Open Source Project

조선대학교 IT 융합대학 컴퓨터공학과

3학년 20124855 박사홍

3학년 20124826 최용선

3학년 20134844 박효빈

Open Source Software

강문수 교수님

Open Source Project

참여자

조선대학교 IT융합대학 컴퓨터공학과 3학년 박사홍

조선대학교 IT융합대학 컴퓨터공학과 3학년 최용선

조선대학교 IT융합대학 컴퓨터공학과 3학년 박효빈

주제

https://docs.angr.io 한글화 및 버그 찾기

한글화 작업 부분

박사홍: Introduction, Core Concepts, Extending angr, Examples

최용선: Built-in Analyses, Advanced Topics, Appendix

박효빈: 문서작업 (한글화 통합, 히스토리)

오픈 소스 SW 선택 이유

angr는 바이너리 분석 및 수학 연산에 탁월한 방법을 제시하는 오픈 소스 프로젝트 입니다.

최근 해킹 대회에서는 사람이 쉽게 계산하지 못하는 수학적 연산 및 단순 사칙 연산임에도 불구하고 엄청난 많은 연산 횟수를 갖는 바이너리가 출제됩니다.

이럴때 angr를 이용할 경우 찾고자하는 주소 값과 피해야할 주소 값만 찾으면 쉽게 해결할 수 있습니다.

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 몇가지 문제를 해결해야 합니다.

```
- 분석 프로그램에 바이너리를 로드하기.
- 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.

시작하기

설치 방법은 여기에서 찾을 수 있습니다.

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}
}
```

지원

angr를 사용하는데 도움을 받으려면 아래를 참고해 주세요.

- Mail: angr@lists.cs.ucsb.edu

- Slack channel: angr.slack.com
- IRC channel: #angr on freenode

angr 설치하기

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

angr를 사용하고 실행하려면 Python 가상환경을 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 또는 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 설ㅈ치되지 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 capstone 라이브러리는
/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.

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 협약에 맞추고 있습니다. vim을 사용한다면 파이썬 모드 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?

sphinx를 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 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
2. claripy
3. cle
4. pyvex

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-management는 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.

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 API문서를 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.GenericJumpslotRelo
c at 0x4221fb0>,
 u'__tls_get_addr':
<cle.backends.relocations.generic.GenericJumpslotRelo
c at 0x425d150>,
 u'_dl_argv':
<cle.backends.relocations.generic.GenericJumpslotRelo
c at 0x4254d90>,
 u'_dl_find_dso_for_object':
<cle.backends.relocations.generic.GenericJumpslotRelo
c at 0x425d130>,
 u'_dl_starting_up':
<cle.backends.relocations.generic.GenericJumpslotRelo
c at 0x42548d0>,
 u'_rtld_global':
<cle.backends.relocations.generic.GenericJumpslotRelo
c at 0x4221e70>,
 u'_rtld_global_ro':
<cle.backends.relocations.generic.GenericJumpslotRelo
c 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		no

	PyELFTools 기반 ELF 정적 로더	
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 펌웨어를 예로 들어보겠습니다. 이 바이너리의 소스코드를 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\x
00 \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 방법을 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 crackme예제를 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!
```

시뮬레이션 및 연구

angr는 SimEngine 클래스를 사용하여 입력 상태에 미치는 영향을 애뮬레이션 합니다. 아래 목록은 기본 엔진 목록입니다.

- 이전 단계에서 불연속 상태일 때 실패 엔진이 작동합니다.
- 이전 단계에서 syscall 내에서 끝날 때 syscall 엔진이 작동합니다.
- 현재 주소가 후킹됐을 때 hook 엔진이 작동합니다.
- UNICORN 상태가 활성화 되거나 심볼릭 데이터가 존재하지 않는 경우 unicorn 엔진이 작동합니다.
- 마지막 대체로 VEX 엔진이 작동됩니다.

SimSuccessors

실제로 모든 엔진을 시도하는 코드는 project.factory.successors(state, **kwargs) 입니다. state.step()과 simulation_manager.step()가 핵심입니다.
SimSuccessors 객체를 반환하며 이전 문서에서 간략히 설명했습니다.
SimSuccessors의 목적은 successor 상태의 간단한 분류를 수행하고 변수 속성을 저장하는 것입니다.

| 속성 | 보호 조건 | 명령 포인터 | 설명 | --- | --- | | successors | True(심볼릭일 수 있지만 True로 제한함.) | 심볼릭일 수 있습니다.(256개의 솔루션이 있고 unconstrained_successors를 참고하세요.) | 일반적으로 만족스러운 successor 상태는 엔진에 의해 처리됩니다. 이 상태의 명령 포인터는 심볼릭일 것입니다. (입력에 따른 계산된 점프.) 실제로 몇 가지 잠재적인 실행을 나타낼지 모릅니다. | | unsat_successors | False(심볼릭일 수 있지만 False로 제한함.) | 심볼릭일 수 있습니다. | 만족하지 못한 successors. 이 successrs는 보호 조건이 오직 false 입니다. (점프가 이루어지지 않거나 기본 브랜치로 점프하는 것.) | | flat_successors | True(심볼릭일 수 있지만 True로 제한함.) | 구체적인 값 | 위에서 언급한 것 처럼, successors 목록은 symbolic 명령 포인터를 갖습니다. 상태가 흘러갈 때 simEngineVEX.process 처럼 복잡합니다. 이를 완화하기 위해서 symbolic 명령 포인터를 사용하여 다음 상태를 만다면 가능한 구체적인 모든 솔루션(256개 까지) 계산하고 솔루션의 상태를 복사합니다. 이 프로세스를 'flattening'이라 부릅니다. flat_successors는 상태이며 각각 다른 명령 포인터를 갖습니다. 예를들어,

successors의 상태가 x+5인 명령어 포인터라면 X가 X > 0x800000이고 X < 0x800010일 때 16개의 다른 flat_successors 상태로 만듭니다. 0x800006, 0x800007처럼 0x800015까지. | | unconstrained_successors | True(심볼릭일 수 있지만 True로 제한함.) | 256개 이상의 가능한 솔루션이 나타난 경우 명령 포인터는 덮어 쓰여집니다.(스택 오버플로우 같은.) | | unsat | save_unsat 옵션이 SimulationManager에 전달되면 만족할 수 없는 상태가 만들어집니다. | | all_successors | 아무거나 | 심볼릭일 수 있음. | succesors + unsat_successors + unconstrained_successors|

Breapoints

진행 중: 다시 내용 작성.

angr는 breakpoint를 지원합니다.

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

# get our state
>>> s = b.factory.entry_state()

# add a breakpoint. This breakpoint will drop into
ipdb right before a memory write happens.
>>> s.inspect.b('mem_write')

# on the other hand, we can have a breakpoint trigger
right *after* a memory write happens.
# we can also have a callback function run instead of
opening 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_func)

# or, you can have it drop you in an embedded
```

```
IPython!
>>> s.inspect.b('mem_write', when=angr.BP_AFTER,
action=angr.BP_IPYTHON)
```

메모리 쓰기 이외에 많은 break 구문이 존재합니다. BP_BEFORE 또는 BP_AFTER로 breakpoiont를 설정할 수 있습니다.

이벤트	의미
mem_read	메모리 읽기
mem_write	메모리 쓰기
reg_read	레지스터 읽기
reg_write	레지스터 쓰기
tmp_read	temp 읽기
tmp_write	temp 쓰기
expr	산술 연산 또는 상수의 결과
statement	IR 상태 해석
instruction	새로운 명령어 해석
irsb	새로운 블록 해석
consraints	새로운 제약이 상태에 추가
exit	실행으로부터 successor가 발생
symbolic_variable	새로운 심볼릭 변수 생성
call	call 명렁어 실행
address_concretization	심볼릭 메모리 접근

이벤트 들은 다른 속성을 나타냅니다.

이벤트	속성 이름	가능한
mem_read	mem_read_address	BP_BE 또는 BP_AF
mem_read	mem_read_length	BP_BE 또는 BP_AF
mem_read	mem_read_expr	BP_AF
mem_write	mem_write_address	BP_BE 또는 BP_AF
mem_write	mem_write_length	BP_BE 또는 BP_AF
mem_write	mem_write_expr	BP_BE 또는 BP_AF
reg_read	reg_read_offset	BP_BE 또는 BP_AF
reg_read	reg_read_length	BP_BE 또는 BP_AF
reg_read	reg_read_expr	BP_AF
reg_write	reg_write_offset	BP_BE 또는 BP_AF
reg_write	reg_write_length	BP_BE 또는

reg_write	reg_write_length	또는 BP_AF
reg_write	reg_write_expr	BP_BE 또는 BP_AF
tmp_read	tmp_read_num	BP_BE 또는 BP_AF
tmp_read	tmp_read_expr	BP_AF
tmp_write	tmp_write_num	BP_BE 또는 BP_AF
tmp_write	tmp_write_length	BP_AF
expr	expr	BP_AF
statement	statement	BP_BF 또는 BP_AF
call	function_address	BP_BE or BP_AF
exit	exit_target	BP_BE or BP_AF
exit	exit_guard	BP_BE or BP_AF
exit	jumpkind	BP_BE or BP_AF

symbolic_variable	symbolic_name	BP_BE or BP_AF
symbolic_variable	symbolic_size	BP_BE or BP_AF
symbolic_variable	symbolic_expr	BP_AF
address_concretization	address_concretization_strategy	BP_BE or BP_AF
address_concretization	address_concretization_action	BP_BE or BP_AF
address_concretization	address_concretization_memory	BP_BE or BP_AF
address_concretization	address_concretization_expr	BP_BE or BP_AF
address_concretization	address_concretization_add_constraints	BP_BE or BP_AF
address_concretization	address_concretization_result	BP_AF

위 속성은 breakpoint 콜백 중에 state.inspect으로 적절한 값에 접근할 수 있습니다. 이 값을 변경하여 수정할 수 있습니다.

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

추가적으로 각각의 inspect.b 키워드 인수로 사용하여 breakpoint를 조건에 맞춰 사용할 수 있습니다.

```
# This will break before a memory write if 0x1000 is
a possible value of its target expression
>>> s.inspect.b('mem_write',
mem_write_address=0x1000)

# This will break before a memory write if 0x1000 is
the *only* value of its target expression
>>> s.inspect.b('mem_write',
mem_write_address=0x1000,
mem_write_address_unique=True)

# This will break after instruction 0x8000, but only
0x1000 is a possible value of the last expression
that was read from memory
>>> s.inspect.b('instruction', when=angr.BP_AFTER,
instruction=0x8000, mem_read_expr=0x1000)
```

사실, 조건으로 함수를 지정할 수 있습니다.

```
# this is a complex condition that could do anything!
```

```
In this case, it makes sure that RAX is 0x41414141
and
# that the basic block starting at 0x8004 was
executed sometime in this path's history
>>> def cond(state):
...     return state.eval(state.regs.rax,
cast_to=str) == 'AAAA' and 0x8004 in
state.inspect.backtrace

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

2_Built-in Analyses

CFGAccurate

이 장에서는 angr의 context sensitivity나 Function Manager와 같은 중요한 개념을 살펴보면서 angr의 **CFGAccurate** 분석에 대해 자세하게 알아볼 것입니다.

개요

바이너리에서 할 수 있는 가장 기본적인 분석 방법은 **Control Flow Graph**입니다. 이 CFG는 바이너리에서 jumps/calls/rets/etc 등 실행의 분기가 일어나는 basic blocks를 시각적으로 표현하는 그래프입니다. angr에서는 _fast CFG_와 *accurate CFG(CFGAccurate)* 두 가지 형태의 CFG를 생성할 수 있습니다. 이름에서 유추할 수 있듯이, fast CFG를 생성하는 것은 일반적으로 accurate 방식보다 더 빠릅니다. 일반적으로 CFGFast가 사용자들이 원하는 형태일 것입니다. 이 페이지는 CFGAccurate에 대해 살펴볼 것입니다.

accurateCFG는 아래와 같은 명령어를 통해 만들어낼 수 있습니다.

```
>>> import angr
# load your project
>>> b = angr.Project('/bin/true', load_options=
{'auto_load_libs': False})

# generate an accurate CFG
>>> cfg = b.analyses.CFGAccurate(keep_state=True)
```

또한 CFG의 커스터마이징을 위한 옵션들도 존재합니다.

옵션	설명
	이는 context sensitivity의 분석 레벨을 세팅합니다. 이에 대한 정보를

context_sensitivity_level	확인하고싶다면 아래의 context sensitivity level 섹션을 살펴보세요. default 값은 1입니다.
starts	분석할 때 entry point로서 사용하기 위한 주소들의 리스트입니다.
avoid_runs	분석할 때 무시하고 진행하기 위한 주소들의 리스트입니다.
call_depth	몇몇 number calls에서 분석의 depth를 제한합니다. 이것은 "call_depth를 1로 세팅"하면서 직접 점프할 수 있는 특정한 함수들을 체크할 때 유용합니다.
initial_state	initial state는 CFG에 제공될 수 있으며, 이는 분석을 통해 사용됩니다.
keep_state	메모리를 절약하기 위해 기본적으로 각각의 basic block의 state가 삭제됩니다. 만약 _keep_state_가 True라면, 이 state는 CFGNode에 저장됩니다.
enable_symbolic_back_traversal	indirect jump를 해결하기 위해 집중할지의 여부
enable_advanced_backward_slicing	direct jump를 해결하기 위해 집중할지의 여부
more!	최근에 업데이트 되는 옵션들에 대한 정보는 b.analyses.CFGAccurate의 docstring을 확인하세요.

Context Sensitivity Level

angr은 모든 basic block을 실행하고 그 블록들을 검토하면서 CFG를 생성합니다. 이로 인해 몇몇 문제점들이 발생합니다. basic block은 다른 context들에서 다르게 작동할 수 있습니다. 예를 들어, block이 함수의 return으로 종료된다면 해당 basic block을 포함하는 함수의 다른 caller에 따라 return하는 대상이 달라집니다.

context sensitivity level은 개념적으로 caller가 callstack을 유지할 수 있는 수입니다. 이 개념을 설명하기 위해서 아래 코드를 살펴보겠습니다.

```
void error(char *error)
{
    puts(error);
}

void alpha()
{
    puts("alpha");
    error("alpha!");
}

void beta()
{
    puts("beta");
    error("beta!");
}

void main()
{
    alpha();
    beta();
}
```

위의 샘플은 main>alpha>puts, main>alpha>error>puts, main>beta>puts, main>beta>error>puts의 네 가지 콜 체인이 있습니다. 이 경우 angr은 두 체인은 실행할 수 있지만 큰 바이너리에서는 불가능합니다. 따라서 angr은 context sensitivity level에 의해 제한된 상태로 블록을 실행합니다. 즉, 각각의 함수는 호출된 고유한

context마다 재분석합니다.

예를 들어, 위의 puts() 함수는 다양한 context sensitivity level에서 아래와 같은 context로 분석됩니다.

레벨	의미	Contexts
0	Callee-only	puts
1	One caller, plus callee	alpha>puts beta>puts error>puts
2	Two callers, plus callee	alpha>error>puts main>alpha>puts beta>error>puts main>beta>puts
3	Three callers, plus callee	main>alpha>error>puts main>alpha>puts main>beta>error>puts main>beta>puts

context sensitivity level을 올리게 되면 CFG로부터 더 많은 정보를 얻을 수 있다는 장점이 있습니다. 예를 들어, context sensitivity가 1일 경우 CFG는 alpha에서 호출될 때 puts는 alpha를 return하며, error가 호출되면 puts에서 alpha를 return해줍니다. context sensitivity가 0일 경우 CFG는 간단하게 puts는 alpha, beta, 그리고 error를 return해줍니다. 이것은 구체적으로 IDA에서 사용되는 context sensitivity level입니다. context sensitivity level을 높이게 되면 분석 시간이 기하급수적으로 증가한다는 단점이 있습니다.

Using the CFG

CFG의 코어는 NetworkX di-graph입니다. 즉, 모든 일반 NetworkX API들을 사용할 수 있다는 말이 되겠죠.

```
>>> print "This is the graph:", cfg.graph
>>> print "It has %d nodes and %d edges" %
(len(cfg.graph.nodes()), len(cfg.graph.edges()))
```

CFG 그래프의 노드는 CFGNode 클래스의 인스턴스입니다. context sensitivity로 인해 주어진 basic block은 그래프에서 다중 노드를 가질 수 있습니다.(다중 context의 경우)

```
# this grabs *any* node at a given location:
>>> entry_node = cfg.get_any_node(b.entry)

# on the other hand, this grabs all of the nodes
>>> print "There were %d contexts for the entry
block" % len(cfg.get_all_nodes(b.entry))

# we can also look up predecessors and successors
>>> print "Predecessors of the entry point:",
entry_node.predecessors
>>> print "Successors of the entry point:",
entry_node.successors
>>> print "Successors (and type of jump) of the entry
point:", [ jumpkind + " to " + str(node.addr) for
node,jumpkind in
cfg.get_successors_and_jumpkind(entry_node) ]
```

Viewing the CFG

CFG의 렌더링은 어려운 문제입니다. angr은 CFG 분석 결과를 렌더링하기 위한 built-in 메커니즘을 제공하지 않습니다. 그리고 matplotlib와 같은 기존의 그래프 렌더링 라이브러리를 사용하려고 시도한다면 이미지를 사용할 수 없는 결과를 초래하게 됩니다. angr CFG를 보기 위한 하나의 솔루션은 axt's angr-utils repository입니다.

Shared Libraries

CFG 분석은 서로 다른 이진 객체의 코드 흐름을 구분하지 않습니다. 즉, 이것은 기본적으로 로드된 공유 라이브러리를 통해 control flow를 분석하려고 시도한다는 것입니다. 분석 시간을 며칠로 연장할 것이기 때문에 이것은 의도된 행동이 아닙니다. 공유 라이브러리 없이 바이너리를 로드하기 위해서는 Project constructor에 다음의 키워드 인자를 추가하세요: load_options={'auto_load_libs': False}

Function Manager

CFG의 결과는 cfg.kb.functions를 통해 접근할 수 있는 Function Manager라고 불리는 오브젝트를 생성합니다. 이 객체를 가장 일반적으로 사용하는 경우는 dictionary와 같은 것에 접근하는 것입니다. 이것은 주소를 Function 객체에 매핑합니다. 이 객체는 함수에 대한 속성을 알려줍니다.

```
>>> entry_func = cfg.kb.functions[b.entry]
```

Functions에는 몇가지 중요한 속성이 있습니다.

- entry_func.block_addrs는 함수에 속하는 basic block이 시작하는 주소들의 집합입니다.
- entry_func.blocks는 capstone을 사용하여 디스어셈블하고 탐색할 수 있는 함수에 속한 basic block의 집합입니다.
- entry_func.string_references()는 함수의 어느 지점에서든 참조된 모든 상수 문자열들의 리스트를 반환합니다. 그것들은 (addr, string) 튜플의 형태인데, addr은 바이너리의 데이터 섹션에 있는 주소이고, strings는 문자열의 값을 포함하는 python string입니다.
- entry_func_returning은 함수가 return할 수 있는지 없는지 검증하는 boolean 값입니다. False는 모든 경로가 반환되지 않는 것을 나타냅니다.
- entry_func.callable은 이 함수를 참조하는 angr Callable 객체입니다. python 인자를 가진 python 함수와 같이 호출할 수 있으며, 이러한 인수를 사용하여 함수를 실행한 것처럼 실제 결과(symbolic일 수 있다)를 얻어낼 수 있습니다.
- entry_func.transition_graph는 함수 자체 내의 control flow를 설명하는 NetworkX DiGraph입니다. IDA가 기능별 수준에서 표시해주는 CFG와 비슷합니다.
- entry_func.name은 함수의 이름입니다.
- entry_func.has_unresolved_calls와 entry.has_unresolved_jumps는 CFG 내의 부정확성을 감지하는 것과 관련이 있습니다. 때대로 간접 호출이나 jump가 가능한 대상이 무엇인지 감지할 수 없습니다. 만약 함수 내에서 발생한다면 그 함수는 적절한 has_unresolved_*의 값을 True로 세팅합니다.

- entry_func.get_call_sites()는 다른 함수의 호출에서 끝의 basic block에 해당하는 모든 주소들의 집합을 반환합니다.
- entry_func.get_call_target(callsite_addr)은 callsite_addr이 call site address의 리스트에서 주어질 경우 해당 callsite가 호출할 위치를 반환합니다.
- entry_func.get_call_return(callsite_addr)은 callsite_addr이 call site address의 리스트에서 주어질 경우 해당 callsite가 반환될 위치를 반환합니다.

이 외에도 많은 속성들이 있습니다!

2_Built-in Analyses

백워드 슬라이싱

프로그램 슬라이스는 일반적으로 0개 이상의 명령문을 제거하여 원본 프로그램으로부터 가져온 명령문의 하위 집합입니다. 슬라이싱은 디버깅과 프로그램을 이해하는데 종종 도움이 됩니다. 예를 들어, 일반적으로 프로그램 슬라이스에서 변수의 원천을 찾는 것이 더 쉽습니다. 백워드슬라이스는 프로그램의 대상에서 구성되며, 대상에서 이 슬라이스 끝의 모든 데이터 플로우를 나타냅니다. angr은 백워드슬라이스라는 기본 내장 분석 기능이 있어 백워드 프로그램 슬라이스를 구성합니다. 이 섹션에서는 angr의 백워드슬라이스 분석이 어떻게 동작하는지, 그리고 구현 선택과 제약 사항에 대한 심도 깊은 토론이 이어집니다.

First Step First

백워드슬라이스를 만들기 위해서는 아래의 정보들을 입력해야합니다.

- **Required** CFG. 프로그램의 CFG. 이 CFG는 반드시 accurate CFG(CFGAccurate)여야만 합니다.
- **Required** Target. 백워드 슬라이스가 끝나는 최종 목적지입니다.
- **Optional** CDG. Control Dependence Graph(CDG)는 CFG에서 파생된 것입니다. angr은 이러한 목적을 위해 CDG 분석 기능이 내장되어있습니다.
- **Optional** DDG. Data Dependence Graph(DDG)는 CFG의 위에 구축되어있습니다. angr은 이러한 목적을 위해 DDG 분석 기능이 내장되어있습니다.

백워드슬라이스는 아래의 코드로 구성될 수 있습니다.

```
>>> import angr
# Load the project
>>> b = angr.Project("examples/fauxware/fauxware",
load_options={"auto_load_libs": False})
```

```
# Generate a CFG first. In order to generate data
dependence graph afterwards,
# you'll have to keep all input states by specifying
keep_stat=True. Feel free
# to provide more parameters (for example,
context_sensitivity_level)for CFG
# recovery based on your needs.
>>> cfg =
b.analyses.CFGAccurate(context_sensitivity_level=2,
keep_state=True)

# Generate the control dependence graph
>>> cdg = b.analyses.CDG(cfg)

# Build the data dependence graph. It might take a
while. Be patient!
>>> ddg = b.analyses.DDG(cfg)

# See where we wanna go... let's go to the exit()
call, which is modeled as a
# SimProcedure.
>>> target_func =
cfg.kb.functions.function(name="exit")
# We need the CFGNode instance
>>> target_node = cfg.get_any_node(target_func.addr)

# Let's get a BackwardSlice out of them!
# `targets` is a list of objects, where each one is
either a CodeLocation
# object, or a tuple of CFGNode instance and a
statement ID. Setting statement
# ID to -1 means the very beginning of that CFGNode.
A SimProcedure does not
# have any statement, so you should always specify -1
for it.
>>> bs = b.analyses.BackwardSlice(cfg, cdg=cdg,
ddg=ddg, targets=[ (target_node, -1) ])

# Here is our awesome program slice!
```

```
>>> print bs
```

때때로 DDG를 얻는것이 힘들 수 있습니다. 그렇다면 간단하게 CFG 위에 프로그램 슬라이스를 만들 수도 있습니다. 이것이 기본적으로 DDG가 선택적 파라미터인 이유입니다. 다음을 통해 CFG를 기반으로 백워드슬라이스를 만들 수 있습니다.

```
>>> bs = b.analyses.BackwardSlice(cfg,
control_flow_slice=True)
BackwardSlice (to [(<CFGNode exit (0x10000a0) [0]>,
-1)])
```

백워드슬라이스 객체 사용하기

백워드슬라이스 객체를 사용하기 전에 이 클래스의 디자인이 현재 상당히 임의적이라는 것을 알아두세요. 그리고 근 시일 내에 변경될 수 있습니다.

구성

구성 후 백워드슬라이스는 프로그램 슬라이스를 설명하는 아래와 같은 구성 요소들이 있습니다.

멤버	모드	의미
runs_in_slice	CFG-only	프로그램 슬라이스에 있는 블록 및 SimProcedure의 주소와 이들 사이의 전환을 보여주는 networkx.DiGraph 인스턴스
cfg_nodes_in_slice	CFG-only	프로그램 슬라이스에 있는 CFGNode 사이의 전환을 보여주는 networkx.DiGraph 인스턴스
chosen_statements	With DDG	basic block 주소를 프로그램 슬라이스의 일부인 statement ID의 리스트에 매핑하는 딕셔너리

chosen_exits	With DDG	basic block 주소를 "exits" 목록에 매핑하는 딕셔너리. 리스트의 각 exit는 프로그램 슬라이스에서 유효한 전환입니다.

chosen_exit의 각 "exit"는 statement ID와 대상 주소 리스트를 포함하는 튜플입니다. 예를 들어, "exit"는 다음과 같을 수 있습니다.

```
(35, [ 0x400020 ])
```

만약 "exit"가 basic block의 default exit이라면 다음과 같을 것입니다.

```
(“default”, [ 0x400085 ])
```

Export an Annotated Control Flow Graph

공식 문서 추가 예정

User-friendly Representation

BackwardSlice.dbg_repr()를 살펴보세요!

공식 문서 추가 예정

Implementation Choices

공식 문서 추가 예정

Limitations

공식 문서 추가 예정

Completeness

공식 문서 추가 예정

Soundness

공식 문서 추가 예정

2_Built-in Analyses

Identifier

Identifier는 CGC 바이너리에서 common library function을 식별하기 위해 테스트 케이스를 사용합니다. 스택의 변수/인자에 대한 기본 정보를 찾아 사전에 필터링합니다. 스택 변수에 대한 정보는 일반적으로 다른 프로젝트에서 유용할 수 있습니다.

```
>>> import angr

# get all the matches
>>> p =
angr.Project("../binaries/tests/i386/identifiable")
>>> idfer = p.analyses.Identifier()
# note that .run() yields results so make sure to
iterate through them or call list() etc
>>> for addr, symbol in idfer.run():
...     print hex(addr), symbol

0x8048e60 memcmp
0x8048ef0 memcpy
0x8048f60 memmove
0x8049030 memset
0x8049320 fdprintf
0x8049a70 sprintf
0x8049f40 strcasecmp
0x804a0f0 strcmp
0x804a190 strcpy
0x804a260 strlen
0x804a3d0 strncmp
0x804a620 strtol
0x804aa00 strtol
0x80485b0 free
0x804aab0 free
0x804aad0 free
```

```
0x8048660 malloc
0x80485b0 free
```

3_Advanced topics

Gotchas when using angr

이번 섹션에서는 angr에서 users/victims가 자주 실행되는 gotchas의 리스트가 포함되어있습니다.

SimProcedure inaccuracy

symbolic execution을 하기 쉽게 만들기 위해 angr은 common library function을 파이썬으로 작성한 summary들로 교체합니다. 이러한 summary들을 SimProcedures라고 부릅니다. SimProcedures는 symbolic string에서 실행되는 strlen과 같이 path explosion를 막아줍니다.

1. SimProcedure를 비활성화 하세요(angr.Project class에 옵션을 전달하여 특정 SimProcedures를 제외할 수 있습니다). 이것은 문제의 함수에 대한 입력을 제한하는 것에 매우 주의하지 않는 한 path explosion으로 이어질 가능성이 있다는 단점이 있습니다. path explosion은 Veritesting과 같은 다른 angr 기능을 통해 부분적으로 막을 수 있습니다.
2. SimProcedure를 문제의 상황에 직접 작성된 것으로 대체하세요. 예를 들어, 우리의 scanf 구현은 완벽하지 않지만 알려진 format string의 단일 문자열을 지원하기만 하면 정확히 수행 할 hook를 작성할 수 있습니다.
3. SimProcedure를 수정하세요.

Unsupported syscalls

시스템 콜은 SimProcedures로 구현됩니다. 안타깝게도 angr에서는 아직 구현되지 않은 시스템 콜이 존재합니다. 지원되지 않는 시스템 콜은 몇 가지 해결 방안이 있습니다.

1. 시스템 콜을 구현하세요. TODO: 추후 공식 문서 업데이트 예정
2. project.hook를 사용하여 callsite를 ad-hoc 방식으로 상태에 필요한 수정을 하기 위해 후킹합니다.
3. syscall 반환 값을 큐에 넣으려면 state.posix.queued_syscall_returns 리스트를 사용하세요. 만약 반환 값이 큐에 있다면 시스템 콜이 실행되지 않고 해당

값이 대신 사용됩니다. 게다가 함수를 "반환 값"으로 대신 큐에 넣을 수 있으므로 시스템 콜이 트리거 될 때 함수가 그 상태에 적용됩니다.

Symbolic memory model

angr에서 사용하는 기본 메모리 모델은 Mayhem에서 영감을 얻었습니다. 이 메모리 모델은 제한된 symbolic R/W를 지원합니다. 읽기의 메모리 인덱스가 symbolic이고 이 인덱스의 가능한 범위 값의 범위가 너무 넓으면 인덱스는 단일 값으로 구체화됩니다. 쓰기의 메모리 인덱스가 전반적으로 symbolic이라면 인덱스는 단일 값으로 구체화됩니다. 이것은 state.memory의 메모리 구체화 전략을 변경하여 구성할 수 있습니다.

Symbolic lengths

SimProcedures와 특히 read()및 write()와 같은 시스템 콜은 버퍼의 길이가 symbolic인 상황으로 진행될 수 있습니다. 일반적으로 이것을 처리하는 것은 매우 어렵습니다: 많은 경우에, 이 길이는 나중에 실행 단계에서 철저하게 구체화되거나 소급적으로 구체화됩니다. 심지어 그렇지 않은 경우에도 원본 또는 대상 파일이 약간 "이상하게" 보일 수 있습니다.

3_Advanced topics

Understanding the Execution Pipeline

만약 이것을 만드는데 코어에 대한 지식이 거의 없다면, angr는 매우 유연하고 강력한 에뮬레이터입니다. 가장 많은 효과를 얻으려면, path_group.step()을 말할때마다 어떤 일이 일어나는지 알고 싶을 것입니다. 이것은 더 진보된 문서로 만들 생각입니다; PathGroup, ExplorationTechnique, Path, SimState 및 SimEngine의 기능과 의도를 이해해야만 지금 무엇에 관해 말하려는지 이해할 수 있을 것입니다! 이제 angr 소스를 열어서 이 작업을 수행할 수 있습니다.

Path Groups

첫 발을 내딛어보죠.

step()

PathGroup.step() 함수는 많은 선택적 파라미터를 가지고 있습니다. 이 중 가장 중요한 것은 stash, n, until과 step_func입니다. n은 즉시 사용됩니다. step() 함수는 _one_step() 함수를 호출하고 n 단계가 발생하거나 다른 종료 조건이 발생할 때까지 모든 파라미터를 전달하면서 루프를 반복합니다. n이 제공되지 않으면 until 함수가 제공되지 않는 한 1이 default 값입니다. 이 경우 until은 100000 – 무한입니다.

그러나 종료 조건을 확인하기 전에 step_func가 적용됩니다. 이 함수는 현재 경로의 그룹을 가져오고, 새로운 경로 그룹을 반환하여 이를 대체합니다. step function을 작성할 때 대부분의 공통 경로 그룹 함수가 경로 그룹을 반환한다는 점을 상기하는 것이 유용합니다 – 만약 경로 그룹이 변경 불가능할 경우(생성자에서 immutable=True), 새 오브젝트이지만 그것은 이전과 같은 동일한 오브젝트입니다.

이제 종료 조건을 확인해봅시다 – 우리가 작업중인 stash("active"가 default)가 비었거나, until 콜백 함수가 True를 반환할 때입니다. 만약 이러한 조건이 모두 충족되지 않을 경우, _one_step()을 다시 호출하기 위해 루프백합니다.

_one_step()

이것은 ExplorationTechnique가 무언가에 영향을 끼칠 수 있는 곳입니다. 만약 어떤 exploration technique가 step override를 제공했다면, 이것이 어디선가 호출되었다는 것입니다. 이 기술이 영리한 점은 이들의 효과가 결합될 수 있다는 점입니다. 어떻게 이런 일이 가능한걸까요? step을 구현하는 exploration technique에는 경로 그룹이 주어지며, 새 경로 그룹을 반환하고, 아주 약간 전진하며 exploration technique의 효과를 적용시킵니다. 이것은 필연적으로 경로 그룹에서 step()을 호출하는 exploration technique와 관련됩니다. 그러면 이 문서가 다시 시작하면서 설명하는 사이클은 프로세스가 _one_step()에 도달했을 때를 제외하고 현재 exploration technique가 스텝 콜백의 리스트에서 튀어나온다는 것입니다. 이런 스텝 콜백을 제공하는 exploration technique들이 더 많이 있다면, 다음 것이 호출되며 리스트가 빌 때까지 반복합니다. 콜백에서 반환이 되면, _one_step은 콜백을 콜백 스택으로 푸시하고 반환합니다.

요약하자면, 스텝 콜백을 제공하는 exploration technique는 다음과 같이 처리됩니다:

- 엔드 유저가 step()을 호출
- step()이 _one_step()을 호출
- _one_step()이 active step exploration technique 콜백 리스트로부터 단일 exploration technique를 꺼내고, 현재 작업중인 경로 그룹으로 호출
- 이 콜백은 호출 된 경로 그룹에서 step()을 호출
- 이 프로세스는 더 이상의 콜백이 없을 때까지 반복

일단 스텝 콜백이 더이상 없거나 시작하는 스텝 콜백이 더이상 없을 경우 디폴트 스테핑 프로시저로 돌아갑니다. 여기에는 원래 PathGroup.step() – selector_func에 전달되었던 파라미터가 하나 더 포함됩니다. 만약 존재하는 경우에는, 우리가 실제로 작동할 작업 stash의 경로를 필터링하는 데 사용됩니다. 이 각각의 경로에 대해 PathGroup._one_path_step()을 호출하여 아직 사용되지 않은 모든 파라미터를 다시 전달합니다. _one_path_step()은 해당 경로를 스테핑 한 경로(normal, unconstrained, unsat, pruned, errored)를 분류한 리스트의 튜플을 반환합니다. 유틸리티 함수인 PathGroup._record_step_results()는 이러한 목록들을 작업하여 경로 그룹에 포함된 stash들의 새로운 집ㅎ바을 반복하여 작성하고, exploration technique가 제공할 수 있는 필터 콜백을 적용합니다.

_one_path_step()

PathGroup에 대한 전반을 거의 다 만들었습니다. 먼저, step_path exploration technique hook을 적용해야합니다. 이러한 hook은 스텝 콜백만큼 예쁘게 적용되지는 않습니다. 단 하나만 적용할 수 있으며, 나머지는 실패한 경우에만 사용됩니다. step_path hook가 성공하면 _one_path_step()에서 즉시 결과가 반환됩니다. 필터 콜백에 대한 요구사항은 _one_path_step()이 반환해야하는 리스트와 동일한 튜플을 반환하는 것입니다. 이들 모두가 실패하거나, 더이상 시작할 것이 없다면 다시 default 프로시저로 돌아갑니다.

Note: 공식 문서에서 리팩토링 예정입니다.

먼저, 이 경로에 에러가 있는지 확인합니다. 이 작업은 check_func 파라미터를 통해 step()으로 진행되며, _one_path_step()으로 완전히 전달되거나 해당 함수가 제공되지 않으면 errored 속성을 통해 경로가 전달됩니다. 경로에 오류가 발생하면 이는 즉시 중단되고, errored stash에 경로를 표시합니다. 그리고 다음 경로로 이동합니다. 만약 successor_func가 step()에 대한 파라미터로 제공되면 사용됩니다 - "normal" successor의 리스트를 반환합니다. 만약 파라미터가 제공되지 않으면 경로에서 step()을 호출합니다. 이 함수는 normal successor의 리스트를 반환하는 동일한 속성을 갖습니다. 그리고 다음 경로의 unconstrained_successors와 unsat_successors 속성에 접근하여 해당하는 리스트들을 검색합니다. 이들이 모두 거대한 튜플의 적절한 위치에 반환됩니다.

Paths

Path는 약간 재앙이며, 곧 사라질 것입니다. 현재로서는 Path.step()에 대한 대부분의 파라미터를 successor generation 프로세스로 전달한 다음 successor 각각을 가져와서 새 경로로 매핑한다는 것만 알면 될 것입니다. 그리고 그 방법에 따라 error-catching을 수행하고 실행 계보에 대한 일부 메타데이터를 효율적으로 기록합니다. kicker는 step()이 호출되면 Path가 결국 project.factory.successors (state, **kwargs)를 호출한다는 것입니다.

Engine Selection

바라건대, angr 문서는 당신이 이 페이지에 도달할 때까지 SimEngine이 어떻게 상태를 취하고 successor를 생성할지 알고 있는 장치라는 것을 안다고 생각하겠습니다. 어떤 엔진을 사용할 지 어떻게 알 수 있을까요? 각각의 프로젝트는 factory에서 엔진 목록을

가지고 있으며, project.factory.successors의 기본 동작은 작업의 첫번째 결과를 순서대로 모두 가져오는 역할을 합니다. 물론 이 동작을 변경할 수 있는 몇가지 방법이 있습니다.

- default_engine=True 매개변수가 전달되면 시도 할 엔진은 일반적으로 last-resort default 엔진인 SimEngineVEX입니다.
- 리스트가 매개변수 엔진에서 전달되면 기본 엔진 목록 대신 사용됩니다.

기본 엔진 목록은 기본적으로 다음과 같습니다.

- SimEngineFailure
- SimEngineSyscall
- SimEngineHook
- SimEngineUnicorn
- SimEngineVEX

각 엔진에는 check() 메소드가 있습니다. 이는 사용하는 것이 적절한지 신속하게 판별합니다. 만약 check()가 통과되면 process()가 실제로 successor를 생성하는데 사용됩니다. 반면 check()가 통과되더라도 .processed 속성이 False로 설정된 SimSuccessors 객체를 반환하면 process()가 실패할 수 있습니다. 이 두 메소드는 모두 쌓여있는 프로시저들의 우위에 의해 아직 필터링되지 않은 모든 키워드 인수의 파라미터로 전달됩니다. 유용한 파라미터 중 일부는 addr과 jumpkind이며, 일반적으로 상태에 대해 추출되는 각 정보에 대한 override로 사용됩니다.

마지막으로 엔진이 상태를 처리하면 시스템 콜의 경우 명령 포인터를 수정하기 위해 결과가 잠시 후 처리됩니다. 실행이 ljk_Sys*라는 jumpkind와 함께 종료되면 SimOS를 호출하고, 현재 syscall에 대한 주소를 검색합니다. 그리고 결과 상태의 명령 포인터가 해당 주소로 변경됩니다. 원래 주소는 ip_at_syscall이라는 상태 레지스터에 저장됩니다. 이것은 순수한 실행에는 필요하지 않지만 정적 분석에서는 syscall을 일반 코드와 별도의 주소에 두는 점에서 유용합니다.

Engines

SimEngineFailure는 에러 케이스를 처리합니다. 이전의 jumpkind가 ljk_EmFail, ljk_Sig*, ljk_NoDecode(주소가 연결되지 않은 경우에만 해당) 또는 ljk_Exit 중 하나일 때만 사용됩니다. 처음 네가지 경우는 예외를 발생시키는 행동을 합니다. 마지막

경우는 successor를 단순히 생성하지 않는 방식입니다.

SimEngineSyscall은 syscall을 서비스합니다. 이전의 jumpkind가 Ijk_Sys* 형식의 항목일 때 사용됩니다. SimOS를 호출하여 syscall에 응답하면서 실행해야하는 SimProcedure를 검색한 다음 실행합니다. 매우 간단하죠!

SimEngineHook은 angr에서 후킹 기능을 제공해줍니다. 상태가 후킹된 주소에 있고 이전 jumpkind가 Ijk_NoHook이 아닌 경우에 사용됩니다. 단순히 주어진 훅을 찾고, SimProcedure 인스턴스를 검색하기 위해 hook.instantiate()를 호출한 다음 해당 프로시저를 실행합니다. 이 클래스는 후킹을 위해 특수화된 SimEngineProcedure 클래스의 하위 서브클래스입니다. 파라미터 프로시저가 필요하므로 항상 검사가 성공하게 되고, 훅에서 가져올 SimProcedure 대신 이 프로시저가 사용됩니다.

SimEngineUnicorn은 Unicorn Engine으로 구체적인 실행을 수행합니다. 상태 옵션인 o.UNICORN이 활성화되었을 때 사용되고, 최대 효율성을 위해 설계된 다른 조건이 충족될 때 사용됩니다.(아래 설명 참조)

SimEngineVEX는 큰 역할을 합니다. 이전의 항목들을 사용할 수 없을 때마다 사용됩니다. 현재 주소에서 IRSB로 바이트를 lift 시도한 다음 IRSB를 symbolic하게 실행합니다. 이 프로세스를 제어할 수 있는 많은 파라미터가 있으므로 API reference 참조하세요.

SimEngineVEX가 IRSB를 파면서 실행하는 정확한 프로세스는 문서화가 잘 되어있습니다.

Engine instances

스테핑 프로세스에 대한 파라미터 외에도 이 엔진들의 새로운 버전을 인스턴스화 할수도 있습니다. 각 엔진이 어떤 옵션을 사용할 수 있는지 확인하려면 API 문서를 참조하세요. 새 엔진 인스턴스가 생기면 step 프로세스로 전달하거나 자동으로 사용되도록 project.factory.engines 목록에 직접 입력할 수도 있습니다.

When using Unicorn Engine

o.UNICORN 상태에 옵션을 추가하면 모든 단계에서 SimEngineUnicorn이 호출되며, 구체적으로 Unicorn을 사용하여 실행되는지 확인할 수 있습니다.

보통 원하는 것은 미리 정의된 o.unicorn(소문자) 옵션 세트를 프로그램의 상태에 추가하는 것이겠죠.

```
unicorn = { UNICORN, UNICORN_SYM_REGS_SUPPORT,
INITIALIZE_ZERO_REGISTERS,
UNICORN_HANDLE_TRANSMIT_SYSCALL }
```

이것들은 당신의 경험을 크게 향상시킬 몇 가지 추가 기능과 default값들을 가능하게 할 것입니다. 또한 state.unicorn 플러그인에서 조정할 수 있는 많은 옵션이 있습니다.

unicorn이 작동하는 방식을 이해하는 좋은 방법은 로그를 출력하면서 확인하는 것입니다.

(logging.getLogger('angr.engines.unicorn_engine').setLevel('DEBUG');
logging.getLogger('angr.state_plugins.unicorn_engine').setLevel('DEBUG')
을 unicorn에서 실행한 샘플입니다.

```
INFO    | 2017-02-25 08:19:48,012 |
angr.state_plugins.unicorn | started emulation at
0x4012f9 (1000000 steps)
```

여기서 angr은 0x4012f9의 기본 블록부터 unicorn 엔진으로 전환됩니다. 최대 단계 수는 1000000으로 설정되어 있으므로 실행하는동안 Unicorn에서 1000000 블록 동안 유지되면 자동으로 튀어나옵니다. 이것은 무한 루프에 빠지는 것을 방지하기 위한 것입니다. 블록 수는 state.unicorn.max_steps 변수를 통해 구성할 수 있습니다.

```
INFO    | 2017-02-25 08:19:48,014 |
angr.state_plugins.unicorn | mmap [0x401000,
0x401fff], 5 (symbolic)
INFO    | 2017-02-25 08:19:48,016 |
angr.state_plugins.unicorn | mmap [0x7fffffffffe0000,
0x7fffffffffeffff], 3 (symbolic)
INFO    | 2017-02-25 08:19:48,019 |
angr.state_plugins.unicorn | mmap [0x6010000,
```

```
0x601ffff], 3
INFO    | 2017-02-25 08:19:48,022 |
angr.state_plugins.unicorn | mmap [0x602000,
0x602fff], 3 (symbolic)
INFO    | 2017-02-25 08:19:48,023 |
angr.state_plugins.unicorn | mmap [0x400000,
0x400fff], 5
INFO    | 2017-02-25 08:19:48,025 |
angr.state_plugins.unicorn | mmap [0x7000000,
0x7000fff], 5
```

angr는 unicorn 엔진이 접근할 때 엑세스하는 데이터의 지연 매핑을 수행합니다. 0x401000은 실행중인 명령어들의 페이지이고, 0x7fffffffffe0000은 스택입니다. 이 페이지 중 일부는 symbolic입니다. 즉, 액세스 할 때 Unicorn에서 실행이 중단되는 데이터가 적어도 일부는 포함되어 있는 것을 의미합니다.

```
INFO    | 2017-02-25 08:19:48,037 |
angr.state_plugins.unicorn | finished emulation at
0x7000080 after 3 steps: STOP_STOPPOINT
```

실행은 3개의 basic block(계산 된 낭비, 필요한 설정 등을 고려하는 것)을 위해 Unicorn에서 머무르며, 그 이후 simprocedure의 위치에 도달하고 angr에서 simproc을 실행하기 위해 점프합니다.

```
INFO    | 2017-02-25 08:19:48,076 |
angr.state_plugins.unicorn | started emulation at
0x40175d (1000000 steps)
INFO    | 2017-02-25 08:19:48,077 |
angr.state_plugins.unicorn | mmap [0x401000,
0x401fff], 5 (symbolic)
INFO    | 2017-02-25 08:19:48,079 |
angr.state_plugins.unicorn | mmap [0x7fffffffffe0000,
0x7fffffffffeffff], 3 (symbolic)
INFO    | 2017-02-25 08:19:48,081 |
```

```
angr.state_plugins.unicorn | mmap [0x6010000,
0x601ffff], 3
```

simprocedure 이후에 실행은 Unicorn으로 점프하여 돌아옵니다.

```
WARNING | 2017-02-25 08:19:48,082 |
angr.state_plugins.unicorn | fetching empty page
[0x0, 0xfff]
INFO    | 2017-02-25 08:19:48,103 |
angr.state_plugins.unicorn | finished emulation at
0x401777 after 1 steps: STOP_EXECNONE
```

바이너리가 zero-page에 접근했기 때문에 실행은 거의 바로 Unicorn에서 나타납니다.

```
INFO    | 2017-02-25 08:19:48,120 |
angr.engines.unicorn_engine | not enough runs since
last unicorn (100)
INFO    | 2017-02-25 08:19:48,125 |
angr.engines.unicorn_engine | not enough runs since
last unicorn (99)
```

쓰레싱과 Unicorn(비용이 큼)에서 벗어나기 위해 Unicorn으로 다시 진입하기 전에 특정 조건(i.e. X 블록에 대한 symbolic 메모리 액세스 없이)을 기다리는 cooldown(state.unicorn 플로그인의 속성)이 있습니다. unicorn의 실행은 simprocedure 또는 syscall 이외에 다른 조건 때문에 중단됩니다. 여기서 100 블록만큼 기다리기 위한 조건은 점프해서 돌아오기 전에 점프하는 것입니다.

3_Advanced topics

Speed considerations

분석 도구 또는 에뮬레이터로서 angr의 속도는 파이썬으로 작성되었기 때문에 사실 좀 느립니다. 어쨌든, angr를 더 빠르게 하기 위한 최적화 옵션과 조정값들은 많이 존재합니다.

General tips

- pypy를 사용하세요. Pypy는 파이썬 코드를 최적화된 jitting을 수행하는 파이썬 인터프리터의 대체품입니다.
- 필요 없으면 공유 라이브러리를 로드하지 마세요. angr의 기본 설정은 OS 라이브러리에서 직접 로드하는 것을 포함하여 한 바이너리와 호환되는 공유 라이브러리를 찾기 위해 큰 비용을 들입니다. 이것은 많은 시나리오에서 상황을 복잡하게 만들 수 있죠. bare-bone symbolic execution보다 추상적인 분석을 수행할 경우, 특히 CFG를 만들 경우에는 다루기 쉽도록 정밀도를 희생시키려는 trade-off를 원할 수 있습니다. angr은 존재하지 않는 함수에 대한 라이브러리 호출이 발생할 때 수행하지 않습니다.
- 후킹과 SimProcedures를 사용하세요. 공유 라이브러리를 사용하는 경우에는 점프하려고 하는 복잡한 라이브러리 함수에 대해 SimProcedure를 작성해야합니다.
- SimInspect를 사용하세요. SimInspect는 많이 사용되지 않지만 angr의 가장 강력한 기능 중 하낭비니다. 메모리 index 분석(angr에서 다소 느림)을 포함하여 거의 모든 angr 동작을 연결하고 수정할 수 있습니다.
- 구체적인 전략을 세우세요. 메모리 index 문제에 대한것 보다 더욱 효과적인 해결책은 전략을 구체화 하는 것입니다.
- 대체 Solver를 사용하세요. angr.options.REPLACEMENT_SOLVER 옵션을 사용하여 대체 Solver를 사용할 수 있습니다. 대체 솔버를 사용하면 solving time을 구체적으로 지정할 수 있습니다. 약간 문제가 있고 까다로운편이긴 하지만 좋은 해결책이 될 수 있을 것입니다.

If you're performing lots of concrete or partially-concrete

execution

- unicorn 엔진을 사용하세요. 만약 unicorn 엔진을 설치할 경우 구체적인 에뮬레이션을 위해 angr에서 얻는 이점이 많을 것입니다. 이를 활성화하려면 angr.options.unicorn 옵션을 추가하세요.
- fast memory와 fast register를 활성화하세요. angr.options.FAST_MEMORY와 angr.options.FAST_REGISTERS가 이를 도와줄 것입니다. 이것들은 메모리/레지스터를 정확도를 다소 포기하는 대신 속도를 향상시키는 메모리 모델을 사용합니다.
- 입력 시간을 최소화하세요. 실행하기 전에 입력을 대표하는 symbolic data로 state.posix.files[0]을 채운 다음 원하는 방식으로 symbolic data를 제한한 다음 구체적인 파일 크기를 설정합니다.(state.posix.files[0].size = ???)
- afterburner를 사용하세요. unicorn을 사용하는 동안 UNICORN_THRESHOLD_CONCRETIZATION 옵션을 추가하면 angr는 symbolic 값들을 구체화하여 unicorn에서 더 많은 시간을 할애할 수 있도록 임계값을 늘립니다. 값의 종류는 아래와 같습니다.
    - state.se.unicorn.concretization_threshold_memory
    - state.se.unicorn.concretization_threshold_registers
    - state.se.unicorn.concretization_threshold_instruction
    - state.se.unicorn.always_concretize
    - state.se.unicorn.never_concretize
    - state.se.unicorn.concretize_at

3_Advanced topics

Intermediate Representation

일반적인 x86 뿐만 아니라 MIPS, ARM 및 PowerPC같은 다양한 CPU 아키텍쳐의 기계어 코드를 분석하고 실행하기 위해 angr는 수행되는 기본 동작의 대부분의 분석을 수행합니다. angr의 IR, VEX(Valgrind에서 VEX를 따옴)를 이해하면서 매우 빠른 정적분석을 할 수 있고, angr이 작동하는 방식을 더 잘 이해할 수 있습니다.

VEX IR은 다른 아키텍처를 다룰 때 몇 가지 아키텍처들의 차이를 추상화하여 모든 아키텍처에 대해 단일 분석을 수행할 수 있도록 합니다.

- **Register names**. 레지스터의 크기와 이름은 아키텍처마다 다르지만 최신 CPU들은 보통 공통적인 아키텍처를 가집니다. CPU에는 여러가지 범용 레지스터, 스택 포인터를 보관할 레지스터, 조건 플래그를 저장하는 레지스터 세트 등이 있습니다. IR은 다른 플랫폼의 레지스터에 일관되고 추상화 된 인터페이스를 제공합니다. 특히, VEX는 레지스터를 정수 오프셋을 갖는 별도의 메모리 공간으로 모델링합니다.(예를 들어, AMD64의 rax는 메모리 공간의 어드레스 16에서 시작하여 저장)

- **Memory access**. 다른 아키텍처는 각기 다른 방식으로 메모리에 엑세스합니다. 예를 들어 ARM은 리틀 엔디안 및 빅 엔디안 모드로 메모리에 엑세스 할 수 있습니다. IR은 이러한 차이를 추상화하여 진행합니다.

- **Memory segmentation**. x86과 같은 일부 아키텍처는 특수한 세그먼트 레지스터를 이용하여 메모리 분할을 지원합니다. IR은 이러한 메모리 액세스 메커니즘을 이해하며 진행합니다.

- **Instruction side-effects**. 대부분의 instruction에는 side-effect를 가지고 있습니다. 예를 들어 ARM에서 Thumb 모드의 대부분의 연산은 조건 플래그를 업데이트하고 push/pop 명령은 스택 포인터를 업데이트합니다. 분석할 때 이러한 side-effect를 임시 방편으로 추적하는 것은 말도 안되는 일이기 때문에 IR은 이러한 영향을 확실하게 만듭니다. IR에는 많은 선택 사항이 있습니다. 우리는 VEX를 사용하는데, 바이너리 코드를 VEX로 올리는 것이 매우 잘 되어 있기

때문입니다. VEX는 여러 아키텍처를 지원하며, 기계어 타겟의 수를 표현하는데 부작용이 없습니다. 이는 기계어 코드를 프로그램 분석을 쉽게 하기 위한 표현으로 추상화시켜줍니다. 이 표현에는 네 가지 주요 클래스의 객체가 있습니다.

- **Expressions**. IR 표현식은 계산된 값 또는 상수 값을 나타냅니다. 여기에는 메모리 로드, 레지스터 읽기 및 산술 연산 결과가 포함됩니다.

- **Operations**. IR 오퍼레이션은 IR 표현식의 수정을 설명합니다. 여기에는 정수 산술, 부동 소수점 산술, 비트 연산 등이 포함됩니다. IR 표현식에 적용된 IR 연산은 결과적으로 IR 표현식을 산출해냅니다.

- **Temporary variables**. VEX는 임시 변수를 내부 레지스터로 사용합니다. IR 표현식은 사용 시 임시 변수에 저장됩니다. 임시 변수의 내용은 IR 표현식을 사용하여 검색할 수 있습니다. 이 때 t0부터 시작하여 번호가 매겨집니다. 이러한 임시 변수는 확실한 타입으로 정의됩니다.(e.g. "64-bit integer" 또는 "32-bit float" 등)

- **Statements**. IR문은 메모리 저장 및 레지스터 쓰기에 영향을 미치는 대상 시스템의 상태 변화를 모델링합니다. IR문은 필요할만한 값에 IR 표현식을 사용합니다. 예를 들어, 메모리를 저장하는 IR 표현식은 쓰기 대상 주소에 대해 IR 표현식을 사용하고, 내용에 대해 다른 IR 표현식을 사용합니다.

- **Blocks**. IR 블록은 대상 아키텍쳐에서 extended basic block("IR Super Block" 또는 "IRSB"로 표기)을 나타내는 IR문의 모음입니다. 블록에는 여러개의 exit가 포함될 수 있습니다. basic block 중간중간에서 조건부 exit문의 경우 특별한 Exit IR문이 사용됩니다. IR 표현식은 블록 끝의 무조건 종료 대상을 나타내는데 사용됩니다.

VEX IR은 실제로 VEX 저장소의 libvex_ir.h 파일 (https://github.com/angr/vex/blob/master/pub/libvex_ir.h)에 자세히 나와있습니다.

Condition flags computation (for x86 and ARM)

x86 및 ARM CPU에서 가장 일반적인 instruction side-effect중 하나는 zero flag, carry flag, overflow flag와 같은 조건 플래그를 업데이트하는 것입니다. 컴퓨터

설계자는 일반적으로 특수 레지스터(x86에서는 EFLAGS/RFLAGS, ARM에서는 APSR/CPSR)에 이런 플래그를 연결시킵니다.(각 조건 플래그가 1비트에 해당하기 때문에 플래그를 연결) 이 특수 레지스터는 프로그램 상태에 대한 중요한 정보를 저장하며, CPU의 올바른 에뮬레이션에서 중요한 역할을 합니다.

3_Advanced topics

Working with Data and Conventions

종종 분석중인 프로그램에서 구조화된 데이터에 엑세스 하길 원하는 경우가 있습니다. angr은 이러한 부분을 해소하기 위한 몇 가지 기능이 있습니다.

Working with types

angr에는 type을 나타내는 시스템이 있습니다. 이러한 SimTypes는 angr.types에 있습니다. 이 클래스중 하나의 인스턴스는 type을 나타냅니다. 대부분의 type은 SimState로 대체하지 않으면 불완전합니다. 크기는 실행중인 아키텍쳐에 따라 다릅니다. ty.with_state를 사용하여 이 작업을 수행할 수 있으며, 지정된 상태로 자신의 복사본을 반환합니다.

angr는 또한 pycparser 주위에 경량 래퍼를 가지고 있는데, 이것은 C 파서입니다. 이렇게 하면 객체의 타입에 대한 인스턴스를 가져오는데 도움이 됩니다.

```
>>> import angr

# note that SimType objects have their __repr__
defined to return their c type name,
# so this function actually returned a SimType
instance.
>>> angr.types.parse_type('int')
int

>>> angr.types.parse_type('char **')
char**

>>> angr.types.parse_type('struct aa {int x; long
y;}')
struct aa

>>> angr.types.parse_type('struct aa {int x; long
```

```
y;}').fields
OrderedDict([('x', int), ('y', long)])
```

또한 C definition을 파싱하여 변수/함수 선언 또는 새로 정의된 타입을 dictionary로 반환할 수 있습니다.

```
>>> angr.types.parse_defns("int x; typedef struct
llist { char* str; struct llist *next; } list_node;
list_node *y;")
{'x': int, 'y': struct llist*}

>>> defs = angr.types.parse_types("int x; typedef
struct llist { char* str; struct llist *next; }
list_node; list_node *y;")
>>> defs
{'list_node': struct llist}

# if you want to get both of these dicts at once, use
parse_file, which returns both in a tuple.

>>> defs['list_node'].fields
OrderedDict([('str', char*), ('next', struct
llist*)])

>>> defs['list_node'].fields['next'].pts_to.fields
OrderedDict([('str', char*), ('next', struct
llist*)])

# If you want to get a function type and you don't
want to construct it manually,
# you have to use parse_defns, not parse_type
>>> angr.types.parse_defns("int x(int y, double z);")
{'x': (int, double) -> int}
```

마지막으로 나중에 사용할 수 있도록 구조체의 정의를 등록할 수 있습니다.

```
>>> angr.types.define_struct('struct abcd { int x;
int y; }')
>>>
angr.types.register_types(angr.types.parse_types('typ
edef long time_t;'))
>>> angr.types.parse_defns('struct abcd a; time_t
b;')
{'a': struct abcd, 'b': long}
```

이러한 타입 객체는 그 자체로는 유용하지 않지만 angr의 다른 부분으로 전달되어 데이터 타입을 지정할 수 있습니다.

Accessing typed data from memory

이제 angr의 타입 시스템이 어떻게 작동하는지 알았으니 state.mem 인터페이스의 모든 기능을 사용할 수 있습니다. types 모듈에 등록된 모든 유형을 사용하여 메모리에서 데이터를 추출할 수 있습니다.

```
>>> import angr
>>> b = angr.Project('examples/fauxware/fauxware')
>>> s = b.factory.entry_state()
>>> s.mem[0x601048]
<<untyped> <unresolvable> at 0x601048>

>>> s.mem[0x601048].long
<long (64 bits) <BV64 0x4008d0> at 0x601048>

>>> s.mem[0x601048].long.resolved
<BV64 0x4008d0>

>>> s.mem[0x601048].long.concrete
0x4008d0

>>> s.mem[0x601048].abcd
<struct abcd {
```

```
  .x = <int (32 bits) <BV32 0x4008d0> at 0x601048>,
  .y = <int (32 bits) <BV32 0x0> at 0x60104c>
} at 0x601048>

>>> s.mem[0x601048].long.resolved
<BV64 0x4008d0>

>>> s.mem[0x601048].long.concrete
4196560L

>>> s.mem[0x601048].deref
<<untyped> <unresolvable> at 0x4008d0>

>>> s.mem[0x601048].deref.string
<string_t <BV64 0x534f534e45414b59> at 0x4008d0>

>>> s.mem[0x601048].deref.string.resolved
<BV64 0x534f534e45414b59>

>>> s.mem[0x601048].deref.string.concrete
'SOSNEAKY'
```

인터페이스는 다음과 같이 동작합니다.

- 먼저 [array index notation]을 사용하여 로드하려는 주소를 지정하세요.
- 그 주소에 포인터가 있다면, deref 속성에 접근하여 메모리에 있는 주소에 SimMemView를 반환할 수 있습니다.
- 그 다음 해당 이름의 속성에 액세스하여 데이터 타입을 지정합니다. 지원되는 타입 목록을 확인하려면 state.mem.types를 확인하세요.
- 그리고 타입을 조정할 수 있습니다. 모든 타입은 선호하는 모든 상세 검색을 지원합니다. 현재 지원되는 유일한 기능은 멤버 이름으로 구조체의 멤버에 액세스할 수 있으며, 해당 요소에 액세스하기 위해 문자열이나 배열에 인덱스 할 수 있다는 것입니다.
- 지정한 주소가 처음에 해당 타입의 배열을 가리키는 경우 .array(n)을 사용하여 데이터를 n개의 요소로 구성된 배열로 둘 수 있습니다.
- 마지막으로 .resolved 또는 .concrete로 구조화된 데이터를 추출하세요.

.resolved는 비트 벡터 값을 반환하고, .concrete는 데이터를 가장 잘 나타내는 정수, 문자열, 배열 등의 값을 반환합니다.

- 또는 생성한 속성 체인에 할당하여 메모리에 값을 저장할 수 있습니다. 파이썬이 작동하는 방식때문에 x = s.mem [...]. prop; x = val은 작동하지 않기 때문에 s.mem [...]. prop=val이라고 작성해야합니다.

define_struct 또는 register_types를 사용하여 구조체를 정의하면 여기서 타입으로 액세스 할 수 있습니다.

```
>>> s.mem[b.entry].abcd
<struct abcd {
  .x = <int (32 bits) <BV32 0x8949ed31> at 0x400580>,
  .y = <int (32 bits) <BV32 0x89485ed1> at 0x400584>
} at 0x400580>
```

3_Advanced topics

Solver Engine

angr의 솔버 엔진은 Claripy라고 불립니다. Claripy는 다음을 노출합니다.

- Claripy AST(claripy.ast.Base의 하위 클래스)는 복잡하고 symbolic한 식과 상호작용할 수 있는 통일된 방법을 제공합니다.
- Claripy 프론트엔드는 서로 다른 백엔드에서 식을 해결(제약적인 solving 포함)에 대한 통일된 인터페이스를 제공합니다.

내부적으로, Claripy는 여러가지 다른 백엔드(복잡한 bitvector, VSA construct, SAT solver)의 공동 작업을 원활하게 중재합니다. 이것들 참 골치아픈 녀석들입니다.

대부분의 angr 사용자는 Claripy와 직접 소통할 필요는 없습니다(단, claripy AST 객체는 symbolic 식을 나타내는데, 이는 제외). --angr는 Claripy와의 대부분의 상호작용을 내부적으로 처리합니다. 그러나 식을 다루기 위해서는 Claripy에 대한 이해가 있는 편이 좋을 수 있습니다.

Claripy ASTs

Claripy AST는 Claripy가 지원하는 구문간의 차이점을 추상화시킵니다. 이들을 기본 데이터의 타입에 대한 작업 트리(즉, (a + b) / c)를 정의합니다. Claripy는 요청을 백엔드에 전달하여 기본 객체 자체에 이러한 작업의 적용을 처리하도록 합니다.

현재 Claripy는 다음과 같은 유형의 AST를 지원합니다.

이름	설명	서포팅 주체(Claripy 백엔드)	예제 코드
	이것은 bitvector이며,		1) 32비트 symbolic bitvector "x" : claripy.BVS('x', 32) 2) 0xc001b3475의 값을 갖는 32비트 bitvector :

BV	symbolic(이름 포함)이거나 concrete(값) 합니다. 이는 비트 단위의 크기를 갖습니다.	BackendConcrete, BackendVSA, BackendZ3	claripy.BVV(0xc001b3a75, 32) 3) 1000과 2000 사이의 10으로 나눌 수 있는 32비트 "strided interval"(VSA 문서 참조) : claripy.SI(name='x', bits=32, lower_bound=1000, upper_bound=2000, stride=10)
FP	이것은 소숫점을 가진 수이며, symbolic(이름 포함)이거나 concrete(값) 입니다.	BackendConcrete, BackendZ3	TODO
Bool	이것은 boolean 연산입니다 (True 또는 False)	BackendConcrete, BackendVSA, BackendZ3	claripy.BoolV(True) 또는 claripy.true 또는 claripy.false 또는 두 AST를 비교할 때는 claripy.BVS('x', 32) < claripy.BVS('y', 32)와 같이 사용

위의 생성 코드는 모두 claripy.AST 객체를 반환하며, 이 객체를 사용하여 작업을 수행할 수 있습니다. AST는 여러가지 유용한 작업을 제공합니다.

```
>>> import claripy

>>> bv = claripy.BVV(0x41424344, 32)

# Size - you can get the size of an AST with .size()
>>> assert bv.size() == 32
```

```
# Reversing - .reversed is the reversed version of
the BVV
>>> assert bv.reversed is claripy.BVV(0x44434241, 32)
>>> assert bv.reversed.reversed is bv

# Depth - you can get the depth of the AST
>>> print bv.depth
>>> assert bv.depth == 1
>>> x = claripy.BVS('x', 32)
>>> assert (x+bv).depth == 2
>>> assert ((x+bv)/10).depth == 3
```

AST에 조건(==, != 등)을 적용하면 수행되는 조건을 나타내는 AST가 반환됩니다.

```
>>> r = bv == x
>>> assert isinstance(r, claripy.ast.Bool)

>>> p = bv == bv
>>> assert isinstance(p, claripy.ast.Bool)
>>> assert p.is_true()
```

이런 조건들을 다양한 방법으로 조합할 수 있습니다.

```
>>> q = claripy.And(claripy.Or(bv == x, bv * 2 == x,
bv * 3 == x), x == 0)
>>> assert isinstance(p, claripy.ast.Bool)
```

이것의 유용한 점은Claripy solver를 사용할 때 명확해질 것입니다. 일반적으로 Claripy는 모든 일반 파이썬 작업(+, -, |, == 등)을 지원하며 Claripy 인스턴스 객체를 통해 추가 작업을 제공합니다. 후자에서 사용할 수 있는 작업 목록이 있습니다.

이름	설명	예시

LShR	bit 표현식(BVV, BV, SI) 오른쪽 논리 쉬프트 연산	claripy.LShR(x, 10)
SignExt	bit 표현식에서 Sign-extend	claripy.SignExt(32, x) 또는 x.sign_extend(32)
ZeroExt	bit 표현식에서 Zero-extend	claripy.ZeroExt(32, x) 또는 x.zero_extend(32)
Extract	bit 표현식에서 주어진 비트 (우측부터 포함하여 에서 zero-indexed)를 추출	x의 가장 우측 바이트 추출 : Claripy.Extract(7, 0, x) 또는 x[7:0]
Concat	몇 개의 bit 표현식을 합치거나 새로운 표현식을 만듬	claripy.Concat(x, y, z)
RotateLeft	bit 표현식을 좌측으로 회전	claripy.RotateLeft(x, 8)
RotateRight	bit 표현식을 우측으로 회전	claripy.RotateRight(x, 8)
Reverse	bit 표현식을 반전	claripy.Reverse(x) 또는 x.reversed
And	논리 and	claripy.And(x == y, x > 0)
Or	논리 Or	claripy.Or(x == y, y < 10)
Not	논리 Not	claripy.Not(x == y)는 x != y와 동일
If	if-then-else	최대 값을 선택 : claripy.If(x > y, x, y)
ULE	Unsigned, 작거나 같을 때	claripy.ULE(x, y)
ULT	Unsigned, 작을 때	claripy.ULT(x, y)
UGE	Unsigned, 크거나 같을 때	claripy.UGE(x, y)

UGT	Unsigned, 클 때	claripy.UGT(x, y)
SLE	Signed, 작거나 같을 때	claripy.SLE(x, y)
SLT	Signed, 작을 때	claripy.SLT(x, y)
SGE	Signed, 크거나 같을 때	claripy.SGE(x, y)
SGT	Signed, 클 때	claripy.SGT(x, y)

3_Advanced topics

Symbolic memory addressing

angr은 symbolic memory addressing을 지원합니다. 즉 메모리로의 오프셋은 symbolic일 수 있습니다. 우리는 "Mayhem"에서 영감을 얻었습니다. 특히 이것은 angr이 symbolic address를 쓰기 대상으로 사용할 때 구체화된다는 것을 의미합니다. symbolic write를 순수하게 symbolic하게 다루거나 symbolic read를 취급하는것처럼 "symbolically"하게 기대하는 경향이 있기 때문에 약간 놀랐습니다. 그러나 이는 기본 기능이 아닙니다. 하지만 angr의 대부분의 것들처럼 이 또한 구성 가능합니다.

주소를 해석하는 동작은 구체화된 전략에 의해 관리되며, 구체화된 전략은 angr.concretization_stategies.SimConcretizationStrategy의 하위 클래스입니다. 읽기에 대한 구체화 전략은 state.memory.read_strategies에 설정되고 쓰기는 state.memory.write_strategies에 저장됩니다. 이러한 전략은 순서대로 호출되며, 그 중 하나가 symbolic index의 주소를 확인할 수 있습니다. 자신의 구체화 전략을 설정하거나(위에서 설명한 SimInspect address_concretization breakpoint를 이용) angr가 symbolic address를 해석하는 방법을 변경할 수 있습니다.

예를 들어 angr의 기본 구체화 전략은 아래와 같습니다.

1. angr.plugins.symbolic_memory.MultiwriteAnnotation으로 주석 처리된 모든 index에 대한 symbolic write(최대 128개의 가능한 솔루션 포함)을 허용하는 조건부 구체화 전략
2. symbolic index의 가능한 최대 해를 선택하는 구체화 전략 모든 인덱스에 대해 symbolic write를 가능하게 하려면 상태 생성 시 SYMBOLIC_WRITE_ADDRESSES 옵션을 추가하거나

angr.concretization_strategies.SimConcretizationStrategyRange 객체를 state.memory.write_strategies에 저장합니다. strategy 객체는 하나의 인수를 취하는데, 이것은 가능한 포기할 수 있는 최대 범위의 솔루션을 포기하고 다음(아마 non-symbolic) 전략으로 이동합니다.

Writing concretization strategies

공식 문서 추가 예정

후킹과 SimProcedures 상세 설명

angr의 후킹은 엄청 강력합니다. 상상하는 어떤 방법으로든 프로그램을 수정할 수 있습니다. 후킹을 프로그래밍하는 것은 정확하지 않을 수 있는데 이번 장은 SimProcedures를 프로그래밍하는데 도움이 될 것입니다.

시작

어떤 프로그램에서 버그를 제거하는 예제입니다.

```
>>> from angr import Project, SimProcedure
>>> project = Project('examples/fauxware/fauxware')

>>> class BugFree(SimProcedure):
...     def run(self, argc, argv):
...         print 'Program running with argc=%s and
argv=%s' % (argc, argv)
...         return 0

# this assumes we have symbols for the binary
>>> project.hook_symbol('main', BugFree)

# Run a quick execution!
>>> simgr = project.factory.simulation_manager()
>>> simgr.run()  # step until no more active states
Program running with argc=<SAO <BV64 0x0>> and argv=
<SAO <BV64 0x7fffffffffeffa0>>
<SimulationManager with 1 deadended>
```

프로그램 실행시 main 함수에 도달하면 실제 main 함수를 실행하는 대신에 procedure가 실행됩니다. 그리고 메시지와 리턴 값을 출력합니다.

함수에 진입할 때 인자 값은 run()을 이용하여 정의할 수 있습니다. SimProcedure는 인자 값을 호출 규약에 의해서 자동으로 뽑아내 함수를 실행합니다. 마찬가지로, 반환 값도 호출 규약에 따라 반환 됩니다. 아키텍처에 의존하고 연결된 레지스터나 스택의 pop

결과로 이동합니다.

context 구현

Project 클래스의 dict는 SimProcedure의 project._sim_procedures에 매핑됩니다. 실행 파이프라인이 dict 안에 존재하는 후킹된 주소라면 project._sim_procedures[address].execute(state)를 실행합니다. 호출 규약에 따라 인자 값을 가져오고 안정성을 유지하기 위해 복사를 하고 run() 함수를 실행합니다. SimProcedure를 실행하는 프로세스는 반드시 SimProcedure에서 상태를 변경해야하기 때문에 각 단계마다 별도의 인스턴스가 필요하므로 실행될 때마다 SimProcedure의 새 인스턴스를 만드는 것이 중요합니다.

kwargs

계층 구조는 단일 SimProcedure를 여러개의 후킹에서 사용할 수 있습니다. 같은 SimProcedure를 후킹하고 싶을 때 run()의 args를 추가하면 됩니다.

데이터 타입

위 예제를 보면 run() 함수의 인자 값을 추력하면 바뀐 <<SAO <BV64 0xSTUFF>> 클래스가 출력된 것을 볼 수 있습니다. 이것은 SimActionObject입니다. SimProcedure에서 정확이 어떤 것을 해야할지 추적하고 정적분석에 도움을 줍니다.

또한 프로시저에서 Pyhon int 0을 반환 된 것을 볼 수 있습니다. 자동으로 word 크기의 bitvector로 확장됩니다. native 수, bitvector, SimActionObejct가 반환됩니다.

부동 소수점을 처리하는 프로시저를 작성하려면 호출 규약을 수동으로 지정해야 합니다. cc에서 후킹을 제공합니다. : cc=project.factory.cc_from_arg_kinds((True, True), ret_fp=True) 및 project.hook(address, ProcedureClass(cc=mycc))

제어흐름

SimProcedure를 종료하려면 run()에서 반환하면 됩니다. self.ret(value)를 호출하는 약어입니다. self.ret()는 함수에서 특정 작업을 수행하는 방법을 알고있는 함수입니다.

- ret(expr) : 함수에서 반환

- jump(addr) : 해당 주소로 점프
- exit(code) : 프로그램 종료
- call(addr, args, continue_at) : 함수 호출
- inline_call(procedure, *args) : SimProcedure를 호출하고 결과를 반환.

조건부 이탈

SimProcedure에서 조건 분기를 추가하려면 현재 실행 단계에서 SimSuccessor 객체로 직접 작업해야 합니다.

이를 위한 인터페이스는 self.successors.add_successor(state, addr, guard, jumpkind)입니다.

매개변수가 의미가 있어야 합니다. 통과한 상태는 복사되지 않으며 많은 작업을 할 경우 미리 사본을 만들어야 합니다.

SimProcedure 연속

이전 함수를 호출하고 SimProcedure가 실행을 다시 하려면 self.call(addr, args, continue_at)를 사용하면 됩니다. addr은 호출하고자하는 주소이며, args는 인자값의 튜플입니다. continue_at은 다른 함수의 SimProcedure 클래스의 이름입니다. 호출 규약으로 cc를 전달하여 호출 상대와 통신해야 합니다.

이렇게 하면 현재 단계에서 종료하고 다음 단계에서 다시 시작합니다. 함수가 복귀되면 특정 주소로 돌아가야 합니다. 이 주소는 SImProcedure 런타임에 의해 지정됩니다. 주소는 지정된 함수 호출로 돌아가기 위해 angr의 externs 세그먼트에 할당 합니다. 그 후, run() 대신 지정된 continue_at 함수를 실행하기 위해 복사가 완료된 인스턴스에 후킹됩니다.

서브 시스템을 사용하려면 SimProcedure 클래스에 연결해야 하는 2개의 메타데이터가 있습니다.

- 클래스 변수 IS_FUNCTION = True로 설정합니다.
- 클래스 변수 local_vars를 문자열 튜플로 설정합니다. 각 문자열은 반환 된 경우 저장할 값을 갖는 SimProcedure에서 인스턴스 변수의 이름입니다. 지역 변수는 인스턴스를 변경하지 않는 한 어떠한 형태라도 상관없습니다.

데이터를 저장하기 위해 어떤 종류의 보조 기억 장치가 있는지 짐작하고 있을 겁니다. 상태 플러그인 state.callstack은 현재의 호출 프레임에 위치 정보를 저장하는데 SimProcedure 런타임에 의해 사용되는 .procedure_data라는 항목이 있습니다. angr는 state.callstack의 로컬 데이터 저장소를 위해 스택 포인터를 추적합니다. 스택 프레임의 메모리에 저장해야 하는 것이지만 데이터 직렬화 및 메모리 할당이 어렵습니다.

예로 angr가 Linux 프로그램 full_init_state를 위해 모든 공유 라이브러리 초기화를 실행하기 위해 내부적으로 사용하는 SimProcedure를 보도록 하겠습니다.

```
class LinuxLoader(angr.SimProcedure):
    NO_RET = True
    IS_FUNCTION = True
    local_vars = ('initializers',)

    def run(self):
        self.initializers =
self.project.loader.initializers
        self.run_initializer()

    def run_initializer(self):
        if len(self.initializers) == 0:

self.project._simos.set_entry_register_values(self.st
ate)
            self.jump(self.project.entry)
        else:
            addr = self.initializers[0]
            self.initializers = self.initializers[1:]
            self.call(addr, (self.state.posix.argc,
self.state.posix.argv, self.state.posix.environ),
'run_initializer')
```

이것은 SimProcedure 연속의 사용법입니다. 먼저 프로젝트가 프로시저 인스턴스에서 사용가능한지 살펴봐야 합니다. 안전성을 위해서 일반적으로 프로젝트를 읽기 전용 또는 추가 전용 데이터 구조로 사용합니다. 여기에서 로더에서 동적 이니셜 라이저 목록을

검색하면 됩니다. 목록이 비어있지 않으면 목록에서 하나의 함수 포인터를 가져옵니다. 그리고 그것을 호출하고 run)initializer 함수로 돌아갑니다.

전역 변수

간단히 말하면 전역 변수를 state.globals에 저장할 수 있습니다. SimProcedure 연속의 로컬 변수와 동일한 규칙이 적용됩니다. 정확히 모른다면 전역 변수로 사용되는 항목을 변경하지 않도록 해야합니다.

정적 분석 도움

이미 클래스 변수 IS_FUNCTION을 봤습니다. SimProcedure 연속을 사용할 수 있습니다. 설정 가능한 클래스 변수가 있지만 이 함수는 직접적인 것은 없습니다. 함수의 속성을 표시하고 정적 분석이 무엇을 하는지만 알 수 있습니다.

- NO_RET : 제어 흐름이 함수에서 돌아오지 않는 경우 이를 true로 설정합니다.
- ADDS_EXITS : 반환 이외의 제어 흐름을 실행하려면 true로 설정합니다.
- IS_SYSCALL : syscall인지 확인합니다.

또한 ADDS_EXITS를 설정한 경우 static_exits() 함수를 정의 할 수 있습니다. 이 함수는 실행시 IRSB 목록인 단일 매개 변수를 취하고 그 경우 함수가 생성하는 것으로 보입니다. 반환 값은 (address(int) jumpkind(str)) 튜플로 예상 됩니다.

유저 후킹

SimProcedure를 설명하고 사용하는 과정은 함수 전체에 연결하는 것을 전제로 하고 있습니다. 코드 섹션을 연결하는 프로세스를 간소화하기 유저 후킹이 있습니다.

```
>>> @project.hook(0x1234, length=5)
... def set_rax(state):
...     state.regs.rax = 1
```

SimProcedure의 서브 클래스 전체가 아닌 하나의 함수를 사용하는 것입니다. 인자 값을 가져오는 것은 실행되지 않고 복잡한 제어 흐름은 발생하지 않습니다.

제어 흐름은 length 인자 값에서 제어됩니다. 이 예제에서는 함수의 실행이 완료되면 후킹된 주소에 다섯 바이트 후에 다음 단계가 시작됩니다. length 인자가 생략되어 있거나 0으로 설정된 경우, 후킹을 다시 하지 않고 바이너리 코드의 실행이 후킹된 주소에서 다시 시작합니다. Ijk_NoHook 점프는 이것을 가능하게 합니다.

만약 유저 후킹에서 나오는 제어 흐름을 효율적으로 관리하고 싶은 경우 다음 상태의 목록을 반환할 수 있습니다. 다음 상태는 state.regs.ip, state.scratch.guard, state.scratch.jumpkindset이 필요합니다.

심볼 후킹

바이너리 로드를 다시 생각해 봅시다. 동적 링킹된 프로그램은 종송성으로 라이브러리에서 가져와야 하는 목록을 갖고 있습니다. project.hook_symbol API를 사용해 주소 연결을 할 수 있습니다.

```
>>> class NotVeryRand(SimProcedure):
...     def run(self, return_values=None):
...         if 'rand_idx' in self.state.globals:
...             rand_idx =
self.state.globals['rand_idx']
...         else:
...             rand_idx = 0
...
...         out = return_values[rand_idx %
len(return_values)]
...         self.state.globals['rand_idx'] = rand_idx
+ 1
...         return out

>>> project.hook_symbol('rand',
NotVeryRand(return_values=[413, 612, 1025, 1111]))
```

Analyses 작성하기

Analyses는 angr.Analysis 클래스를 상속하여 만들 수 있습니다. 이 장에서는 다양한 기능을 타나내는 analyses를 만듭니다.

```
>>> import angr

>>> class MockAnalysis(angr.Analysis):
...     def __init__(self, option):
...         self.option = option

>>> angr.register_analysis(MockAnalysis,
'MockAnalysis')
```

새로운 analysis를 보도록 합시다.

```
>>> proj = angr.Project("/bin/true")
>>> mock = proj.analyses.MockAnalysis('this is my
option')
>>> assert mock.option == 'this is my option'
```

프로젝트를 가져온 후 새로운 analysis를 등록한 경우 proj.analyses.reload_analyses()를 이용하여 프로젝트의 등록 된 분석의 목록을 업데이트 해야 합니다.

project와 작업

Analysis는 자동으로 self.project 속성 아래에서 실행하는 프로젝트를 가지고 있습니다. 이를 이용하여 프로젝트와 상화 작용하고 분석합니다.

```
>>> class ProjectSummary(angr.Analysis):
...     def __init__(self):
```

```
...         self.result = 'This project is a %s
binary with an entry point at %#x.' %
(self.project.arch.name, self.project.entry)

>>> angr.register_analysis(ProjectSummary,
'ProjectSummary')
>>> proj = angr.Project("/bin/true")

>>> summary = proj.analyses.ProjectSummary()
>>> print summary.result
This project is a AMD64 binary with an entry point at
0x401410.
```

Naming Analyses

register_analysis 호출은 셀제로 angr에 분석을 추가하는 것입니다. 이름은 project.analyses 객체 아래에 어떻게 나타내는 것인지 입니다. 일반적으로 분석 클래스와 동일한 이름을 사용해야 하지만, 짧은 이름을 사용하는 경우도 가능합니다.

```
>>> class FunctionBlockAverage(angr.Analysis):
...     def __init__(self):
...         self._cfg = self.project.analyses.CFG()
...         self.avg = len(self._cfg.nodes()) /
len(self._cfg.function_manager.functions)

>>> angr.register_analysis(FunctionBlockAverage,
'FuncSize')
```

그런 다음 특정 이름을 사용하여 분석을 호출 할 수 있습니다. b.analyses.FuncSize()가 있습니다.

분석 복원

때때로 코드가 예외가 발생할 수 있습니다.

Analysis 기본 클래스는 self.resilience에서 복원 context manager 를 제공합니다.

```
>>> class ComplexFunctionAnalysis(angr.Analysis):
...     def __init__(self):
...         self._cfg = self.project.analyses.CFG()
...         self.results = { }
...         for addr, func in
self._cfg.function_manager.functions.iteritems():
...             with self._resilience():
...                 if addr % 2 == 0:
...                     raise ValueError("can't
handle functions at even addresses")
...                 else:
...                     self.results[addr] = "GOOD"
```

context manager는 throw된 예외를 잡아 유형, 메시지 및 추적 튜플로 self.errors에 기혹합니다. (단, 추적은 삭제되지 않습니다.)

예제

angr를 이용해 실제 해킹대회 및 바이너리를 분석할 때 어떻게 사용되는지 알아보겠습니다.

```c
#include <stdio.h>

int main(void)
{
        int num = 0;
        printf("Input : ");
        scanf("%d", &num);

        if(num == 12)
        {
                printf("Ok!\n");
        }
        else
        {
                printf("No");
        }

        return 0;
}
```

위 c코드를 살펴보면 Ok!가 출력되려면 num 값이 12가 되야 하는 것을 쉽게 알 수 있습니다. 하지만 코드는 주어지지 않고 바이너리만 제공되는 경우 어떻게 분석을 해야 할까요?

실제 해당 코드를 컴파일 하고 gdb를 이용해 어셈블리어를 살펴보겠습니다.

```
pwndbg> b*main
Breakpoint 1 at 0x400646
pwndbg> r
```

```
Starting program: /home/k3y6reak/Desktop/test

Breakpoint 1, 0x0000000000400646 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[──────────────────────────────────────────────────────────────
──────────────────────REGISTERS──────────────────────────────
──────────────────────────────────────────────────────────────]
*RAX  0x400646 (main) ◂— push   rbp
 RBX  0x0
 RCX  0x0
*RDX  0x7fffffffdde8 —▸ 0x7fffffffe18d ◂—
'XDG_VTNR=7'
*RDI  0x1
*RSI  0x7fffffffddd8 —▸ 0x7fffffffe171 ◂—
0x336b2f656d6f682f ('/home/k3')
*R8   0x400740 (__libc_csu_fini) ◂— ret
*R9   0x7ffff7de7ab0 (_dl_fini) ◂— push   rbp
*R10  0x846
*R11  0x7ffff7a2d740 (__libc_start_main) ◂— push
r14
*R12  0x400550 (_start) ◂— xor    ebp, ebp
*R13  0x7fffffffddd0 ◂— 0x1
 R14  0x0
 R15  0x0
*RBP  0x4006d0 (__libc_csu_init) ◂— push   r15
*RSP  0x7fffffffdcf8 —▸ 0x7ffff7a2d830
(__libc_start_main+240) ◂— mov    edi, eax
*RIP  0x400646 (main) ◂— push   rbp
[──────────────────────────────────────────────────────────────
──────────────────────DISASM──────────────────────────────
──────────────────────────────────────────────────────────────]
 ► 0x400646 <main>       push   rbp
<0x4006d0>
   0x400647 <main+1>     mov    rbp, rsp
   0x40064a <main+4>     sub    rsp, 0x10
   0x40064e <main+8>     mov    rax, qword ptr fs:
[0x28]
   0x400657 <main+17>    mov    qword ptr [rbp - 8],
rax
```

```
   0x40065b <main+21>    xor    eax, eax
   0x40065d <main+23>    mov    dword ptr [rbp -
0xc], 0
   0x400664 <main+30>    mov    edi, 0x400754
   0x400669 <main+35>    mov    eax, 0
   0x40066e <main+40>    call   printf@plt
<0x400510>

   0x400673 <main+45>    lea    rax, [rbp - 0xc]
[──────────────────────────────────────────────────
──────────────────────────STACK──────────────────────
──────────────────────────────────────────────────]
00:0000│ rsp  0x7fffffffdcf8 —▸ 0x7ffff7a2d830
(__libc_start_main+240) ◂— mov    edi, eax
01:0008│      0x7fffffffdd00 ◂— 0x0
02:0010│      0x7fffffffdd08 —▸ 0x7fffffffddd8 —▸
0x7fffffffe171 ◂— 0x336b2f656d6f682f ('/home/k3')
03:0018│      0x7fffffffdd10 ◂— 0x100000000
04:0020│      0x7fffffffdd18 —▸ 0x400646 (main) ◂—
push   rbp
05:0028│      0x7fffffffdd20 ◂— 0x0
06:0030│      0x7fffffffdd28 ◂— 0x7ce28d65fee07d37
07:0038│      0x7fffffffdd30 —▸ 0x400550 (_start) ◂—
xor    ebp, ebp
[──────────────────────────────────────────────────
──────────────────────BACKTRACE─────────────────────
──────────────────────────────────────────────────]
 ► f 0           400646 main
   f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint *main
```

```
pwndbg> disassemble main
Dump of assembler code for function main:
=> 0x0000000000400646 <+0>:     push   rbp
   0x0000000000400647 <+1>:     mov    rbp,rsp
   0x000000000040064a <+4>:     sub    rsp,0x10
```

```
   0x000000000040064e <+8>:     mov    rax,QWORD PTR
fs:0x28
   0x0000000000400657 <+17>:    mov    QWORD PTR
[rbp-0x8],rax
   0x000000000040065b <+21>:    xor    eax,eax
   0x000000000040065d <+23>:    mov    DWORD PTR
[rbp-0xc],0x0
   0x0000000000400664 <+30>:    mov    edi,0x400754
   0x0000000000400669 <+35>:    mov    eax,0x0
   0x000000000040066e <+40>:    call   0x400510
<printf@plt>
   0x0000000000400673 <+45>:    lea    rax,[rbp-0xc]
   0x0000000000400677 <+49>:    mov    rsi,rax
   0x000000000040067a <+52>:    mov    edi,0x40075d
   0x000000000040067f <+57>:    mov    eax,0x0
   0x0000000000400684 <+62>:    call   0x400530
<__isoc99_scanf@plt>
   0x0000000000400689 <+67>:    mov    eax,DWORD PTR
[rbp-0xc]
   0x000000000040068c <+70>:    cmp    eax,0xc
   0x000000000040068f <+73>:    jne    0x40069d
<main+87>
   0x0000000000400691 <+75>:    mov    edi,0x400760
   0x0000000000400696 <+80>:    call   0x4004f0
<puts@plt>
   0x000000000040069b <+85>:    jmp    0x4006a7
<main+97>
   0x000000000040069d <+87>:    mov    edi,0x400764
   0x00000000004006a2 <+92>:    call   0x4004f0
<puts@plt>
   0x00000000004006a7 <+97>:    mov    eax,0x0
   0x00000000004006ac <+102>:   mov    rdx,QWORD PTR
[rbp-0x8]
   0x00000000004006b0 <+106>:   xor    rdx,QWORD PTR
fs:0x28
   0x00000000004006b9 <+115>:   je     0x4006c0
<main+122>
   0x00000000004006bb <+117>:   call   0x400500
<__stack_chk_fail@plt>
```

```
   0x00000000004006c0 <+122>:   leave
   0x00000000004006c1 <+123>:   ret
End of assembler dump.
```

위 어셈을 확인해 보면 0x000000000040068c <+70>: cmp eax,0xc에서 0xc와 비교하고 있습니다. 해당 위치에서 조건을 확인한 다는 것을 할 수 있고 이를 직접 디버거로 살펴봐야 합니다. 만약 num 값을 비교하기 전 사람이 쉽게 풀지 못하는 수학적 연산이 존재한다면 어떻게 할까요?

angr를 이용하면 하나씩 어셈블리어를 살펴보지 않아도 찾아가야할 주소와 찾아가지 말아야 할 주소만을 이용해 값을 쉽게 찾을 수 있습니다.

위 예제에서는 0x400691로 이동해야 "Ok"가 출력되고 0x40069d로 이동하면 "No"가 출력되는 것을 알 수 있습니다.

```
import angr

def main():
        proj = angr.Project('./test', load_options=
{'auto_load_libs': False})
        path_group =
proj.factory.path_group(threads=4)
        path_group.explore(find=0x40096b,
avoid=0x40069d)
        return
path_group.found[0].state.posix.dumps(1)

if __name__ == '__main__':
        print repr(main())
```

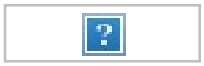
위와 같이 angr를 import 하고 find에 0x400691을 넣고 avoid에 0x40069d를 넣고 실행합니다.

```
root@ubuntu:/home/k3y6reak/Desktop# python crack.py
WARNING | 2017-12-08 20:00:44,866 |
simuvex.plugins.symbolic_memory | Concretizing
symbolic length. Much sad; think about implementing.
'Input : Ok!\n'
```

단순히 python 코드만을 이용해 찾아갈 주소와 피해야할 주소만을 이용하여 ok가 출력된 것을 볼 수 있다. 이렇게 특정 입력값을 직접 분석하지 않고 찾아갈 수 있다.

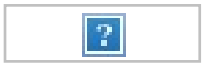
또 다른 예로는 실제 해킹 대회 DEFCON에서 출제된 baby-re 라는 문제를 풀어보겠다.

baby-re는 IDA로 디컴파일한 결과를 보면 아래와 같다.



0부터 12까지 총 13개의 값을 입력하고 CeckSolution 값이 true가 돼야 flag를 출력해준다.

구조를 살펴보면 아래와 같다.



flag를 출력해 주는 부분과 wrong을 출력해 주는 부분이 있는 것을 알 수 있다. find는 0x40294b avoid는 0x402941로 하면 된다.

```
import angr

def main():
        proj = angr.Project('./test', load_options=
{'auto_load_libs': False})
        path_group =
proj.factory.path_group(threads=4)
        path_group.explore(find=0x40294b,
avoid=0x402941)
        return
```

```
  path_group.found[0].state.posix.dumps(1)

  if __name__ == '__main__':
          print repr(main())
```

위 python 코드를 작성하여 실행하면 아래와 같이 Math is hard!가 출력된다.

6_Appendix

List of Claripy Operations

Arithmetic and Logic

이름	설명	예시
LShR	오른쪽 논리 쉬프트 연산	x.LShR(10)
RotateLeft	표현식을 좌측으로 회전	x.RotateLeft(8)
RotateRight	표현식을 우측으로 회전	x.RotateRight(8)
And	논리 and	solver.And(x == y, x > 0)
Or	논리 Or	solver.Or(x == y, y < 10)
Not	논리 Not	solver.Not(x == y)는 x != y와 동일
If	if-then-else	최대 값을 선택 : solver.If(x > y, x, y)
ULE	Unsigned, 작거나 같을 때	x.ULE(y)
ULT	Unsigned, 작을 때	x.ULT(y)
UGE	Unsigned, 크거나 같을 때	x.UGE(y)
UGT	Unsigned, 클 때	x.UGT(y)
SLE	Signed, 작거나 같을 때	x.SLE(y)
SLT	Signed, 작을 때	x.SLT(y)
SGE	Signed, 크거나 같을 때	x.SGE(y)

SGT	Signed, 클 때	x.SGT(y)

Bitvector Manipulation

이름	설명	예시
SignExt	n개의 부호 비트가 있는 왼쪽에 bitvector 채움	x.sign_extend(n)
ZeroExt	n개의 zero 비트가 있는 왼쪽에 bitvector 채움	x.zero_extend(n)
Extract	표현식에서 주어진 비트(우측부터 포함하여 에서 zero-indexed)를 추출	x의 LSB 추출 : x[7:0]
Concat	몇 개의 표현식을 합치거나 새로운 표현식을 만듬	x.Concat(y, ...)

Extra Functionality

AST를 분석하고 연산 집합을 구성하여 구현할 수 있는 미리 패키징된 작업들이 있지만, 더 쉬운 방식이 아래 있습니다.

- bitvector를 val.chop(n)을 이용하여 n 비트 덩어리 목록으로 잘라낼 수 있습니다.
- bitvector를 x.reversed로 endian-reverse 할 수 있습니다.
- val.length를 사용하여 bitvector의 너비를 비트 단위로 가져올 수 있습니다.
- AST에 val.symbolic이 있는 symbolic component가 있는지 테스트할 수 있습니다.
- val.variables를 사용하여 AST를 작성하는데 관련된 모든 symbolic variable의 이름 집합을 가져올 수 있습니다.

6_Appendix

List of State Options

State Modes

이것들은 mode = xxx를 상태 생성자에 건네주면서 가능합니다.

모드 이름	설명
symbolic	기본 모드. 대부분의 에뮬레이션 및 분석 작업에 유용
symbolic_approximating	symbolic 모드. 제약 조건 해결에 대한 근사값을 사용할 수 있음
static	정적 분석에 유용한 설정. 메모리 모델은 추상적인 영역 매핑 시스템이 되고, 호출을 건너 뛰는 "후위 반환" successor가 추가됨
fastpath	매우 가벼운 정적 분석을 위한 설정. 실행은 코드의 동작을 빠르게 살펴볼 수 있도록 모든 집중적인 처리를 건너뜀
tracing	주어진 입력을 가진 프로그램을 통해 구체적으로 실행하기 위한 설정. unicorn을 활성화하고 resilience 옵션을 활성화하며 access violation을 올바르게 에뮬레이트 하려고 시도

Option Sets

이들은 angr.option.xxx와 같은 옵션 세트입니다.

세트 이름	설명
common_options	기본 실행에 필요한 옵션

symbolic	기본 symbolic 실행에 필요한 옵션
resilience	지원되지 않은 작업에 대한 angr의 에뮬레이션을 강화하고, 결과를 unconstrained symbolic value로 처리하고, state.history.events에 상황을 기록하여 계속 수행하려고 시도하는 옵션
refs	angr이 history.actions의 종속성 정보로 완료되는 모든 메모리, 레지스터 및 임시 참조의 로그를 유지하도록 하는 옵션. 메모리 소비 큼
approximation	z3를 호출하는 대신 value-set 분석을 통해 제약 조건의 근사치를 해결할 수 있는 옵션
simplification	메모리 또는 레지스터 저장 공간에 도달하기 전에 z3의 단순화기를 통해 데이터를 실행하는 옵션
unicorn	concrete 데이터에서 실행되도록 unicorn 엔진을 활성화하는 옵션

Options

이들은 개별 옵션 개체로 angr.options.XXX로 사용합니다.

옵션 이름	설명	세트
ABSTRACT_MEMORY	SimAbstractMemory를 사용하여 메모리를 개별 영역으로 모델링	
ABSTRACT_SOLVER	단순화 도중 제약 조건 세트의 분할을 허용	
ACTION_DEPS	SimActions의 종속적 추적	
APPROXIMATE_GUARDS	방어 조건을 평가할 때 VSA 사용	

APPROXIMATE_MEMORY_INDICES	메모리 indice를 평가할 때 VSA 사용	approximatior
APPROXIMATE_MEMORY_SIZES	메모리 load/store 크기를 평가할 때 VSA 사용	approximatior
APPROXIMATE_SATISFIABILITY	상태 안정성을 평가하기 위해 VSA 사용	approximatior
AST_DEPS	모든 명확한 AST에 대한 종속성 추적을 가능하게 함	
AUTO_REFS	SimProcedures에서 종속성을 추적하는데 사용되는 내부 옵션	
AVOID_MULTIVALUED_READS	기호화된 주소를 가진 읽기에 대해 메모리를 건드리지 않고 기호 값을 반환	
AVOID_MULTIVALUED_WRITES	기호화된 주소를 가진 쓰기를 수행하지 않음	
BEST_EFFORT_MEMORY_STORING	실제로 작아보이지만 큰 심볼 사이즈의 쓰기를 다룸	
BLOCK_SCOPE_CONSTRAINTS	각 블록 끝에 있는 제약 리스트 제거	
BREAK_SIRSB_END	디버그 : 각 블록의 끝에 breakpoint 트리거	
BREAK_SIRSB_START	디버그 : 각 블록 시작에 breakpoint 트리거	
BREAK_SIRSTMT_END	디버그 : 각 IR문의 끝에 breakpoint 트리거	

BREAK_SIRSTMT_START	디버그 : 각 IR문의 시작에 breakpoint 트리거	
BYPASS_ERRORED_IRCCALL	unconstrained symbolic value를 반환해서 실패한 helper 처리	resilience
BYPASS_ERRORED_IROP	unconstrained symbolic value를 반환해서 실패한 작업 처리	resilience
BYPASS_UNSUPPORTED_IRCCALL	unconstrained symbolic value를 반환해서 지원되지 않은 helper 처리	resilience
BYPASS_UNSUPPORTED_IRDIRTY	unconstrained symbolic value를 반환해서 지원되지 않은 더티 helper 처리	resilience
BYPASS_UNSUPPORTED_IREXPR	unconstrained symbolic value를 반환해서 지원되지 않은 IR식 처리	resilience
BYPASS_UNSUPPORTED_IROP	unconstrained symbolic value를 반환해서 지원되지 않은 연산 처리	resilience
BYPASS_UNSUPPORTED_IRSTMT	unconstrained symbolic value를 반환해서 지원되지 않은 IR식 처리	resilience

BYPASS_UNSUPPORTED_SYSCALL	unconstrained symbolic value를 반환해서 지원되지 않는 syscall 처리	resilience