Table of Contents

How to be angry

This is a collection of documentation for angr. By reading this, you'll become an angr pro and will be able to fold binaries to your whim.

We've tried to make using angr as pain-free as possible - our goal is to create a user-friendly binary analysis suite, allowing a user to simply start up iPython and easily perform intensive binary analyses with a couple of commands. That being said, binary analysis is complex, which makes angr complex. This documentation is an attempt to help out with that, providing narritive explanation and exploration of angr and its design.

Get Started

Installation instructions can be found here.

To dive right into angr's capabilities, start with the top level methods, or read over the overview.

A searchable HTML version of this documentation is hosted at docs.angr.io, and an HTML API reference can be found at angr.io/api-doc.

Citing angr

If you use angr in an academic work, please cite the papers for which it was developed:

```
@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, Ni
ck and Polino, Mario and Dutcher, Andrew and Grosen, John and Feng, Siji and Hauser, C
hristophe 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, Andrew a
nd 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 B
inary Firmware},
  author={Shoshitaishvili, Yan and Wang, Ruoyu and Hauser, Christophe and Kruegel, Chr
istopher and Vigna, Giovanni},
  booktitle={NDSS},
  year={2015}
}
```

Support

To get help with angr, you can ask via:

- the mailing list: angr@lists.cs.ucsb.edu
- the IRC channel: **#angr** on freenode
- opening an issue on the appropriate github repository

Going further:

You can read this paper, explaining some of the internals, algorithms, and used techniques to get a better understanding on what's going on under the hood.

What is angr?

angr is a multi-architecture binary analysis platform, with the capability to perform dynamic symbolic execution (like Mayhem, KLEE, etc.) and various static analyses on binaries. Several challenges must be overcome to do this. They are, roughly:

- Loading a binary into the analysis program.
- Translating a binary into an intermediate representation (IR).
- Translating that IR into a semantic representation (i.e., what it *does*, not just what it *is*).
- Performing the actual analysis. This could be:
  - A partial or full-program static analysis (i.e., dependency analysis, program slicing).
  - A symbolic exploration of the program's state space (i.e., "Can we execute it until we find an overflow?").
  - Some combination of the above (i.e., "Let's execute only program slices that lead to a memory write, to find an overflow.")

angr has components that meet all of these challenges. This book will explain how each one works, and how they can all be used to accomplish your evil goals.

Loading a Binary

After angr is installed, you can load a binary for analysis. This process, and the angr component that powers it (called CLE) is described here.

Intermediate Representation

angr uses an intermediate representation (specifically, VEX) to enable it to run analyses on binaries of different architectures. This IR is described here.

Solver Engine

Constraint solving and other computational needs are provided by an angr sub-module called Claripy. Most users of angr will not need to know anything about Claripy, but documentation is provided in case it is needed. Claripy is detailed here.

Program States

angr provides an interface to the emulated machine states. Understanding this is critical to successfully using angr. It is detailed here.

Program Paths

Programs can be analyzed in terms of the possible *path* that execution takes through them. angr exposes information about what the paths execute and *do*. This section gives an overview of how to use this capability of angr.

Semantic Representation

A powerful feature of angr is the ability to represent basic blocks in terms of their effects on a program state. In other words, angr can reason about what basic blocks *do*, not just what they *are*. This is accomplished by a module named SimuVEX, further described here.

Symbolic Execution

angr provides a capable symbolic execution engine. The interface to this engine, and how to use it, is described here.

Full-program Analysis

All of the above components come together to enable complex, full-program analyses to be easily run with angr. The mechanism for running and writing these analyses is detailed here.

Examples

We've written some examples for using angr! You can read more here.

Installing angr

angr is a python library, so it must be installed into your python environment before it can be used. It is built for Python 2: Py3k support is feasible somewhere out in the future, but we are a little hesitant to make that commitment right now (pull requests welcome!).

We highly recommend using a python virtual environment to install and use angr. Several of angr's dependencies (z3, pyvex) require libraries of native code that are forked from their originals, and if you already have libz3 or libVEX installed, you definitely don't want to overwrite the official shared objects with ours. In general, don't expect support for problems arising from installing angr outside of a virtualenv.

Dependencies

All of the python dependencies should be handled by pip and/or the setup.py scripts. You will, however, need to build some C to get from here to the end, so you'll need a good build environment as well as the python development headers. At some point in the dependency install process, you'll install the python library cffi, but (on linux, at least) it won't run unless you install your operating system's libffi package.

On Ubuntu, you will want: `sudo apt-get install python-dev libffi-dev build-essential virtualenvwrapper`. If you are trying out angr-management, you will need `sudo apt-get install libqt4-dev graphviz-dev`.

Most Operating systems, all *nix systems

`mkvirtualenv angr && pip install angr` should usually be sufficient to install angr in most cases, since angr is published on the Python Package Index.

Fish (shell) users can either use virtualfish or the virtualenv package.
`vf new angr && vf activate angr && pip install angr`

Failing that, you can install angr by installing the following repositories (and the dependencies listed in their requirements.txt files), in order, from https://github.com/angr:

- claripy
- archinfo
- pyvex
- cle

- simuvex
- angr

Mac OS X

Before you say `pip install angr`, you need to rebuild our fork of z3 with `pip install -I --no-use-wheel angr-only-z3-custom`.

Windows

You cannot install angr from pip on windows. You must install all of its components individually.

Capstone is difficult to install on windows. You might need to manually specify a wheel to install, but sometimes it installs under a name different from "capstone", so if that happens you want to just remove capstone from the requirements.txt files in angr and archinfo.

Z3 might compile on windows if you have a l33t enough build environment. If this isn't the case for you, you should download a wheel from somewhere on the Internet. One location for pre-built Windows wheel files is https://github.com/Owlz/angr-Windows.

If you build z3 from source, make sure you're using the unstable branch of z3, which includes floating point support. In addition, make sure to have `Z3PATH=path/to/libz3.dll` in your environment.

Development install

We created a repo with scripts to make life easier for angr developers. You can set up angr in development mode by doing:

```
git clone https://github.com/angr/angr-dev
cd angr-dev
mkvirtualenv angr
./setup.sh
```

This clones all of the repositories and installs them in editable mode. `setup.sh` can even create a PyPy virtualenv for you, resulting in significantly faster performance and lower memory usage.

You can branch/edit/recompile the various modules in-place, and it will automatically reflect in your virtual environment.

Docker install

For convenience, we ship a Docker image that is 99% guaranteed to work. You can install via docker by doing:

```
# 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
```

Synchronization of files in and out of docker is left as an exercise to the user (hint: check out `docker -v`).

Troubleshooting

libgomp.so.1: version GOMP_4.0 not found

This error represents an incompatibility between the pre-compiled version of `angr-only-z3-custom` and the installed version of `libgomp`. A Z3 recompile is required. You can do this by executing:

```
pip install -I --no-use-wheel angr-only-z3-custom
```

Can't import mulpyplexer

There are sometimes issues with installing mulpyplexer. Doing `pip install --upgrade 'git+https://github.com/zardus/mulpyplexer'` should fix this.

Can't import angr because of capstone

Sometimes capstone isn't installed correctly for use by angr. There's a good chance just reinstalling capstone will solve this issue:

```
pip install -I --no-use-wheel capstone
```

ImportError due to failure in loading capstone while importing angr

There's a known issue in installing capstone_3.0.4 using pip in virtualenv/virtualenvwrapper environment. Several users have further reported to be affected by the same bug in native Python installation, too. (See the discussion in Github bug report).

In virtual environment, if capstone Python files are installed in `/home/<username>/.virtualenvs/<virtualenv>/lib/python2.7/site-packages/capstone/*.py(c)`, capstone library file will be found in `/home/<username>/.virtualenvs/<virtualenv>/lib/python2.7/site-packages/home/<username>/.virtualenvs/<virtualenv>/lib/python2.7/site-packages/capstone/libcapstone.so`

In native environment, if capstone Python files are installed in `/usr/local/lib/python2.7/dist-packages/capstone/*.py(c)`, capstone library file will be found in `/usr/local/lib/python2.7/dist-packages/usr/lib/python2.7/dist-packages/capstone/libcapstone.so`

Moving `libcapstone.so` to the same directory as that of Python files will fix the problem.

Claripy and z3

Z3 is a bit weird to compile. Sometimes it just completely fails to build for no reason, saying that it can't create some object file because some file or directory doesn't exist. Just retry the build:

```
pip install -I --no-use-wheel angr-only-z3-custom
```

No such file or directory: 'pyvex_c'

Are you running 12.04? If so, please upgrade!

You can also try upgrading pip (`pip install -U pip`), which might solve the issue.

Reporting Bugs

If you've found something that angr isn't able to solve and appears to be a bug, please let us know!

1. Create a fork off of angr/binaries and angr/angr
2. Give us a pull request with angr/binaries, with the binaries in question
3. Give us a pull request for angr/angr, with testcases that trigger the binaries in `angr/tests/broken_x.py`, `angr/tests/broken_y.py`, etc

Please try to follow the testcase format that we have (so the code is in a test_blah function), that way we can very easily merge that and make the scripts run. An example is:

```
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()
```

This will *greatly* help us recreate your bug and fix it faster. The ideal situation is that, when the bug is fixed, your testcases passes (i.e., the assert at the end does not raise an AssertionError). Then, we can just fix the bug and rename `broken_x.py` to `test_x.py` and the testcase will run in our internal CI at every push, ensuring that we do not break this feature again.

Developing angr

These are some guidelines so that we can keep the codebase in good shape!

Coding style

We try to get as close as the PEP8 code convention as is reasonable without being dumb. If you use Vim, the python-mode plugin does all you need. You can also manually configure vim to adopt this behavior.

Most importantly, please consider the following when writing code as part of angr:

- Try to use attribute access (see the `@property` decorator) instead of getters and setters wherever you can. This isn't Java, and attributes enable tab completion in iPython. That being said, be reasonable: attributes should be fast. A rule of thumb is that if something could require a constraint solve, it should not be an attribute.
- Use our `.pylintrc`. It's fairly permissive, but our CI server will fail your builds if pylint complains under those settings.
- DO NOT, under ANY circumstances, `raise Exception` or `assert False`. **Use the right exception type**. If there isn't a correct exception type, subclass the core exception of the module that you're working in (i.e., `AngrError` in angr, `SimError` in SimuVEX, etc) and raise that. We catch, and properly handle, the right types of errors in the right places, but `AssertionError` and `Exception` are not handled anywhere and force-terminate analyses.
- Avoid tabs; use space indentation instead. Even though it's wrong, the de facto standard is 4 spaces. It is a good idea to adopt this from the beginning, as merging code that mixes both tab and space indentation is awful.
- Avoid super long lines. It's okay to have longer lines, but keep in mind that long lines are harder to read and should be avoided. Let's try to stick to **120 characters**.
- Avoid extremely long functions, it is often better to break them up into smaller functions.
- Prefer `_` to `__` for private members (so that we can access them when debugging). *You* might not think that anyone has a need to call a given function, but trust us, you're wrong.

Documentation

Document your code. Every *class definition* and *public function definition* should have some description of:

- What it does.
- What are the type and the meaning of the parameters.
- What it returns.

We use Sphinx to generate the API documentation. Sphinx supports special keywords to document function parameters, return values, return types etc.

Here is an example of function documentation. Ideally the parameter descriptions should be aligned vertically to make the docstrings as readable as possible.

```
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
    """
```

This format has the advantage that the function parameters are clearly identified in the generated documentation. However, it can make the documentation repetitive, in some cases a textual description can be more readable. Pick the format you feel is more appropriate for the functions or classes you are documenting.

```
  def read_bytes(self, addr, n):
     """
     Read `n` bytes at address `addr` in memory and return an array of bytes.
     """
```

Unit tests

If you're pushing a new feature and it is not accompanied by a test case it **will be broken** in very short order. Please write test cases for your stuff.

We have an internal CI server to run tests to check functionality and regression on each commit. In order to have our server run your tests, write your tests in a format acceptable to nosetests in a file matching `test_*.py` in the `tests` folder of the appropriate repository. A test file can contain any number of functions of the form `def test_*():`. Each of them will be run as a test, and if they raise any exceptions or assertions, the test fails. Use the `nose.tools.assert_*` functions for better error messages.

Look at the existing tests for examples. Many of them use an alternate format where the `test_*` function is actually a generator that yields tuples of functions to call and their arguments, for easy parametrization of tests. Do not add docstrings to your test functions.

"Help Wanted"

angr is a huge project, and it's hard to keep up. Here, we list some big TODO items that we would love community contributions for in the hope that it can direct community involvement. They (will) have a wide range of complexity, and there should be something for all skill levels!

Documentation

There are many parts of angr that suffer from little or no documentation. We desperately need community help in this area.

API

We are always behind on documentation. We've created several tracking issues on github to understand what's still missing:

1. angr
2. simuvex
3. claripy
4. cle
5. pyvex

GitBook

This book is missing some core areas. Specifically, the following could be improved:

1. Finish some of the TODOs floating around the book.
2. Organize the Examples page in some way that makes sense. Right now, most of the examples are very redunant. It might be cool to have a simple table of most of them so that the page is not so overwhelming.

angr course

Developing a "course" of sorts to get people started with angr would be really beneficial. Steps have already been made in this direction here, but more expansion would be beneficial.

Ideally, the course would have a hands-on component, of increasing difficulty, that would require people to use more and more of angr's capabilities.

Research re-implementation

Unfortunately, not everyone bases their research on angr ;-). Until that's remedied, we'll need to periodically implement related work, on top of angr, to make it reusable within the scope of the framework. This section lists some of this related work that's ripe for reimplementation in angr.

Redundant State Detection for Dynamic Symbolic Execution

Bugrara, et al. describe a method to identify and trim redundant states, increasing the speed of symbolic execution by up to 50 times and coverage by 4%. This would be great to have in angr, as an ExplorationTechnique. The paper is here: http://nsl.cs.columbia.edu/projects/minestrone/papers/atc13-bugrara.pdf

In-Vivo Multi-Path Analysis of Software Systems

Rather than developing symbolic summaries for every system call, we can use a technique proposed by S2E for concretizing necessary data and dispatching them to the OS itself. This would make angr applicable to a *much* larger set of binaries than it can currently analyze.

While this would be most useful for system calls, once it is implemented, it could be trivially applied to any location of code (i.e., library functions). By carefully choosing which library functions are handled like this, we can greatly increase angr's scalability.

Development

We have several projects in mind that primarily require development effort.

angr-management

The angr GUI, angr-management needs a *lot* of work. Here is a non-exhaustive list of what is currently missing in angr-management:

- A navigator toolbar showing content in a program’s memory space, just like IDA Pro’s navigator toolbar.
- A text-based disassembly view of the program.

- Better view showing details in program states during path exploration, including modifiable register view, memory view, file descriptor view, etc.
- A GUI for cross referencing.

Exposing angr's capabilities in a usable way, graphically, would be really useful!

IDA Plugins

Much of angr's functionality could be exposed via IDA. For example, angr's data dependence graph could be exposed in IDA through annotations, or obfuscated values can be resolved using symbolic execution.

additional architectures

More architecture support would make angr all the more useful. Supporting a new architecture with angr would involve:

1. Adding the architecture information to archinfo
2. Adding an IR translation. This may be either an extension to PyVEX, producing IRSBs, or another IR entirely.
3. If your IR is not VEX, add a `simuvex.SimEngine` to support it.
4. Adding a calling convention (`simuvex.SimCC`) to support SimProcedures (including system calls)
5. Adding or modifying an `angr.SimOS` to support initialization activities.
6. Creating a CLE backend to load binaries, or extending the CLE ELF backend to know about the new architecture if the binary format is ELF.

ideas for new architectures:

- PIC, AVR, other embedded architectures
- SPARC (there is some preliminary libVEX support for SPARC here)

ideas for new IRs:

- LLVM IR (with this, we can extend angr from just a Binary Analysis Framework to a Program Analysis Framework and expand its capabilities in other ways!)
- SOOT (there is no reason that angr can't analyze Java code, although doing so would require some extensions to our memory model)

environment support

We use the concept of "function summaries" in angr to model the environment of operating systems (i.e., the effects of their system calls) and library functions. Extending this would be greatly helpful in increasing angr's utility. These function summaries can be found here.

A specific subset of this is system calls. Even more than library function SimProcedures (without which angr can always execute the actual function), we have very few workarounds for missing system calls. Every implemented system call extends the set of binaries that angr can handle!

Design Problems

There are some outstanding design challenges regarding the integration of additional functionalities into angr.

type annotation and type information usage

angr has fledgling support for types, in the sense that it can parse them out of header files. However, those types are not well exposed to do anything useful with. Improving this support would make it possible to, for example, annotate certain memory regions with certain type information and interact with them intelligently.

Consider, for example, interacting with a linked list like this: `print state.memory[state.regs.rax:].next.next.value`.

Research Challenges

Historically, angr has progressed in the course of research into novel areas of program analysis. Here, we list several self-contained research projects that can be tackled.

semantic function identification/diffing

Current function diffing techniques (TODO: some examples) have drawbacks. For the CGC, we created a semantic-based binary identification engine (https://github.com/angr/identifier) that can identify functions based on testcases. There are two areas of improvement, each of which is its own research project:

1. Currently, the testcases used by this component are human-generated. However, symbolic execution can be used to automatically generate testcases that can be used to recognize instances of a given function in other binaries.
2. By creating testcases that achieve a "high-enough" code coverage of a given function,

we can detect changes in functionality by applying the set of testcases to another implementation of the same function and analyzing changes in code coverage. This can then be used as a sematic function diff.

applying AFL's path selection criteria to symbolic execution

AFL does an excellent job in identifying "unique" paths during fuzzing by tracking the control flow transitions taken by every path. This same metric can be applied to symbolic exploration, and would probably do a depressingly good job, considering how simple it is.

Overarching Research Directions

There are areas of program analysis that are not well explored. We list general directions of research here, but readers should keep in mind that these directions likely describe potential undertakings of entire PhD dissertations.

process interactions

Almost all work in the field of binary analysis deals with single binaries, but this is often unrealistic in the real world. For example, the type of input that can be passed to a CGI program depend on pre-processing by a web server. Currently, there is no way to support the analysis of multiple concurrent processes in angr, and many open questions in the field (i.e., how to model concurrent actions).

intra-process concurrency

Similar to the modeling of interactions between processes, little work has been done in understanding the interaction of concurrent threads in the same process. Currently, angr has no way to reason about this, and it is unclear from the theoretical perspective how to approach this.

A subset of this problem is the analysis of signal handlers (or hardware interrupts). Each signal handler can be modeled as a thread that can be executed at any time that a signal can be triggered. Understanding when it is meaningful to analyze these handlers is an open problem. One system that does reason about the effect of interrupts is FIE.

path explosion

Many approaches (such as Veritesting) attempt to mitigate the path explosion problem in symbolic execution. However, despite these efforts, path explosion is still *the* main problem preventing symbolic execution from being mainstream.

angr provides an excellent base to implement new techniques to control path explosion. Most approaches can be easily implemented as Exploration Techniques and quickly evaluated (for example, on the CGC dataset.

Top-level interfaces

So you've loaded a project. Now what?

This document explains all the attributes that are available directly from instances of `angr.Project`.

Basic properties

```
>>> import angr, monkeyhex, claripy
>>> b = angr.Project('/bin/true')

>>> b.arch
<Arch AMD64 (LE)>
>>> b.entry
0x401410
>>> b.filename
'/bin/true'
>>> b.loader
<Loaded true, maps [0x400000:0x4004000]>
```

- *arch* is an instance of an `archinfo.Arch` object for whichever architecture the program is compiled. There's lots of fun information on there! The common ones you care about are `arch.bits`, `arch.bytes` (that one is a `@property` declaration on the main `Arch` class), `arch.name`, and `arch.memory_endness`.
- *entry* is the entry point of the binary!
- *filename* is the absolute filename of the binary. Riveting stuff!
- *loader* is the cle.Loader instance for this project. Details on how to use it are found here.

Analyses and Surveyors

```
>>> b.analyses
<angr.analysis.Analyses object at 0x7f5220d6a890>
>>> b.surveyors
<angr.surveyor.Surveyors object at 0x7f52191b9dd0>

>>> filter(lambda x: '_' not in x, dir(b.analyses))
['BackwardSlice',
 'BinDiff',
 'BoyScout',
 'BufferOverflowDetection',
 'CDG',
 'CFG',
 'DDG',
 'GirlScout',
 'SleakMeta',
 'Sleakslice',
 'VFG',
 'Veritesting',
 'XSleak']
>>> filter(lambda x: '_' not in x, dir(b.surveyors))
['Caller', 'Escaper', 'Executor', 'Explorer', 'Slicecutor', 'started']
```

`analyses` and `surveyors` are both just container objects for all the Analyses and Surveyors, respectively.

Analyses are customizable analysis routines that can extract some sort of information from the program. The most common two are `CFG`, which constructs a control-flow graph, and `VFG`, which performs value-set analysis. Their use, as well as how to write your own analyses, is documented here.

Surveyors are basic tools for performing symbolic execution with common goals. The most common one is `Explorer`, which searches for a target address while avoiding some others. Read about using surveyors here. Note that while surveyors are cool, an alternative to them is Path Groups (below), which are the future.

The Factory

`b.factory`, like `b.analyses` and `b.surveyors`, is a container object that has a lot of cool stuff in it. It is not a factory in the Java sense, it is merely a home for all the functions that produce new instances of important angr classes and should be sitting on Project.

```
>>> import claripy # used later

>>> block = b.factory.block(addr=b.entry)
>>> block = b.factory.block(addr=b.entry, byte_string='\xc3')
>>> block = b.factory.block(addr=b.entry, num_inst=1)

>>> state = b.factory.blank_state(addr=b.entry)
>>> state = b.factory.entry_state(args=['./program', claripy.BVS('arg1', 20*8)])
>>> state = b.factory.call_state(0x1000, "hello", "world")
>>> state = b.factory.full_init_state(args=['./program', claripy.BVS('arg1', 20*8)])

>>> path = b.factory.path()
>>> path = b.factory.path(state)

>>> group = b.factory.path_group()
>>> group = b.factory.path_group(path)
>>> group = b.factory.path_group([path, state])

>>> strlen_addr = b.loader.main_bin.plt['strlen']
>>> strlen = b.factory.callable(strlen_addr)
>>> assert claripy.is_true(strlen("hello") == 5)

>>> cc = b.factory.cc()
```

- *factory.block* is the angr's lifter. Passing it an address will lift a basic block of code from the binary at that address, and return an angr Block object that can be used to retrieve multiple representations of that block. More below.
- *factory.blank_state* returns a SimState object with little initialization besides the parameters passed to it. States as a whole are discussed in depth here.
- *factory.entry_state* returns a SimState initialized to the program state at the binary's entry point.
- *factory.call_state* returns a SimState initialized as if you'd just called the function at the given address, with the given args.
- *factory.full_init_state* returns a SimState that initialized similarly to `entry_state`, but instead of at the entry point, the program counter points to a SimProcedure that serves the purpose of the dynamic loader and will call the initializers of each shared library before jumping to the entry point.
- *factory.path* returns a Path object. Since Paths are at their start just light wrappers around SimStates, you can call `path` with a state as an argument and get a path wrapped around that state. Alternately, for simple cases, any keyword arguments you pass `path` will be passed on to `entry_state` to create a state to wrap. It is discussed in depth here.
- *factory.path_group* creates a path group! Path groups are the future. They're basically very smart lists of paths, so you can pass it a path, a state (which will be wrapped into a path), or a list of paths and states. They are discussed in depth here.

- *factory.callable* is *very* cool. Callables are a FFI (foreign functions interface) into arbitrary binary code. They are discussed in depth here.
- *factory.cc* intiializes a calling convention object. This can be initialized with different args or even a function prototype, and then passed to factory.callable or factory.call_state to customize how arguments and return values and return addresses are laid out into memory. It is discussed in depth here.

Lifter

Access the lifter through *factory.block*. This method has a number of optional parameters, which you can read about here! The bottom line, though, is that `block()` gives you back a generic interface to a basic block of code. You can get properties like `.size` (in bytes) from the block, but if you want to do interesting things with it, you need a more specific representation. Access `.vex` to get a PyVEX IRSB, or `.capstone` to get a Capstone block.

Filesystem Options

There are a number of options which can be passed to the state initialization routines which affect filesystem usage. These include the `fs`, `concrete_fs`, and `chroot` options.

The `fs` option allows you to pass in a dictionary of file names to preconfigured SimFile objects. This allows you to do things like set a concrete size limit on a file's content.

Setting the `concrete_fs` option to `True` will cause angr to respect the files on disk. For example, if during simulation a program attempts to open 'banner.txt' when `concrete_fs` is set to `False` (the default), a SimFile with a symbolic memory backing will be created and simulation will continue as though the file exists. When `concrete_fs` mode is set to `True`, if 'banner.txt' exists a new SimFile object will be created with a concrete backing, reducing the resulting state explosion which would be caused by operating on a completely symbolic file. Additionally in `concrete_fs` mode if 'banner.txt' mode does not exist, a SimFile object will not be created upon calls to open during simulation and an error code will be returned. Additionally, it's important to note that attempts to open files whose path begins with '/dev/' will never be opened concretely even with `concrete_fs` set to `True`.

The `chroot` option allows you to specify an optional root to use while using the `concrete_fs` option. This can be convenient if the program you're analyzing references files using an absolute path. For example, if the program you are analyzing attempts to open '/etc/passwd', you can set the chroot to your current working directory so that attempts to access '/etc/passwd' will read from '$CWD/etc/passwd'.

```
>>> import simuvex
>>> files = {'/dev/stdin': simuvex.storage.file.SimFile("/dev/stdin", "r", size=30)}
>>> s = b.factory.entry_state(fs=files, concrete_fs=True, chroot="angr-chroot/")
```

This example will create a state which constricts at most 30 symbolic bytes from being read from stdin and will cause references to files to be resolved concretely within the new root directory `angr-chroot`.

Important note that needs to go in this initial version before I write the rest of the stuff: the `args` and `env` keyword args work on `entry_state` and `full_init_state`, and are a list and a dict, respectively, of strings or claripy BV objects, which can represent a variety of concrete and symbolic strings. Read the source if you wanna know more about these!

Hooking

```
>>> def set_rax(state):
...     state.regs.rax = 10

>>> b.hook(0x10000, set_rax, length=5)
>>> b.is_hooked(0x10000)
True
>>> b.unhook(0x10000)
>>> b.hook_symbol('strlen', simuvex.SimProcedures['stubs']['ReturnUnconstrained'])
```

A hook allows you to intercept the program's execution at specific points. If you hook a program at an address, whenever the program's execution reaches that point, it will run the python code in the hook. Execution will then skip `length` bytes ahead of the hooked address and resume. You can omit the `length` argument for execution to skip zero bytes and resume at the hooked address.

In addition to a basic function, you can hook an address with a `SimProcedure`, which is a more complex system for having fine-grained control over program execution. To do this, use the exact same `hook` function, but supply a class (not an instance!) that subclasses `simuvex.SimProcedure`.

The `is_hooked` and `unhook` methods should be self-explanitory.

`hook_symbol` is a different function that serves a different purpose. Instead of an address, you pass it the name of a function that that binary imports. The internal (GOT) pointer to the code that function resolved to will be replaced with a pointer to the SimProcedure or hook function you specify in the third argument. You can also pass a plain integer to make replace pointers to the symbol with that value.

Loading a Binary - CLE and angr Projects

angr's binary loading component is CLE, which stands for CLE Loads Everything. CLE is responsible for taking a binary (and any libraries that it depends on) and presenting it to the rest of angr in a way that is easy to work with.

CLE's main goal is to load binaries in a robust way, i.e., the same way the actual loader (e.g., GNU LD in the case of ELF binaries) would load them. It means that some information that may be present in the binaries will be ignored by CLE, because such information may be stripped, voluntarily or involuntarily corrupted, etc.. It is not rare in the embedded world to see such things happening.

angr, in turn, encompasses this in a *Project* class. A Project class is the entity that represents your binary. Much of your interaction with angr will go through it.

To load a binary with angr (let's say "/bin/true"), you would do the following:

```
>>> import angr

>>> b = angr.Project("/bin/true")
```

After this, *b* is angr's representation of your binary (the "main" binary), along with any libraries that it depends on. There are several basic things that you can do here without further knowledge of the rest of the platform:

```
# this is the entry point of the binary
>>> print b.entry

# these are the minimum and maximum addresses of the binary's memory contents
>>> print b.loader.min_addr(), b.loader.max_addr()

# this is the full name of the binary
>>> print b.filename
```

CLE exposes the binary's information through the Loader class. The CLE loader (cle.Loader) represents an entire conglomerate of loaded CLE binary objects, loaded and mapped into a single memory space. Each binary object is loaded by a loader backend that can handle its filetype (a subclass of cle.Backend). For example, cle.ELF is used to load ELF binaries.

CLE can be interfaced with as follows:

```
# this is the CLE Loader object
>>> print b.loader

# this is a dictionary of the objects that are loaded as part of loading the binary (t
heir types depend on the backend)
>>> print b.loader.shared_objects

# this is the memory space of the process after being loaded. It maps addresses to the
 byte at that address.
>>> print b.loader.memory[b.loader.min_addr()]

# this is the object for the main binary (its type depends on the backend)
>>> print b.loader.main_bin

# this retrieves the binary object which maps memory at the specified address
>>> print b.loader.addr_belongs_to_object(b.loader.max_addr())

# Get the address of the GOT slot for a symbol (in the main binary)
>>> print b.loader.find_symbol_got_entry('__libc_start_main')
```

It is also possible to interface directly with individual binary objects:

```
# this is a list of the names of libraries the program depend on. We obtain it
# *statically* by reading the DT_NEEDED field of the dynamic section of the Elf
# binary.
>>> print b.loader.main_bin.deps

# this is a dict of the memory contents of *just* the main binary
>>> print b.loader.main_bin.memory

# this is a dict (name->ELFRelocation) of imports required by the libc which was loaded

>>> b.loader.shared_objects['libc.so.6'].imports

# this is a dict (name->ELFRelocation) of imports of the main binary, where addr is us
ually 0 (see the misc section below).
>>> print b.loader.main_bin.imports
```

Loading dependencies

By default, CLE will attempt to load all the dependencies of the main binary (e.g., libc.so.6, ld-linux.so.2, etc.), unless `auto_load_libs` is set to `False` in the loading options. When loading libraries, if it cannot find one of them, it will silently ignore the error and mark all the dependencies on that library as unresolved. If you like, you can change this behavior.

Loading Options

Loading options can be passed to Project (which in turn will pass it to CLE).

CLE expects a dict as a set of parameters. Parameters which must be applied to libraries which are not the target binary must be passed through the lib_opts parameter in the following form:

```
load_options = {'main_opts':{options0}, 'lib_opts': {libname1:{options1}, path2:{optio
ns2}, ...}}

# Or in a more readable form:
load_options = {}
load_options['main_opts'] = {k1:v1, k2:v2 ...}
load_options['lib_opts'] = {}
load_options['lib_opts'][path1] = {k1:v1, k2:v2, ...}
load_options['lib_opts'][path2] = {k1:v1, k2:v2, ...}
etc.
```

Valid options

```
>>> load_options = {}

# shall we also load dynamic libraries?
>>> load_options['auto_load_libs'] = False

# A list of libraries to load regardless of whether they're required by the loaded obj
ect
>>> load_options['force_load_libs'] = ['libleet.so']

# specific libs to skip
>>> load_options['skip_libs'] = ['libc.so.6']

# Options to be used when loading the main binary
>>> load_options['main_opts'] = {'backend': 'elf'}

# A dictionary mapping library names to a dictionary of objects to be used when loadin
g them.
>>> load_options['lib_opts'] = {'libc.so.6': {'custom_base_addr': 0x13370000}}

# A list of paths we can additionally search for shared libraries
>>> load_options['custom_ld_path'] = ['/my/fav/libs']

# Whether libraries with different version numbers in the filename will be considered
equivilant, for example libc.so.6 and libc.so.0
>>> load_options['ignore_import_version_numbers'] = False

# The alignment to use for rebasing shared objects
>>> load_options['rebase_granularity'] = 0x1000

# Throw an Exception if a lib cannot be found (the default is fail silently on missing
 libs)
>>> load_options['except_missing_libs'] = True
```

The following options are applied on a per object basis and override CLE's automatic detection. They can be applied through either 'main_opts' or 'lib_opts'.

```
# Base address to load the binary
>>> load_options['main_opts'] = {'custom_base_addr':0x4000}

# Specify the object's backend (backends discussed below)
>>> load_options['main_opts'] = {'backend': 'elf'}
```

Example with multiple options for the same binary:

```
>>> load_options['main_opts'] = {'backend':'elf', 'custom_base_addr': 0x10000}
```

Backends

CLE currently has backends for statically loading ELF, PE, CGC and ELF core dump files, as well as loading binaries with IDA and loading files into a flat address space. CLE will automatically detect the correct backend to use in most all cases, so you shouldn't need to specify which backend you're using unless you're doing some pretty weird stuff.

You can specify the backend for a binary by including a key in its options dictionary. If you need to force the architecture of a certain binary instead of having it auto-detected, you can specify it with the `custom_arch` key. The key doesn't need to match any list of arches; angr will identify which architecture you mean given almost any common identifier for any supported arch.

```
>>> load_options = {}
>>> load_options['main_opts'] = {'backend': 'elf', 'custom_arch': 'i386'}
>>> load_options['lib_opts'] = {'libc.so.6': {'backend': 'elf'}}
```

backend key	description	requires `custom_arch` ?
elf	Static loader for ELF files based on PyELFTools	no
pe	Static loader for PE files based on PEFile	no
cgc	Static loader for Cyber Grand Challenge binaries	no
backedcgc	Static loader for CGC binaries that allows specifying memory and register backers	no
elfcore	Static loader for ELF core dumps	no
ida	Launches an instance of IDA to parse the file	yes
blob	Loads the file into memory as a flat image	yes

Now that you have loaded a binary, interesting information about the binary is now accessible in `b.loader.main_bin`. For example, the shared library dependencies, the list of imported libraries, memory, symbols and others. Make heavy use of IPython's tab-completion to see available functions and options here.

Now it's time to look at the IR support

Misc

Imports

The following is ELF specific. On most architectures, imports, i.e., symbols that refer to functions or global names that are outside of the binary (in shared libraries) appear in the symbol table, most of the time with an undefined address (0). On some architectures like MIPS, it contains the address of the function's PLT stub (which resides in the text segment). If you are looking for the address of the GOT entry related to a specific symbol (which resides in the data segment), take a look at jmprel. It is a dict (symbol-> GOT addr):

Whether you are after a PLT or GOT entry depends on the architecture. Architecture specific stuff is defined in a class in the Archinfo repository. The way we deal with absolute addresses of functions depending on the architecture is defined in this class, in the got_section_name property.

For more details about ELF loading and architecture specific details, check the Executable and linkable format document as well as the ABI supplements for each architecture (MIPS, PPC64, AMD64)..

```
>>> rel = b.loader.main_bin.jmprel
```

Symbolic analysis: function summaries

By default, Project tries to replace external calls to libraries' functions by using symbolic summaries termed *SimProcedures* (these are summaries of how functions affect the state).

When no such summary is available for a given function:

- if `auto_load_libs` is `True` (this is the default), then the *real* library function is executed instead. This may or may not be what you want, depending on the actual function. For example, some of libc's functions are extremely complex to analyze and will most likely cause an explosion of the number of states for the path trying to execute them.
- if `auto_load_libs` is `False`, then external functions are unresolved, and Project will resolve them to a generic "stub" SimProcedure called `ReturnUnconstrained`. It does what its name says: it returns unconstrained values.
- if `use_sim_procedures` (this is a parameter to `angr.Project`, not `cle.Loader`) is `False` (it is `True` by default), then no SimProcedures besides `ReturnUnconstrained` will be used.
- you may specify specific symbols to exclude from being replaced with SimProcedures with the parameters to `angr.Project`: `exclude_sim_procedures_list` and `exclude_sim_procedures_func`.
- Look at the code for `angr.Project._use_sim_procedures` for the exact algorithm.

Intermediate Representation

Because angr deals with widely diverse architectures, it must carry out its analysis on an intermediate representation. We use Valgrind's IR, "VEX", for this. The VEX IR abstracts away several architecture differences when dealing with different architectures, allowing a single analysis to be run on all of them:

- **Register names.** The quantity and names of registers differ between architectures, but modern CPU designs hold to a common theme: each CPU contains several general purpose registers, a register to hold the stack pointer, a set of registers to store condition flags, and so forth. The IR provides a consistent, abstracted interface to registers on different platforms. Specifically, VEX models the registers as a separate memory space, with integer offsets (e.g., AMD64's `rax` is stored starting at address 16 in this memory space).
- **Memory access.** Different architectures access memory in different ways. For example, ARM can access memory in both little-endian and big-endian modes. The IR abstracts away these differences.
- **Memory segmentation.** Some architectures, such as x86, support memory segmentation through the use of special segment registers. The IR understands such memory access mechanisms.
- **Instruction side-effects.** Most instructions have side-effects. For example, most operations in Thumb mode on ARM update the condition flags, and stack push/pop instructions update the stack pointer. Tracking these side-effects in an *ad hoc* manner in the analysis would be crazy, so the IR makes these effects explicit.

There are lots of choices for an IR. We use VEX, since the uplifting of binary code into VEX is quite well supported. VEX is an architecture-agnostic, side-effects-free representation of a number of target machine languages. It abstracts machine code into a representation designed to make program analysis easier. This representation has four main classes of objects:

- **Expressions.** IR Expressions represent a calculated or constant value. This includes memory loads, register reads, and results of arithmetic operations.
- **Operations.** IR Operations describe a *modification* of IR Expressions. This includes integer arithmetic, floating-point arithmetic, bit operations, and so forth. An IR Operation applied to IR Expressions yields an IR Expression as a result.
- **Temporary variables.** VEX uses temporary variables as internal registers: IR Expressions are stored in temporary variables between use. The content of a temporary variable can be retrieved using an IR Expression. These temporaries are numbered,

starting at `t0`. These temporaries are strongly typed (e.g., "64-bit integer" or "32-bit float").

- **Statements.** IR Statements model changes in the state of the target machine, such as the effect of memory stores and register writes. IR Statements use IR Expressions for values they may need. For example, a memory store *IR Statement* uses an *IR Expression* for the target address of the write, and another *IR Expression* for the content.
- **Blocks.** An IR Block is a collection of IR Statements, representing an extended basic block (termed "IR Super Block" or "IRSB") in the target architecture. A block can have several exits. For conditional exits from the middle of a basic block, a special *Exit* IR Statement is used. An IR Expression is used to represent the target of the unconditional exit at the end of the block.

VEX IR is actually quite well documented in the `libvex_ir.h` file (https://github.com/angr/vex/blob/master/pub/libvex_ir.h) in the VEX repository. For the lazy, we'll detail some parts of VEX that you'll likely interact with fairly frequently. To begin with, here are some IR Expressions:

IR Expression	Evaluated Value	VEX Output Example
Constant	A constant value.	0x4:I32
Read Temp	The value stored in a VEX temporary variable.	RdTmp(t10)
Get Register	The value stored in a register.	GET:I32(16)
Load Memory	The value stored at a memory address, with the address specified by another IR Expression.	LDle:I32 / LDbe:I64
Operation	A result of a specified IR Operation, applied to specified IR Expression arguments.	Add32
If-Then-Else	If a given IR Expression evaluates to 0, return one IR Expression. Otherwise, return another.	ITE
Helper Function	VEX uses C helper functions for certain operations, such as computing the conditional flags registers of certain architectures. These functions return IR Expressions.	function_name()

These expressions are then, in turn, used in IR Statements. Here are some common ones:

IR Statement	Meaning	VEX Output Example
Write Temp	Set a VEX temporary variable to the value of the given IR Expression.	WrTmp(t1) = (IR Expression)
Put Register	Update a register with the value of the given IR Expression.	PUT(16) = (IR Expression)
Store Memory	Update a location in memory, given as an IR Expression, with a value, also given as an IR Expression.	STle(0x1000) = (IR Expression)
Exit	A conditional exit from a basic block, with the jump target specified by an IR Expression. The condition is specified by an IR Expression.	if (condition) goto (Boring) 0x4000A00:I32

An example of an IR translation, on ARM, is produced below. In the example, the subtraction operation is translated into a single IR block comprising 5 IR Statements, each of which contains at least one IR Expression (although, in real life, an IR block would typically consist of more than one instruction). Register names are translated into numerical indices given to the *GET* Expression and *PUT* Statement. The astute reader will observe that the actual subtraction is modeled by the first 4 IR Statements of the block, and the incrementing of the program counter to point to the next instruction (which, in this case, is located at `0x59FC8`) is modeled by the last statement.

The following ARM instruction:

```
subs R2, R2, #8
```

Becomes this VEX IR:

```
t0 = GET:I32(16)
t1 = 0x8:I32
t3 = Sub32(t0,t1)
PUT(16) = t3
PUT(68) = 0x59FC8:I32
```

Now that you understand VEX, you can actually play with some VEX in angr: We use a library called PyVEX (https://github.com/angr/pyvex) that exposes VEX into Python. In addition, PyVEX implements its own pretty-printing so that it can show register names instead of register offsets in PUT and GET instructions.

PyVEX is accessable through angr through the `Project.factory.block` interface. There are many different representations you could use to access syntactic properties of a block of code, but they all have in common the trait of analyzing a particular sequence of bytes.

Through the `factory.block` constructor, you get a `Block` object that can be easily turned into several different representations. Try `.vex` for a PyVEX IRSB, or `.capstone` for a Capstone block.

Let's play with PyVEX:

```
>>> import angr

# load the program binary
>>> b = angr.Project("/bin/true")

# translate the starting basic block
>>> irsb = b.factory.block(b.entry).vex
# and then pretty-print it
>>> irsb.pp()

# translate and pretty-print a basic block starting at an address
>>> irsb = b.factory.block(0x401340).vex
>>> irsb.pp()

# this is the IR Expression of the jump target of the unconditional exit at the end of
 the basic block
>>> print irsb.next

# this is the type of the unconditional exit (e.g., a call, ret, syscall, etc)
>>> print irsb.jumpkind

# you can also pretty-print it
>>> irsb.next.pp()

# iterate through each statement and print all the statements
>>> for stmt in irsb.statements:
...     stmt.pp()

# pretty-print the IR expression representing the data, and the *type* of that IR expr
ession written by every store statement
>>> import pyvex
>>> for stmt in irsb.statements:
...     if isinstance(stmt, pyvex.IRStmt.Store):
...         print "Data:",
...         stmt.data.pp()
...         print ""
...         print "Type:",
...         print stmt.data.result_type
...         print ""

# pretty-print the condition and jump target of every conditional exit from the basic
block
... for stmt in irsb.statements:
...     if isinstance(stmt, pyvex.IRStmt.Exit):
...         print "Condition:",
...         stmt.guard.pp()
```

```
...         print ""
...         print "Target:",
...         stmt.dst.pp()
...         print ""

# these are the types of every temp in the IRSB
>>> print irsb.tyenv.types

# here is one way to get the type of temp 0
>>> print irsb.tyenv.types[0]
```

Keep in mind that this is a *syntactic* respresentation of a basic block. That is, it'll tell you what the block means, but you don't have any context to say, for example, what *actual* data is written by a store instruction. We'll get to that next.

Solver Engine

angr's solver engine is called Claripy. Claripy exposes the following:

- Claripy ASTs (the subclasses of claripy.ast.Base) provide a unified way to interact with concrete and symbolic expressions
- Claripy frontends provide a unified interface to expression resolution (including constraint solving) over different backends

Internally, Claripy seamlessly mediates the co-operation of multiple disparate backends -- concrete bitvectors, VSA constructs, and SAT solvers. It is pretty badass.

Most users of angr will not need to interact directly with Claripy (except for, maybe, claripy AST objects, which represent symbolic expressions) -- SimuVEX handles most interactions with Claripy internally. However, for dealing with expressions, an understanding of Claripy might be useful.

Claripy ASTs

Claripy ASTs abstract away the differences between the constructs that Claripy supports. They define a tree of operations (i.e., `(a + b) / c`) on any type of underlying data. Claripy handles the application of these operations on the underlying objects themselves by dispatching requests to the backends.

Currently, Claripy supports the following types of ASTs:

Name	Description	Supported By (Claripy Backends)	Example Code
BV	This is a bitvector, whether symbolic (with a name) or concrete (with a value). It has a size (in bits).	BackendConcrete, BackendVSA, BackendZ3	• Create a 32-bit symbolic bitvector "x": `claripy.BVS('x', 32)` • Create a 32-bit bitvector with the value `0xc001b3475`: `claripy.BVV(0xc001b3a75, 32)` • Create a 32-bit "strided interval" (see VSA documentation) that can be any divisible-by-10 number between 1000 and 2000: `claripy.SI(name='x', bits=32, lower_bound=1000, upper_bound=2000, stride=10)`
FP	This is a floating-point number, whether symbolic (with a name) or concrete (with a value).	BackendConcrete, BackendZ3	TODO
Bool	This is a boolean operation (True or False).	BackendConcrete, BackendVSA, BackendZ3	`claripy.BoolV(True)`, or `claripy.true` or `claripy.false`, or by comparing two ASTs (i.e., `claripy.BVS('x', 32) < claripy.BVS('y', 32)`

All of the above creation code returns claripy.AST objects, on which operations can then be carried out.

ASTs provide several useful operations.

```
>>> 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
```

Applying a condition (==, !=, etc) on ASTs will return an AST that represents the condition being carried out. For example:

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

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

You can combine these conditions in different ways.

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

The usefulness of this will become apparent when we discuss Claripy solvers.

In general, Claripy supports all of the normal python operations (+, -, |, ==, etc), and provides additional ones via the Claripy instance object. Here's a list of available operations from the latter.

Name	Description	Example
LShR	Logically shifts a bit expression (BVV, BV, SI) to the right.	`claripy.LShR(x, 10)`
SignExt	Sign-extends a bit expression.	`claripy.SignExt(32, x)` or `x.sign_extend(32)`
ZeroExt	Zero-extends a bit expression.	`claripy.ZeroExt(32, x)` or `x.zero_extend(32)`
Extract	Extracts the given bits (zero-indexed from the *right*, inclusive) from a bit expression.	Extract the rightmost byte of x: `claripy.Extract(7, 0, x)` or `x[7:0]`
Concat	Concatenates several bit expressions together into a new bit expression.	`claripy.Concat(x, y, z)`
RotateLeft	Rotates a bit expression left.	`claripy.RotateLeft(x, 8)`
RotateRight	Rotates a bit expression right.	`claripy.RotateRight(x, 8)`
Reverse	Reverses a bit expression.	`claripy.Reverse(x)` or `x.reversed`
And	Logical And (on boolean expressions)	`claripy.And(x == y, x > 0)`
Or	Logical Or (on boolean expressions)	`claripy.Or(x == y, y < 10)`
Not	Logical Not (on a boolean expression)	`claripy.Not(x == y)` is the same as `x != y`
If	An If-then-else	Choose the maximum of two expressions: `claripy.If(x > y, x, y)`
ULE	Unsigned less than or equal to.	Check if x is less than or equal to y: `claripy.ULE(x, y)`
ULT	Unsigned less than.	Check if x is less than y: `claripy.ULT(x, y)`
UGE	Unsigned greater than or equal to.	Check if x is greater than or equal to y: `claripy.UGE(x, y)`
UGT	Unsigned greater than.	Check if x is greater than y: `claripy.UGT(x, y)`
SLE	Signed less than or equal to.	Check if x is less than or equal to y: `claripy.SLE(x, y)`
SLT	Signed less than.	Check if x is less than y: `claripy.SLT(x, y)`
SGE	Signed greater than or equal to.	Check if x is greater than or equal to y: `claripy.SGE(x, y)`
SGT	Signed greater than.	Check if x is greater than y: `claripy.SGT(x, y)`

NOTE: The default python `>`, `<`, `>=`, and `<=` are unsigned in Claripy. This is different than their behavior in Z3, because it seems more natural in binary analysis.

Solvers

The main point of interaction with Claripy are the Claripy Solvers. Solvers expose an API to interpret ASTs in different ways and return usable values. There are several different solvers.

Name	Description
Solver	This is analogous to a `z3.Solver()`. It is a solver that tracks constraints on symbolic variables and uses a constraint solver (currently, Z3) to evaluate symbolic expressions.
SolverVSA	This solver uses VSA to reason about values. It is an *approximating* solver, but produces values without performing actual constraint solves.
SolverReplacement	This solver acts as a pass-through to a child solver, allowing the replacement of expressions on-the-fly. It is used as a helper by other solvers and can be used directly to implement exotic analyses.
SolverHybrid	This solver combines the SolverReplacement and the Solver (VSA and Z3) to allow for *approximating* values. You can specify whether or not you want an exact result from your evaluations, and this solver does the rest.
SolverComposite	This solver implements optimizations that solve smaller sets of constraints to speed up constraint solving.

Some examples of solver usage:

```
# create the solver and an expression
>>> s = claripy.Solver()
>>> x = claripy.BVS('x', 8)

# now let's add a constraint on x
>>> s.add(claripy.ULT(x, 5))

>>> assert sorted(s.eval(x, 10)) == [0, 1, 2, 3, 4]
>>> assert s.max(x) == 4
>>> assert s.min(x) == 0

# we can also get the values of complex expressions
>>> y = claripy.BVV(65, 8)
>>> z = claripy.If(x == 1, x, y)
>>> assert sorted(s.eval(z, 10)) == [1, 65]

# and, of course, we can add constraints on complex expressions
>>> s.add(z % 5 != 0)
>>> assert s.eval(z, 10) == (1,)
>>> assert s.eval(x, 10) == (1,) # interestingly enough, since z can't be y, x can onl
y be 1!
```

Custom solvers can be built by combining a Claripy Frontend (the class that handles the actual interaction with SMT solver or the underlying data domain) and some combination of frontend mixins (that handle things like caching, filtering out duplicate constraints, doing opportunistic simplification, and so on).

Claripy Backends

Backends are Claripy's workhorses. Claripy exposes ASTs to the world, but when actual computation has to be done, it pushes those ASTs into objects that can be handled by the backends themselves. This provides a unified interface to the outside world while allowing Claripy to support different types of computation. For example, BackendConcrete provides computation support for concrete bitvectors and booleans, BackendVSA introduces VSA constructs such as StridedIntervals (and details what happens when operations are performed on them, and BackendZ3 provides support for symbolic variables and constraint solving.

There are a set of functions that a backend is expected to implement. For all of these functions, the "public" version is expected to be able to deal with claripy's AST objects, while the "private" version should only deal with objects specific to the backend itself. This is distinguished with Python idioms: a public function will be named func() while a private function will be _func(). All functions should return objects that are usable by the backend in

its private methods. If this can't be done (i.e., some functionality is being attempted that the backend can't handle), the backend should raise a BackendError. In this case, Claripy will move on to the next backend in its list.

All backends must implement a `convert()` function. This function receives a claripy AST and should return an object that the backend can handle in its private methods. Backends should also implement a `_convert()` method, which will receive anything that is *not* a claripy AST object (i.e., an integer or an object from a different backend). If `convert()` or `_convert()` receives something that the backend can't translate to a format that is usable internally, the backend should raise BackendError, and thus won't be used for that object. All backends must also implement any functions of the base `Backend` abstract class that currently raise `NotImplementedError()`.

Claripy's contract with its backends is as follows: backends should be able to handle, in their private functions, any object that they return from their private *or* public functions. Claripy will never pass an object to any backend private function that did not originate as a return value from a private or public function of that backend. One exception to this is `convert()` and `_convert()`, as Claripy can try to stuff anything it feels like into _convert() to see if the backend can handle that type of object.

Model Objects

To perform actual, useful computation on ASTs, Claripy uses model objects. A model object is a result of the operation represented by the AST. Claripy expects these objects to be returned from the backends, and will pass such objects into that backend's other functions.

Machine State - memory, registers, and so on

angr (actually, a submodule of angr, called SimuVEX) tracks machine states in a `SimState` object. This object tracks concrete and/or symbolic values for the machine's memory, registers, along with various other information, such as open files. You can get a `SimState` by using one of a number of convenient constructors in `Project.factory`. The different basic states you can construct are described here.

```
>>> import angr, simuvex
>>> b = angr.Project('/bin/true')

# let's get a state at the program entry point:
>>> s = b.factory.entry_state()

# we can access the memory of the state here
>>> print "The first 5 bytes of the binary are:", s.memory.load(b.loader.min_addr(), 5
)

# and the registers, of course
>>> print "The stack pointer starts out as:", s.regs.sp
>>> print "The instruction pointer starts out as:", s.regs.ip

# and the temps, although these are currently empty
>>> # print "This will throw an exception because there is no VEX temp t0, yet:", s.sc
ratch.tmp_expr(0)
```

Accessing Data

The data that's stored in the state (i.e., data in registers, memory, temps, etc) is stored as an internal *expression*. This exposes a single interface to concrete (i.e., `0x41414141`) and symbolic (i.e., "whatever the user might input on stdin") expressions. In fact, this is the core of what enables angr to analyze binaries *symbolically*. However, this complicates matters by not exposing the actual *value*, if it's concrete, directly. For example, if you try the above examples, you will see that the type that is printed is a claripy AST, which is the internal expression representation. For now, you might want to know how to get the actual values out of these expressions.

```
# get the integer value of the content of rax:
>>> print s.se.any_int(s.regs.rax)

# or, the string value of the 10 bytes stored at 0x1000
>>> print s.se.any_str(s.memory.load(0x1000, 10))

# get the value of the 4 bytes stored at 0x2000, i.e. a little-endian int
# note that unless otherwise specified, all loads from memory are big-endian by default

>>> print s.se.any_int(s.memory.load(0x2000, 4, endness='Iend_LE'))
```

Here, `s.se` is a solver engine that holds the symbolic constraints on the state.

This syntax might seem a bit strange -- we get the expression from the state, and then we pass it back *into* the state to get its actual value. This is, in fact, quite intentional. As we mentioned earlier, these expressions could be either concrete or symbolic. In the case of the latter, a symbolic expression might resolve to two different meanings in two different states. We'll go over symbolic expressions in more detail later on. For now, accept the mystery.

Storing Data

If you want to store content in the state's memory or registers, you'll need to create an expression out of it. You can do it like so:

```
# this creates a BVV (which stands for BitVector Value). A BVV is a bitvector that's u
sed to represent
# data in memory, registers, and temps. This BVV represents a 32 bit bitvector of four
 ascii `A` characters
>>> import claripy
>>> aaaa = claripy.BVV(0x41414141, 32)

# While we're at it, we can do various operations on these bitvectors:
>>> aa = aaaa[31:16] # this extracts the most significant 16 bits
>>> aa00 = aaaa & claripy.BVV(0xffff0000, 32)
>>> aaab = aaaa + 1
>>> aaaaaaaa = claripy.Concat(aaaa, aaaa)

# this can then be stored in memory or registers. Since the bitvector
# has a length, only the address to store it at is required
>>> s.regs.rax = aaaa
>>> s.memory.store(0x1000, aaaa)

# of course, you can address memory using expressions as well
>>> s.memory.store(s.regs.rax, aaaa)
```

For convenience, there are special accessor functions stack operations:

```
# push our "AAAA" onto the stack
>>> s.stack_push(aaaa)

# and pop it off
>>> aaaa = s.stack_pop()
```

Copying and Merging

A state supports very fast copies, so that you can explore different possibilities:

```
>>> s1 = s.copy()
>>> s2 = s.copy()

>>> s1.memory.store(0x1000, s1.se.BVV(0x41414141, 32))
>>> s2.memory.store(0x1000, s2.se.BVV(0x42424242, 32))
```

States can also be merged together.

```
# 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.memory.load(0x1000, 4)
```

This is where we truly start to enter the realm of symbolic expressions. In the above example, the value of `aaaa_or_bbbb` can be, as it implies, either "AAAA" or "BBBB".

Symbolic Expressions

Symbolic values are expressions that, under different situations, can take on different values. Our symbolic expression, `aaaa_or_bbbb` is a great example of this. The solver engine provides ways to get at both values:

```
# this will return a sequence of up to n possible values of the expression in this sta
te.
# in our case, there are only two values, and it'll return [ "AAAA", "BBBB" ]
>>> print "This has 2 values:", s_merged.se.any_n_str(aaaa_or_bbbb, 2)
>>> print "This *would* have up to 5, but there are only two available:", s_merged.se.
any_n_str(aaaa_or_bbbb, 5)

# there's also the same for the integer value
>>> print s_merged.se.any_n_int(aaaa_or_bbbb, 2)
```

Of course, there are other ways to encounter symbolic expression than merging. For example, you can create them outright:

```
# This creates a simple symbolic expression: just a single symbolic bitvector by itsel
f. The bitvector is 32-bits long.
# An auto-incrementing numerical ID, and the size, are appended to the name, since nam
es of symbolic bitvectors must be unique.
>>> v = s.se.BVS("some_name", 32)

# If you want to prevent appending the ID and size to the name, you can, instead, do:
>>> v = s.se.BVS("some_name", 32, explicit_name=True)
```

Symbolic expressions can be interacted with in the same way as normal (concrete) bitvectors. In fact, you can even mix them:

```
# Create a concrete and a symbolic expression
>>> v = s.se.BVS("some_name", 32)
>>> aaaa = s.se.BVV(0x41414141, 32)

# Do operations involving them, and retrieve possible numerical solutions
>>> print s.se.any_int(aaaa)
>>> print s.se.any_int(aaaa + v)
>>> print s.se.any_int((aaaa + v) | s.se.BVV(0xffff0000, 32))

# You can tell between symbolic and concrete expressions fairly easily:
>>> assert s.se.symbolic(v)
>>> assert not s.se.symbolic(aaaa)

# You can even tell *which* variables make up a given expression.
>>> assert s.se.variables(aaaa) == set()
>>> #assert s.se.variables(aaaa + v) == { "some_name_4_32" } # that's the ID and size
appended to the name
# This assertion will fail because it depends on precisely the number of symbolic valu
es previously created
```

As you can see, symbolic and concrete expressions are pretty interchangeable, which is an extremely useful abstraction provided by SimuVEX. You might also notice that, when you read from memory locations that were never written to, you receive symbolic expressions:

```
# Try it!
>>> m = s.memory.load(0xbbbb0000, 8)

# The result is symbolic
>>> assert s.se.symbolic(m)

# Along with the ID and length, the address at which this expression originated is als
o added to the name
>>> #assert s.se.variables(m) == { "mem_bbbb0000_5_64" }

# And, of course, we can get the numerical or string solutions for the expression
>>> print s.se.any_n_int(m, 10)
>>> print s.se.any_str(m)
```

So far, we've seen addition being used. But we can do much more. All of the following examples return new expressions, with the operation applied.

```
# mods aaaa by 0x100, creating an expression, of the same size as aaaa, with all but t
he last byte zeroed out
>>> print aaaa % 0x100

# same effect, but with a bitwise and
>>> print aaaa & 0xff

# extracts the most significant (leftmost) byte of aaaa. The range is inclusive on bot
h sides, and indexed with the rightmost bit being 0
>>> print aaaa[31:24]

# concatenates aaaa with itself
>>> print aaaa.Concat(aaaa)

# zero-extends aaaa by 32 bits
>>> print aaaa.zero_extend(32)

# sign-extends aaaa by 32 bits
>>> print aaaa.sign_extend(32)

# shifts aaaa right arithmetically by 8 bits (i.e., sign-extended)
>>> print aaaa >> 8

# shifts aaaa right logically by 8 bits (i.e., not sign-extended)
>>> print aaaa.LShR(8)

# reverses aaaa, i.e. reverses the order of the bytes as if stored big-endian and load
ed little-endian
>>> print aaaa.reversed

# returns a list of expressions, representing the individual *bits* of aaaa (expressio
ns of length 1)
>>> print aaaa.chop()

# same, but for the bytes
>>> print aaaa.chop(bits=8)

# and the dwords
>>> print aaaa.chop(bits=16)
```

More details on the operations supported by the solver engine are available at the solver engine's documentation.

Symbolic Constraints

Symbolic expressions would be pretty boring on their own. After all, the last few that we created could take *any* numerical value, as they were completely unconstrained. This makes them uninteresting. To spice things up, SimuVEX has the concept of symbolic constraints.

Symbolic constraints represent, aptly, constraints (or restrictions) on symbolic expressions. It might be easier to show you:

```
# make a copy of the state so that we don't screw up the original with our experimenta
tion
>>> s3 = s.copy()

# Let's read some previously untouched section of memory to get a symbolic expression
>>> m = s.memory.load(0xbbbb0000, 1)

# We can verify that *any* solution would do
>>> assert s3.se.solution(m, 0)
>>> assert s3.se.solution(m, 10)
>>> assert s3.se.solution(m, 20)
>>> assert s3.se.solution(m, 30)
# ... and so on

# Now, let's add a constraint, forcing m to be less than 10
>>> s3.add_constraints(m < 10)

# We can see the effect of this right away!
>>> assert s3.se.solution(m, 0)
>>> assert s3.se.solution(m, 5)
>>> assert not s3.se.solution(m, 20)
>>> assert not s3.se.solution(m, 30)

# But the constraint does not affect the original state
>>> assert s.se.solution(m, 0)
>>> assert s.se.solution(m, 10)
>>> assert s.se.solution(m, 20)
>>> assert s.se.solution(m, 30)
```

One cautionary piece of advice is that the comparison operators (`>` , `<` , `>=` , `<=`) are *unsigned* by default. That means that, in the above example, this is the case:

```
# This is actually -1
assert not s3.se.solution(m, 0xff)
```

If we want *signed* comparisons, we need to use the unsigned versions of the operators (`SGT` , `SLT` , `SGE` , `SLE`). If you'd like to be explicit about your unsigned comparisons, the operators (`UGT` , `ULT` , `UGE` , `ULE`) are available.

```
# Add an unsigned comparison
>>> s4 = s.copy()
>>> s4.add_constraints(claripy.SLT(m, 10))

# We can see the effect of this right away!
>>> assert s4.se.solution(m, 0)
>>> assert s4.se.solution(m, 5)
>>> assert not s4.se.solution(m, 20)
>>> assert s4.se.solution(m, 0xff)
```

Amazing. Of course, constraints can be arbitrarily complex:

```
>>> s4.add_constraints(claripy.And(claripy.UGT(m, 10), claripy.Or(claripy.ULE(m, 100),
 m % 200 != 123, claripy.LShR(m, 8) & 0xff != 0xa)))
```

There's a lot there, but, basically, m has to be greater than 10 *and* either has to be less than 100, or has to be 123 when modded with 200, or, when logically shifted right by 8, the least significant byte must be 0x0a.

State Options

There are a lot of little tweaks that can be made to the internals of simuvex that will optimize behavior in some situations and be a detriment in others. These tweaks are controlled through state options.

On each SimState object, there is a set (state.options) of all its enabled options. The full domain of options, along with the defaults for different state types, can be found in (s_options.py)[https://github.com/angr/simuvex/blob/master/simuvex/s_options.py], available as `simuvex.o`.

When creating a SimState through any method, you may pass the keyword arguments `add_options` and `remove_options`, which should be sets of options that modify the initial options set from the default.

```
# Example: enable lazy solves, a behavior that causes state satisfiability to be check
ed as infrequently as possible.
# This change to the settings will be propogated to all successor states created from
this state after this line.
>>> s.options.add(simuvex.o.LAZY_SOLVES)

# Create a new state with lazy solves enabled
>>> s9 = b.factory.entry_state(add_options={simuvex.o.LAZY_SOLVES})
```

Symbolic Execution

Symbolic execution allows at a time T to determine for a branch all conditions necessary to take the branch or not. Every variable is represented as a symbolic value, and each branch as a constraint. Thus, symbolic execution allows us to see which conditions allows the program to go from a point A to a point B, by resolving the constraints.

Basic architecture of angr's symbolic execution:

- simuvex.md is the core engine and provides the concept of a symbolic machine state
- Also the means to tick that state forward through simulating VEX or running python code
- Use Paths to control execution easily and also to track history
- Use Path Groups to bulk-control execution

SimuVEX and Bare-Bones Symbolic Execution

Most analyses require an understanding of what the code is *doing* (semantic meaning), not just what the code *is* (syntactic meaning). For this, we developed a module called SimuVEX (https://github.com/angr/simuvex). SimuVEX provides a semantic understanding of what a given piece of VEX code does on a given machine state.

In a nutshell, SimuVEX is a symbolic VEX emulator. Given a machine state and a VEX IR block, SimuVEX provides a resulting machine state (or, in the case of condition jumps, *several* resulting machine states).

SimEngines

SimuVEX uses a series of engines to emulate the effects that of a given section of code has on an input state. This mechanism has changed recently, so we have removed much related documentation pending a rewrite. This information is not critical to the use of angr, since it is abstracted away by `Path` and `PathGroup`, but it provides useful insight into angr's functionality.

TODO: much things

SimSuccessors

`SimEngine.process` takes an input state and engine-specific arguments (such as a block of VEX IR for `SimEngineVEX`) and returns a SimSuccessors object that contains the successor states, with modifications applied. Since simuvex supports symbolic execution, there can be *multiple* output successor states for a single input state. The successor states are stored in individual lists. They are:

Attribute	Guard Condition	Instruction Pointer	Descr
`successors`	True (can be symbolic, but	Can be symbolic (but 256 solutions or less; see	A normal, sat successor sta state process engine. The i pointer of this be symbolic (computed jun

	constrained to True)	`unconstrained_successors`).	user input), s might actuall *several* poter continuations going forwarc
`unsat_successors`	False (can be symbolic, but constrained to False).	Can be symbolic.	Unsatisfiable These are su whose guard can only be fa jumps that ca taken, or the branch of jum be taken).
`flat_successors`	True (can be symbolic, but constrained to True).	Concrete value.	As noted abo the `successor` have symboli pointers. This confusing, as in the code (i `SimEngineVEX` when it's time state forward assumptions program state represents th of a single sp code. To alle when we enc in `successors` symbolic inst pointers, we possible conc solutions (up arbitrary thres for them, and copy of the st such solution process "flatt These `flat_` are states, ea has a differer instruction po example, if th pointer of a s `successors` where `X` ha of `X > 0x8000` `0x800010`, we flatten it into `flat_success` one with an ir

			pointer of `0x` with `0x80000` until `0x80001`
`unconstrained_successors`	True (can be symbolic, but constrained to True).	Symbolic (with more than 256 solutions).	During the fla procedure de above, if it tu there are mo possible solu instruction po assume that instruction po been overwri unconstraine stack overflo data). *This as not sound in* Such states a `unconstraine` and not in `su`
`all_successors`	Anything	Can be symbolic.	This is `succe` `unsat_success` `unconstrained`

SimProcedures

SimProcedures are, first and foremost, *symbolic function summaries*: angr handles functions imported into the binary by executing a SimProcedure that symbolically implements the given library function, if one exists. SimProcedures are a generic enough interface to do more than this, though - they can be used to run Python code to mutate a state at any point in execution.

SimProcedures are injected into angr's execution pipeline through an interface called *hooking*. The full interface is described here, but the most important part is the `Project.hook(address, procedure)` method. After running this, whenever execution in this project reaches `address`, instead of running the binary code at that address, we run the SimProcedure specified by the `procedure` argument.

`Project.hook` can also take a plain python function as an argument, instead of a SimProcedure class. That function will be automatically wrapped by a SimProcedure and executed (with the current SimState) as its argument.

TODO: Programming SimProcedures. Cover all the kinds of control flow, inline calls, etc. If you want to program a SimProcedure now, look at the library of already-written ones.

Breakpoints

Like any decent execution engine, SimuVEX supports breakpoints. This is pretty cool! A point is set as follows:

```
>>> import angr, simuvex
>>> 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 ha
ppens.
>>> s.inspect.b('mem_write')

# on the other hand, we can have a breakpoint trigger right *after* a memory write hap
pens.
# 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=simuvex.BP_AFTER, action=debug_func)

# or, you can have it drop you in an embedded ipython!
>>> s.inspect.b('mem_write', when=simuvex.BP_AFTER, action='ipython')
```

There are many other places to break than a memory write. Here is the list. You can break at BP_BEFORE or BP_AFTER for each of these events.

Event type	Event meaning
mem_read	Memory is being read.
mem_write	Memory is being written.
reg_read	A register is being read.
reg_write	A register is being written.
tmp_read	A temp is being read.
tmp_write	A temp is being written.
expr	An expression is being created (i.e., a result of an arithmetic operation or a constant in the IR).
statement	An IR statement is being translated.
instruction	A new (native) instruction is being translated.
irsb	A new basic block is being translated.
constraints	New constraints are being added to the state.
exit	A successor is being generated from execution.
symbolic_variable	A new symbolic variable is being created.
call	A call instruction is hit.
address_concretization	A symbolic memory access is being resolved.

These events expose different attributes:

Event type	Attribute name	Attribute availability	
mem_read	mem_read_address	BP_BEFORE or BP_AFTER	Tl m
mem_read	mem_read_length	BP_BEFORE or BP_AFTER	Tl re
mem_read	mem_read_expr	BP_AFTER	Tl ac
mem_write	mem_write_address	BP_BEFORE or BP_AFTER	Tl m
mem_write	mem_write_length	BP_BEFORE or BP_AFTER	Tl w
mem_write	mem_write_expr	BP_BEFORE or	Tl

		BP_AFTER	
reg_read	reg_read_offset	BP_BEFORE or BP_AFTER	Tl be
reg_read	reg_read_length	BP_BEFORE or BP_AFTER	Tl re
reg_read	reg_read_expr	BP_AFTER	Tl re
reg_write	reg_write_offset	BP_BEFORE or BP_AFTER	Tl be
reg_write	reg_write_length	BP_BEFORE or BP_AFTER	Tl w
reg_write	reg_write_expr	BP_BEFORE or BP_AFTER	Tl be
tmp_read	tmp_read_num	BP_BEFORE or BP_AFTER	Tl be
tmp_read	tmp_read_expr	BP_AFTER	Tl te
tmp_write	tmp_write_num	BP_BEFORE or BP_AFTER	Tl w
tmp_write	tmp_write_expr	BP_AFTER	Tl th
expr	expr	BP_AFTER	Tl ex
statement	statement	BP_BEFORE or BP_AFTER	Tl st bl
instruction	instruction	BP_BEFORE or BP_AFTER	Tl in
irsb	address	BP_BEFORE or BP_AFTER	Tl bl
constraints	added_constrints	BP_BEFORE or BP_AFTER	Tl ex

call	function_name	BP_BEFORE or BP_AFTER	Tl be
exit	exit_target	BP_BEFORE or BP_AFTER	Tl re a
exit	exit_guard	BP_BEFORE or BP_AFTER	Tl re a
exit	jumpkind	BP_BEFORE or BP_AFTER	Tl re Si
symbolic_variable	symbolic_name	BP_BEFORE or BP_AFTER	Tl va Tl m ap ar sy sy
symbolic_variable	symbolic_size	BP_BEFORE or BP_AFTER	Tl va
symbolic_variable	symbolic_expr	BP_AFTER	Tl re sy
address_concretization	address_concretization_strategy	BP_BEFORE or BP_AFTER	Tl Si be ac m ha st ap ha th sk
address_concretization	address_concretization_action	BP_BEFORE or BP_AFTER	Tl be m
address_concretization	address_concretization_memory	BP_BEFORE or BP_AFTER	Tl w ta
		BP_BEFORE	Tl m

address_concretization	address_concretization_expr	or BP_AFTER	ha af re
address_concretization	address_concretization_add_constraints	BP_BEFORE or BP_AFTER	W sh th
address_concretization	address_concretization_result	BP_AFTER	Tl m (ir ha th re

These attributes can be accessed as members of `state.inspect` during the appropriate breakpoint callback to access the appropriate values. You can even modify these value to modify further uses of the values!

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

Additionally, each of these properties can be used as a keyword argument to `inspect.b` to make the breakpoint conditional:

```
# This will break before a memory write if 0x1000 is a possible value of its target ex
pression
>>> 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 ex
pression
>>> 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=simuvex.BP_AFTER, instruction=0x8000, mem_read_exp
r=0x1000)
```

Cool stuff! In fact, we can even specify a function as a condition:

```
# 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.any_str(state.regs.rax) == 'AAAA' and 0x8004 in state.inspect.bac
ktrace

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

That is some cool stuff!

Symbolic memory indexing

SimuVEX supports *symbolic memory addressing*, meaning that offsets into memory may be symbolic. Our implementation of this is inspired by "Mayhem". Specifically, this means that angr concretizes symbolic addresses when they are used as the target of a write. This causes some surprises, as users tend to expect symbolic writes to be treated purely symbolically, or "as symbolically" as we treat symbolic reads, but that is not the default behavior. However, like most things in angr, this is configurable.

The address resolution behavior is governed by *concretization strategies*, which are subclasses of `simuvex.concretization_strategies.SimConcretizationStrategy`. Concretization strategies for reads are set in `state.memory.read_strategies` and for writes in `state.memory.write_strategies`. These strategies are called, in order, until one of them is able to resolve addresses for the symbolic index. By setting your own concretization strategies (or through the use of SimInspect `address_concretization` breakpoints, described above), you can change the way SimuVEX resolves symbolic addresses.

For example, angr's default concretization strategies for writes are:

1. A conditional concretization strategy that allows symbolic writes (with a maximum range of 128 possible solutions) for any indices that are annotated with `simuvex.plugins.symbolic_memory.MultiwriteAnnotation`.
2. A concretization strategy that simply selects the maximum possible solution of the symbolic index.

To enable symbolic writes for all indices, you can either add the `SYMBOLIC_WRITE_ADDRESSES` state option at state creation time or manually insert a `simuvex.concretization_strategies.SimConcretizationStrategyRange` object into `state.memory.write_strategies`. The strategy object takes a single argument, which is the maximum range of possible solutions that it allows before giving up and moving on to the next (presumably non-symbolic) strategy.

SimuVEX Options

SimuVEX is extremely customizable through the use of *state options*, a set of constants stored in `state.options`. These options are documented in the source code.

Program Paths - Controlling Execution

SimuVEX provides an incredibly awkward interface for performing symbolic execution. Paths are angr's primary interface to provide an abstraction to control execution, and are used in most interactions with angr and its analyses.

A path through a program is, at its core, a sequence of basic blocks (actually, individual executions of a `simuvex.SimEngine`) representing what was executed since the program started. These blocks in the paths can repeat (in the case of loops) and a program can have a near-infinite amount of paths (for example, a program with a single branch will have two paths, a program with two branches nested within each other will have 4, and so on).

To create an empty path at the program's entry point, do:

```
# load a binary

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

# load the path
>>> p = b.factory.path()

# this is the address that the path is *about to* execute
>>> assert p.addr == b.entry
```

After this, `p` is a path representing the program at the entry point. We can see that the callstack and the path's history are blank:

```
# this is the number of basic blocks that have been analyzed by the path
>>> assert p.length == 0

# we can also look at the current backtrace of program execution
# contains only the dummy frame for execution start
>>> assert len(p.callstack) == 1
>>> print p.callstack
Backtrace:
Func 0x401410, sp=0x7fffffffffeffd8, ret=0x0
```

Moving Forward

Of course, we can't be stuck at the entry point forever. call `p.step()` to run the single block of symbolic execution. We can look at the `successors` of a path to see where the program goes after this point. `p.step()` also returns the successors if you'd like to chain calls. Most of the time, a path will have one or two successors. When there are two successors, it usually means the program branched and there are two possible ways forward with execution. Other times, it will have more than two, such as in the case of a jump table.

```
>>> p.step()
>>> print "The path has", len(p.successors), "successors!"

# each successor is a path, keeping track of an execution history
>>> s = p.successors[0]
>>> assert s.addr_trace[-1] == p.addr

# and, of course, we can drill down further!
# alternate syntax: s.step() returns the same list as s.successors
>>> ss = s.step()[0].step()[0].step()[0]
>>> len(ss.addr_trace.hardcopy) == 4
```

To efficiently store information about path histories, angr employs a tree structure that resembles the actual symbolic execution tree. You should never have to worry about this, since through the magic of python we provide efficient accessors for information stored in the tree as it pertains to each stored historical property. The one thing you have to know is that this data structure doesn't allow efficient iteration through the historical lists in forward order - only in reverse order, from most recent to oldest. If you need to iterate or access items from these sequences starting from the beginning, you may access the `.hardcopy` property on them, which will extract the entirety of the property's history as a flat list for you to peruse at leisure.

For example: part of the history of a path is the *types* of jumps that occur. These are stored (as strings representing VEX exit type enums), in the `jumpkinds` attribute.

```
# recall: s is the path created when we stepped forward the initial path once
>>> print s.jumpkinds
<angr.path.JumpkindIter object at 0x7f8161e584d0>

>>> assert s.jumpkinds[-1] == 'Ijk_Call'
>>> print s.jumpkinds.hardcopy
['Ijk_Call']

# Don't do this! This will throw an exception
>>> # for jk in ss.jumpkinds: print jk

# Do this instead:
>>> for jk in reversed(ss.jumpkinds): print jk
Ijk_Call
Ijk_Call
Ijk_Boring
Ijk_Call

# Or, if you really need to iterate in forward order:
>>> for jk in ss.jumpkinds.hardcopy: print jk
Ijk_Call
Ijk_Boring
Ijk_Call
Ijk_Call
```

Here is a list of the properties in the path history:

Property	Description
Path.addr_trace	The addresses of basic blocks that have been executed so far, as integers
Path.trace	The SimSuccessors objects that have been generated so far, as strings
Path.targets	The targets of the jumps/successors that have been taken so far
Path.guards	The guard conditions that had to be satisfied in order to take the branch listed in Path.targets
Path.jumpkinds	The type of the exit from each basic block we took, as VEX struct strings
Path.events	A log of the events that have happened in symbolic execution
Path.actions	A filtering of Path.events to only include the actions taken by the exeution engine. See below.

Here are the different types of jumpkinds:

Type	Description
Ijk_Boring	A normal jump to an address.
Ijk_Call	A call to an address.
Ijk_Ret	A return.
Ijk_Sig*	Various signals.
Ijk_Sys*	System calls.
Ijk_NoHook	A jump out of an angr hook.

Merging Paths

Like states, paths can be merged. Truly understanding this requires concepts that will be explained in future sections, but in a nutshell, we can combine two paths that reached the same program point in different ways. For example, let's say that we have a branch:

```
# step until branch
p = b.factory.path()
p.step()
while len(p.successors) == 1:
    print 'step'
    p = p.successors[0]
    p.step()

print p
branched_left = p.successors[0]
branched_right = p.successors[1]
assert branched_left.addr != branched_right.addr

# Step the branches until they converge again
after_branched_left = branched_left.step()[0]
after_branched_right = branched_right.step()[0]
assert after_branched_left.addr == after_branched_right.addr

# this will merge both branches into a single path. Values in memory and registers
# will hold any possible values they could have held in either path.
merged = after_branched_left.merge(after_branched_right)
assert merged.addr == after_branched_left.addr and merged.addr == after_branched_right
.addr
```

Paths can also be unmerged later.

```
merged_successor = merged.step()[0].step()[0]
unmerged_paths = merged_successor.unmerge()

assert len(unmerged_paths) == 2
assert unmerged_paths[0].addr == unmerged_paths[1].addr
```

Non-entry point start

Sometimes, you might want to start the analysis of a program partway through the program. For example, you might be interested in what a specific part of a function does, but don't know how to (or don't want to) guide a path to that point. To handle this, we allow the creation of a path at any point in the program:

```
>>> st = b.factory.blank_state(addr=0x800f000)
>>> p = b.factory.path(st)

>>> assert p.addr == 0x800f000
```

At this point, all memory, registers, and so forth of the path are blank. In a nutshell, this means that they are fully symbolic and unconstrained, and execution can procede from this point as an overapproximation of what could happen on a real CPU. If you have outside knowledge about what the state should look like at this point, you can craft the blank state into a more precise description of machine state by adding constraints and setting the contents of memory, registers, and files.

SimActions Redux

The SimActions from deep within simuvex are exported for much easier access through the Path. Actions are part of the path's history (Path.actions), so the same rules as the other history items about iterating over them still apply.

When paths grow long, stored SimActions can be a serious source of memory consumption. Because of this, by default all but the most recent SimActions are discarded. To disable this behavior, enable the `TRACK_ACTION_HISTORY` state option.

There is a convenient interface for filtering through a potentially huge list of actions to find a specific write or read operation. Take a look at the api documentation for Path.filter_actions.

Bulk Execution and Exploration - Path Groups

Path groups are just a bunch of paths being executed at once. They are also the future.

Path groups let you wrangle multiple paths in a slick way. Paths are organized into “stashes”, which you can step forward, filter, merge, and move around as you wish. There are different kind of stashes, which are specified in Paths. This allows you to, for example, step two different stashes of paths at different rates, then merge them together.

Here are some basic examples of pathgroups capabilities:

```
>>> import angr


>>> p = angr.Project('examples/fauxware/fauxware', load_options={'auto_load_libs': Fal
se})
>>> pg = p.factory.path_group()
```

Exploring a path:

```
# While there are active path, we step
>>> while len(pg.active) > 0:
...     pg.step()

>>> print(pg)
<PathGroup with 1 deadended>
```

We now have a deadended path, let's see what we can do with it

```
>>> path = pg.deadended[0]
>>> print('Path length: {0} steps'.format(path.length))
Path length: 51 steps
```

Get path trace:

```
>>> print('Trace:')
>>> for step in path.trace:
...     print(step)
Trace:
<IRSB from 0x400580: 1 sat>
<IRSB from 0x400540: 1 sat>
<SimProcedure __libc_start_main from 0x1000030: 1 sat>
```

```
<IRSB from 0x4007e0: 1 sat>
<IRSB from 0x4004e0: 1 sat>
<IRSB from 0x4005ac: 1 sat 1 unsat>
<IRSB from 0x4005be: 1 sat>
<IRSB from 0x4004e9: 1 sat>
<IRSB from 0x400640: 1 sat 1 unsat>
<IRSB from 0x400660: 1 sat>
<IRSB from 0x4004ee: 1 sat>
<IRSB from 0x400880: 1 sat 1 unsat>
<IRSB from 0x4008af: 1 sat>
<IRSB from 0x4004f3: 1 sat>
<IRSB from 0x400825: 1 sat 1 unsat>
<IRSB from 0x400846: 1 sat>
<SimProcedure __libc_start_main from 0x1000040: 1 sat>
<IRSB from 0x40071d: 1 sat>
<IRSB from 0x400510: 1 sat>
<SimProcedure puts from 0x1000000: 1 sat>
<IRSB from 0x40073e: 1 sat>
<IRSB from 0x400530: 1 sat>
<SimProcedure read from 0x1000020: 1 sat>
<IRSB from 0x400754: 1 sat>
<IRSB from 0x400530: 1 sat>
<SimProcedure read from 0x1000020: 1 sat>
<IRSB from 0x40076a: 1 sat>
<IRSB from 0x400510: 1 sat>
<SimProcedure puts from 0x1000000: 1 sat>
<IRSB from 0x400774: 1 sat>
<IRSB from 0x400530: 1 sat>
<SimProcedure read from 0x1000020: 1 sat>
<IRSB from 0x40078a: 1 sat>
<IRSB from 0x400530: 1 sat>
<SimProcedure read from 0x1000020: 1 sat>
<IRSB from 0x4007a0: 1 sat>
<IRSB from 0x400664: 1 sat>
<IRSB from 0x400550: 1 sat>
<SimProcedure strcmp from 0x1000050: 1 sat>
<IRSB from 0x40068e: 2 sat>
<IRSB from 0x400692: 1 sat>
<IRSB from 0x4006eb: 1 sat>
<IRSB from 0x4007b3: 1 sat 1 unsat>
<IRSB from 0x4007bd: 1 sat>
<IRSB from 0x4006ed: 1 sat>
<IRSB from 0x400510: 1 sat>
<SimProcedure puts from 0x1000000: 1 sat>
<IRSB from 0x4006fb: 1 sat>
<IRSB from 0x4007c7: 1 sat>
<IRSB from 0x4007d3: 1 sat>
<SimProcedure __libc_start_main from 0x1000040: 1 sat>
```

Get constraints applied to the path:

```
>>> print('There are %d constraints.' % len(path.state.se.constraints))
There are 2 constraints.
```

Get memory state at the end of the traversal:

```
>>> print('rax: {0}'.format(path.state.regs.rax))
rax: <BV64 0x37>
>>> assert path.state.se.any_int(path.state.regs.rip) == path.addr  # regs are BitVect
ors
```

PathGroup.Explorer()

Pathgroups are supposed to replace `surveyors.Explorer`, being more clever and efficient. When launching path_group.Explore with a `find` argument, multiple paths will be launched and step until one of them finds one of the address we are looking for. Paths reaching the `avoided` addresses, if any, will be put into the `avoided` stash. If an active path reaches an interesting address, it will be stashed into the `found` stash, and the other ones will remain active. You can then explore the found path, or decide to discard it and continue with the other ones.

Let's look at a simple crackme example:

First, we load the binary.

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

Next, we create a path group.

```
>>> pg = p.factory.path_group()
```

Now, we symbolically execute until we find a path that matches our condition (i.e., the "win" condition).

```
>> pg.explore(find=lambda p: "Congrats" in p.state.posix.dumps(1))
<PathGroup with 1 active, 1 found>
```

Now, we can get the flag out of that state!

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


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

Pretty simple, isn't it?

Other examples can be found by browsing the examples.

TODO: STASHES

Stash types

Paths are put into different stashes during a PathGroup's execution. These are:

Stash	Description
active	This stash contains the paths that will be stepped by default (unless an alternate stash is specified for `path_group.step()`.
deadended	A path goes to the deadended stash when it cannot continue the execution for some reason, including no more valid instructions, unsat state of all of its successors, or an invalid instruction pointer.
found	A path goes to the found stash when the path group determines that it matches the condition passed to the `find` argument of `path_group.explore`.
avoided	A path goes to the avoided stash when the path group determines that it matches the condition passed to the `avoid` argument of `path_group.explore`.
pruned	When using `LAZY_SOLVES`, paths are not checked for satisfiability unless absolutely necessary. When a state is found to be unsat in the presence of `LAZY_SOLVES`, the path hierarchy is traversed to identify when, in its history, it initially became unsat. All paths that are descendent from that point (which will also be unsat, since a state cannot become un-unsat) are pruned and put in this stash.
errored	Paths are put in this stash when they cause a Python exception to be raised during execution. This implies a bug in angr or in your custom code (if any).
unconstrained	If the `save_unconstrained` option is provided to the PathGroup constructor, paths that are determined to be unconstrained (i.e., with the instruction pointer controlled by user data or some other source of symbolic data) are placed here.
unsat	If the `save_unsat` option is provided to the PathGroup constructor, paths that are determined to be unsatisfiable (i.e., they have constraints that are contradictory, like the input having to be both "AAAA" and "BBBB" at the same time) are placed here.

You can move paths between stashes by using the `path_group.move` function. This function accepts many options to control which paths are moved between which stashes.

Symbolic Execution - Surveyors

At heart, angr is a symbolic execution engine. angr exposes a standard way to write and perform dynamic symbolic execution: the `Surveyor` class. A `Surveyor` is the *engine* that drives symbolic execution: it tracks what paths are active, identifies which paths to step forward and which paths to prune, and optimizes resource allocation.

/!\ `Surveyors` are an old API that is rather unwieldy. It's recommended to use PathGroups instead. /!\

The `Surveyor` class is not meant to be used directly. Rather, it should be subclassed by developers to implement their own analyses. That being said, the most common symbolic analysis (i.e., "explore from A to B, trying to avoid C") has already been implemented in the `Explorer` class.

Explorer

`angr.surveyors.Explorer` is a `Surveyor` subclass that implements symbolic exploration. It can be told where to start, where to go, what to avoid, and what paths to stick to. It also tries to avoid getting stuck in loops.

In the end, one cannot be told what the `Explorer` is. You have to see it for yourself:

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

# By default, a Surveyor starts at the entry point of the program, with
# an exit created by calling `Project.initial_exit` with default arguments.
# This involves creating a default state using `Project.initial_state`.
# A custom SimExit, with a custom state, can be provided via the optional
# "start" parameter, or a list of them via the optional "starts" parameter.
>>> e = b.surveyors.Explorer()

# Now we can take a few steps! Printing an Explorer will tell you how
# many active paths it currently has.
>>> print e.step()

# You can use `Explorer.run` to step multiple times.
>>> print e.run(10)

# Or even forever. By default, an Explorer will not stop running until
# it runs out of paths (which will likely be never, for most programs),
# so be careful. In this case, we should be ok because the program does
# not loop.
>>> e.run()

# We can see which paths are active (running), and which have deadended
# (i.e., provided no valid exits), and which have errored out. Note that,
# in some instances, a given path could be in multiple lists (i.e., if it
# errored out *and* did not produce any valid exits)
>>> print "%d paths are still running" % len(e.active)
>>> print "%d paths are backgrounded due to lack of resources" % len(e.spilled)
>>> print "%d paths are suspended due to user action" % len(e.suspended)
>>> print "%d paths had errors" % len(e.errored)
>>> print "%d paths deadended" % len(e.deadended)
```

So far, everything we have discussed applies to all `Surveyors`. However, the nice thing about an Explorer is that you can tell it to search for, or avoid certain blocks. For example, in the `fauxware` sample, we can try to find the "authentication success" function while avoiding the "authentication failed" function.

```
# This creates an Explorer that tries to find 0x4006ed (successful auth),
# while avoiding 0x4006fd (failed auth) or 0x4006aa (the authentication
# routine). In essense, we are looking for a backdoor.
>>> e = b.surveyors.Explorer(find=(0x4006ed,), avoid=(0x4006aa,0x4006fd))
>>> e.run()

# Print our found backdoor, and how many paths we avoided!
>>> if len(e.found) > 0:
...     print "Found backdoor path:", e.found[0]

>>> print "Avoided %d paths" % len(e.avoided)
```

Some helper properties are provided for easier access to paths from ipython:

```
>>> print "The first found path is", e._f
# Also available are _d (deadended), _spl (spilled), and _e (errored)
```

Caller

The `Caller` is a surveyor that handles calling functions to make it easier to figure out what the heck they do. It can be used as so:

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

# get the state ready, and grab our username and password symbolic expressions for lat
er
# checking. Here, we'll cheat a bit since we know that username and password should bo
th
# be 8 chars long
>>> p = b.factory.path()
>>> username = p.state.memory.load(0x1000, 9)
>>> password = p.state.memory.load(0x2000, 9)

# call the authenticate function with *username being 0x1000 and *password being 0x2000

>>> c = b.surveyors.Caller(0x400664, (0x1000,0x2000), start=p)

# look at the different paths that can return. This should print 3 paths:
>>> print tuple(c.iter_returns())

# two of those paths return 1 (authenticated):
>>> print tuple(c.iter_returns(solution=1))

# now let's see the required username and password to reach that point. `c.map_se`
# calls state.se.any_n_str (or whatever other function is provided) for the provided
# arguments, on each return state. This example runs state.se.any_n_str(credentials, 1
0)
>>> credentials = username.concat(password)
>>> tuple(c.map_se('any_n_str', credentials, 10, solution=1))

# you can see the secret password "SOSNEAKY" in the first tuple!
```

Caller is a pretty powerful tool. Check out the comments on the various functions for more usage info! HOWEVER, there is a much easier tool you can use to call functions, called `callable`. This is described elsewhere in the docs.

Interrupting Surveyors

A surveyor saves its internal state after every tick. In ipython, you should be able to interrupt a surveyor with `Ctrl-C`, and then check what results it has so far, but that's a pretty ugly way of doing it. There are two official ways of doing this cleanly: `SIGUSR1` and `SIGUSR2`.

If you send `SIGUSR1` to a python process running a surveyor, it causes the main loop in `Surveyor.run()` to terminate at the end of the current `Surveyor.step()`. You can then analyze the result. To continue running the surveyor, call `angr.surveyor.resume_analyses()` (to clear the "signalled" flag) and then call the surveyor's `run()` function. Since `SIGUSR1` causes `run()` to return, this is rarely useful in a scripted analysis, as the rest of the program will run after `run()` returns. Instead, `SIGUSR1` is meant to provide a clean alternative to `Ctrl-C`.

Sending SIGUSR2 to the python process, on the other hand, causes `run()` to invoke an `ipdb` breakpoint after every `step()`. This allows you to debug, then continue your program. Make sure to run `angr.surveyor.disable_singlestep()` before continuing to clear the "signalled" flag.

Working with Data and Conventions

Frequently, you'll want to access structured data from the program you're analyzing. angr has several features to make this less of a headache.

Working with types

SimuVEX has a system for representing types. These SimTypes are found in `simuvex/s_type.py` - an instance of any of these classes represents a type. Many of the types are incomplete unless they are supplimented with a SimState - their size depends on the architecture you're running under. You may do this with `ty.with_state(state)`, which returns a copy of itself, with the state specified.

SimuVEX also has a light wrapper around `pycparser`, which is a C parser. This helps with getting instances of type objects:

```
>>> import simuvex

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

>>> simuvex.s_type.parse_type('char **')
char**

>>> simuvex.s_type.parse_type('struct aa {int x; long y;}')
struct aa

>>> simuvex.s_type.parse_type('struct aa {int x; long y;}').fields
OrderedDict([('x', int), ('y', long)])
```

Additionally, you may parse C defininitions and have them returned to you in a dict:

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

>>> 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
>>> simuvex.s_type.parse_defns("int x(int y, double z);")
{'x': (int, double) -> int}
```

And finally, you can register struct definitions for future use:

```
>>> simuvex.s_type.define_struct('struct abcd { int x; int y; }')
>>> simuvex.s_type.parse_type('struct abcd')
struct abcd
```

These type objects aren't all that useful on their own, but they can be passed to other parts of angr to specify data types.

Accessing typed data from memory

If you're reading this book in order, you'll recall that you can retrieve data from memory with `state.memory.load(addr, len, endness=endness)`. This can get to be a little cumbersome when working with structures, strings, etc. Instead, there is an alternate interface in `state.mem`, the SimMemView. This allows you to specify the type of the data you're looking at.

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

>>> s.mem[0x601048].int
<int (32 bits) <BV32 0x4008d0> 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
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'
```

The interface works like this:

- You first use [array index notation] to specify the address you'd like to load from
- If at that address is a pointer, you may access the `deref` property to return a SimMemView at the address present in memory.
- You then specify a type for the data by simply accesing a property of that name. For a list of supported types, look at `state.mem.types`.
- You can then *refine* the type. Any type may support any refinement it likes. Right now the only refinements supported are that you may access any member of a struct by its member name, and you may index into a string or array to access that element.
- If the address you specified initially points to an array of that type, you can say `.array(n)` to view the data as an array of n elements.
- Finally, extract the structured data with `.resolved` or `.concrete`. `.resolved` will return bitvector values, while `.concrete` will return integer, string, array, etc values, whatever best represents the data.
- Alternately, you may store a value to memory, by assigning to the chain of properties that you've constructed. Note that because of the way python works, `x =`

`s.mem[...].prop; x = val` will NOT work, you must say `s.mem[...].prop = val`.

If you define a struct using `s_type.define_struct`, you can access it here as a type:

```
>>> 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>
```

Working with Calling Conventions

A calling convention is the specific means by which code passes arguments and return values through function calls. While angr comes with a large number of pre-built calling conventions, and a lot of logic for refining calling conventions for specifc circumstances (e.g. floating point arguments need to be stored in different locations, it gets worse from there), it will inevitably be insufficient to describe all possible calling conventions a compiler could generate. Because of this, you can *customize* a calling convention by describing where the arguments and return values should live.

angr's abstraction of calling conventions lives in Simuvex as SimCC. You can construct new SimCC instances through the angr object factory, with `b.factory.cc(...)`.

- Pass as the `args` keyword argument a list of argument storage locations
- Pass as the `ret_val` keyword argument the location where the return value should be stored
- Pass as the `func_ty` keyword argument a SymType for the function prototype.
- Pass it none of these things to use a sane default for the current architecture!

To specify a value location for the `args` or `ret_val` parameters, use instances of the `SimRegArg` or `SimStackArg` classes. You can find them in the factory - `b.factory.cc.Sim*Arg`. Register arguments should be instanciated with the name of the register you're storing the value in, and the size of the register in bytes. Stack arguments should be instanciated with the offset from the stack pointer *at the time of entry into the function* and the size of the storage location, in bytes.

Once you have a SimCC object, you can use it along with a SimState object to extract or store function arguments more cleanly. Take a look at the API documentation for details. Alternately, you can pass it to an interface that can use it to modify its own behavior, like `b.factory.call_state`, or...

Callables

Callables are a Foreign Functions Interface (FFI) for symbolic execution. Basic callable usage is to create one with `myfunc = b.factory.callable(addr)`, and then call it! `result = myfunc(args, ...)` When you call the callable, angr will set up a `call_state` at the given address, dump the given arguments into memory, and run a `path_group` based on this state until all the paths have exited from the function. Then, it merges all the result states together, pulls the return value out of that state, and returns it.

All the interaction with the state happens with the aid of a `SimCC`, to tell where to put the arguments and where to get the return value. By default, it uses a sane default for the archetecture, but if you'd like to customize it, you can pass a `SimCC` object in the `cc` keyword argument when constructing the callable.

You can pass symbolic data as function arguments, and everything will work fine. You can even pass more complicated data, like strings, lists, and structures as native python data (use tuples for structures), and it'll be serialized as cleanly as possible into the state. If you'd like to specify a pointer to a certain value, you can wrap it in a `PointerWrapper` object, available as `b.factory.callable.PointerWrapper`. The exact semantics of how pointer-wrapping work are a little confusing, but they can be boiled down to "unless you specify it with a PointerWrapper or a specific SimArrayType, nothing will be wrapped in a pointer automatically unless it gets to the end and it hasn't yet been wrapped in a pointer yet and the original type is a string, array, or tuple." The relevant code is actually in SimCC - it's the `setup_callsite` function.

If you don't care for the actual return value of the call, you can say `func.perform_call(arg, ...)`, and then the properties `func.result_state` and `func.result_path_group` will be populated. They will actually be populated even if you call the callable normally, but you probably care about them more in this case!

Analyses

angr's goal is to make it easy to carry out useful analyses on binary programs. This section will discuss how to run and create these analyses.

Built-in Analyses

angr comes with several built-in analyses:

Name	Description
CFGFast	Constructs a fast *Control Flow Graph* of the program. `b.analyses.CFG()` is what you want.
CFGAccurate	Constructs an accurate *Control Flow Graph* of the program. The simple way to do is via `b.analyses.CFGAccurate()` .
VFG	Performs VSA on every function of the program, creating a *Value Flow Graph* and detecting stack variables.
DDG	Calculates a data dependency graph, allowing one to determine what statements a given value depends on.
DFG	Constructs a *Data Flow Graph* for each basic block present in the CFG
BackwardSlice	Computes a backward slice of a program w.r.t. a certain target.
More!	angr has quite a few analyses, most of which work! If you'd like to know how to use one, please submit an issue requesting documentation.

VFG

TODO

DDG

TODO

Running Analyses

Now that you understand how to load binaries in angr, and have some idea of angr's internals, we can discuss how to carry out analyses! angr provides a standardized interface to perform analyses.

Resilience

Analyses can be written to be resilient, and catch and log basically any error. These errors, depending on how they're caught, are logged to the `errors` or `named_errors` attribute of the analysis. However, you might want to run an analysis in "fail fast" mode, so that errors are not handled. To do this, the `fail_fast` keyword argument can be passed into `analyze`.

```
>> b.analyses.CFG(fail_fast=True)
```

Creating Analyses

An analysis can be created by subclassing the `Analysis` class. In this section, we'll create a mock analysis to show off the various features. Let's start with something simple:

```
>>> import angr

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

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

This is a quite simple analysis -- it takes an option, and stores it. Of course, it's not useful, but what can you do? Let's see how to call:

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

Working with projects

Via some python magic, your analysis will automatically have the project upon which you are running it under the `self.project` property. Use this to interact with your project and analyze it!

```
>>> 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')
>>> b = angr.Project("/bin/true")

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

Naming Analyses

The `register_analysis` call is what actually adds the analysis to angr. Its arguments are the actual analysis class and the name of the analysis. The name is how it appears under the `project.analyses` object. Usually, you should use the same name as the analysis class, but if you want to use a shorter name, you can.

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

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

After this, you can call this analysis using it's specified name. For example, `b.analyses.FuncSize()`.

If you've registered a new analysis since loading the project, refresh the list of registered analyses on your project with `b.analyses.reload_analyses()`.

Analysis Resilience

Sometimes, your (or our) code might suck and analyses might throw exceptions. We understand, and we also understand that oftentimes a partial result is better than nothing. This is specifically true when, for example, running an analysis on all of the functions in a program. Even if some of the functions fails, we still want to know the results of the functions that do not.

To facilitate this, the `Analysis` base class provides a resilience context manager under `self._resilience`. Here's an example:

```
>>> 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"
```

The context manager catches any exceptions thrown and logs them (as a tuple of the exception type, message, and traceback) to `self.errors`. These are also saved and loaded when the analysis is saved and loaded (although the traceback is discarded, as it is not picklable).

You can tune the effects of the resilience with two optional keyword parameters to `self._resilience()`.

The first is `name`, which affects where the error is logged. By default, errors are placed in `self.errors`, but if `name` is provided, then instead the error is logged to `self.named_errors`, which is a dict mapping `name` to a list of all the errors that were caught under that name. This allows you to easily tell where thrown without examining its traceback.

The second argument is `exception`, which should be the type of the exception that `_resilience` should catch. This defaults to `Exception`, which handles (and logs) almost anything that could go wrong. You can also pass a tuple of exception types to this option, in which case all of them will be caught.

Using `_resilience` has a few advantages:

1. Your exceptions are gracefully logged and easily accessible afterwards. This is really nice for writing testcases.
2. When creating your analysis, the user can pass `fail_fast=True`, which transparently disable the resilience, which is really nice for manual testing.
3. It's prettier than having `try` / `except` everywhere.

Have fun with analyses! Once you master the rest of angr, you can use analyses to understand anything computable!

CFGAccurate

Here we describe angr's CFGAccurate analysis in details, as well as some important concepts like context sensitivity and Function Manager of angr.

General ideas

A basic analysis that one might carry out on a binary is a Control Flow Graph. A CFG is a graph with (conceptually) basic blocks as nodes and jumps/calls/rets/etc as edges.

In angr, there are two types of CFG that can be generated: a fast CFG (CFGFast) and an accurate CFG (CFGAccurate). As their names suggested, generating a fast CFG is usually much faster than generating the accurate one. In general, CFGFast is what you need. This page discusses CFGAccurate.

An accurate CFG can be constructed by doing:

```
>>> 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)
```

Of course, there are several options for customized CFGs.

Option	Description
context_sensitivity_level	This sets the context sensitivity level of the analysis. See the context sensitivity level section below for more information. This is 1 by default.
starts	A list of addresses, to use as entry points into the analysis.
avoid_runs	A list of addresses to ignore in the analysis.
call_depth	Limit the depth of the analysis to some number calls. This is useful for checking which functions a specific function can directly jump to (by setting `call_depth` to 1).
initial_state	An initial state can be provided to the CFG, which it will use throughout its analysis.
keep_state	To save memory, the state at each basic block is discarded by default. If `keep_state` is True, the state is saved in the CFGNode.
enable_symbolic_back_traversal	Whether to enable an intensive technique for resolving indirect jumps
enable_advanced_backward_slicing	Whether to enable another intensive technique for resolving direct jumps
more!	Examine the docstring on b.analyses.CFGAccurate for more up-to-date options

Context Sensitivity Level

angr constructs a CFG by executing every basic block and seeing where it goes. This introduces some challenges: a basic block can act differently in different *contexts*. For example, if a block ends in a function return, the target of that return will be different, depending on different callers of the function containing that basic block.

The context sensitivity level is, conceptually, the number of such callers to keep on the callstack. To explain this concept, let's look at the following code:

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

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

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

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

The above sample has four call chains: `main>alpha>puts`, `main>alpha>error>puts` and `main>beta>puts`, and `main>beta>error>puts`. While, in this case, angr can probably execute both call chains, this becomes unfeasible for larger binaries. Thus, angr executes the blocks with states limited by the context sensitivity level. That is, each function is re-analyzed for each unique context that it is called in.

For example, the `puts()` function above will be analyzed with the following contexts, given different context sensitivity levels:

Level	Meaning	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`

The upside of increasing the context sensitivity level is that more information can be gleamed from the CFG. For example, with context sensitivity of 1, the CFG will show that, when called from `alpha`, `puts` returns to `alpha`, when called from `error`, `puts` returns to `error`, and so forth. With context sensitivity of 0, the CFG simply shows that `puts`

returns to `alpha`, `beta`, and `error`. This, specifically, is the context sensitivity level used in IDA. The downside of increasing the context sensitivity level is that it exponentially increases the analysis time.

Using the CFG

The CFG, at its core, is a NetworkX di-graph. This means that all of the normal NetworkX APIs are available:

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

The nodes of the CFG graph are instances of class `CFGNode`. Due to context sensitivity, a given basic block can have multiple nodes in the graph (for multiple contexts).

```
# 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 " + s
tr(node.addr) for node,jumpkind in cfg.get_successors_and_jumpkind(entry_node) ]
```

Viewing the CFG

Control-flow graph rendering is a hard problem. angr does not provide any built-in mechanism for rendering the output of a CFG analysis, and attempting to use a traditional graph rendering library, like matplotlib, will result in an unusable image.

One solution for viewing angr CFGs is found in axt's angr-utils repository.

Shared Libraries

The CFG analysis does not distinguish between code from different binary objects. This means that by default, it will try to analyze control flow through loaded shared libraries. This is almost never intended behavior, since this will extend the analysis time to several days,

probably. To load a binary without shared libraries, add the following keyword argument to the `Project` constructor: `load_options={'auto_load_libs': False}`

Function Manager

The CFG result produces an object called the *Function Manager*, accessible through `cfg.kb.functions`. The most common use case for this object is to access it like a dictionnary. It maps addresses to `Function` objects, which can tell you properties about a function.

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

Functions have several important properties!

- `entry_func.block_addrs` is a set of addresses at which basic blocks belonging to the function begin.
- `entry_func.blocks` is the set of basic blocks belonging to the function, that you can explore and disassemble using capstone.
- `entry_func.string_references()` returns a list of all the constant strings that were referred to at any point in the function. They are formatted as `(addr, string)` tuples, where addr is the address in the binary's data section the string lives, and string is a python string that contains the value of the string.
- `entry_func.returning` is a boolean value signifying whether or not the function can return. `False` indicates that all paths do not return.
- `entry_func.callable` is an angr Callable object referring to this function. You can call it like a python function with python arguments and get back an actual result (may be symbolic) as if you ran the function with those arguments!
- `entry_func.transition_graph` is a NetworkX DiGraph describing control flow within the function itself. It resembles the control-flow graphs IDA displays on a per-function level.
- `entry_func.name` is the name of the function.
- `entry_func.has_unresolved_calls` and `entry.has_unresolved_jumps` have to do with detecting imprecision within the CFG. Sometimes, the analysis cannot detect what the possible target of an indirect call or jump could be. If this occurs within a function, that function will have the appropriate `has_unresolved_*` value set to `True`.
- `entry_func.get_call_sites()` returns a list of all the addresses of basic blocks which end in calls out to other functions.
- `entry_func.get_call_target(callsite_addr)` will, given `callsite_addr` from the list of call site addresses, return where that callsite will call out to.
- `entry_func.get_call_return(callsite_addr)` will, given `callsite_addr` from the list of call site addresses, return where that callsite should return to.

and many more !

Backward Slicing

A *program slice* is a subset of statements that is obtained from the original program, usually by removing zero or more statements. Slicing is often helpful in debugging and program understanding. For instance, it's usually easier to locate the source of a variable on a program slice.

A backward slice is constructed from a *target* in the program, and all data flows in this slice end at the *target*.

angr has a built-in analysis, called `BackwardSlice`, to construct a backward program slice. This section will act as a how-to for angr's `BackwardSlice` analysis, and followed by some in-depth discussion over the implementation choices and limitations.

First Step First

To build a `BackwardSlice`, you will need the following information as input.

- **Required** CFG. A control flow graph (CFG) of the program. This CFG must be an accurate CFG (CFGAccurate).
- **Required** Target, which is the final destination that your backward slice terminates at.
- **Optional** CDG. A control dependence graph (CDG) derived from the CFG. angr has a built-in analysis `CDG` for that purpose.
- **Optional** DDG. A data dependence graph (DDG) built on top of the CFG. angr has a built-in analysis `DDG` for that purpose.

A `BackwardSlice` can be constructed with the following code:

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

# 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
```

Sometimes it's difficult to get a data dependence graph, or you may simply want build a program slice on top of a CFG. That's basically why DDG is an optional parameter. You can build a `BackwardSlice` solely based on CFG by doing:

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

Using The `BackwardSlice` Object

Before go ahead and using `BackwardSlice` object, you should be noticed that the design of this class is pretty arbitrary right now, and it is by no means to be stable in the near future. I'll try my best to keep this documentation updated when a refactoring/redesigning occurs.

Members

After construction, a `BackwardSlice` has the following members that describe a program slice:

Member	Mode	Meaning
runs_in_slice	CFG-only	A `networkx.DiGraph` instance showing addresses of blocks and SimProcedures in the program slice, as well as transitions between them
cfg_nodes_in_slice	CFG-only	A `networkx.DiGraph` instance showing CFGNodes in the program slice and transitions in between
chosen_statements	With DDG	A dict mapping basic block addresses to lists of statement IDs that are part of the program slice
chosen_exits	With DDG	A dict mapping basic block addresses to a list of “exits”. Each exit in the list is a valid transition in the program slice

Each “exit” in `chosen_exit` is a tuple including a statement ID and a list of target addresses. For example, an “exit” might look like the following:

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

If the “exit” is the default exit of a basic block, it’ll look like the following:

```
(“default”, [ 0x400085 ])
```

Export an Annotated Control Flow Graph

TODO

User-friendly Representation

Take a look at `BackwardSlice.dbg_repr()` !

TODO

Implementation Choices

TODO

Limitations

TODO

Completeness

TODO

Soundness

TODO

Speed considerations

The speed of angr as an analysis tool or emulator is greatly handicapped by the fact that it is written in python. Regardless, there are a lot of optimizations and tweaks you can use to make angr faster.

General tips

- *Use pypy*. Pypy is an alternate python interpreter that performs optimized jitting of python code. In our tests, it's a 10x speedup out of the box.
- *Don't load shared libraries unless you need them*. The default setting in angr is to try at all costs to find shared libraries that are compatible with the binary you've loaded, including loading them straight out of your OS libraries. This can complicate things in a lot of scenarios. If you're performing an analysis that's anything more abstract than bare-bones symbolic execution, you might want to make the tradeoff of sacrificing accuracy for tractability. angr does a reasonable job of making sane things happen when library calls to functions that don't exist try to happen.
- *Use hooking and SimProcedures*. If you're enabling shared libraries, then you definitely want to have SimProcedures written for any complicated library function you're jumping into. If there's no autonomy requirement for this project, you can often isolate individual problem spots where analysis hangs up and summarize them with a hook.
- *Use SimInspect*. SimInspect is the most underused and one of the most powerful features of angr. You can hook and modify almost any behavior of angr, including memory index resolution (which is often the slowest part of any angr analysis).
- *Write a concretization strategy*. A more powerful solution to the problem of memory index resolution is a concretization strategy.
- *Use the Replacement Solver*. You can enable it with the `simuvex.o.REPLACEMENT_SOLVER` state option. The replacement solver allows you to specify AST replacements that are applied at solve-time. If you add replacements so that all symbolic data is replaced with concrete data when it comes time to do the solve, the runtime is greatly increased. The API for adding a replacement is `state.se._solver.add_replacement(old, new)`. The replacement solver is a bit finicky, so there are some gotchas, but it'll definitely help.

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

- *Use the unicorn engine*. If you have unicorn engine installed, Simuvex can be built to take advantage of it for concrete emulation. To enable it, add the options in the set `simuvex.o.unicorn` to your state. Keep in mind that while most items under `simuvex.o` are individual options, `simuvex.o.unicorn` is a bundle of options, and is thus a set. *NOTE*: At time of writing the official version of unicorn engine will not work with angr - we have a lot of patches to it to make it work well with angr. They're all pending pull requests at this time, so sit tight. If you're really impatient, ping us about uploading our fork!
- *Enable fast memory and fast registers*. The state options `simuvex.o.FAST_MEMORY` and `simuvex.o.FAST_REGISTERS` will do this. These will switch the memory/registers over to a less intensive memory model that sacrifices accuracy for speed. TODO: document the specific sacrifices. Should be safe for mostly concrete access though. NOTE: not compatible with concretization strategies.
- *Concretize your input ahead of time*. This is the approach taken by driller. Before execution begins, we fill state.posix.files[0] with symbolic data representing the input, then constrain that symbolic data to what we want the input to be, then set a concrete file size (state.posix.files[0].size = whatever). If you don't require any tracking of the data coming from stdin, you can forego the symbolic part and just fill it with concrete data. If there are other sources of input besides standard input, do the same for those.
- *Use the afterburner*. While using unicorn, if you add the `UNICORN_THRESHOLD_CONCRETIZATION` state option, SimuVEX will accept thresholds after which it causes symbolic values to be concretized so that execution can spend more time in Unicorn. Specifically, the following thresholds exist:
  - `state.se.unicorn.concretization_threshold_memory` - this is the number of times a symbolic variable, stored in memory, is allowed to kick execution out of Unicorn before it is forcefully concretized and forced into Unicorn anyways.
  - `state.se.unicorn.concretization_threshold_registers` - this is the number of times a symbolic variable, stored in a register, is allowed to kick execution out of Unicorn before it is forcefully concretized and forced into Unicorn anyways.
  - `state.se.unicorn.concretization_threshold_instruction` - this is the number of times that any given instruction can force execution out of Unicorn (by running into symbolic data) before any symbolic data encountered at that instruction is concretized to force execution into Unicorn.

  You can get further control of what is and isn't concretized with the following sets:
  - `state.se.unicorn.always_concretize` - a set of variable names that will always be concretized to force execution into unicorn (in fact, the memory and register thresholds just end up causing variables to be added to this list).
  - `state.se.unicorn.never_concretize` - a set of variable names that will never be concretized and forced into Unicorn under any condition.

- `state.se.unicorn.concretize_at` - a set of instruction addresses at which data should be concretized and forced into Unicorn. The instruction threshold causes addresses to be added to this set.

Once something is concretized with the afterburner, you will lose track of that variable. The state will still be consistent, but you'll lose dependencies, as the stuff that comes out of Unicorn is just concrete bits with no memory of what variables they came from. Still, this might be worth it for the speed in some cases, if you know what you want to (or do not want to) concretize.

angr examples

To help you get started with angr, we've created several examples. These mostly stem from CTF problems solved with angr by Shellphish. Enjoy!

Introduction example - Fauxware

This is a basic script that explains how to use angr to symbolically execute a program and produce concrete input satisfying certain conditions.

Binary, source, and script are found here.

CTF Problems

ReverseMe example: TUMCTF 2016 - zwiebel

Script author: Fish

Script runtime: 2 hours 31 minutes with Pypy and Unicorn - expect much longer with CPython only

We were given a binary that unpacks and executes a small amount of code each time, which checks the input. You may refer to other writeup about the internals of this binary. I didn't reverse too much since I was pretty confident that angr is able to solve it :-)

This example shows how to enable Unicorn support and self-modification support in angr. Unicorn support is essential to solve this challenge within a reasonable amount of time - simulating the unpacking code is *very* slow.

The long-term goal of optimizing angr is to execute this script within 10 minutes. Pretty ambitious :P

Here is the binary and the script.

ReverseMe example: HackCon 2016 - angry-reverser

Script author: Stanislas Lejay (github: @P1kachu)

Script runtime: ~31 minutes

Here is the binary and the script

ExploitMe example: SecuInside 2016 Quals - mbrainfuzz

Script author: nsr (nsr@tasteless.eu)

Script runtime: ~15 seconds per binary

Originally, a binary was given to the ctf-player by the challenge-service, and an exploit had to be crafted automatically. Four sample binaries, obtained during the ctf, are included in the example. All binaries followed the same format; the command-line argument is validated in a bunch of functions, and when every check succeeds, a memcpy() resulting into a stack-based bufferoverflow is executed. Angr is used to find the way through the binary to the memcpy() and to generate valid inputs to every checking function individually.

Both sample binaries and the script are located here and additional information be found at the author's Write-Up.

ReverseMe example: SecurityFest 2016 - fairlight

Script author: chuckleberryfinn (github: @chuckleberryfinn)

Script runtime: ~20 seconds

A simple reverse me that takes a key as a command line argument and checks it against 14 checks. Possible to solve the challenge using angr without reversing any of the checks.

Here is the binary and the script

ReverseMe example: DEFCON Quals 2016 - baby-re

- Script 0

  author: David Manouchehri (github: @Manouchehri)

  Script runtime: 8 minutes

- Script 1

  author: Stanislas Lejay (github: @P1kachu)

  Script runtime: 11 sec

Here is the binary and the scripts:

- script0
- script1

ReverseMe example: Google CTF - Unbreakable Enterprise Product Activation (150 points)

Script 0 author: David Manouchehri (github: @Manouchehri)

Script runtime: 4.5 sec

Script 1 author: Adam Van Prooyen (github: @docileninja)

Script runtime: 6.7 sec

A Linux binary that takes a key as a command line argument and check it against a series of constraints.

Challenge Description:

> We need help activating this product -- we've lost our license key :(
>
> You're our only hope!

Here are the binary and scripts: script 0, script_1

ReverseMe example: WhiteHat Grant Prix Global Challenge 2015 - Re400

Author: Fish Wang (github: @ltfish)

Script runtime: 5.5 sec

A Windows binary that takes a flag as argument, and tells you if the flag is correct or not.

"I have to patch out some checks that are difficult for angr to solve (e.g., it uses some bytes of the flag to decrypt some data, and see if those data are legit Windows APIs). Other than that, angr works really well for solving this challenge."

The binary and the script.

ReverseMe example: EKOPARTY CTF 2015 - rev 100

Author: Fish Wang (github: @ltfish)

Script runtime: 5.5 sec

This is a painful challenge to solve with angr. I should have done things in a smarter way.

Here is the binary and the script.

ReverseMe example: ASIS CTF Finals 2015 - fake

Author: Fish Wang (github: @ltfish)

Script runtime: 1 min 57 sec

The solution is pretty straight-forward.

The binary and the script.

ReverseMe example: ASIS CTF Finals 2015 - license

Author: Fish Wang (github: @ltfish)

Script runtime: 3.6 sec

This is a good example that showcases the following:

- Create a custom file, and load it during symbolic execution.
- Create an inline call to SimProcedure `strlen`, and use it to determine the length of a string in memory - even if the string may not be null-terminated.
- `LAZY_SOLVES` should be disabled sometimes to avoid creating too many paths.

Here are the binary and the script.

ReverseMe example: Defcamp CTF Qualification 2015 - Reversing 100

Author: Fish Wang (github: @ltfish)

angr solves this challenge with almost zero user-interference.

See the script and the binary.

ReverseMe example: Defcamp CTF Qualification 2015 - Reversing 200

Author: Fish Wang (github: @ltfish)

angr solves this challenge with almost zero user-interference. Veritesting is required to retrieve the flag promptly.

The script and the binary. It takes a few minutes to run on my laptop.

ReverseMe example: MMA CTF 2015 - HowToUse

Author: Andrew Dutcher (github: @rhelmot)

We solved this simple reversing challenge with angr, since we were too lazy to reverse it or run it in Windows. The resulting script shows how we grabbed the flag out of the DLL.

CrackMe example: MMA CTF 2015 - SimpleHash

Author: Chris Salls (github: @salls)

This crackme is 95% solveable with angr, but we did have to overcome some difficulties. The script describes the difficulties that were encountered and how we worked around them. The binary can be found here.

ReverseMe example: FlareOn 2015 - Challenge 10

Author: Fish Wang (github: @ltfish)

angr acts as a binary loader and an emulator in solving this challenge. I didn't have to load the driver onto my Windows box.

The script demonstrates how to hook at arbitrary program points without affecting the intended bytes to be executed (a zero-length hook). It also shows how to read bytes out of memory and decode as a string.

By the way, here is the link to the intended solution from FireEye.

ReverseMe example: FlareOn 2015 - Challenge 2

Author: Chris Salls (github: @salls)

This reversing challenge is simple to solve almost entirely with angr, and a lot faster than trying to reverse the password checking function. The script is here

ReverseMe example: FlareOn 2015 - Challenge 5

Author: Adrian Tang (github: @tangabc)

Script runtime: 2 mins 10 secs

This is another reversing challenge from the FlareOn challenges.

"The challenge is designed to teach you about PCAP file parsing and traffic decryption by reverse engineering an executable used to generate it. This is a typical scenario in our malware analysis practice where we need to figure out precisely what the malware was doing on the network"

For this challenge, the author used angr to represent the desired encoded output as a series of constraints for the SAT solver to solve for the input.

For a detailed write-up please visit the author's post here and you can also find the solution from the FireEye here

ReverseMe example: 0ctf 2016 - momo

Author: Fish Wang (github: @ltfish), ocean (github: @ocean1)

This challenge is a movfuscated binary. To find the correct password after exploring the binary with Qira it is possible to understand how to find the places in the binary where every character is checked using capstone and using angr to load the binary and brute-force the single characters of the flag. Be aware that the script is really slow. Runtime: > 1 hour.

ReverseMe example: 0ctf quals 2016 - trace

Author: WGH (wgh@bushwhackers.ru)

Script runtime: 1 min 50 secs (CPython 2.7.10), 1 min 12 secs (PyPy 4.0.1)

In this challenge we're given a text file with trace of a program execution. The file has two columns, address and instruction executed. So we know all the instructions being executed, and which branches were taken. But the initial data is not known.

Reversing reveals that a buffer on the stack is initialized with known constant string first, then an unknown string is appended to it (the flag), and finally it's sorted with some variant of quicksort. And we need to find the flag somehow.

angr easily solves this problem. We only have to direct it to the right direction at every branch, and the solver finds the flag at a glance.

CrackMe example: Layer7 CTF 2015 - Windows challenge OnlyOne

Author: Fish Wang (github: @ltfish)

We solved this crackme with angr's help. (Fish: This is my first time solving a reversing challenge without understanding what's going on.) The challenge binary is here, and the solving script here.

The solving script demonstrates the following:

- How to load a Windows binary (no difference than an ELF).
- How to use hook to replace arbitrary code in a loaded program.

- How to use Explorer to perform a symbolic exploration (although everyone else thinks PathGroup is the future).
- How to enable Veritesting, and why it is useful.

CrackMe example: Whitehat CTF 2015 - Crypto 400

Author: Yan Shoshitaishvili (github: @Zardus)

We solved this crackme with angr's help. The resulting script will help you understand how angr can be used for crackme assistance. You can find this script here and the binary here.

CrackMe example: CSAW CTF 2015 Quals - Reversing 500, "wyvern"

Author: Andrew Dutcher (github: @rhelmot)

angr can outright solve this challenge with very little assistance from the user. The script to do so is here and the binary is here.

CrackMe example: 9447 CTF 2015 - Reversing 330, "nobranch"

Author: Andrew Dutcher (github: @rhelmot)

angr cannot currently solve this problem natively, as the problem is too complex for z3 to solve. Formatting the constraints to z3 a little differently allows z3 to come up with an answer relatively quickly. (I was asleep while it was solving, so I don't know exactly how long!) The script for this is here and the binary is here.

CrackMe example: ais3_crackme

Author: Antonio Bianchi, Tyler Nighswander

ais3_crackme has been developed by Tyler Nighswander (tylerni7) for ais3 summer school. It is an easy crackme challenge, checking its command line argument.

ReverseMe: Modern Binary Exploitation - CSCI 4968

Author: David Manouchehri (GitHub @Manouchehri)

This folder contains scripts used to solve some of the challenges with angr. At the moment it only contains the examples from the IOLI crackme suite, but eventually other solutions will be added.

CrackMe example: Android License Check

Author: Bernhard Mueller (GitHub @b-mueller)

A native binary for Android/ARM that validates a license key passed as a command line argument. It was created for the symbolic execution tutorial in the OWASP Mobile Testing Guide.

Exploitation Examples

Beginner Exploitation example: strcpy_find

Author: Kyle Ossinger (github: @k0ss)

This is the first in a series of "tutorial scripts" I'll be making which use angr to find exploitable conditions in binaries. The first example is a very simple program. The script finds a path from the main entry point to `strcpy`, but **only** when we control the source buffer of the `strcpy` operation. To hit the right path, angr has to solve for a password argument, but angr solved this in less than 2 seconds on my machine using the standard python interpreter. The script might look large, but that's only because I've heavily commented it to be more helpful to beginners. The challenge binary is here and the script is here.

Beginner Exploitation example: CADET_0001

Author: Antonio Bianchi, Jacopo Corbetta

This is a very easy binary containing a stack buffer overflow and an easter egg. CADET_00001 is one of the challenge released by DARPA for the Cyber Grand Challenge: link The binary can run in the DECREE VM: link CADET_00001.adapted (by Jacopo Corbetta) is the same program, modified to be runnable in an Intel x86 Linux machine.

Grub "back to 28" bug

Author: Andrew Dutcher (github: @rhelmot)

This is the demonstration presented at 32c3. The script uses angr to discover the input to crash grub's password entry prompt.

script - vulnerable module

Insomnihack Simple AEG

Author: Nick Stephens (github: @NickStephens)

Demonstration for Insomni'hack 2016. The script is a very simple implementation of AEG.

script

FAQ

This is a collection of commonly-asked "how do I do X?" questions and other general questions about angr, for those too lazy to read this whole document.

How do I load a binary?

A binary is loaded by doing:

```
p = angr.Project("/path/to/your/binary")
```

Why am I getting terrifying error messages from LibVEX printed to stderr?

This is something that LibVEX does when it gets fed invalid instructions. VEX is not designed for static analysis, it's designed for instrumentation, so it's mode of handling bad data is to freak out as badly as it possibly can. There's no way of shutting it up, short of patching it.

We've already patched VEX so that instead of exiting, bringing down the python interpreter with it, it sends up a message that turns into a python exception than can later be caught by analysis. Long story short, *this should not affect your analysis if you're just using builtin angr routines.*

How can I get verbose debug messages for specific angr modules ?

angr uses the standard `logging` module for logging, with every package and submodule creating a new logger.

Debug messages for everything

The most simple way to get a debug output is the following:

```
import logging
logging.basicConfig(level=logging.DEBUG) # ajust to the wanted debug level
```

You may want to use `logging.INFO` or whatever else instead.

More granular control

Each angr module has its own logger string, usually all the python modules above it in the hierarchy, plus itself, joined with dots. For example, `angr.analyses.cfg`. Because of the way the python logging module works, you can set the verbosity for all submodules in a module by setting a verbosity level for the parent module. For example, `logging.getLogger('angr.analyses').setLevel(logging.INFO)` will make the CFG, as well as all other analyses, log at the INFO level.

Automatic log settings

If you're using angr through ipython, you can add a startup script in your ipython profile to set various logging levels.

Why is a CFG taking forever to construct?

You want to load the binary without shared libraries loaded. If they are loaded, like they are by default, the analysis will try to construct a CFG through your libraries, which is almost always a really bad idea. Add the following option to your `Project` constructor call: `load_options={'auto_load_libs': False}`

Why did you choose VEX instead of another IR (such as LLVM, REIL, BAP, etc)?

We had two design goals in angr that influenced this choice:

1. angr needed to be able to analyze binaries from multiple architectures. This mandated the use of an IR to preserve our sanity, and required the IR to support many architectures.
2. We wanted to implement a binary analysis engine, not a binary lifter. Many projects start and end with the implementation of a lifter, which is a time consuming process. We needed to take something that existed and already supported the lifting of multiple architectures.

Searching around the internet, the major choices were:

- LLVM is an obvious first candidate, but lifting binary code to LLVM cleanly is a pain. The two solutions are either lifting to LLVM through QEMU, which is hackish (and the only implementation of it seems very tightly integrated into S2E), or mcsema, which only supports x86.
- TCG is QEMU's IR, but extracting it seems very daunting as well and documentation is very scarse.
- REIL seems promising, but there is no standard reference implementation that supports all the architectures that we wanted. It seems like a nice academic work, but to use it, we would have to implement our own lifters, which we wanted to avoid.
- BAP was another possibility. When we started work on angr, BAP only supported lifting x86 code, and up-do-date versions of BAP were only available to academic collaborators of the BAP authors. These were two deal-breakers. BAP has since become open, but it still only supports x86_64, x86, and ARM.
- VEX was the only choice that offered an open library and support for many architectures. As a bonus, it is very well documented and designed specifically for program analysis, making it very easy to use in angr.

While angr uses VEX now, there's no fundamental reason that multiple IRs cannot be used. There are two parts of angr, outside of the `simuvex.vex` package, that are VEX-specific:

- the jump lables (i.e., the `Ijk_Ret` for returns, `Ijk_Call` for calls, and so forth) are VEX enums.
- VEX treats registers as a memory space, and so does angr. While we provide accesses to `state.regs.rax` and friends, on the backend, this does `state.registers.load(8, 8)`, where the first `8` is a VEX-defined offset for `rax` to the register file.

To support multiple IRs, we'll either want to abstract these things or translate their labels to VEX analogues.

My load options are ignored when creating a Project.

CLE options are an optional argument. Make sure you call Project with the following syntax:

```
b = angr.Project('/bin/true', load_options=load_options)
```

rather than:

```
b = angr.Project('/bin/true', load_options)
```

Why are some ARM addresses off-by-one?

In order to encode THUMB-ness of an ARM code address, we set the lowest bit to one. This convention comes from LibVEX, and is not entirely our choice! If you see an odd ARM address, that just means the code at `address - 1` is in THUMB mode.

I get an exception that says `AttributeError: 'FFI' object has no attribute 'unpack'` What do I do?

You have an outdated version of the `cffi` Python module. angr now requires at least version 1.7 of cffi. Try `pip install --upgrade cffi`. If the problem persists, make sure your operating system hasn't pre-installed an old version of cffi, which pip may refuse to uninstall. If you're using a Python virtual environment with the pypy interpreter, ensure you have a recent version of pypy, as it includes a version of cffi which pip will not upgrade.

When importing angr, I get an exception that says `ImportError: ERROR: fail to load the dynamic library.`

The capstone pip package is sometimes broken. You can reinstall capstone with:

```
pip install -I --no-use-wheel capstone
```

This will rebuild the dynamic library and your install will work.

Gotchas when using angr

This section contains a list of gotchas that users/victims of angr frequently run into.

SimProcedure inaccuracy

To make symbolic execution more tractable, angr replaces common library functions with summaries written in Python. We call these summaries SimProcedures. SimProcedures allow us to mitigate path explosion that would otherwise be introduced by, for example, `strlen` running on a symbolic string.

Unfortunately, our SimProcedures are far from perfect. If angr is displaying unexpected behavior, it might be caused by a buggy/incomplete SimProcedure. There are several things that you can do:

1. Disable the SimProcedure (you can exclude specific SimProcedures by passing options to the angr.Project class). This has the drawback of likely leading to a path explosion, unless you are very careful about constraining the input to the function in question. The path explosion can be partially mitigated with other angr capabilities (such as Veritesting).
2. Replace the SimProcedure with something written directly to the situation in question. For example, our `scanf` implementation is not complete, but if you just need to support a single, known format string, you can write a hook to do exactly that.
3. Fix the SimProcedure.

Unsupported syscalls

System calls are also implemented as SimProcedures. Unfortunately, there are system calls that we have not yet implemented in angr. There are several workarounds for an unsupported system call:

1. Implement the system call. *TODO: document this process*
2. Hook the callsite of the system call (using `project.hook`) to make the required modifications to the state in an ad-hoc way.
3. Use the `state.posix.queued_syscall_returns` list to queue syscall return values. If a return value is queued, the system call will not be executed, and the value will be used instead. Furthermore, a function can be queued instead as the "return value", which will result in that function being applied to the state when the system call is triggered.

Symbolic memory model

The default memory model used by angr is inspired by Mayhem. This memory model supports limited symbolic reads and writes. If the memory index of a read is symbolic and the range of possible values of this index is too wide, the index is concretized to a single value. If the memory index of a write is symbolic at all, the index is concretized to a single value. This is configurable by changing the memory concretization strategies of `state.memory`.

Symbolic lengths

SimProcedures, and especially system calls such as `read()` and `write()` might run into a situation where the *length* of a buffer is symbolic. In general, this is handled very poorly: in many cases, this length will end up being concretized outright or retroactively concretized in later steps of execution. Even in cases when it is not, the source or destination file might end up looking a bit "weird".

Changelog

This lists the *major* changes in angr. Tracking minor changes are left as an exercise for the reader :-)

angr 6.7.1.13

For the last month, we have been working on a major refactor of the angr to change the way that angr reasons about the code that it analyzes. Until now, angr has been bound to the VEX intermediate representation to lift native code, supporting a wide range of architectures but not being very expandable past them. This release represents the ground work for what we call translation and execution engines. These engines are independent backends, pluggable into the angr framework, that will allow angr to reason about a wide range of targets. For now, we have restructured the existing VEX and Unicorn Engine support into this engine paradigm, but as we discuss in our blog post, the plan is to create engines to enable angr's reasoning of Java bytecode and source code, and to augment angr's environment support through the use of external dynamic sandboxes.

For now, these changes are mostly internal. We have attempted to maintain compatibility for end-users, but those building systems atop angr will have to adapt to the modern codebase. The following are the major changes:

- simuvex: we have introduced SimEngine. SimEngine is a base class for abstractions over native code. For example, angr's VEX-specific functionality is now concentrated in SimEngineVEX, and new engines (such as SimEngineLLVM) can be implemented (even outside of simuvex itself) to support the analysis of new types of code.
- simuvex: as part of the engines refactor, the SimRun class has been eliminated. Instead of different subclasses of SimRun that would be instantiated from an input state, engines each have a `process` function that, from an input state, produces a SimSuccessors instance containing lists of different successor states (normal, unsat, unconstrained, etc) and any engine-specific artifacts (such as the VEX statements. Take a look at `successors.artifacts`).
- simuvex: `state.mem[x:] = y` now *requires* a type for storage (for example `state.mem[x:].dword = y`).
- simuvex: the way of calling inline SimProcedures has been changed. Now you have to create a SimProcedure, and then call `execute()` on it and pass in a program state as well as the arguments.
- simuvex: accessing registers through `SimRegNameView` (like `state.regs.eax`) always

triggers SimInspect breakpoints and creates new actions. Now you can access a register by prefixing its name with an underscore (e.g. `state.regs._eax` or `state._ip`) to avoid triggering breakpoints or creating actions.
- angr: the way hooks work has slightly changed, though is backwards-compatible. The new angr.Hook class acts as a wrapper for hooks (SimProcedures and functions), keeping things cleaner in the `project._sim_procedures` dict.
- angr: we have deprecated the keyword argument `max_size` and changed it to to `size` in the `angr.Block` constructor (i.e., the argument to `project.factory.block` and more upstream methods (`path.step`, `path_group.step`, etc).
- angr: we have deprecated `project.factory.sim_run` and changed it to to `project.factory.successors`, and it now generates a `SimSuccessors` object.
- angr: `project.factory.sim_block` has been deprecated and replaced with `project.factory.successors(default_engine=True)`.
- angr: angr syscalls are no longer hooks. Instead, the syscall table is now in `project._simos.syscall_table`. This will be made "public" after a usability refactor. If you were using `project.is_hooked(addr)` to see if an address has a related SimProcedure, now you probably want to check if there is a related syscall as well (using `project._simos.syscall_table.get_by_addr(addr) is not None`).
- pyvex: to support custom lifters to VEX, pyvex has introduced the concept of backend lifters. Lifters can be written in pure python to produce VEX IR, allowing for extendability of angr's VEX-based analyses to other hardware architectures.

As usual, there are many other improvements and minor bugfixes.

- claripy: support `unsat_core()` to get the core of unsatness of constraints. It is in fact a thin wrapper of the `unsat_core()` function provided by Z3. Also a new state option `CONSTRAINT_TRACKING_IN_SOLVER` is added to SimuVEX. That state option must be enabled if you want to use `unsat_core()` on any state.
- simuvex: `SimMemory.load()` and `SimMemory.store()` now takes a new parameter `disable_actions`. Setting it to True will prevent any SimAction creation.
- angr: CFGFast has a better support for ARM binaries, especially for code in THUMB mode.
- angr: thanks to an improvement in SimuVEX, CFGAccurate now uses slightly less memory than before.
- angr: `len()` on path `trace` or `addr_trace` is made much faster.
- angr: Fix a crash during CFG generation or symbolic execution on platforms/architectures with no syscall defined.
- angr: as part of the refactor, `BackwardSlicing` is temporarily disabled. It will be re-enabled once all DDG-related refactor are merged to master.

Additionally, packaging and build-system improvements coordinated between the angr and Unicorn Engine projects have allowed angr's Unicorn support to be built on Windows. Because of this, `unicorn` is now a dependency for `simuvex`.

Looking forward, angr is poised to become a program analysis engine for binaries *and more*!

angr 5.6.12.3

It has been over a month since the last release 5.6.10.12. Again, we've made some significant changes and improvements on the code base.

- angr: Labels are now stored in KnowledgeBase.
- angr: Add a new analysis: `Disassembly`. The new Disassembly analysis provides an easy-to-use interface to render assembly of functions.
- angr: Fix the issue that `ForwardAnalysis` may prematurely terminate while there are still un-processed jobs.
- angr: Many small improvements and bug fixes on `CFGFast`.
- angr: Many small improvements and bug fixes on `VFG`. Bring back widening support. Fix the issue that `VFG` may not terminate under certain cases. Implement a new graph traversal algorithm to have an optimal traversal order. Allow state merging at non-merge-points, which allows faster convergence.
- angr-management: Display a progress during initial CFG recovery.
- angr-management: Display a "Load binary" window upon binary loading. Some analysis options can be adjusted there.
- angr-management: Disassembly view: Edge routing on the graph is improved.
- angr-management: Disassembly view: Support starting a new symbolic execution task from an arbitrary address in the program.
- angr-management: Disassembly view: Support renaming of function names and labels.
- angr-management: Disassembly view: Support "Jump to address".
- angr-management: Disassembly view: Display resolved and unresolved jump targets. All jump targets are double-clickable.
- SimuVEX: Move region mapping from `SimAbstractMemory` to `SimMemory`. This will allow an easier conversion between `SimAbstractMemory` and `SimSymbolicMemory`, which is to say, conversion between symbolic states and static states is now possible.
- SimuVEX & claripy: Provide support for `unsat_core` in Z3. It returns a set of constraints that led to unsatness of the constraint set on the current state.
- archinfo: Add a new Boolean variable `branch_delay_slot` for each architecture. It is set to True on MIPS32.

angr 5.6.8.22

Major point release! An incredible number of things have changed in the month run-up to the Cyber Grand Challenge.

- Integration with Unicorn Engine supported for concrete execution. A new SimRun type, SimUnicorn, may step through many basic blocks at once, so long as there is no operation on symbolic data. Please use our fork of unicorn engine, which has many patches applied. All these patches are pending merge into upstream.
- Lots of improvements and bug fixes to CFGFast. Rumors are angr's CFG was only "optimized" for x86-64 binaries (which is really because most of our test cases are compiled as 64-bit ELFs). Now it is also "optimized" for x86 binaries :) (editor's note: angr is built with cross-architecture analysis in mind. CFG construction is pretty much the only component which has architecture-specific behavior.)
- Lots of improvements to the VFG analysis, including speed and accuracy. However, there is still a lot to be done.
- Lots of speed optimizations in general - CFGFast should be 3-6x faster under CPython with much less memory usage.
- Now data dependence graph gives you a real dependence graph between variable definitions. Try `data_graph` and `simplified_data_graph` on a DDG object!
- New state option `simuvex.o.STRICT_PAGE_ACCESS` will cause a `SimSegfaultError` to be raised whenever the guest reads/writes/executes memory that is either unmapped or doesn't have the appropriate permissions.
- Merging of paths (as opposed to states) is performed in a much smarter way.
- The behavior of the `support_selfmodifying_code` project option is changed: Before, this would allow the state to be used as a fallback source of instruction bytes when no backer from CLE is available. Now, this option makes instruction lifting use the state as the source of bytes always. When the option is disabled and execution jumps outside the normal binary, the state will be used automatically.
- *Actually* support self-modifying code - if a basic block of code modifies itself, the block will be re-lifted before the next instruction starts.
- Syscalls are handled differently now - Before you would see a SimRun for a syscall helper, now you'll just see a SimProcedure for the given syscall. Additionally, each syscall has its own address in a "syscalls segment", and syscalls are treated as jumps to this segment. This simplifies a lot of things analysis-wise.
- CFGAccurate accepts a `base_graph` keyword to its constructor, e.g. `CFGFast().graph`, or even `.graph` of a function, to use as a base for analysis.
- New fast memory model for cases where symbolic-addressed reads and writes are unlikely.
- Conflicts between the `find` and `avoid` parameters to the Explorer otiegnqwvk are resolved correctly. (credit clslgrnc)
- New analysis `StaticHooker` which hooks library functions in unstripped statically linked binaries.

- `Lifter` can be used without creating an angr Project. You must manually specify the architecture and bytestring in calls to `.lift()` and `.fresh_block()`. If you like, you can also specify the architecture as a parameter to the constructor and omit it from the lifting calls.
- Add two new analyses developed for the CGC (mostly as examples of doing static analysis with angr): Reassembler and BinaryOptimizer.

angr 4.6.6.28

In general, there have been enormous amounts of speed improvements in this release. Depending on the workload, angr should run about twice as fast. Aside from this, there have also been many submodule-specific changes:

angr

Quite a few changes and improvements are made to `CFGFast` and `CFGAccurate` in order to have better and faster CFG recovery. The two biggest changes in `CFGFast` are jump table resolution and data references collection, respectively. Now `CFGFast` resolves indirect jumps by default. You may get a list of indirect jumps recovered in `CFGFast` by accessing the `indirect_jumps` attribute. For many cases, it resolves the jump table accurately. Data references collection is still in alpha mode. To test data references collection, just pass `collect_data_references=True` when creating a fast CFG, and access the `memory_data` attribute after the CFG is constructed.

CFG recovery on ARM binaries is also improved.

A new paradigm called an "otiegnqwvk", or an "exploration technique", allows the packaging of special logic related to path group stepping.

SimuVEX

Reads/writes to the x87 fpu registers now work correctly - there is special logic that rotates a pointer into part of the register file to simulate the x87 stack.

With the recent changes to Claripy, we have configured SimuVEX to use the composite solver by default. This should be transparent, but should be considered if strange issues (or differences in behavior) arise during symbolic execution.

Claripy

Fixed a bug in claripy where `__div__` was not always doing unsigned division, and added new methods `SDiv` and `SMod` for signed division and signed remainder, respectively.

Claripy frontends have been completely rewritten into a mixin-centric solver design. Basic frontend functionality (i.e., calling into the solver or dealing with backends) is handled by frontends (in `claripy.frontends`), and additional functionality (such as caching, deciding when to simplify, etc) is handled by frontend mixins (in `claripy.frontend_mixins`). This makes it considerably easier to customize solvers to your specific needE. For examples, look at `claripy/solver.py`.

Alongside the solver rewrite, the composite solver (which splits constraints into independent constraint sets for faster solving) has been immensely improved and is now functional and fast.

angr 4.6.6.4

Syscalls are no longer handled by `simuvex.procedures.syscalls.handler`. Instead, syscalls are now handled by `angr.SimOS.handle_syscall()`. Previously, the address of a syscall SimProcedure is the address right after the syscall instruction (e.g. `int 80h`), which collides with the real basic block starting at that address, and is very confusing. Now each syscall SimProcedure has its own address, just as a normal SimProcedure. To support this, there is another region mapped for the syscall addresses, `Project._syscall_obj`.

Some refactoring and bug fixes in `CFGFast`.

Claripy has been given the ability to handle *annotations* on ASTs. An annotation can be used to customize the behavior of some backends without impacting others. For more information, check the docstrings of `claripy.Annotation` and `claripy.Backend.apply_annotation`.

angr 4.6.5.25

New state constructor - `call_state`. Comes with a refactor to `SimCC`, a refactor to `callable`, and the removal of `PathGroup.call`. All these changes are thoroughly documented, in `angr-doc/docs/structured_data.md`

Refactor of `SimType` to make it easier to use types - they can be instanciated without a SimState and one can be added later. Comes with some usability improvements to SimMemView. Also, there's a better wrapper around PyCParser for generating SimType instances from c declarations and definitions. Again, thoroughly documented, still in the structured data doc.

`CFG` is now an alias to `CFGFast` instead of `CFGAccurate`. In general, `CFGFast` should work under most cases, and it's way faster than `CFGAccurate`. We believe such a change is necessary, and will make angr more approachable to new users. You will have to change your code from `CFG` to `CFGAccurate` if you are relying on specific functionalities that only exist in `CFGAccurate`, for example, context-sensitivity and state-preserving. An exception will be raised by angr if any parameter passed to `CFG` is only supported by `CFGAccurate`. For more detailed explanation, please take a look at the documentation of `angr.analyses.CFG`.

angr 4.6.3.28

PyVEX has a structural overhaul. The `IRExpr`, `IRStmt`, and `IRConst` modules no longer exist as submodules, and those module names are deprecated. Use `pyvex.expr`, `pyvex.stmt`, and `pyvex.const` if you need to access the members of those modules.

The names of the first three parameters to `pyvex.IRSB` (the required ones) have been changed. If you were passing the positional args to IRSB as keyword args, consider switching to positional args. The order is `data`, `mem_addr`, `arch`.

The optional parameter `sargc` to the `entry_state` and `full_init_state` constructors has been removed and replaced with an `argc` parameter. `sargc` predates being able to have claripy ASTs independent from a solver. The new system is to pass in the exact value, ast or integer, that you'd like to have as the guest program's arg count.

CLE and angr can now accept file-like streams, that is, objects that support `stream.read()` and `stream.seek()` can be passed in wherever a filepath is expected.

Documentation is much more complete, especially for PyVEX and angr's symbolic execution control components.

angr 4.6.3.15

There have been several improvements to claripy that should be transparent to users:

- There's been a refactoring of the VSA StridedInterval classes to fix cases where operations were not sound. Precision might suffer as a result, however.
- Some general speed improvements.
- We've introduced a new backend into claripy: the ReplacementBackend. This frontend generates replacement sets from constraints added to it, and uses these replacement sets to increase the precision of VSA. Additionally, we have introduced the HybridBackend, which combines this functionality with a constraint solver, allowing for memory index resolution using VSA.

angr itself has undergone some improvements, with API changes as a result:

- We are moving toward a new way to store information that angr has recovered about a program: the knowledge base. When an analysis recovers some truth about a program (i.e., "there's a basic block at 0x400400", or "the block at 0x400400 has a jump to 0x400500"), it gets stored in a knowledge-base. Analysis that used to store data (currently, the CFG) now store them in a knowledge base and can *share* the global knowledge base of the project, now accessible via `project.kb`. Over time, this knowledge base will be expanded in the course of any analysis or symbolic execution, so angr is constantly learning more information about the program it is analyzing.
- A forward data-flow analysis framework (called ForwardAnalysis) has been introduced, and the CFG was rewritten on top of it. The framework is still in alpha stage - expect more changes to be made. Documentation and more details will arrive shortly. The goal is to refactor other data-flow analysis, like CFGFast, VFG, DDG, etc. to use ForwardAnalysis.
- We refactored the CFG to a) improve code readability, and b) eliminate some bad designs that linger due to historical reasons.

angr 4.5.12.?

Claripy has a new manager for backends, allowing external backends (i.e., those implemented by other modules) to be used. The result is that `claripy.backend_concrete` is now `claripy.backends.concrete`, `claripy.backend_vsa` is now `claripy.backends.vsa`, and so on.

angr 4.5.12.12

Improved the ability to recover from failures in instruction decoding. You can now hook specific addresses at which VEX fails to decode with `project.hook`, even if those addresses are not the beginning of a basic block.

angr 4.5.11.23

This is a pretty beefy release, with over half of claripy having been rewritten and major changes to other analyses. Internally, Claripy has been unified -- the VSA mode and symbolic mode now work on the same structures instead of requiring structures to be created differently. This opens the door for awesome capabilities in the future, but could also result in unexpected behavior if we failed to account for something.

Claripy has had some major interface changes:

- claripy.BV has been renamed to claripy.BVS (bit-vector symbol). It can now create bitvectors out of strings (i.e., claripy.BVS(0x41, 8) and claripy.BVS("A") are identical).
- state.BV and state.BVV are deprecated. Please use state.se.BVS and state.se.BVV.
- BV.model is deprecated. If you're using it, you're doing something wrong, anyways. If you really need a specific model, convert it with the appropriate backend (i.e., claripy.backend_concrete.convert(bv)).

There have also been some changes to analyses:

- Interface: CFG argument `keep_input_state` has been renamed to `keep_state`. With this option enabled, both input and final states are kept.
- Interface: Two arguments `cfg_node` and `stmt_id` of `BackwardSlicing` have been deprecated. Instead, `BackwardSlicing` takes a single argument, `targets`. This means that we now support slicing from multiple sources.
- Performance: The speed of CFG recovery has been slightly improved. There is a noticeable speed improvement on MIPS binaries.
- Several bugs have been fixed in DDG, and some sanity checks were added to make it more usable.

And some general changes to angr itself:

- StringSpec is deprecated! You can now pass claripy bitvectors directly as arguments.