# Binary Analysis with angr

Or: VEX was a good idea

# Who am I? Who are we? Who cares?

---

- Researchers at the University of California Santa Barbara Seclab
- People interested in finding bugs in software
- People interested in publishing papers about finding bugs in software
- CTF players
- **People who want there to be a reasonable system for performing static analysis and symbolic execution on binary code**

# That system is called angr

(at least, that's the system we built)

**angr is a highly modular Python framework that performs binary analysis using VEX as an intermediate representation**

- The name "angr" is a pun on VEX, since, you know, when something is vexing it makes you angry
- Made of many interlocking parts to provide useful abstractions for analysis

# Part 1: the pile of abstractions called angr

# Interlocking part #1: PyVEX

---

PyVEX is a big FFI wrapper around libVEX.

For any sort of analysis to even start, we need to have an IRSB and then be able to look at it! PyVEX lets you do this.

```
>>>> import pyvex, archinfo
>>>> bb = pyvex.IRSB('\xc3', 0,
         archinfo.ArchAMD64())
>>>> bb.pp()
IRSB {
   t0:Ity_I64 t1:Ity_I64 t2:Ity_I64
t3:Ity_I64 t4:Ity_I64

   00 | ------ IMark(0x0, 1, 0) ------
   01 | t0 = GET:I64(rsp)
   02 | t1 = LDle:I64(t0)
   03 | t2 = Add64(t0,0x8)
   04 | PUT(rsp) = t2
   05 | t3 = Sub64(t2,0x80)
   06 | === AbiHint(0xt3, 128, t1) ===
   NEXT: PUT(rip) = t1; Ijk_Ret
}
```

# Interlocking part #1: PyVEX

---

There are python classes for each VEX struct, and enums are represented as strings.

Data is deepcopied out of libVEX between lifts so we don't run afoul of the memory management.

Technically independent of libVEX, lifters can be written in pure python! We have written lifters for AVR, MSP430, and Brainfuck.

```
>>>> bb.statements[3]
<pyvex.stmt.WrTmp object>
>>>> bb.statements[3].data
<pyvex.expr.Binop object>
>>>> bb.statements[3].data.op
'Iop_Add64'
>>>> bb.statements[3].data.args
[<pyvex.expr.RdTmp object>,
<pyvex.expr.Const object>]
```

# Interlocking part #4: SimuVEX

---

Symbolic execution with IRSBs

Technically supports execution from many other sources (plugin interface) but VEX was the first and is the best-supported. Also it's in the name.

Contains symbolic implementations of the effects of:

- Statements
- Expressions
- Operations
- Clean helpers
- Dirty helpers

The primary abstraction we get from simuvex is the **SimState**, a representation of program state at a given time. The symbolic execution process is one that produces **successors** to SimStates, each successor being a copy of its parent with additional data and **constraints**

# Interlocking part #4: SimuVEX

---

This is the part where we have to begin considering how we model our environment. SimuVEX must also handle:

- Modeling memory and registers
- Syscalls
- Files and other data sources from outside the program
- Providing symbolic summaries (SimProcedures) of common library functions

# Interlocking part #2: Claripy

Allows us to move execution from the domain of integers to anything else we could possibly imagine!

The most important other domain is **symbolic bitvectors**. This lets us build up symbolic trees of expressions over variables, add constraints on their value, and then solve for possible concrete values they could take on. This operation is backed up by z3.

Other domains are useful for special kinds of static analysis! See: abstract interpretation

```
>>>> import claripy
>>>> s = claripy.Solver()
>>>> a = claripy.BVS('a', 32)
>>>> s.add(a > 4)
>>>> s.add(a < 10)
>>>> s.eval(a, 10)
(9, 5, 7, 6, 8)

>>>> s.add((a + 1) % 2 == a / 2)
>>>> s.eval(a, 10)
ERROR: UNSATISFIABLE
```

# Interlude: Symbolic Execution Example

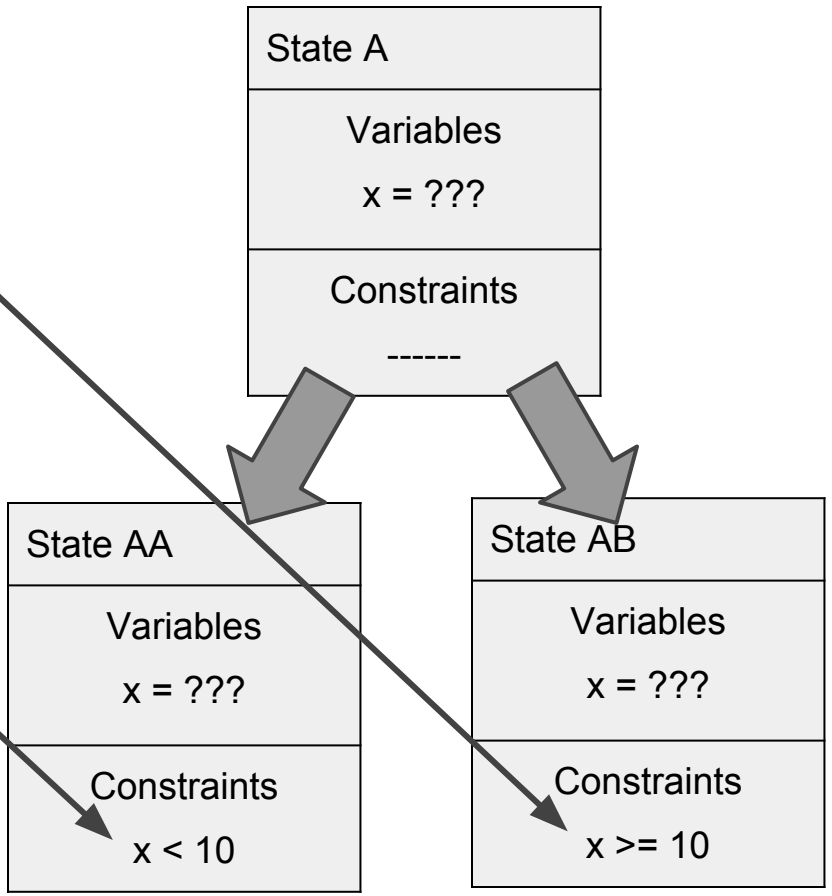
We're gonna show how symbolic execution executes a program and what we can do with that!

(these slides stolen from Every Single Angr Presentation Ever)

```
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

| State A |
| --- |
| Variables<br>x = ??? |
| Constraints<br>------ |

```
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

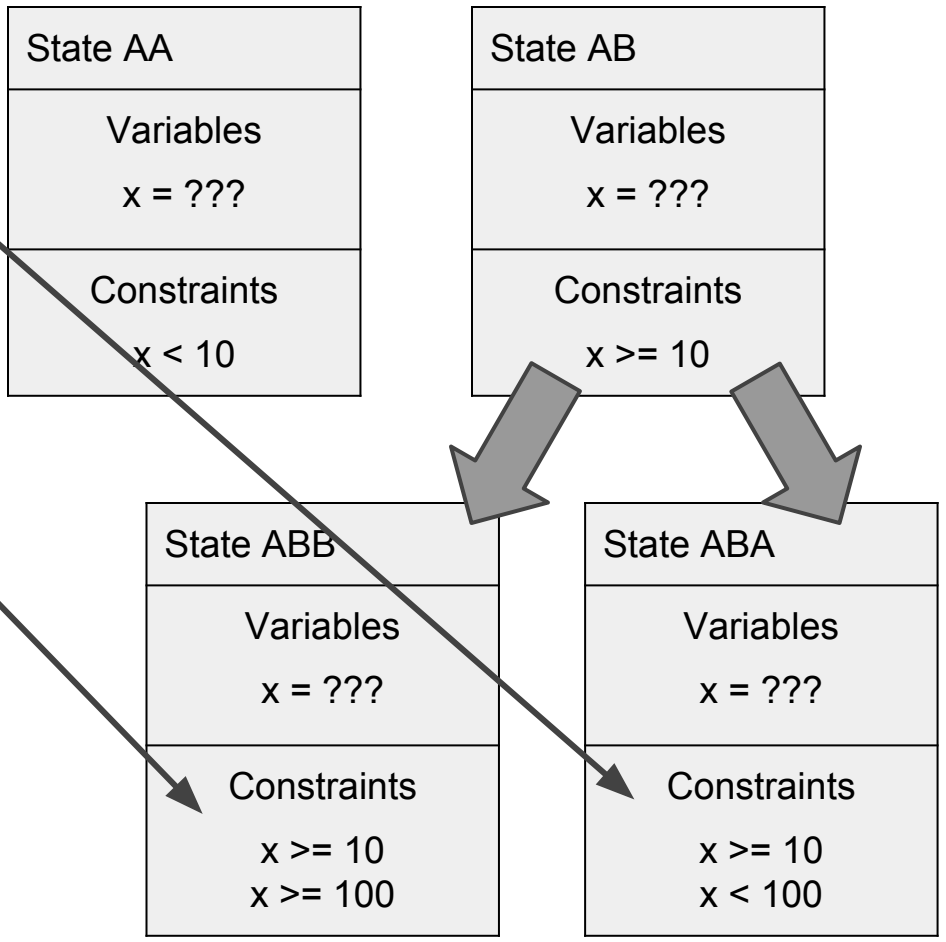



```
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

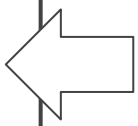
| State AA |
| --- |
| Variables<br>x = ??? |
| Constraints<br>x < 10 |

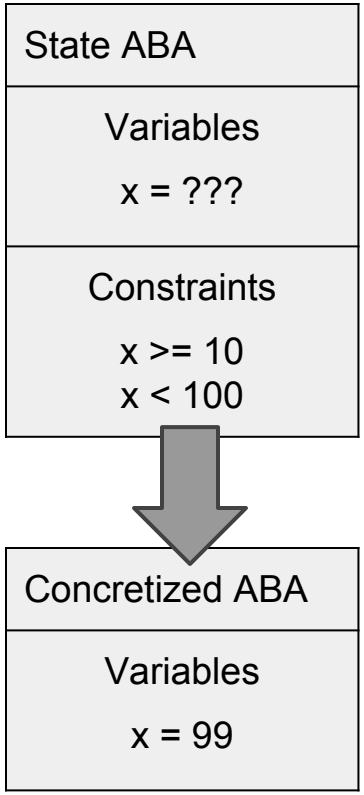
| State AB |
| --- |
| Variables<br>x = ??? |
| Constraints<br>x >= 10 |

```
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```



```
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```





# Interlocking part #3: CLE

---

- A binary loader.
- Very complicated.
- Not at all within the scope of this presentation.

**BASICALLY**, it provides the ability to turn an executable file and libraries and turn them into a usable address space.

(I would very much like to spend several hours talking about the challenges of designing a generic binary loader interface and then implementing the linux/windows/macos dynamic loaders on top of it, but that’s not why we’re here today)

# Interlocking part #5: angr

---

The analysis module! Ties all the abstractions together into a control interface: the **Project**.

Allows convenient access to symbolic execution and also to several built-in **analyses** that do a lot of common tasks, like CFG recovery, data-flow analysis, etc.

Has a **knowledge base** to accumulate analysis results

```
>>> import angr
>>> proj = angr.Project('./fauxware')
>>> cfg = proj.analyses.CFG()
>>> dict(proj.kb.functions)
{4195552L: <Function _init (0x4004e0)>,
 4195712L: <Function _start (0x400580)>,
 4195756L: <Function call_gmon_start
(0x4005ac)>,
 4195904L: <Function frame_dummy (0x400640)>,
 4195940L: <Function authenticate
(0x400664)>,
 4196077L: <Function accepted (0x4006ed)>,
 4196093L: <Function rejected (0x4006fd)>,
 4196125L: <Function main (0x40071d)>,
 4196320L: <Function __libc_csu_init
(0x4007e0)>,
 4196480L: <Function __do_global_ctors_aux
(0x400880)>}
>>> pg = proj.factory.path_group()
>>> pg.explore(find=0x4006ed)
>>> pg.found[0].state.posix.dumps(0)
'\x00\x00\x00\x00\x00\x00\x00\x00SOSNEAKY
\x00'
```

# That was a lot

angr is big and complicated, but a lot of care has been taken to make it a stack of useful abstractions so that any part of the binary analysis process can be easily instrumented.

# What can we do with angr?

## Analyze a lot of binaries

- Symbolic execution
- Built-in analyses: CFG, BinDiff, Disassembly, Backward-Slice, Data-Flow Analysis, Value-Set Analysis, etc
- Binary rewriting
- Type inference
- Symbolically-assisted fuzzing (driller)
- Automatic exploit generation
- Win 3rd place in the Cyber Grand Challenge

# What can we do with angr?

## Build a community

A lot of people are using angr for some reason!!

- > 100 people on #angr on freenode
- > 100 people on angr.slack.com
- Daily issues, pull requests, and discussion on github
- Patches have been submitted and friends have been made with other open source projects: z3, capstone, unicorn engine, qemu

# And all this is because we can lift binary code to the VEX IR and execute it symbolically!

---

Under the hood, pretty much every primitive operation that angr does is a call into SimuVEX to execute some code.

libVEX sure is great!

...but was it the only option?

# Part 2: a brief summary of other analysis IRs

# BAP - Binary Analysis Platform

---

BAP is developed by CMU for their research. Most notably, it powers Mayhem, which is their bug-finding tool.

CMU research's spinoff company, ForAllSecure, used Mayhem to win 1st place in the Cyber Grand Challenge.

**PROS**

- Written in ocaml
- Written by people with a solid theoretical background

**CONS**

- Written in ocaml
- The IR is tied to the larger analysis platform
- Only supports x86/amd64/arm
- When we started angr in 2013, BAP was heavily fragmented and very difficult to use. Since then it has been completely rewritten.

# REIL - Reverse Engineering Intermediate Language

---

REIL is a 2009 paper describing an IR that is ideal for binary analysis.

**PROS**

- Ideal for binary analysis

**CONS**

- Doesn't actually exist
- If you decide to write a binary lifter, you will spend three years writing a binary lifter

https://static.googleusercontent.com/media/www.zynamics.com/en//downloads/csw09.pdf

# LLVM - Low Level Virtual Machine

---

LLVM is the Clang IR. It is wildly popular for program analysis (as opposed to binary analysis).

**PROS**

- Robust library
- Large community and body of knowledge about how to use it

**CONS**

- Designed for compilers, cannot reason about e.g. register allocation. The parts of the LLVM assembler that do this are divorced from the actual IR.
- No official lifter from ANY binary format! Various communities have projects to do this for x86, but not very well.

# TCG - TinyCode (Generator)

---

This is the IR that qemu uses internally to do its translation, optimization, and JIT compilation!

**PROS**

- Official lifter available for TONS of languages

**CONS**

- Lifter is buried in the depths of qemu
- We at one point burned out one of our interns trying to extract it into a libTCG
- In the last few months, someone has successfully done this!

# VEX - not sure what this stands for, if anything

---

**PROS**

- Official lifter implementation available!
- Supports tons of architectures (anything valgrind supports)
- Designed for binary analysis and instrumentation
- Written in C
- Under active development
- Excellent ISA coverage for x86 and ARM

# VEX - not sure what this stands for, if anything

---

**CONS**

- Designed for the valgrind use-case: dynamically executing user-mode programs on a platform that could natively run the guest code
- The actual intermediate representation is never exported from the library
- There are approximately ten billion IROps and CCalls to implement
- Not truly a single static assignment (SSA) IR

# VEX is the only one where the cons aren't showstoppers

# Part 3: overcoming the problems with libVEX

# We forked VEX

and made a lot of changes to it

warning: this is the part where things get **really** technical

- Split LibVEX_Translate into LibVEX_Lift and LibVEX_Codegen
- Made multiarch mode actually work on all platforms (...sort of)
- Add options to disable some optimizations that are useful for valgrind but death for static analysis
- Remove restrictions that only make sense for userland programs
- Improve the meta-properties of the IR that dynamic execution ignores but static analysis needs
- TONS more

**We want to upstream all these changes!**

# Let’s take a look!

**https://github.com/angr/vex**

```
for c in $(git log | grep commit | cut -d ' ' -f 2 | tac); do git show
--color=always $c | less -R; done
```

# That was a lot

...that was actually way less than it could have been; I spent the last two weeks cleaning up our three-year mess of a commit history and packaging it into this series of thirty meaningful patches.

# Future ideas

Not urgent but sure would be cool

- Remove CCalls entirely
- Thread-safety
- Make multiarch work from big-endian hosts
- Support for more ISA features that valgrind doesn't have to care about (looking at you, x86 `enter`), including supervisor instructions
- Undo that one commit from a few weeks ago removing support for the mips mfc0 instruction
- There are several postprocessors in PyVEX that should probably be made into patches

# Future ideas

VERY urgent

- Change indentation from three spaces to literally anything else

# Licensing

---

libVEX is GPL. This is a little scary. We do not want angr to be GPL, at least partially because the university wants all our code to be BSD.

We believe that we have sufficiently insulated ourselves from libVEX (e.g. by making PyVEX able to use any number of lifter backends, of which libVEX is one), but this is a little sketchy, since SimuVEX needs, for example, to be able to enumerate all the IROps from libVEX.

This is a complicated legal issue - what parts of the VEX IR are inherant to libVEX, and are thus governed by the terms of the libVEX license? We don't really want to find out.

**We would greatly appreciate libVEX being re-licensed with a runtime library exception similar to the one in glibc.**

# That's pretty much it!

angr project

http://angr.io/
angr@lists.cs.ucsb.edu

Andrew Dutcher

@rhelmot