前言

- angr是BlackHat2015上公布的一款二进制自动化分析工具。在今年（2019）复盘DDCTF的时候看到夜影大佬的WP并尝试使用这个工具解决类似ollvm的反混淆问题。在查找文档的时候才发现国内相关的介绍比较少。一方面可能是因为这个工具本身比较复杂，另外一方面也是因为这个文档仍然在不断更新中。但是由于急着要用，就萌生了将文档翻译过来的想法。
- 当然需要说明的是由于本人水平限制，这个文档几乎就是在机翻的基础上对于一些翻译错误的专有词汇以及一些长句结合自己的理解进行了处理。大致看了一下还是比直接看英文文档稍微好一些，因此也就斗胆把这个文档上传，希望能够帮到像我一样的英文不太好的同学。同时由于本人水平以及时间原因本文档会不可避免的出现较多错误或者表述不当，因此对于有问题的地方还请大家多多包涵，也欢迎大家指出问题与改正文档，一切以angr原文档表述为准。
- 此外，由于angr更新换代仍然在不停的进行中，本文档是截止到翻译完成日期时（2019/5/30）angr文档的Core Concept部分的翻译，此后angr文档以及后续版本的接口可能会出现较大变动，希望及时关注angr官网。

Top Level Interfaces

- 在开始学习angr之前，您需要对一些基本的angr概念以及如何构造一些基本的angr对象有一个基本的了解。我们将通过检查在您加载了二进制文件之后，您可以直接调用的函数以及属性来研究这个问题!
- 使用angr的第一个操作总是将二进制文件加载到项目中。我们将使用/bin/true作为示例。

```
>>> import angr
>>> proj=angr.Project('/bin/true')
```

- proj(`angr.Project`类型）是angr中的控制基础。有了它，您将能够在刚刚加载的可执行文件上执行分析和模拟的动作。在angr中,几乎每个对象的使用都依赖于某种形式的proj的存在。

Basic properties

- 首先，我们拥有关于项目的一些基本属性:它的CPU架构、文件名和入口点的地址（OEP）。

```
>>> import monkeyhex      # 将数字格式的结果转换为16进制
>>> proj.arch
<Arch AMD64 (LE)>
>>> proj.entry
0x401670
>>> proj.filename
'/bin/true'
```

- `arch`，作为`archinfo.Arch`结构的一个实例，是程序编译的架构（Archtecture）。此程序的结构是小端字节序的amd64结构。其中包含了大量CPU运行时的数据。您可以在您空闲时阅读。您所关注的共通的是`arch.bits`，`arch.bytes`（这是一个 main 处的一个`@property`声明），`arch.name`以及`arch.memory_endness`等类型。
- `entry`是二进制文件的入口点
- `filename`是二进制文件的绝对路径名

The loader

- 将二进制文件从其在虚拟地址空间中加载获取是一件非常复杂的事!因此我们通过一个叫CLE的模块来完成这件事。而CLE的结果，称之为加载器的对象，可以通过`.loader`获得。我们将很快详细介绍如何使用它，但现在您只需要知道您可以使用它来查看与程序一同被angr加载的共享库，并执行有关被加载的地址空间的基本访问。

```
>>> proj.loader
<Loaded true, maps [0x400000:0x5004000]>
>>> proj.loader.shared_objects # 可能与您使用的时候的显示结果有所不同（不
{'ld-linux-x86-64.so.2': <ELF Object ld-2.24.so, maps [0x2000000
 'libc.so.6': <ELF Object libc-2.24.so, maps [0x1000000:0x13c699
>>> proj.loader.min_addr
0x400000
>>> proj.loader.max_addr
```

```
0x5004000
>>> proj.loader.main_object  # 我们在这个proj中同时加载了多个二进制文件
Here's the main one!
<ELF Object true, maps [0x400000:0x60721f]>
>>> proj.loader.main_object.execstack  # 查询: 该二进制文件的栈空间是
False
>>> proj.loader.main_object.pic  # 查询: 该二进制文件是否开启了PIE
True
```

The factory

- angr中有很多类，其中大多数需要实例化一个project。我们提供`project.factory`来避免您到处传递project。它有几个方便的构造函数，用于处理您经常使用的对象。
- 本节还将介绍几个基本的angr概念。未完待续！！

Blocks

- 首先，我们有`project.factory.block()`，它用于从给定地址提取基本代码块。这里有一个非常重要的事实——**angr以基本块为单位分析代码**。你会得到一个Block的返回对象，它将告诉你关于代码块的很多有趣的事情:

```
>>> block = proj.factory.block(proj.entry) # 从程序的入口点获取一个代
<Block for 0x401670, 42 bytes>
>>> block.pp()                              # 将反汇编输出到标准输出
0x401670:       xor     ebp, ebp
0x401672:       mov     r9, rdx
0x401675:       pop     rsi
0x401676:       mov     rdx, rsp
0x401679:       and     rsp, 0xfffffffffffffff0
0x40167d:       push    rax
0x40167e:       push    rsp
0x40167f:       lea     r8, [rip + 0x2e2a]
0x401686:       lea     rcx, [rip + 0x2db3]
0x40168d:       lea     rdi, [rip - 0xd4]
0x401694:       call    qword ptr [rip + 0x205866]
>>> block.instructions                      # 代码块一共有多少指令？
0xb
```

```
>>> block.instruction_addrs              # 指令的开始地址？
[0x401670, 0x401672, 0x401675, 0x401676, 0x401679, 0x40167d, 0x4
```

- 此外，你可以使用Block对象来获得代码块的其他表示形式:

```
>>> block.capstone                           # capstone的反汇编块（另一
<CapstoneBlock for 0x401670>
>>> block.vex                                # VEX IRSB (此处地址是pyt
<pyvex.block.IRSB at 0x7706330>
```

States

- 这里还有一个关于angr的事实——对象`Projectt`只表示程序的“初始化图像”。当您使用angr执行时，您使用的是表示**模拟程序状态的特定对象**—`SimState`。我们现在就去获取一个吧!

```
>>> state = proj.factory.entry_state()
<SimState @ 0x401670>
```

- SimState包含程序的内存、寄存器、文件系统数据…任何可以通过执行来更改的“活动数据”都可以在状态中找到。稍后我们将深入讨论如何与状态交互，但是现在，让我们使用`state.regs`以及`state.mem`来访问该状态的寄存器和内存:

```
>>> state.regs.rip         # 获取当前状态的RIP
<BV64 0x401670>
>>> state.regs.rax
<BV64 0x1c>
>>> state.mem[proj.entry].int.resolved  # 将内存中入口点处数据转化为一
<BV32 0x8949ed31>
```

- 注意，此处并非python的int，而是**位向量**。python的interger类型与CPU上的数据含义不同。例如CPU上的溢出在python的int类型中并不存在。因此我们在angr中使用位向量。你可以将其理解为一个用一串bit来表示的int。我们以此来模拟CPU中的数据。注意，每个位向量都有一个`.length`属性，以位为单位描述它的宽度。

- 我们将很快学习如何使用它们，但现在，我们先学习一下如何将 python int转换为位向量，然后再转换回来:

```
>>> bv = state.solver.BVV(0x1234, 32)         # 创建一个32位大小的，数
<BV32 0x1234>                                 # BVV 代表位向量的值
>>> state.solver.eval(bv)                   # 转换回python的int类型
0x1234
```

- 你可以把这些位向量存储回寄存器和内存，或者你可以直接存储一个python整数，它会被转换成一个适当大小的位向量:

```
>>> state.regs.rsi = state.solver.BVV(3, 64)
>>> state.regs.rsi
<BV64 0x3>
>>> state.mem[0x1000].long = 4
>>> state.mem[0x1000].long.resolved
<BV64 0x4>
```

- `.mem`接口一开始有点令人困惑，因为它使用了一些非常强大的python magic 。它的使用简介如下:
- 使用 array[index] 来获取一个特定地址的数据
- 使用`.<type>`指定内存应该转换为指定格式(`.type`常见值:char、short、int、long、size_t、uint8_t、uint16_t...)
- 除此之外，您也可以：
    - 存储一个值到指定地址，无论是位向量或者python int都是可行的
    - 使用`.resolve`获取作为位向量的值
    - 使用`.concrete`获取一个python int的值
- 之后我们将介绍更多高级用法!
- 最后，如果您尝试读取更多寄存器，您可能会遇到一个非常奇怪的值:

```
>>> state.regs.rdi
<BV64 reg_48_11_64{UNINITIALIZED}>
```

- 这仍然是一个64位位向量，但它不包含数值。相反，它有一个名字!这被称为**符号变量**，它是符号执行的基础。别慌!我们将在接下来的两章详细讨论所有这些。

Simulation Managers

- 如果一个状态能够在给定的时间点上代表一个程序，那么一定有一种方法可以让它到达下一个时间点。仿真管理器是angr中执行带状态仿真的主要接口(不管您想叫它什么)。作为一个简短的介绍，让我们展示如何标记前面创建的几个基本块的状态。
- 首先，我们创建将要使用的仿真管理器。构造函数可以接受状态或状态列表作为参数。

```
>>> simgr = proj.factory.simulation_manager(state)
<SimulationManager with 1 active>
>>> simgr.active
[<SimState @ 0x401670>]
```

- 仿真管理器可以包含多个隐藏的状态。默认的隐藏状态，`active`，是用我们传入的状态初始化的。我们可以看看`simgr.active[0]`来查看我们的状态。
- 现在…准备好，我们要开始执行了。

```
>>> simgr.step()
```

- 我们刚刚执行了一个基本块的符号执行!我们可以再次查看活动的隐藏状态，注意到它(仿真管理器)已经更新，而且没有修改我们的原始状态。执行无法修改SimState对象——您可以安全地使用单个状态作为多轮执行的“基础”。

```
>>> simgr.active
[<SimState @ 0x1020300>]
>>> simgr.active[0].regs.rip                # 全新的状态!
<BV64 0x1020300>
>>> state.regs.rip                          # 仍然相同！
<BV64 0x401670>
```

- `/bin/true`不是一个很好的例子来描述如何用符号执行来做有趣的事情，所以我们现在就到这里。

Analyses

- angr预先包含了几个内置的分析模块，您可以使用这些分析模块从程序中提取一些有趣的信息。如下所示:

```
>>> proj.analyses.             # 在iPython中输入到此，之后按TAB键列出自
 proj.analyses.BackwardSlice         proj.analyses.CongruencyChec
 proj.analyses.BinaryOptimizer       proj.analyses.DDG
 proj.analyses.BinDiff               proj.analyses.DFG
 proj.analyses.BoyScout              proj.analyses.Disassembly
 proj.analyses.CDG                   proj.analyses.GirlScout
 proj.analyses.CFG                   proj.analyses.Identifier
 proj.analyses.CFGEmulated           proj.analyses.LoopFinder
 proj.analyses.CFGFast               proj.analyses.Reassembler
```

- 本书后面将对其中的一些进行说明，但是一般来说，如果您想了解如何使用给定的分析，您应该查看api文档。举一个非常简单的例子:下面是如何构造和使用一个快速的控制流程图:

```
# 一般情况下，当我们用angr加载二进制文件时，angr也同时加载了该二进制文件的依赖
# 对于大部分的分析来说，这个是没有必要的
>>> proj = angr.Project('/bin/true', auto_load_libs=False)
>>> cfg = proj.analyses.CFGFast()
<CFGFast Analysis Result at 0x2d85130>

# cfg.graph i是一个用CFGNode的实例组成的networkx.DiGraph
# 您应该参考networkx的API来学习如何使用它
>>> cfg.graph
<networkx.classes.digraph.DiGraph at 0x2da43a0>
>>> len(cfg.graph.nodes())
951

# 使用cfg.get_any_node来获取指定地址的CFGNode
>>> entry_node = cfg.get_any_node(proj.entry)
>>> len(list(cfg.graph.successors(entry_node)))
2
```

Now What ？

- 阅读了本文之后，您现在应该掌握了几个重要的angr概念:基本块、状态、位向量、模拟管理器和分析。不过，除了使用angr作为一个美化的调试器之外，您实际上不能做任何有趣的事情!继续阅读，你会获得更强大的力量……

Loading a Binary

- 在此之前，您只看到了angr作为加载工具的最简单的功能—您加载了`/bin/true`，然后在没有共享库的情况下再次加载它。您也看到了`proj.loader`和一些它能做的事情。现在，我们将深入研究这些接口的细微差别以及它们可以告诉您的东西。
- 我们先前简要介绍了angr的二进制文件加载器CLE。CLE代表“CLE加载所有东西”，负责获取二进制文件(以及它所依赖的任何库)，并以一种易于使用的方式将其提供给angr的其余部分。

The loader

- 让我们加载`examples/fauxware/fauxware`并深入了解它如何与加载器交互

```
>>> import angr, monkeyhex
>>> proj = angr.Project('examples/fauxware/fauxware')
>>> proj.loader
<Loaded fauxware, maps [0x400000:0x5008000]>
```

Loaded Objects

- CLE加载器(`cle.Loader`)表示所有被加载的二进制对象的组合。它将所有二进制对象加载并映射到单个内存空间。每个二进制对象都由一个后

端的加载器加载，加载器后端装载器可以处理该二进制文件的文件类型(`cle.Backend`的一个子类)。例如,`cle.ELF`用于加载ELF二进制文件。

- 内存中也会有与任何加载的二进制文件都对应不上的对象。例如，用于支持本地线程存储的对象和用于提供未解析符号的扩展对象。
- 您可以通过`loader.all_objects`获得CLE已加载对象的完整列表，以及一些更有针对性的分类：

```
# 已加载对象的完整列表
>>> proj.loader.all_objects
[<ELF Object fauxware, maps [0x400000:0x60105f]>,
 <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]>]

# 这是“main”对象，该对象在加载项目时直接指定
>>> proj.loader.main_object
<ELF Object fauxware, maps [0x400000:0x60105f]>

# 这是一个以对象名称为键，对象为键值的字典映射
>>> proj.loader.shared_objects
{ 'fauxware': <ELF Object fauxware, maps [0x400000:0x60105f]>,
  'libc.so.6': <ELF Object libc-2.23.so, maps [0x1000000:0x13c99
  'ld-linux-x86-64.so.2': <ELF Object ld-2.23.so, maps [0x200000

# 以下是所有的从ELF文件加载的对象
# 如果这是Windows的项目，我们会使用all_pe_objects!
>>> proj.loader.all_elf_objects
[<ELF Object fauxware, maps [0x400000:0x60105f]>,
 <ELF Object libc-2.23.so, maps [0x1000000:0x13c999f]>,
 <ELF Object ld-2.23.so, maps [0x2000000:0x2227167]>]

# 以下是我们为无法解析的导入部件以及angr的内部部件提供的地址，我们称之为"扩展对
>>> proj.loader.extern_object
<ExternObject Object cle##externs, maps [0x4000000:0x4008000]>

# 这个对象用于为仿真时的系统调用提供地址
>>> proj.loader.kernel_object
<KernelObject Object cle##kernel, maps [0x5000000:0x5008000]>

# 最后，你可以通过给定地址来查找该地址所属的对象
```

```
>>> proj.loader.find_object_containing(0x400000)
<ELF Object fauxware, maps [0x400000:0x60105f]>
```

- 您可以直接与这些对象交互，从中提取元数据:

```
>>> obj = proj.loader.main_object

# 对象的入口点
>>> obj.entry
0x400580

>>> obj.min_addr, obj.max_addr
(0x400000, 0x60105f)

# 搜索这个ELF文件的段以及区块
>>> obj.segments
<Regions: [<ELFSegment memsize=0xa74, filesize=0xa74, vaddr=0x40
           <ELFSegment memsize=0x238, filesize=0x228, vaddr=0x60
>>> obj.sections
<Regions: [<Unnamed | offset 0x0, vaddr 0x0, size 0x0>,
           <.interp | offset 0x238, vaddr 0x400238, size 0x1c>,
           <.note.ABI-tag | offset 0x254, vaddr 0x400254, size 0
            ...etc

# 你可以通过一个地址获取该地址所属的段或区块
>>> obj.find_segment_containing(obj.entry)
<ELFSegment memsize=0xa74, filesize=0xa74, vaddr=0x400000, flags
>>> obj.find_section_containing(obj.entry)
<.text | offset 0x580, vaddr 0x400580, size 0x338>

# 通过符号获取其在plt表中的地址
>>> addr = obj.plt['strcmp']
>>> addr
0x400550
>>> obj.reverse_plt[addr]
'strcmp'

# 展示该文件的静态链接的基地址以及CLE实际将其装载到的基地址
>>> obj.linked_base
0x400000
>>> obj.mapped_base
0x400000
```

Symbols and Relocations

- 您还可以使用CLE处理符号。符号是可执行格式世界中的一个基本概念，能够有效地将符号名映射到地址。
- 从CLE获取符号的最简单方法是使用 `loader.find_symbol` ，它可以接受名称或地址作为参数，并返回一个Symbol对象。
- 符号上最有用的属性是它的名称、所属对象和地址，但是符号的“地址”可能是模糊的。符号对象有三种方式表示其地址:
- `.rebased_addr` 是其在全局地址空间中的地址。这是输出中显示的内容。
- `.linked_addr` 是它相对于二进制预链接基址的地址。例如： `readelf (1)` 就是其返回的一个结果
- `.relative_addr` 是它相对于对象基地址的地址。这在文献(尤其是Windows文献)中称为RVA(相对虚拟地址)。

```
>>> strcmp.name
'strcmp'

>>> strcmp.owner
<ELF Object libc-2.23.so, maps [0x1000000:0x13c999f]>

>>> strcmp.rebased_addr
0x1089cd0
>>> strcmp.linked_addr
0x89cd0
>>> strcmp.relative_addr
0x89cd0
```

- 除了提供调试信息外，符号还支持动态链接的概念。libc提供strcmp符号作为导出函数，而主二进制文件依赖于它。如果我们要求CLE直接从主对象中给我们一个strcmp符号，它会告诉我们这是一个导入符号。导入符号没有与它们关联的有意义的地址，但是它们提供了一个对符号的引用，用于解析导入符号，如 `.resolvedby` 。

```
>>> strcmp.is_export
True
>>> strcmp.is_import
```

```
False

# 在Loader上，方法是find_symbol，因为它执行搜索操作来查找符号。
# 对于单个对象，方法是get_symbol，因为一个名称只能有一个符号。
>>> main_strcmp = proj.loader.main_object.get_symbol('strcmp')
>>> main_strcmp
<Symbol "strcmp" in fauxware (import)>
>>> main_strcmp.is_export
False
>>> main_strcmp.is_import
True
>>> main_strcmp.resolvedby
<Symbol "strcmp" in libc.so.6 at 0x1089cd0>
```

- 导入和导出之间的链接的特定方式应当在内存中注册，并由另一个称为重定位的概念处理。重定位表示，“当您将`[import]`与导出符号匹配时，请将导出地址写入`[location]`，格式为`[format]`。”我们可以通过`obj.relocs`看到一个对象(作为重定位实例)的完整重定位列表，或者只是一个从符号名到重定位的映射（`obj,imports`）。
- 可以通过`.symbol`访问重定位对应的导入符号。而任何可以用作符号的地址标识符，都可用于访问重定位将写入的地址。此外，对于请求重定位的对象，您还可以使用`.owner`获得对该对象的引用。

```
# 重定位不是很好打印，因此这些地址是python程序内部的，与我们的程序无关
>>> proj.loader.shared_objects['libc.so.6'].imports
{'__libc_enable_secure': <cle.backends.elf.relocation.amd64.R_X8
 '__tls_get_addr': <cle.backends.elf.relocation.amd64.R_X86_64_J
 '_dl_argv': <cle.backends.elf.relocation.amd64.R_X86_64_GLOB_DA
 '_dl_find_dso_for_object': <cle.backends.elf.relocation.amd64.R
 '_dl_starting_up': <cle.backends.elf.relocation.amd64.R_X86_64_
 '_rtld_global': <cle.backends.elf.relocation.amd64.R_X86_64_GLO
 '_rtld_global_ro': <cle.backends.elf.relocation.amd64.R_X86_64_
```

- 例如，如果由于找不到共享库的原因，我们不能将导入解析为任何导出。那么在该种情况下CLE将把externs对象(`loader.extern_obj`)自动更新，以声明它，进而将符号作为导出提供。

Loading Options

- 如果在您用angr加载 `angr.Project` 时希望将一个选项传递给项目隐式创建的 `cle.Loader` 实例，那么您可以直接将关键字参数传递给项目构造函数，它将被传递给CLE。如果您想查看所有可以传递的选项，您可以查看CLE API docs，在此处我们只介绍一些重要的或频繁使用的选项

Basic Options

- 我们先前已经讨论过 `auto_load_libs` 选项了——它允许或禁止CLE自动解析共享库依赖关系，并且默认选项为允许。此外，还有一个相反的选项 `except_missing_libs` ，如果将其设置为true，当二进制文件具有无法解析的共享库依赖关系时，就会引发异常。

- 你可以传递一个字符串列表给 `force_load_libs` ,其中列出的所有都将被视为无法解析的共享库依赖。或者你可以传递一个字符串列表给 `skip_libs` 以防止任何库的名称解析为依赖。此外，您可以将字符串列表(或单个字符串)传递给 `ld_path` 。在搜索默认路径中的共享库前，我们将从 `ld_path` 中的路径搜索共享库。默认搜索路径有：与加载的程序相同的目录、当前工作目录和系统库。

Pre-Binary Options

- 如果您想指定一些只适用于特定二进制对象的选项，CLE也会让您这样做。参数 `main_ops` 和 `lib_opts` 通过接受选项字典来实现这一点。 `main_opts` 是一个从选项名到选项值的映射，而 `lib_opts` 是一个从库名到字典的映射，将选项名映射到选项值。

- 你可以使用的选项因backend而异，但一些常见的选项是:

- `backend` -使用哪个后端装载器作为类或名称

- `base_addr` -使用的基地址
- `entry_point` -使用的入口点

- `arch` -使用的架构

- 例：

```
>>> angr.Project('examples/fauxware/fauxware', main_opts={'backe
<Project examples/fauxware/fauxware>
```

Backends

- CLE目前有用于静态加载ELF、PE、CGC、Mach-O和ELF内核dump文件的后端装载器，以及用IDA加载二进制文件和将文件加载到平面地址空间的后端装载器。CLE将在大多数情况下自动检测到要使用的正确后端装载器，所以您不应该指定使用哪个后端装载器，除非您正在做一些非常奇怪的事情。
- 您可以通过在对象的选项字典中包含一个键来强制CLE为对象使用特定的后端装载器，如上所述。有些后端装载器不能自动检测要使用哪个体系结构，必须指定架构。字典的键不需要匹配任何架构的列表;angr将识别您所表示的体系结构，为任何受支持的arch提供几乎任何通用标识符。
- 若要引用后端装载器，请使用下表中的名称:

装载器名字	描述	需要指定架构与否
elf	ELF文件的静态装载器，基于PyElFTools	否
pe	PE文件的静态装载器，基于pefile	否
mach-o	mach-o文件的静态装载器，不支持动态链接或基址重定位	否
cgc	CyperGrandChallenge二进制文件的静态加载器	否
backedcgc	CGC二进制文件静态加载器，允许指定内存与寄存器	否
elfcore	elf内核转存文件的静态加载器	否
ida	运行一个IDA的实例来分析文件	是
blob	以平面镜像的形式加载文件到内存	是

Symbolic Function Summary

- 默认情况下，Project试图通过使用称为SimProcedures的符号摘要来替换对库函数的外部调用——实际上就是模仿库函数对状态的影响的python函数。我们实现了一系列SimProcedure函数。我们可以在`angr.SIM_PROCEDURES`中使用这些函数。同时，`angr.SIM_PROCEDURES`的字典是两级的，第一级的键是包名称(libc、posix、win32、存根)，然后每个键值都是一个字典，键值是库函数的名称。执行SimProcedure而不是从系统中加载的实际库函数，可以使分析更加容易处理，但可能会导致一些错误。

- 当某一给定函数没有摘要时:

- 如果`auto_load_libs`为真(这是默认值)，则执行实际的库函数。这可能是您想要的，也可能不是，这取决于实际的函数。例如，libc的一些函数分析起来非常复杂，并且很可能会导致尝试执行它们的路径的状态数暴增。

- 如果`auto_load_libs`为假，那么外部函数将无法解析，项目将把它们解析为一个名为`ReturnUnconstrained`的通用“Stub”SimProcedure。该状态，如同其名称所说，每次调用它时，它都返回一个惟一的无约束符号值。

- 当`use_sim_procedures`（不是`cle.Loader`的成员而是`angr.Project`的成员）被设置为假（默认为真），那么只有扩展对象提供的符号将被SimProcedure替换，它们将被`ReturnUnconstrained`的stub替换，该stub只返回一个符号值。

- 通过`angr.Project`中的选项`exclude_sim_procedures_list`或者`exclude_sim_procedures_func`，您可以指定特定符号不被SimProcedures替换

- 您可以查看`angr.Project._register_object`来查看具体的算法

Hooking

- angr用python摘要替换库代码的机制称为Hook，您也可以这样做！在执行模拟时，在每一步angr都会检查当前地址是否已被Hooked，如果是，则运行钩子而不是该地址的二进制代码。让你这样做的API是`proj.hook`

`(addr, hook)`，其中`hook`是SimProcedure实例。您可以使用`.is_hook`、`.unhook`和`.hooked_by`管理项目的钩子。

- 还有一个用于Hook指定地址的替代API，通过使用`proj.hook(addr)`作为函数装饰器，您可以指定自己的现成函数作为钩子使用。如果这样做，还可以选择指定length关键字参数，使执行在钩子完成后向前跳转一定数量的字节。

```
>>> stub_func = angr.SIM_PROCEDURES['stubs']['ReturnUnconstraine
>>> proj.hook(0x10000, stub_func())  # hook with an instance of

>>> proj.is_hooked(0x10000)            # these functions should
True
>>> proj.hooked_by(0x10000)
<ReturnUnconstrained>
>>> proj.unhook(0x10000)

>>> @proj.hook(0x20000, length=5)
... def my_hook(state):
...     state.regs.rax = 1

>>> proj.is_hooked(0x20000)
True
```

- 此外，您还可以使用`proj.hook_symbol(name, hook)`，提供了一个符号的名称作为第一个参数，用来钩住符号所在的地址。它的一个非常重要的用途是扩展angr的内置库SimProcedure的行为。由于这些库函数只是类，所以可以对它们进行子类化，覆盖它们的行为片段，然后在钩子中使用子类。

So far so good!

- 到目前为止，您应该对如何在CLE加载器和angr项目的级别上控制分析的环境选项有了合理的理解。您还应该了解，angr通过将复杂的库函数与总结函数效果的SimProcedure连接起来，合理地尝试简化其分析。
- 为了了解您可以使用CLE加载器及其后端装载器做的所有事情，请查看CLE API文档。

Solver Engine

- angr的强大之处不在于它是一个模拟器，而在于它能够使用我们所称的符号变量运行。与其说变量有一个*具体*的数值，不如说它包含一个*符号*。实际上这只是一个名称。然后，使用该变量执行算术操作将生成操作树(根据编译器理论，称为抽象语法树或AST)。AST可以转换为SMT求解器(如z3)的约束，以便解决诸如“给定这个操作序列的输出，求输入?”的问题。在这里，您将学习如何使用angr来回答这个问题。

Working with Bitvectors

- 让我们获取一个模拟的项目和状态，这样我们就可以开始玩具体的值了。

```
>>> import angr, monkeyhex
>>> proj = angr.Project('/bin/true')
>>> state = proj.factory.entry_state()
```

- 位向量就是一组位的序列，可以将其理解为有界整数。让我们创建几个这样的位向量

```
# 64位的有着具体值1和100的位向量
>>> one = state.solver.BVV(1, 64)
>>> one
 <BV64 0x1>
>>> one_hundred = state.solver.BVV(100, 64)
>>> one_hundred
 <BV64 0x64>

# 新建一个有着9的具体值的27位的位向量
>>> weird_nine = state.solver.BVV(9, 27)
>>> weird_nine
<BV27 0x9>
```

- 正如您所看到的，您可以创建任何位的序列，并将它们称为位向量。您也可以用它们做数学运算:

```
>>> one + one_hundred
<BV64 0x65>

# 您可以将python的常规interger用于运算，结果将被转化为合适的数据类型
>>> one_hundred + 0x100
<BV64 0x164>

# 应用普通包装算法的语义
>>> one_hundred - one*200
<BV64 0xffffffffffffff9c>
```

- 不过，你不能使用`one + weird_nine`。将不同长度的位向量一同执行运算是一种类型错误。但是，您可以扩展`weird_nine`使得它拥有合适的比特数:

```
>>> weird_nine.zero_extend(64 - 27)
<BV64 0x9>
>>> one + weird_nine.zero_extend(64 - 27)
<BV64 0xa>
```

- `zero_extend`将用给定的数值为0的位填充左边的位向量。您还可以使用`sign_extend`填充最高位，使得位向量在保留符号的情况下保持值不变。
- 现在，让我们引入一些符号混合。

```
# 创建一个名为"x"的64位的符号向量
>>> x = state.solver.BVS("x", 64)
>>> x
<BV64 x_9_64>
>>> y = state.solver.BVS("y", 64)
>>> y
<BV64 y_10_64>
```

- `x`和`y`现在是符号变量，有点像你们在7年级代数里学过的变量。注意，您提供的名称已经被附加的递增计数器所破坏。您可以使用它们执行任意数量的算术操作，但是您不会从结果中得到一个数字，而是得到一个AST。

```
>>> x + one
<BV64 x_9_64 + 0x1>
```

```
>>> (x + one) / 2
<BV64 (x_9_64 + 0x1) / 0x2>

>>> x - y
<BV64 x_9_64 - y_10_64>
```

- 每个AST都有一个`.op`和一个`.args`的属性。op是正在执行的操作的字符串的名字，args是该操作输入的值。除非op是`BVV`或`BVS`(或其他一些...)，否则arg都是AST，AST树最终以BVV或BVS结束。

```
>>> tree = (x + 1) / (y + 2)
>>> tree
<BV64 (x_9_64 + 0x1) / (y_10_64 + 0x2)>
>>> tree.op
'__floordiv__'
>>> tree.args
(<BV64 x_9_64 + 0x1>, <BV64 y_10_64 + 0x2>)
>>> tree.args[0].op
'__add__'
>>> tree.args[0].args
(<BV64 x_9_64>, <BV64 0x1>)
>>> tree.args[0].args[1].op
'BVV'
>>> tree.args[0].args[1].args
(1, 64)
```

- 从这里开始，我们将使用“位向量”这个词来指代其最顶层操作生成位向量的任何AST。还可以通过AST表示其他数据类型，包括浮点数和我们即将看到的布尔值。

Symbolic Constraints

- 在任意两个类型相似的AST之间执行比较操作将生成另一个AST—-不是位向量，而是符号布尔值。

```
>>> x == 1
<Bool x_9_64 == 0x1>
>>> x == one
<Bool x_9_64 == 0x1>
>>> x > 2
<Bool x_9_64 > 0x2>
```

```
>>> x + y == one_hundred + 5
<Bool (x_9_64 + y_10_64) == 0x69>
>>> one_hundred > 5
<Bool True>
>>> one_hundred > -5
<Bool False>
```

- 从这里可以看出，比较结果在默认情况下是无符号的。最后一个例子中的-5被强制类型转换为`<BV64 0xfffffffffffffb>`，这个值一定不小于100。如果希望对比较结果有符号化，可以使用`one_hundred.SGT(-5)`(即“有符号大于”)。完整的操作列表可以在本章末尾找到。

- 这段代码还说明了使用angr的一个重要问题——永远不要在if- or - while语句的条件下直接使用变量之间的比较，因为答案可能没有一个具体的真值。即使存在一个具体的真值，例如`if one > one_hundred`，也将引发异常。作为替代，您应该使用`solver.is_true`和`solver.is_false`，它在不执行约束解的情况下测试具体的真假。

```
>>> yes = one == 1
>>> no = one == 2
>>> maybe = x == y
>>> state.solver.is_true(yes)
True
>>> state.solver.is_false(yes)
False
>>> state.solver.is_true(no)
False
>>> state.solver.is_false(no)
True
>>> state.solver.is_true(maybe)
False
>>> state.solver.is_false(maybe)
False
```

Constraints Solving

- 通过将符号布尔值添加为状态的约束，您可以将任何符号布尔值视为关于符号变量有效值的断言（assertions ）。然后，您可以通过对符号表达式求值来查询符号变量的有效值。

- 一个例子可能比这里讲解的更清楚:

```
>>> state.solver.add(x > y)
>>> state.solver.add(y > 2)
>>> state.solver.add(10 > x)
>>> state.solver.eval(x)
4
```

- 通过向状态添加这些约束，我们迫使约束求解器将它们视为返回值必须满足的断言。如果运行这段代码，可能会得到x的不同值，但是这个值肯定大于3(因为y必须大于2,x必须大于y)，并且小于10。此外，如果您输入`state.solver.eval(y)`，您将得到一个与您得到的x值一致的y值。如果您不在两个查询之间添加任何约束，那么结果将彼此一致。
- 从这里，很容易看到如何完成我们在本章开始时提出的任务——查找产生给定输出的输入。

```
# 获得一个没有约束的新状态
>>> state = proj.factory.entry_state()
>>> input = state.solver.BVS('input', 64)
>>> operation = (((input + 4) * 3) >> 1) + input
>>> output = 200
>>> state.solver.add(operation == output)
>>> state.solver.eval(input)
0x3333333333333381
```

- 注意，同样，这个解决方案只适用于位向量语义。如果我们在整数域上操作，就没有解!
- 如果我们添加冲突或矛盾的约束，这样就没有能够使约束得到满足的值，状态就变得不可满足，对结果的查询将引发异常。您可以通过`state.satisfiable()`来检查状态的可满足性。

```
>>> state.solver.add(input < 2**32)
>>> state.satisfiable()
False
```

- 您还可以计算更复杂的表达式，而不仅仅是单个变量。

```
# 新的状态
>>> state = proj.factory.entry_state()
>>> state.solver.add(x - y >= 4)
>>> state.solver.add(y > 0)
```

```
>>> state.solver.eval(x)
5
>>> state.solver.eval(y)
1
>>> state.solver.eval(x + y)
6
```

- 从这里我们可以看出，`eval`是一种通用的方法，它可以将任何位向量转换成python的类型，同时又保证了状态的完整性。这也是为什么我们使用`eval`将具体的位向量转换为python int的原因!
- 同时还请注意，尽管我们使用旧状态创建了x和y变量，但x，y仍然可以在这个新状态中使用。变量不受任何一种状态的约束，可以自由存在。

Floating point numbers

- z3支持IEEE754浮点数的理论，因此angr也可以使用它们。主要的区别是，浮点数有一个`sort`属性，而不是`width`。您可以使用`FPV`和`FPS`创建浮点符号和值。

```
# 新的状态
>>> state = proj.factory.entry_state()
>>> a = state.solver.FPV(3.2, state.solver.fp.FSORT_DOUBLE)
>>> a
<FP64 FPV(3.2, DOUBLE)>

>>> b = state.solver.FPS('b', state.solver.fp.FSORT_DOUBLE)
>>> b
<FP64 FPS('FP_b_0_64', DOUBLE)>

>>> a + b
<FP64 fpAdd('RNE', FPV(3.2, DOUBLE), FPS('FP_b_0_64', DOUBLE))>

>>> a + 4.4
<FP64 FPV(7.6000000000000005, DOUBLE)>

>>> b + 2 < 0
<Bool fpLT(fpAdd('RNE', FPS('FP_b_0_64', DOUBLE), FPV(2.0, DOUBL
```

- 因此，这里有一些需要解释的地方——对于初学者来说，漂亮的打印对于浮点数不是一个好主意。但除此之外，大多数操作实际上都有第三个参数，这个参数在二进制操作符的使用中隐式添加——舍入模式。IEEE754规范支持多种舍入模式(`round-to-nearest`、`round-to-zero`

、`round-to-positive`等)，因此z3必须支持这些模式。如果要为操作指定舍入模式，请显式使用fp操作（例如，`solver.fpAdd`使用舍入模式）并调用舍入模式（`solver.fp.RM_*`中的一个）作为第一个参数。(其中一个solver.fp.RM_*)

- 约束和求解方法是相同的，但是`eval`返回一个浮点数:

```
>>> state.solver.add(b + 2 < 0)
>>> state.solver.add(b + 2 > -1)
>>> state.solver.eval(b)
-2.4999999999999996
```

- 这很好，但有时我们需要能够直接将浮点数表示为位向量。可以用`raw_to_bv`和`raw_to_fp`方法将位向量解释为浮点数，反之亦然:

```
>>> a.raw_to_bv()
<BV64 0x400999999999999a>
>>> b.raw_to_bv()
<BV64 fpToIEEEBV(FPS('FP_b_0_64', DOUBLE))>

>>> state.solver.BVV(0, 64).raw_to_fp()
<FP64 FPV(0.0, DOUBLE)>
>>> state.solver.BVS('x', 64).raw_to_fp()
<FP64 fpToFP(x_1_64, DOUBLE)>
```

- 这些转换保留位模式，就像将浮点指针转换为int指针一样，反之亦然。但是，如果希望尽可能地保留值，就像将浮点数转换为int(反之亦然)一样，可以使用另一组方法val_to_fp和val_to_bv。由于浮点数的特性，这些方法必须将目标值的大小或排序作为参数。

```
>>> a
<FP64 FPV(3.2, DOUBLE)>
>>> a.val_to_bv(12)
<BV12 0x3>
>>> a.val_to_bv(12).val_to_fp(state.solver.fp.FSORT_FLOAT)
<FP32 FPV(3.0, FLOAT)>
```

- 这些方法还可以接受带符号的参数，指定源或目标位向量的符号性。

More Solving Methods

- `eval` 将为表达式提供一个可行的解决方案，但是如果您想要多个呢?如果您想确保解决方案是惟一的怎么办?solver为您提供了几种常见的求解的方法:
- `solver.eval(expression)`：给出给定表达式的一个解
- `solver.eval_one(expression)`：给出给定表达式的解，如果可能有多个解，则抛出错误。
- `solver.eval_upto(expression, n)`：给出给定表达式的至多n个解，如果可能小于n，返回的解个数将小于n。
- `solver.eval_atleast(expression, n)`：给出给定表达式的n个解，如果可能解个数小于n，则抛出一个错误。
- `solver.eval_exact(expression, n)`：给出给定表达式的n个解，如果可能解的个数小于或大于n，则抛出一个错误。
- `solver.min(expression)`：给出给定表达式的最小可能解。
- `solver.max(expression)`：将给出给定表达式的最大可能解。
- 此外，所有这些方法都可以采用以下关键字参数:
- `extra_constraints`：可以作为约束的元组传递。这些限制将被考虑到这个评估，但不会添加到状态。
- `cast_to`：可以指定结果转换的数据类型。目前，这只能是`int`和`bytes`，这将导致方法返回底层数据的对应表示。例如，`state.solver.eval(state.solver.BVV(0x41424344, 32), cast_to=bytes)`，返回结果：`b'ABCD'`

Summary

- 真是太多了!!阅读本文之后，您应该能够创建并操作位向量、布尔值和浮点值，以形成操作树，然后查询附加状态的约束求解器，以在一组约束下寻找可能的解决方案。希望至此您已经理解了使用AST表示计算的能力，以及约束求解器的能力。

- 在附录中，您可以找到适用于AST的所有附加操作的引用，以防您需要一个快速表来查看。

Program State

- 到目前为止，我们只以最简单的方式使用了angr的模拟程序状态(`SimState` 对象)，以演示angr操作的基本概念。在这里，您将了解状态对象的结构，以及如何以各种有用的方式与之交互。

Basic Execution

- 前面，我们展示了如何使用模拟管理器执行一些基本的执行。在下一章中，我们将展示模拟管理器的全部功能，但是现在我们可以使用一个简单得多的接口来演示符号执行的工作原理: `state.step()` 。这个方法将执行符号执行的一个步骤，并返回一个名为 `SimSuccessors` 的对象。与普通模拟不同，符号执行可以生成多个继承状态，这些状态可以用多种方式进行分类。现在，我们关心的是这个对象的 `.successors` 属性，它是一个包含给定步骤的所有“正常”继承者的列表。

- 为什么是一个列表，而不是一个单一的继承状态?angr的符号执行过程就是执行编译到程序中的各个指令，并对SimState进行修改。当到达像 `if (x > 4)` 这样的一行代码时，如果x是一个符号位向量会发生什么?在angr的某个深度，将执行比较 `x > 4` ，结果将是 `<Bool x_32_1 > 4>` 。

- 这很好，但是下一个问题是，我们是选择“真”分支还是“假”分支?答案是，我们两者都要!我们生成两个完全独立的继承状态——一个模拟条件为真和条件为假的情况。在第一种状态中，我们添加 `x > 4` 作为约束，在第二种状态中，我们添加 `!(x > 4)` 作为约束。这样，每当我们使用这些继承状态中的任何一个执行约束解时，状态上的条件都确保我们得到的任何解都是有效的输入，这些输入将导致执行遵循给定状态所遵循的相同路径。

- 为了演示这一点，让我们以一个假固件映像为例。如果您查看这个二进制文件的源代码，您会发现固件的身份验证机制是颠倒的;任何用户名都可以通过密码“SOSNEAKY”作为管理员进行身份验证。此外，与用户输入的第一个比较是与后门的比较，因此，如果我们执行步骤，获得多个

继承状态，其中一个状态将包含限制用户输入为后门密码的条件。下面的代码片段实现了这一点:

```
>>> proj = angr.Project('examples/fauxware/fauxware')
>>> state = proj.factory.entry_state(stdin=angr.SimFile)  # igno
>>> while True:
...     succ = state.step()
...     if len(succ.successors) == 2:
...         break
...     state = succ.successors[0]

>>> state1, state2 = succ.successors
>>> state1
<SimState @ 0x400629>
>>> state2
<SimState @ 0x400699
```

- 不要直接查看这些状态上的约束—我们刚刚讨论的分支涉及`strcmp`的结果，这是一个很难用符号模拟的函数，并且产生的约束非常复杂。
- 我们模拟的程序从标准输入中获取数据，默认情况下，angr将标准输入视为无穷无尽的符号数据流。要执行一个约束求解并获得一个可以用来满足约束的输入可能值，我们需要获得对stdin实际内容的引用。稍后，我们将详细讨论文件和输入子系统的工作方式，但是现在，只使用`state.posix.stdin。load(0, state.posix.stdin.size)`来检索一个位向量，该位向量表示到目前为止从stdin读取的所有内容。

```
>>> input_data = state1.posix.stdin.load(0, state.posix.stdin.si

>>> state1.solver.eval(input_data, cast_to=bytes)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00SOSNEAKY\x00\x00\x00'

>>> state2.solver.eval(input_data, cast_to=bytes)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00S\x00\x80N\x00\x00 \x00\x0
```

- 正如您所看到的，为了沿着`state1`路径走下去，您必须将后门字符串“SOSNEAKY”作为密码。为了沿着`state2`的道路走下去，你必须给一些除了“SOSNEAKY”以外的东西。z3提供了符合这个标准的数十亿字符串中的一个。

- Fauxware是angr在2013年成功开发的第一个符号执行程序。通过使用angr找到它的后门，您将参与一个宏大的传说，即对如何使用符号执行从二进制文件中提取信息有一个基本的了解!

State Presets

- 到目前为止，无论何时处理状态，我们都使用`project.factory.entry_state()`创建它。这只是项目工厂中可用的几个状态构造函数之一:
- `.blank_state()`构造一个“空白石板”的空白状态，其中大部分数据未初始化。当访问未初始化的数据时，将返回一个不受约束的符号值。
- `.entry_state()`构造一个主二进制文件的入口点的状态。
- `.full_init_state()` 构造一个准备完成的状态，该状态需要通过在主二进制文件入口点之前运行的初始化器执行，例如，共享库构造器或预初始化器。当它完成这些，它将跳转到入口点。
- `.call_state()` 构造准备执行给定函数的状态。
- 您可以通过以下构造函数的几个参数自定义状态:
    - 所有这些构造函数都可以使用`addr`参数来指定要开始的确切地址。
    - 如果执行的环境可以接受命令行参数或环境，则可以通过`args`将参数列表传递给`entry_state`和`full_init_state`，并通过`env`将环境变量字典传递给`entry_state`和`full_init_state`。这些结构中的值可以是字符串或位向量，并将序列化为状态，作为模拟执行的参数和环境。默认的`args`是一个空列表，所以如果您正在分析的程序希望至少找到一个`argv[0]`，您应该始终提供它!
    - 如果希望`argc`是符号的，可以将符号位向量`argc`传递给`entry_state`和`full_init_state`构造函数。但是要小心:如果这样做，还应该为结果状态添加一个约束，即argc的值不能大于传递给`args`的arg的数量。
    - 要使用调用状态，应该使用`.call_state(addr, arg1, arg2, ...)`调用它，其中`addr`是要调用的函数的地址，`argN`是该函数的第n个参数，可以是python整数、字符串、数组或位向量。如果您希望

分配内存并实际传递一个指向对象的指针，您应该将它封装在一个`PointerWrapper`中，即`angr.PointerWrapper("point to me !")`。这个API的结果可能有点不可预测，但是我们正在努力。

- 在这些构造函数中可以使用更多的选项!请参阅`project.factory`的文档(一个`AngrObjectFactory`)获取更多细节

Low level interface for memory

- `state.mem`接口可以方便地从内存中加载类型化数据，但是当您希望在某个内存范围内执行原始数据加载和存储时，`state.mem`就显得过于笨重。`state.mem`实际上是一组正确访问底层内存存储的逻辑。而底层内存只是一个平面地址空间，其中充满了位向量数据:`state.memory`。您可以直接通过`.load(addr, size)`和`.store(addr, val)`方法访问`state.memory`：

```
>>> s = proj.factory.blank_state()
>>> s.memory.store(0x4000, s.solver.BVV(0x0123456789abcdef012345
>>> s.memory.load(0x4004, 6) # load-size is in bytes
<BV48 0x89abcdef0123>
```

- 正如您所看到的，由于`state.memory`的主要用途是加载没有附加语义的存储段数据，数据是以大端字节序方式加载和存储的。但是，如果希望对加载或存储的数据执行字节交换，可以传递关键字参数endness——如果指定小端字节序，则会发生字节交换。endness应该是`archinfo`包中`Endness`的成员之一，该包用于保存关于angr CPU架构的声明性数据。此外，所分析程序的结束度可以用`arch.memory_endness`表示——例如`state.arch.memory_endness`。

```
>>> import archinfo
>>> s.memory.load(0x4000, 4, endness=archinfo.Endness.LE)
<BV32 0x67453201>
```

- 还有一个用于寄存器访问的底层接口`state.registers`，它使用与`state.memory`完全相同的API。但是，解释它的行为需要深入到angr用于无缝地与多个体系结构协作的抽象中。简而言之，它只是一个寄存器文件，在archinfo中定义了寄存器和偏移量之间的映射。

State Options

- 我们可以对angr的内部进行许多小小的调整，从而优化某些情况下的行为，但也会对其他情况造成损害。这些调整是通过状态选项控制的。
- 在每个SimState对象上，都有一组(`state.options`)所有启用的选项。每个选项(实际上只是一个字符串)都以某种细微的方式控制angr执行引擎的行为。附录中列出了完整的选项域，以及不同状态类型的默认值。您可以通过 `anger.options` 访问用于添加状态的单个选项。单独的选项使用大写字母命名，但是也有一些常见的对象分组，您可能希望将它们捆绑在一起使用，使用小写字母命名。
- 当通过任何构造函数创建SimState时，可以传递关键字参数 `add_options` 和 `remove_options` ，这两个参数应该是修改默认值的初始选项集的选项集。

```
# 示例:启用lazy solver选项，可以尽可能不频繁地检查状态可满足性。
# 对该设置的更改将影响到从该状态创建的后所有继承状态。
>>> s.options.add(angr.options.LAZY_SOLVES)

# 创建一个开启lazy solver选项的状态
>>> s = proj.factory.entry_state(add_options={angr.options.LAZY_

# 创建一个不开启简化选项的状态
>>> s = proj.factory.entry_state(remove_options=angr.options.sim
```

State Plugins

- 除了刚才讨论的一组选项之外，SimState中存储的所有内容实际上都存储在附加到该状态的插件中。到目前为止，我们讨论的状态的几乎所有属性都是插件— `memory` 、 `registers` 、 `mem` 、 `regs` 、 `solvers` 等等。这种设计允许代码模块化以及为模拟状态的其他方面轻松实现新类型的数据存储的能力，或者提供插件的替代实现的能力。
- 例如，普通 `memory` 插件模拟平面内存空间，但是分析可以选择启用“抽象内存”插件，它使用替代数据类型来模拟独立于地址的自由浮动内存映射，从而提供 `state.memory` 。相反，插件可以降低代码复杂度: `state.`

`memory` 和 `state.register` 实际上是同一个插件的两个不同实例，因为寄存器也是用地址空间模拟的。

The globals plugin

- `state.globals` 是一个非常简单的插件:它实现了标准python dict的接口，允许您在状态上存储任意数据。

The history plugin

- `state.history` 是一个非常重要的插件，它存储关于一个状态在执行过程中所采取的路径的历史数据。它实际上是一个由几个历史节点组成的链表，每个节点代表一轮执行——您可以使用 `state.history.parent.parent` 遍历这个列表。
- 为了更方便地使用这个结构，history还提供了几个针对特定值的历史的有效迭代器。通常，这些值存储为 `history.recent_NAME` ，它们上面的迭代器就是 `history.NAME` 。例如， `for addr in state.history.bbl_addrs: print hex(addr)` 将为二进制文件打印一个基本的块地址跟踪，而 `state.history.recent_bbl_addrs` 是在最近的步骤中执行的基本块的列表， `state.history.parent.recent_bbl_addrs` 是在前面的步骤中执行的基本块的列表，等等。如果需要快速获得这些值的平面列表，可以访问 `.hardcopy` ，例如 `state.history.bbl_addr.hardcopy` 。但是请记住，基于索引的访问是在interators上实现的。
- 以下是历史中存储的一些值的简短列表:
- `.callstack.func_addr` 是在状态上执行的每轮执行的字符串描述的列表。
- `.callstack.call_site_addr` 是调用当前函数的基本块的地址
- `.callstack.stack_ptr` 是从当前函数开始的堆栈指针的值
- `.callstack.ret_addr` 是当前函数返回时将返回的位置

More about I/O:Files, file systems, and network sockets

- 有关如何在angr中建模I/O的更完整和更详细的文档，请参考使用文件系统、套接字和管道。

Copying and Merging

- 状态支持非常快的复制，所以你可以探索不同的可能性:

```
>>> proj = angr.Project('/bin/true')
>>> s = proj.factory.blank_state()
>>> s1 = s.copy()
>>> s2 = s.copy()

>>> s1.mem[0x1000].uint32_t = 0x41414141
>>> s2.mem[0x1000].uint32_t = 0x42424242
```

Simulation Managers

- angr中最重要的控制接口是SimulationManager，它允许您同时控制状态组的符号执行，并应用搜索策略来探索程序的状态空间。在这里，您将学习如何使用它。

- 仿真管理器允许您以一种灵活的方式选取多个状态。状态被组织成“stashes”，您可以向前运行、筛选、合并和移动。例如，这允许您以不同的速率运行两个不同的状态，然后将它们合并在一起。大多数操作的默认存储是 `active` 存储，当您初始化一个新的模拟管理器时，您的状态将放在活动存储中。

Step

- 仿真管理器最基本的功能是将给定存储中的所有状态向前推进一个基本块。使用 `.step()` 执行此操作。

```
>>> import angr
>>> proj = angr.Project('examples/fauxware/fauxware', auto_load_
>>> state = proj.factory.entry_state()
>>> simgr = proj.factory.simgr(state)
>>> simgr.active
[<SimState @ 0x400580>]

>>> simgr.step()
```

```
>>> simgr.active
[<SimState @ 0x400540>]
```

- 当然，隐藏模型的真正强大之处在于，当一个状态遇到符号分支条件时，两个继承状态都会出现在隐藏中，您可以同步执行这两个状态。当您不需要非常小心地控制分析，而只想逐步执行到没有其他步骤可以执行时，您可以使用`.run()`方法。

```
# 执行到第一个符号分支处
>>> while len(simgr.active) == 1:
...     simgr.step()

>>> simgr
<SimulationManager with 2 active>
>>> simgr.active
[<SimState @ 0x400692>, <SimState @ 0x400699>]

# 执行到所有状态结束
>>> simgr.run()
>>> simgr
<SimulationManager with 3 deadended>
```

- 我们现在有3个结束状态!例如，当一个状态在执行过程中无法产生任何后继，因为它到达了一个`exit`的syscall时，它将从活动存储中删除，并放入`deadended`存储中。

Stash Management

- 让我们看看如何使用其他隐藏。
- 要在stash之间移动状态，可以使用`.move()`，它接受`from_stash`、`to_stash`和`filter_func`(可选，默认情况下是移动所有内容)。例如，让我们移动输出中有特定字符串的所有东西:

```
>>> simgr.move(from_stash='deadended', to_stash='authenticated',
>>> simgr
<SimulationManager with 2 authenticated, 1 deadended>
```

- 只需请求将状态移动到它，我们就能够创建一个名为“authenticated”的新存储。这个隐藏中的所有状态的标准输出中都有“Welcome”，这是目前的一个很好的度量标准。

- 每个隐藏都只是一个列表，您可以在列表中建立索引或迭代，以访问每个状态，但是也有一些其他方法可以访问这些状态。如果您在一个stash的名称前面加上一个 `one_` ，您将得到该 `stash` 中的第一个状态。如果您在stash的名称前加上 `mp_` ，您将得到该stash的多路复用版本。

```
>>> for s in simgr.deadended + simgr.authenticated:
...     print(hex(s.addr))
0x1000030
0x1000078
0x1000078

>>> simgr.one_deadended
<SimState @ 0x1000030>
>>> simgr.mp_authenticated
MP([<SimState @ 0x1000078>, <SimState @ 0x1000078>])
>>> simgr.mp_authenticated.posix.dumps(0)
MP(['\x00\x00\x00\x00\x00\x00\x00\x00\x00SOSNEAKY\x00',
    '\x00\x00\x00\x00\x00\x00\x00\x00\x00S\x80\x80\x80\x80@\x80@
```

- 当然， `step` 、 `run` 和任何其他操作单个路径隐藏的方法都可以使用一个 `stash` 参数，指定要操作哪个隐藏。
- 模拟管理器为您管理您的状态提供了很多工具与服务。我们现在不会介绍其中的其余部分,但您可以检查API文档。

Stash types

- 你可以用stashes来做你喜欢的东西,但有一些stashes会被用来对一些特殊的状态进行分类。这些是:

Stash	Description
active	此隐藏包含默认情况下将逐步执行的状态，除非指定了替代隐藏。
deadended	当一个状态由于某种原因不能继续执行时，它将进入死区存储区。这些原因包括没有更多有效的指令、它的所有后继的状态无法满足约束，或者指令指针指向一个非法地址。

Stash	Description
pruned	在使用 `lazy_solutions` 时，除非绝对必要，否则不会检查状态的可满足性。当在 `lazy_solutions` 存在的情况下发现一个状态无法满足约束条件时，将遍历该状态层次结构，以确定它最初在其历史中何时不满足约束条件。所有继承该点的状态(这些点也将是不满足约束条件)都被修剪并放入这个存储中。
unconstrained	如果将 `save_unrestricted` 选项提供给SimulationManager构造函数，则确定为unrestricted的状态(即，指令指针由用户数据或其他符号数据源控制)放在这里。
unsat	如果将 `save_unsat` 选项提供给SimulationManager构造函数，则确定为不可满足的状态(即，它们有相互矛盾的约束，比如输入必须同时为“AAAA”和“BBBB”)。

- 还有另一份不属于隐藏的状态的列表: `errored` 。如果在执行过程中引发错误，则状态将被包装在 `ErrorRecord` 对象中，该对象包含状态及其引发的错误，然后将记录插入 `errored` 。您可以通过 `record.state` 获得在执行开始时导致错误的状态。同时您也可以使用 `state.error` 查看发生的错误。此外您还可以使用 `record.debug()` 在错误的位置启动调试shell。这是一个非常宝贵的调试工具!

Simple exploration

- 符号执行中一个极其常见的操作是找到到达某个地址的状态，同时在这些状态中丢弃经过另一个地址的所有状态。仿真管理器为这个模式提供了一个快捷方式， `.explore()` 方法。

- 当通过 `find` 参数启动 `.explore()` 时，将运行直到找到至少一个状态，这些状态匹配设置好的条件。这些条件可以是经过一条指令的地址，可以是经过一个地址列表，或一个函数需要一个返回一个特定状态。当活动隐藏中的任何状态匹配 `find` 的条件时，这些状态将被放置在 `found` 的stash中，执行将终止。然后，您可以探索发现的状态，或者决定丢弃它，继续使用其他状态。您还可以使用与 `find` 相同的格式指定一个 `avoid` 条件。当一个状态匹配了 `avoid` 条件，它就会被放入 `avoided` 的stash存储中，然后继续执行。最后， `num_find` 参数控制返回前应该找

到的状态数，默认值为1。当然，如果您在找到这么多解决方案之前耗尽了活动存储中的状态，那么无论如何执行都会停止。

- 让我们来看一个简单的crackme例子:
- 首先我们加载二进制文件

```
>>> proj = angr.Project('examples/CSCI-4968-MBE/challenges/crack
```

- 接下来，我们创建一个仿真模拟器。

```
>>> simgr = proj.factory.simgr()
```

- 现在，我们进行符号执行，直到找到与我们的条件匹配的状态(即“win”的条件)。

```
>>> simgr.explore(find=lambda s: b"Congrats" in s.posix.dumps(1)
<SimulationManager with 1 active, 1 found>
```

- 现在，我们可以把flag从那个状态中拿出来了!

```
>>> s = simgr.found[0]
>>> print(s.posix.dumps(1))
Enter password: Congrats!

>>> flag = s.posix.dumps(0)
>>> print(flag)
g00dJ0B!
```

- 很简单,不是吗?
- 其他示例可以通过浏览示例找到。

Exploration Techniques

- angr附带了一些固定的功能，可以定制模拟管理器的行为，称为探索技术。为什么需要探索技术？典型例子是修改探索程序状态空间的模式——默认的“一次完成所有事情”策略实际上是广度优先搜索，但是使用探索技术可以实现，例如深度优先搜索。然而，这些技术的检测功能要

灵活得多——您可以完全改变angr的步进过程的行为。编写您自己的探索技术将在后面的章节中介绍。

- 要使用探索技术，请调用`simgr.use_technology(tech)`，其中`tech`是`ExplorationTechnique`子类的一个实例。angr内置的探测技术可以在`angr.exploration_techniques`中找到。
- 下面是一些内置功能的快速概述:
- `DFS`:如前所述，深度优先搜索。一次只保持一个状态为活动状态，将其余状态保存在`deferred`中，直到它死区或错误。
- `Explorer`:此技术实现`.explore()`功能，允许您搜索和避免地址。
- `LengthLimiter`:设置状态经过的路径的最大长度上限。
- `LoopSeer`:使用合理的循环计数近似值来丢弃循环次数过多的状态，将它们放入`spinning`中，如果没有其他可行的状态，则再次将它们取出。
- `ManualMergepoint`:将程序中的一个地址标记为合并点，因此到达该地址的状态将被暂时保存，而在超时内到达同一点的任何其他状态将被合并在一起。
- `MemoryWatcher`:在simgr步骤之间监视系统上空闲/可用的内存，如果内存太低，则停止探索。
- `Oppologist`:“操作辩护者”是一个特别有趣的小工具——如果启用了这项技术并且angr遇到不支持的指令，例如bizzare和外部浮点SIMD指令，它将具体化该指令的所有输入，并使用unicorn引擎模拟单个指令，从而允许继续执行。
- `Spiller`:当有太多状态处于活动状态时，此技术可以将其中一些状态转储到磁盘，以保持低内存消耗。
- `Threading`:为步进过程添加线程级并行性。因为python的全局解释器锁，这没有多大帮助。但如果您有一个程序，它的分析花费了大量时间在angr的本地代码依赖项(unicorn、z3、libvex)中，您可以看到一些好处。
- `Tracer`:一种探测技术，它使执行遵循从其他源记录的动态跟踪。动态跟踪存储库有一些工具来生成这些跟踪。

- `Veritesting`:一篇关于自动识别有用的合并点CMU论文的实现。这非常有用，您可以在SimulationManager构造函数中使用`verititing=True`自动启用它！注意，由于它实现静态符号执行的侵入性方式，它经常不能很好地与其他技术配合使用。
- 有关更多信息，请参阅模拟管理器的API文档和探索技术。

Execution Engines

- 当您要求在angr中执行某个步骤时，必须实际执行该步骤。angr使用一系列引擎(`SimEngine`类的子类)来模拟给定代码段对输入状态的影响。angr的执行核心只是按顺序尝试所有可用的引擎，使用第一个能够处理该步骤的引擎。以下是默认的引擎列表，顺序如下:
- 当前面的步骤将我们带到某个不可持续的状态时，故障引擎就会启动
- 当前面的步骤以syscall结束时，syscall引擎就开始工作了
- 当钩住当前地址时，hook引擎就会启动
- 当启用了`UNICORN`状态选项并且状态中没有符号数据时，unicorn引擎就会启动
- VEX引擎作为最后的后备力量发挥作用。

SimSuccessors

- 实际上依次尝试所有引擎的代码是`project.factory.successors(state, **kwargs)`，它将其参数传递给每个引擎。这个函数是`state.step()`和`simulation_manager.step()`的核心。它返回一个我们之前曾简要讨论过的SimSuccessors对象。SimSuccessors的目的是对继承者状态执行简单的分类，并存储在各种列表属性中。它们是:

Attribute	Guard Condition	Instruction Pointer
`successors`	True(可以是符号的，但必须为True)	可以是符号的(但解必须小于256个;见`unconstrained_successors`)。
`unsat_successors`	False(可以是符号的，但必须为False)	可以是符号的
`flat_successors`	True(可以是符号的，但必须为True)	具体数值

Attribute	Guard Condition	Instruction Pointer
`unconstrained_successors`	True(可以是符号的，但必须为True)	符号的(超过256个解决方案
`all_successors`	所有情况	可以是符号

Breakpoints

- 待办事项:重写这个来修正叙述
- 与任何像样的执行引擎一样，angr支持断点。这很酷!设置方式如下:

```
>>> import angr
>>> b = angr.Project('examples/fauxware/fauxware')

# 获取我们的状态
>>> s = b.factory.entry_state()

# 添加一个断点，即将发生内存写入时将会触发断点并将控制权转交ipdb
>>> s.inspect.b('mem_write')

# 此外，我们还能使断点在内存写入之后触发
# 我们也可以使用一个回调函数来替代ipdb
>>> def debug_func(state):
...     print("State %s is about to do a memory write!")

>>> s.inspect.b('mem_write', when=angr.BP_AFTER, action=debug_fu
```

```
# 或者我们也可以使用iPython来替代ipdb的触发
>>> s.inspect.b('mem_write', when=angr.BP_AFTER, action=angr.BP_
```

- 除了内存写入之外，还有许多其他地方需要中断。这是列表。对于这些事件，可以在BP_BEFORE或BP_AFTER中断。

Event type	**Event meaning**
mem_read	内存被读取
mem_write	内存被写入
reg_read	寄存器被读取
reg_write	寄存器被写入
tmp_read	临时变量被读取
tmp_write	临时变量被写入
expr	正在创建一个表达式(即，算术运算的结果或IR中的常数)
statement	正在翻译IR语句
instruction	正在翻译一条新的(原生的)指令
irsb	一个新的基本块正在被翻译
constraints	新的约束被添加到状态中
exit	从执行中生成一个后继
symbolic_variable	正在创建一个新的符号变量
call	调用指令被触发
address_concretization	正在解析符号内存访问

- 这些事件有着不同的属性

事件类型	**属性名称**	**属**
mem_read	mem_read_address	BP BP

事件类型	属性名称	属
mem_read	mem_read_length	BP BP
mem_read	mem_read_expr	BP
mem_write	mem_write_address	BP BP
mem_write	mem_write_length	BP BP
mem_write	mem_write_expr	BP BP
reg_read	reg_read_offset	BP BP
reg_read	reg_read_length	BP BP
reg_read	reg_read_expr	BP
reg_write	reg_write_offset	BP BP
reg_write	reg_write_length	BP BP
reg_write	reg_write_expr	BP BP
tmp_read	tmp_read_num	BP BP
tmp_read	tmp_read_expr	BP
tmp_write	tmp_write_num	BP BP
tmp_write	tmp_write_expr	BP
expr	expr	BP BP
expr	expr_result	BP

事件类型	属性名称	属
statement	statement	BP BP
instruction	instruction	BP BP
irsb	address	BP BP
constraints	added_constraints	BP BP
call	function_address	BP BP
exit	exit_target	BP BP
exit	exit_guard	BP BP
exit	exit_jumpkind	BP BP
symbolic_variable	symbolic_name	BP BP
symbolic_variable	symbolic_size	BP BP
symbolic_variable	symbolic_expr	BP

事件类型	属性名称	属
address_concretization	address_concretization_strategy	BP BP
address_concretization	address_concretization_action	BP BP
address_concretization	address_concretization_memory	BP BP
address_concretization	address_concretization_expr	BP BP
address_concretization	address_concretization_add_constraints	BP BP
address_concretization	address_concretization_result	BP

- 这些属性可以作为`state.inspect`访问。在适当的断点回调期间检查，以访问适当的值。您甚至可以修改这些值来进一步修改这些值的用法

```
>>> def track_reads(state):
...     print('Read', state.inspect.mem_read_expr, 'from', state
...
>>> s.inspect.b('mem_read', when=angr.BP_AFTER, action=track_rea
```

- 此外，这些属性中的每一个都可以用作要检查的关键字`inspect.b`给断点添加条件:

```
# 如果在0x1000的地址处的内存被写入，那么在内存写入之前就会中断
>>> s.inspect.b('mem_write', mem_write_address=0x1000)

# 如果0x1000是它的目标表达式的唯一值，那么在内存写之前就会中断expression
>>> s.inspect.b('mem_write', mem_write_address=0x1000, mem_write

# 这将在指令0x8000之后中断，但是只有0x1000可能是从内存中读取的最后一个表达式
>>> s.inspect.b('instruction', when=angr.BP_AFTER, instruction=0
```

- 酷炫的东西!事实上，我们甚至可以将函数指定为一个条件:

```
# 这是一个复杂的情况，可以做任何事情!在本例中，它确保了RAX是0x41414141，并且
# 从0x8004开始的基本块在此路径历史上的某个时候执行

>>> def cond(state):
...     return state.eval(state.regs.rax, cast_to=str) == 'AAAA'

>>> s.inspect.b('mem_write', condition=cond)
```

Caution about `mem_read` breakpoint

- `mem_read` 断点在执行程序或二进制分析执行内存读取时被触发。如果您在 `mem_read` 上使用断点，同时也使用 `state.mem` 从内存地址加载数据，然后知道技术上来讲断点将在读取内存时触发。
- 因此，如果希望从内存加载数据而不触发已经设置的 `mem_read` 断点，那么使用 `state.memory.load` 并加上关键字参数 `disable_actions=True` 和 `inspect=False` 。
- 对于 `state.find` 也是如此。您可以使用相同的关键字参数来防止触发 `mem_read` 断点。

Analyses

- angr的目标是使对二进制程序进行的分析变得容易。为此，angr允许您以一种通用格式打包分析代码，这种格式可以很容易地应用于任何项目。稍后我们将介绍如何编写您自己的分析，但是我们的想法是所有的

分析都出现在`project.analyses`中(例如`project.analyses.CFGFast()`)。它可以作为函数调用，返回分析结果实例。

Built-in Analyses

名称	描述
CFGFast	构建一个项目的*控制流程图*
CFGEmulated	构建一个项目运行时的*控制流程图*
VFG	对程序的每个函数执行VSA，创建*数值流程图*并检测堆栈变量
DDG	计算*数据依赖关系图*，从而确定给定值所依赖的语句
BackwardSlice	计算程序相对于某个目标的*后向切片*
Identifier	标识CGC二进制文件中的公共库函数
More!	angr有相当多的分析，其中大部分是有效的!如果您想知道如何使用其中的某一个，请提交一个issue来请求获取一个说明文档

Resilience

- 分析可以写得很有弹性，基本上可以捕获并记录任何错误。根据捕获的方式，这些错误将被记录到分析的`errors`或`named_errors`属性中。但是，您可能希望以“fail fast”模式运行分析，这样就不会处理错误。为此，可以将参数`fail_fast=True`传递到分析构造函数中。

Remarks

- 恭喜你!如果您已经通读了本书(编者按:这条注释只适用于我们实际上已经完成所有todo的时候)，那么您已经了解了开始进行二进制分析所需的angr的所有基本组件。

- 最终，angr只是一个模拟器。它是一个高度可测试的、非常独特的仿真器，考虑了很多环境因素，这是真的，但是作为核心，您使用angr所做的工作是提取关于一组字节码在CPU上的行为的知识。在设计angr时，我们尝试在这个仿真器上提供工具和抽象，以使某些常见任务更有用，但是没有什么问题不能通过使用SimState和观察`.step()`的影响来解决。

- 当您进一步阅读本书时，我们将描述更多的技术主题，以及如何针对复杂的场景调整angr的行为。这些知识应该告诉您如何使用angr，以便您能够以最快的方式解决任何给定的问题，但最终，您将希望通过使用您所掌握的工具来发挥创造力来解决问题。如果您可以将一个问题转换成一个定义了可处理的输入和输出的形式，那么您绝对可以使用angr来实现您的目标，因为这些目标涉及到分析二进制文件。我们提供的任何抽象或工具都不是如何为给定任务使用angr的最终目的—angr的设计使其可以按照您希望的方式进行集成或特别使用。如果你看到一条从问题到解决方案的道路，那就走吧。

- 当然，要熟悉像angr这样庞大的技术是非常困难的。为此，您完全可以依靠社区(通过angr slack是最佳选择)来讨论angr并使用它解决问题。

- 祝好运！