18.2 angr

angr 是一个支持多架构的二进制分析平台，具备对二进制文件的动态符号执行能力和多种静态分析能力。在近几年的 CTF 中也大有用处，特别是在一些类似 DARPA Cyber Grand Challenge 的自动化漏洞挖掘、加固和利用的攻防竞赛中，常常能见到它的身影。

angr 拥有一个活跃的社区，每一个大的版本都有不少变化。从 angr 8 开始，已经全面迁移到了 Python 3，本章的内容也将基于这一版本。

18.2.1 安装

在 Ubuntu 16.04 上，你首先需要安装一些依赖：

```
$ sudo apt install python3-dev libffi-dev build-essential virtualenvwrapper
```

另外有一些 angr 的依赖（例如 z3）是从原始库中 fork 而来，如果你已经安装过 z3，那么肯定不希望 angr 的依赖覆盖掉原官方的库，这里作者强烈建议在虚拟环境中安装 angr：

```
$ mkvirtualenv --python=$(which python3) angr
$ pip install angr
```

如果安装失败，那么你可以按照下面的顺序从 angr 的官方仓库依次安装，它们也是 angr 的主要组成模块：

```
1. claripy：符号求解器的抽象
2. archinfo：用于描述各种体系结构
3. pyvex：VEX IR 的 Python 包装
4. cle：二进制文件加载器
5. angr：主程序
```

例如下面这样：

```
$ git clone https://github.com/angr/claripy && cd claripy
$ sudo pip install -r requirements.txt
$ sudo python setup.py build
$ sudo python setup.py install
```

最后，angr 还开发了一个基于 QT 的 GUI，有需要的读者可以到官方文档中查看 angr Management 的相关内容。

18.2.2 快速入门

使用 angr 的第一步是新建一个工程，几乎所有的操作都是围绕这个工程展开的，在参数中将 auto_load_libs 设置为 False，可以关闭程序依赖库的加载，在实践中我们通常使用这种方式加快分析的速度，但同时有可能会对分析的精度有所影响：

```
>>> import angr
>>> proj = angr.Project('/bin/true')
>>> p = angr.Project('/bin/true', auto_load_libs=False)
```

通过这个工程，我们可以得到二进制文件的各种信息，如：

```
>>> proj.filename      # 文件名
'/bin/true'
>>> proj.arch          # 一个 archinfo.Arch 对象
<Arch AMD64 (LE)>
>>> hex(proj.entry)    # 入口点
'0x4013d0'
```

angr 的 CLE 模块用于将二进制文件本身及其依赖的共享库映射到虚拟地址中：

```
>>> proj.loader
<Loaded true, maps [0x400000:0x5008000]>
```

被加载的所有对象文件如下所示，同时我们还可以查看它们的一些相关信息：

```
>>> for obj in proj.loader.all_objects:            # 所有对象文件
...     print(obj)
...
<ELF Object true, maps [0x400000:0x6063bf]>
<ELF Object libc-2.23.so, maps [0x1000000:0x13c999f]>
<ELF Object ld-2.23.so, maps [0x2000000:0x2227167]>
<ELFTLSObject Object cle##tls, maps [0x3000000:0x3015010]>
<ExternObject Object cle##externs, maps [0x4000000:0x4008000]>
<KernelObject Object cle##kernel, maps [0x5000000:0x5008000]>
>>> obj = proj.loader.main_object                  # 主程序
>>> obj
<ELF Object true, maps [0x400000:0x6063bf]>
>>> hex(obj.entry)
'0x4013d0'
>>> hex(obj.min_addr), hex(obj.max_addr)           # 起始地址和结束地址
('0x400000', '0x6063bf')
```

project.factory 提供了几个方便的构造函数用于对二进制文件进行分析，例如 block() 函数可以从指定地址解析一个基本块（basic block），所得到的对象类型为 Block：

```
>>> block = proj.factory.block(proj.entry)     # 从程序头开始解析一个 basic block
>>> block
<Block for 0x4013d0, 41 bytes>
>>> block.pp()                                 # 打印
0x4013d0:   xor ebp, ebp
0x4013d2:   mov r9, rdx
0x4013d5:   pop rsi
0x4013d6:   mov rdx, rsp
0x4013d9:   and rsp, 0xfffffffffffffff0
0x4013dd:   push    rax
0x4013de:   push    rsp
0x4013df:   mov r8, 0x403fc0
0x4013e6:   mov rcx, 0x403f50
0x4013ed:   mov rdi, 0x401330
0x4013f4:   call    0x401180
>>> block.instructions                         # 指令数量
11
>>> block.instruction_addrs                    # 指令地址
[4199376, 4199378, 4199381, 4199382, 4199385, 4199389, 4199390, 4199391, 4199398,
4199405, 4199412]
```

或者将 Block 类型的对象转换成其他类型的对象：

```
>>> block.capstone
<CapstoneBlock for 0x4013d0>
>>> block.capstone.pp()

>>> block.vex
IRSB <0x29 bytes, 11 ins., <Arch AMD64 (LE)>> at 0x4013d0
>>> block.vex.pp()
```

到这里我们已经介绍了一些基于原始对象文件的静态分析方法，接下来考虑让程序“动”起来，这里所说的动就是符号执行，其中对程序状态的管理十分关键。首先需要初始化一个模拟程序状态的 SimState 对象：

```
>>> state = proj.factory.entry_state()     # 从入口点创建 state
>>> state
<SimState @ 0x4013d0>
```

该对象包含了程序在某个节点处的内存、寄存器、文件系统数据等等模拟运行时动态变化的数据，例如下面这些：

```
>>> state.regs                              # 寄存器名对象
<angr.state_plugins.view.SimRegNameView object at 0x7f7afd82ab38>
>>> state.regs.rip                          # BV64 对象
<BV64 0x4013d0>
>>> state.regs.rsp
<BV64 0x7fffffffeff88>
>>> state.regs.rsp.length                   # BV 对象都具有 length 属性
64
>>> state.regs.rdi
WARNING | angr.state_plugins.symbolic_memory | Register rdi has an unspecified
value; Generating an unconstrained value of 8 bytes.
<BV64 reg_rdi_0_64{UNINITIALIZED}>
>>> state.mem[proj.entry].int.resolved # 将入口点的内存解释为 C 语言的 int 类型
<BV32 0x8949ed31>
```

这里的 BV，即 bitvectors，可以理解为一个比特串，在 angr 里用于表示 CPU 数据。另外可以看到 rdi 在这里有点特殊，它是一个符号变量，没有被赋予一个具体的数值。angr 会在我们使用一个未初始化的内存或者寄存器时发出警告。

下面展示 bitvectors 做简单的数学运算，以及 bitvectors 与 Python 的 int 类型之间的转换关系。需要注意的是 bitvectors 可以通过 state.mem 接口直接给寄存器和内存赋值，但如果传入的是 Python int，则该值会被自动转换成 bitvectors：

```
>>> one = state.solver.BVV(1, 64)
>>> one_hundred = state.solver.BVV(100, 64)
>>> one_hundred + one                       # 位数相同时可以直接运算
<BV64 0x65>
>>> one_hundred + one + 0x100
<BV64 0x165>
>>> state.solver.BVV(-1, 64)                # 默认为无符号数
<BV64 0xffffffffffffffff>

>>> five = state.solver.BVV(5, 27)
>>> five
<BV27 0x5>
>>> one + five.zero_extend(64 - 27)     # 位数不同时需要进行扩展
<BV64 0x6>
>>> one + five.sign_extend(64 - 27)     # 或者有符号扩展
<BV64 0x6>

>>> bv = state.solver.BVV(0x1234, 64)  # 创建值 0x1234 的 BV32 对象
>>> bv
<BV64 0x1234>
>>> hex(state.solver.eval(bv))         # 将 BV64 对象转换为 Python int
```

```
'0x1234'

>>> state.regs.rsi = state.solver.BVV(3, 64)
>>> state.regs.rsi
<BV64 0x3>
>>> state.mem[0x1000].long = 4          # 在地址 0x1000 存放一个 long 类型的值
>>> state.mem[0x1000].long.resolved     # .resolved 获取 bitvectors
<BV64 0x4>
>>> state.mem[0x1000].long.concrete     # .concrete 获得 Python int
4
```

理解了 state 作为一个 SimState 对象表示了程序在某一个节点上的状态，以及其包含的静态数据，我们就可以想象，伴随着程序的模拟执行，其状态也随着所处的节点变化而变化。angr 通过一个接口将这些状态统一管理起来，这就是模拟管理器（simulation manager）。

首先，传入一个或一组 state 即可创建 simulation manager：

```
>>> simgr = proj.factory.simulation_manager(state)
>>> simgr
<SimulationManager with 1 active>      # active 是一个默认 stash
>>> simgr.active                       # 当前的 active state
[<SimState @ 0x4013d0>]
>>> simgr.step()                       # 模拟执行一个 basic block
<SimulationManager with 1 active>
>>> simgr.active                       # 当前 active state 被更新
[<SimState @ 0x401180>]
>>> simgr.active[0].regs.rip           # active[0]表示当前的 active state
<BV64 0x401180>
>>> state.regs.rip                     # 但原始的 state 并没有改变
<BV64 0x4013d0>
>>> simgr.run()                        # 执行到程序结束
<SimulationManager with 1 deadended>
>>> simgr.deadended                    # 当前的 deadended state
[<SimState @ 0x10001c8>]
```

在 simulation manager 中，这些 state 会被分类存储，这些类也就是所谓的 stash。active 作为默认 stash，会在创建 simulation manager 时自动创建并初始化，除此之外，还有 deadended、pruned 等其他的 stash，用于放置不同类型的 state，当然也可以定义自己的 stash。

最后，angr 提供了大量的函数方法用于程序分析，这些函数定义在 Project.analyses 中，感兴趣的读者可以查阅 API 文档，这里我们举一个生成 CFG 的例子：

```
>>> cfg = p.analyses.CFGFast()         # 得到 control-flow graph
>>> cfg
<CFGFast Analysis Result at 0x7fffee9bc630>
```

```
>>> cfg.graph                              # 导出有向图
<networkx.classes.digraph.DiGraph object at 0x7fffee9bc9b0>
>>> len(cfg.graph.nodes())                 # 有向图节点数
554
>>> entry_node = cfg.get_any_node(p.entry)          # 得到给定地址处的节点
>>> entry_node
<CFGNode 0x4013d0[41]>
>>> len(list(cfg.graph.successors(entry_node)))    # 后继节点数
1

>>> import networkx as nx                  # 如果你想要更直观地查看 CFG
>>> import matplotlib
>>> matplotlib.use('Agg')
>>> import matplotlib.pyplot as plt
>>> nx.draw(cfg.graph)
>>> plt.savefig('temp.png')
```

18.2.3 二进制文件加载器

angr 是高度模块化的，接下来我们就分别来看看这些组成模块，其中用于二进制加载的模块称为 CLE。主类为 cle.loader.Loader，它加载所有的对象文件并导出一个进程内存的抽象。类 cle.backends 是加载器的后端，根据二进制文件类型区分为 cle.backends.elf、cle.backends.pe、cle.backends.macho 等。

首先我们来看加载器的一些常用参数：

- auto_load_libs：是否自动加载主对象文件所依赖的共享库；
- except_missing_libs：当有共享库没有找到时抛出异常；
- force_load_libs：强制加载列表指定的共享库，不论其是否被依赖；
- skip_libs：不加载列表指定的共享库，即使其被依赖；
- ld_path：可以到列表指定的路径查找共享库。

如果希望对某个对象文件单独指定加载参数，可以使用 main_ops 和 lib_opts 以字典的形式指定参数。一些通用的参数如下：

- backend：使用的加载器后端，如："elf"、"pe"、"mach-o"、"blob"等；
- arch：使用的 archinfo.Arch 对象；
- base_addr：指定对象文件的基址；
- entry_point：指定对象文件的入口点。

举个例子：

```
>>> proj = angr.Project('/bin/true', auto_load_libs=True, main_opts={'backend':
'elf', 'arch': 'i386'}, lib_opts={'libc.so.6': {'backend': 'elf'}})
```

通过操作这些对象文件，可以进一步得到相关信息，下面列举一些常用的 API：

```
>>> obj = proj.loader.main_object       # 主对象文件
>>> proj.loader.shared_objects          # 共享对象文件
```

```
>>> proj.loader.extern_object          # 外部对象文件
>>> proj.loader.all_elf_objects        # 所有 elf 对象文件
>>> proj.loader.kernel_object          # 内核对象文件

>>> proj.loader.shared_objects['libc.so.6']
<ELF Object libc-2.23.so, maps [0x1000000:0x11b5a1b]>
>>> proj.loader.find_object('libc-2.23.so')           # 通过名字查找对象文件
<ELF Object libc-2.23.so, maps [0x1000000:0x11b5a1b]>
>>> proj.loader.find_object_containing(0x1000000)     # 通过地址查找对象文件
<ELF Object libc-2.23.so, maps [0x1000000:0x11b5a1b]>

>>> for seg in obj.segments:           # segments
...     print(seg)
...
<ELFSegment vaddr=0x400000, flags=0x5, memsize=0x5a24, filesize=0x5a24,
offset=0x0>
<ELFSegment vaddr=0x605e10, flags=0x6, memsize=0x5b0, filesize=0x404,
offset=0x5e10>
>>> for sec in obj.sections:           # sections
...     print(sec)
...
<Unnamed | offset 0x0, vaddr 0x0, size 0x0>
<.interp | offset 0x238, vaddr 0x400238, size 0x1c>
<.note.ABI-tag | offset 0x254, vaddr 0x400254, size 0x20>
<.note.gnu.build-id | offset 0x274, vaddr 0x400274, size 0x24>
...

>>> proj.loader.find_object_containing(0x400000)  # 包含指定地址的 object
<ELF Object true, maps [0x400000:0x6063bf]>
>>> free = proj.loader.find_symbol('free')        # 根据名字或地址在 project 中查
找符号
>>> free
<Symbol "free" in libc.so.6 at 0x1071470>
>>> free.name                                     # 符号名
'free'
>>> free.owner                                    # 所属对象文件
<ELF Object libc-2.23.so, maps [0x1000000:0x11b5a1b]>
>>> hex(free.rebased_addr)                        # 全局地址空间中的地址
'0x1071470'
>>> hex(free.linked_addr)                         # 相对于预链接基址的地址
'0x71470'
>>> hex(free.relative_addr)                       # 相对于对象基址的地址
'0x71470'
>>> free.is_export                                # 是否为导出符号
```

```
True
>>> free.is_import                                    # 是否为导入符号
False

>>> obj.find_segment_containing(obj.entry)         # 包含指定地址的 segment
<ELFSegment vaddr=0x400000, flags=0x5, memsize=0x5a24, filesize=0x5a24,
offset=0x0>
>>> obj.find_section_containing(obj.entry)         # 包含指定地址的 section
<.text | offset 0x1330, vaddr 0x401330, size 0x2cc9>
>>> main_free = obj.get_symbol('free')             # 根据名字在 object 中查找符号
>>> main_free
<Symbol "free" in true (import)>
>>> main_free.resolvedby                           # 从哪个对象文件获得解析
<Symbol "free" in libc.so.6 at 0x1071470>

>>> hex(obj.linked_base)                           # 预链接的基址
'0x400000'
>>> hex(obj.mapped_base)                           # 实际映射的基址
'0x400000'

>>> obj.relocs                                     # 重定位符号信息
>>> for imp in obj.imports:                        # 导入符号
...     print(imp, obj.imports[imp])
...
abort <cle.backends.elf.relocation.i386.R_386_JMP_SLOT object at
0x7fffe52372e8>
fflush <cle.backends.elf.relocation.i386.R_386_JMP_SLOT object at
0x7fffe5244cc0>
mbsinit <cle.backends.elf.relocation.i386.R_386_JMP_SLOT object at
0x7fffe51d15c0>
...
>>> obj.imports['free'].symbol                     # 从重定向信息得到导入符号
<Symbol "free" in true (import)>
>>> obj.imports['free'].owner                      # 从重定向信息得到所属的对象
<ELF Object true, maps [0x400000:0x6063bf]>
```

这一节的最后，我们来看 angr 里一个非常有用的功能——hooking，在这之前，首先得知道什么是 SimProcedures。回想我们在创建工程的时候给 angr.Project 传入了一个参数 auto_load_libs=False，表示不加载程序的依赖库，那么当 angr 在分析程序时遇到了调用的外部函数，就会直接返回一个不受约束的符号值（ReturnUnconstrained）作为该外部函数的返回值。但如果传入了 auto_load_libs=True，则 angr 会对外部函数进行分析，为了防止某些复杂的外部函数可能导致的路径爆炸，也为了提高效率、节约时间，angr 为大量的常用函数实现了简化的版本，即 SimProcedures，在程序分析中常称它们为函数摘要（Function Summaries）。

hooking 就是一种将外部库函数替换为函数摘要的机制。angr 在模拟执行每一条指令之前都会先检测该地址处是否被 hook，如果是就转到指定的代码去执行。使用的函数是 proj.hook(addr, hook)和 proj.hook_symbol(name, hook)，其中参数 hook 是一个 SimProcedure 实例：

```
>>> stub_func = angr.SIM_PROCEDURES['stubs']['ReturnUnconstrained']   # 获得某个 SimProcedure 的类
>>> stub_func
<class 'angr.procedures.stubs.ReturnUnconstrained.ReturnUnconstrained'>

>>> proj.hook(0x10000, stub_func())        # 使用该类的一个实例对地址进行 hook
>>> proj.is_hooked(0x10000)                # 某地址是否被 hook
True
>>> proj.hooked_by(0x10000)                # 某地址的 hook 代码
<SimProcedure ReturnUnconstrained>
>>> proj.unhook(0x10000)                   # 取消 hook

>>> proj.hook_symbol('free', stub_func())  # 对符号进行 hook
17241200
>>> proj.is_symbol_hooked('free')          # 某符号是否被 hook
True
>>> proj.unhook_symbol('free')             # 取消 hook
```

当然，如果内置的 SimProcedure 不够用，还可以编写自己的 hook 函数：

```
>>> def myhook(state):                          # 定义 hook 函数
...     state.regs.rax = 1
...
>>> proj.hook(0x20000, hook=myhook, length=5)  # 执行完 myhook 后跳过 length 个字节
>>> proj.is_hooked(0x20000)
True

>>> @proj.hook(0x30000, length=5)               # 通过装饰器实现
... def my_hook(state):
...     state.regs.rax = 1
...
>>> proj.is_hooked(0x30000)
True
```

18.2.4 求解器引擎

angr 作为一个符号执行工具，通过符号表达式来模拟程序的执行，将程序的输出表示成包含这些符号的逻辑或数学表达式，然后利用约束求解器进行求解，当前 angr 使用 z3 作为求解器后端。

从前面的内容中我们已经知道 bitvectors 是一个比特串，并且看到了 bitvectors 可以做的一些具体的数学运算。其实 bitvectors 不仅可以表示具体的数值，还可以表示虚拟的数值，即符号变量。

```
>>> x = state.solver.BVS("x", 64)
>>> y = state.solver.BVS("y", 64)
>>> x, y
(<BV64 x_0_64>, <BV64 y_1_64>)
```

符号变量之间的运算同样不会是具体的数值，而是一个 AST，angr 同样使用 bitvectors 来指代 AST，另外每个 AST 都包含属性.op 和.args：

```
>>> tree = (x + 1) / (y + 2)
>>> tree
<BV64 (x_0_64 + 0x1) / (y_1_64 + 0x2)>
>>> tree.op                                    # 表示操作符的字符串
'__floordiv__'
>>> tree.args                                  # 操作数
(<BV64 x_0_64 + 0x1>, <BV64 y_1_64 + 0x2>)
>>> tree.args[0].op
'__add__'
>>> tree.args[0].args
(<BV64 x_0_64>, <BV64 0x1>)
>>> tree.args[0].args[1].op
'BVV'
>>> tree.args[0].args[1].args
(1, 64)
```

将 AST 进行比较会得到一个符号化表示的布尔值。同时正因为这样，在需要做 if 或者 while 判断的时候，应该使用.is_true 和.is_false 来确定判断的结果：

```
>>> maybe = x == y                             # 符号化的布尔值
>>> maybe
<Bool x_0_64 == y_1_64>
>>> state.solver.BVV(1, 64) > 0                # 无符号数 1
<Bool True>
>>> yes = state.solver.BVV(-1, 64) > 0         # 无符号数 0xffffffffffffffff
>>> yes
<Bool True>
>>> no = state.solver.BVV(-1, 64).SGT(0)       # 有符号数比较
>>> no
<Bool False>

>>> state.solver.is_true(yes)
```

```
True
>>> state.solver.is_false(yes)
False
>>> state.solver.is_true(maybe)
False
>>> state.solver.is_false(maybe)
False
```

这些符号化的布尔值可以作为符号变量有效值的断言加入到 state 中作为限制条件，然后对其进行求解即可得到有效值。当然如果添加了无法满足的限制条件，求解就会失败：

```
>>> state.solver.add(x > y)                 # 添加限制条件
[<Bool x_0_64 > y_1_64>]
>>> state.solver.add(y > 2)
[<Bool y_1_64 > 0x2>]
>>> state.solver.add(10 > x)
[<Bool x_0_64 < 0xa>]
>>> state.satisfiable()                     # 判断条件是否可满足
True

>>> state.solver.eval(x + y)                # 求解，获得任意一个符合条件的值
15
>>> state.solver.eval_one(x + y)            # 求解，当有不止一个有效值时抛出异常
angr.errors.SimValueError: Concretized 2 values (must be exactly 1) in eval_exact
>>> state.solver.eval_upto(x + y, 5)        # 求解，获得最多 5 个值
[7, 10, 15, 8, 9]
>>> state.solver.eval_atleast(x + y, 5)     # 求解，获得至少 5 个值，否则抛出异常
[7, 10, 15, 8, 9]
>>> state.solver.eval_exact(x + y, 5)       # 求解，正好有 5 个值，否则抛出异常
angr.errors.SimValueError: Concretized 6 values (must be exactly 5) in eval_exact
>>> state.solver.min(x + y)                 # 求解，获得最小值
7
>>> state.solver.max(x + y)                 # 求解，获得最大值
17
>>> state.solver.eval(x + y, extra_constraints=[x + y < 10, x + y > 5])
                                            # 添加临时限制条件
9
>>> state.solver.eval(x + y, cast_to=bytes)    # 指定输出格式
b'\x00\x00\x00\x00\x00\x00\x00\t'

>>> state.solver.add(x - y > 10)            # 添加不可满足的限制条件
[<Bool (x_0_64 - y_1_64) > 0xa>]
>>> state.satisfiable()
False
```

18.2.5 程序状态

state.step()用于模拟执行的一个基本块并返回一个 SimSuccessors 类型的对象，由于符号执行可能产生多个 state，所以该对象的.successors 属性是一个列表，包含了所有可能的 state。

程序状态 state 是一个 SimState 类型的对象，angr.factory.AngrObjectFactory 类提供了创建 state 对象的方法：

- .blank_state()：返回一个几乎没有初始化的 state 对象，当访问未初始化的数据时，将返回一个没有约束条件的符号值。
- .entry_state()：从主对象文件的入口点创建一个 state。
- .full_init_state()：与 entry_state()类似，但执行不是从入口点开始，而是从一个特殊的 SimProcedure 开始，在执行到入口点之前调用必要的初始化函数。
- .call_state()：创建一个准备执行给定函数的 state。

下面对这些方法的参数做一些说明：

- 所有方法都可以传入参数 addr 来指定开始地址。
- 可以通过参数 args 传入命令行参数列表，env 传入环境变量。
- 通过传入一个 bitvector 作为 argc，可以将 argc 符号化。
- 对于.call_state(addr, arg1, arg2, ...)，addr 是希望调用的函数地址，argN 是传递给函数的 N 个参数，如果希望分配一个内存空间并传递指针，则需要使用 angr.PointerWrapper()。
- 如果需要指定调用约定，可以传递一个 SimCC 对象作为 cc 参数。

有时，我们在不同的条件下需要对某个状态赋予不同的值，以探索更多的可能性，使用下面的方法可以很方便地进行状态复制和合并：

```
>>> s1 = state.copy()                                    # 复制 state
>>> s2 = state.copy()
>>> s1.mem[0x1000].uint32_t = 0x41414141
>>> s2.mem[0x1000].uint32_t = 0x42424242

>>> (s_merged, m, anything_merged) = s1.merge(s2)        # 合并，返回一个元组
>>> s_merged                                             # 合并后的 state
<SimState @ 0x4013d0>
>>> m                                                    # 描述 state flag
[<Bool state_merge_0_2_16 == 0x0>, <Bool state_merge_0_2_16 == 0x1>]
>>> anything_merged                                      # 是否完全合并完成
True
>>> aaaa_or_bbbb = s_merged.mem[0x1000].uint32_t         # 值根据 state flag 来判断
>>> aaaa_or_bbbb
<uint32_t <BV32 Reverse((if (state_merge_0_2_16 == 0x1) then 0x42424242 else (if
(state_merge_0_2_16 == 0x0) then 0x41414141 else 0x0)))> at 0x1000>
```

对于 state 内存的操作，不仅可以使用 state.mem 接口，还可以使用 state.memory

的.load(addr, size)和.store(addr, val)。前者默认使用小端序，而后者默认使用大端序，同时可以通过设置参数 endness 来修改：

```
>>> state.memory.store(0x4000, state.solver.BVV(0x0123456789abcdef, 128))
                                                    # 默认大端序
>>> state.memory.load(0x4008, 8)                    # 默认大端序
<BV64 0x123456789abcdef>
>>> state.memory.load(0x4008, 8, endness=angr.archinfo.Endness.LE)    # 小端序
<BV64 0xefcdab8967452301>
>>> state.mem[0x4008].uint64_t.resolved             # 与 mem 对比
<BV64 0xefcdab8967452301>
```

18.2.6 模拟管理器

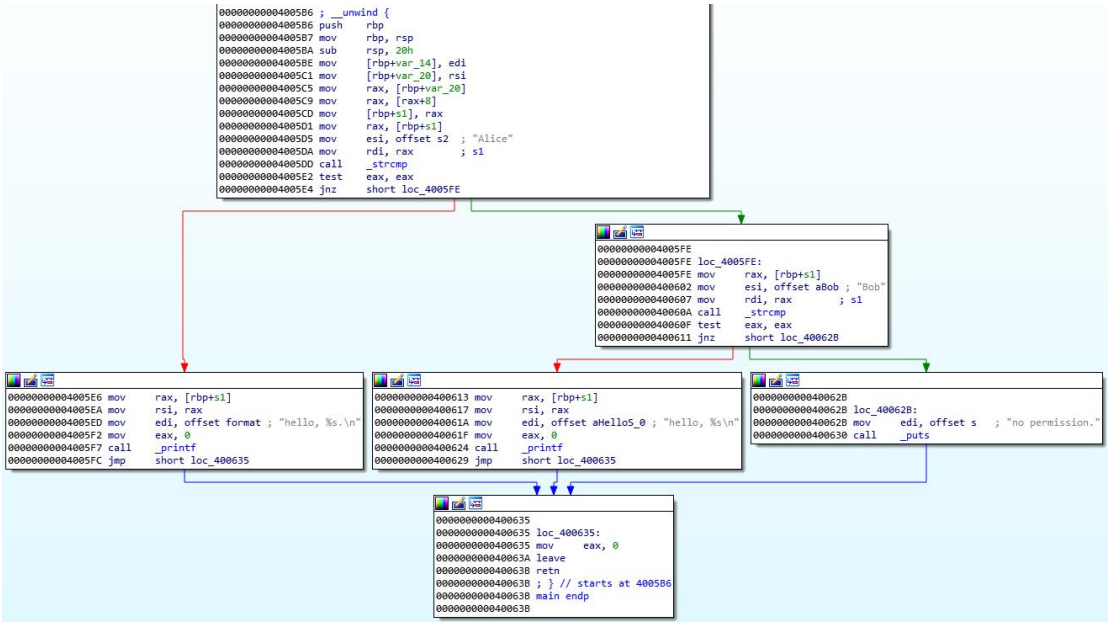
模拟管理器（Simulation Managers）是 angr 最重要的控制接口，它可以对各组 states 的符号执行进行控制，同时应用搜索策略来探索程序的状态空间。states 会被整理到各自的 stash 里，从而进行各种操作。

我们用一个小程序来作例子，它从命令行读入一个名字，判断是否为授权用户，输出有 3 种不同的结果（注释是 IDA 反汇编后各语句所属基本块的地址）：

```
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv) {
   char *name = argv[1];
   if (strcmp(name, "Alice") == 0) {              // 0x004005B6
      printf("hello, %s.\n", name);               // 0x004005E6
   } else if (strcmp(name, "Bob") == 0) {         // 0x004005FE
      printf("hello, %s\n", name);                // 0x00400613
   } else {
      printf("no permission.\n");                 // 0x0040062B
   }
}
```

main 函数在 IDA 中的程序流图如下所示：



假设授权用户的 name 是未知的，但知道授权认证成功后会打印出一段包含”hello”的字符串，那么，我们可以将命令行参数 name 声明为符号变量，将”hello”作为探索条件，通过符号执行到达相应的 state，然后进行约束求解即可求得 name：

```
>>> import angr, claripy
>>> proj = angr.Project('a.out', auto_load_libs=False)
>>> args = claripy.BVS("args", 64)                # 声明符号变量
>>> state = proj.factory.entry_state(args=['./args_test', args])  # 命令行参数
>>> simgr = proj.factory.simgr(state)             # 创建 Simulation Manager

>>> while simgr.active:
...     simgr.explore(find=lambda s: b"hello" in s.posix.dumps(1))
                                                  # 探索程序，find 指定搜索条件
...
<SimulationManager with 2 found, 1 deadended>
>>> simgr.found                                   # 结果保存在 found stash 中
[<SimState @ 0x4005fc>, <SimState @ 0x400629>]
>>> simgr.found[0].posix.dumps(1)                 # 该 state 的标准输出
b'hello, Alice.\n'
>>> simgr.found[0].solver.eval(args, cast_to=bytes)    # 求解该 state 需要的 args
b'Alice\x00\x80\x00'
>>> simgr.found[1].posix.dumps(1)
b'hello, Bob.\n'
>>> simgr.found[1].solver.eval(args, cast_to=bytes)
b'Bob\x00\x80\x01\x00\x00'
```

.explore(avoid=avoid, find=end, n=N)是模拟管理器一个非常重要的方法，用于从 active stash 开始符号执行，并找到至多 n 个 find 条件匹配的 state，忽略 avoid 条件匹配

的 state，将它们分别放入 found stash 和 avoid stash。find 和 avoid 参数可以是一个或者一组地址，也可以是一个函数（输入 state，返回是否匹配）。

.posix.dumps(1)用于打印出某 state 的标准输出，同理，如果程序存在标准输入，则.posix.dumps(0)用于打印出到达某 state 所需标准输入的内容。

stash 被用于存储具有不同特征的 state，模拟管理器中内置了 active、deadended、pruned、unconstrained、unsat 等不同的 stash。.move(from_stash, to_stash, filter_func=None)方法用于移动 stash 里的 state，从中还衍生出了.stash()、.unstash()、.drop()等方法。

```
>>> simgr.move(from_stash='found', to_stash='bob', filter_func=lambda s: b'Bob'
in s.posix.dumps(1))                          # 将 Bob 的 state 移动到 bob stash
<SimulationManager with 1 found, 1 deadended, 1 bob>
>>> simgr.bob
[<SimState @ 0x400629>]
```

18.2.7 VEX IR 翻译器

PyVEX 作为 angr 的重要组成模块，承担着二进制代码与 VEX IR 之间转化的任务。VEX IR 原是 Valgrind 项目开发和使用的 IR，后来这一部分被分离出去作为 libVEX。模块 PyVEX 就是 libVEX 的 Python 包装。

下面简单列举 PyVEX 的几个 API：

```
>>> import pyvex, archinfo
>>> bb = pyvex.IRSB(b'\xc3', 0x400400, archinfo.ArchAMD64())
                        # 将一个位于 0x400400 的 AMD64 基本块'\xc3'（即 ret）转成 VEX
>>> bb.pp()
IRSB {
   t0:Ity_I64 t1:Ity_I64 t2:Ity_I64 t3:Ity_I64 t4:Ity_I64

   00 | ------ IMark(0x400400, 1, 0) ------
   01 | t0 = GET:I64(rsp)
   02 | t1 = LDle:I64(t0)
   03 | t2 = Add64(t0,0x0000000000000008)
   04 | PUT(rsp) = t2
   05 | t3 = Sub64(t2,0x0000000000000080)
   06 | ====== AbiHint(0xt3, 128, t1) ======
   NEXT: PUT(rip) = t1; Ijk_Ret
}
>>> bb.statements[3].pp()                     # 表达式
t2 = Add64(t0,0x0000000000000008)
>>> bb.statements[3].data.pp()                # 数据
Add64(t0,0x0000000000000008)
>>> bb.statements[3].data.op                  # 操作符
'Iop_Add64'
```

```
>>> bb.statements[3].data.args[1].pp()     # 参数
0x0000000000000008
>>> bb.next.pp()                          # 基本块末尾无条件跳转的目标
t1
>>> bb.jumpkind                           # 无条件跳转的类型
'Ijk_Ret'
```

18.2.8 总结

到这里 angr 的核心概念就介绍得差不多了，更多更详细的内容推荐查看官方教程和 API 文档。最后，列举几个基于 angr 的强大能力所开发出的程序分析工具，感兴趣的读者也可以尝试进行开发，不断拓宽 angr 的能力范围：

- angrop：rop 链自动化生成器；
- Patcherex：二进制文件自动化 patch 引擎；
- Driller：用符号执行增强 AFL 的下一代 fuzzer；
- Rex：自动化漏洞利用引擎；