Angr CFG的使用方法

# 一、Angr CFG类简介

利用Angr创建CFG图的时候可以选择调用三种方法CFG(),CFGFast() ,CFGAccurate()

1. **CFGFast()快速方法 //CFG()也是调用的CFGFast()**

高覆盖率的控制流图，而不关心函数之间的可达性

1. **CFGAccurate()精准方法**

使用强制执行(解决直接跳转)，符号执行(解决间接跳转)，轻量后向切片(解决上下文,CFGAccurate从未解决的跳转开始计算一个反向切片。切片延伸到前一个调用上下文的开始处。也就是说，如果间接跳转在一个函数Fa中被分析，同时Fa被Fb和Fc调用，那么切片将会从Fa反向拓展到两个开始节点：一个开始于Fb，另一个开始于Fc。)，和值集分析生成完整的CFG。

其中，以程序入口点的基本块来初始化图。

# 二、Angr生成CFG流程

1. **安装依赖**
2. angr
3. angrutils
4. 报错dot not found in path 的问题

sudo apt-get install graphvi

1. **过程**

import angr

from angrutils import plot\_cfg

proj=angr.Project('mybinary',load\_options={‘auto\_load\_libs’:False})

*#注意加入load\_options，否则会报错*

cfg = proj.analyses.CFGAccurate(para ……..)

*#生成CFG图*

plot\_cfg(cfg, "mycfg",asminst=True, remove\_imports=True, remove\_path\_terminator=True)

*#画cfg*

1. **CFGAccurate参数**

context\_sensitivity\_level

CFG的上下文敏感级别（请参阅文档更多细节）。范围从0到无穷大。默认1。  
avoid\_runs

一个列表，存放避免执行的runs。  
enable\_function\_hints

是否使用函数提示（可能用作退出目标的常量）。  
call\_depth

设置跟踪调用堆栈的深度。  
call\_tracing\_filter

筛选器应用于给定路径并跳转以确定是否在达到call\_depth时跳过它。  
initial\_state

用于开始分析的起始状态。  
iterable starts

一组起始分析的起点。它可以包含以下三种不同类型的条目：指定为整数的地址，包含整数地址和跳转条件的2元组或SimState实例。在启动时不支持的条目会导致引发AngrCFGError。  
keep\_state

是否保留每个CFGNode的SimStates。  
enable\_advanced\_backward\_slicing

是否启用强化技术来解决直接跳转  
enable\_symbolic\_back\_traversal

是否启用强化技术来解决间接跳转  
additional\_edges

一个字典，映射的是基本块地址和successors地址，以便手动包含和前向分析。  
bool no\_construct

跳过构造程序。 仅用于单元测试。  
bool normalize

CFG以及所有功能图是否应该被标准化。  
int max\_iterations

每个基本块应该“执行”的最大迭代次数。1默认。循环分析等复杂分析通常需要较大数量的迭代。  
iterable address\_whitelist

允许的地址列表。 该地址集合之外的任何基本块都将被忽略。（这个好呀，可以自己确定执行边界）  
networkx.DiGraph base\_graph

一个基本的控制流程图。 此图表中的每个节点必须具有以下属性：`addr`和`size`。 CFG恢复将严格遵循图中所示的节点和边，并丢弃任何不符合基本图中现有边的控制流。 例如，您可以将函数本地转换图作为基础图传递，CFGAccurate将遍历节点和边并提取有用的信息。  
int iropt\_level

VEX IR的优化级别（0,1,2）。 如果`iropt\_level`为None，将使用默认级别。（这个就与VEX有关了，似乎处理thumb模式的指令可以改它）  
int max\_steps

在暂停恢复过程之前，从每次启动恢复最长路径的基本块的最大数量。  
state\_add\_options

将添加到初始状态的状态选项。  
state\_remove\_options

将从初始状态中删除的状态选项。

其中最令人期待的应该就是enable\_advanced\_backward\_slicing 和 param enable\_symbolic\_back\_traversal，要知道很多二进制块文件的跳转方式真的是层出不穷，如果识别不出这些跳转，那么控制流图肯定是不完善的。

1. **plot\_cfg参数**

plot\_cfg(cfg, fname, format="png", path=None, asminst=False, vexinst=False, func\_addr=None, remove\_imports=True, remove\_path\_terminator=True, remove\_simprocedures=False, debug\_info=False, comments=True, color\_depth=False)

cfg cfg对象

fname 图的名称

format 图的格式

path 图的路径

asminst 是否在每个基本块中显示每条指令

vexinst 是否显示vex指令

func\_addr 要绘制出的函数地址

remove\_imports 是否移除import,默认T

remove\_path\_terminator是否移除路径终点,默认T

remove\_simprocedures 是否移除simprocedure ,默认F

angrutils还可以画控制依赖图和数据流图，等一切NetworkX DiGraph的实例

# 三、CFG功能

这部分总结了生成的cfg对象可以查询或利用的函数或属性

**1. cfg.kb.functions 返回一个字典类型**

（1）属性

function\_map  
callgraph 函数调用图

block\_map

（2）函数

contains\_addr（addr）  
确定地址是否由功能管理器处理。  
注意：这个函数与python编程习惯用法不一致，但是出于性能原因需要它。  
参数：addr（int） - 函数的地址。  
  
ceiling\_func（addr） （找上限函数）  
返回地址最少或地址大于或等于地址的函数。  
参数：addr（int） - 要查询的地址。  
返回：一个Function实例，如果addr后面没有其他函数，则返回None。  
返回类型：Functions或无  
  
floor\_func（addr） （找下限函数）  
返回地址最大的地址小于或等于地址的函数。  
参数：addr（int） - 要查询的地址。  
返回：一个Function实例，如果在addr之前没有其他函数，则返回None。  
返回类型：Function或无  
  
function（addr = None，name = None，create = False，syscall = False，plt = None）  
从函数管理器中获取一个函数对象。

1. **entry\_func = cfg.kb.functions[addr] 这里得到一个addr地址的函数**  
   addr（int） - 函数的地址。  
   name（str） - 函数的名称。  
   create（bool） - 如果函数不存在，是否创建函数。  
   syscall（bool） - 真正的创建函数作为系统调用，否则为False。  
   或无plt（bool） - 真正找到PLT存根，假找到非PLT存根，无以禁用此限制。  
   传递addr或名称与适当的值。

. block\_addr  该函数的基本块开始的一组地址。  
.blocks      属于该函数的一组基本块，可以使用capstone来explore或反编译。  
.string\_references() 返回函数中任何点引用的所有常量字符串的列表。它们被格式化为（addr，string）元组，其中addr是字符串所在二进制数据段中的地址，而string是一个包含字符串值的python字符串。  
.returning   是一个布尔值，表示函数是否可以返回。 False表示所有路径不返回。  
.callable   是涉及该函数的angr Callable对象。你可以像Python函数一样使用python参数来调用它，并返回实际结果（可能是符号），就好像你用这些参数运行函数一样！  
.transition\_graph 是一个NetworkX DiGraph 描述这个函数里面的控制流。它类似于IDA按功能级别显示的控制流图。（什么是按功能级别？？）  
.entry\_func.name  函数的名称  
.has\_unresolved\_calls 与检测CFG内的不精确性有关。有时，分析无法检测到间接调用或跳转的可能目标。如果这发生在一个函数中，那么该函数将会将相应的has\_unresolved\_ \*值设置为True。（这个很重要哦 如果函数有未检测到的跳转，这个参数就是true）  
.get\_call\_sites() 返回以呼叫输出到其他功能的基本块的所有地址的列表。（就是哪些基本块调用了其他函数，就返回这些基本块的地址）  
.get\_call\_target(callsite\_addr) 返回上述基本块调用的函数

.get\_call\_return(callsite\_addr)  返回上述基本块返回的函数

1. **cfg.remove\_cycles()**

移除图中所有循环

1. **cfg.unroll\_loops(*max\_loop\_unrolling\_times*)**

循环展开

1. **cfg.get\_subgraph(*starting\_node*, *block\_addresses*)**

参数：子图的开始节点

一系列包含在子图中的块地址

1. **cfg.get\_predecessors(*cfgnode*, *excluding\_fakeret=True*, *jumpkind=None*)**

得到前驱节点

1. **cfg.get\_successors(*basic\_block*, *excluding\_fakeret=True*, *jumpkind=None*)**

得到后缀节点

1. **cfg.get\_all\_predecessors(*cfgnode*)**

得到所有前驱结点

1. **cfg.get\_any\_node(*addr*, *is\_syscall=None*, *anyaddr=False*)**

得到这个地址的节点

**10.cfg.get\_node(*block\_id*)**

得到这个块的节点

**11.cfg,irsb\_from\_node(*cfg\_node*)**

从这个节点得到irsb

**12.cfg.get\_any\_irsb(*addr*)**

从这个地址得到irsb

**13.cfg.nodes()**

图中所有节点的iterator

**14.cfg.get\_all\_irsbs(*addr*)**

得到这个地址开始的所有irsb

**15.cfg.get\_branching\_nodes()**

得到出度大于2的所有节点

# 四、其他

利用angr的analysis类还可以生成CFG、CDG（控制依赖图）、DDG（数据依赖图）、DFG

（数据流图）

CFGAccurate底层依赖的是pyvex，也就是说是对转换后的IR语言分析的，所以如果CFGAccurate中出现了问题，可以先查看它的block.vex，或者用capstone反编译看下。

# 五、参考链接

CFGAccurate()参数

<http://angr.io/api-doc/angr.html?highlight=cfgbase#angr.analyses.cfg.cfg_base.CFGBase>

angr-utils源码，里面有各种图绘制的样例

<https://github.com/axt/angr-utils>

angr\_doc

<https://docs.angr.io/docs/analyses/cfg_accurate.html>

生成CFG的调用过程中文版<https://blog.csdn.net/doudoudouzoule/article/details/80208323>

利用cfg 抓取特定位置的函数然后进行一次模拟实际执行

<https://blog.csdn.net/qq_21063873/article/details/70193860>