对函数的静态分析（一）

一、引言

函数是一个极其重要的概念，在开发中它是功能构建的最小单元，在逆向中通过分析重点函数实现对程序的理解和算法的还原。

关于函数概念本身，以及其内涵和外延，可以做出无数的讨论。很多新手会有一种错误认知，觉得这些理论属于学院派的自娱自乐，对逆向实践没有指导意义，这是大错特错。理论没有用的原因只有两个，1 是没有在当前场景上运用合适的理论，2 是对理论的理解不够扎实，没有产生可以解决现实问题的质变。

《对函数的XXX》一系列文章，是对此的粗浅讨论，这是第一篇。

二、问题分析

一个正常的二进制文件里，可能包含成千上万个函数。刚拿到样本时，我们对它的了解很少，这时候就会希望通过某种方式，快速了解样本里的关键函数，就像绝大多数问题，它也分为静态和动态两种解法。

在静态分析时，可以从很多角度去打量一个函数，这包括函数的大小、被引用的次数、算数指令的占比等等，这种观测可以帮助我们找到最关键的那些函数——比如复杂的运算、加密算法、其他重要的程序逻辑。

三、基本分析

3.1 函数的大小

一个大的函数相较于一个小的函数，往往包含了更多更复杂的内部逻辑，这显而易见。文件解析，网络协议，加密算法等等逻辑，它们都很大。那么如何判断一个函数大不大，以及找出最大的那些函数呢，这是本节所关注的问题。

回忆一下在逆向分析样本中，什么时候我们会觉得某个函数很大？这种直观感受往往来自于反编译过程——F5 耗时很久，以及反编译后函数的伪代码有数千甚至上万行。因此，我们可以遵从这种直观的感受，使用反编译的耗时或伪代码的行数来衡量函数体的大小，然后排序找出最大的那部分函数，比如打印前 10 或前 5% 的大函数，然后对它们做进一步分析 。

但实话实说，这并不是一个好主意。对于函数粒度的各种实践，有一个直白且实用的准则——如果反汇编和反编译这两个 level 都能处理某件事，那么只要在反汇编层处理它的复杂度不超过在反汇编层太多，就优先选择反汇编层。原因很多，包括到不限于

- 反编译是相对耗时的操作，在 IDA 中对一个中型样本的全体函数做遍历反编译，可能需要几十分钟，花这么长时间，仅仅得知这个样本中哪几个函数特别大，显然谈不上什么性价比。
- 在存在花指令等代码混淆技术的样本中，许多函数都没办法顺利反编译。
- 许多逆向框架或工具都提供反编译的功能，但因为反编译是一项非常难的技术，所以实际好用的没几个，而反汇编的难度相对来说十分低，很多逆向框架或工具都能处理好这件事。因此我们在 IDA 或某种工具上，基于反汇编 level 所作的操作和实践，可以很方便的迁移到其他工具上良好复现，但基于反编译的操作却不一定行，因为其他工具很可能没法提供和 IDA、Ghidra 一样良好的反编译效果。

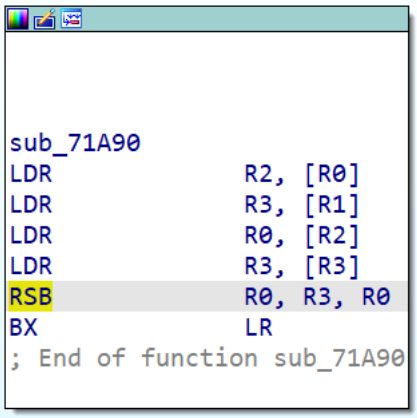
所以我们要将衡量函数大小这件事，放到反汇编层或者说汇编层去处理。比如函数的汇编代码行数，就是和“反编译后函数伪代码的长度“几乎等价的指标。除此之外，函数中基本块的数量、函数占多少字节空间、函数所开辟栈空间的大小、创建多少局部变量等等，都是和”函数大小“这一概念高度正相关的概念，只需要任选其一即可。

比如下面这个函数

Shell | 复制代码

```
1  .text:00071A90                         sub_71A90
2  .text:00071A90
3  .text:00071A90 00 20 90 E5                         LDR             R2, [R0]
4  .text:00071A94 00 30 91 E5                         LDR             R3, [R1]
5  .text:00071A98 00 00 92 E5                         LDR             R0, [R2]
6  .text:00071A9C 00 30 93 E5                         LDR             R3, [R3]
7  .text:00071AA0 00 00 63 E0                         RSB             R0, R3,
8  .text:00071AA4 1E FF 2F E1                         BX              LR
```

控制流图如下



```
sub_71A90
LDR             R2, [R0]
LDR             R3, [R1]
LDR             R0, [R2]
LDR             R3, [R3]
RSB             R0, R3, R0
BX              LR
; End of function sub_71A90
```

函数汇编代码行数是 5 行。

函数所占空间是 20 字节。

函数的基本块数量是 1 个。

函数没有开辟栈空间，也没有局部变量。

在 IDA 的函数窗口里，length 就是函数所占空间，我们可以点击它，就会以升序/降序的方式对函数做排列。

Functions window

Function name	Segment	Start	Length
.datadiv_decode3057749081888972547	.text	00055AB4	00004770
.datadiv_decode12614845752875866503	.text	0004DC88	00002512
sub_62D16	.text	00062D16	00001C5C
sha512_block_data_order	.text	00088BA0	000015F8
sha1_block_data_order	.text	00085700	00001494
sub_B07C8	.text	000B07C8	00001340
DES_encrypt1	.text	0010F410	00001298
DES_encrypt2	.text	001106A8	000011BC
sha256_block_data_order	.text	00086CC0	00000FE4
asn1_template_print_ctx	.text	000BB658	00000F74
.datadiv_decode11761207327020436985	.text	0005B2F4	00000F14
.datadiv_decode4809734306549985915	.text	0004B2EC	00000DE2
X509_verify_cert	.text	000C3E34	00000DDC
sub_FFFBC	.text	000FFFBC	00000DD8
sub_FEF50	.text	000FEF50	00000DBC

但我这里还是想用脚本做一下这件事，1 是多熟悉几个 IDAPython API，2 是看一下IDAPython 原生 API 、Sark、BIP 三者在做这件事上，方便以及 pythonic 程度的比较。

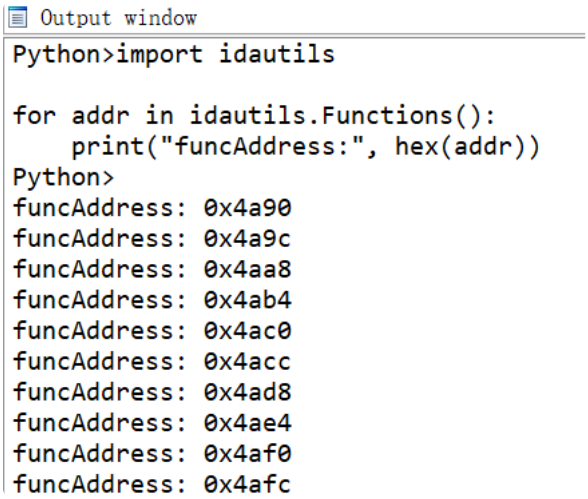
首先我们要遍历当前样本的所有函数

`idautils.Functions(start=None,end=None)` 是 `idautils` 提供的实用 API 之一，它会返回一个列表，包含 start 到 end 范围内所有函数的起始地址，不传参则默认遍历当前样本的所有函数。

Python 复制代码

```python
import idautils

for addr in idautils.Functions():
    print("funcAddress:", hex(addr))
```

运行如下



可以发现，它按照地址从低到高做遍历。

Python 复制代码

```python
import idaapi
import idautils

for addr in idautils.Functions():
    print("funcAddress:", hex(addr))
    # funcName = idc.get_func_name(addr)
    funcName = idaapi.get_ea_name(addr)
    print("funcName:", funcName)
```

`idc.get_func_name(ea)` 和 `idaapi.get_ea_name(ea)` 在此处都可以获取到函数名，但读者需要注意，两者有区别，以下面的代码片段为例。

```
.text:00008E92 sub_8E92                                ; CODE XREF: sub_8E80+A
.text:00008E92
.text:00008E92 var_4           = -4
.text:00008E92
.text:00008E92 ; __unwind {
.text:00008E92                 SUB             SP, SP, #4
.text:00008E94                 STR             R0, [SP,#4+var_4]
.text:00008E96                 LDR             R0, [SP,#4+var_4]
.text:00008E98                 ADD             SP, SP, #4
.text:00008E9A                 BX              LR
.text:00008E9A ; } // starts at 8E92
```

IPyIDA 中分别测试这两个 API

```
import idc
import idaapi

idaapi.get_ea_name(0x8e92)
Out[3]: 'sub_8E92'

idaapi.get_ea_name(0x8e94)
Out[4]: ''

idc.get_func_name(0x8e92)
Out[5]: 'sub_8E92'

idc.get_func_name(0x8e94)
Out[6]: 'sub_8E92'
```

`get_ea_name` 获取当前地址所对应的符号名，因此如果传入函数的首地址，会返回函数名。但如果传入的不是函数首地址，就无法顺利返回符号名。 `get_func_name` 会返回当前地址所属函数的名字。看起来 `get_func_name` 更好用，但它只可以获取函数名，而 `get_ea_name` 可以处理非函数区域的命名，比如各种类型数据的名字，像下面这样。

```
.text:0001052C off_1052C       DCD aThrow_0 - 0x1051C  ; DATA XREF: sub_1050C+6
```

代码演示

```
idaapi.get_ea_name(0x1052C)
Out[8]: 'off_1052C'
```

因此两个 API 各有侧重，但在此处都可以使用。

```python
import idaapi
import idautils

for addr in idautils.Functions():
    print("funcAddress:", hex(addr))
    # funcName = idc.get_func_name(addr)
    funcName = idaapi.get_ea_name(addr)
    print("funcName:", funcName)
    func = idaapi.get_func(addr)
    length = func.size()
    print("length:", length)
```

`idaapi.get_func(ea)` 用于获取函数， `ea` 可以是函数内的任意地址。 `size()` 是函数的一个方法，获取函数的长度。接下来做排序和打印

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

functionList = []
for addr in idautils.Functions():
    # funcName = idc.get_func_name(addr)
    funcName = idaapi.get_ea_name(addr)
    func = idaapi.get_func(addr)
    length = func.size()
    oneFuncDict = {"funcName": funcName, "Address": hex(addr), "length": lengt
    functionList.append(oneFuncDict)

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

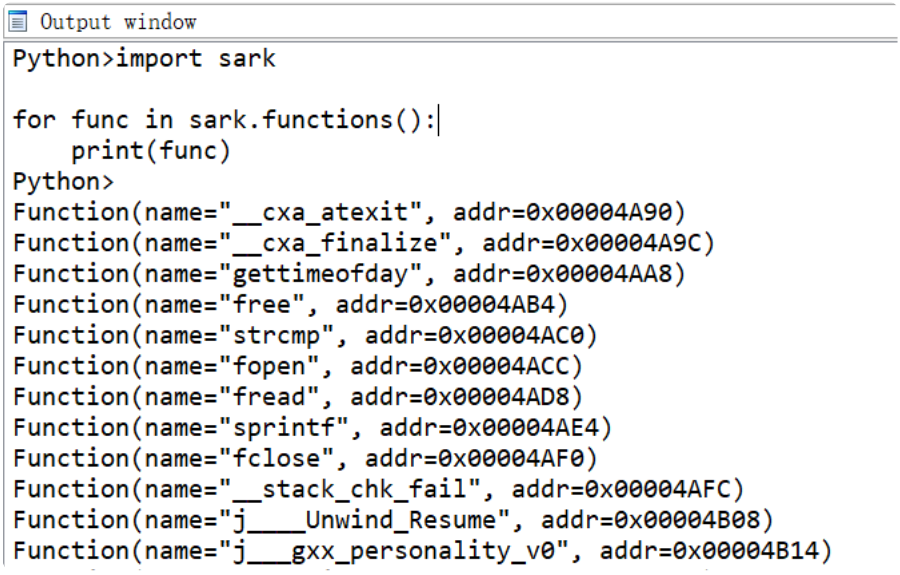
运行结果符合预期

```shell
{'funcName': 'sub_75CC', 'Address': '0x75cc', 'length': 8760}
{'funcName': 'JNI_OnLoad', 'Address': '0x56bc', 'length': 7950}
{'funcName': 'sub_1428C', 'Address': '0x1428c', 'length': 5072}
{'funcName': 'sub_E5C8', 'Address': '0xe5c8', 'length': 2560}
{'funcName': 'sub_C0DC', 'Address': '0xc0dc', 'length': 2316}
{'funcName': 'sub_AC58', 'Address': '0xac58', 'length': 2180}
{'funcName': 'sub_B73C', 'Address': '0xb73c', 'length': 2118}
{'funcName': 'sub_16200', 'Address': '0x16200', 'length': 1656}
{'funcName': 'sub_109C8', 'Address': '0x109c8', 'length': 1332}
{'funcName': 'sub_116EC', 'Address': '0x116ec', 'length': 1332}
{'funcName': 'sub_13BC4', 'Address': '0x13bc4', 'length': 1240}
{'funcName': 'sub_12BBC', 'Address': '0x12bbc', 'length': 1150}
{'funcName': 'sub_169CC', 'Address': '0x169cc', 'length': 1082}
{'funcName': 'sub_1AD94', 'Address': '0x1ad94', 'length': 1008}
{'funcName': 'sub_F71C', 'Address': '0xf71c', 'length': 948}
{'funcName': 'sub_17164', 'Address': '0x17164', 'length': 896}
{'funcName': '__gnu_unwind_execute', 'Address': '0x1b72c', 'length': 892}
{'funcName': '_Unwind_VRS_Pop', 'Address': '0x1b1a0', 'length': 856}
{'funcName': 'sub_15B24', 'Address': '0x15b24', 'length': 854}
{'funcName': 'sub_13790', 'Address': '0x13790', 'length': 698}
```

如果使用 `Sark` 会是怎样?

```python
import sark

for func in sark.functions():
    print(func)
```

`sark.functions()` 是对 `idautils.Functions` 的封装，返回范围内的所有函数而非函数起始地址。



Sark 基于面向对象的设计，相比较原生 IDAPython API ，使用体验好很多。

```python
import sark

for func in sark.functions():
    funcName = func.name
    length = func.func_t.size()
    startAddr = func.start_ea
    oneFuncDict = {"funcName": funcName, "Address": hex(startAddr), "length": 
    print(func)
```

需要注意，底层依赖于 IDAPython，比如 `func.name` 来自于 `get_ea_name` 。

```python
@property
def name(self):
    """Function's Name"""
    return idaapi.get_ea_name(self.start_ea)
```

排序的步骤和先前一样，这里不做讨论。再看 BIP，它作为更新更成熟的工具，在使用体验上类似于 Sark 再进一步。

```python
from bip import BipFunction

for func in BipFunction.iter_all():
    funcName = func.name
    length = func.size
    startAddr = func.ea
    oneFuncDict = {"funcName": funcName, "Address": hex(startAddr), "length": 
    print(func)
```

其中 `func.size` 的实现如下

```python
@property
def size(self):
    return self._funct.size()
```

从开发效率和使用体验上看，我建议大家直接使用 BIP。但因为绝大多数 IDA 脚本和插件都采用原生 API 开发，只有少部分用 Sark 开发，极少部分用 BIP 开发，所以为了方便大家阅读其他项目的代码，我们以写脚本时仍然以原生 IDAPython API 为主，辅之以 BIP 的版本。

3.2 函数的引用数

做研究的朋友应该都有体会，在对某个领域做文献综述时，我们必须要看被引用数较多的那批论文，它们往往是这个领域的奠基之作或有重要贡献。在函数分析上，道理也相通。如果某个函数被引用了很多次，那么它必然是某种类型的重要函数，比如工具函数——内存拷贝、字节比对、字符串解密等等。寻找到样本中高频引用的这些函数，做静态分析或动态Hook，对窥探和理解程序的执行流有很大帮助。

引用存在许多概念，引用和被引用，代码引用和数据引用等等，这里我们不做过度讨论。看IDA脚本的具体实现

getMostConnectedFUnctions Python 复制代码

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

functionList = []
for func in idautils.Functions():
    xrefs = idautils.CodeRefsTo(func, 0)
    xrefCount = len(list(xrefs))
    oneFuncDict = {"funcName":idc.get_func_name(func), "Address": hex(func), '
    functionList.append(oneFuncDict)

function_list_by_countNum = sorted(functionList, key=itemgetter('xrefCount'),r
for func in function_list_by_countNum[:10]:
    print(func)
```

应该说没什么需要特别注意的，看一下 BIP 的代码，和预期一样优雅流畅。

Python 复制代码

```python
from bip import BipFunction
from operator import itemgetter
functionList = []
for func in BipFunction.iter_all():
    oneFuncDict = {"funcName": func.name, "Address": hex(func.ea), "xrefCount'
    functionList.append(oneFuncDict)

function_list_by_countNum = sorted(functionList, key=itemgetter('xrefCount'),r
for func in function_list_by_countNum[:10]:
    print(func)
```

在查找字符串解密函数这件事上，这个脚本特别好用，效果类似下面这样。

Shell | 复制代码

```
{'funcName': 'free', 'Address': '0x4ab4', 'xrefCount': 250}
{'funcName': 'sub_ABB0', 'Address': '0xabb0', 'xrefCount': 239}
{'funcName': 'strlen', 'Address': '0x4b2c', 'xrefCount': 31}
{'funcName': 'malloc', 'Address': '0x4b5c', 'xrefCount': 30}
{'funcName': 'j__ZdlPv', 'Address': '0x4cc4', 'xrefCount': 27}
{'funcName': 'fclose', 'Address': '0x4af0', 'xrefCount': 25}
{'funcName': 'j___cxa_end_cleanup', 'Address': '0x4dfc', 'xrefCount': 25}
{'funcName': '__errno', 'Address': '0x4c34', 'xrefCount': 23}
{'funcName': 'sub_179D0', 'Address': '0x179d0', 'xrefCount': 23}
{'funcName': 'sub_17A38', 'Address': '0x17a38', 'xrefCount': 23}
```

3.3 函数指令占比

加解密函数是逆向分析中一类重要的函数，研究人员发现，这类函数在实现中会大量使用运算类指令，因此如果一个函数的运算类指令占比很高，那么很可能是加解密函数，标准算法或自定义加密都符合这个规律。

在代码混淆十分严重的当下，它的作用被削弱了不少，因为控制流平坦化等混淆技术就会带来大量的指令运算，所以我们这里只是简单的处理一下。

getEorMostFunctions Python 复制代码

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

functionList = []
for addr in list(idautils.Functions()):
    funcName = idc.get_func_name(addr)
    func = idaapi.get_func(addr)
    length = func.size()
    dism_addr = list(idautils.FuncItems(addr))
    count = 0
    if length > 0x10:
        for line in dism_addr:
            m = idc.print_insn_mnem(line)
            if m.startswith("LSL") | m.startswith("AND") | m.startswith("ORR")
                    "ROR"):
                count += 1

        oneFuncDict = {"funcName": funcName, "Address": hex(addr), "rate": cou
        functionList.append(oneFuncDict)

function_list_by_countNum = sorted(functionList, key=itemgetter('rate'), rever

for func in function_list_by_countNum[:20]:
    print(func)
```

其中使用到了 `idautils.FuncItems(start)` ，它返回一个列表，内容是 `start` 所指向的函数中，每一项的地址。这个说法可能有一些晦涩，下面结合示例做讨论。

Python | 复制代码

```
.text:0000E438 sub_E438                                ; CODE XREF: sub_CA58+5
.text:0000E438                                         ; sub_D08C+B56↑p
.text:0000E438
.text:0000E438 var_18          = -0x18
.text:0000E438
.text:0000E438 ; __unwind {
.text:0000E438                 PUSH            {R4-R7,LR}
.text:0000E43A                 ADD             R7, SP, #0xC
.text:0000E43C                 PUSH.W          {R11}
.text:0000E440                 SUB             SP, SP, #8
.text:0000E442                 ADD.W           R0, R0, #0x170
.text:0000E446                 MOV             R5, R1
.text:0000E448                 MOVS            R1, #0x1C
.text:0000E44A                 MOV             R4, R2
.text:0000E44C                 BL              sub_C2F4
.text:0000E450                 MOV             R6, R0
.text:0000E452                 MOV             R0, R5  ; s
.text:0000E454                 BLX             strlen
.text:0000E458                 LDR             R3, [R4]
.text:0000E45A                 LDR             R1, =(asc_15F42 - 0xE460) ; ")"
.text:0000E45C                 ADD             R1, PC  ; ")"
.text:0000E45E                 ADDS            R2, R1, #1
.text:0000E460                 STRD.W          R1, R2, [SP,#0x18+var_18]
.text:0000E464                 ADDS            R2, R5, R0
.text:0000E466                 MOV             R0, R6
.text:0000E468                 MOV             R1, R5
.text:0000E46A                 BL              sub_103D8
.text:0000E46E                 ADD             SP, SP, #8
.text:0000E470                 POP.W           {R11}
.text:0000E474                 POP             {R4-R7,PC}
.text:0000E474 ; End of function sub_E438
.text:0000E474
.text:0000E474 ; ---------------------------------------------------------------
.text:0000E476                 ALIGN 4
.text:0000E478 off_E478        DCD asc_15F42 - 0xE460  ; DATA XREF: sub_E438+2
.text:0000E478 ; } // starts at E438
```

测试代码如下

Python 复制代码

```python
import idautils

items = list(idautils.FuncItems(0xE438))

for item in items:
    print(hex(item))

# 输出
0xe438
0xe43a
0xe43c
0xe440
0xe442
0xe446
0xe448
0xe44a
0xe44c
0xe450
0xe452
0xe454
0xe458
0xe45a
0xe45c
0xe45e
0xe460
0xe464
0xe466
0xe468
0xe46a
0xe46e
0xe470
0xe474
```

我们发现，实际上获取了反汇编界面最左侧的地址列表，也可以理解为函数所包含的每一行数据以及代码的地址。使用 BIP 代码如下，完全避免了这种纠缠，就是优雅。

```python
from bip import BipFunction
from operator import itemgetter
functionList = []
for func in BipFunction.iter_all():
    count = 0
    funcSize = func.size
    if funcSize > 0x10:
        for line in func.instr:
            m = line.mnem
            if m.startswith("LSL") | m.startswith("AND") | m.startswith("ORR"):
                count += 1

        oneFuncDict = {"funcName": func.name, "Address": hex(func.ea), "rate":
        functionList.append(oneFuncDict)

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

读者可以自行将这些方法应用在分析中，应该说有用，但用处不大。这十分合理，因为这些方案都十分朴素、基于经验和直觉，既不高级也不复杂。其中比较靠谱和有效的是高频的交叉引用，对字符串解密函数有很好的检测效果，因为字符串解密往往会调用非常多次。

3.4 导入函数与导出函数

使用 IDAPython 打印所有导出函数的函数名以及地址

```python
import ida_entry

exports = []
n = ida_entry.get_entry_qty()
for i in range(0, n):
    ordinal = ida_entry.get_entry_ordinal(i)
    ea = ida_entry.get_entry(ordinal)
    name = ida_entry.get_entry_name(ordinal)
    exports.append({"funcName": name, "addr": ea})
print(exports)
```

使用 IDAPython 打印所有导入函数的函数名以及地址

Python 复制代码

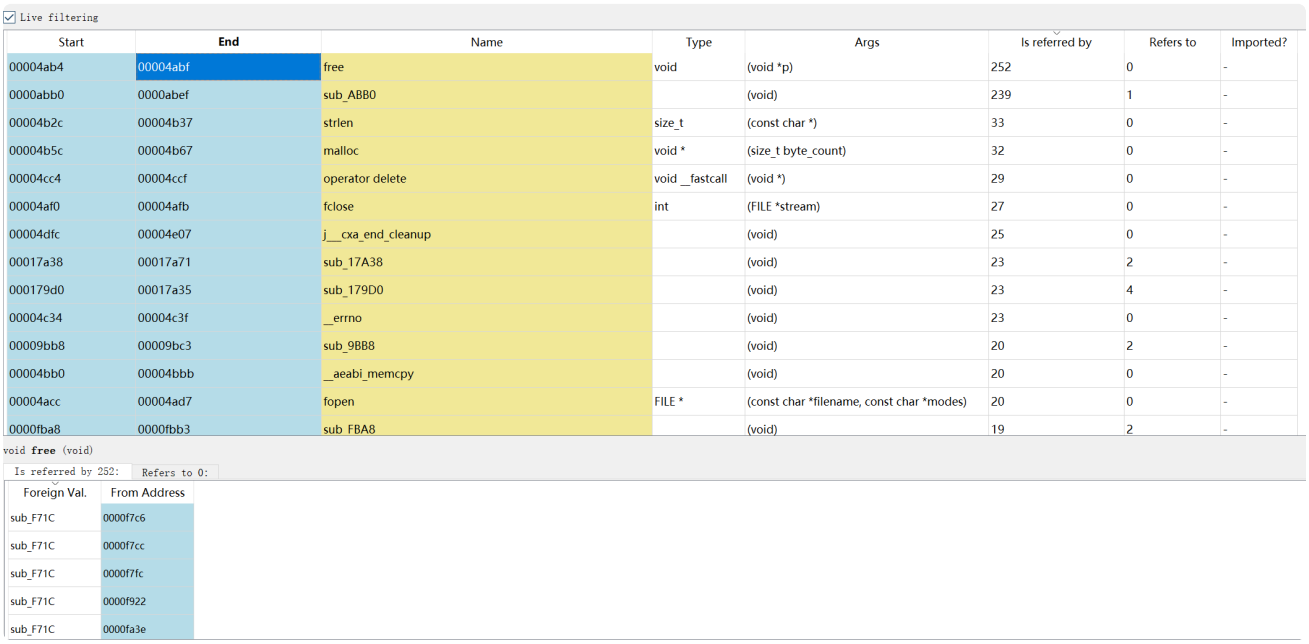
```python
import ida_nalt


def find_imported_funcs():
    def imp_cb(ea, name, ord):
        if not name:
            name = ''
        imports.append({"funcName": name, "addr": ea})
        return True

    imports = []
    nimps = ida_nalt.get_import_module_qty()
    for i in range(0, nimps):
        ida_nalt.enum_import_names(i, imp_cb)

    return imports


imports = find_imported_funcs()
print(imports)
```

四、相关插件介绍

有许多 IDA 插件，和我们做的是相同的事，但要更加好用，让我们来看看这些插件吧。

4.1 IFL

查看高频交叉引用这件事上，我们自己的脚本有两个缺点，1 是没有做成插件，那么每次想使用的时候都要从某个角落里找出这段代码，很不方便。 2 是具体使用后会发现的问题，所统计的高频引用函数打印展示在输出窗口里，IDA 里对这些或其他函数做反编译时会输出各种日志，将高频引用信息盖过去了。因此这个信息最好展示在其他窗口。IFL <https://github.com/hasherezade/ida_ifl> 就是一个旨在增强函数展示效果的 IDA 插件，激活后界面如下

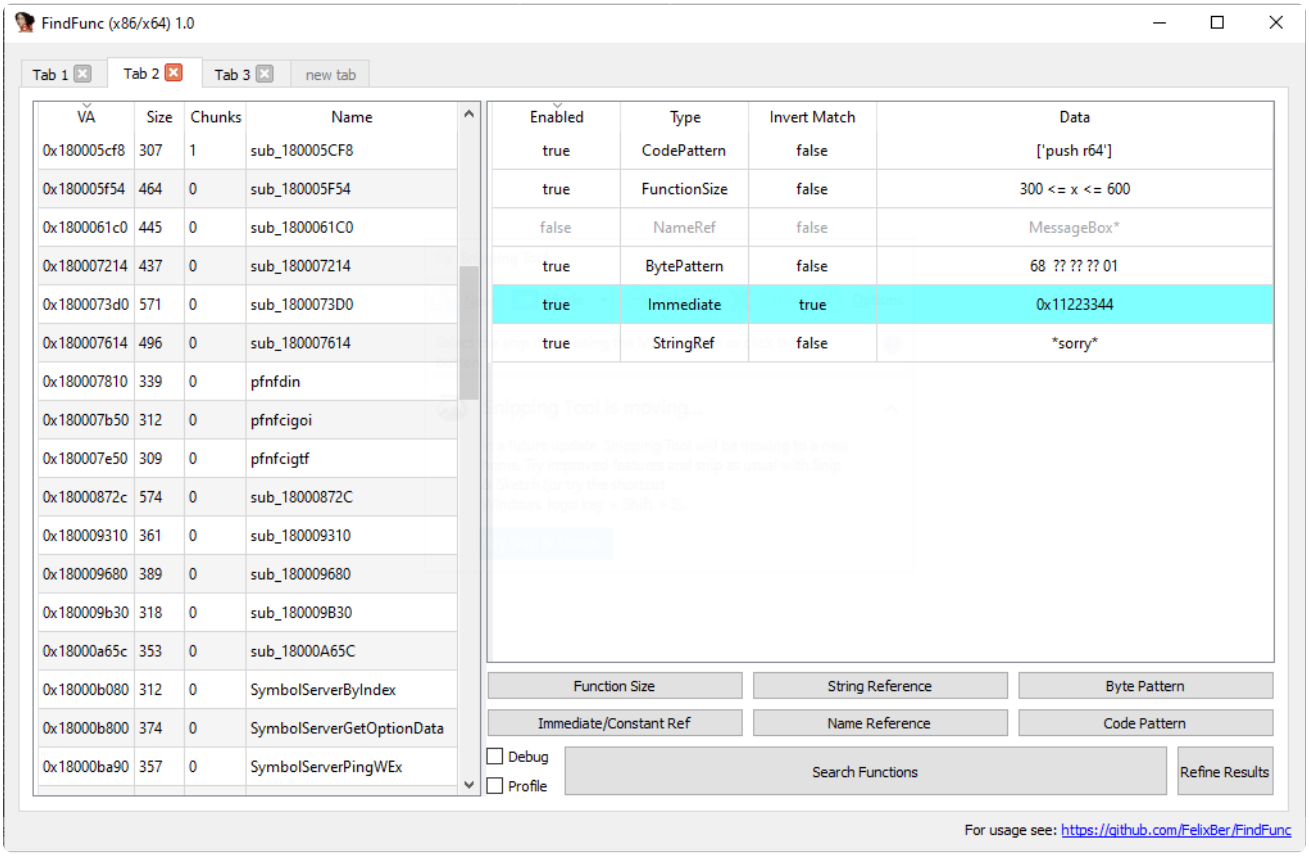


它像是一个更好看的 function View，而且添加了引用和被引用栏，可以排序以及通过底部小窗做查看，感觉不错。

4.2 FindFunc

FindFunc <https://github.com/FelixBer/FindFunc> 这个工具挺有意思，它用于搜索和定位函数，在逻辑上很像内存搜索的那一套。值得一提的是，它还是2022 IDA 插件大赛获奖作品。

首先在界面上就十分好看。



下面描述一下它的使用场景——假设我们要在一个二进制文件中定位某个或某些函数，但它们的特征并不清晰，函数名不存在可检索的特征，函数体也没有固定的代码片段特征，那么你就可以考虑

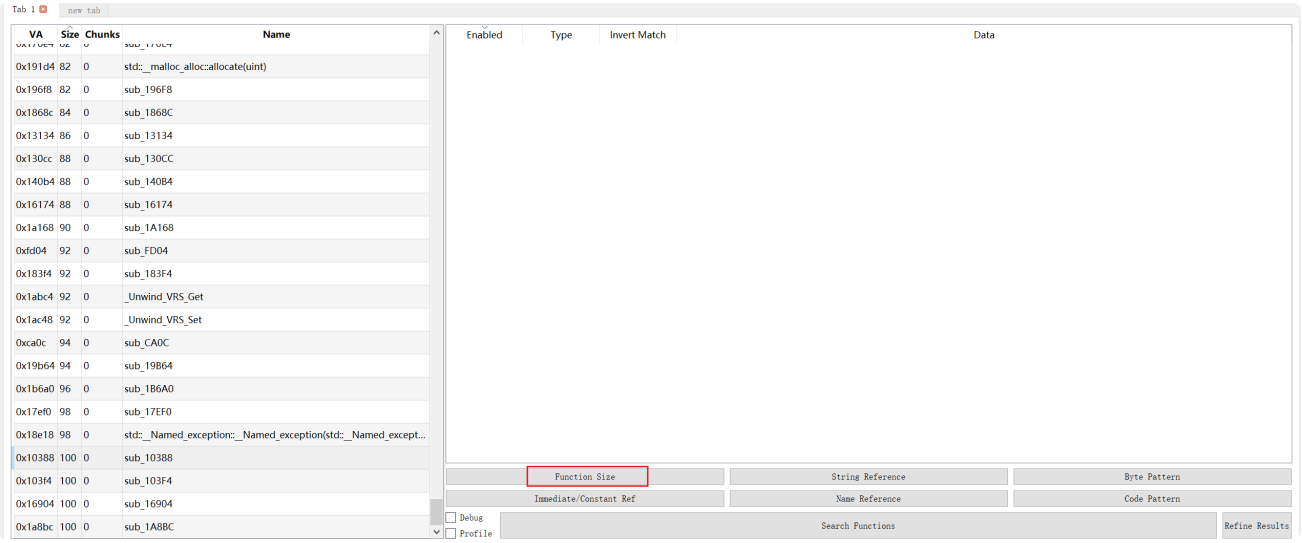
FindFunc。

它允许你从多个维度对函数进行描述，进而约束和搜索。上图是它的工作界面，左侧是符合条件的函数，右侧是条件限定设置。

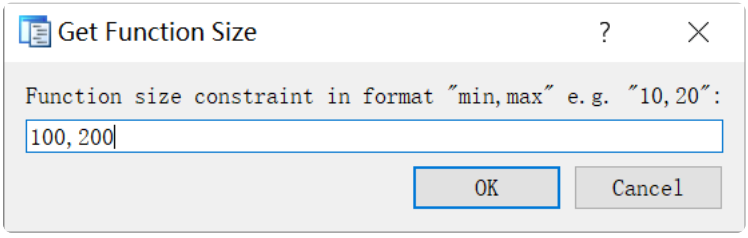
FindFunc 的第一个优点就是这个简洁优雅的工作窗口，它用 PyQt5 做的 GUI。

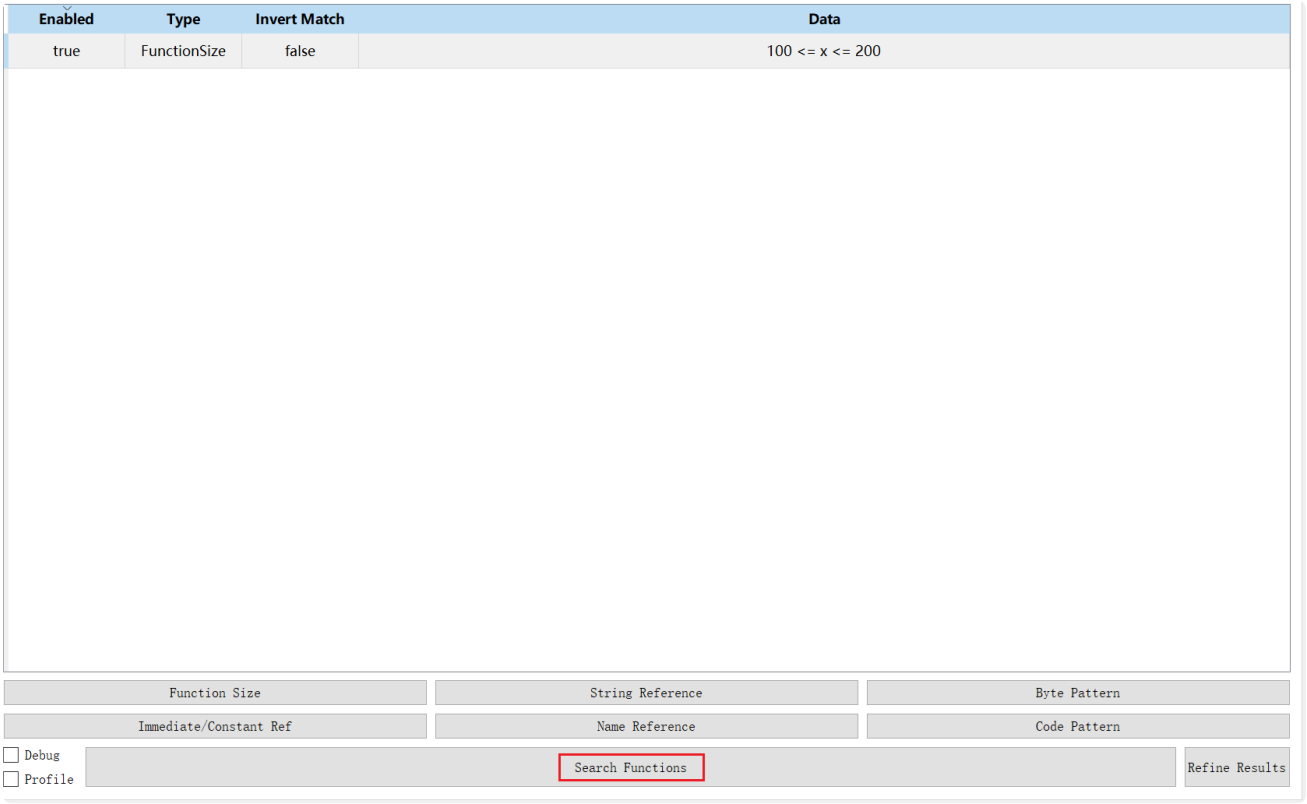
FindFunc 提供了六种对函数的描述或者说约束，下面进行介绍

函数大小，在上文，我们分析过函数大小这一个维度，并且我们认为函数体的字节占用越大，越可能是重要函数，在 FindFunc 中我们可以限制函数的大小在某个区间。



假如限定函数大小为 100 - 200 字节





搜索后左侧就会呈现符合条件的函数。

字符串引用（String Reference），限定函数中出现至少一次对限定字符串的引用，对字符串的描述可以用正则表达式。

立即数/常量引用（Immediate/Constant Ref），我们可以要求函数中出现某个立即数或常数至少一次

符号名/标签引用（Name Reference），限定函数出现至少一次对规定符号名的引用，同样可以用正则，比如限定必须出现 mem*，可以找到调用 memcpy/memcmp 等函数的函数

字节规则（Byte Pattern），和 IDA 的 Search bytes 表现一致，即可以以正则方式匹配字节序列

汇编规则（Code Pattern），这是它的一大亮点，是一种相对灵活的、描述汇编规则的方案，应该说确实不错。比如可以编写如下汇编规则，寻找连续六条汇编所构成的代码片段，规则如下，我们对它逐条做说明。

Python | 复制代码

```
1  mov* any,any
2  mov eax, r32
3  pass
4  mov r64, imm
5  any r64,r64
6  mov
```

`mov* any,any` 此条汇编代码的助记符以 mov 开头，比如 `mov`，`moveq`，`movs`，两个 `any` 意味着它有两个操作数，而且既不限制操作数的类型（寄存器/立即数或其他可以），更不限制具体的值。

`mov eax, r32` 助记符为 `mov` ，第一个操作数是 `eax` 寄存器，第二个操作数是任意 32 位寄存器。

`pass` 不做任何限制，此行可以是任意汇编代码

`mov r64, imm` 助记符是 `mov` ，第一个操作数是任意 64 位寄存器，第二个操作数是立即数

`any r64, r64` 匹配任意一条指令，只要它有两个操作数，而且均是 64 位就行

`mov` 匹配助记符是 mov 的任意指令

可以发现，它的匹配规则非常丰富和灵活，在花指令匹配等场景可以发挥很好的作用。我们只演示了其中简单的规则，复杂的比如像下面这样也可以支持。

Python | 复制代码

```
mov r64, [r32 * 8 + 0x100]
mov r, [r * 8 - 0x100]
mov r64, [r32 * 8 + imm]
mov r, word [eax + r32 * 8 - 0x100]
any r64, r64
push imm
push any
```

比较遗憾的是，想将它用在 ARM/ARM64 分析上的朋友可能要失望了。Code pattern 目前只支持X86/X86-64，需要做更多架构的适配。相比较汇编代码匹配，它的其余五种约束不算特别出彩。但是 FindFunc 用一种非常好的逻辑将它们组织在一起，即可以用六种规则去共同描述和约束一个或一些函数。还可以设置 Invert Match，意指反转该条件，比如对于字符串来说，本意是要求包含，对其反转就是要求不包含。

除此之外，FindFunc 十分关心搜索的处理效率，这体现在多个方面

- 这些匹配条件均作用于反汇编界面，不涉及到反编译过程
- 当设置多个匹配条件时，FIndFunc 会智能排序检索，使得匹配所耗时间最小
- 支持对函数”渐进式“的搜索， Search Functions 是常规的对函数的搜索，Refine Result 是对前一次搜索结果进行进一步的搜索，很像 CE 等工具对内存结果做缩小范围的那个味。

FindFunc 的优点不止于此，它还支持将匹配规则持久化，简单来说，你可以导入和导出匹配规则，下次使用或给他人用。

你可能会感觉它不够”实用“，没办法想象它在哪种场景里使用。但实际上，它的用处不少。举个例子，如果我们持续跟踪和分析某家的 SDK ，其中有一个关键函数每次都会发生些微变化，那么用 FindFunc 描述和定义一套对它的匹配规则，就可以在版本更新中迅速定位到它的位置。

FindFunc 并非完全没有缺点，我们在后文对它做讨论。

4.3 函数重命名

无符号函数在 IDA 中一般被命名为 `sub_xxx` ，当我们对函数进行分析时，常常会根据函数内部的实际功能和 API 调用等，对它进行重命。一些研究人员试图对函数做自动化的重命名，总体来说，这些方法都不算特别好。

思路 1 是识别出函数中存在某些加密算法常量特征，然后将它命名为对应的加密算法。

思路 2 是分析函数中所调用的有符号函数（主要指导入函数）或所引用的字符串，然后进行命名或标记。比如函数里有一个 memcpy 函数，那么自动化工具就将 sub_xxx 修改为 my_memcpy ，这方法好吗？算不上特别好，很容易误判。函数调用某个库函数，就一定只是那个库函数的封装吗？显然不是，又或者如果样本调用多个库函数，那么依照谁命名呢？

auto_re <https://github.com/a1ext/auto_re> 、 ida_autoanalyse <https://github.com/silascutler/IDA_AutoAnalysis> 、IDA-Function-Tagger <https://github.com/alessandrogario/IDA-Function-Tagger> 、idamagicstrings <https://github.com/joxeankoret/idamagicstrings> 等一大批插件都是思路2的产物，试图去理解函数功能并自动重命名，这个事并不好做，但它们是一种好的研究和尝试。

还有一个插件叫 prefix <https://github.com/gaasedelen/prefix> ，它朴素的多，侧重于增强 IDA 的函数重命名功能。你可以用它给一堆函数批量增加前缀，这在一些情况下会有助于分析。