MONTANA STATE UNIVERSITY

10/21/2025

Malicious Code Analysis

Fangtian Zhong
CSCI 591

Gianforte School of Computing
Norm Asbjornson College of Engineering
E-mail: fangtian.zhong@montana.edu

Part One

01

angr

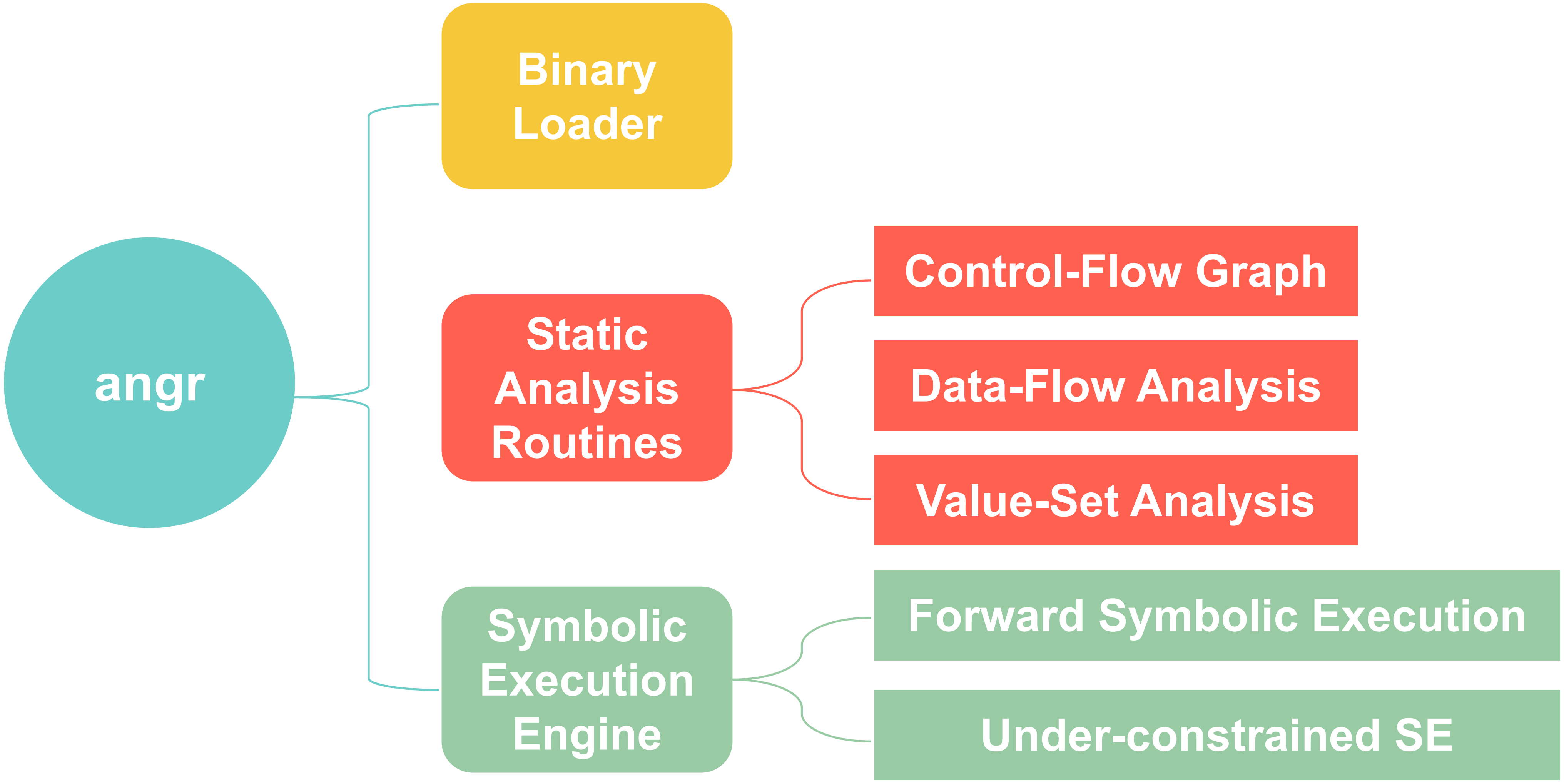
angr

- Framework for the analysis of binaries
- Supports a number of architectures
  - x86, x64, MIPS, ARM, PPC, etc.
- http://angr.io
- https://github.com/angr

angr issues

- angr is heavily developed and continuously re-worked
- Backward-compatibility is not a priority
- Expect some level of frustration
- Expect some things in this tutorial to be outdated

angr Components



Workflow

- Load a binary
- Translate to an intermediate representation
- Instrument
- Analyze
- Examine results

angr's Loader: CLE

- Before analyzing a program, it is necessary to load it into memory and parse it.
- The CLE module is responsible for loading a program into the analysis framework.
  - ELF
  - PE
  - IDA Pro binaries
  - Blobs
- All the information is accessible from the Project object.

angr's IR: VEX

- Binaries are lifted into VEX (Valgrind's IR).
- An intermediate representation allows for the abstraction of architecture-dependent features.
  - 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, etc.
  - Memory access: Different architectures access memory in different ways. For example, ARM can access memory in both little-endian and big-endian mode.

angr's IR: VEX

- Memory segmentation: Some architectures, such as x86, support memory segmentation through the use of special segment registers. The IR understands such memory access mechanisms and abstracts them away.
- 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 challenging. The IR makes these side effects explicit.

- The VEX representation is accessed using the PyVEX module.

angr's Project

- The basic object that represents the current analysis.
- Provides access everything else, such as the loader and the analysis backends.

```
print(proj.arch)
<Arch X86 (LE)>
print(proj.filename)
CIH.exe
print(proj.loader.main_object)
<PE Object CIH.exe, maps [0x1020000:0x10263ff]>
```

project.factory-blocks

- project.factory provides constructors for common objects, such as basic blocks.

```
>> block = p.factory.block(proj.entry)
>> block.pp()
1023750  mov    eax, dword ptr  fs:[0x0]
1023756  push   ebp
1023757  mov    ebp, esp
1023759  push   -0x1
102375b  push   0x10210f8
1023760  push   0x1023878
1023765  push   eax
1023766  mov    eax, dword ptr
[0x10249bc]
102376b  mov    dword ptr  fs:[0x0], esp
```

```
1023772  mov    ecx, dword ptr [0x1021008]
1023778  sub    esp, 0x1c
102377b  mov    dword ptr [ecx], eax
102377d  mov    edx, dword ptr
[0x10249b8]
1023783  mov    eax, dword ptr [0x1021050]
1023788  push   ebx
1023789  push   esi
102378a  push   edi
102378b  mov    dword ptr [ebp-0x18], esp
102378e  mov    dword ptr [eax], edx
1023790  call   0x102386c
```

project.factory-States

- A SimState contains a snapshot of the program state.
  - Memory (state.mem)
  - Registers (state.regs)
  - Filesystem
  - ...

```
>> state = proj.factory.entry_state()
>> state.regs.esp
<BV32 0xfffefef8>
>> state.mem[proj.entry].int.resolved
<BV32 0x1af1e9>
```

- SimStates are immutable

Bitvectors

- Bitvectors are used to represent integers in a way that is consistent with how the architecture represents them.

```
>> bv = state.solver.BVV(0xFF, 16)
>> print(bv)
<BV16 0xff>
>> state.solver.eval(bv)
0xff
```

Bitvectors

- Bitvectors can be stored in memory directly, however if a Python integer is used, it is automatically translated into a bitvector.

```
>> state.regs.eax = state.solver.BVV(3, 32)
>> state.regs.eax
<BV32 0x3>
>> state.regs.eax = 4
>>state.regs.eax
<BV32 0x4>
```

Memory

- The SimState's memory is accessed like an array
- Access to a location need to be qualified by type

```
>> state.mem[100000].uint8_t = 0
>> state.mem[100000].uint32_t = state.solver.BVV(3, 32)
```

- Values in memory can be retrieved using the .resolved (bitvector) and .concrete (Python int)

```
>> state.mem[100000].uint32_t.resolved
<BV32 0x3>
>> state.mem[100000].uint32_t.concrete
0x3
```

Simulation Managers

- Simulation Managers are responsible for producing new SimStates given an initial set of SimStates.
- SimStates are organized in stashes.
  - active: SimStates being executed
  - deadended: SimStates that cannot progress
  - unconstrained: SimStates in which the instruction pointer can be controlled (e.g., it has a symbolic value)
  - unsat: SimStates whose constraints are unsatisfiable
  - <custom>
- step() executes a basic block of the active SimStates.

Simulation Managers

```
>> sm = proj.factory.simulation_manager(state)
>> sm.active
 [<SimState @ 0x14001128a>]
>> sm.step()
<SimulationManager with 1 active>
…
>> sm.step()
<SimulationManager with 2 active>
…
>> sm.step()
<SimulationManager with 4 active>
```

History

- state.history allows one to access the historical execution path up to the current state.
  - state.history.parent: The parent state
  - state.history.bbl_addrs: The basic block addresses executed by the state
  - state.history.jump_guards: The conditions guarding each of the branches that the state has encountered

SimProcedures

- SimProcedures are procedures that model calls to external functions, specifying the effect of the function on the SimState.
  - Libraries
  - System calls

```
>> angr.SIM_PROCEDURES['libc'].keys()
['strncmp',
'sscanf',
 'snprintf', …
```

- They are also used for hooking, i.e., to associate an address with a SimProcedure.
  - When the address is reached the SimProcedure is invoked

SimProcedures

```
>> func = angr.SIM_PROCEDURES['stubs']['ReturnUnconstrained']
# Func is actually a class
>> p.hook(0x10000, func())
>> p.is_hooked(0x10000)
True
>> p.hooked_by(0x10000)
<SimProcedure ReturnUnconstrained>
```

Symbolic Values

- Symbolic values are “placeholders” for an unknown value
- Operations on symbolic values return an AST representing the operations (accessible with .op and .args)

Symbolic Values

```
>> x = state.solver.BVS("x", 64)
<BV64 x_3_64>
>> x + 1
 <BV64 x_3_64 + 0x1>
>> (x + 1) / 2 * x
<BV64 ((x_3_64 + 0x1) / 0x2) * x_3_64>
>> ((x + 1) / 2 * x).op
'__mul__'
>> ((x + 1) / 2 * x).args
(<BV64 (x_3_64 + 0x1) / 0x2>, <BV64 x_3_64>)
```

Symbolic Constraints

- Comparing symbolic values will return an AST with a Boolean type, which represents a constraint
- Constraints can be added to a solver
- The solver can then be asked to evaluate the constraints

Symbolic Constraints

```
>> state.solver.add(a > b)
[<Bool a_5_8 > b_6_8>]
>> state.solver.add(a == b * 2)
[<Bool a_5_8 == (b_6_8 * 2)>]
>> state.solver.add(b > 6)
[<Bool b_6_8 > 6>]
>> state.solver.constraints
[<Bool a_5_8 > b_6_8>,<Bool a_5_8==(b_6_8 * 2)>,<Bool b_6_8 > 6>]
>> state.solver.eval(a)
0xeL
>> state.solver.eval(b)
0x7L
```

Execution Engines

- Execution Engines are responsible for evolving the state when step() is invoked on an Execution Manager.
  - The failure engine is invoked when the previous step took us to some un-continuable state
  - The syscall engine is invoked when the previous step ended in a syscall
  - The hook engine is invoked if the current address is hooked
  - The unicorn engine is invoked when the UNICORN state option is enabled and there is no symbolic data in the state
  - The VEX engine is invoked as the final fallback

Breakpoints

- angr allows one to set breakpoints and test for sophisticated conditions.
  - Memory and registers reads/writes
  - A new symbolic variable is created
  - A call instruction is invoked
  - ...

```
>> def debug_f (state):
...     print "State %s just performed a memory write!”
>>> s.inspect.b('mem_write', when=angr.BP_AFTER, action=debug_f)
```

Analyses

- angr provides a number of analyses:

```
>> p.analyses.[TAB]
p.analyses.BackwardSlice        p.analyses.CFGFast             p.analyses.Reassembler
p.analyses.BinaryOptimizer      p.analyses.CongruencyCheck     p.analyses.reload_analyses
p.analyses.BinDiff              p.analyses.DDG                 p.analyses.StaticHooker
p.analyses.BoyScout             p.analyses.DFG                 p.analyses.VariableRecovery
p.analyses.CalleeCleanupFinder  p.analyses.Disassembly         p.analyses.VariableRecoveryFast
p.analyses.CDG                  p.analyses.GirlScout           p.analyses.Veritesting
p.analyses.CFG                  p.analyses.Identifier          p.analyses.VFG
p.analyses.CFGAccurate          p.analyses.LoopFinder          p.analyses.VSA_DDG
```

- The details about the analysis can be found in the API documentation

MONTANA STATE UNIVERSITY

THE END

Fangtian Zhong

CSCI 591

Gianforte School of Computing
Norm Asbjornson College of Engineering
E-mail: fangtian.zhong@montana.edu

10/21/2025