angr.io: an open-source binary analysis platform for Python

Andrea Munarin (879607)

Year 2022/2023

1 What is angr.io

angr is an open-source binary analysis platform for Python. It combines both **static** and **dynamic** symbolic ("concolic") analysis, providing tools to solve a variety of tasks.
The main feature ***angr.io*** offers are:

- ***Cross-Platform***
  - Runs on Windows, macOS, and Linux. Built for Python 3.8+.
- ***Open Source***
  - Released as Free and Open Source Software under the permissive BSD license. Contributions are welcome.
- ***Control-Flow Graph Recovery***
  - Provides advanced analysis techniques for control-flow graph recovery.
- ***Symbolic Execution***
  - Provides a powerful symbolic execution engine, constraint solving, and instrumentation.
- ***Disassembly & Lifting***
  - Provides convenient methods to disassemble code and lift to an intermediate language.
- ***Decompilation***
  - Decompile machine code to angr Intermediate Language (AIL) and C pseudocode.
- ***Architecture Support***
  - Supports analysis of several CPU architectures, loading from several executable formats.
- ***Extensibility***
  - Provides powerful extensibility for analyses, architectures, platforms, exploration techniques, hooks, and more.

As an introduction to angr's capabilities, here are some of the things that you can do using angr and the tools built with it (some of these are cited before):

- *Control-flow graph recovery.*
- *Symbolic execution.*
- Automatic ROP chain building using angrop.
- Automatic binary hardening using patcherex.

- Automatic exploit generation (for DECREE and simple Linux binaries) using rex.
- Use angr-management, a (very alpha state!) GUI for angr, to analyze binaries!
- Achieve cyber-autonomy in the comfort of your own home, using Mechanical Phish, the third-place winner of the DARPA Cyber Grand Challenge.

2 Setup of the environment

angr is a library for **Python 3.8+**, and must be installed into a Python environment before it can be used[1].

angr is published on **PyPI**, and using this is the easiest and recommended way to install angr. It can be installed angr with pip:

```
pip install angr
```

For what we are going to use it is also useful to install, in the directory in which we are going to work, some other useful package:

- `git clone https://github.com/axt/bingraphvis`
  `pip install -e ./bingraphvis`
  - The aim of this library is to provide a generic visualization for the graphs (CFG, CG, DDG, CDG, etc) produced by various binary analysis frameworks.
- `git clone https://github.com/axt/angr-utils`
  `pip install -e ./angr-utils`
  - Angr-utils is a collection of utilities for angr binary analysis framework.
- `sudo apt-get install xdot`
  - Interactive viewer for graphs written in Graphviz's dot language
- `pip install monkeyhex`
  - A library which permits to format numerical results in hexadecimal

3 Core concepts

Your first action with angr will always be to **load a binary** into a project. We'll use */bin/true* for these examples.

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

A project is your control base in angr. With it, you will be able to dispatch analyses and simulations on the executable you just loaded. Almost every single object you work with in angr will depend on the existence of a project in some form.

we have some basic properties about the project: its **CPU architecture**, its **filename**, and the address of its **entry point**.

```
import monkeyhex # this will format numerical results in hexadecimal
proj.arch # <Arch AMD64 (LE)>
proj.entry # 0x401670
proj.filename # '/bin/true'
```

[1]Tip: It is recommended to use an isolated python environment rather than installing angr globally. Doing so reduces dependency conflicts and aids in reproducibility while debugging.

There are a lot of classes in angr, and most of them require a project to be instantiated. Instead of making you pass around the project everywhere, we provide `project.factory`, which has several convenient constructors for common objects you'll want to use frequently.

3.1 BLOCKS

First, we have `project.factory.block()`, which is used to extract a basic block of code from a given address. This is an important fact - angr analyzes code in units of **basic blocks**. You will get back a Block object, which can tell you lots of fun things about the block of code:

```
block = proj.factory.block(proj.entry) # lift a block of code from the program's entry point
block.pp() # pretty-print a disassembly to stdout
block.instructions # how many instructions are there?
block.instruction\_addrs # what are the addresses of the instructions?
```

3.2 STATES

Here's another fact about angr - the Project object only represents an "*initialization image*" for the program. When you're performing execution with angr, you are working with a specific object representing a simulated program state - a ***SimState***. Let's grab one right now!

```
state = proj.factory.entry_state()
```

A SimState contains a **program's memory, registers, filesystem data**... any "***live data***" that can be changed by execution has a home in the state. For now, let's use `state.regs` and `state.mem` to access the registers and memory of this state:

```
state.regs.rip # get the current instruction pointer
state.regs.rax
state.mem[proj.entry].int.resolved # interpret the memory at the entry point as a C int
```

Those aren't Python ints! Those are **bitvectors**. Python integers don't have the same semantics as words on a CPU, e.g., wrapping on overflow, so we work with bitvectors, which you can think of as an **integer as represented by a series of bits**, to represent **CPU data in angr**. Note that each bitvector has a .length property describing how wide it is in bits.

```
bv = state.solver.BVV(0x1234, 32) # create a 32-bit-wide bitvector with
state.solver.eval(bv) # convert to Python int
```

You can store these bitvectors back to registers and memory, or you can **directly store a Python integer**, and it'll be converted to a bitvector of the appropriate size:

```
state.regs.rsi = state.solver.BVV(3, 64)
state.regs.rsi

state.mem[0x1000].long = 4
state.mem[0x1000].long.resolved
```

The mem interface is a little confusing at first, since it's using some **pretty hefty Python magic**.

3.3 SIMULATION MANAGERS

If a state lets us represent a program at a given point in time, there must be a way to **get it to the next point** in time. A **simulation manager** is the primary interface in angr for performing execution, simulation, whatever you want to call it, with states. As a brief introduction, let's show how to tick that state we created earlier forward a few basic blocks.

First, we create the simulation manager we're going to be using. The constructor can take a state or a list of states.

```
simgr = proj.factory.simulation_manager(state)
```

Now... get ready, we're going to do some execution.

```
simgr.step()
```

We've just performed a **basic block's worth of symbolic execution!** We can look at the active stash again, noticing that it's been updated, and furthermore, **that it has not modified our original state**. **SimState** objects are treated as **immutable by execution** - you can safely use a single state as a "base" for multiple rounds of execution.

```
simgr.active
simgr.active[0].regs.rip
state.regs.rip
```

3.4 ANALYSIS

angr's goal is to make it easy to carry out useful analyses on binary programs. To this end, angr allows you to package analysis code in a common format that can be easily applied to any project. We will cover writing your own analyses Writing Analyses, but the idea is that all the analyses appear under project.analyses (for example, `project.analyses.CFGFast()`) and can be called as functions, returning analysis result instances.

4 Control Flow Graph Recovery

4.1 General Idea

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 static CFG (**CFGFast**) and a dynamic CFG (**CFGEmulated**).

- ***CFGFast*** uses static analysis to generate a CFG. It is significantly faster, but is theoretically bounded by the fact that some control-flow transitions can only be resolved at execution-time. This is the same sort of CFG analysis performed by other popular reverse-engineering tools, and its results are comparable with their output.
- ***CFGEmulated*** uses symbolic execution to capture the CFG. While it is theoretically more accurate, it is dramatically slower. It is also typically less complete, due to issues with the accuracy of emulation (system calls, missing hardware features, and so on)

If you are unsure which CFG to use, or are having problems with CFGEmulated, try CFGFast first.

A CFG can be constructed by doing:

```
import angr
p = angr.Project('/bin/true', load_options={'auto_load_libs': False})

# Generate a static CFG
cfg = p.analyses.CFGFast()

# generate a dynamic CFG
cfg = p.analyses.CFGEmulated(keep_state=True)
```

The CFG, at its core, is a *NetworkX di-graph*. This means that all 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.edges())))
```

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).

```
entry_node = cfg.get_any_node(p.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(p.entry)))

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

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.

4.2 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}`

4.3 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 dictionary**. It maps addresses to Function objects, which can tell you properties about a function.

```
entry_func = cfg.kb.functions[p.entry]
```

Functions have several important properties that can be looked up into the documentation of angr.

5 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.

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 (**CFGEmulated**).
- **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.

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

# Generate a CFG first. In order to generate data dependence graph afterwards, you'll have
                                                    to:
# - keep all input states by specifying keep_state=True.
# - store memory, register and temporary values accesses by adding the angr.options.refs
                                                    option set.
# Feel free to provide more parameters (for example, context_sensitivity_level) for CFG
# recovery based on your needs.
cfg = b.analyses.CFGEmulated(keep_state=True,
                             state_add_options=angr.sim_options.refs,
                             context_sensitivity_level=2)

# 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 program slice!
print(bs)
```

Sometimes it's **difficult to get a data dependence graph**, or you may simply want to 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)])
```

User-friendly Representation Take a look at BackwardSlice.dbg_repr()!

6 Identifier

The identifier uses **test cases to identify common library functions in CGC binaries**. It prefilters by finding some basic information about **stack variables/arguments**. The information of about stack variables can be generally useful in other projects.

```
import angr
# get all the matches
p = angr.Project("../binaries/tests/i386/identifiable")
# note analysis is executed via the Identifier call
idfer = p.analyses.Identifier()
for funcInfo in idfer.func_info:
    print(hex(funcInfo.addr), funcInfo.name)
```

7 Example and try Yourself

These are some introductory examples to give an idea of how to use angr's API.

7.1 Fauxware

This is a basic script that explains how to use angr to symbolically execute a program and produce concrete input satisfying certain conditions.
In order to understand how we can solve it, it is useful to know some function:

- The **simulation manager** has a function `sm.run()` which can takes several parameters. In our case, we are interested in the `until` parameter. This parameter expects a lambda function that tells him when to stop the execution
- Remember that `sm.active` contains the states that will be stepped by default, unless an alternate stash is specified.

7.2 Try Yourself

In the code that I will give to you, there is a simple ***crackme*** example. Try yourself using whatever technique you prefer (*CFG analysis*, *Simulation manager*, etc.). I suggest you to first try to execute the program and see its behavior and then work on it.
If you have any doubt, there will also be the source file.

7.3 Other Examples

There are many other examples in the docs: "*https://docs.angr.io/en/latest/examples.html*". If you are curious and what to try with some more complex binaries, you can go to the website and try some challenge. There are several possibilities with angr, in particular:

- Reversing
- Vulnerability Discovery
- Exploitation

8 References

If you want to explore more in details and learn **advanced topics** of angr I strongly suggest you to visit the documentation on the site: "*https://docs.angr.io/en/latest/quickstart.html*". There you will find more examples and some concepts that have been ignored in this presentation due to the time. Moreover, that are **several papers** for which the angr tool was developed:

- *SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis*[2]
- *Driller: Augmenting Fuzzing Through Selective Symbolic Execution*[3]
- *Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware*[1]

Whole bibliography

[1] Yan Shoshitaishvili et al. "Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware". In: (2015).

[2] Yan Shoshitaishvili et al. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: (2016).

[3] Nick Stephens et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution". In: (2016).