(State of) The Art of War: Offensive Techniques in Binary Analysis

Authors

Yan Shoshitaishvili
Christopher Salls
Mario Polino
John Grosen
Christophe Hauser
Giovanni Vigna
Ruoyu Wang
Nick Stephens
Andrew Dutcher
Siji Feng
Christopher Kruegel

http://ieeexplore.ieee.org/abstract/document/7546500/

Symbolic Execution with angr

Proactive Computer Security
2017

Petur Andrias Højgaard Jacobsen
Andrew Tristan Parli
Robert Schannong Rasmussen

Contents (20 pages)

Introduction: Why angr?

- Tool for the research community for reproducing, improving and creating analysis techniques
- Design Goals
  - cross-architecture/platform support
  - modularity for different analysis
  - easy reproduction of analysis techniques (under-constrained symbolic execution = 2 days)
- For use in the DARPA Cyber Grand Challenge (i.e., automating binary analysis)

Automated Binary Analysis

2 main trade-offs for feasibility

Replayability

- understanding how to trigger the vulnerability

Semantic insight

- understanding why code is executed
- understanding what part of the input caused the behavior

Background

Static Vulnerability Discovery

✗ Control Flow Graph (CFG) Recovery

✗ Value Set Analysis (VSA)

Dynamic Vulnerability Discovery

✓ Dynamic Symbolic Execution (SE)
  with veritesting to mitigate path explosion

✓ Under-constrained Symbolic Execution

✗ Symbolic Assisted Fuzzing (angr + AFL = Driller)

Symbolic Execution: Intuition

$$x \in N_0$$

$$x + y = 5$$

Q: What is the value of x?

Symbolic Execution: Intuition

$$x \in N_0$$

$$x + y = 5$$

Q: What is the value of x?

A: infinitely many values, but depends on y

Symbolic Execution: Intuition

$$x \in N_0$$

$$x + y = 5$$

$$xy = 0$$

Q: What is the value of x?

Symbolic Execution: Intuition

$$x \in N_0$$

$$x + y = 5$$

$$xy = 0$$

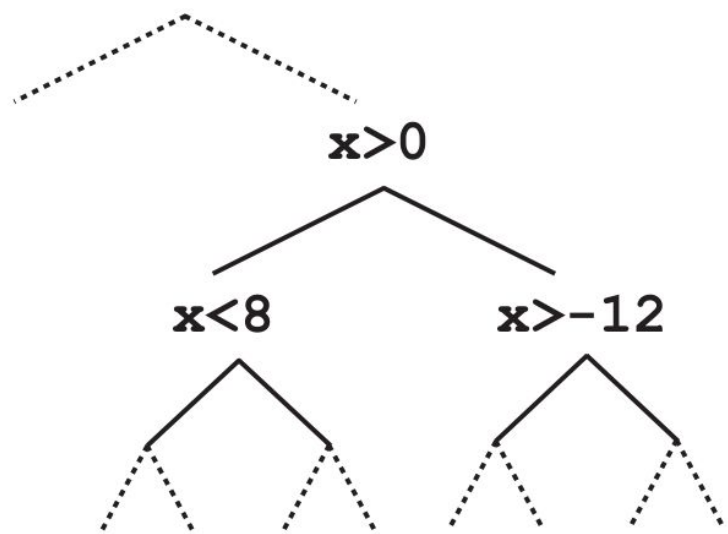
Q: What is the value of x?

A: $\mathbf{x} \in \{0,5\}$

Symbolic Execution: Intuition

During execution we maintain a set of paths, each with an associated set of symbolic values and constraints.

```
I = <input>;
x = I-2;
if (x > 0) {
  if (x < 8) {
    ...
  ...
} else {
  if (x > -12) {
    ...
  ...
}
```



Symbolic Execution: Intuition

Want to know more?

There is a long lecture from MITOpenCourseware[3]

Check it out!

Where there is anger
there is always pain
underneath.

Analysis Engine: Overview

- Implemented mainly in Python
- Intended to be used with IPython
- Can be run in PyPy to reduce language overhead (i.e., increased speed)

Analysis Engine: CLE

PROBLEM: How do we run analysis on a binary?

SOLUTION: Use a “binary loader”

- CLE Loads Everything
- Loads the binary and library dependencies in a meaningful way for angr to work with…

Analysis Engine: CLE

In angr the Project class exposes the CLE module to the user

```
>>> import angr

>>> b = angr.Project(“/bin/grep”)

>>> print b.filename, hex(b.entry)

/bin/grep 0x404ca8
```

Analysis Engine: libVEX

PROBLEM: How do we support different architectures (i.e., ARM, MIPS, x86, etc.)?

SOLUTION: Translate the binary to a common intermediate representation (IR)

- Known as an “IR lifter”
- libVEX is implemented by the Valgrind project
- Produces VEX IR which is exposed to Python by PyVEX

Analysis Engine: libVEX

In angr the PyVEX module is exposed by the `Project.factory.block` interface.

```
>>> import angr
>>> b = angr.Project(“/bin/grep”)
>>> b.factory.block(b.entry).size    # in bytes
41

# pretty-print the VEX IR of the entry block
>>> ir = b.factory.block(b.entry).vex
>>> ir.pp()
```

Analysis Engine: Claripy Solvers

- angr’s data model provider
- Claripy has different Solvers for different analyses
- Uses `Z3.Solver` for SE constraint solving[2]
- provides symbolic values through AST’s for SE
  - bit-vectors   (`Claripy.BVS`)
- SimVEX module handles most internal interactions with Claripy.

Analysis Engine: Claripy Solvers

```
# create a solver
>>> import claripy
>>> s = claripy.Solver()

# symbolic 4-bit value named `x`
>>> sym_x = claripy.BVS('x', 4)
>>> assert sorted(s.eval(sym_x, 16)) == range(16)

# now reset and add an unsigned less than condition
>>> sym_x = claripy.BVS('x', 4)
>>> s.add(claripy.ULT(sym_x, 5))
>>> print sorted(s.eval(sym_x, 16))
[0, 1, 2, 3, 4]
```

Analysis Engine: SimuVEX

- controls program state representation and makes modifications to the state
- a symbolic VEX emulator (SimEngines)
  input: state + VEX block
  output: state[]
- tracks both concrete and symbolic values for memory, registers, open files, etc.
- `SimState` objects are accessed from the `Project.factory`

Analysis Engine: SimuVEX

```
>>> import angr, simuvex
>>> b = angr.Project(‘/bin/grep’)

# we can get a state at the entry point
>>> s = b.factory.entry_state()

# we can access data from the register values
>>> print “Stack pointer is at: ”, s.regs.sp
>>> print “Instruction pointer is at: “, s.regs.ip

# s.se is the solver engine with symbolic constraints on the state
# also note the endianness is big-endian by default
>>> print s.se.any_int(s.regs.eax)
>>> print s.se.any_int(s.memory.load(0x2000, 4, endness='Iend_LE'))
```

Analysis Engine: SimuVEX

```
# we can create and store data in registers, in memory, on the stack...
>>> aaaa = claripy.BVV(0x41414141, 32)   # 32-bit 'aaaa'
>>> s.regs.eax = aaaa
>>> s.memory.store(0x1000, aaaa)
>>> s.stack_push(aaaa)
>>> s.stack_pop(aaaa)

# we can copy states
s1 = s.copy()
s2 = s.copy()

# we can merge states
(s_merged, m, any_merged) = s1.merge(s2)

# then analyze the merged state
s_merged.se.any_n_int(aaaa)
```

Analysis Engine: SimuVEX

```
# we can check is a value is symbolic
>>> assert s.se.symbolic(aaaa)

# we can check which symbolic variables make up an expression
>>> print s.se.variables(aaaa)
```

SE Top Level

- Path primary interface to control execution
- PathGroup a lot of Paths being executed
- Hook allows angr to intercept program execution at a specific memory address.
- SimProcedures symbolically implement a function (i.e., library functions)

SE Top Level: Paths

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

# get a path
>>> p = b.factory.path()

# the path will load at the binary's entry point
>>> p.addr == b.entry

>>> p.length     # number of VEX blocks analyzed in path (0 right now)
>>> p.callstack  # current backtrace

# see how many successor paths exist
>>> p.step()
>>> print len(p.successors)
```

SE Top Level: PathGroups

- These are the self-professed future of angr
- Basically bulk execution of Paths
- Paths organized into stashes
  (i.e., found, avoided, deadended, errored, etc.)
- PathGroup.Explorer() allows for stepping using find, avoid and until parameters

SE Top Level: PathGroups

```
>>> import angr
>>> p = angr.Project(‘any_binary’)
>>> pg = p.factory.path_group()

# try to reach a specific address
>>> pg.explore(find=0x4016A0)

# OR try to reach a specific address
>>> pg.explore(find=0x4016A0, avoid=[0x403200, 0x4032A1])

# OR try to find specific output
>>> pg.explore(find=lambda p: “Approved!” in p.state.posix.dumps(1))
>>> pg.found[0].state.posix.dumps(1)    # stdout
enter passwd: Approved!
>>> print pg.found[0].state.posix.dumps(0)    # stdin, the password
im_so_angr_e
```

SE Top Level: Hooks

- hooks let angr intercept a binary's execution at a specific memory address and redefine its behavior
- `Project.hook(address, procedure, length)`
- the procedure can be a Python function or `SimProcedure`

SE Top Level: Hooks

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



# hook the mem address with given SimProcedure,
# then skip length bytes and resume execution
>>> b.hook(0x10000, set_rax, length=5)

>>> b.is_hooked(0x10000)
>>> b.unhook(0x10000)

# insert hook by symbol
>>> b.hook_symbol('strlen',
                  simuvex.SimProcedures['stubs']['ReturnUnconstrained'])
```

SE Top Level: SimProcedure

```
>>> from simuvex import SimProcedure
>>> from angr import Hook, Project
>>> project = Project('any_binary')

# define a SimProcedure with a run() method
>>> class BypassMain(SimProcedure):
...     def run(self, argc, argv):
...         print 'argc=%s and argv=%s' % (argc, argv)
...         return 0

# assuming a binary with symbols
>>> project.hook(project.kb.labels.lookup('main'), Hook(BypassMain))
>>> pg = project.factory.path_group()

# step until no more active paths
>>> pg.run()
```

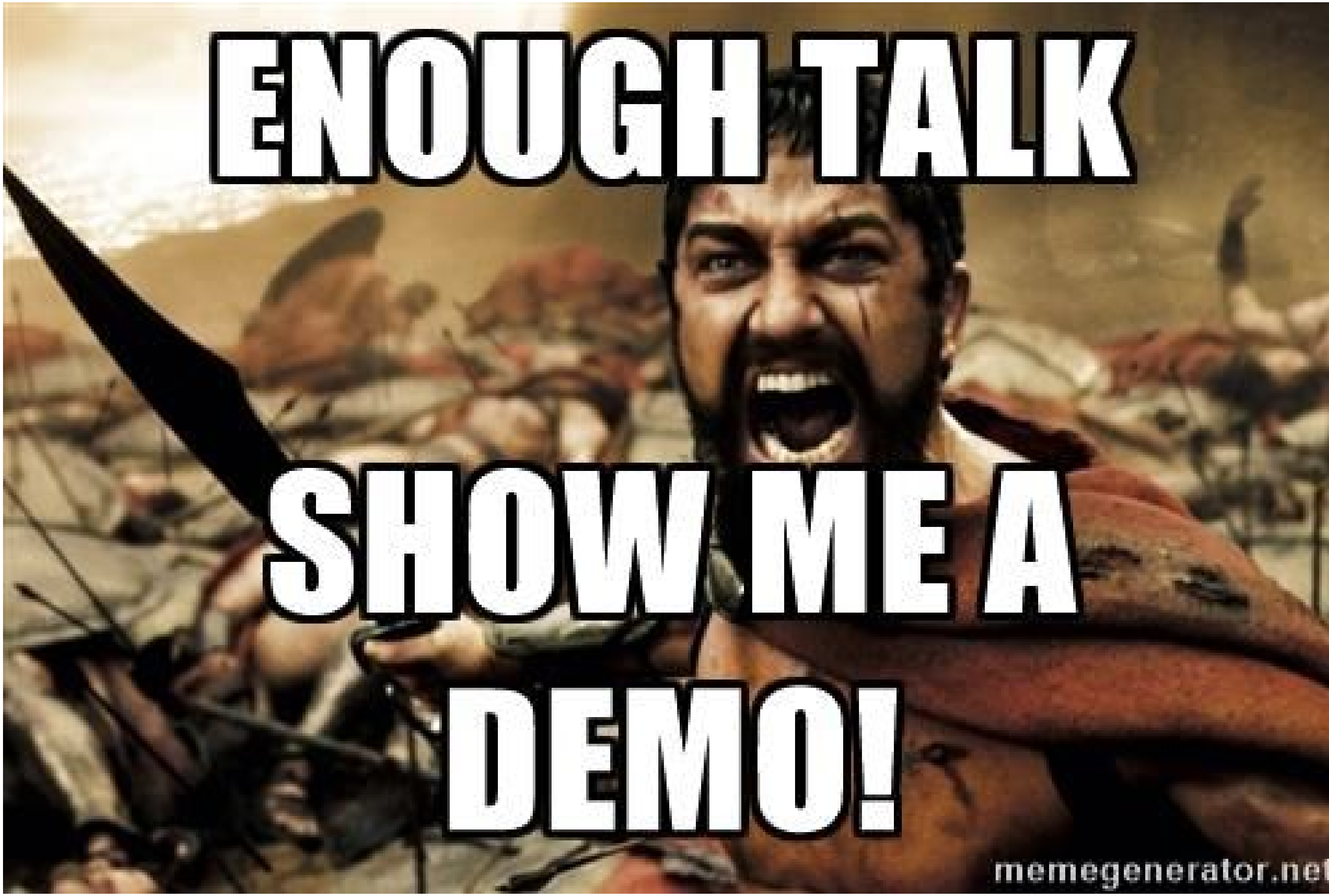
Implementation: Dynamic SE

- Uses `Claripy` interface into `Z3.Solver` to populate the symbolic memory model
- `Path` objects for each execution path
- Managed in `PathGroup`, allows
  - splitting
  - merging
  - filtering

Implementation: Under-constrained SE

- UC-angr is modeled after UC-KLEE with some differences
  - Global data is under-constrained
  - 64 path limit on functions
  - False positive filtering

```
# Use a blank state and arbitrary starting point
>>> import angr
>>> point_of_entry = 0x0800b000
>>> b = angr.Project('any_binary')
>>> start = b.factory.blank_state(addr=point_of_entry)
>>> path = b.factory.path(start)
```


ENOUGH TALK
SHOW ME A
DEMO!
memegenerator.net

Comparative Evaluation

Technique	Replayable	Semantic Insight	Scalability	Crashes	False Positives
Dynamic Symbolic Execution	Yes	High	Low	16	0
Dynamic Symbolic Execution + Veritesting	Yes	High	Medium	23	0
Under-constrained Symbolic Execution	No	High	High	25	346

References

[1]
Shoshitaishvili, Yan, et al. "SOK:(State of) The Art of War: Offensive Techniques in Binary Analysis." *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016.

[2]
De Moura, Leonardo, and Nikolaj Bjørner. "Z3: An efficient SMT solver." *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2008.

[3]
Solar-Lezama. “Lecture 10: Symbolic Execution.” *Computer Systems Security*. MITOpenCourseWare, MIT. 2014. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-858-computer-systems-security-fall-2014/video-lectures/lecture-10-symbolic-execution/

<EOF>