ollvm 的混淆反混淆和定制修改

最近各大杀毒公司陆续都出了混淆，网上关于 ollvm 的资料比较少，于是就有了这篇文章,这篇文章介绍，android 的 native 代码，也就是 so 和 linux 的 c/c++代码均可使用的混淆工具 ollvm 的编译，混淆，反混淆，和反反混淆

第一篇.ollvm 的编译环境搭建------混淆

教你搭建编译和使用 ollvm3.4 3.5 3.6 的环境，非常详细

第二篇.ollvm 的还原---反混淆

根据网上的一些文章，对 ollvm 混淆后的代码进行还原，写下我详细的心得和代码注释和环境搭建(目前只能还原 linux x86，对于 arm 有兴趣的可以进一步研究)

第三篇.ollvm 的定制---反反混淆

由于公司原因，这里介绍修改后的结果

==========第一篇 ollvm 的编译环境搭建------混淆==========

一、androidNDK 搭建 ollvm 环境和使用

注意这里有编译环境和编译后的版本，ubuntu64 位的系统依然可以使用 ndk32，但是只能编译 clang64,所以你不需要 ndk64 就不需要编译 64 位的 ollvm64

1.编译 ollvm 32 位

版本有三个我们选择 obfuscator-llvm-3.4 解压得到文件夹 obfuscator-llvm-3.4

ollvm 的下载地址 https://github.com/obfuscator-llvm/obfuscator/tree/llvm-3.4

ndk 选择 android-ndk-r10b-linux-x86.tar.bz2 和

环境选择 ubuntu14.0.4 x32 or x64

(0)安装 ndk

将 ndk 解压到/opt/android/ndk/

解压后的目录

/opt/android/ndk/android-ndk-r10e

$ sudo gedit /etc/profile，在文件末尾加入如下内容：

```
#set NDK env
export NDK_HOME=/opt/android/ndk/android-ndk-r10b
export PATH=$NDK_HOME:$PATH
```

$ source /etc/profile　使之生效

(1)编译 ollvm 的工具

```
apt-get install cmake
```

sudo apt-get install g++

正式编译 ollvm

```
cd obfuscator-llvm-3.4
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE:String=Release ../
```

make –j4　(不要复制哦,手动输入命令)(注意这里一定要加 j4，如果只是 make –j 他默认只会用一个 cpu 然后会卡到蛋疼的)

注意:分配内存和 cpu 多点不然卡死，我这里是 8G 内存+(2 处理器数量每个处理器 2 个核心)

编译完后得到二进制程序都在 build/bin 和 build/lib

(2)下面来配置 32 位的 ndk

[1]打开 ndk 的 toolchains 目录新建目录 obfuscator-llvm-3.4

并将 llvm-3.3 目录下的 prebuilt 目录和文件 config.mk、setup.mk 和 setup-common.mk 拷贝到 obfuscator-llvm-3.4 目录中

然后替换 obfuscator-llvm-3.4/prebuilt/linux-x86 下的 bin 和 lib 为我们编译好的 bin 和 lib

然后将下面文件复制一份，改名称如下，比如 arm-linux-androideabi-clang3.4 复制一行改名为 arm-linux-androideabi-obfuscator3.4

arm-linux-androideabi-clang3.4-> arm-linux-androideabi-obfuscator3.4

mipsel-linux-android-clang3.4-> mipsel-linux-android-obfuscator3.4

x86-clang3.4-> x86-obfuscator3.4

分别修改以上三个文件的 setup.mk 中的 LLVM_NAME ，即将其指定到开始建立的 obfuscator-llvm-3.4 目录，也就是把把 LLVM_NAME := llvm-$(LLVM_VERSION)改成 LLVM_NAME := obfuscator-llvm-$(LLVM_VERSION)

如果是配置 64 位的 ndk 配置,还要额外修改$NDK_PATH/build/core/setup-toolchain.mk 文件，在 NDK_64BIT_TOOLCHAIN_LIST := 加入 obfuscator 对应的 NDK_TOOLCHAIN_VERSION
NDK_64BIT_TOOLCHAIN_LIST := obfuscator3.4 clang3.6 clang3.5 clang3.4 4.9

2.使用

在 Application.mk 中指定编译器名字：
NDK_TOOLCHAIN_VERSION := obfuscator3.4
在 Android.mk 中设置混淆参数：
LOCAL_CFLAGS += -mllvm -sub -mllvm -bcf -mllvm –fla
正常编译 ndk 就行了
例子

Application.mk

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
APP_ABI := armeabi
NDK_TOOLCHAIN_VERSION := obfuscator
include $(BUILD_EXECUTABLE)
```

Android.mk

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE        := hello
LOCAL_SRC_FILES := hello.c
LOCAL_CFLAGS += -mllvm -sub -mllvm -bcf -mllvm –fla    混淆参数
LOCAL_ARM_MODE := arm
include $(BUILD_EXECUTABLE)
```

hello.c

```
#include <stdio.h>
int main(int argc, char **args){
      int a=1;
       int b=0;
       if(a>b){
            printf("snow:%d\n", a);
       }
       else{
            printf("test:%d\n", b);
       }
       return 0;
}
```

3.更多使用
bcf 可以配合下面参数使用
-mllvm -perBCF=20：对所有函数都混淆的概率是 20%，默认 100%
-mllvm -boguscf-loop=3：对函数做 3 次混淆，默认 1 次
-mllvm -boguscf-prob=40：代码块被混淆的概率是 40%，默认 30%
给某个函数单独加入混淆；注意注意经过我测试只有 ollvm3.5 和 ollvm3.6 可以使用单独函数加混淆

```
int main(int argc, char **args)__attribute((__annotate__(("bcf"))));
```

linux 可执行文件搭建 ollvm 环境和使用

环境跟前面那个一样，在编译完毕 ollvm 之后有目录 obfuscator-llvm-3.4/build/bin/此目录下面有一个 clang，指向 clang-3.4 我们编译 linux 可执行的程序可以用

obfuscator-llvm-3.4/build/bin/clang　xx.c –o xx –mllvm –fla 就是控制流平展了

==========第二篇.ollvm 的还原---反混淆==========

一、网上的 Decllvm 的分析

是 F8LEFT 写的一个工具，这个工具出现在吾爱破解 2016 的安全挑战赛第七题的解答里面，详细第七题分析内容见我的知识百科->match 目录->吾爱破解 2016 的安全挑战赛

此工具给的 demo 有 AliLLVM.py 针对阿里第二届安全挑战赛 crackme3 哦

下面只是说这个工具的使用和原理

1.入口 360LLVM.py 文件

```
if __name__ == "__main__":
    print("============360LLVMStart================")
    ins = C360LLVM()
    reg = ArmReg()
    dbgEng = DbgEngine(reg, ins)
    fd = open("F:/trace.log", "w+")
    dbgEng.start_run(GetRegValue("PC"), 1000, fd)
    fd.close()
    del dbgEng
    del reg
    del ins
    print("============360LLVMEnd================")
```

大致原理这是一个 ida 的脚本，运行需要动态调试程序才行，根据程序运行的时候把寄存器的参数打印下来并且写到 txt 文件里面，方便分析，这个用处不是太大，只是利用 ida 打印程序流程,但是可以通吃 NDK 和 linux 的混淆，顺便感谢一下他的脚本里面包含很多 ida api 的使用哦！

二、又找的的另外一篇利用符号执行

参考文章 https://security.tencent.com/index.php/blog/msg/1122　利用符号执行去除控制流平坦化

这篇文章的办法可以完美的恢复控制流平坦化但是只是 linux 的 64 和 32 位可执行文件 x86 格式的混淆后完美还原，对于 android 的 so 和可执行文件不行,为 arm 格式的指令找不到规则不像 x86 那样找到规则，需要我们自己分析规则

注意作者给的 deflat.py 依赖的 barf 只能运行于 linux64 位，所以就算在 linux32 位上混淆了，我们也要拿到 linux64 位上运行 deflat.py 脚本哦！！！

我的电脑是 ubuntu14.04 x64

1.搭建环境和运行

[1]安装 python

ubuntu 自带了 Python 2.7.6

[2]安装 pip 和？？

apt-get install python-pip

sudo apt-get install python-dev libffi-dev build-essential

[3]安装 barf
下载 barf 解压 cd 到目录运行
python setup.py install

[4]安装 angr
sudo pip install angr 报错，再次安装 sudo apt-get install python-dev libffi-dev build-essential

[5]运行
python deflat.py check_passwd_flat 0x400530

注意在 ubuntu14.0.4 x32 环境运行出错
ImportError: ERROR: fail to load the dynamic library
因为 deflat.py 会去调用 BARF，BARF 这货只能在 linux x64 位运行坑爹啊，但是我们可以在 x32 上编译了，拿到 x64 上面跑，
这里我们自己编译和混淆在 ubuntu14.04_x32 位下面在 ubuntu14.0.4_64 里面还原混淆
[1]编译

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int check_password(char *passwd)
{
    int i, sum = 0;
    for (i = 0; ; i++)
    {
        if (!passwd[i])
        {
            break;
        }
        sum += passwd[i];
    }
    if (i == 4)
    {
        if (sum == 0x1a1 && passwd[3] > 'c' && passwd[3] < 'e' && passwd[0] == 'b')
        {
            if ((passwd[3] ^ 0xd) == passwd[1])
            {
                return 1;
            }
            puts("Orz...");
        }
    }
    else
    {
        puts("len error");
    }
    return 0;
}

int main(int argc, char **argv)
{
    if (argc != 2)
    {
```

```
        puts("error");
        return 1;
    }
    if (check_password(argv[1]))
    {
        puts("Congratulation!");
    }
    else
    {
        puts("error");
    }
    return 0;
}
```

gcc check_passwd.c -o check_passwd

[2]混淆控制流平坦

root/桌面/software/my _compile_ollvm/obfuscator-llvm-3.4/build/bin/clang-3.4 check_passwd.c -o check_passwd_32_flat -mllvm -fla

[3]拿到 x64 位上还原混淆

python deflat.py check_passwd_32_flat 0x80488B0 //main 函数

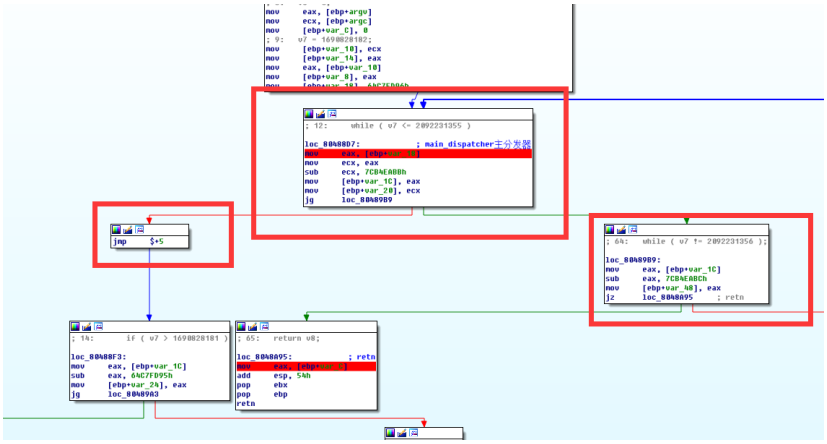
[4]全混淆

python deflat.py check_passwd_32_flat_sub_bcf 0x8048420

依然可以还原-fla

2. deflat.py 脚本分析心得和我的注释

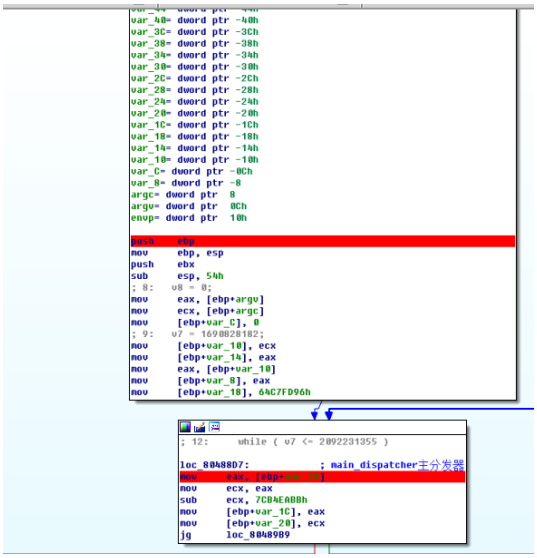
在 ida 函数中可以看见很多块，一块一块的



#第一步:找出 6 大块,可以用静态分析得到

序言	保留
主分发器	去掉
预处理器	去掉
retn 块	保留
真实块	保留
无用块	去掉

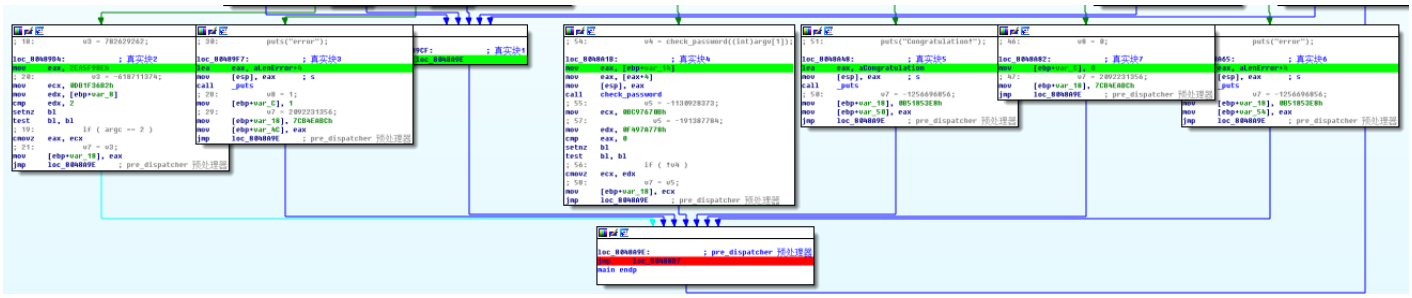
[1]序言和主分发器，函数开始就是序言，序言下面紧接着就是主分发器



[2]预处理器

ida 查找办法谁调用了主分发器，谁就是预处理器(在主分发器按 x，看见的就是预处理器)

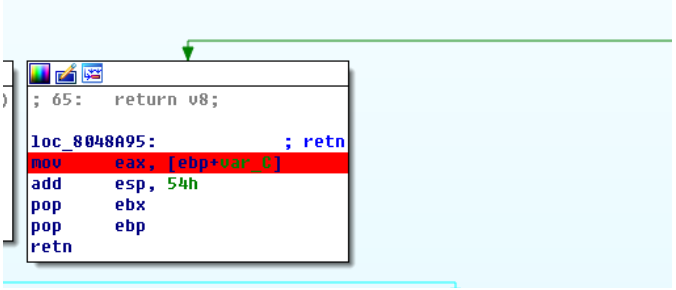
[3]真实块，



ida 查找办法所有调用预处理器的块都是预处理器(在预处理器按 x 查看到的都是真实块)

Direction	Typ	Address	Text		
Up	j	main:loc_80489CF	jmp	loc_8048A9E	; pre_dispatcher
Up	j	main+142	jmp	loc_8048A9E	; pre_dispatcher
Up	j	main+166	jmp	loc_8048A9E	; pre_dispatcher
Up	j	main+193	jmp	loc_8048A9E	; pre_dispatcher
Up	j	main+1B0	jmp	loc_8048A9E	; pre_dispatcher
Up	j	main+1CD	jmp	loc_8048A9E	; pre_dispatcher
Up	j	main+1E0	jmp	loc_8048A9E	; pre_dispatcher

[4]retn 块



ida 查找办法，没有后继的块就是，也就是下线没有分支的

[5]无用块

其余均是无用块

#第二步:找出真实块和序言和 retn 之间的调用关系,必须动态运行

#难点一:使用 symbolic_execution 找出真实块和序言的调用关系，必须使用他的引擎运行或者动态运行

如下的关系

#执行到这里已经获得了原函数的调用关系，下面是恢复之后的关系，真实块 7->retn 返回块

#0x8048a82: ['0x8048a95'] 真实块 6 : retn 返回块

#0x8048a65: ['0x8048a82']　真实块 5： 真实块 6

#0x8048a48: ['0x8048a82']　真实块 4： 真实块 6

#0x80488b0: ['0x80489d4']　序言　　： 真实块 1

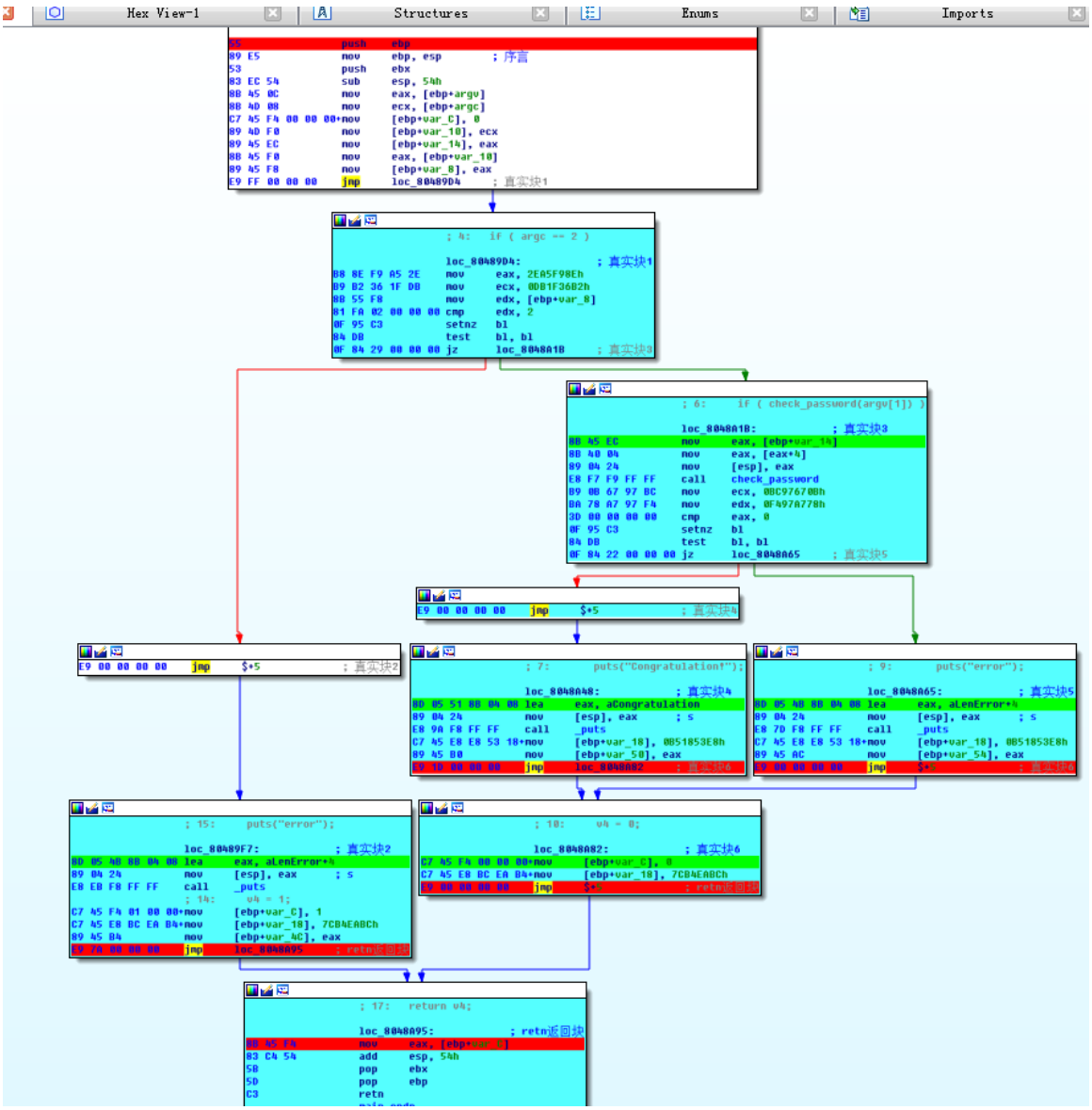
#0x80489d4: ['0x8048a1b', '0x80489f7']　真实块 1 :真实块 3，真实块 2

#0x8048a95: []　　retn 返回块

#0x80489f7: ['0x8048a95']　　真实块 2 : retn 返回块

#0x8048a1b: ['0x8048a65', '0x8048a48']　真实块 3 :真实块 5，真实块 4

下面是修复后他们的调用图



#第三步:nop 掉无用块和主分发器和预处理器,可以静态分析得到

#第四步:修复真实块序言部分的跳转指令，修复办法，

#情况一:有一个 childs，也就是一个后继的块，找到块的最后一条指令，将其抹掉后改成新的 jmp，jmp 到自己的 childs

#情况二:有多个 childs 的

下面是我的代码注释

```
#coding: UTF-8
from barf.barf import BARF
import angr
import simuvex
import pyvex
import claripy
import struct
import sys
```

```
def get_retn_predispatcher(cfg):
    global main_dispatcher
    for block in cfg.basic_blocks:
        if len(block.branches) == 0 and block.direct_branch == None:
            retn = block.start_address
        elif block.direct_branch == main_dispatcher:
            pre_dispatcher = block.start_address
    return retn, pre_dispatcher

def get_relevant_nop_blocks(cfg):
    global pre_dispatcher, prologue, retn
    relevant_blocks = []
    nop_blocks = []
    for block in cfg.basic_blocks:
        if block.direct_branch == pre_dispatcher and len(block.instrs) != 1:
            relevant_blocks.append(block.start_address)
        elif block.start_address != prologue and block.start_address != retn:
            nop_blocks.append(block)
    return relevant_blocks, nop_blocks

def statement_inspect(state):
    global modify_value
    expressions = state.scratch.irsb.statements[state.inspect.statement].expressions
    if len(expressions) != 0 and isinstance(expressions[0], pyvex.expr.ITE):
        state.scratch.temps[expressions[0].cond.tmp] = modify_value
        state.inspect._breakpoints['statement'] = []

def symbolic_execution(start_addr, hook_addr=None, modify=None, inspect=False):
    global b, relevants, modify_value
    if hook_addr != None:
        b.hook(hook_addr, retn_procedure, length=5)
    if modify != None:
        modify_value = modify
    state = b.factory.blank_state(addr=start_addr, remove_options={simuvex.o.LAZY_SOLVES})
    if inspect:
        state.inspect.b('statement', when=simuvex.BP_BEFORE, action=statement_inspect)
    p = b.factory.path(state)
    p.step()
    while p.successors[0].addr not in relevants:
        p = p.successors[0]
        p.step()
    return p.successors[0].addr

def retn_procedure(state):
    global b
    ip = state.se.any_int(state.regs.ip)
    b.unhook(ip)
    return

def fill_nop(data, start, end):
    global opcode
    for i in range(start, end):
```

```
        data[i] = opcode['nop']

def fill_jmp_offset(data, start, offset):
    jmp_offset = struct.pack('<i', offset)
    for i in range(4):
        data[start + i] = jmp_offset[i]

#python deflat.py check_passwd_32_flat 0x80488B0
if __name__ == '__main__':
    if len(sys.argv) != 3:
        print 'Usage: python deflat.py filename function_address(hex)'
        exit(0)
    opcode = {'a':'\x87', 'ae': '\x83', 'b':'\x82', 'be':'\x86', 'c':'\x82', 'e':'\x84', 'z':'\x84', 'g':'\x8F',
              'ge':'\x8D', 'l':'\x8C', 'le':'\x8E', 'na':'\x86', 'nae':'\x82', 'nb':'\x83', 'nbe':'\x87', 'nc':'\x83',
              'ne':'\x85', 'ng':'\x8E', 'nge':'\x8C', 'nl':'\x8D', 'nle':'\x8F', 'no':'\x81', 'np':'\x8B', 'ns':'\x89',
              'nz':'\x85', 'o':'\x80', 'p':'\x8A', 'pe':'\x8A', 'po':'\x8B', 's':'\x88', 'nop':'\x90', 'jmp':'\xE9', 'j':'\x0F'}
    filename = sys.argv[1]    #check_passwd_32_flat
    start = int(sys.argv[2], 16)      #0x80488B0
    barf = BARF(filename)
    base_addr = barf.binary.entry_point >> 12 << 12
    print "snowtest="+str(barf.binary)
    print 'snowtest--base_addr%#x' % base_addr    #base_addr=0x8048000
    b = angr.Project(filename, load_options={'auto_load_libs': False, 'main_opts':{'custom_base_addr': 0}})
    cfg = barf.recover_cfg(ea_start=start)
    blocks = cfg.basic_blocks
#第一步:找出 6 大块,可以用静态分析得到
    #1.序言:序言为函数开始地址
    prologue = start
    #2.主分发器:序言的后继为主分发器(也就是序言指向的第一个块)
    main_dispatcher = cfg.find_basic_block(prologue).direct_branch
    #3.预处理器:后继为主分器的块位预处理器(也就是后面一个代码块是主分器的)
    #ida 查找办法，对着主分发器按 x 键看交叉引用，调用主分发器的那个块
    #4.retn 块:无后继的块为 retn 块(也就是没有任何下线分支的块)
    retn, pre_dispatcher = get_retn_predispatcher(cfg)
    #5.真实块:后继为预处理器的块为真实块
    #6.无用块:剩下的就是无用块
    relevant_blocks, nop_blocks = get_relevant_nop_blocks(cfg) #无用块(剩下的就是无用块)
    print '******************relevant blocks**********************'
    print 'func start addr prologue:%#x' % start       #0x80488b0  函数首地址为序言的地址
    print 'main_dispatcher:%#x' % main_dispatcher      #0x80488d7  主分发器
    print 'pre_dispatcher:%#x' % pre_dispatcher        #0x8048a9e  预处理器
    print 'func retn:%#x' % retn                       #0x8048a95 retn 块
    print 'relevant_blocks:', [hex(addr) for addr in relevant_blocks]

    print '******************symbolic execution*********************'
    relevants = relevant_blocks
    relevants.append(prologue)     #加入序言
    relevants_without_retn = list(relevants)
    relevants.append(retn)
    flow = {}
    for parent in relevants: #parent 依次是 0x80489d4，0x80489f7 等块的首地址
        # print "snow_parent=%x" % parent
```

```
        flow[parent] = []
    modify_value = None
    patch_instrs = {}
    for relevant in relevants_without_retn:
    #dse 0x80489d4----------真实块 1
    #dse 0x80489f7----------真实块 2
    #dse 0x8048a1b----------真实块 3
    #dse 0x8048a48----------真实块 4
    #dse 0x8048a65----------真实块 5
    #dse 0x8048a82----------真实块 6
    #dse 0x80488b0----------序言
        print '----------dse %#x----------' % relevant
        block = cfg.find_basic_block(relevant) #找到上面块的范围
        has_branches = False
        hook_addr = None
        for ins in block.instrs:     #遍历这些块打印出操作码
            #print "snowinstr="+ins.asm_instr.mnemonic
            if ins.asm_instr.mnemonic.startswith('cmov'): #有 cmov 结尾的指令说明有分支的块有 1,3 块
                print "snow_has_branches=%s" % ins.asm_instr.mnemonic
                patch_instrs[relevant] = ins.asm_instr
                has_branches = True
            elif ins.asm_instr.mnemonic.startswith('call'): #这些块中有 call 指令有 2,4,5,6,序言块
                hook_addr = ins.address
                print "snow_hook_addr=%x" % hook_addr
        #难点一:使用 symbolic_execution 找出真实块和序言的调用关系，必须使用他的引擎运行或者动态运行
#第二步:找出真实块和序言和 retn 之间的调用关系,必须动态运行
        if has_branches:    #flow[relevant]分别是 flow[0x80489d4],flow[0x80489f7]等等签名已经清空
            #下面可能是修改标志寄存器达到往两个分支运行
            flow[relevant].append(symbolic_execution(relevant, hook_addr, claripy.BVV(1, 1), True))
            flow[relevant].append(symbolic_execution(relevant, hook_addr, claripy.BVV(0, 1), True))
        else:
            flow[relevant].append(symbolic_execution(relevant, hook_addr))

    print '**********flow**********'
    #**********flow**********
    #执行到这里已经获得了原函数的调用关系，下面是恢复之后的关系，真实块 7->retn 返回块
    #0x8048a82: ['0x8048a95']    真实块 6 : retn 返回块
    #0x8048a65: ['0x8048a82']    真实块 5 :  真实块 6
    #0x8048a48: ['0x8048a82']    真实块 4 :  真实块 6
    #0x80488b0: ['0x80489d4']    序言       :  真实块 1
    #0x80489d4: ['0x8048a1b', '0x80489f7']    真实块 1 :真实块 3，真实块 2
    #0x8048a95: []                                    retn 返回块
    #0x80489f7: ['0x8048a95']                      真实块 2 : retn 返回块
    #0x8048a1b: ['0x8048a65', '0x8048a48']    真实块 3 :真实块 5，真实块 4

    for (k, v) in flow.items():
        print '%#x:' % k, [hex(child) for child in v]

    print '**********patch**********'
    flow.pop(retn)
    origin = open(filename, 'rb')
    origin_data = list(origin.read())
```

```
    origin.close()
    recovery = open(filename + '.recovered', 'wb') #输出文件路径
#第三步:nop 掉无用块和主分发器和预处理器,可以静态分析得到
    for nop_block in nop_blocks:
    #无用块开始地址
    #下面是吧无用块填充 0
    #snow_nop_block.start_address=80488d7
    #snow_nop_block.start_address=80488ee
    #snow_nop_block.start_address=80488f3
    #snow_nop_block.start_address=8048904
    #snow_nop_block.start_address=8048909
    #snow_nop_block.start_address=804891a
    #snow_nop_block.start_address=804891f
    #snow_nop_block.start_address=8048930
    #snow_nop_block.start_address=8048935
    #snow_nop_block.start_address=8048946
    #snow_nop_block.start_address=804894b
    #snow_nop_block.start_address=804895c
    #snow_nop_block.start_address=8048961
    #snow_nop_block.start_address=8048972
    #snow_nop_block.start_address=8048977
    #snow_nop_block.start_address=8048988
    #snow_nop_block.start_address=804898d
    #snow_nop_block.start_address=804899e
    #snow_nop_block.start_address=80489a3
    #snow_nop_block.start_address=80489b4
    #snow_nop_block.start_address=80489b9
    #snow_nop_block.start_address=80489ca
    #snow_nop_block.start_address=80489cf
    #snow_nop_block.start_address=8048a9e
        #print "snow_nop_block.start_address=%x" % nop_block.start_address
        fill_nop(origin_data, nop_block.start_address - base_addr, nop_block.end_address - base_addr + 1)
#第四步:修复真实块序言部分的跳转指令，修复办法，
    #情况一:有一个 childs，也就是一个后继的块，找到块的最后一条指令，将其抹掉后改成新的 jmp，jmp 到自己的 childs
    #情况二:有多个 childs 的, 针对产生分支的真实块把 CMOV 指令改成相应的条件跳转指令跳向符合条件的分支，例如 CMOVZ 改成 JZ ，再在这条之后添加 JMP 指令跳向另一分支
    for (parent, childs) in flow.items():
    #snow_parent if=8048a82 真实块 6
    #snow_parent if=8048a65 真实块 5
    #snow_parent if=8048a48 真实块 4
    #snow_parent if=80488b0 序言
    #snow_parent if=80489f7 真实块 2
        if len(childs) == 1:  #有一个 childs 的
            #print "snow_parent if=%x" % parent
              last_instr = cfg.find_basic_block(parent).instrs[-1].asm_instr
            #print "snow_last_instr addr=%x" % last_instr.address #找到最后一条指令地址，也就是 jmp 的地址
              file_offset = last_instr.address - base_addr   #偏移地址
              origin_data[file_offset] = opcode['jmp']
              file_offset += 1
              fill_nop(origin_data, file_offset, file_offset + last_instr.size - 1)#先填充为 0
              fill_jmp_offset(origin_data, file_offset, childs[0] - last_instr.address - 5)#然后填充 jmp
    #snow_parent else=80489d4 真实块 1
```

```
#snow_parent else=8048a1b 真实块 3
        else:                       #有 2 个 childs 的

            #print "snow_parent else=%x" % parent
            instr = patch_instrs[parent]
            #print "snow_parent instr.address=%x" % instr.address #分别是 80489ec 和 8048a3d,也就是 cmov..指令的地址
            file_offset = instr.address - base_addr
            #nop 掉 cmov...指令到块结尾所有部分
            fill_nop(origin_data, file_offset, cfg.find_basic_block(parent).end_address - base_addr + 1)
            origin_data[file_offset] = opcode['j']
            origin_data[file_offset + 1] = opcode[instr.mnemonic[4:]]
            fill_jmp_offset(origin_data, file_offset + 2, childs[0] - instr.address - 6)
            file_offset += 6
            origin_data[file_offset] = opcode['jmp']
            fill_jmp_offset(origin_data, file_offset + 1, childs[1] - (instr.address + 6) - 5)
    recovery.write(''.join(origin_data))        #把结果写回去
    recovery.close()
    print 'Successful! The recovered file: %s' % (filename + '.recovered')
```

==========第三篇:ollvm 的定制---反反混淆==========

1.控制流平展模式 Control Flow Flattening，使用办法加入参数-ollvm-fla，效果如下，
原始 fla 的效果，可以看见程序逻辑被打乱，出现很多分支，fla 只会处理存在分支的函数

```
int __cdecl damage(int a1, int a2)
{
  signed int v2; // eax@9
  signed int v4; // [sp+18h] [bp-20h]@1
  int v5; // [sp+28h] [bp-10h]@0

  v4 = 922355287;
  do
  {
    while ( v4 > 258517458 )
    {
      switch ( v4 )
      {
        case 258517459:
          v5 = a1 & (a1 - a2);
          v4 = -1390597462;
          break;
        case 922355287:
          v2 = 1731190932;
          if ( a1 > a2 )
            v2 = 258517459;
          v4 = v2;
          break;
        case 1731190932:
          v5 = 1;
          v4 = -1390597462;
          break;
      }
    }
  }
  while ( v4 != -1390597462 );
  return v5;
}
```

修改后的效果,在 switch 分支插入更多垃圾代码和在骨干函数插入垃圾代码，去掉 switch 特征

```
int __cdecl damage(int a1, int a2)
{
  signed int v2; // eax@12
  signed int v4; // [sp+20h] [bp-20h]@1
  int v5; // [sp+30h] [bp-10h]@0

  v4 = 1541750864;
  do
  {
    while ( 1 )
    {
      while ( 1 )
      {
        while ( 1 )
        {
          while ( v4 <= 478286215 )
          {
            if ( v4 == -727579137 )
            {
              v5 = 1;
              v4 = 1816710535;
            }
          }
          if ( v4 > 1239641174 )
            break;
          if ( v4 == 478286216 )
          {
            v5 = a1 & (a1 - a2);
            v4 = 1816710535;
          }
        }
        if ( v4 != 1239641175 )
          break;
        v5 = 1;
        v4 = 1816710535;
      }
      if ( v4 != 1541750864 )
        break;
      v2 = 1239641175;
      if ( a1 > a2 )
        v2 = 478286216;
      v4 = v2;
    }
  }
  while ( v4 != 1816710535 );
  return v5;
}
```

2. 指令替换模式 Instructions Substitution,将正常的运算逻辑（+，-，&，|等）替换成更加复杂的操作，如有表达式 a=b-c. 等价形式是 a=b+r-r-c,经过随机处理，可以将原来的表达式随机复杂化，使用办法加入参数-ollvm-sub
使用原始 ollvm 得到的效果，可以看见运算逻辑变得更加复杂

```
int __cdecl damage(int a1, int a2)
{
  int v3; // [sp+Ch] [bp-4h]@2

  if ( a1 <= a2 )
    v3 = 1;
  else
    v3 = ~(~a1 | ~(a1 - 264813991 - a2 + 264813991));
  return v3;
}
```

因为运算逻辑存在加减可还原，将原始数据拆分，加大分析难度，使得自动化脚本编写成本增加

```
int __cdecl damage(int a1, int a2)
{
  int v3; // [sp+Ch] [bp-4h]@2

  if ( a1 <= a2 )
    v3 = 1;
  else
    v3 = (~(a1 + 1663824241 - a2 - 1663824241) ^ ~a1 ^ (a1 + 1663824241 - a2 - 1663824241)) & (~a1 ^ (a1 + 1663824241 - a2 - 1663824241));
  return v3;
}
```

3. 控制流伪造模式 Bogus Control Flow ，在原有代码块随机插入新的代码块，随机概率是否插入新的块，原始块被克隆并且插入垃圾代码，而且是随机的，bcf 可以利用参数进行随机
原始效果谓词很明显，有经验的人会调试函数然后断点排除垃圾块(比如百度 JNI_ONLoad 加固使用 bcf 混淆 2 次 100%)

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  if ( y2 >= 10 && (((_BYTE)x1 - 1) * (_BYTE)x1 & 1) != 0 )
    goto LABEL_4;
  while ( 1 )
  {
    damage(20, 10);
    if ( y2 < 10 || (((_BYTE)x1 - 1) * (_BYTE)x1 & 1) == 0 )
      break;
LABEL_4:
    damage(20, 10);
  }
  return 0;
}
```

修改后的效果将谓词长度变短，减少可见化，让其难以区别是程序逻辑还是谓词

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  if ( y2 >= 2 && x1 != 3 )
    goto LABEL_4;
  while ( 1 )
  {
    damage(20, 10);
    if ( y2 < 2 || x1 == 3 )
      break;
LABEL_4:
    damage(20, 10);
  }
  return 0;
}
```

4.联合使用

sub 和 fla 全加并且 bcf 混淆 100%处理一次效果

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  signed int v3; // eax@9
  signed int v4; // ecx@12
  signed int v6; // [sp+34h] [bp-14h]@1
  bool v7; // [sp+3Ah] [bp-Eh]@1
  bool v8; // [sp+3Bh] [bp-Dh]@1

  v7 = (((_BYTE)x1 - 1) * (_BYTE)x1 & 1) == 0;
  v8 = y2 < 10;
  v6 = -1958700926;
  while ( 1 )
  {
    while ( v6 <= 911995501 )
    {
      if ( v6 == -1958700926 )
      {
        v3 = 1805705469;
        if ( v8 || v7 )
          v3 = 1231613844;
        v6 = v3;
      }
    }
    if ( v6 == 911995502 )
      break;
    if ( v6 == 1231613844 )
    {
      damage(20, 10);
      v4 = 1805705469;
      if ( y2 < 10 || (((_BYTE)x1 - 1) * (_BYTE)x1 & 1) == 0 )
        v4 = 911995502;
      v6 = v4;
    }
    else if ( v6 == 1805705469 )
    {
      v6 = 1231613844;
      damage(20, 10);
    }
  }
  return 0;
}
```

修改后的效果

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  signed int v3; // eax@12
  int v4; // eax@15
  signed int v5; // ecx@15
  int v7; // eax@19
  int v8; // eax@20
  int v9; // [sp-10h] [bp-78h]@19
  int v10; // [sp+0h] [bp-68h]@20
  int v11; // [sp+4h] [bp-64h]@20
  int v12; // [sp+8h] [bp-60h]@20
  int v13; // [sp+Ch] [bp-5Ch]@19
  int v14; // [sp+10h] [bp-58h]@19
  int v15; // [sp+14h] [bp-54h]@19
  int v16; // [sp+18h] [bp-50h]@15
  int v17; // [sp+1Ch] [bp-4Ch]@15
  int v18; // [sp+20h] [bp-48h]@15
  int v19; // [sp+24h] [bp-44h]@15
  int v20; // [sp+28h] [bp-40h]@10
  int v21; // [sp+2Ch] [bp-3Ch]@8
  int v22; // [sp+30h] [bp-38h]@7
  int v23; // [sp+34h] [bp-34h]@6
  int v24; // [sp+38h] [bp-30h]@5
  int v25; // [sp+3Ch] [bp-2Ch]@3
  int v26; // [sp+40h] [bp-28h]@2
  int v27; // [sp+44h] [bp-24h]@2
  int v28; // [sp+48h] [bp-20h]@1
  int v29; // [sp+4Ch] [bp-1Ch]@1
  int v30; // [sp+50h] [bp-18h]@1
  bool v31; // [sp+56h] [bp-12h]@1
  bool v32; // [sp+57h] [bp-11h]@1
  int *v33; // [sp+58h] [bp-10h]@19

  v29 = 5;
  v28 = y2;
  v31 = (x1 - 1770100509 + 1770100508) * x1 % 5u == 0;
  v32 = y2 < 11;
  v30 = -537140100;
  do
  {
    while ( 1 )
    {
      while ( 1 )
      {
        while ( 1 )
        {
          while ( 1 )
          {
            while ( 1 )
            {
              v27 = v30;
              v26 = v30 + 537140101;
              if ( v30 > -537140101 )
                break;
              v25 = v27 + 1134947918;
              if ( v27 == -1134947918 )
                goto LABEL_15;
            }
            v24 = v27 - 2025995775;
            if ( v27 <= 2025995775 )
              break;
            v20 = v27 - 2025995776;
            if ( v27 == 2025995776 )
            {
              *v33 = 0;
              v12 = 20;
              v11 = 10;
              v8 = damage(20, 10);
              v30 = 1525347129;
              v10 = v8;
LABEL_15:
              v19 = 20;
              v18 = 10;
              v4 = damage(20, 10);
              v5 = -183815396;
              v17 = v4;
              v16 = 1525347129;
              if ( ((x1 - 1960776406 + 1960776405) * x1 % 5u == 0) & (unsigned __int8)((y2 >= 11) ^ ((x1
                                                                                                    - 1960776406
                                                                                                    + 1960776405)
                                                                                                   * x1
                                                                                                   % 5u == 0)) & 1 )
                v5 = 1525347129;
              v30 = v5;
            }
          }
          v23 = v27 + 537140100;
          if ( v27 != -537140100 )
            break;
          v3 = -183815396;
          if ( (unsigned __int8)(!v31 ^ !v32 ^ v31) & (unsigned __int8)(!v32 ^ v31) & 1 )
            v3 = -1134947918;
          v30 = v3;
        }
        v22 = v27 + 183815396;
        if ( v27 != -183815396 )
          break;
        v33 = &v9;
        v15 = 20;
        v14 = 10;
        v7 = damage(20, 10);
        v30 = -1134947918;
        v13 = v7;
      }
      v21 = v27 - 1525347129;
    }
    while ( v27 != 1525347129 );
    return 0;
```

参考文章

感谢下面的作者的分享

http://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html Deobfuscation: recovering an OLLVM-protected program

http://www.freebuf.com/articles/terminal/130142.html 反混淆：恢复被 OLLVM 保护的程序

https://www.oschina.net/p/decllvm 针对 ollvm 的 ida 分析插件 Decllvm

https://security.tencent.com/index.php/blog/msg/1122 利用符号执行去除控制流平坦化