对字符串加解密的处理（二）

一、引言

上一篇中，我们学习了 dump 回填与 hook 回填，这一篇里，我们做更多的讨论。

二、主动解密字符串

2.1 PDD_Secure

在一些情况中，主动调用的代码覆盖率太低，大量的密文字符串没有被解密，又或者我们不希望依赖 Android 设备和环境，希望能主动调用加密函数。这种情况该怎么做？让我们来一探究竟吧。

首先我们一起回顾上一篇中对 pdd_secure.so 的处理，一个函数中字符串加密如下

Java 复制代码

```
byte_6225C ^= 0x5Eu;
byte_6225D ^= 0x69u;
byte_6225E ^= 0x5Fu;
byte_6225F ^= 0xA9u;
byte_62260 ^= 0x21u;
byte_62261 ^= 0xA3u;
byte_62262 ^= 0x8Cu;
byte_62263 ^= 0x8Bu;
byte_62264 ^= 0xE4u;
byte_62265 ^= 5u;
byte_62266 ^= 0x78u;
byte_62267 ^= 0x23u;
```

然后我们手写脚本进行 patch

Java 复制代码

```
from idaapi import patch_byte, get_byte

patch_byte(0x6225c, get_byte(0x6225c) ^ 0x5e)
Out[3]: True

patch_byte(0x6225d, get_byte(0x6225d) ^ 0x69)
Out[4]: True

patch_byte(0x6225e, get_byte(0x6225e) ^ 0x5f)
Out[5]: True

patch_byte(0x6225f, get_byte(0x6225f) ^ 0xa9)
Out[6]: True

patch_byte(0x62260, get_byte(0x62260) ^ 0x21)
Out[7]: True

patch_byte(0x62261, get_byte(0x62261) ^ 0xa3)
Out[8]: True

patch_byte(0x62262, get_byte(0x62262) ^ 0x8c)
Out[9]: True

patch_byte(0x62263, get_byte(0x62263) ^ 0x8b)
Out[10]: True

patch_byte(0x62264, get_byte(0x62264) ^ 0xe4)
Out[11]: True

patch_byte(0x62265, get_byte(0x62265) ^ 0x5)
Out[12]: True

patch_byte(0x62266, get_byte(0x62266) ^ 0x78)
Out[13]: True

patch_byte(0x62267, get_byte(0x62267) ^ 0x23)
Out[14]: True
```

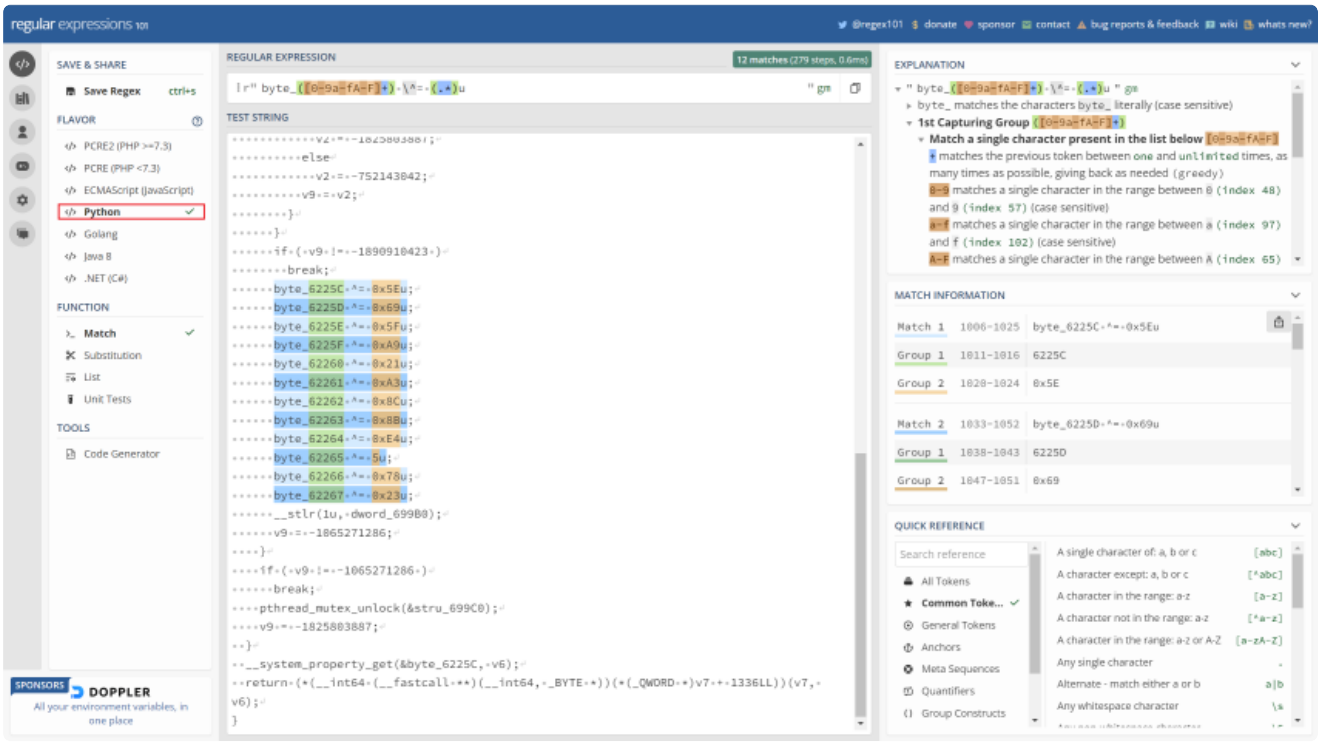
那么是不是说，只需要能解析到密文地址和异或值，我们就可以主动做解密过程，而非被动 dump 或 hook？还是以上一节所展示的函数举例

Python 复制代码

```
__int64 __fastcall sub_2D9B4(__int64 a1)
{
    unsigned int v1; // w8
    int v2; // w8
    int v4; // w8
    _BYTE v6[96]; // [xsp-60h] [xbp-D0h] BYREF
    __int64 v7; // [xsp+0h] [xbp-70h]
    unsigned __int64 v8; // [xsp+8h] [xbp-68h]
    int v9; // [xsp+10h] [xbp-60h]
    unsigned int v10; // [xsp+14h] [xbp-5Ch]
    __int64 v11; // [xsp+18h] [xbp-58h]

    v7 = a1;
    v8 = _ReadStatusReg(ARM64_SYSREG(3, 3, 13, 0, 2));
    v11 = *(_QWORD *)(v8 + 40);
    v1 = __ldar(dword_699B0);
    v9 = 527729684;
    v10 = v1;
    while ( 1 )
    {
        while ( 1 )
        {
            while ( v9 > -752143043 )
            {
                if ( v9 == -752143042 )
                {
                    pthread_mutex_lock(&stru_699C0);
                    if ( __ldar(dword_699B0) )
                        v4 = -1065271286;
                    else
                        v4 = -1890910423;
                    v9 = v4;
                }
                else
                {
                    if ( v10 )
                        v2 = -1825803887;
                    else
                        v2 = -752143042;
                    v9 = v2;
                }
            }
            if ( v9 != -1890910423 )
                break;
            byte_6225C ^= 0x5Eu;
            byte_6225D ^= 0x69u;
            byte_6225E ^= 0x5Fu;
            byte_6225F ^= 0xA9u;
            byte_62260 ^= 0x21u;
            byte_62261 ^= 0xA3u;
            byte_62262 ^= 0x8Cu;
            byte_62263 ^= 0x8Bu;
```

```
53              byte_62264 ^= 0xE4u;
54              byte_62265 ^= 5u;
55              byte_62266 ^= 0x78u;
56              byte_62267 ^= 0x23u;
57              __stlr(1u, dword_699B0);
58              v9 = -1065271286;
59          }
60          if ( v9 != -1065271286 )
61              break;
62          pthread_mutex_unlock(&stru_699C0);
63          v9 = -1825803887;
64      }
65      __system_property_get(&byte_6225C, v6);
66      return (*(__int64 (__fastcall **)(__int64, _BYTE *))(*(_QWORD *)v7 + 1336LL))
```

回忆一下正则表达式， regex101 <https://regex101.com/> 可一个优雅的正则可视化工具。正则规则为 `byte_([0-9a-fA-F]+) \^= (.*)u` ，可以轻松的匹配到异或解密的地址以及值。



接下来挑几个别的函数，测试是否有误判或漏判。越大的函数，其中使用的字符串往往也较多，解密的量也往往更大，更可能暴露问题。使用脚本找到样本中最大的 20 个函数。

Python 复制代码

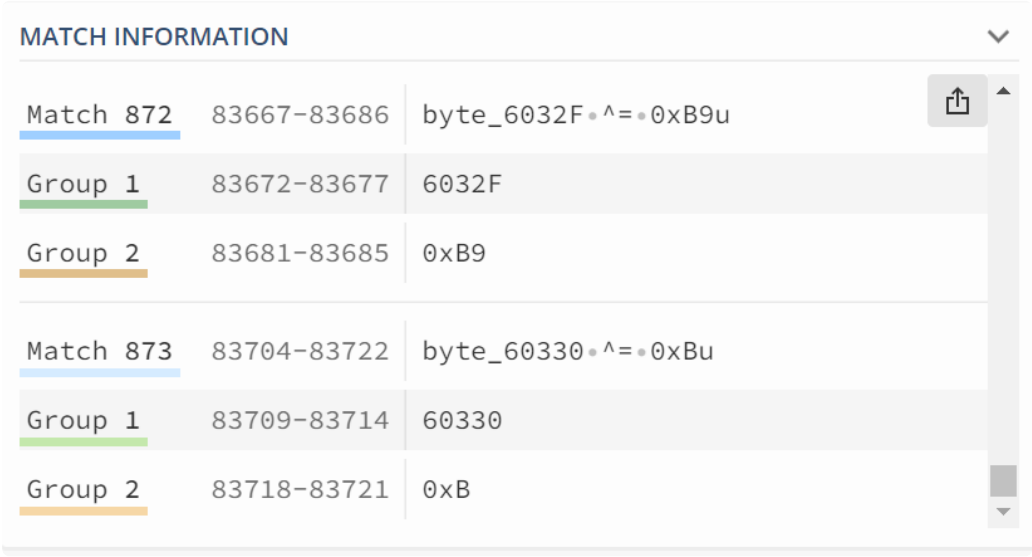
```python
import idaapi
import idautils
from operator import itemgetter
import idc

functionList = []

for addr in idautils.Functions():
    # funcName = idc.get_func_name(addr)
    funcName = idaapi.get_ea_name(addr)
    funcLength = len(list(idautils.FuncItems(addr)))
    oneFuncDict = {"funcName": funcName, "Address": hex(addr), "length": funcLength}
    functionList.append(oneFuncDict)

function_list_by_countNum = sorted(functionList, key=itemgetter('length'), reverse=1
for func in function_list_by_countNum[:20]:
    print(func)
```

结果如下

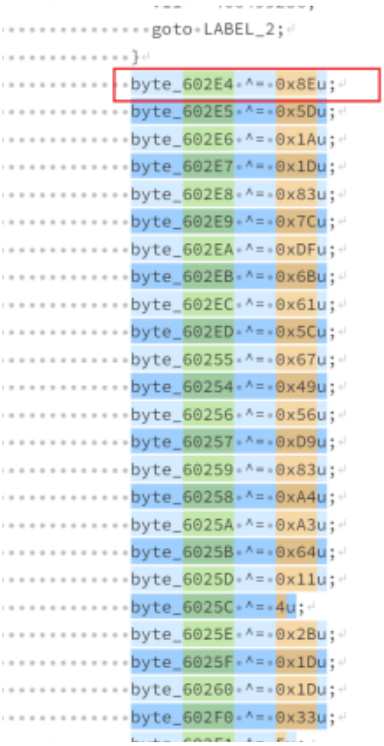
Python 复制代码

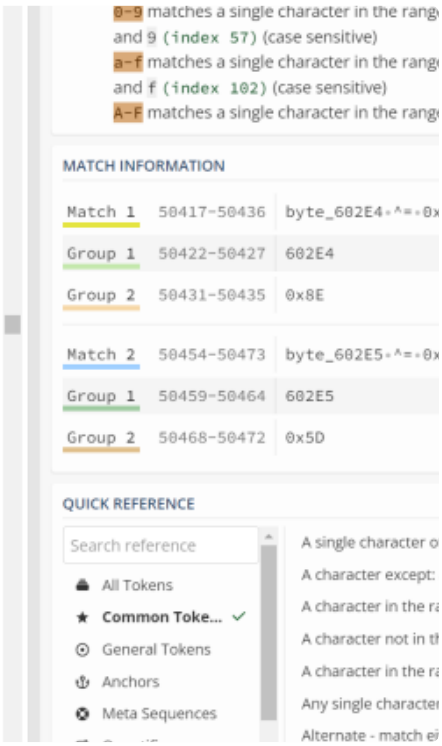
```
{'funcName': 'Java_com_xunmeng_pinduoduo_secure_SecureNative_nativeGenerate2', 'Addr
{'funcName': 'sub_22588', 'Address': '0x22588', 'length': 3863}
{'funcName': 'sub_28044', 'Address': '0x28044', 'length': 2365}
{'funcName': 'sub_2A70C', 'Address': '0x2a70c', 'length': 2155}
{'funcName': 'sub_56630', 'Address': '0x56630', 'length': 1810}
{'funcName': 'Java_com_xunmeng_pinduoduo_secure_SecureNative_nativeGetSysInfo', 'Adc
{'funcName': 'sub_2646C', 'Address': '0x2646c', 'length': 1714}
{'funcName': 'sub_1B784', 'Address': '0x1b784', 'length': 1632}
{'funcName': 'sub_4EB8C', 'Address': '0x4eb8c', 'length': 1537}
{'funcName': 'sub_51C14', 'Address': '0x51c14', 'length': 1528}
{'funcName': 'sub_5046C', 'Address': '0x5046c', 'length': 1498}
{'funcName': 'sub_19FB0', 'Address': '0x19fb0', 'length': 1334}
{'funcName': 'sub_32158', 'Address': '0x32158', 'length': 1223}
{'funcName': 'sub_1F264', 'Address': '0x1f264', 'length': 1222}
{'funcName': 'sub_3E720', 'Address': '0x3e720', 'length': 1195}
{'funcName': 'sub_33BDC', 'Address': '0x33bdc', 'length': 1085}
{'funcName': 'sub_2EF84', 'Address': '0x2ef84', 'length': 1067}
{'funcName': 'sub_5A38C', 'Address': '0x5a38c', 'length': 1066}
{'funcName': 'JNI_OnLoad', 'Address': '0x13598', 'length': 966}
{'funcName': 'sub_59028', 'Address': '0x59028', 'length': 961}
```

最大函数的伪代码有三千多行，我们来看一下匹配效果



一共 873 个匹配项，比较多。首先看一下是否有误判。match 1 点进去，是这一串字符串解密运算的第一个，match 873 点进去，是这一串字符串解密的最后一个，所以应该不存在误判，即所谓的将函数中非解密部分的逻辑识别为字符串解密的情况。





那么漏判是否存在？仔细一看确实不少，有下面这些模式。

```c
// mode 1
byte_6004A = ~byte_6004A;
byte_600E7 = ~byte_600E7;

// mode 2
BYTE1(dword_601EC) ^= 0x3Eu;
LOBYTE(dword_601EC) = dword_601EC ^ 0xA9;
HIBYTE(dword_601EC) ^= 0x26u;
BYTE2(dword_601EC) ^= 0x5Du;

// mode 3
LOBYTE(xmmword_60430) = xmmword_60430 ^ 0x6F;
BYTE1(xmmword_60430) ^= 0xBAu;
BYTE2(xmmword_60430) ^= 5u;
BYTE3(xmmword_60430) ^= 0x10u;
BYTE4(xmmword_60430) ^= 9u;
BYTE5(xmmword_60430) ^= 0x7Eu;
BYTE6(xmmword_60430) ^= 0xBEu;
BYTE7(xmmword_60430) ^= 0x1Fu;
BYTE8(xmmword_60430) ^= 0x8Du;
BYTE9(xmmword_60430) ^= 9u;
BYTE10(xmmword_60430) ^= 0x1Du;
BYTE11(xmmword_60430) ^= 0xBEu;
BYTE12(xmmword_60430) ^= 0xB6u;
BYTE13(xmmword_60430) ^= 0x9Du;
BYTE14(xmmword_60430) ^= 0x57u;
HIBYTE(xmmword_60430) ^= 0xD5u;
```

看其他较长的函数，还有下面这一种模式。

```c
// mode 4
LOBYTE(word_61990) = word_61990 ^ 6;
HIBYTE(word_61990) ^= 0x8Du;
```

遇到问题不要慌，我们从模式一开始看。它给人的感觉很怪，样本中每个密文字节的解密步骤都是和另一个字节异或，即使是模式2-模式4，所作的操作依然是异或，为什么这个字节使用取反运算?

请大家回忆一下，异或和取反运算的具体规则。异或和取反都作用于位，并逐位执行操作。取反运算将二进制数 0 变成 1，1 变成 0； 异或运算规定两个二进制数相同为 0 ， 相异为 1 。用表格看起来更清楚。

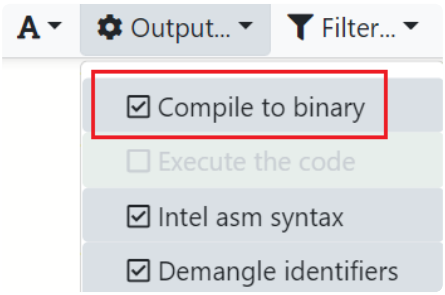
异或运算

P	Q	result
1	1	0
0	1	1
1	0	1
0	0	0

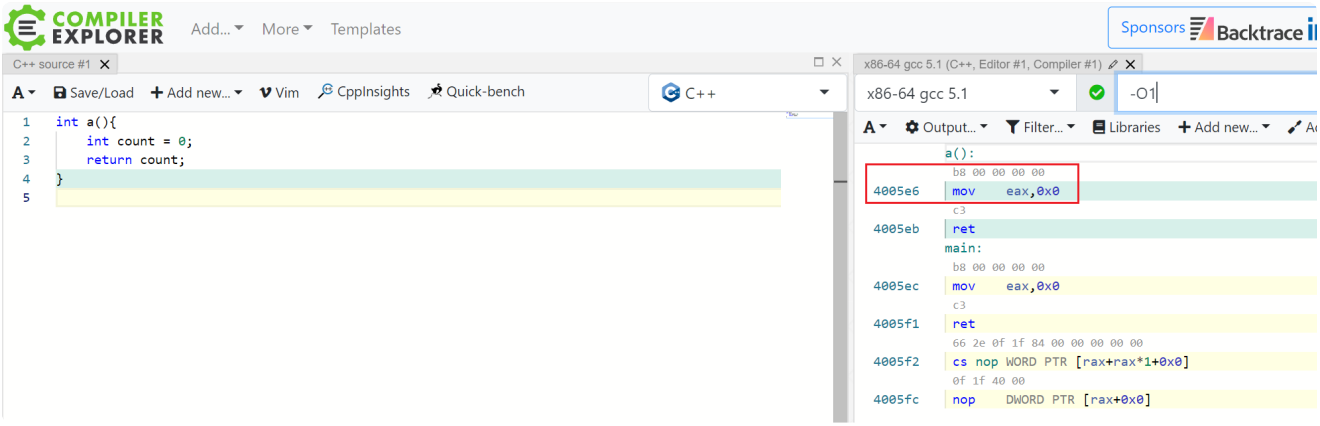
取反运算

P	result
1	0
0	1

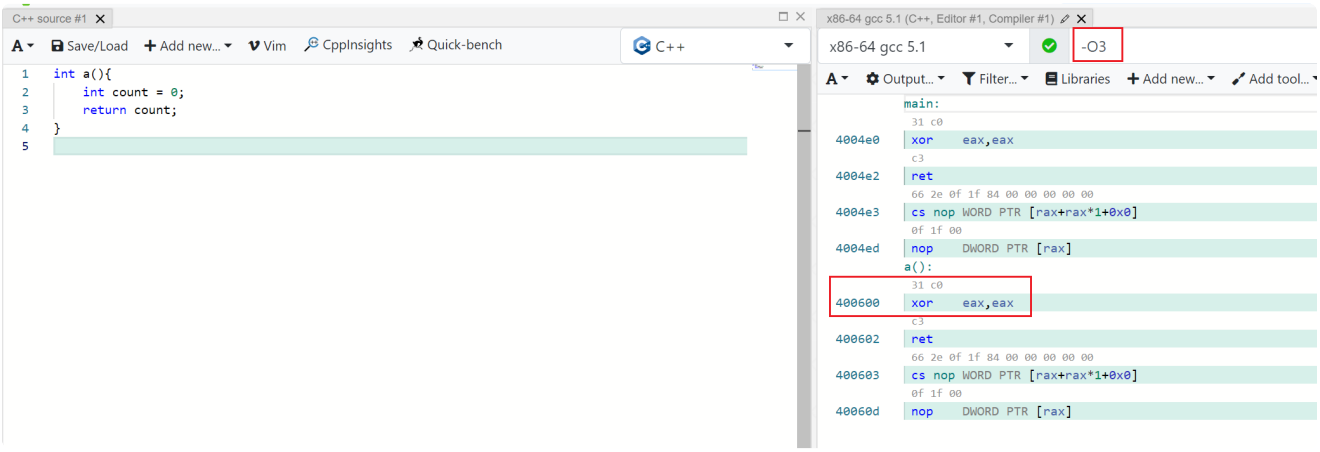
请读者注意，当异或中 Q 为 1时，不论 P 取 0 还是 1，此处异或和取反的结果相同。扩展到8位，或者说单字节时，也就是异或中 Q 为 0b11111111 （0xFF）时，此处异或和取反等价。`byteA & 0xFF = ~byteA` 。基于 godbolt <https://godbolt.org/> 这个方便的工具，我们做简单的演示，观察编译器的优化操作。请在使用时勾选 Compile to binary，可以看的更真切一些。



首先举个例子，在 X86 这样的变长指令架构中，将 eax 寄存器赋值为 0 ，直译是 `mov eax,0` ，这条指令占 5 个字节长度。



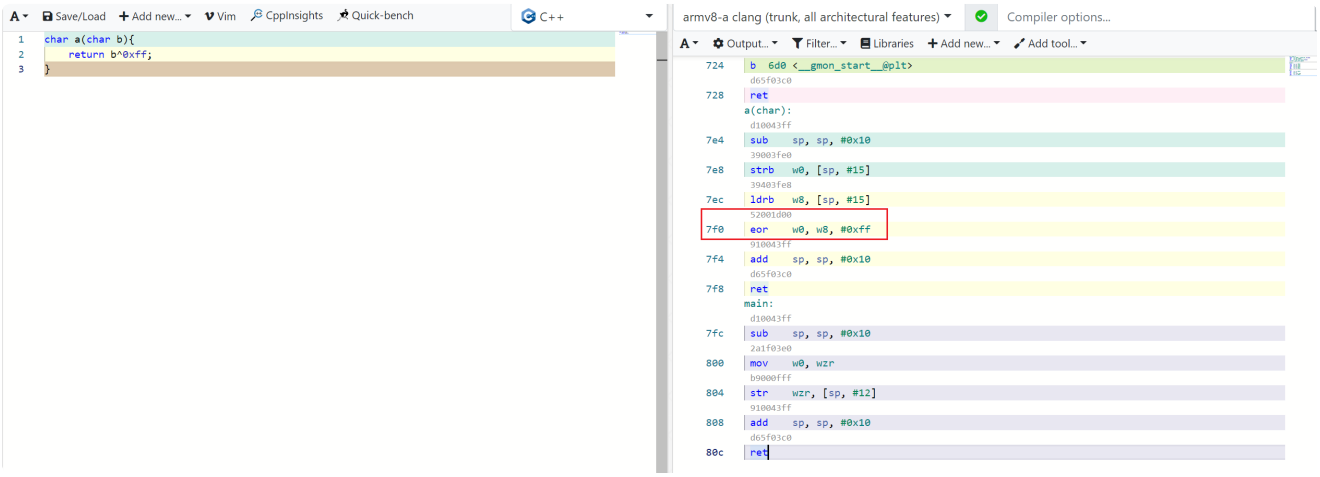
如果我们开启最高等级的编译优化



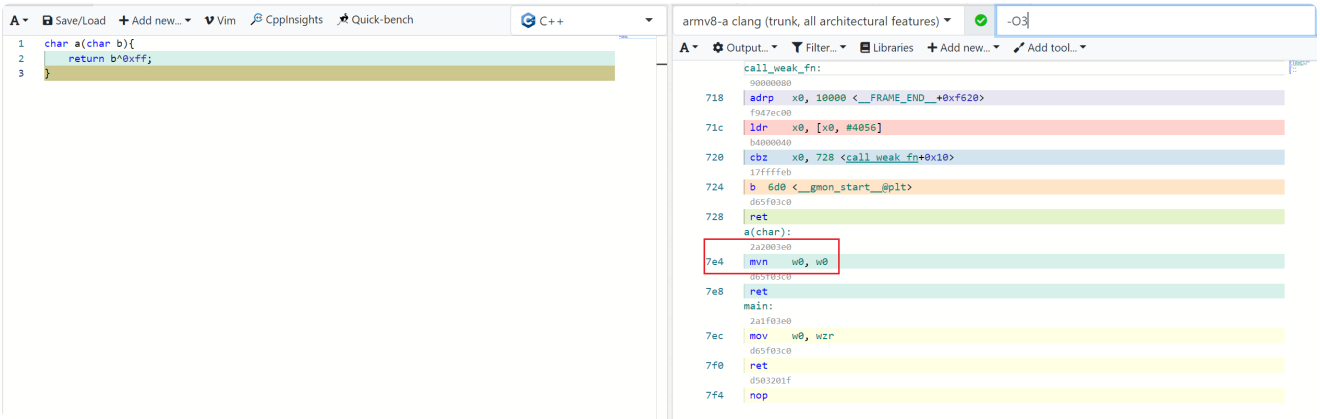
将 eax 寄存器赋值为0 的操作由 `xor eax eax` 完成。按照异或规则，异或双方相同，结果为 0，那么一个数和自身异或时，因为每个位都相同，所以结果总是为 0 。因此 `xor eax eax` 和 `mov eax,0` 等价。

那么给 eax 赋值为 0，用哪种办法更好？编译器的优化已经给出了答案—— `xor eax ea` 更好，因为占的字节更少，从五个字节变成了两个字节。

回到我们的场景里，单字节异或 0xFF 和取反这两个操作等价时，优化为哪一种更好？



开启最高等级的编译优化 -O3



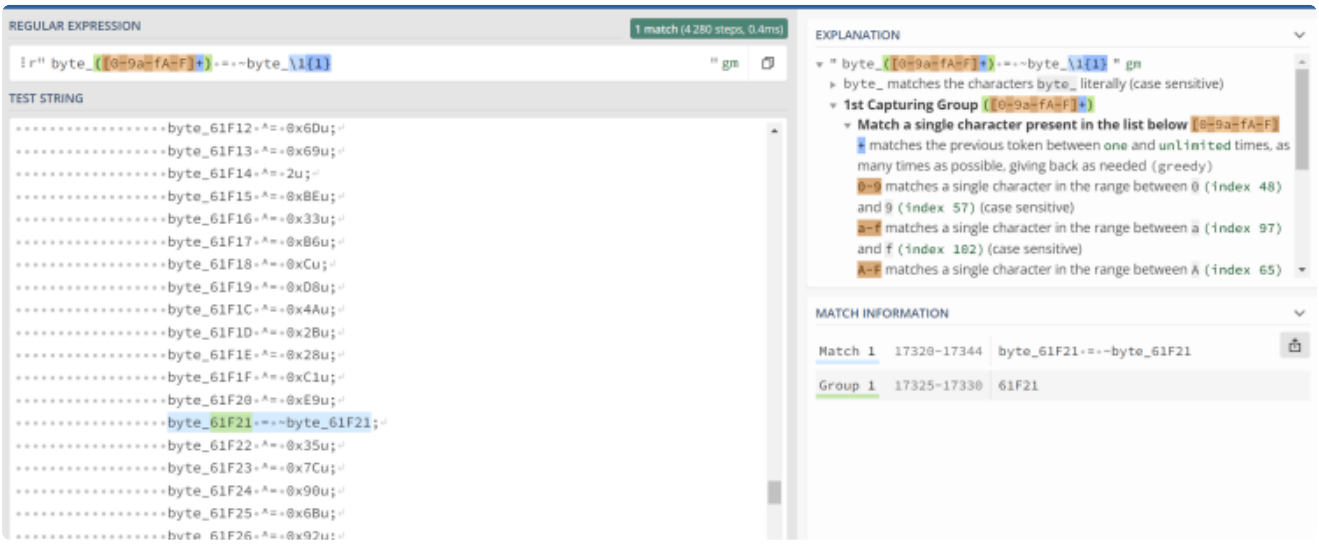
可以发现异或被优化为取反运算。有的朋友可能会感到困惑，X86那样的变长指令集中，两种实现分别占用2字节和5字节，出于节省体积的意图，所做出的优化在情理之中。ARM 这样的固定长度指令集里，此时的取反和异或都由四字节长的单行指令实现，为什么也会优化？

因为除了长度外，我们还应该考虑性能等因素。异或对两个操作数做逐位运算，取反是对自身做逐位的 01 翻转，后者的逻辑更简单，效率应该也更高。

因此所谓的模式 1，其实就是密文字节所需要异或的值恰好是 0xFF，然后在编译中被优化成了取反运算。

```
// mode 1
byte_6004A = ~byte_6004A;
byte_600E7 = ~byte_600E7;
```

为了对它进行匹配，我们需要新的规则： `byte_([0-9a-fA-F]+) = ~byte_\1`



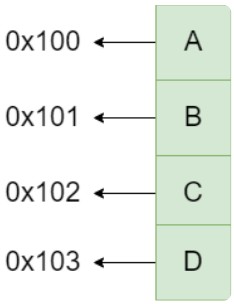
`\1` 意指第一个匹配项，也就是前面 `byte_` 后面括号里的内容，语义即两个 `byte_` 后面跟着的地址一致， `byte_XXX = ~byte_XXX` ，这么做是为了尽量避免产生混淆或误判。

接下来看 mode2 - mode4

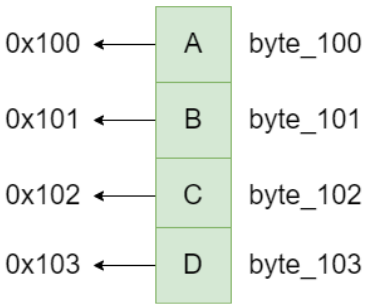
C 复制代码

```
// mode 2
BYTE1(dword_601EC) ^= 0x3Eu;
LOBYTE(dword_601EC) = dword_601EC ^ 0xA9;
HIBYTE(dword_601EC) ^= 0x26u;
BYTE2(dword_601EC) ^= 0x5Du;

// mode 3
LOBYTE(xmmword_60430) = xmmword_60430 ^ 0x6F;
BYTE1(xmmword_60430) ^= 0xBAu;
BYTE2(xmmword_60430) ^= 5u;
BYTE3(xmmword_60430) ^= 0x10u;
BYTE4(xmmword_60430) ^= 9u;
BYTE5(xmmword_60430) ^= 0x7Eu;
BYTE6(xmmword_60430) ^= 0xBEu;
BYTE7(xmmword_60430) ^= 0x1Fu;
BYTE8(xmmword_60430) ^= 0x8Du;
BYTE9(xmmword_60430) ^= 9u;
BYTE10(xmmword_60430) ^= 0x1Du;
BYTE11(xmmword_60430) ^= 0xBEu;
BYTE12(xmmword_60430) ^= 0xB6u;
BYTE13(xmmword_60430) ^= 0x9Du;
BYTE14(xmmword_60430) ^= 0x57u;
HIBYTE(xmmword_60430) ^= 0xD5u;

// mode 4
LOBYTE(word_61990) = word_61990 ^ 6;
HIBYTE(word_61990) ^= 0x8Du;
```

其实它们都是一个问题—— IDA 对数据的错误分析。下面是四个字节的连续数据



左侧是地址，右侧 ABCD 指代其具体的值。如果 IDA 将这四个字节识别为彼此无关的字节（byte），按照IDA 默认命名法，应该是下面这样



这符合语义和我们的期待，反编译代码呈现下面这种，很好匹配

```c
byte_60030 ^= 0xC3u;
byte_60031 ^= 0xA5u;
byte_60032 ^= 0x95u;
byte_60033 ^= 0x8Bu;
```

但 IDA 有时候会分析错误，比如将两个连续的 byte 识别为一个 word （字）。在这种情况下，访问这个 word 的某一个字节时，IDA 用 LOBYTE 和 HIBYTE 标识第一和第二个字节。这并不难理解，LOBYTE 是 low byte 的缩写，即低的字节，HIBYTE 是 high byte 的缩写，即较高的那个字节。

我们的模式 4 就是这样的，0x61990 和 0x61991 地址处的两个字节被识别为一个 word。

```c
// mode 4
LOBYTE(word_61990) = word_61990 ^ 6;
HIBYTE(word_61990) ^= 0x8Du;
```

因此它等价于

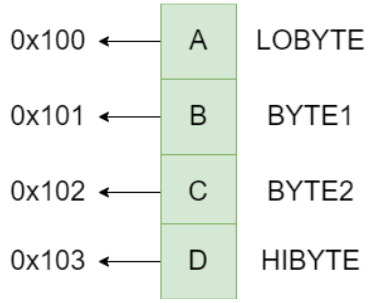
```c
byte_61990 ^= 6u;
byte_61991 ^= 0x8Du;
```

我们希望 IDA 识别为第二个形式，出于两方面原因。首先从语义上看，密文解密是许多密文字节和字节做异或，其本意就是许多单个字节。其次从方便匹配的角度看，byte 显然格式更简单，都是byte的话可以统一处理。

继续往下看，mode 2

```c
BYTE1(dword_601EC) ^= 0x3Eu;
LOBYTE(dword_601EC) = dword_601EC ^ 0xA9;
HIBYTE(dword_601EC) ^= 0x26u;
BYTE2(dword_601EC) ^= 0x5Du;
```

IDA 将这四个 byte 识别为一个 double word（双字），简写为dword。LOBYTE 代表最低字节，HIBYTE 代表最高字节。BYTEX 代表从较低地址到较高地址排序的多个字节。



因此模式 2 等价于下面这种更简单的形式

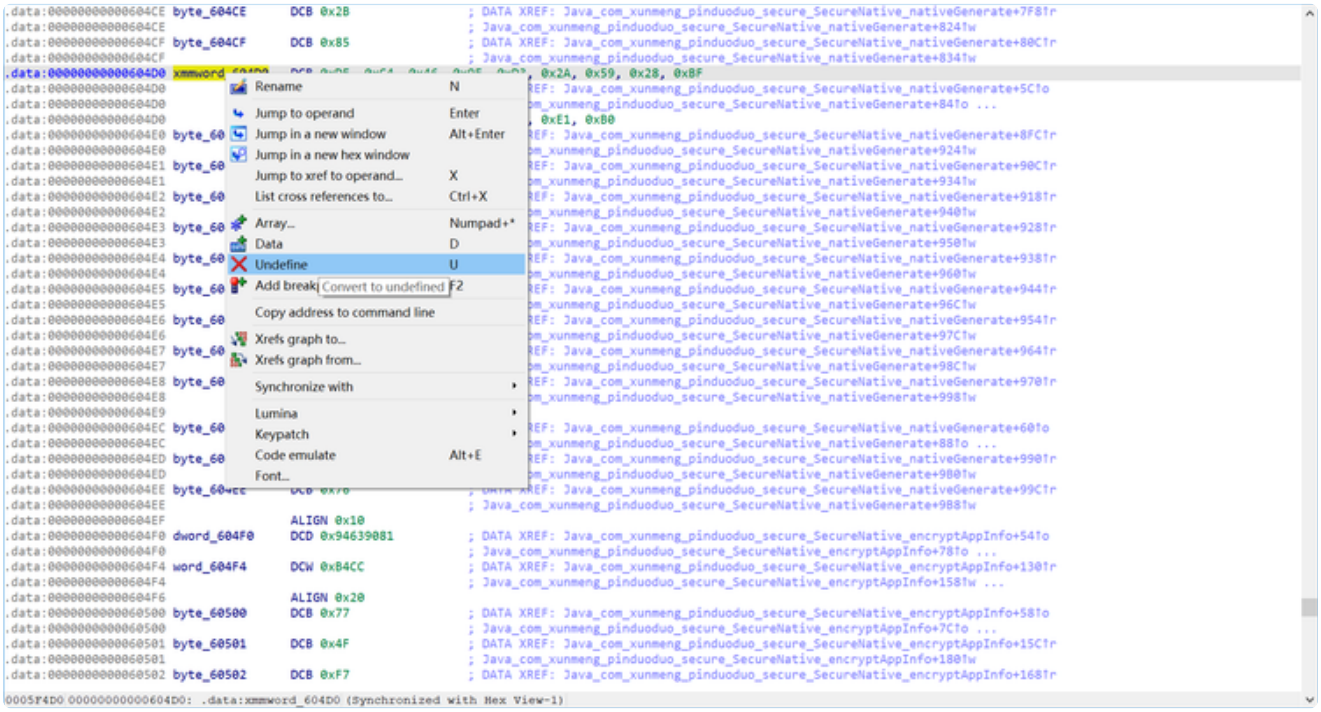
```
byte_601ED ^= 0x3Eu;
byte_601EC ^= 0xA9u;
byte_601EF ^= 0x26u;
byte_601EE ^= 0x5Du;
```

mode 3则是更大面积的错误识别—— 16个字节被识别为一个 xmmword 。LOBYTE、HIBYTE标识最低以及最高的那个字节，BYTE1-BYTE14代表中间的14个字节。

```
// mode 3
LOBYTE(xmmword_60430) = xmmword_60430 ^ 0x6F;
BYTE1(xmmword_60430) ^= 0xBAu;
BYTE2(xmmword_60430) ^= 5u;
BYTE3(xmmword_60430) ^= 0x10u;
BYTE4(xmmword_60430) ^= 9u;
BYTE5(xmmword_60430) ^= 0x7Eu;
BYTE6(xmmword_60430) ^= 0xBEu;
BYTE7(xmmword_60430) ^= 0x1Fu;
BYTE8(xmmword_60430) ^= 0x8Du;
BYTE9(xmmword_60430) ^= 9u;
BYTE10(xmmword_60430) ^= 0x1Du;
BYTE11(xmmword_60430) ^= 0xBEu;
BYTE12(xmmword_60430) ^= 0xB6u;
BYTE13(xmmword_60430) ^= 0x9Du;
BYTE14(xmmword_60430) ^= 0x57u;
HIBYTE(xmmword_60430) ^= 0xD5u;
```

现在我们已经完全理解这三种模式的产生原因了，我们必须要学会如何处理这种情况——即当IDA对数据的识别出错时，如何撤销它所作的分析，以及转成你认为正确的数据类型。前一篇已经讲过如何处理了，这里不做赘述。

在IDA给出的定义上右键，选择 undefine ， 撤销 IDA 所分析的类型信息。



对目标函数重新 F5

可以看到数据类型信息变成 unk 类型 （unknow 未知），而且也更容易匹配了，我们用脚本做的更轻松和优雅。

Python 复制代码

```python
import ida_bytes
import ida_idaapi
import idautils
import idc

start = 0
end = 0
for seg in idautils.Segments():
    name = idc.get_segm_name(seg)
    if name == ".data":
        start = idc.get_segm_start(seg)
        end = idc.get_segm_end(seg)
for address in range(start, end):
    ida_bytes.del_items(address, 0, 1)
    ida_bytes.create_data(address, 0, 1, ida_idaapi.BADADDR)
```

找到 data 段，将其中所有数据先 `del_items` 转成 undefined，再 `create_data` 转成 byte_xxx。有的读者可能会心存顾虑，因为 data 段中并非所有数据都是待解密的密文，除此之外的其余部分的定义，可能并无问题，不应该撤销 IDA 对这部分数据的定义，否则就降低了反汇编和反编译的效果。

不必担心，我们在匹配完字节地址和异或值后，会 patch 改变到原 SO 中，然后重新打开 SO，使得 IDA 重新分析。接下来写成完整的脚本，首先需要遍历函数，做反编译。

Python 复制代码

```python
import ida_bytes
import ida_idaapi
import idaapi
import idautils
import idc
import re


def init():
    start = 0
    end = 0
    for seg in idautils.Segments():
        name = idc.get_segm_name(seg)
        if name == ".data":
            start = idc.get_segm_start(seg)
            end = idc.get_segm_end(seg)
    for address in range(start, end):
        ida_bytes.del_items(address, 0, 1)
        ida_bytes.create_data(address, 0, 1, ida_idaapi.BADADDR)


def mode1(codes):
    matches = re.findall(r"byte_([0-9a-fA-F]+) \^= (.*)u", codes)
    if len(matches) > 0:
        for match in matches:
            address = int(match[0], 16)
            xorValue = int(match[1], 16)
            decrypt_c = xorValue ^ idaapi.get_byte(address)
            ida_bytes.patch_byte(address, decrypt_c)


def mode2(codes):
    matches = re.findall(r"byte_([0-9a-fA-F]+) = ~byte_\1", codes)
    if len(matches) > 0:
        for match in matches:
            address = int(match, 16)
            xorValue = 0xFF
            decrypt_c = xorValue ^ idaapi.get_byte(address)
            ida_bytes.patch_byte(address, decrypt_c)


def decryptOne(addr):
    decompilerStr = str(idaapi.decompile(addr))
    mode1(decompilerStr)
    mode2(decompilerStr)


def decryptAll():
    for func in idautils.Functions(0, ida_idaapi.BADADDR):
        try:
            decryptOne(func)
        except:
```

```
            pass


decryptAll()
```

运行结束后，为了识别字符串，如法炮制

Python 复制代码

```python
import ida_auto
import ida_bytes
import idautils

for i in idautils.Segments():
    seg = idaapi.getseg(i)
    segName = idaapi.get_segm_name(seg)
    if "data" in segName:
        startAddress = seg.start_ea
        endAddress = seg.end_ea
        ida_bytes.del_items(startAddress, 0, endAddress)
        ida_auto.plan_and_wait(startAddress, endAddress)
```

运行效果还是很好的，代码质量得到显著提升。

```
sprintf(v217, "eth_check=%d", v50);
v258 = (*(__int64 (__fastcall **)(__int64 *, char *))(*v9 + 1336))(v9, v217);
v259 = sub_30114((__int64)v9, v49);
v260 = sub_30B14((__int64)v9, aSharedpreferen, v222);
v261 = sub_30B14((__int64)v9, aMemoryId, 0LL);
v262 = sub_30B14((__int64)v9, aSdcardId, v302);
v263 = sub_30B14((__int64)v9, aImei, v314);
v264 = sub_30B14((__int64)v9, aImei0, v315);
v265 = sub_30B14((__int64)v9, aImei1, v316);
v266 = sub_30B14((__int64)v9, aDevice0, v317);
v267 = sub_30B14((__int64)v9, aDevice1, v318);
v268 = sub_30B14((__int64)v9, aDevice2, v319);
v269 = sub_30B14((__int64)v9, aMeid, v320);
v270 = sub_30B14((__int64)v9, aMeid0, v321);
v271 = sub_30B14((__int64)v9, aMeid1, v322);
v272 = sub_30B14((__int64)v9, aMac, v235);
v273 = sub_30B14((__int64)v9, aBattery, 0LL);
v274 = sub_30B14((__int64)v9, aUid, v157);
v275 = sub_30B14((__int64)v9, aCookie, v159);
v276 = sub_30B14((__int64)v9, aImsi, v323);
v277 = sub_30B14((__int64)v9, aImsi0, v324);
v278 = sub_30B14((__int64)v9, aIms, v325);
v279 = sub_30B14((__int64)v9, aAndroidId, v236);
v280 = sub_30B14((__int64)v9, aSno, v237);
v281 = sub_30B14((__int64)v9, aCpuinfo, v238);
v282 = sub_30B14((__int64)v9, aBuildId, v239);
v283 = sub_30B14((__int64)v9, aFingerprint, v240);
v284 = sub_30B14((__int64)v9, aCharacteristic, v241);
v285 = sub_30B14((__int64)v9, aBatteryStatus, v242);
v286 = sub_30B14((__int64)v9, aNdevid, v329);
v287 = sub_30B14((__int64)v9, aWifi, v259);
v288 = sub_30B14((__int64)v9, aSerial, v326);
v289 = sub_30B14((__int64)v9, aSerial0, v327);
v290 = sub_30B14((__int64)v9, aSerial1, v328);
v291 = sub_30B14((__int64)v9, aPddid, v158);
```

```
1432                v11 = 466499236;
1433                goto LABEL_2;
1434              }
1435              aMemoryId[0] ^= 0x8Eu;
1436              aMemoryId[1] ^= 0x5Du;
1437              aMemoryId[2] ^= 0x1Au;
1438              aMemoryId[3] ^= 0x1Du;
1439              aMemoryId[4] ^= 0x83u;
1440              aMemoryId[5] ^= 0x7Cu;
1441              aMemoryId[6] ^= 0xDFu;
1442              aMemoryId[7] ^= 0x6Bu;
1443              aMemoryId[8] ^= 0x61u;
1444              aMemoryId[9] ^= 0x5Cu;
1445              aMeid0Detect[1] ^= 0x67u;
1446              aMeid0Detect[0] ^= 0x49u;
1447              aMeid0Detect[2] ^= 0x56u;
1448              aMeid0Detect[3] ^= 0xD9u;
1449              aMeid0Detect[5] ^= 0x83u;
1450              aMeid0Detect[4] ^= 0xA4u;
1451              aMeid0Detect[6] ^= 0xA3u;
1452              aMeid0Detect[7] ^= 0x64u;
1453              aMeid0Detect[9] ^= 0x11u;
1454              aMeid0Detect[8] ^= 4u;
1455              aMeid0Detect[10] ^= 0x2Bu;
1456              aMeid0Detect[11] ^= 0x1Du;
1457              aMeid0Detect[12] ^= 0x1Du;
1458              aSdcardId[0] ^= 0x33u;
1459              aSdcardId[1] ^= 5u;
1460              aSdcardId[2] ^= 0x4Cu;
1461              aSdcardId[3] ^= 0x7Au;
1462              aSdcardId[4] ^= 0xAAu;
1463              aSdcardId[5] ^= 0xBBu;
1464              aSdcardId[6] ^= 0xB4u;
1465              aSdcardId[7] ^= 0x9Fu;
1466              aSdcardId[8] ^= 0xAEu;
1467              aSdcardId[9] ^= 0x83u;
1468              unk_603B0 ^= 0x67u;
1469              unk_603B1 ^= 0xCFu;
1470              unk_603B2 ^= 0x38u;
1471              unk_603B3 ^= 0x1Fu;
1472              unk_603B4 ^= 0x9Cu;
1473              unk_603B5 ^= 0x3Bu;
1474              unk_603B6 ^= 0x48u;
```

绝大多数字符串都得到了良好的识别。有些字符串本就是字节数组的形式，不是可见字符串，这种没什么办法。

应该说，对伪代码做正则匹配，这方法效果挺好。但有的时候，我们没法这么做，因为考虑到其他的一些因素，这些因素包括

- 逐个函数反编译，对于大型二进制文件来说，时间开销很大，可能要十几分钟甚至更久，在一些场景里，我们需要非常快的做这种检测和匹配
- 对有些函数的反编译会失败，比如存在花指令，可能大多数函数都无法顺利 F5

在汇编代码做这种匹配，可以规避这两个问题。不需要反编译，带来了很好的速度和兼容性。可以对所有的已识别代码进行处理。但这件事做起来也麻烦多了，以此处为例，需要遍历所有的EOR指令，然后找到两个操作数所对应的寄存器，以及是否符合要求。这其中对应着很多种模式和规则。在伪代码的层面，这些汇编模式和规则被 IDA 的反编译步骤转化成规整的伪C代码，不需要我们处理这种问题。不妨这么说，如果一件事在汇编层面和伪代码层面都可以做成，那么如果汇编代码的布局并不特别清楚直白，那么通过伪代码层面做会更好，否则通过汇编层面做会更好。所谓清晰直白的布局，可见这个项目 <https://github.com/jstrosch/XOR-Decode-Strings-IDA-Plugin/blob/main/xor_string_deobfuscator.py> 以及其对应脚本。

2.2 JDMobileSec

样本：JD.7z (124.9 MB)

SO 是 libJDMobileSec.so，解密函数是 `Sub_ABB0`

```
_BYTE *__fastcall sub_ABB0(float *a1, int a2)
{
  _BYTE *result; // r0
  _BYTE *v5; // r1
  int v6; // r2
  float v7; // s0

  result = malloc(a2 + 1);
  if ( a2 >= 1 )
  {
    v5 = result;
    v6 = a2;
    do
    {
      v7 = *a1++;
      --v6;
      *v5++ = ((int)(float)(v7 + v7) ^ 0xDE) + 34;
    }
    while ( v6 );
  }
  result[a2] = 0;
  return result;
}
```

找一个具体调用处看看

```
int sub_13404()
{
  _BYTE *v0; // r4
  char v2[92]; // [sp+0h] [bp-70h] BYREF

  v0 = sub_ABB0(flt_26E70, 20);
  _system_property_get(v0, v2);
  free(v0);
  return atoi(v2);
}
```

sub_ABB0 参数 1 是指向密文内容的指针，参数2是解密长度。解密的具体算法只是简单的异或和加法，解密后的明文放在 malloc 分配的堆内存里，并予以返回，比如上面的 v0 就是这样。

读者可以 Hook 测试一番

JavaScript 复制代码

```javascript
function hookInit(moduleName){
  var linkername;
  var call_constructor_addr = null;
  var arch = Process.arch;
  if (arch.endsWith("arm")) {
    linkername = "linker";
  } else {
    linkername = "linker64";
    LogPrint("arm64 is not supported yet!");
  }

  var symbols = Module.enumerateSymbolsSync(linkername);
  for (var i = 0; i < symbols.length; i++) {
    var symbol = symbols[i];
    if (symbol.name.indexOf("call_constructor") !== -1) {
      call_constructor_addr = symbol.address;
    }
  }

  // 如果找到call_constructor符号
  if (call_constructor_addr.compare(NULL) > 0) {
    console.log("get construct address");
    Interceptor.attach(call_constructor_addr, {
      onEnter: function (args) {
        if(moduleName){
          const targetModule = Process.findModuleByName(moduleName);
          if (targetModule !== null) {
            // 模块加载的第一时间，早于目标SO的init_proc 和 init_array
            console.log("hook:"+moduleName);
            // 找到了之后阻止后续寻找工作
            moduleName = null;
            hook_ABB0();
          }
        }

      },
      onLeave: function (retval) {

      }
    });
  }
}


function hook_ABB0(){
  var base_addr = Module.findBaseAddress("libJDMobileSec.so");
  var abb0_addr = base_addr.add(0xABB1);
  Interceptor.attach(abb0_addr, {
    onEnter: function (args) {

    },
    onLeave: function (retval) {
```

```
        console.log(retval.readUtf8String());
    }
  });


}
hookInit("libJDMobileSec.so")
```

它比较特殊的地方就是密文是浮点数组成的数组，一个四字节的 Float 解密后对应于一字节明文，复写解密函数如下

Python 复制代码

```python
import idc


def decrypt(address, length):
    '''

    :param address: 解密起始地址
    :param length: 待解密的明文长度
    :return:
    '''
    plainText = ""
    for i in range(length):
        targetAddress = address + i * 4
        encrypt_f = idc.GetFloat(targetAddress)
        decrypt_c = (((int(encrypt_f + encrypt_f)) ^ 0xde) + 0x22) & 0xff
        plainText += chr(decrypt_c)

    print(plainText)
```

我们可以拿上面的 `sub_ABB0(flt_26E70, 20)` 做测试，运行环境是 IPyIDA。

```
Jupyter QtConsole 5.2.2
Python 3.7.6 (tags/v3.7.6:43364a7ae0, Dec 19 2019, 00:42:30) [MSC v.1916 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.31.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import idc
   ...:
   ...:
   ...: def decrypt(address, length):
   ...:     '''
   ...:
   ...:     :param address: 解密起始地址
   ...:     :param length: 待解密的明文长度
   ...:     :return:
   ...:     '''
   ...:     plainText = ""
   ...:     for i in range(length):
   ...:         targetAddress = address + i * 4
   ...:         encrypt_f = idc.GetFloat(targetAddress)
   ...:         decrypt_c = (((int(encrypt_f + encrypt_f)) ^ 0xde) + 0x22) & 0xff
   ...:         plainText += chr(decrypt_c)
   ...:
   ...:     print(plainText)
   ...:

In [2]: decrypt(0x26e70, 20)
ro.build.version.sdk
```

完整流程应该分四步走

- 通过交叉引用获取所有调用解密函数的上层函数
- 获取密文首地址以及其长度
- 调用 decrypt 函数
- 明文以注释或回填或其他某种方式予以展示，增强静态分析时的体验

第一步

Python 复制代码

```python
import idautils
xrefs = idautils.CodeRefsTo(0xabb0, 0)
print(list(xrefs)[:3])
# 输出
# [20844, 20862, 20910]
```

第二步，做匹配，我们同样需要先运行前面的脚本，将数据打散成 byte，以方便匹配。

Python 复制代码

```python
import ida_bytes
import ida_idaapi
import idautils
import idc

start = 0
end = 0
for seg in idautils.Segments():
    name = idc.get_segm_name(seg)
    if name == ".data":
        start = idc.get_segm_start(seg)
        end = idc.get_segm_end(seg)
for address in range(start, end):
    ida_bytes.del_items(address, 0, 1)
    ida_bytes.create_data(address, 0, 1, ida_idaapi.BADADDR)
```

大家可能有两个困惑

- 这里明明是一堆 float，转成 byte 合适吗?

  没关系，我们只是为了方便匹配地址，格式全部统一成下面这样，美滋滋。

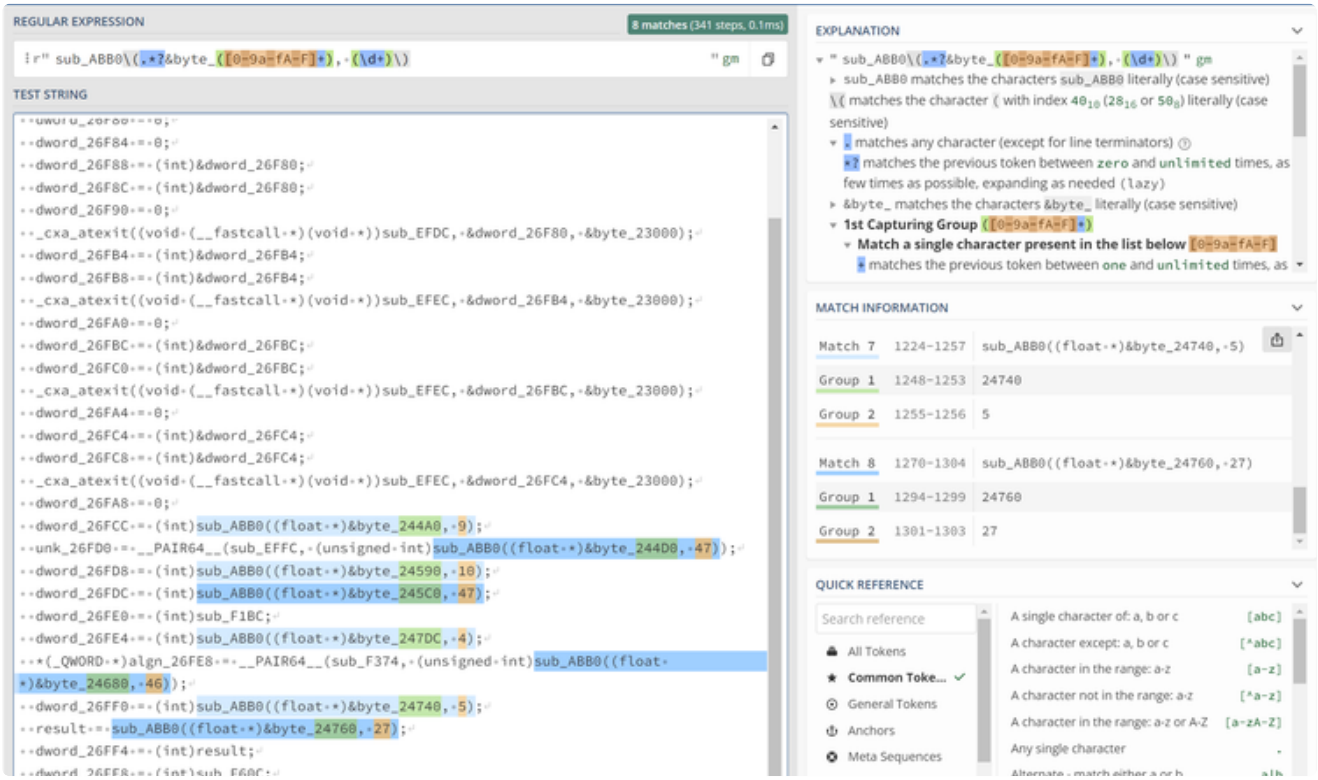
Python 复制代码

```python
sub_ABB0((float *)&byte_26E70, 20);
```

- 这种转化肯定很多是误识别，影响其他数据了啊

  没关系，解密结束后再 undefine 重新分析不就行了

下面进行匹配，正则表达式 `sub_ABB0\(.*?&byte_([0-9a-fA-F]+), (\d+)\)`



第三步没什么好说的，第四步我选择将明文 patch 到密文原来的空间里，这不是一个好选择，读者可以尝试其他方案。

整合脚本如下

Python | 复制代码

```python
import re
import ida_idaapi
import idaapi
import idc
import ida_auto
import ida_bytes
import idautils


def init():
    start = 0
    end = 0
    for seg in idautils.Segments():
        name = idc.get_segm_name(seg)
        if name == ".data":
            start = idc.get_segm_start(seg)
            end = idc.get_segm_end(seg)
    for address in range(start, end):
        ida_bytes.del_items(address, 0, 1)
        ida_bytes.create_data(address, 0, 1, ida_idaapi.BADADDR)


def final():
    for i in idautils.Segments():
        seg = idaapi.getseg(i)
        segName = idaapi.get_segm_name(seg)
        if "data" in segName:
            startAddress = seg.start_ea
            endAddress = seg.end_ea
            ida_bytes.del_items(startAddress, 0, endAddress)
            ida_auto.plan_and_wait(startAddress, endAddress)


def decrypt(address, length):
    '''

    :param address: 解密起始地址
    :param length: 待解密的明文长度
    :return:
    '''
    plainText = ""
    for i in range(length):
        targetAddress = address + i * 4
        encrypt_f = idc.GetFloat(targetAddress)
        decrypt_c = (((int(encrypt_f + encrypt_f)) ^ 0xde) + 0x22) & 0xff
        plainText += chr(decrypt_c)
        ida_bytes.patch_byte(address + i, decrypt_c)
    # # 在明文末尾按照C字符串的风格添加终止符
    ida_bytes.patch_byte(address + length, 0)
    # # 然后呢，在目标地址处创建字符串，参数1起始地址，参数2是长度，参数3是字符串格式，我
    idaapi.create_strlit(address, length, idaapi.STRTYPE_TERMCHR)
    return plainText
```

```
53
54
55  def matchArgs(xrefaddress):
56      cfun = idaapi.decompile(xrefaddress)
57      codeStr = str(cfun)
58      argsMatch = []
59      callList = re.findall("sub_ABB0\(.*?&byte_([0-9a-fA-F]+), (\d+)\)", codeStr)
60      for one in callList:
61          argsprint = []
62          argsprint.append(int(one[0], 16))
63          argsprint.append(int(one[1], 10))
64          argsMatch.append(argsprint)
65      return argsMatch
66
67  init()
68  xrefs = idautils.CodeRefsTo(0xABB0, 0)
69  funcList = []
70  addressList = []
71  for xref in list(xrefs):
72      # 尝试反编译交叉引用所涉及到的函数，decompile方法需要传入地址，就会反编译其所属函数
73      # 一些函数反编译会失败，所以需要加个异常处理
74      try:
75          funcName = idc.get_func_name(xref)
76          if funcName not in funcList:
77              funcList.append(funcName)
78              argsMatch = matchArgs(xref)
79              for one in argsMatch:
80                  address = one[0]
81                  if address not in addressList:
82                      addressList.append(address)
83                      length = one[1]
84                      plaintext = decrypt(address, length)
85                      print(plaintext)
86                  else:
87                      pass
88          else:
89              pass
90      except:
91          pass
```

打印效果不错，但有部分字符串未被顺利识别，原因是什么？如何避免这种情况？我想把这作为思考题留给读者。除此之，你或许会意识到一个问题，如果你想将明文注释在函数调用处，这是较难实现的事，因为正则对伪代码的匹配并不包含地址或任意其他信息，它只是作为单纯的文本处理。

三、其他内容

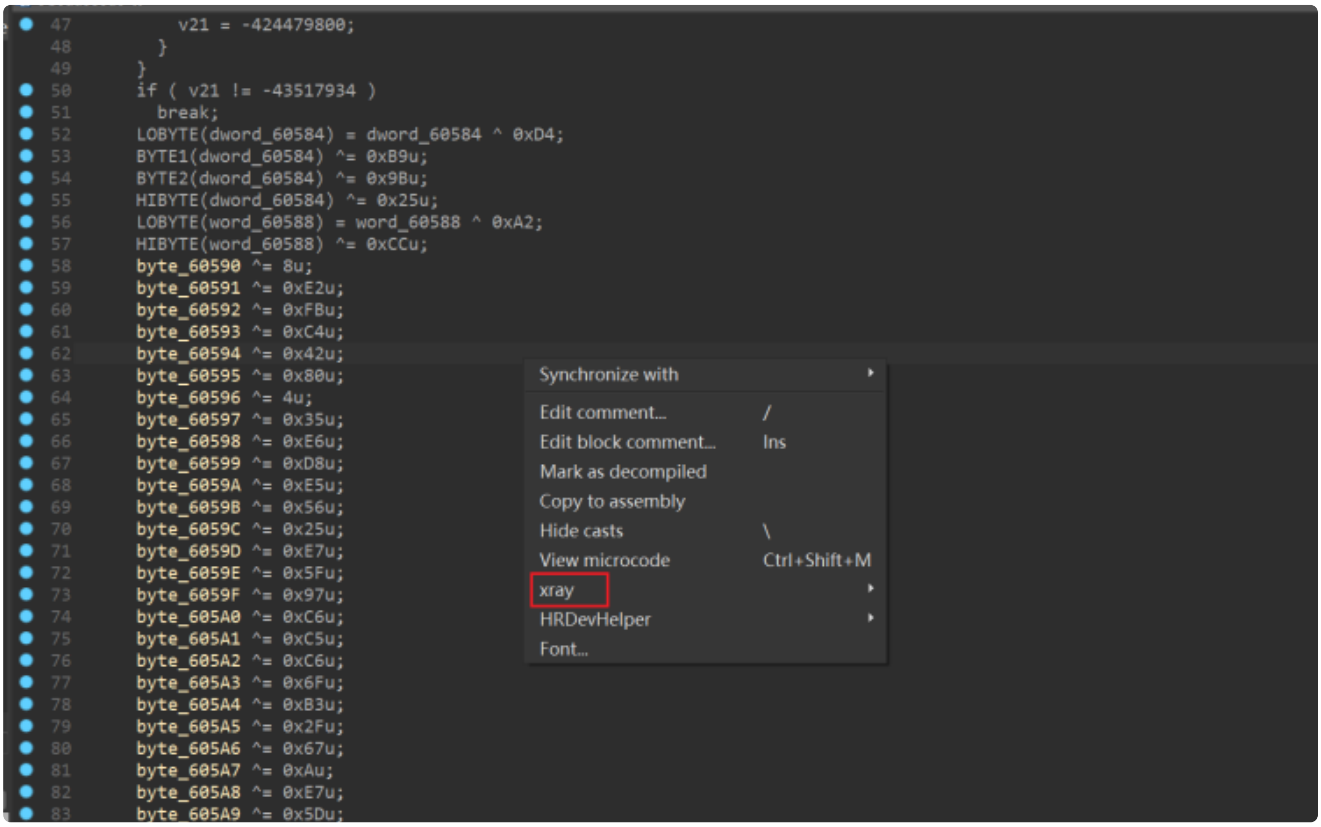
本节主要讨论的内容是通过正则表达式匹配所需内容。在测试正则表达式，我们需要将IDA反编译界面生成的伪代码文本拷贝出来，用python re 模块进行测试，或者像文中那样，使用 regex101 或类似的工具。但如果你不喜欢 regex101，而且希望用正则表达式做匹配的过程可以和 IDA 结合的更紧密，那么你可以尝试 xray <https://github.com/patois/xray> 这个插件，下面简单介绍它的安装和使用。

3.1 对 xray 的介绍

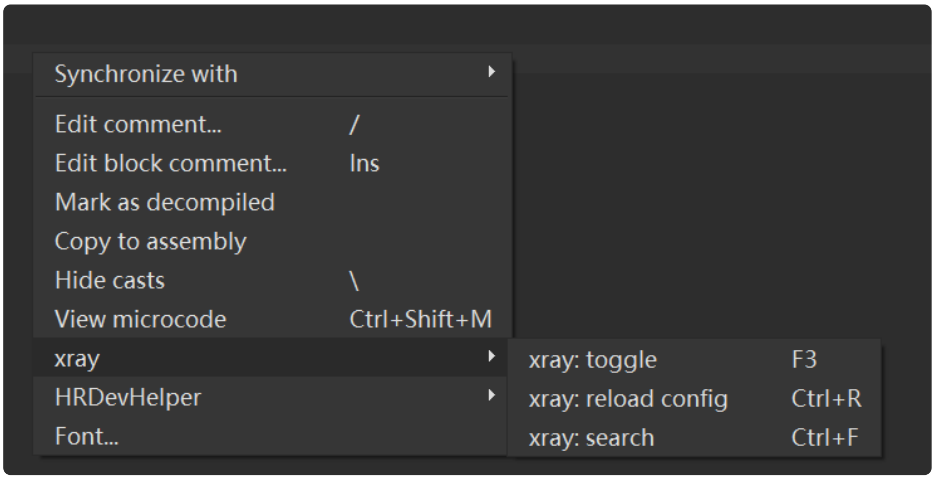
xray 旨在让 IDA 可以在反编译代码界面做正则匹配，并仅显示匹配到的内容，或对此部分进行着色。它的安装步骤就像其他插件一样，将 xray.py 放到 plugin 目录下即可。

3.2 xray 的使用

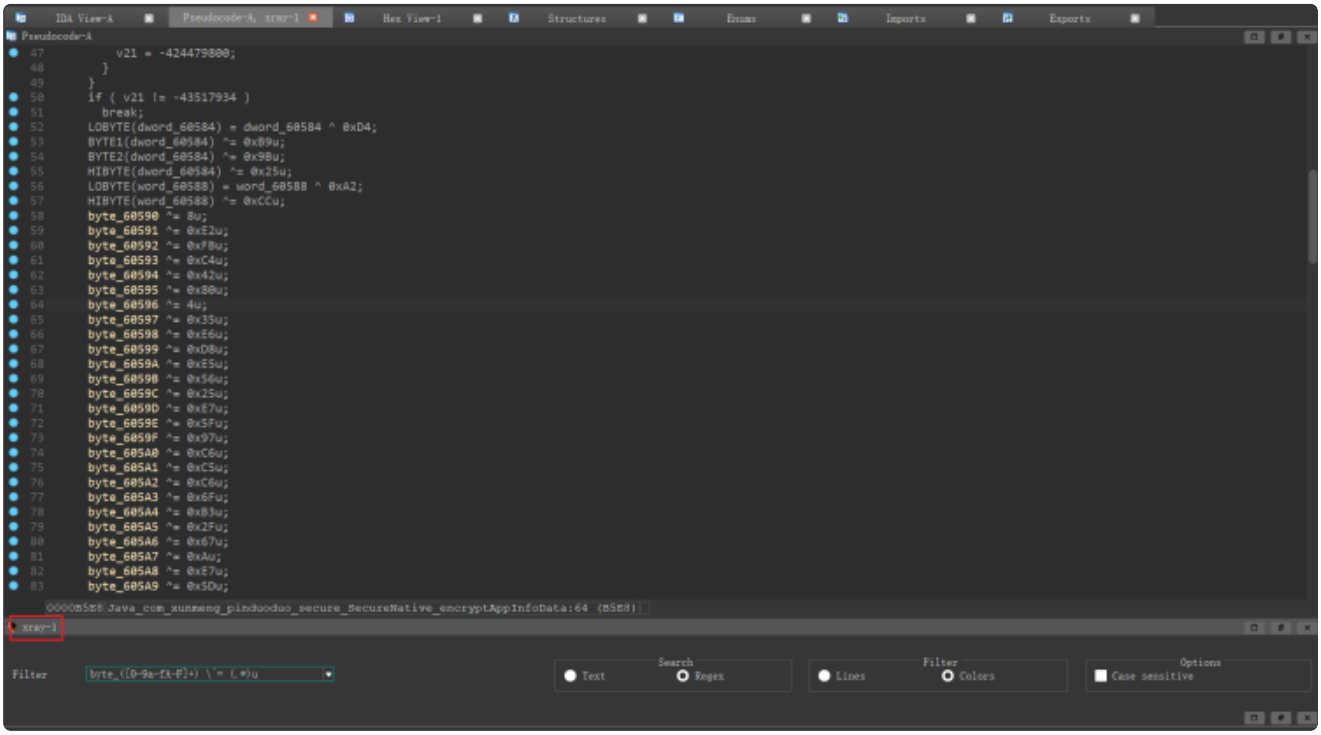
顺利安装后，当在反编译界面右键时，会出现 xray 子菜单。



这里只介绍第三个子选项—— search ，其余两个选项是激活和持久化，和我们的关系不是特别大。



点击它，出现 xray 窗口



最左侧用于输入匹配规则，右侧有三个选项。

选项一是匹配模式，可以是简单的文本匹配或复杂的正则匹配

选项二是展示方式，即被匹配的内容如何展示，Lines 意味着反编译界面只显示匹配到的那些行，其余伪代码行会隐藏；Colors 意味着会将匹配行进行染色处理，比如上图中明黄色显示的行就是匹配的内容。

选项三是匹配是否要求大小写一致。

xray 的基本使用就是这些。

四、总结

我们通过正则匹配获取函数或运算的具体参数，格式并不完全统一，因为样本会存在编译优化，这是问题一。IDA 会做数据识别与展示上的优化，当 IDA 的这种识别和优化发生偏差时，我们要对它做纠正，这是问题二。

我还想再一次讨论匹配作用于汇编层面还是伪代码层面比较好的问题。

- 基于汇编，匹配会比较复杂，需要兼容很多情况，好处是运行非常快，对于大型二进制文件或时间敏感型的需求很必要。
- 基于伪代码搞，匹配会轻松简单很多，但因为要逐个函数做反编译，所以速度会相对慢，而且函数无法 F5 的情况中不可用。

本篇大致就讲了这么些内容。