|  |
| --- |
| **Open Source Project** |

조선대학교 IT 융합대학 컴퓨터공학과

3학년 20124855 박사홍

3학년 20124826 최용선

3학년 20134844 박효빈

Open Source Software

강문수 교수님

목차

오픈소스 소프트웨어 선택 및 개요

소개

설치방법

기여 방법

핵심 개념

추후 추가 함.

1. **오픈소스 소프트웨어 선택 및 개요**

* **angr가 무엇이며, 어떻게 사용하는가?**

angr는 Mayhem, KLEE 등 Dynamic symbolic execution와 Static analyses를 수행할 수 있는 Multi-architecture binary 분석 툴입니다.

바이너리 분석이 복잡하기 때문에 angr도 복잡하게 만들어졌지만

문서를 통해서 쉽게 사용할 수 있도록 했습니다.

바이너리를 프로그래밍적으로 분석하기 위해서 몇 가지 문제를 해결해야 합니다.

- 분석 프로그램에 바이너리를 로드 하기.

- 바이너리를 중간 표현(Intermediate Representation (IR))로 변환하기.

- 실제 분석 수행

- 종속성 분석, 프로그램 분할과 같은 부분 또는 전체 프로그램 정적 분석.

- Overflow가 일어날 수 있을 때까지 실행할 수 있는 것처럼 프로그램의 상태공간 분석.

- 위 두 예시의 결합. (Overflow를 찾을 때까지 프로그램 실행)

angr는 이러한 문제를 해결하기 위한 요소가 있으며, 이 문서는 어떻게 작동하는지 사용자가 원하는 목표를 이룰 수 있도록 설명합니다.

### **인용**

angr는 학업에 사용하는 경우 논문을 인용해 주세요.

@article{shoshitaishvili2016state,

title={SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis},

author={Shoshitaishvili, Yan and Wang, Ruoyu and Salls, Christopher and Stephens, Nick and Polino, Mario and Dutcher, Audrey and Grosen, John and Feng, Siji and Hauser, Christophe and Kruegel, Christopher and Vigna, Giovanni},

booktitle={IEEE Symposium on Security and Privacy},

year={2016}

}

@article{stephens2016driller,

title={Driller: Augmenting Fuzzing Through Selective Symbolic Execution},

author={Stephens, Nick and Grosen, John and Salls, Christopher and Dutcher, Audrey and Wang, Ruoyu and Corbetta, Jacopo and Shoshitaishvili, Yan and Kruegel, Christopher and Vigna, Giovanni},

booktitle={NDSS},

year={2016}

}

@article{shoshitaishvili2015firmalice,

title={Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware},

author={Shoshitaishvili, Yan and Wang, Ruoyu and Hauser, Christophe and Kruegel, Christopher and Vigna, Giovanni},

booktitle={NDSS},

year={2015}

}

1. **소개**

## angr 설치

angr는 Python libary로 사용하려면 Python 환경에서 설치해야 합니다. Python2를 기반으로 만들어졌으며 Python3는 추후 지원됩니다.

angr를 사용하고 실행하려면 [Python 가상환경](https://virtualenvwrapper.readthedocs.org/en/latest/)을 사용하는 것을 추천합니다.

angr에 의존하는 z3, pyvex 는 원시코드를 요구합니다. 하지만 libz3나 libVEX가 이미 설치된 경우 덮어쓰지 않습니다.

### 종속성

Python 모듈을 사용하기 위해서는 pip나 setup.py 스크립트를 사용합니다. Python 라이브러리인 cffi를 설치해야 합니다.

우분투에서 sudo apt-get install python-dev libffi-dev build-essential virtualenvwrapper 명령어를 통해 설치합니다. angr-management를 사용하려면 sudo apt-get install libqt4-dev graphviz-dev 를 설치해야 합니다.

### 대부분의 운영체제(\*nix 시스템)

angr는 python package index에 게시되어 있기 때문에 일반적으로 mkvirtualenv angr && pip install angr 명령을 통해 설치할 수 있습니다.

Fish(shell) 사용자는 [virtualfish](https://github.com/adambrenecki/virtualfish) 또는 [virtualenv](https://pypi.python.org/pypi/virtualenv) 패키지를 사용할 수 있습니다. vf new angr && vf activate angr && pip install angr

### Mac OS X

pip install angr 명령으로 설치할 수 있지만 몇가지 주의사항이 있습니다.

만약 Clang 으로 설치가 되지 않는다면 GCC를 이용해야 합니다.

brew install gcc

env CC=/usr/local/bin/gcc-6 pip install angr

angr를 설치한 뒤에 몇가지 공유 라이브러리 경로를 수정해야 합니다.

BASEDIR=/usr/local/lib/python2.7/site-packages

# If you don't know where your site-packages folder is, use this to find them:

python2 -c "import site; print(site.getsitepackages())"

install\_name\_tool -change libunicorn.1.dylib "$BASEDIR"/unicorn/lib/libunicorn.dylib "$BASEDIR"/angr/lib/angr\_native.dylib

install\_name\_tool -change libpyvex.dylib "$BASEDIR"/pyvex/lib/libpyvex.dylib "$BASEDIR"/angr/lib/angr\_native.dylib

### Windows

angr는 Windows에서 pip를 이용해 설치할 수 있습니다. (Visutal studio 빌드 툴을 이용해서)

Windows에서 Capstone을 설치하기 어렵습니다.

requirements.txt 파일 안에서 capstone을 지우는 것이 좋습니다.

### 개발자를 위한 설치

angr 개발자를 위해서 repo와 스크립트를 만들었습니다.

아래 명령을 통해서 쉽게 설치할 수 있습니다.

git clone git@github.com:angr/angr-dev.git

cd angr-dev

mkvirtualen angr

/setup.sh

### Docker 설치

편의성을 위해서 docker 이미지를 제공합니다.

# install docker

curl -sSL https://get.docker.com/ | sudo sh

# pull the docker image

sudo docker pull angr/angr

# run it

sudo docker run -it angr/angr

### angr container 수정

apt를 통해서 추가적인 패키지를 설치해야 하는 경우 권한 상승이 필요합니다. 아래 명령어를 통해서 권한을 설정해야 합니다.

# assuming the docker container is running

# with the name "angr" and the instance is

# running in the background.

docker exec -ti -u root angr bash

### **문제 해결**

#### libgomp.so.1: version GOMP\_4.0 not found, or other z3 issues

angr-only-z3-custom 과 미리 설치된 버전 간 호환되지 않은 문제 입니다. z3의 재컴파일이 필요합니다. pip install -I --no-use-wheel z3-solver

#### capstone 때문에 angr를 import 할 수 없는 경우

종종 capstone 때문에 angr가 제대로 설치되지 않은 경우가 있습니다. capstone을 재빌드해야 합니다. pip install -I --pre --no-use-wheel capstone

만약 해결되지 않는다면 몇가지 버그 때문일 수 있습니다. virtualenv/virtualenvwrapper 환경에서 pip를 이용한 capstone\_3.0.4 설치에서 버그가 있습니다.

가상 환경에서 /home/<username>/.virtualenvs/<virtualenv>/lib/python2.7/site-packages/capstone/\*.py(c) capstone python을 설치하면,

capstone라이브러리는 /home/<username>/.virtualenvs/<virtualenv>/lib/python2.7/site-packages/home/<username>/.virtualenvs/<virtualenv>/lib/python2.7/site-packages/capstone/libcapstone.so에서 찾을 수 있습니다.

native 환경에서 /usr/local/lib/python2.7/dist-packages/capstone/\*.py(c) capstone python을 설치하면, capstone 라이브러리는 /usr/local/lib/python2.7/dist-packages/usr/lib/python2.7/dist-packages/capstone/libcapstone.so에서 찾을 수 있습니다.

libcapstone.so파일을 파이썬 파일과 같은 디렉토리로 이동하면 문제가 해결됩니다.

#### No such file or directory: 'pyvex\_c'

Ubuntu 12.04를 사용하고 있다면 업데이트를 하는 것이 좋습니다. pip install -U pip를 업그레이드 해서 해결할 수 있습니다.

#### AttributeError: 'FFI' object has no attribute 'unpack'

오래된 cffi 버전의 모듈을 사용하고 있을 수 있습니다. angr는 최소 1.7 이상의 cffi가 필요합니다. pip install --upgrade cffi 명령을 실행해보고 문제가 계속 발생한다면 cffi가 설치되어있는지 확인해보세요.

pypy와 같은 인터프리터를 사용한다면 cffi가 오래된 버전일 수 있습니다. 최신 버전의 pypy를 설치하세요.

### **지원**

angr를 사용하는데 도움을 받으려면 아래를 참고해 주세요.

* Mail: [angr@lists.cs.ucsb.edu](mailto:angr@lists.cs.ucsb.edu)
* Slack channel: [angr.slack.com](https://angr.slack.com/)
* IRC channel: #angr on [freenode](https://freenode.net/)

### **기여 방법**

### **버그 리포팅**

만약 angr가 해결하지 못하거나 버그가 나타난다면 알려주세요!

1. angr/binaries 와 angr/angr fork 만들기
2. 문제된 바이너리와 함께 angr/binaries의 pull 요청하기.
3. angr/tests/borken\_x.py, angr/tests/broken\_y.py등 테스트 케이스와 함께 angr/angr pull 요청하기.

angr에서 제공하는 테스트 케이스 형식을 따르도록 해주세요.

def test\_some\_broken\_feature():

p = angr.Project("some\_binary")

result = p.analyses.SomethingThatDoesNotWork()

assert result == "what it should \*actually\* be if it worked"

if \_\_name\_\_ == '\_\_main\_\_':

test\_some\_broken\_feature()

이러한 형식은 버그를 빨리 고치는데 도움을 줍니다.

### **angr 개발**

좋은 상태를 유지하기 위한 몇가지 가이드라인이 있습니다.

#### 코딩 스타일

[PEP8 코드 협약](http://legacy.python.org/dev/peps/pep-0008/)에 맞추고 있습니다. vim을 사용한다면 [파이썬 모드](https://github.com/python-mode/python-mode) 플러그인만 있으면 됩니다.

angr의 일부분의 코드를 작성할 경우 다음의 경우를 생각해야 합니다.

getter와 setter 대신에 @property decorator와 같은 attribute 접근을 사용해주세요. Java가 아니고 iPython이기 때문에 속성은 탭 완성을 가능하게 합니다.

우리가 제공하는 .pylintrc를 사용하세요. CI 서버에서 빌드 실패가 나타날 수 있습니다.

절대로 raise Exception이나 assert False를 하지마세요. 올바른 예외 처리를 사용해야 합니다. 올바른 예외 처리가 없다면 AngerError나 SimError를 사용하세요.

tabs 사용을 자제하고 들여쓰기를 사용하세요. 표준은 4칸이며 스페이스와 탭을 혼용해서 사용하는 것은 좋지 못합니다.

긴 줄은 코드를 읽기 불편하기 때문에 120자 내로 작성하도록 합니다.

큰 기능은 작은 기능으로 나누는 것이 좋습니다.

항상 디버깅 할 때 접근할 수 있도록 \_\_ 대신에 \_를 사용하세요.

#### 문서

코드를 문서화 하고, 모든 클래스 정의와 함수 정의에는 설명이 있어야합니다.

* 무엇을 하는지?
* 매개 변수의 타입과 의미는 무엇인지?
* 반환 값은 무엇인지?

[sphinx](http://www.sphinx-doc.org/en/stable/)를 이용하여 API 문서를 작성합니다. sphinx는 함수 매개변수, 반환 값, 형식 등을 문서화 하는 특수 키워드를 지원합니다.

아래 내용은 함수 문서의 예시 입니다. 변수 설명은 가능한 읽을 수 있도록 작성해야 합니다.

def prune(self, filter\_func=None, from\_stash=None, to\_stash=None):

"""

Prune unsatisfiable paths from a stash.

:param filter\_func: Only prune paths that match this filter.

:param from\_stash: Prune paths from this stash. (default: 'active')

:param to\_stash: Put pruned paths in this stash. (default: 'pruned')

:returns: The resulting PathGroup.

:rtype: PathGroup

"""

위와 같은 방식은 함수의 매개변수가 명확하게 인식할 수 있다는 장점이 있습니다. 문서를 반복적으로 작성할 경우 아래의 방식 처럼 작성할 수 있습니다.

def read\_bytes(self, addr, n):

"""

Read `n` bytes at address `addr` in memory and return an array of bytes.

"""

#### 단위 시험

새로운 기능을 추가하고 테스트 케이스가 없으면 해당 기능은 정상적으로 작동하지 않을 수 있기 때문에 테스트 케이스를 작성해야 합니다.

커밋에서 기능을 검사하는 CI 서버를 구동 중입니다. 서버에서 테스트를 실행하게 하려면 해당 repository의 폴더에 일치하는 파일의 [nosetests](https://nose.readthedocs.org/en/latest/) 에 혀용되는 형식으로 작성하세요.

### **도움이 필요합니다.**

angr는 거대한 프로젝트로 유지하기가 어렵습니다. 다양한 사람들이 참여할 수 있도록 TODO 항목이 나열되어 있습니다.

#### 문서

문서화가 거의 돼 있지 않습니다. 많은 사람들의 도움이 필요합니다.

#### API

누락된 사항을 파악하기 위해 github에 tracking 이슈를 만들었습니다.

1. [angr](https://github.com/angr/angr/issues/145)
2. [claripy](https://github.com/angr/claripy/issues/17)
3. [cle](https://github.com/angr/cle/issues/29)
4. [pyvex](https://github.com/angr/pyvex/issues/34)

#### GitBook

이 GitBook에는 몇 가지 핵심 부분이 있습니다.

1. TODO 작성
2. 페이지를 보기 쉽게 간단한 표 사용.

#### angr 코스

angr를 이용하여 개발을 한다면 정말 유익할 것입니다. 사람들이 점점 angr의 기능을 사용해야 할 필요가 있습니다.

#### 재 연구

아쉽게도 모든 사람들이 angr를 연구하는 것은 아닙니다. 이를 해결할 때까지 프레임워크 내에서 재사용이 가능하다록 angr를 정기적으로 관련 작업을 구현해야 합니다.

### **개발**

#### angr 관리

angr GUI인 [angr-management](https://github.com/angr/angr-management)는 많은 작업이 필요합니다. 아래 내용은 현재 angr-management에서 누락된 항목입니다.

* IDA Pro의 네비게이터 툴바처럼 프로그램의 메모리 공간에 내용을 보여주는 것.
* 프로그램의 텍스트 기반 Disassembly.
* 프로그램의 상태의 세부 정보 (레지스터, 메모리 등)
* 상호 참조

#### IDA 플러그인

angr의 많은 기능들이 IDA에 이용될 수 있습니다.

1. **핵심 개념**

angr를 시작하기 전에 몇 가지 기본 개념과 오브젝트를 구성하는 방법을 알아야 합니다.

angr를 사용하는 첫 번째 작업은 프로젝트에 바이너리를 로드하는 것입니다. /bin/true를 예로 사용할 것입니다.

>>> import angr

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

위와 같이 작성하면 /bin/true에 대한 분석 및 시뮬레이션을 할 수 있습니다.

#### 기본 속성

먼저 프로젝트에 대한 기본 속성인 아키텍쳐, 파일 이름, 시작 주소를 갖습니다.

>>> import monkeyhex # this will format numerical results in hexadecimal

>>> proj.arch

<Arch AMD64 (LE)>

>>> proj.entry

0x401670

>>> proj.filename

'/bin/true'

* arch는 archinfo.Arch 객체의 인스턴스 입니다. 위 경우 리틀 엔디안 amd64 입니다.
* entry는 바이너리의 시작 지점입니다.
* filename은 바이너리의 파일 이름입니다.

#### loader

가상 주소 공강에서 바이너리 파일을 나타내는 것은 어렵습니다. 이를 처리하기 위해서 CLE라는 모듈을 사용합니다. CLE의 결과는 .loader 속성에서 사용할 수 있습니다.

>>> proj.loader

<Loaded true, maps [0x400000:0x5004000]>

>>> proj.loader.shared\_objects # may look a little different for you!

{'ld-linux-x86-64.so.2': <ELF Object ld-2.24.so, maps [0x2000000:0x2227167]>,

'libc.so.6': <ELF Object libc-2.24.so, maps [0x1000000:0x13c699f]>}

>>> proj.loader.min\_addr

0x400000

>>> proj.loader.max\_addr

0x5004000

>>> proj.loader.main\_object # we've loaded several binaries into this project. Here's the main one!

<ELF Object true, maps [0x400000:0x60721f]>

>>> proj.loader.main\_object.execstack # sample query: does this binary have an executable stack?

False

>>> proj.loader.main\_object.pic # sample query: is this binary position-independent?

True

#### factory

대부분의 project를 인스턴스화 해야합니다. 공통 객체에 대한 project.factory 생성자를 제공합니다.

##### **block**

project.factory.block()은 입력된 주소로 기본 블록 단위로 추출합니다.

>>> block = proj.factory.block(proj.entry) # lift a block of code from the program's entry point

<Block for 0x401670, 42 bytes>

>>> block.pp() # pretty-print a disassembly to stdout

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 # how many instructions are there?

0xb

>>> block.instruction\_addrs # what are the addresses of the instructions?

[0x401670, 0x401672, 0x401675, 0x401676, 0x401679, 0x40167d, 0x40167e, 0x40167f, 0x401686, 0x40168d, 0x401694]

block를 사용해서 코드 블록의 다른 표현 방식을 출력할 수 있습니다.

>>> block.capstone # capstone disassembly

<CapstoneBlock for 0x401670>

>>> block.vex # VEX IRSB (that's a python internal address, not a program address)

<pyvex.block.IRSB at 0x7706330>

##### **states**

project 객체는 프로그램의 초기화 이미지를 나타냅니다. angr를 사용하여 실행 할 때 프로그램 상태를 나타내는 SimState가 있습니다.

>>> state = proj.factory.entry\_state()

<SimState @ 0x401670>

SimState는 프로그램의 메모리, 레지스터, 파일 시스템 데이터를 포함하고 있습니다.

>>> state.regs.rip # get the current instruction pointer

<BV64 0x401670>

>>> state.regs.rax

<BV64 0x1c>

>>> state.mem[proj.entry].int.resolved # interpret the memory at the entry point as a C int

<BV32 0x8949ed31>

python의 int가 아니며 bitvector 입니다.

>>> bv = state.solver.BVV(0x1234, 32) # create a 32-bit-wide bitvector with value 0x1234

<BV32 0x1234> # BVV stands for bitvector value

>>> state.solver.eval(bv) # convert to python int

0x1234

bitvector 를 레지스터와 메모리에 다시 저장하거나 python 정수를 직접 저장 할 수 있고 적절한 비트 벡터로 변환됩니다.

>>> 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>

mem을 사용하는 방법은 아래와 같습니다.

* array[index] 표기법을 사용하여 주소 지정
* .<type> 이 char, short, int, long 등 으로 해석되야 할 때.
* 다음 중 하나를 수행 할 수 있습니다.
  + bitvector나 python int 중 하나에 값 저장.
  + .resolved 값을 비트 벡터로 가져오기.
  + .concrete 값을 int로 가져오기.

##### 분석

angr는 프로그램에서 정보를 추출하는데 사용할 수 있는 몇 가지 제공되는 패키지가 있습니다.

>>> proj.analyses. # Press TAB here in ipython to get an autocomplete-listing of everything:

proj.analyses.BackwardSlice proj.analyses.CongruencyCheck proj.analyses.reload\_analyses

proj.analyses.BinaryOptimizer proj.analyses.DDG proj.analyses.StaticHooker

proj.analyses.BinDiff proj.analyses.DFG proj.analyses.VariableRecovery

proj.analyses.BoyScout proj.analyses.Disassembly proj.analyses.VariableRecoveryFast

proj.analyses.CDG proj.analyses.GirlScout proj.analyses.Veritesting

proj.analyses.CFG proj.analyses.Identifier proj.analyses.VFG

proj.analyses.CFGAccurate proj.analyses.LoopFinder proj.analyses.VSA\_DDG

proj.analyses.CFGFast proj.analyses.Reassembler

사용하는 방법을 찾으려면 [API문서](http://angr.io/api-doc/angr.html?highlight=cfg#module-angr.analysis)를 살펴보세요.

# Originally, when we loaded this binary it also loaded all its dependencies into the same virtual address space

# This is undesirable for most analysis.

>>> proj = angr.Project('/bin/true', auto\_load\_libs=False)

>>> cfg = proj.analyses.CFGFast()

<CFGFast Analysis Result at 0x2d85130>

# cfg.graph is a networkx DiGraph full of CFGNode instances

# You should go look up the networkx APIs to learn how to use this!

>>> cfg.graph

<networkx.classes.digraph.DiGraph at 0x2da43a0>

>>> len(cfg.graph.nodes())

951

# To get the CFGNode for a given address, use cfg.get\_any\_node

>>> entry\_node = cfg.get\_any\_node(proj.entry)

>>> len(list(cfg.graph.successors(entry\_node)))

2

### 바이너리 로딩 - CLE와 angr project

저번 문서에서는 angr의 로딩 기능을 자세히 알아보지 못했습니다. /bin/true를 로드한 다음 다시 로드할 때 공유 라이브러리 없이 했습니다. 또한 proj.loader를 사용해봤는데 인터페이스의 작은 차이들을 살펴보도록 하겠습니다.

angr의 CLE에 대해서 간략히 알아봤는데 CLE는 "CLE Loads Everything"의 약자이며 바이너리와 라이브러리를 가져와서 작업하기 쉬운 방식으로 사용됩니다.

#### loader

다시 로드하고 로드와 상호작용하는 방법을 알아보겠습니다.

>>> import angr, monkeyhex

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

>>> proj.loader

<Loaded true, maps [0x400000:0x5008000]>

##### **로드된 객체**

cle.loader는 로드된 바이너리 객체의 전체 집합을 나타내며 메모리 공간에 로드 및 매핑 됩니다. 각 바이너리 객체는 cle.Backend에 의해서 로드됩니다.

CLE가 로드한 객체의 목록을 loader.all\_objects로 확인 할 수 있습니다.

# All loaded objects

>>> proj.loader.all\_objects

[<ELF Object fauxware, maps [0x400000:0x60105f]>,

<ELF Object libc.so.6, maps [0x1000000:0x13c42bf]>,

<ELF Object ld-linux-x86-64.so.2, maps [0x2000000:0x22241c7]>,

<ELFTLSObject Object cle##tls, maps [0x3000000:0x300d010]>,

<KernelObject Object cle##kernel, maps [0x4000000:0x4008000]>,

<ExternObject Object cle##externs, maps [0x5000000:0x5008000]>

# This is the "main" object, the one that you directly specified when loading the project

>>> proj.loader.main\_object

<ELF Object true, maps [0x400000:0x60105f]>

# This is a dictionary mapping from shared object name to object

>>> proj.loader.shared\_objects

{ 'libc.so.6': <ELF Object libc.so.6, maps [0x1000000:0x13c42bf]>

'ld-linux-x86-64.so.2': <ELF Object ld-linux-x86-64.so.2, maps [0x2000000:0x22241c7]>}

# Here's all the objects that were loaded from ELF files

# If this were a windows program we'd use all\_pe\_objects!

>>> proj.loader.all\_elf\_objects

[<ELF Object true, maps [0x400000:0x60105f]>,

<ELF Object libc.so.6, maps [0x1000000:0x13c42bf]>,

<ELF Object ld-linux-x86-64.so.2, maps [0x2000000:0x22241c7]>]

# Here's the "externs object", which we use to provide addresses for unresolved imports and angr internals

>>> proj.loader.extern\_object

<ExternObject Object cle##externs, maps [0x5000000:0x5008000]>

# This object is used to provide addresses for emulated syscalls

>>> proj.loader.kernel\_object

<KernelObject Object cle##kernel, maps [0x4000000:0x4008000]>

# Finally, you can to get a reference to an object given an address in it

>>> proj.loader.find\_object\_containing(0x400000)

<ELF Object true, maps [0x400000:0x60105f]>

객체와 직접 상호작용하여 메타데이터를 추출 할 수 있습니다.

>>> obj = proj.loader.main\_object

# The entry point of the object

>>> obj.entry

0x400580

>>> obj.min\_addr, obj.max\_addr

(0x400000, 0x60105f)

# Retrieve this ELF's segments and sections

>>> obj.segments

<Regions: [<ELFSegment offset=0x0, flags=0x5, filesize=0xa74, vaddr=0x400000, memsize=0xa74>,

<ELFSegment offset=0xe28, flags=0x6, filesize=0x228, vaddr=0x600e28, memsize=0x238>]>

>>> 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 0x20>,

...etc

# You can get an individual segment or section by an address it contains:

>>> obj.find\_segment\_containing(obj.entry)

<ELFSegment offset=0x0, flags=0x5, filesize=0xa74, vaddr=0x400000, memsize=0xa74>

>>> obj.find\_section\_containing(obj.entry)

<.text | offset 0x580, vaddr 0x400580, size 0x338>

# Get the address of the PLT stub for a symbol

>>> addr = obj.plt['\_\_libc\_start\_main']

>>> addr

0x400540

>>> obj.reverse\_plt[addr]

'\_\_libc\_start\_main'

# Show the prelinked base of the object and the location it was actually mapped into memory by CLE

>>> obj.linked\_base

0x400000

>>> obj.mapped\_base

0x400000

##### **symbols과 재배치**

CLE를 이용하여 sybol을 작업할 수 있습니다. 심볼은 실행 포맷의 기본이되는 개념입니다.

CLE에서 loader.find\_symbol을 이용하여 심볼을 쉽게 얻을 수 있습니다.

>>> malloc = proj.loader.find\_symbol('malloc')

>>> malloc

<Symbol "malloc" in libc.so.6 at 0x1054400>

심볼의 주소는 3가지 방식으로 출력할 수 있습니다.

* .rebased\_addr은 전역 주소 공간을 나타냅니다.
* .linked\_addr은 바이너리에 미리 링킹된 상대적인 주소입니다.
* .relative\_addr은 객체 기준의 상대적인 주소입니다. RVA(relative virtual address)입니다.

>>> malloc.name

'malloc'

>>> malloc.owner\_obj

<ELF Object libc.so.6, maps [0x1000000:0x13c42bf]>

>>> malloc.rebased\_addr

0x1054400

>>> malloc.linked\_addr

0x54400

>>> malloc.relative\_addr

0x54400

추가적으로 디버그 정보도 제공합니다. libc는 malloc을 export하고 메인 바이너리에 의존합니다. 만약 CLE가 malloc 심볼을 메인 객체에 직접 준다면 import 입니다.

>>> malloc.is\_export

True

>>> malloc.is\_import

False

# On Loader, the method is find\_symbol because it performs a search operation to find the symbol.

# On an individual object, the method is get\_symbol because there can only be one symbol with a given name.

>>> main\_malloc = proj.loader.main\_object.get\_symbol("malloc")

>>> main\_malloc

<Symbol "malloc" in true (import)>

>>> main\_malloc.is\_export

False

>>> main\_malloc.is\_import

True

>>> main\_malloc.resolvedby

<Symbol "malloc" in libc.so.6 at 0x1054400>

export, import 관계를 메모리에 저장해야 할 경우 재배치에 의해 처리됩니다.

# Relocations don't have a good pretty-printing, so those addresses are python-internal, unrelated to our program

>>> proj.loader.shared\_objects['libc.so.6'].imports

{u'\_\_libc\_enable\_secure': <cle.backends.relocations.generic.GenericJumpslotReloc at 0x4221fb0>,

u'\_\_tls\_get\_addr': <cle.backends.relocations.generic.GenericJumpslotReloc at 0x425d150>,

u'\_dl\_argv': <cle.backends.relocations.generic.GenericJumpslotReloc at 0x4254d90>,

u'\_dl\_find\_dso\_for\_object': <cle.backends.relocations.generic.GenericJumpslotReloc at 0x425d130>,

u'\_dl\_starting\_up': <cle.backends.relocations.generic.GenericJumpslotReloc at 0x42548d0>,

u'\_rtld\_global': <cle.backends.relocations.generic.GenericJumpslotReloc at 0x4221e70>,

u'\_rtld\_global\_ro': <cle.backends.relocations.generic.GenericJumpslotReloc at 0x4254210>}

#### 로딩 옵션

angr.Project로 로딩할 때 cle.Loader에 옵션을 전달하려면 Project로 전달하면 CLE로 전달됩니다

##### 기본 옵션

CLE는 자동적으로 공유 라이브러리의 의존을 활성화하거나 비활성화를 할 수 있습니다. 추가적으로 except\_missing\_libs가 true로 설정 돼 있으면 공유 라이브러리에 종속적인 바이너리가 있을 때 마다 예외가 발생합니다.

공유 라이브러이 종속성에 의존이 해결되지 않을 때 force\_load\_libs에 문자열로 전달되고 처리할 수 있습니다. 또한 skip\_libs에 문자열로 전달할 경우 의존성을 해결할 수 있습니다. 추가적으로, custom\_ld\_path에 전달하는 경우 공유 라이브러리를 위한 경로를 찾습니다.

##### **바이너리 별 옵션**

특정 바이너리 객체에만 적용되게 main\_ops와 lib\_opts을 이용할 수 있습니다.

* backend : 클래스 또는 이름으로 사용할 백엔드
* custom\_base\_addr : 사용할 기본 주소
* custom\_arch : 사용할 아키텍쳐 이름

예시)

angr.Project(main\_opts={'backend': 'ida', 'custom\_arch': 'i386'}, lib\_opts={'libc.so.6': {'backend': 'elf'}})

##### **백엔드**

CLE는 ELF, PE, CGC, Mach-O, ELF 코어 덤프 파일을 정적으로 로드하고 마찬가지로 IDA와 함께 로드할 수 있습니다.

일부 백엔드는 아키텍처를 자동으로 찾을 수 없기 때문에 custom\_arch를 지정해야 합니다.

| **백엔드** | **설명** | **custtom\_arch 필요여부** |
| --- | --- | --- |
| elf | PyELFTools 기반 ELF 정적 로더 | no |
| pe | PEFile 기반 PE 정적 로더 | no |
| mach-o | Mach-O 정적 로더. dynamic linking이나 리베이스를 지원하지 않습니다. | no |
| cgc | Cyber Grand Challenge 바이너리 정적 로더 | no |
| backedcgc | CGC 바이너리를 위한 정적 로더 | no |
| ida | IDA 인스턴스 | yes |
| blob | 메모리 안에 파일을 로드 | yes |

#### Symbolic 함수 요약

기본적으로 Project는 SimProcedures symbolic 요약에 의해 외부 호출을 라이브러리 함수로 대체합니다. 내장 프로시저인 angr.SIM\_PROCEDURES 딕셔너리에서 사용할 수 있습니다. libc, posix, win32, stubs 패키지와 라이브러리 함수 이름을 입력합니다. 실제 라이브러리 함수 대신에 SimProcdure를 실행하면 많은 분석을 쉽게 할 수 있습니다.

주어진 함수를 위한 가낭한 요약이 없는 경우:

* auto\_load\_libs이면 실제 라이브러리 함수가 실행됩니다. blic의 일부 함수는 분석하기 매우 복잡하고 실행하려고 하는 경로의 수가 엄청나게 증가할 수 있습니다.
* auto\_load\_libs가 False일 경우 Project는 ReturnUnconstrained라 불리는 SimProcdeure에 의해 해결됩니다.
* angr.Project가 아닌 cle.Loader 를 사용하는 use\_sim\_procedures가 False이면 (기본값은 True) SimProcedures와 함께 외부 객체에 의해 symbol이 제공됩니다.

##### 후킹

이 방식은 후킹이라 불리는 python 요약과 함께 라이브러리 코드로 대체될 수 있습니다. 또한 이를 수행할 수 있습니다. 모든 단계에서 angr가 현재 주소를 검사하고 일치하는 경우 해당 주소에서 바이너리 코드 대신 후킹을 진행합니다. proj.hook(addr, hook)의 hook은 SimProcedure 인스턴스입니다.

>>> stub\_func = angr.SIM\_PROCEDURES['stubs']['ReturnUnconstrained'] # this is a CLASS

>>> proj.hook(0x10000, stub\_func()) # hook with an instance of the class

>>> proj.is\_hooked(0x10000) # these functions should be pretty self-explanitory

True

>>> proj.unhook(0x10000)

>>> proj.hooked\_by(0x10000)

<ReturnUnconstrained>

>>> @proj.hook(0x20000, length=5)

... def my\_hook(state):

... state.regs.rax = 1

>>> proj.is\_hooked(0x20000)

True

### Symbolic 표현과 제약 해결

angr의 강력함은. 애뮬레이터가 아니라 symbolic 변수를 실행가능하는 것에 있습니다. 변수가 실제 값을 갖고있다고 말하는 것 대신에 단순히 이름을 효과적으로 나타내는 것입니다. 산술 연산을 수행하면 연산트리가 생성됩니다. AST는 z3와 같은 SMT solver 조건으로 변환 할 수 있습니다. "일정 순서의 결과가 주어지면 입력된 값은 무엇일까요?"와 같은 질문을 할 수 있는데 이를 angr로 사용하는 방법을 배우게 됩니다.

#### Bitvector로 작업

간단한 프로젝트를 만들어 봅니다.

>>> import angr, monkeyhex

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

>>> state = proj.factory.entry\_state()

bitvector는 어떤 수의 비트를 말합니다.

# 64-bit bitvectors with concrete values 1 and 100

>>> one = state.solver.BVV(1, 64)

>>> one

<BV64 0x1>

>>> one\_hundred = state.solver.BVV(100, 64)

>>> one\_hundred

<BV64 0x64>

# create a 27-bit bitvector with concrete value 9

>>> weird\_nine = state.solver.BVV(9, 27)

>>> weird\_nine

<BV27 0x9>

이러한 비트를 bitvector라고 부르고 이를 이용해 산술연산을 할 수 있습니다.

>>> one + one\_hundred

<BV64 0x65>

# You can provide normal python integers and they will be coerced to the appropriate type:

>>> one\_hundred + 0x100

<BV64 0x164>

# The semantics of normal wrapping arithmetic apply

>>> one\_hundred - one\*200

<BV64 0xffffffffffffff9c>

one + weird\_nine의 연산을 타입이 맞지 않아 연산을 할 수 없습니다. 따라서 weird\_nine의 길이를 확장할 수 있습니다.

>>> weird\_nine.zero\_extend(64 - 27)

<BV64 0x9>

>>> one + weird\_nine.zero\_extend(64 - 27)

<BV64 0xa>

zero\_extnd는 주어진 bitvector의 왼쪽에 0으로 채워넣습니다.

# Create a bitvector symbol named "x" of length 64 bits

>>> x = state.solver.BVS("x", 64)

>>> x

<BV64 x\_9\_64>

>>> y = state.solver.BVS("y", 64)

>>> y

<BV64 y\_10\_64>

x와 y는 중학교 수학에서 배운 변수와 같습니다. 이를 산술연산 할 수 있지만 숫자가 출력되지 않습니다. 대신 AST가 출력됩니다.

>>> 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>

AST에는 .op와 .args가 있습니다. op는 연산자의 이름을 나타내고 args는 사용되는 값을 말합니다. 연산이 아닌 경우에는 BVV, BVS 등으로 표현됩니다.

>>> tree = (x + 1) / (y + 2)

>>> tree

<BV64 (x\_9\_64 + 0x1) / (y\_10\_64 + 0x2)>

>>> tree.op

'\_\_div\_\_'

>>> 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)

#### Symbolic 표현

두 개의 AST에서 비교 연산을 하면 bitvector가 아닌 bool 값이 출력됩니다.

>>> 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>

마지막 연산을 보면 one\_hundred는 -5보다 큰 경우이지만 -5가. <BV64 0xfffffffffffffffb>로 표현되기 때문에 one\_hundred.SGT(-5)를 사용해야 합니다.

정확한 값이 없기 때문에 if나 while문에서 조건에 직접 변수를 비교해서는 안되며 있더라고 if one > one\_hundred는 예외를 발생합니다. 대신 solver.is\_true와 solver\_is\_false를 사용해야 합니다.

>>> 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

#### 제약 해결

>>> state.solver.add(x > y)

>>> state.solver.add(y > 2)

>>> state.solver.add(10 > x)

>>> state.solver.eval(x)

4

위와 같은 제약조건을 추가하면 state.solver.eval(y) 를 통해 x에 따른 y 값을 얻을 수 있습니다.

# get a fresh state without constraints

>>> 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

위 연산은 출력에 따른 입력을 찾는 것입니다. 만약 모순된 제약 조건을 추가하면 출력값이 이상하거나 예외를 발생시킵니다. state.satisfiable()를 이용해 만족을 하는지에 대한 여부도 알 수 있습니다.

>>> state.solver.add(input < 2\*\*32)

>>> state.satisfiable()

False

변수만 아니라 복잡한 식도 가능합니다.

# fresh state

>>> 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

#### 해결 함수

eval은 해답을 출력해주는데 여러개의 값을 원할 수 있습니다.

* solver.eval(expression) : 하나의 해결책을 출력합니다.
* solver.eval\_one(expression) : 해결책을 출력하지만 둘 이상의 해결책이 있다면 오류를 출력합니다.
* solver.eval\_upto(expression, n) : 최대 n개의 해결책을 주며 n보다 작으면 n보다 작은 수를 출력합니다
* solver.eval\_atleast(expression, n) : n보다 작으면 오류를 출력합니다.
* solver.eval\_exact(expression, n) : n개의 해답을 출력하고 더 적거나 많은 경우 오류를 출력합니다.
* solver.min(expression) : 최소한의 해답을 출력합니다.
* solver.max(expression) : 최대한의 해답을 출력합니다.

## 상태 - 메모리, 레지스터 등

angr의 작동 방식을 이해하기 위해서 SimState객체를 사용했습니다.

### 리뷰 : 메모리와 레지스터 쓰기 및 읽기

문서를 순서대로 읽었다면 메모리와 레지스터에 접근하는 방법을 이미 봤을 겁니다. state.regs는 레지스터의 이름, 속성, 읽기, 쓰기가 가능하고 state.mem은 메모리에 읽고 쓰는 것을 제공합니다.

레지스터와 메모리에 저장하기 위해서 bitvector 형식의 AST를 이해해야 합니다.

>>> import angr

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

>>> state = proj.factory.entry\_state()

# copy rsp to rbp

>>> state.regs.rbp = state.regs.rsp

# store rdx to memory at 0x1000

>>> state.mem[0x1000].uint64\_t = state.regs.rdx

# dereference rbp

>>> state.regs.rbp = state.mem[state.regs.rbp].uint64\_t.resolved

# add rax, qword ptr [rsp + 8]

>>> state.regs.rax += state.mem[state.regs.rsp + 8].uint64\_t.resolved

### 기본 실행

앞서 Simulation Manager를 통해서 기본 실행 방법을 봤습니다. 다음 장에서 Simulation Manager의 모든 기능에 대해서 설명하겠지만 state.setp()을 이용해 간단한 실행 방법을 보도록 하겠습니다. 이 함수는 한 단계씩 Simsuccessors에 의해 호출된 symbolic 실행과 리턴을 합니다.

Angr의 symbolic execution은 컴파일 된 개별 명령어의 작업을 수행하고 SimState을 변경하기 위해 수행하는 것입니다. if(x > 4)와 같은 코드에 도달했을 때, 만약 x가 bitvector일 경우 무슨 일이 일어날까요? angr가 분석하는 어딘가에서 x>4를 비교할 것이고 수행할 것입니다. 그 결과는 <Bool x\_32\_1 > 4>가 됩니다.

그렇다면 해당 비교에서 값이 "참"일지 "거짓" 중 어떤 값을 가져야 할까요? 답은 두 가지 모두를 갖는 것입니다. 참인 경우에 시뮬레이션을 하고 거짓인 경우 시뮬레이션하는 것을 각각 생성합니다.

이를 증명하기 위해서 [가짜 펌웨어](https://docs.angr.io/examples/fauxware/fauxware)를 예로 들어보겠습니다. 이 바이너리의 [소스코드](https://docs.angr.io/examples/fauxware/fauxware.c)를 살펴보면 펌웨어의 인증 메커니즘이 어떤 누구라도 관리자로 "SOSNEAKY"라는 비밀번호로 접근할 수 있습니다.

>>> proj = angr.Project('examples/fauxware/fauxware')

>>> state = proj.factory.entry\_state()

>>> 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

strcmp는 애뮬레이트하기 까다로운 함수이며 제약 조건이 매우 복잡합니다.

애뮬레이트 된 프로그램은 표준 입력에서 데이터를 가져오며 angr는 기본적으로 symbolic data 스트림으로 취급됩니다. 제약 조건을 해결하고 수행할 수 있는 가능한 입력을 얻으려면 stdin의 실제 내용을 참조해야 합니다. state.posix.files[0].all\_bytes()는 stdin에서 읽은 모든 내용을 bitvector를 검색하는데 사용합니다.

>>> input\_data = state1.posix.files[0].all\_bytes()

>>> state1.solver.eval(input\_data, cast\_to=str)

'\x00\x00\x00\x00\x00\x00\x00\x00\x00SOSNEAKY\x00\x00\x00'

>>> state2.solver.eval(input\_data, cast\_to=str)

'\x00\x00\x00\x00\x00\x00\x00\x00\x00S\x00\x80N\x00\x00 \x00\x00\x00\x00'

state1을 따라 가려면 "SOSNEAKY"를 암호로 입력해야 하고 state2로 가려면 다른 값을 입력해야 합니다. z3가 해당 조건에 맞는 문자열 중 하나를 찾아낸 것입니다.

### 상태 조절

상태를 사용하기 위해서 project.factory.entry\_state()를 사용했습니다. 이거슨 factory에서 사용할 수 있는 상태 중 하나입니다.

* .black\_state() : 공백 상태를 만듭니다.
* .entry\_state() : 바이너리의 초기 주소에서 실행 준비가 된 상태를 만듭니다.
* .full\_init\_state() : 공유 라이브러리 생성자 또는 미리 초기화하는 프로그램 처럼 바이너리에 접근하기 전 실행하는 초기화 프로그램을 통해 실행할 준비가 된 상태를 만듭니다.
* .call\_state() : 지정된 함수를 실행할 준비가 된 상태를 만듭니다.

위 생성자를 통해서 여러 인수를 통해 사용자가 정의할 수 있습니다.

* 모든 생성자는 addr로 시작할 주소를 설정할 수 있습니다.
* args를 이용해 환경 변수 env에 entry\_state와 full\_init\_date를 사용할 수 있습니다. 이 구조는 문자열이나 bitvector 입니다. args는 항상 비어있으며 찾고자 한다면 하나 이상이 있어야 합니다.
* entry\_state와 full\_init\_state에 argc로 bitvector를 넣을 수 있습니다. argc로 전달한 값이 args 수 보다 클 수 없습니다.
* call state를 사용하려면 .call\_state(addr, arg1, arg2, ...)을 호출할 수 있습니다. addr은 호출할 함수의 주소이며 argN은 N번째 함수의 인수입니다. 메모리를 할당하고 객체에 대한 포인터를 전달하려면 포인터를 PointerWrapper를 이용해야 합니다.
* 함수의 호출 규약을 지정하려면 cc 인자와 같은 SimCCinstance를 이용해야 합니다.

### 메모리용 Low level 인터페이스

state.mem은 형식이 정해진 데이터를 로드하는데 편리하지만 raw 로드나 저장을 할 경우 복잡합니다. state.memory에서 .load(addr, size)와 .store(addr, val) 함수를 이용해 직접 할 수 있습니다.

>>> s = proj.factory.blank\_state()

>>> s.memory.store(0x4000, s.solver.BVV(0x0123456789abcdef0123456789abcdef, 128))

>>> s.memory.load(0x4004, 6) # load-size is in bytes

<BV48 0x89abcdef0123>

데이터는 로드 된 후에 Big-Endian 방식으로 저장됩니다. state.memroy의 목적은 저장된 데이터를 로드하는 것이기 때문입니다. 방식을 변경하려는 경우 endness로 정할 수 있습니다.

>>> import archinfo

>>> s.memory.load(0x4000, 4, endness=archinfo.Endness.LE)

<BV32 0x67453201>

### 상태 플러그인

SimState는 실제로 플러그인에 저장됩니다. 플러그인 속성은 memory, registers, mem, regs, solber 등등이 있습니다. 이러한 방식은 모듈화 뿐만 아니라 쉽게 기능을 사용할 수 있습니다.

예를들어 memory 플러그인은 메모리 공간을 시뮬레이션 하지만 분석은 추상 메모리 프러그인을 사용하도록 선택할 수 있습니다.

#### Global 플러그인

state.globals는 간단한 플러그인이며, Python의 dictionary를 구현하여 임의의 데이터를 상태에 저장할 수 있게 합니다.

#### History 플러그인

state.history는 실행 중에 경로에 대한 기록 데이터를 저장하는 중요한 플러그인 입니다. 여러개의 노드가 연결되어 있고 각각 하나의 실행 횟수를 말합니다. state.history.parent.paret로 확인할 수 있습니다.

히스토리를 작업하기 편하도록 반복자를 제공합니다. history.recent\_NAME에 값이 저장되고 history.NAME으로 사용할 수 있습니다. 예를들어 for addr in state.history.bbl\_addrs: print hex(addr)은 바이너리가 실행되면서 가장 최근에 실행된 주소를 출력합니다. state.history.parent.reccent\_bbl\_addrs는 이전 단계를 말합니다.

* history.descriptions : 실행 횟수의 상태를 문자열로 나타냅니다.
* history.bbl\_addrs : state에의해 실행되는 기본 블록 주소입니다.
* history.jumpkinds : VEX 문자열과 같이 제어 흐름을 나타냅니다.
* history.guards : 상태의 각 지점의 조건 목록을 나타냅니다.
* history.events : 프로그램이 메시지를 띄우거나 종료하는 것과 같이 이벤트가 발생하는 목록을 말합니다.
* history.actions : 기본적으로 비어있는 것이지만 angr.options.refs 옵션을 추가하면 프로그램에서 수행한 메모리, 레지스터, 임시 값에 접근한 기록을 표시합니다.

#### CallStack 플러그인

angr는 애뮬레이트된 프로그램의 스택을 추적합니다. history와 마찬가지로 callstack도 연결된 노드이지만 반복자는 제공되지 않습니다. state.callstack을 이용하여 직접 반복하여 얻을 수 있습니다.

* callstack.func\_addr : 현재 실행중인 함수의 주소
* callstack.call\_site\_addr : 현재 함수를 호출한 블록 주소
* callstack\_stack\_ptr : 현재 함수의 첫 스택 포인터 값
* callstack.ret\_addr : 함수가 리턴할 경우 리턴할 위치

#### posix 플러그인

진행 중

### 파일 시스템 작업

진행 중: SimFile이 무엇인지.

파일 시스템을 효과적으로 사용하는 많은 상태 초기화 루틴 옵션이 있습니다. fs, concrete\_fs, chroot 옵션이 있습니다.

fs 옵션은 SimFile 객체에 파일 이름을 전달할 수 있습니다. 이렇게 사용하면 파일 내용에 구체적인 크기 제한을 설정하는 등의 작업을 수행할 수 있습니다.

concrete\_fs 옵션을 True로 설정하면 디스크에 있는 파일을 보호합니다. 예를들어, concrete\_fs을 false로 실행되면 시뮬레이션 중 프로그램이 'banner.txt' 열려고 시도하면 SimFile이 만들어지고 파일이 있는 것 처럼 시뮬레이션을 계속 합니다. concrete\_fs가 True이면 새로운 SimFile을 만들고 실행되는 결과의 영향을 최소화합니다. 만약 'banner.txt'가 존재하지 않는다면 SimFile은 시뮬레이션 중에 오류가 출력됩니다. 또한 경로가 '/dev/'로 시작하는 파일을 열려고 한다면 conrete\_fs가 true로 설정되어도 열리지 않습니다.

chroot 옵션은 경로를 지정할 수 있습니다. 분석중인 프로그램이 절대 경로를 이용해 파일을 참조할 때 편리할 수 있습니다. 예를 들어 /etc/passwd를 열려고 시도하는 경우 /etc/passwd가 $CMD/etc/passwd에서 읽힐 수 있도록 작업 디렉토리를 설정할 수 있습니다.

>>> files = {'/dev/stdin': angr.storage.file.SimFile("/dev/stdin", "r", size=30)}

>>> s = proj.factory.entry\_state(fs=files, concrete\_fs=True, chroot="angr-chroot/")

위 예제는 최대 30바이트를 stdin에서 읽도록 제한하는 상태를 만듭니다.

### 복사 및 병합

state는 엄청 빠른 복사를 지원합니다.

>>> 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

state를 병합할 수 있습니다.

# merge will return a tuple. the first element is the merged state

# the second element is a symbolic variable describing a state flag

# the third element is a boolean describing whether any merging was done

>>> (s\_merged, m, anything\_merged) = s1.merge(s2)

# this is now an expression that can resolve to "AAAA" \*or\* "BBBB"

>>> aaaa\_or\_bbbb = s\_merged.mem[0x1000].uint32\_t

진행 중: 병합의 한계

## Simulation Manager

angr에서 가장 중요한 제어 인터페이스는 SimulationManager입니다. 이를 사용하면 프로그램의 상태 공간을 탐색하기위한 검색 방법윽 적용하여 상태 그룹에 대한 symbolic execution을 동시에 제어할 수 있습니다.

Simulation Manager는 여러 상태를 다룰 수 있습니다. state는 숨겨져 있어 필터링하고 병합하고 원하는대로 이동할 수 있습니다.

### 단계

Simulation Manager는 .step()을 이용해 기본 블록에 의해 숨겨져있는 상태를 한 단계 이동시키는 것입니다.

>>> import angr

>>> proj = angr.Project('examples/fauxware/fauxware', auto\_load\_libs=False)

>>> state = proj.factory.entry\_state()

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

>>> simgr.active

[<SimState @ 0x400580>]

>>> simgr.step()

>>> simgr.active

[<SimState @ 0x400540>]

숨겨진 모델의 진정한 능력은 조건 분기문을 만나면 두가지 상태 모두 동기화 할 수 있다는 것입니다. 단계 별로 실행할 때 .run()을 사용할 수 있습니다.

# Step until the first symbolic branch

>>> while len(simgr.active) == 1:

... simgr.step()

>>> simgr

<SimulationManager with 2 active>

>>> simgr.active

[<SimState @ 0x400692>, <SimState @ 0x400699>]

# Step until everything terminates

>>> simgr.run()

>>> simgr

<SimulationManager with 3 deadended>

3개의 deadended 상태를 갖고 있습니다. 시스템의 exit에 도달했기 때문에 실행 중 다음 상태를 만들지 못하면 deadended가 출력됩니다.

### 은닉 관리

다른 은닉 방법을 살펴보겠습니다.

은닉된 것으로 이동할 경우 .move()를 사용할 수 있습니다.

>>> simgr.move(from\_stash='deadended', to\_stash='authenticated', filter\_func=lambda s: 'Welcome' in s.posix.dumps(1))

>>> simgr

<SimulationManager with 2 authenticated, 1 deadended>

### 은닉 타입

| **은닉** | **설명** |
| --- | --- |
| active | 다음 은닉 상태가 존재합니다. |
| deadended | 상태가 더 이상 유효하지 않고 unsat 상태 또는 유효하지 않은 명령어 포인터를 포함 등 실행할 수 없는 상태. |
| pruned | LAZY\_SOLVES를 사용할 때 상태는 필요한 경우를 제외하고 만족 여부를 확인하지 않습니다. 상태가 unsat로 판명되면 언제 unsat 상태가 됐는지 탐색합니다. |
| unconstrained | save\_unconstrained 옵션이 SimulationManager에 전달되면 명령 포인터를 사용해 제한되지 않은 상태가 만들어집니다. |
| unsat | save\_unsat 옵션이 SimulationManager에 전달되면 만족할 수 없는 상태가 만들어집니다. |

### 단순 탐색

가장 일반적인 작업은 특정 주소에 도달하는 상태를 찾고 다른 주소를 통과하는 상태를 제거하는 것입니다. .explore() 함수를 이용해 경로를 찾습니다.

find를 사용하면 해당 주소로 이동할 수 있을 때 까지 만족하는 조건을 찾아 실행합니다. avoid는 해당 주소를 회피하는데 사용합니다.

간단한 [crackme](https://docs.angr.io/docs/examples.html" \l "reverseme-modern-binary-exploitation---csci-4968)예제를 보겠습니다.

>>> proj = angr.Project('examples/CSCI-4968-MBE/challenges/crackme0x00a/crackme0x00a')

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

>>> simgr.explore(find=lambda s: "Congrats" in s.posix.dumps(1))

<SimulationManager with 1 active, 1 found>

>>> s = simgr.found[0]

>>> print s.posix.dumps(1)

Enter password: Congrats!

>>> flag = s.posix.dumps(0)

>>> print(flag)

g00dJ0B!