], [93, 166, 67, 181], [113, 52, 117, 199], [130, 106, 217, 199]], [[246, 60, 80, 37], [36, 159, 252, 234], [190, 92, 99, 22], [85, 130,
 111, 189]], [[247, 65, 89, 82], [25, 100, 142, 25], [207, 94, 145, 132], [115, 255, 60, 21]], [[248, 79, 2, 181], [103, 96, 248, 149], [193, 162, 130,
74], [235, 142, 195, 186]], [[249, 64, 224, 21], [40, 5, 157, 251], [92, 249, 85, 88], [254, 45, 165, 222]], [[250, 68, 22, 1], [200, 205, 225, 176],
[204, 204, 6, 24], [75, 104, 32, 60]], [[251, 204, 99, 252], [131, 168, 250, 240], [187, 6, 13, 110], [93, 10, 128, 216]], [[252, 19, 74, 225], [150,
119, 130, 75], [103, 30, 136, 138], [244, 213, 130, 127]], [[253, 171, 42, 86], [198, 202, 47, 98], [151, 186, 122, 110], [234, 159, 112, 222]], [[254,
223, 45, 196], [44, 92, 189, 58], [150, 248, 160, 177], [20, 24, 179, 96]], [[255, 141, 87, 51], [96, 74, 44, 211], [106, 171, 199, 12], [227, 25, 59,
181]]]
b'\tT\x7fCx\xc8\xb8\x00\xfc`\xebmTan\x90'
0.0321040153503418

Process finished with exit code 0
```

运行时间：0.0321040153503418s

CPU 型号: 12th Gen Intel(R) Core(TM) i5-1240P   1.70 GHz

结果展示了明文加密后的结果。

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

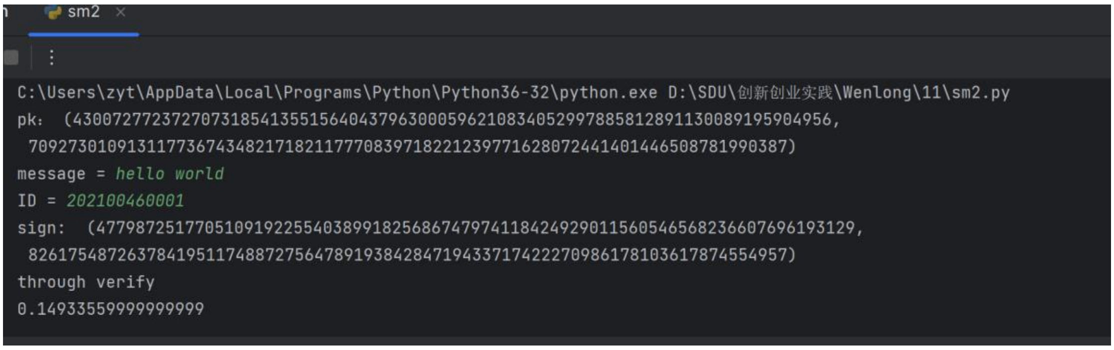
```
C:\Users\zyt\AppData\Local\Programs\Python\Python36-32\python.exe D:\SDU\创新创业实践\Wenlong\10\deduce_pk.py
true pk: (57043747887221245774920880458372928325075544669202642663822313368684365554282,
 53492530457094799165247107043885873707056409915048395868147080955296457603505)
Sign: (55138607159078966712349747491611413959464481264345920800067620626220637965250,
 63103576187208553015472415551600504745091655720264029483809323572821896328628)
deduce pk according to signature......
Candidate pk1: (27493714151511131224517365120096263734640404377330716255398842287393763071513,
 88767733960864823920508091699079813017345641113268486505192388870776361136887)
Candidate pk2: (57043747887221245774920880458372928325075544669202642663822313368684365554282,
 53492530457094799165247107043885873707056409915048395868147080955296457603505)
0.3418415

Process finished with exit code 0
```

运行时间：0.3418415s

CPU 型号: 12th Gen Intel(R) Core(TM) i5-1240P   1.70 GHz

Project11: impl sm2 with RFC6979

运行结果：



```
sm2
C:\Users\zyt\AppData\Local\Programs\Python\Python36-32\python.exe D:\SDU\创新创业实践\Wenlong\11\sm2.py
pk:  (43007277237270731854135515640437963000596210834052997885812891130089195904956,
 70927301091311773674348217182117770839718221239771628072441401446508781990387)
message = hello world
ID = 202100460001
sign:  (47798725177051091922554038991825686747974118424929011560546568236607696193129,
 82617548726378419511748872756478919384284719433717422270986178103617874554957)
through verify
0.14933559999999999
```

运行时间：0.14933559999999999s

CPU 型号: 12th Gen Intel(R) Core(TM) i5-1240P   1.70 GHz

实现 SM2 数字签名的理论依据

密钥生成算法

Alice 选择随机数 dA 做为私钥，其中 0

Alice 计算公钥 PA=dA·G

输出密钥对（sk=dA, pk=PA）签名算法

设 Alice 发签名消息 M 给 Bob，IDA 是 Alice 的标识符，ENTLA 是 IDA 的长度，

dA 是 A 的私钥，基点 G= (xG, yG)，A 的公钥 PA=dA · G= (xA, yA).。

ZA=H (ENTLA ||IDA ||a||b|| xG || yG || xA || yA)， H 是 SM3 算法

①设置 M*=ZA ||M 并计算 e = H(M*)

②产生随机数 k∈[1, n-1]

③计算椭圆曲线点 G1=k · G= (x1, y1)

④计算 r=(e+x1) mod n，若 r=0 或 r+k=n 则返回②

⑤计算 s=(1+ dA)−1·(k −r ·dA)mod n，若 s=0 则返回②

⑥以(r, s)作为对消息 M 的签名

验证算法

接收到的消息为 M′，签名为(r′, s′)和发送者 Alice 的公钥 PA，Bob 执行如下步骤验证合法性：

检验 r′∈[1, n-1]是否成立，若不成立则验证不通过

检验 s′∈[1, n-1]是否成立，若不成立则验证不通过

设置 M*=ZA||M′

计算 e′= H(M*)

计算 t= (r′ + s′) mod n，若 t=0，则验证不通过

计算椭圆曲线点 (x1′，y1′)= s′ · G + t · PA

计算 v=(e′+ x1′) mod n，检验 v=r′是否成立，若成立则验证通过；否则验证不通过

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

运行结果：

sm2_pitfalls：

```
==============================k泄露导致d泄露==================================
sk_a            (private key of A)       0xef44e6ca9d3e5912e049949ab1dd2f96c9ba7a774cc684071f737ac73159be71
msg             (msg from A)             wwl
Sign_ska_msg    (A sign msg using sk_a) (59915580959784634351540781968685675837690550961936092705503698618537558063244,
 53061934677044771804068647973927345518173820176302868174852542446183088209090)
d               (B deduced sk_a)         0xef44e6ca9d3e5912e049949ab1dd2f96c9ba7a774cc684071f737ac73159be71
d=sk_a, B get true sk_a!!!
msg_f           (msg for forge from B)   not wwl
Sign_f          (B Sign msg_f using d)   (61383147509409133153777515988065691835955857074656565265806876853995941395321,
 20460489658249699760103764285414108682692386017862752084863802403832349966267)
B Verify using pk_a...
pass...forge successfully!

======================对不同的消息使用相同的k签名导致d泄露==========================
sk_a            (private key of A)       0xefd241f15cf1ef344d3f4ec4495314b9e664870b29c506fc1d2a7047248004e2
msg1            (msg from A)             sdu
msg2            (msg from A)             wwl
d               (B deduced sk_a)         0xefd241f15cf1ef344d3f4ec4495314b9e664870b29c506fc1d2a7047248004e2
d=sk_a, B get true sk_a!!!
msg_f           (msg for forge from B)   20000460010
Sign_f          (B Sign msg_f using d)   (84504364810504382787115857633194887562839893310546583829782037104284604580064,
 76095508042042306375329232316966684700082714276357221129434369790440707885065)
B Verify using pk_a...
pass...forge successfully!

=================两个不同的user使用相同的k,可以相互推测对方的私钥===================
sk_a1           (private key of A1)      0x3f46a597851a3530028b4906bc51aa48ebdef17d301dc35d92f9fdb716d6cc14
d1              (A2 deduced sk_a)        0x3f46a597851a3530028b4906bc51aa48ebdef17d301dc35d92f9fdb716d6cc14
d1=sk_a1, A2 get true sk_a1!!!
sk_a2           (private key of A2)      0x4e4cc142e1ed0deac7c7847c71ac53349273fe339ab1fbf1ba360709651af094
d2              (A1 deduced sk_a)        0x4e4cc142e1ed0deac7c7847c71ac53349273fe339ab1fbf1ba360709651af094
d2=sk_a2, A1 get true sk_a2!!!
```

```
=======================ECDSA与SM2使用相同的d和k导致d泄露========================
same sk          0x2265dc4b97257590e8d219e5fa167e66fb5e22331f71dc219f0024271bd766e7
d (deduced sk)   0x2265dc4b97257590e8d219e5fa167e66fb5e22331f71dc219f0024271bd766e7
d=sk, get true sk_a!!!
```

Schnorr_pitfalls ：

```
===============================k泄露导致d泄露====================================
sk_a            (private key of A)       0xaa3e1e62e028359af2d7c40ff7c48fb841cbb196266d78bc87b72cf2e36578c3
d               (B deduced sk_a)         0xaa3e1e62e028359af2d7c40ff7c48fb841cbb196266d78bc87b72cf2e36578c3
d=sk_a, B get true sk_a!!!
B Verify using pk_a...
pass...forge successfully!

=======================对不同的消息使用相同的k签名导致d泄露==========================
sk_a            (private key of A)       0x814d8c418eb0c3b2542810ddb73f73153df5349dc115ee8327ea59ed3dbeaf47
d               (B deduced sk_a)         0x814d8c418eb0c3b2542810ddb73f73153df5349dc115ee8327ea59ed3dbeaf47
d=sk_a, B get true sk_a!!!
B Verify using pk_a...
pass...forge successfully!

==================两个不同的user使用相同的k,可以相互推测对方的私钥====================
sk_a1           (private key of A1)      0x35ebc757436d29ce3c9eb5c704498fbd1ef30898454fc49cd99c05dab81e0beb
d1              (A2 deduced sk_a)        0x35ebc757436d29ce3c9eb5c704498fbd1ef30898454fc49cd99c05dab81e0beb
d1=sk_a1, A2 get true sk_a1!!!
sk_a2           (private key of A2)      0x0e5d5ad35e22d228dc681b1a32556bcfccdb4e5c769ad7fc2479c539bc0132aa
d2              (A1 deduced sk_a)        0x0e5d5ad35e22d228dc681b1a32556bcfccdb4e5c769ad7fc2479c539bc0132aa
d2=sk_a2, A1 get true sk_a2!!!

=======================验证(r,s) and (r,-s)均为合法签名========================
B Verify (r,-s)...
pass!

========================ECDSA与SM2使用相同的d和k导致d泄露========================
same sk          0xb24ca7f4220f3e3b159db87806cb62ca97e50d3b15d7e4f734f67da5b2e2ee1e
d (deduced sk)   0xb24ca7f4220f3e3b159db87806cb62ca97e50d3b15d7e4f734f67da5b2e2ee1e
d=sk, get true sk_a!!!
```

ECDSA_pitfalls：

```
===============================k泄露导致d泄露====================================
sk_a            (private key of A)       0x21657e2fb7ea97c31de089df5e3100f18a61a694290a788ed4951dae1854d9aa
d               (B deduced sk_a)         0x21657e2fb7ea97c31de089df5e3100f18a61a694290a788ed4951dae1854d9aa
d=sk_a, B get true sk_a!!!
B Verify using pk_a...
pass...forge successfully!

=======================对不同的消息使用相同的k签名导致d泄露==========================
sk_a            (private key of A)       0x9e0684df7df4d458cd7456dcfc2eb33850ea1fb6c9ad2b48631ce52e564bcf11
d               (B deduced sk_a)         0x9e0684df7df4d458cd7456dcfc2eb33850ea1fb6c9ad2b48631ce52e564bcf11
d=sk_a, B get true sk_a!!!
Sign1 Sign2 use same k
B Verify using pk_a...
pass...forge successfully!

==================两个不同的user使用相同的k,可以相互推测对方的私钥====================
sk_a1           (private key of A1)      0xb4ba83329bc661087ff27a50063a71dce6c8b5f0f2541c4d4d9c55ce2da061e3
d1              (A2 deduced sk_a)        0xb4ba83329bc661087ff27a50063a71dce6c8b5f0f2541c4d4d9c55ce2da061e3
d1=sk_a1, A2 get true sk_a1!!!
sk_a2           (private key of A2)      0xff4edaef88c2e934fa8c3132c74a10b62658a95d45a5ddf3548f7cb3eb101ce7
d2              (A1 deduced sk_a)        0xff4edaef88c2e934fa8c3132c74a10b62658a95d45a5ddf3548f7cb3eb101ce7
d2=sk_a2, A1 get true sk_a2!!!

=======================验证(r,s) and (r,-s)均为合法签名========================
 Verify (r,-s)...
pass!
```

Project13: Implement the above ECMH scheme

运行结果：

```
IDLE Shell 3.9.2
File Edit Shell Debug Options Window Help
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
============= RESTART: C:\Users\huang zy\Desktop\Wenlong\13\ECMH.py ============
单个数哈希之后的值为： 12042441011350399735813979975775326031347426332159617353842759561220888929458
单个数哈希之后的值为： 74224135656430943465202424871434282189032300147177532361828858952239447387622
多个数哈希之后的值为： 34002279613794327700626273780499190742188203698034069732489314384918455174249
多个数哈希之后的值为： 70208165169380754533388899702743799676532831777001389174407025825512606053781
>>>
```

CPU 型号: 12th Gen Intel(R) Core(TM) i5-1240P　 1.70 GHz

该项目为 ECMH 方案的实现，将集合中的元素映射为椭圆曲线上的点，然后利用椭圆曲线上的加法求解哈希值。

Project14: Implement a PGP scheme with SM2

运行结果：

```
C:\Users\zyt\AppData\Local\Programs\Python\Python36-32\python.exe D:\SDU\创新创业实践\Wenlong\14\14.py
-----------------------Sender 加密-------------------------
Generate sm2Key......
pk_s: B9C9A6E04E9C91F7BA880429273747D7EF5DDEB0BB2FF6317EB00BEF331A83081A6994B8993F3F5D6EADDDB81872266C87C018FB4162F5AF347B483E24620207
pk_r: 893cb9392dabd2fac095f657a7e0bc308e32f4b79380d478547f57123dccb3bc4a3a2d009f5826b6624c99dd41baf470a8baf46722f2d36b1d26f19af112c5cd

Generate Session Key......
Session key:  b'ab56fa791ee472e9'

Enc Session key with pk_r......
encryptKey b'lUV/121GaYJHgmd7ykdngfhIsrNT+mgyW3uPAC8MstQX9oBn31zPQYG3WV331qDyrCq3MnTbKSayL/Mn4IgRFm9brZyXauDTgF6MAauHyXmtuNCHQ
 /BGVBQ8iJqlSfOurATE51DaWF6JPjBge/dE9A=='

Enc data with sm4Key......
data:202100460001
encryptData: xstZwDi+CIRMSrMKVl0UAQ==

Sign Session Key with sk_s......
sign:
 YjQwYjMwZjljNmNmZThjNjYwZDI1YzU1MGNkZjViNDI0OTIxY2E5NTM4YTU2ZjBhZjJkYzUxN2FkYjA3OThhOGEzYTgxOWQzMWRiYzhjNjlkMjlmZjlmODhkMGNjMzM3YTNkZWIxNzRjMzNkNjEwZDA0Y
 WZmODNjYzczMWE0NjA=

sender send msg(encryptData||encryptKey||sign) to receiver......
{'encryptData': 'xstZwDi+CIRMSrMKVl0UAQ==', 'encryptKey':
```

```
Sign Session Key with sk_s......
sign:
 YjQwYjMwZjljNmNmZThjNjYwZDI1YzU1MGNkZjViNDI0OTIxY2E5NTM4YTU2ZjBhZjJkYzUxN2FkYjA3OThhOGEzYTgxOWQzMWRiYzhjNjlkMjlmZjlmODhkMGNjMzM3YTNkZWIxNzRjMzNkNjEwZDA0Y
 WZmODNjYzczMWE0NjA=

sender send msg(encryptData||encryptKey||sign) to receiver......
{'encryptData': 'xstZwDi+CIRMSrMKVl0UAQ==', 'encryptKey':
 b'lUV/121GaYJHgmd7ykdngfhIsrNT+mgyW3uPAC8MstQX9oBn31zPQYG3WV331qDyrCq3MnTbKSayL/Mn4IgRFm9brZyXauDTgF6MAauHyXmtuNCHQ/BGVBQ8iJqlSfOurATE51DaWF6JPjBge/dE9A
 ==', 'sign':
 'YjQwYjMwZjljNmNmZThjNjYwZDI1YzU1MGNkZjViNDI0OTIxY2E5NTM4YTU2ZjBhZjJkYzUxN2FkYjA3OThhOGEzYTgxOWQzMWRiYzhjNjlkMjlmZjlmODhkMGNjMzM3YTNkZWIxNzRjMzNkNjEwZDA0
 YWZmODNjYzczMWE0NjA='}
-----------------------Receiver 解密-------------------------

Dec encryptKey with sk_r to obtain Session Key......
Session Key that receiver obtain: b'ab56fa791ee472e9'

Verify Sign with pk_s.....

Dec encryptData with Session Key to obtain data.....
data that receiver obtain: 202100460001

Process finished with exit code 0
```

运行时间：0.0415196s

CPU 型号: 12th Gen Intel(R) Core(TM) i5-1240P 1.70 GHz

本次实验旨在实现一个简易 PGP，调用 GMSSL 库中封装好的 SM2/SM4 加解密函数。

加密时使用对称加密算法 SM4 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 SM4 和会话密钥求解原消息。

Project15: implement sm2 2P sign with real network communication

运行结果：

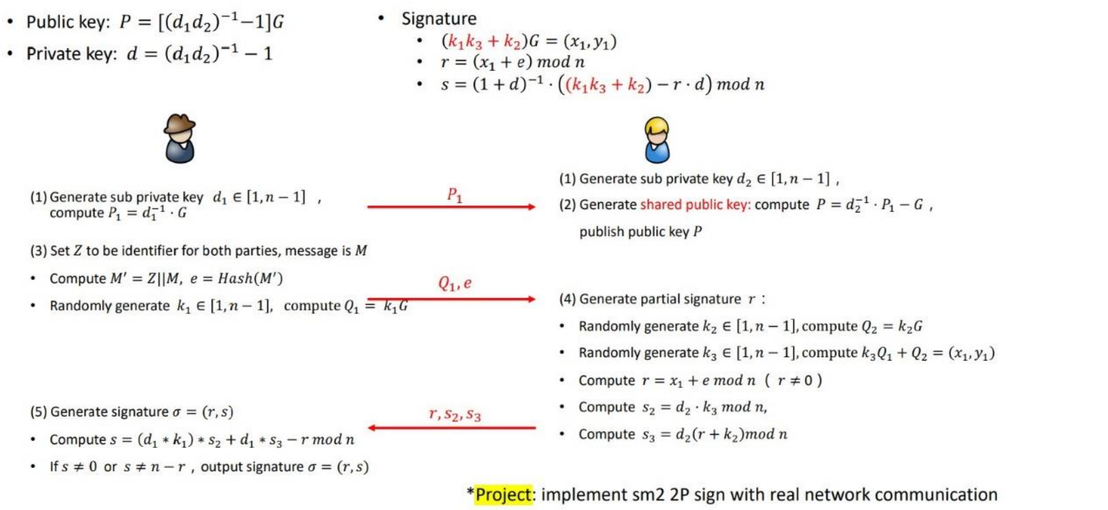
```
C:\Users\zyt\AppData\Local\Programs\Python\Python36-32\python.exe D:\SDU\创新创业实践\Wenlong\15\client.py
connected...
massage:202100460001
Sign: (7631314167974973552692003917778004466614039341327204267790646465746784967888,
 68385233558982600568024706168374334178333974830780738034295372462186219110932)
client finished...

Process finished with exit code 0
```

```
C:\Users\zyt\AppData\Local\Programs\Python\Python36-32\python.exe D:\SDU\创新创业实践\Wenlong\15\server.py
server start...
server finished...

Process finished with exit code 0
```

运行时间：0.12121320000000001s

CPU 型号: 12th Gen Intel(R) Core(TM) i5-1240P   1.70 GHz

两方协同签名方案是指客户端与可信方服务器共同完成签名与验证的方案。 具体操作方式如图所示



通过这种协同验签方式，客户端保留单方私钥 d1，服务器生成单方私钥 d2，二者再生成协商私钥和协商公钥，这样一来单方不再能够生成合法的签名，双方协同生成签名，增加了签名的安全性，可信性。

Project16: implement sm2 2P decrypt with real network communication

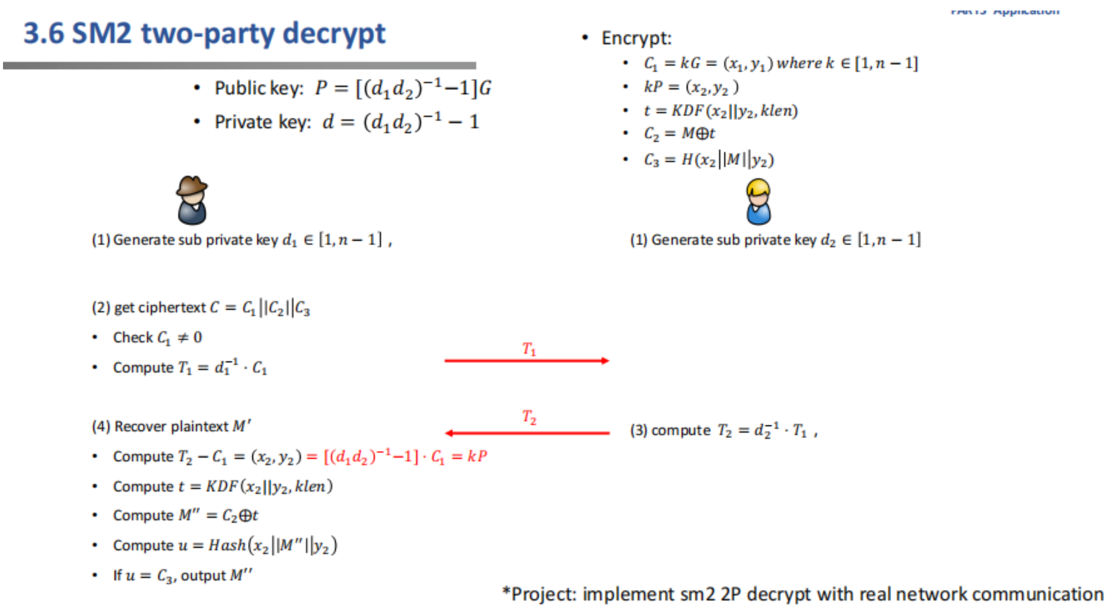
运行结果：

```
C:\Users\zyt\AppData\Local\Programs\Python\Python36-32\python.exe D:\SDU\创新创业实践\Wenlong\16\client.py
pk:  (100066988705817660070993047099007356104367013320992784346313178280699847993567,
 43534556426428413228179464359990739943202279102363296837557655753246595146947)
M: 202100460001
解密结果:  202100460001
client finished...
0.1722736
```

```
C:\Users\zyt\AppData\Local\Programs\Python\Python36-32\python.exe D:\SDU\创新创业实践\Wenlong\16\server.py
server started...
server finished...

Process finished with exit code 0
```

运行时间：0.1722736s

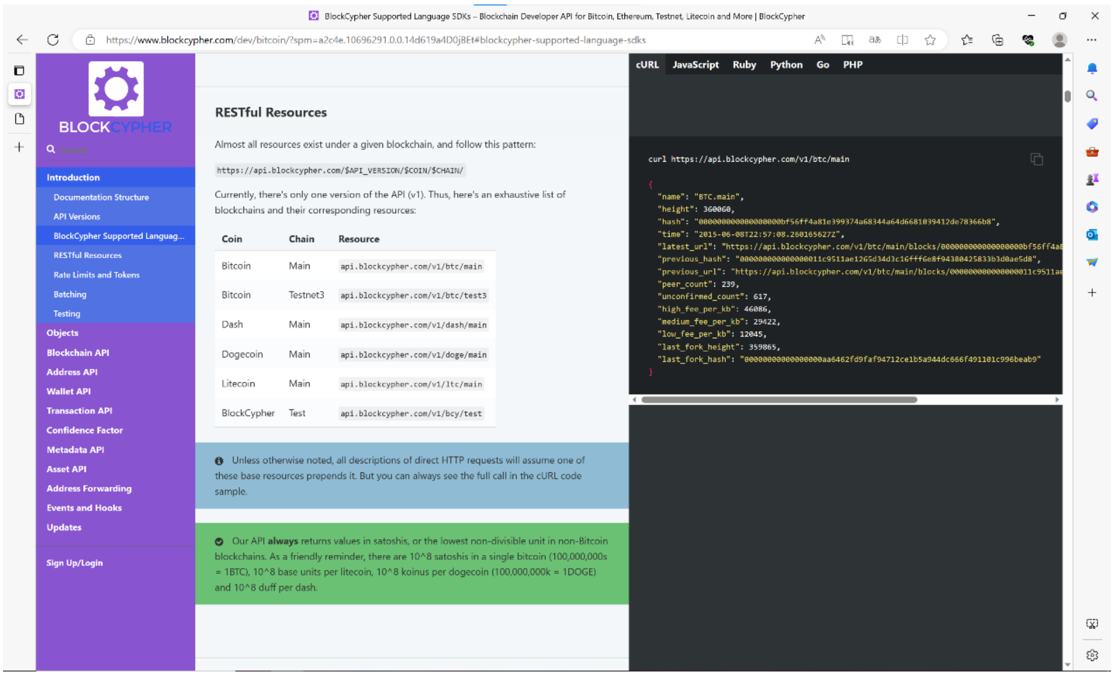
CPU 型号: 12th Gen Intel(R) Core(TM) i5-1240P 1.70 GHz



运行指导：这两个 project 都是先运行 server 再运行 client 即可

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

运行结果：




```python
import requests

url="https://api.blockcypher.com/v1/btc/main"

#get操作访问指定的url（这里即为有关比特币信息的网站）
response=requests.get(url)
response.encoding="utf-8"
message=response.text

#打开文件bitcoin.txt，将数据写入记事本中
with open("bitcoin.txt","wb") as f:
    f.write(message.encode("utf-8"))

```

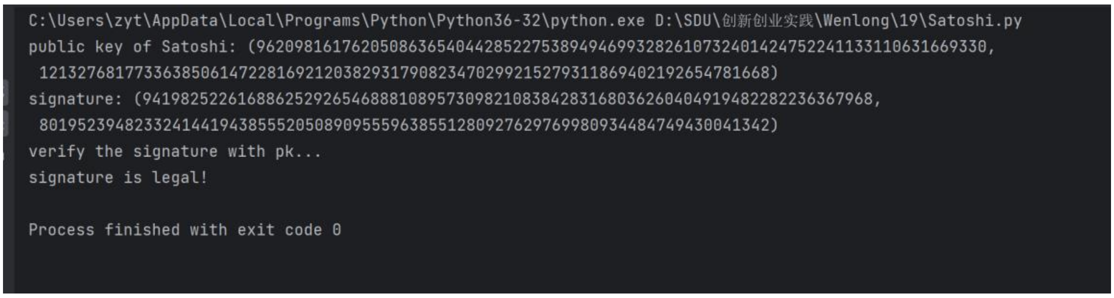
数据：

```
{
  "name": "BTC.main",
  "height": 801512,
  "hash": "000000000000000000030520024f620a1f422dfb59456ade1a726502e530aded",
  "time": "2023-08-03T13:08:13.60290197Z",
  "latest_url": "https://api.blockcypher.com/v1/btc/main/blocks/000000000000000000030520024f620a1f422dfb59456ade1a726502e530aded",
  "previous_hash": "00000000000000000001e2229d30a607e941ba9cce5c442ac14b5c5b4edcdc42",
  "previous_url": "https://api.blockcypher.com/v1/btc/main/blocks/00000000000000000001e2229d30a607e941ba9cce5c442ac14b5c5b4edcdc42",
  "peer_count": 324,
  "unconfirmed_count": 12734,
  "high_fee_per_kb": 22262,
  "medium_fee_per_kb": 9942,
  "low_fee_per_kb": 7545,
  "last_fork_height": 796038,
  "last_fork_hash": "0000000000000000000552fdbbe1edbff2887ea7879dc777b33f8cefc4ba665e"
}
```


CPU 型号: 12th Gen Intel(R) Core(TM) i5-1240P   1.70 GHz

Project19: forge a signature to pretend that you are Satoshi

运行结果：



```
C:\Users\zyt\AppData\Local\Programs\Python\Python36-32\python.exe D:\SDU\创新创业实践\Wenlong\19\Satoshi.py
public key of Satoshi: (96209816176205086365404428522753894946993282610732401424752241133110631669330,
 121327681773363850614722816921203829317908234702992152793118694021926547816668)
signature: (94198252261688625292654688810895730982108384283168036260404919482282236367968,
 80195239482332414419438555205089095559638551280927629769980934484749430041342)
verify the signature with pk...
signature is legal!

Process finished with exit code 0
```

运行时间：0.2774919s

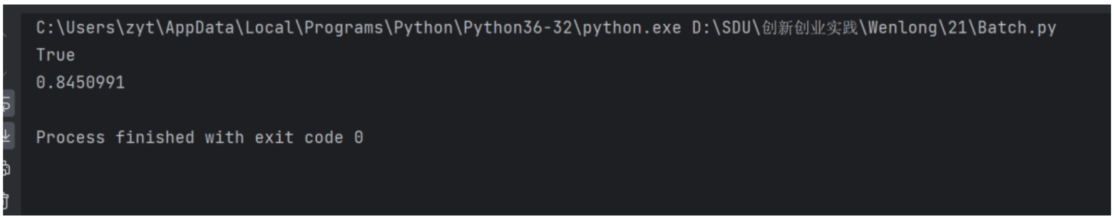
CPU 型号: 12th Gen Intel(R) Core(TM) i5-1240P   1.70 GHz

Project20: ECMH PoC

与 project13 类似

Project21: Schnorr Bacth

运行结果：



```
C:\Users\zyt\AppData\Local\Programs\Python\Python36-32\python.exe D:\SDU\创新创业实践\Wenlong\21\Batch.py
True
0.8450991

Process finished with exit code 0
```

运行时间：0.8450991s

CPU 型号: 12th Gen Intel(R) Core(TM) i5-1240P   1.70 GHz

签名流程：

- Key Generation
  - $P = dG$
- Sign on given message M
  - randomly k, let $R = kG$
  - $e = hash(R||M)$
  - $s = k + ed \bmod n$
  - Signature is : (R, s)
- Verify (R, s) of M with P
  - Check sG vs $R + eP$
  - $sG = (k + ed)G = kG + edG = R + eP$

验签过程：

Schnorr Signature – Batch Verification

Utilize the linear property of Schnorr signature's verification process

- Recall Schnorr signature's verification: $sG = (k + ed)G = kG + edG = R + eP$
- Batch verification equation is :
  - $(\sum_{i=1}^{n} s_i) * G = (\sum_{i=1}^{n} R_i) + (\sum_{i=1}^{n} e_i * P_i)$
  - Attacker can forge signature to pass the batch verification
- Suppose attacker's public key $P_1 = x_1 * G$, to forge signature for public key $P_2 = x_2 * G$
  - x_2 is not known to attacker
  - Attacker randomly choose $r_2, s_2, R_2 = r_2 * G$, and computes $e_2 = h(P_2||R_2||m_2)$
  - Attacker set $R_1 = -(e_2 * P_2)$, and computes $e_1 = h(P_1||R_1||m_1)$
  - Then he derive $s_1 = r_2 + e_1x_1 - s_2 \bmod p$
  - It can be verified that signatures $(R_1, s_1), (R_2, s_2)$ pass the batch verification:
  - $(s_1 + s_2) * G = R_1 + R_2 + e_1P_1 + e_2P_2$
- Defense: randomly choose $a_i \in [0, p-1], i \in [2, n]$ and verifies the following equation:
  - $(s_1 + \sum_{\{i=2\}}^{\{n\}} a_i\, s_i) * G = (R_1 + \sum_{\{i=2\}}^{n} a_i * R_i) + (e_1 * P_1 + \sum_{\{i=2\}}^{n} (e_i a_i) * P_i)$

Project22: research report on MPT

详情请见 GitHub 仓库中的报告