how to angr lol

Nathan Huckleberry

The University of Texas at Austin

table of contents

table of contents

Compiler Theory

Angr is primarily based off of some compiler theory concepts.

Understanding the compiler theory concepts is key to understanding angr.

Basic Blocks

Basic blocks are straight-line code sequences with no branches in except to the entry and no branches out except at the exit.

```
add edx, 1
cdqe
mov eax, dword [rsi + rax*4]
add ecx, eax
cmp eax, 0xf
jne 0x286f
```

Figure 1: Basic Block

Basic Blocks

We can decompose programs into basic blocks to make analysis easier.

Instead of understanding every instruction, we just need to understand each basic block.

Control Flow Graphs

Using the last instruction of a basic block we can determine its possible successors.

This allows us to build a graph of the program's control flow.

Control Flow Graphs

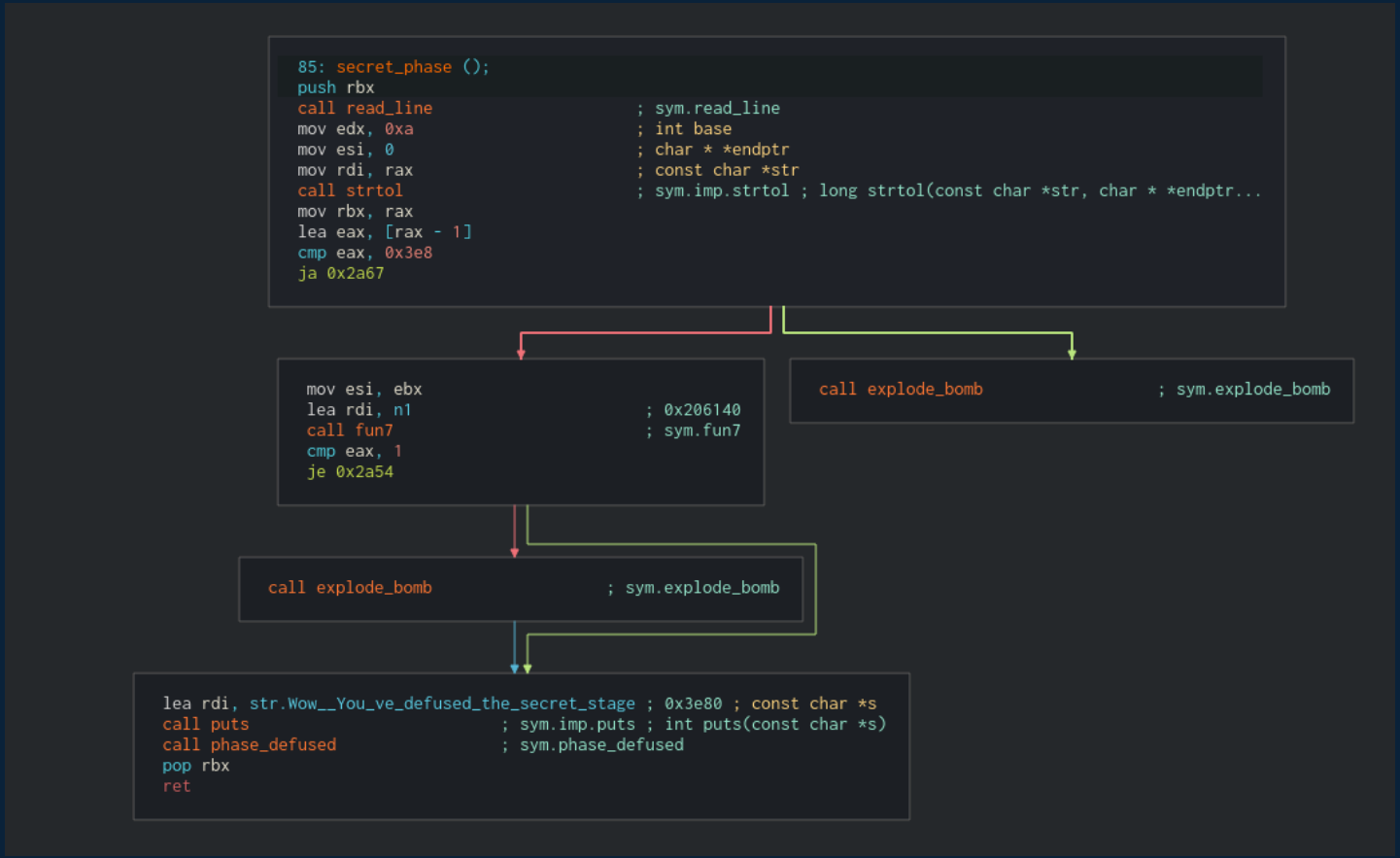


Figure 1: Control Flow Graph

Control Flow Graphs

Graphs of this form are called control flow graphs (CFGs)

CFGs represent every possible path a program can take during execution

Control Flow Graphs

Control flow graphs are often used by compilers to optimize programs

They are also used by security engineers to analyze programs

table of contents

Symbolic Execution

Symbolic Execution is a type of static analysis that can be thought of as graph walks on a CFG

We keep a running system of equations.

Each time we exit a basic block, we append the implied conditions of our exit to the system.

Symbolic Execution

```
add edx, 1
cdqe
mov eax, dword [rsi + rax*4]
add ecx, eax
cmp eax, 0xf
jne 0x286f
```

Figure 1: Basic Block

If we follow the true branch it must be the case that

$$\text{eax} \neq \text{0xf}$$

Which implies

$$[\text{rsi} + \text{rax} * 4] \neq 0xf$$

Symbolic Execution

```
add edx, 1
cdqe
mov eax, dword [rsi + rax*4]
add ecx, eax
cmp eax, 0xf
jne 0x286f
```

Figure 1: Basic Block

If we follow the false branch it must be the case that

$$\text{eax} = 0\text{xf}$$

Which implies

$$[\text{rsi} + \text{rax} * 4] = 0xf$$

Symbolic Execution

Iterating this process will allow us to build up huge systems of equations for each possible path.

This system of equations tells us the conditions required for a program to follow a specific path.

Symbolic Execution

Notice that there are about 2^n possible paths through the program.

This exponential blow-up makes symbolic execution very slow.

Symbolic Execution

Angr is mostly based on the concept of symbolic execution.

table of contents

Reachability

How do we use symbolic execution in a CTF?

Basic Crackme

Reverse Engineering CTF problems often following this format:

- Prompt for password
- Encode password with secret encoding scheme
- Compare password to encoded flag
- Output snarky message if they get it wrong

Basic Crackme

Normally, we just reverse engineer the encoding scheme.

Unfortunately, reversing is hard and a lot of work.

Basic Crackme

Consider a reversing problem that does the following:

- Ask for the user to enter the flag
- Encode the flag using some really complicated encoding scheme
- Compare the user's input with the actual encoded flag
- Output "nice flag" if they got it right
- Output "git gud" if they got it wrong

Basic Crackme

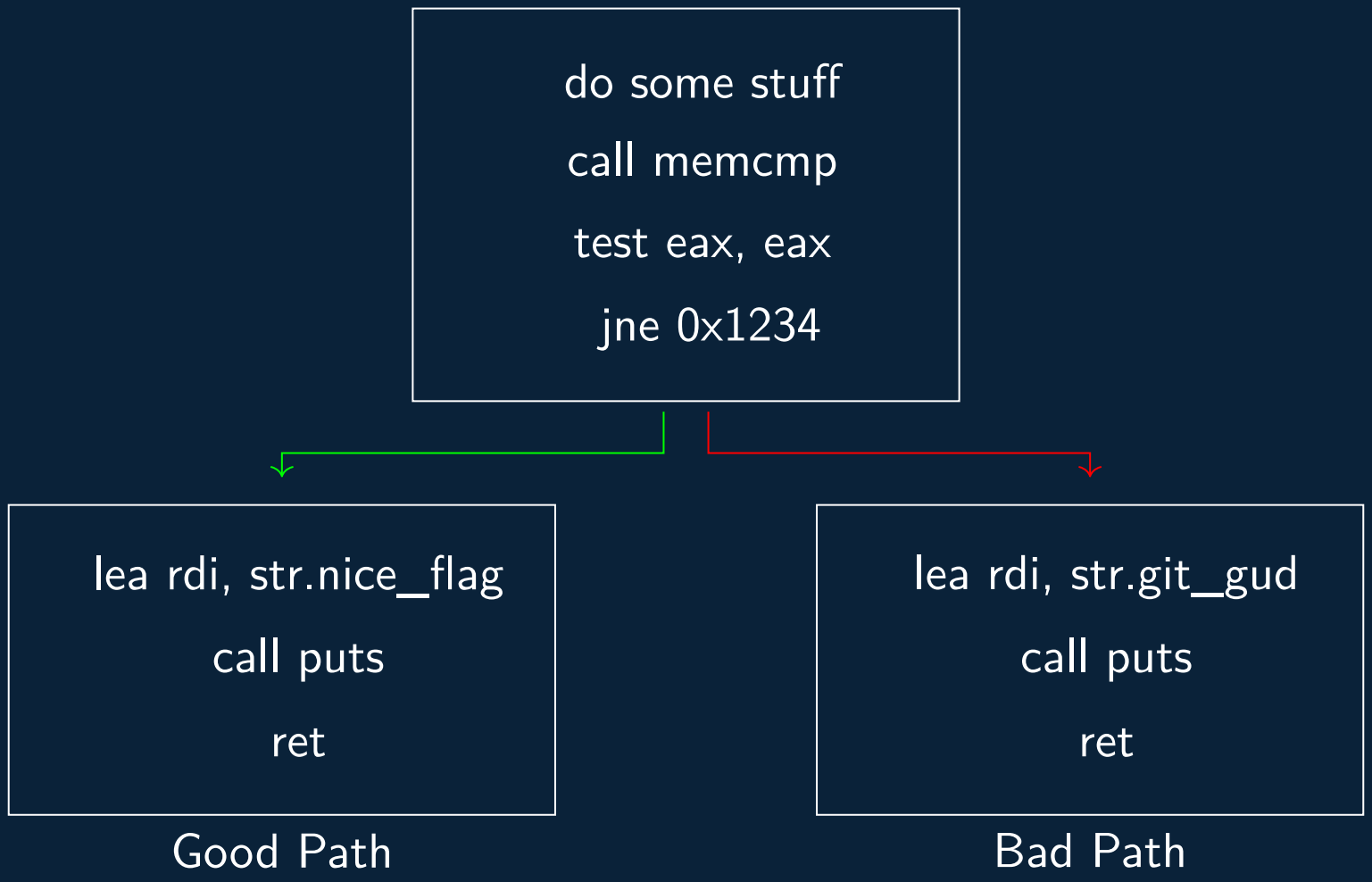


Figure 1: Simplified CFG

Basic Crackme

Traverse the CFG using our favorite graph traversal.

Throw away any paths that hit git_gud.

Stop when we find a path that hits nice_flag.

Find some input that solves the system of constraints.

This should be the flag.

Basic Crackme

Similar techniques can be used on most reverse engineering problems.

This even works on self-modifying code.

Notice that we didn't even read the actual "reversing" part of the challenge.

table of contents

Angr

Ok, but how do we actually use angr?

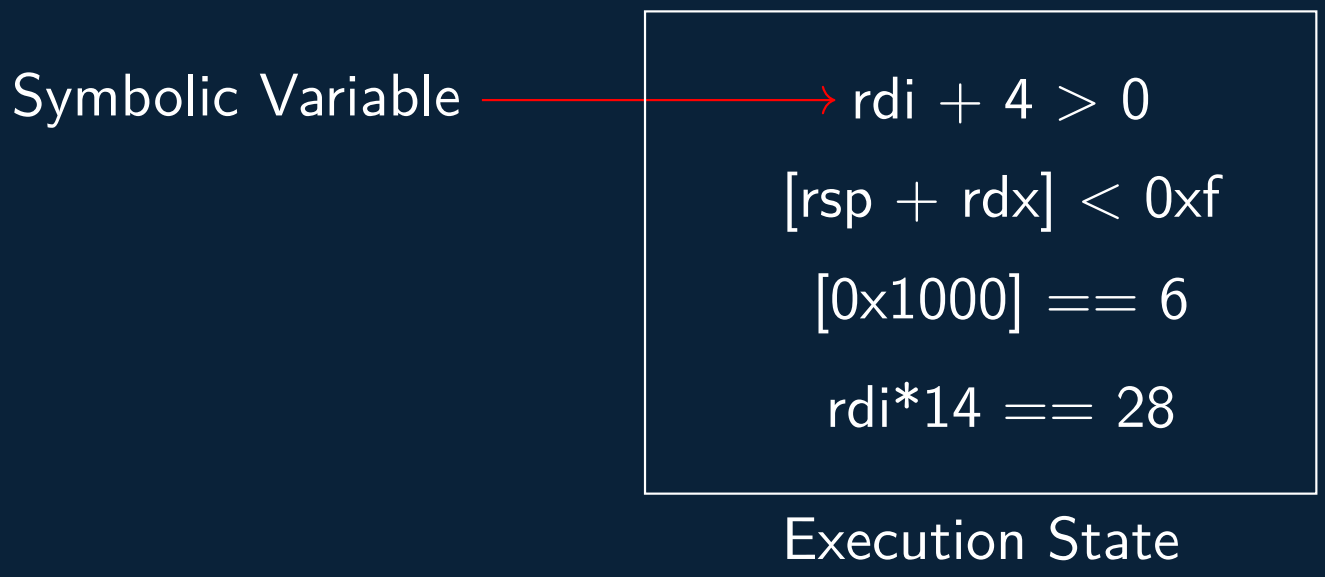
Angr just provides all the necessary parts for useful symbolic execution.

Symbolic Variables

Symbolic variables are the basic building block of symbolic execution.

We use symbolic variables to create each symbolic execution state.

Symbolic Variables



Symbolic Variables

A symbolic variable can be thought of as unset variables from math class.

We define our symbolic execution states as a list of relationships between symbolic variables.

Symbolic Variables

Angr uses Claripy to represent symbolic variables which are often referred to as ASTs.

Claripy allows us to create symbolic variables, constrain them, then pass them into angr.

We can also evaluate ASTs to find a concrete value that matches a constraint.

Symbolic Variables

The process of determining whether satisfiable input exists for a given constraint is called SAT solving.

SAT solving is extremely hard.

Fortunately there are open source SAT solvers to do stuff for us.

Symbolic Variables

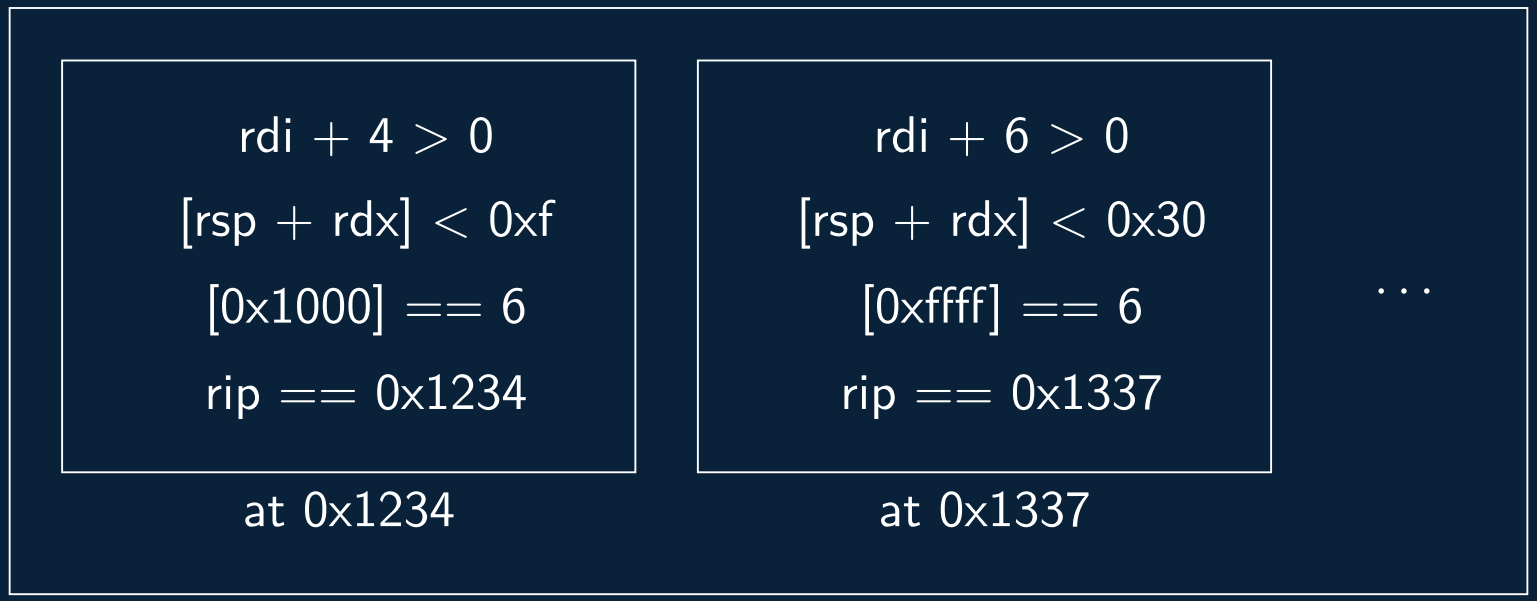
```
1 x = claripy.BVS("x", 64)
2 state.solver.add(x > 0x30)
3 state.solver.add(x <0x61)
4 state.solver.eval(x) # 56
```


Simulation Manager

The simulation manager is the main control interface of angr.

It stores all the simulation states and allows you to symbolically execute code.

Simulation Manager



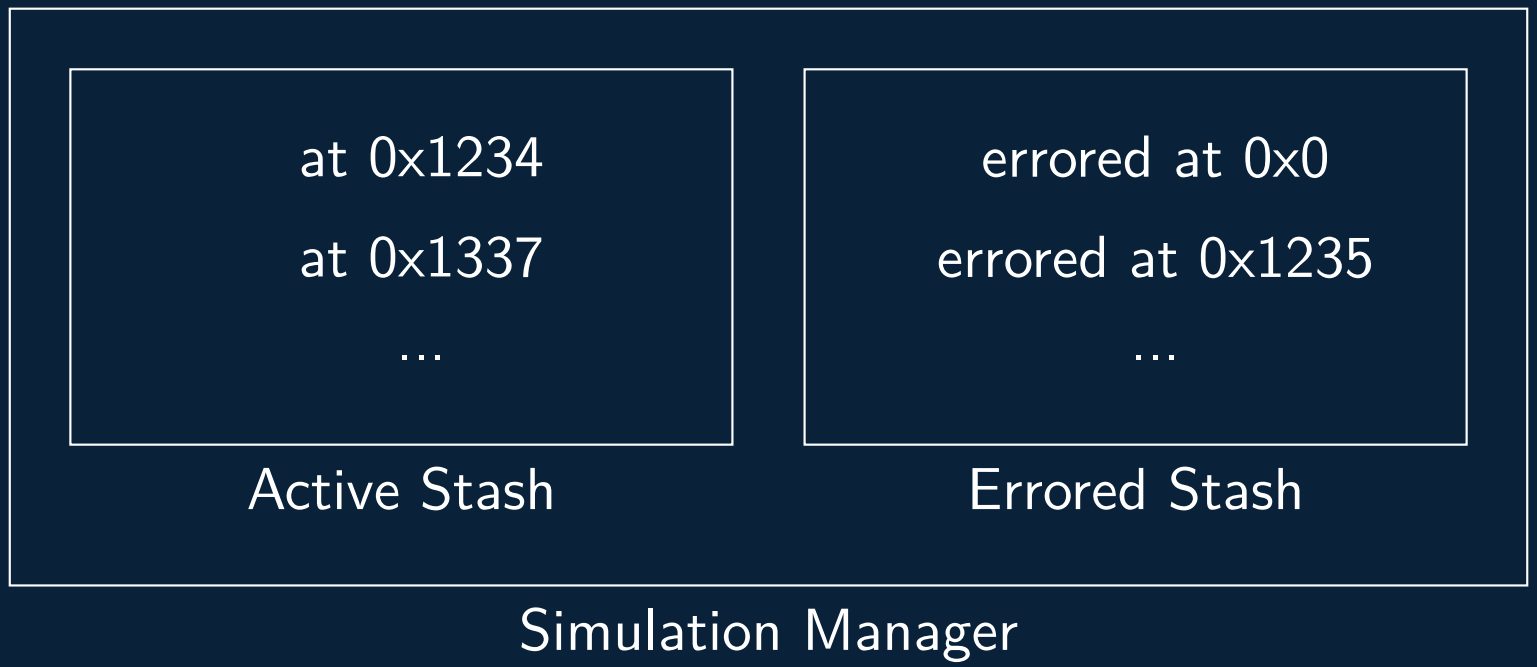
Simulation Manager

Simulation managers categorize simulation states in stashes.

Each stash is just a list of states.

The most important stashes are active, deadended, errored and unconstrained.

Simulation Manager



Simulation Manager

The active stash is the default stash for states.

The deadended stash holds states that have called exit or otherwise finished execution.

The errored stash holds states that ended with an error.

The unconstrained stash holds states that have a symbolic program counter and can no longer run.

Simulation Manager

The simulation manager exposes functions to allow us to perform the symbolic execution.

The step() function allows us to step each simulation state one basic block forward.

Each state in the active stash is stepped.

Simulation Manager

The run() function calls step until the active stash is empty.

The explore() function calls step until some condition is satisfied by a state.

Program State

Before defining a simulation manager, we must define the initial program state.

We can directly constrain the registers and memory of the initial state.

Program State

blank_state is a state with no constraints

entry_state is a state with rip constrained to the start of main, allows passing argc/argv.

call_state is a state with rip constrained to the start of some function, allows passing arguments.

Each of these also allows you to pass a Claripy AST as stdin.

Program State

We can also directly constrain registers and memory.

```
1 x = claripy.BVS("x", 64)
2 state.solver.add(x > 0x30)
3 state.solver.add(x < 0x61)
4 state.regs.rdi = x
```


Using Angr

In basic cases we just need an entry_state with unconstrained stdin.

Then we call explore() to find a state that reaches the win address.

We can then evaluate the input symbolic variables for stdin to find a valid input.

table of contents

Exponential Explosions

The number of states we need to keep track of is roughly 2^n where n is the number of basic blocks we've processed.

This prevents us from doing analysis on larger programs.

Exponential Explosions

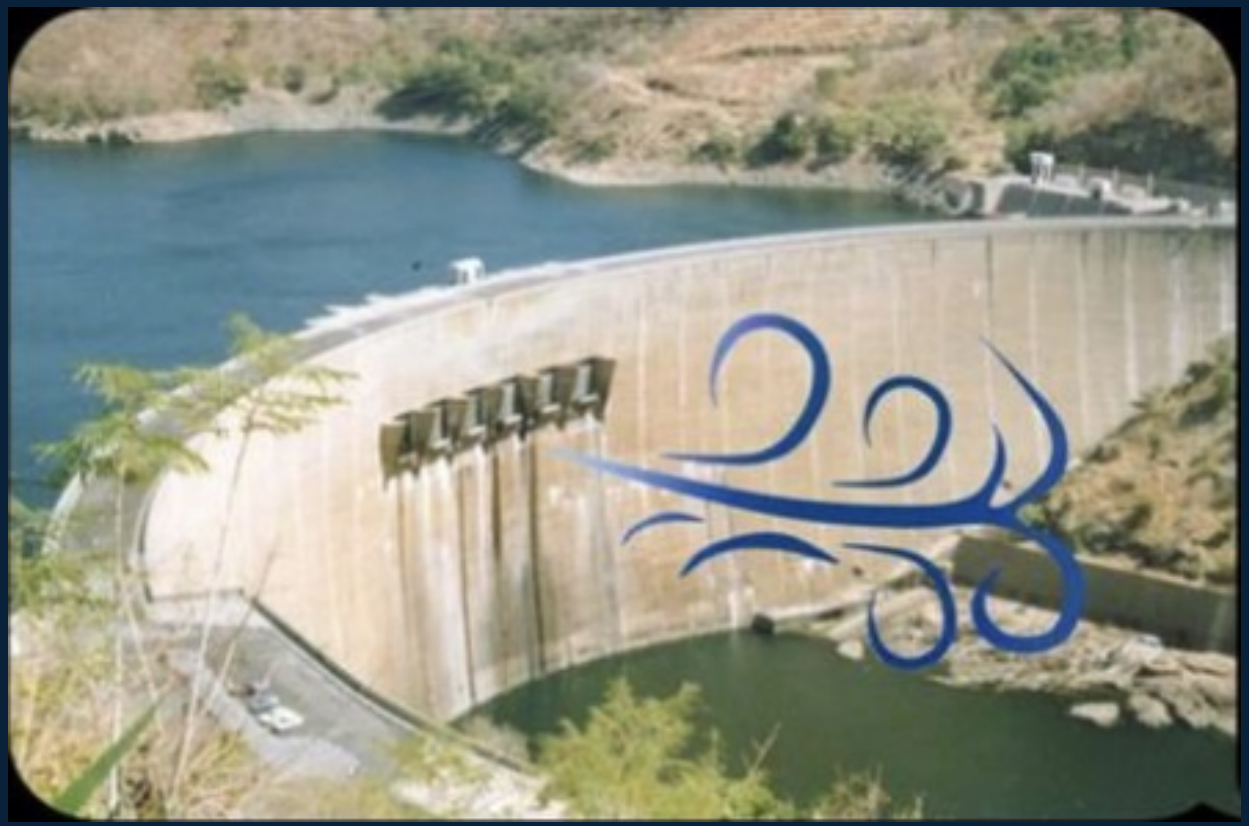


Figure 1: Dam that blows

Exponential Explosions

This makes it important that we analyze as little code as possible.

call_state becomes very useful since we generally only need to analyze one function.

Unconstrained Loops

Loops whose conditional variables are not constrained behave poorly in our current model.

We can either exit the loop or continue the loop.

This generates one new state per iteration of the loop.

If the loop condition is not well constrained, one state will always stay in the loop and will continually generate more states.

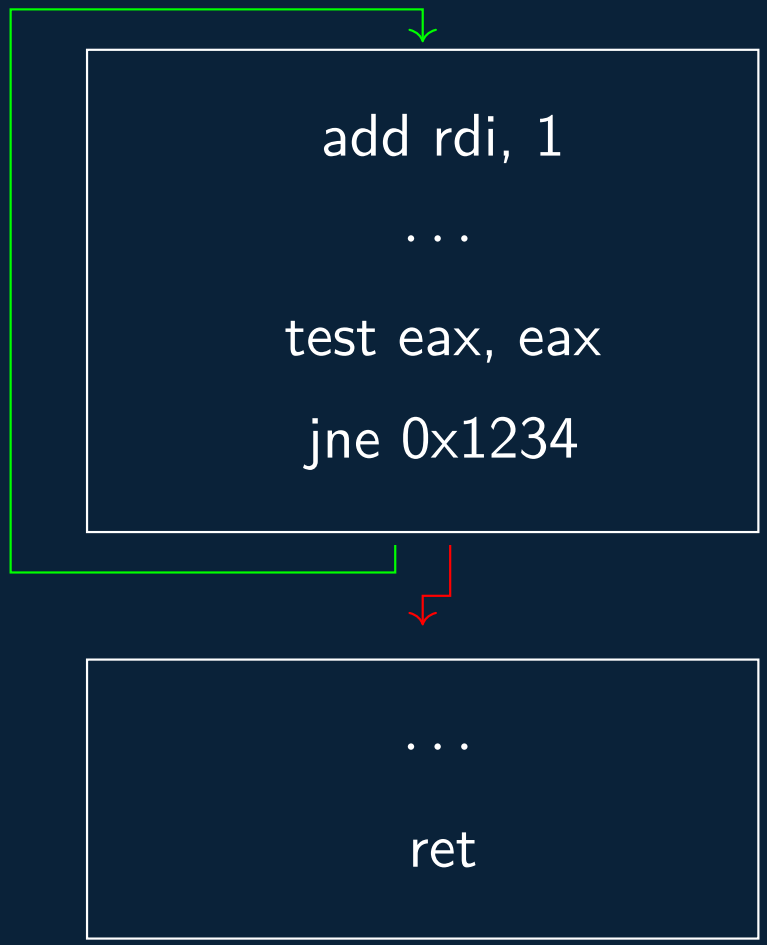
Unconstrained Loops

This commonly occurs when we pass symbolic strings to a program.

Ex: while loops that are conditioned on null bytes.

We can just say ”the string hasn’t ended yet”, and the state won’t leave the loop.

Unconstrained Loops



Unconstrained Loops

To fix this problem we need to make sure that loop variables are properly constrained in our initial state.

We also need to make sure that we constrain the last byte of strings to 0.

Libc Functions

Libc functions contains a ton of code that we don't really want to analyze.

This causes a massive slowdown on code that we don't even care about.

Libc Functions

Figure 1: well that sucks

Libc Functions

By default angr doesn't perform symbolic execution on libc functions.

Instead angr creates hooks for each libc function that will get called when a libc function is used in symbolic execution.

Libc Functions

Each hook runs some python code that properly constrains the program state without the overhead of analyzing libc.

This requires angr to have python hook implementations for every libc function.

Libc Functions

For example, the hook for malloc might look like

```
1 generate some address
2 rax == address
3 mark address as used
```

This is much faster than symbolically executing the ≈ 2000 lines that make up malloc.

Libc Functions

We can also write custom hooks to skip the overhead of analyzing arbitrary uninteresting functions.

This hooking model is also used with syscalls and other dynamic libraries.

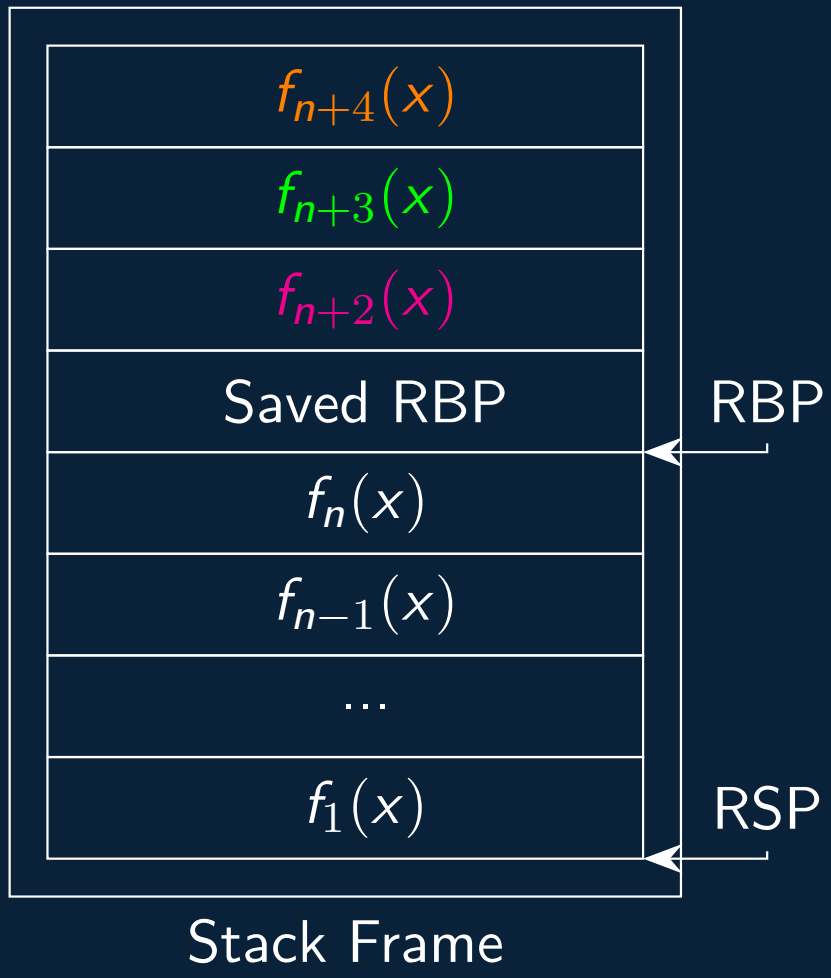
table of contents

Exploit Generation

Angr can also be useful for generating exploit payloads.

Ex: buffer overflow with some encoded data where the encoding is particularly difficult to predict.

Exploit Generation



Where $f(x)$ is particularly difficult to predict.

Exploit Generation

The simulation manager stash "unconstrained" holds states where the program counter is symbolic.

If we find an unconstrained state then we can solve for an input that sets rip to a specific output.

In other words, we constrain $[f_{n+1}(x) == \text{addr}]$ and SAT solve for x.

Exploit Generation

The same technique can be used to automate exploit generation.

table of contents

Static vs Dynamic Analysis

Static Analysis is generally not used for bug finding.

Instead a technique called fuzzing is used.

Static vs Dynamic Analysis

Fuzzing is the process of generating random inputs to a program and seeing what happens.

The concrete execution model of fuzzing is much faster than the symbolic execution model of static analysis.

Static vs Dynamic Analysis

The downside to fuzzing is that we're unlikely to hit bugs that have very few reproducing inputs.

Finding Bugs

Static Analysis is nice for finding very uncommon bugs that wouldn't be picked up by fuzzing.

We can mix symbolic execution and concrete execution into "concolic" execution.

This gives us the power of static analysis and the speed of dynamic analysis.

Finding Bugs

If we fuzz normally and find some very un-taken path, we can use angr to find an input that hits that path.

This gives us better code coverage than otherwise possible.