Symbolic Execution

Introduction to Security (192.019)

Lorenzo **Veronese**

Security & **Privacy** Research Unit (192-06)
https://**secpriv.wien**

Outline

- Introduction to SMT Solvers and the Z3 theorem prover
  - **Goal:** understand one of the main building blocks of symbolic execution
- Symbolic execution using `angr`
  - **Goal:** apply the theory of symbolic execution using a practical binary analysis framework

Satisfiability Modulo Theories (SMT)

Satisfiability Modulo Theories (SMT)

- **Boolean Satisfiability Problem (SAT)**

  Is there an interpretation of variables that makes a given formula true?

  $p \vee \neg q$ **is** sat $\qquad$ $p \wedge \neg p$ **is** unsat

  - SAT is NP-complete!

- **Satisfiability Modulo Theories (SMT)**

  Formulas in which some symbols have a specific interpretation:

  - e.g, *linear arithmetic constraints*, *arrays*, *uninterpreted functions*, *bit-vectors*

$$2x + 2x - z \leq 4 \wedge q$$

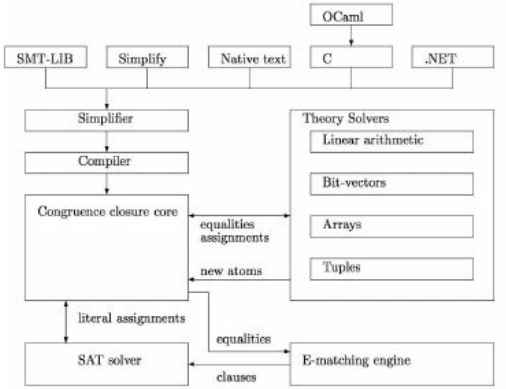
$$read(array, i - 2) = f(y - x + 1)$$

Z3 Theorem Prover

Efficient SMT Solver

- Z3 is a theorem prover from Microsoft Research that can efficiently solve SMT problems (and more!)

Released	February 2007
Size	500K+ loc
License	MIT
Bindings	python, c/c++, ocaml, java, ...



- It can be used as a standalone executable or as a library
  - We will use the python bindings:

```
pip install --user z3-solver
```

Z3 Theorem Prover

Example

- A formula is **satisfiable** if it has an interpretation that makes it logically true
- A formula is **valid** if it is logically true in any interpretation

Satisfiability

Is this formula satisfiable?

$$(p \vee \neg q) \wedge (\neg p \vee q \vee r) \wedge \neg p$$

```
from z3 import *

p = Bool('p')
q = Bool('q')
r = Bool('r')

s = Solver()
s.add( Or(p, Not(q)), Or(Not(p), q, r), Not(p) )

print( s.check() )
print( s.model() )
```

```
sat
[p = False, q = False, r = False]
```

Z3 Theorem Prover

Example

- A formula is **satisfiable** if it has an interpretation that makes it logically true
- A formula is **valid** if it is logically true in any interpretation

Validity

Is this formula valid?

$$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q \quad \textit{(De Morgan)}$$

It is equivalent to show that:

$$\neg(\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q)$$ **is** unsatisfiable

```
from z3 import *

p = Bool('p')
q = Bool('q')

s = Solver()
s.add(Not( Not(Or(p, q)) == And(Not(p), Not(q)) ))

print( s.check() )   unsat
```

Z3 Theorem Prover

Example

- A formula is **satisfiable** if it has an interpretation that makes it logically true
- A formula is **valid** if it is logically true in any interpretation

Theories

(Mathematical) Integers

```
from z3 import *

x = Int('x')
y = Int('y')
s = Solver()

s.add( 2*x + y == 12 )
s.add( x == 14 )

s.check()
print s.model()
```

```
[x = 14, y = -16]
```

BitVectors (Machine Integers)

```
from z3 import *

x = BitVec('x', 32)
y = BitVec('y', 32)
s = Solver()

s.add( Extract(31,24, x) == 10 )
s.add( x + y == 23 )

s.check()
print s.model()
```

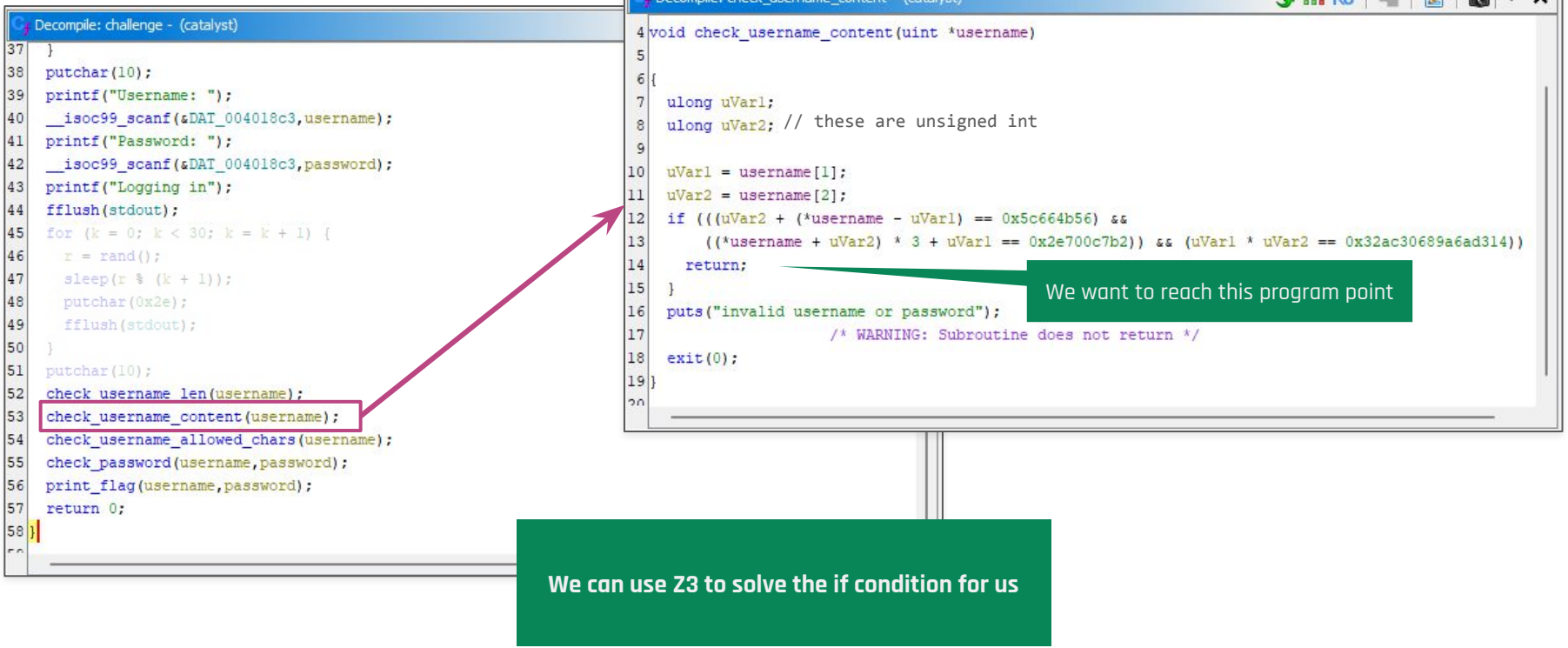
```
[y = 4127195159, x = 167772160]
```

CTF Challenge

`RE3: Catalyst System` **AlexCTF 2017** Reversing 150 points

- Simple crackme challenge: find the password to get the flag
- Our running example for today

CTF Challenge



CTF Challenge

```
from z3 import *
s = Solver()
```

```
username0 = BitVec('username0', 32)
username1 = BitVec('username1', 32)
username2 = BitVec('username2', 32)
uVar1 = BitVec('uVar1', 32)
uVar2 = BitVec('uVar2', 32)
```

```
s.add(uVar1 == username1)
s.add(uVar2 == username2)
s.add((uVar2 + (username0 - uVar1) == 0x5c664b56))
s.add((username0 + uVar2) * 3 + uVar1 == 0x2e700c7b2)
s.add((uVar1 * uVar2 == 0x32ac30689a6ad314))
```

```
username = (username0, username1, username2)
for var in username:
    for hi,lo in [(31,24), (23, 16), (15, 8), (7, 0)]:
      byte = Extract(hi, lo, var)
      s.add(Or(And(byte >= ord('a'), byte <= ord('z')), byte == ord('_')))
```

Decompile: check_username_content - (catalyst)

```
4  void check_username_content(uint *username)
5
6  {
7    ulong uVar1;
8    ulong uVar2; // these are unsigned int
9
10   uVar1 = username[1];
11   uVar2 = username[2];
12   if (((uVar2 + (*username - uVar1) == 0x5c664b56) &&
13       ((*username + uVar2) * 3 + uVar1 == 0x2e700c7b2)) && (uVar1 * uVar2 == 0x32ac30689a6ad314))
14     return;
15   }
16   puts("invalid username or password");
17                     /* WARNING: Subroutine does not return */
18   exit(0);
19 }
```

We declare our symbols to be 32 bit integers
The username is split into 3 4-byte symbols

We add all constraints to the solver
Note that we split the && into separate add calls

The solver gives us one of the possible solutions in no particular order.

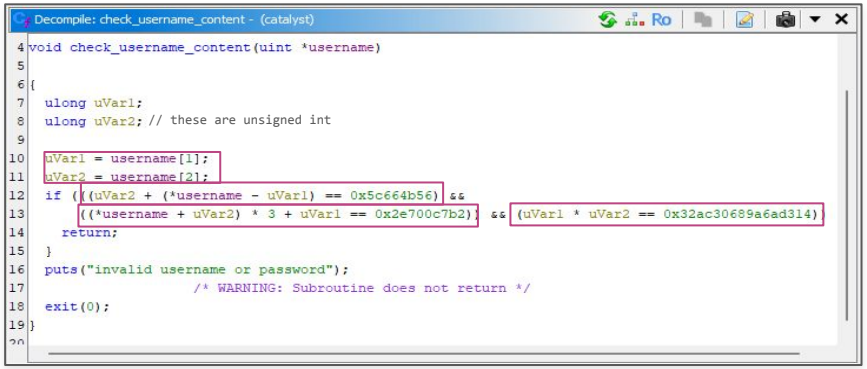
Normally we are interested in a specific solution:
All username chars are lowercase letters or `_`

CTF Challenge

```
if s.check() == sat:
        m = s.model()
        print(m)
        print(b''.join(
                struct.pack('I', m[x].as_long())
                for x in username))
```

```
[username1 = 1953724780, username2 = 1868915551, username0 =
1635017059, uVar1 = 1953724780, uVar2 = 1868915551]
```

```
b'catalyst_ceo'
```



Symbolic Execution with angr

Symbolic Execution

- Symbolic execution is a technique for reachability analysis that tries to explore all possible execution paths of a program
  - Originally introduced by James C. King in a **1976** paper as a static analysis technique for software testing

Key Ideas:

- Each input variable is associated with a symbol — `int i;` $\rightarrow \alpha$
- Each symbol represents a set of input variables — $\alpha \in [0, 2^{32}-1]$
- Program statements generate formulas over symbols — `i * 2 + 5` $\rightarrow 2\alpha + 5$

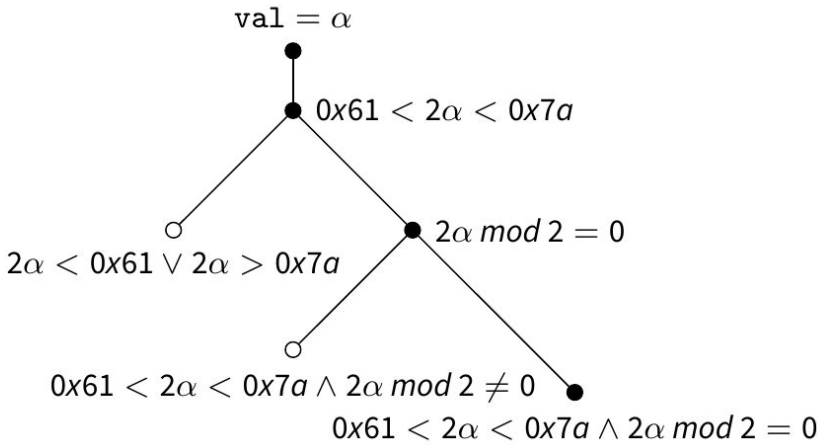
Symbolic Execution

Conditional Branching

Fork the execution and consider each possible branch

- Every path accumulates **path conditions**
  - We can check these formulas *automatically* with Z3!

```
int val = rand();
int z = val * 2;
if (z < 0x7A && z > 0x61) {
  if (z % 2 == 0)
    puts("YAY!");
  else
    puts("NOPE.");
} else
...
```



Symbolic Execution

Limitations

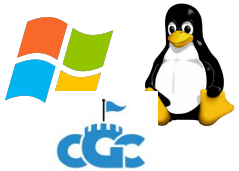
Not everything can be solved symbolically

- Loops and Path Selection
  If there are too many branches or loops the number of program paths may be too large to handle efficiently
  - **Solution:** We can put a bound in the number of loop iterations or change the search strategy. For instance, DFS may not terminate for infinite loops, but BFS can!
- Complex functions
  Some code may be too complex to analyze with a theorem prover
  - **Solution:** We can replace that code with a formula that *summarizes* its behavior without symbolically evaluating it. This technique is known as *symbolic summaries*.

The angr binary analysis platform

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.

https://angr.io

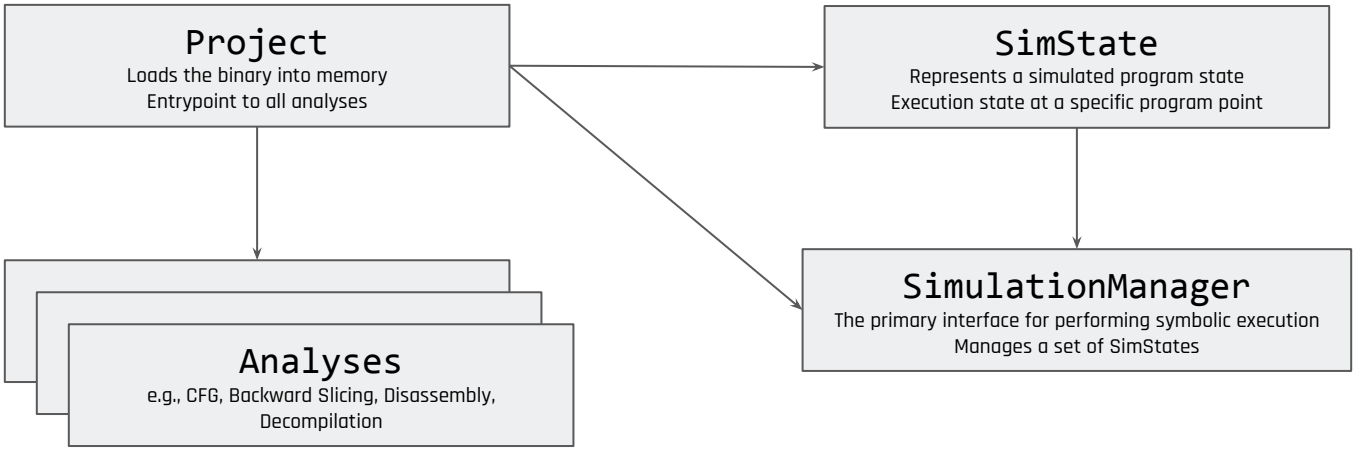
Binary Loader	Intermediate Representation	Data Model Abstraction
	x86 AMD64 ARM ARM64 MIPS MIPS64	$\alpha \in [0, 2^{32}-1]$
CLE	PyVEX Valgrind VEX IR	claripy Z3 Theorem Prover

A symbolic execution engine

A collection of static analysis procedures for binary programs

Includes a disassembler and decompiler

The angr binary analysis platform



The angr binary analysis platform

Project

Loads the binary into memory
Entrypoint to all analyses

`arch`	Program architecture
`entry`	Program Entrypoint
`filename`	Loaded file name
`analyses`	List of analyses
`factory`	Factory methods for this project

Challenge

Let's use `angr` to open the catalyst challenge!

```
import angr

p = angr.Project("./catalyst", auto_load_libs=False)
```

We can, for instance, create a CFG (control flow graph) and decompile the main function:

```
cfg = p.analyses.CFGFast(normalize=True)

print(
 p.analyses.Decompiler(cfg.functions['main'])
 .codegen.text
)
```

The angr binary analysis platform

SimState

Represents a simulated program state
including its memory, registers, and so forth at a specific execution point

`regs`	Register values
`mem`	Memory of the state
`solver`	A reference to the solver for this state (including all constraints)
`step()`	Perform one step of symbolic execution

The solver object allows us to add additional constraints to the state:
`state.solver.add( ... )`

Challenge

We can use the project factory to create SimStates:

```
p.factory.blank_state()
```

creates a mostly-uninitialized state;

```
p.factory.entry_state()
```

creates a stete in the entry point of the program;

```
p.factory.call_state(addr, ...)
```

Is the state at the start of a the function at `addr` with … arguments

The angr binary analysis platform

SimulationManager

SimulationManager

The primary interface for performing symbolic execution
Manages a set of SimStates

`step()`	Step a stash (set) of states forward and categorize the successors appropriately
`explore(find, avoid)`	Perform multiple symbolic execution steps looking for condition **`find`** and avoiding **`avoid`**. Both values can be addresses or functions returning a boolean.

Challenge

We can create a Simulation manager by calling the simgr factory, passing an initial state:

```
st = p.factory.entry_state()

simgr = p.factory.simgr(st)
simgr.step()
```

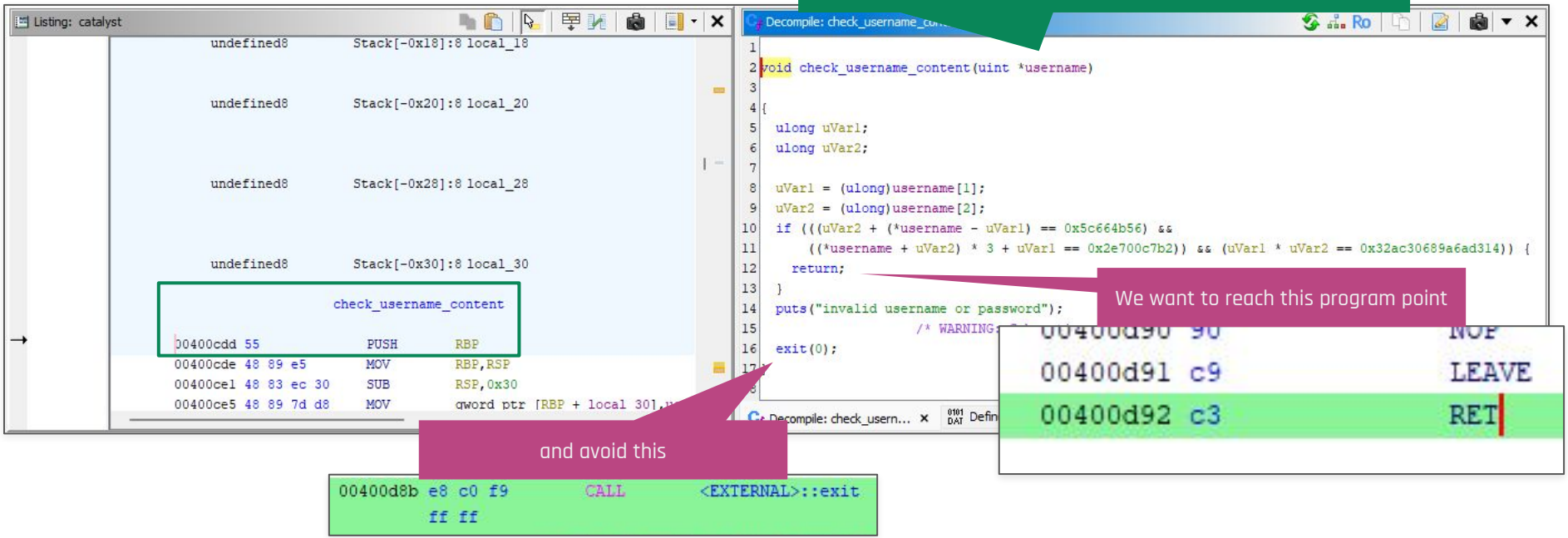
Active states (which will be executed) are stored in the active attribute:

```
simgr.active
```

Catalyst Challenge Revisited

Finding the username

We can setup a call_state at the start of the function passing a symbolic username and use a SimulationManager to explore until the return!



We want to reach this program point

and avoid this

allenge Revisited

Finding the username

We could use the name symbol to add constraints to the state:

```
st.solver.add(
    name.get_byte(0) >= b'a' )
```

The claripy library wraps the z3 theorem prover and it is used to create angr-compatible symbols

We can setup a call_state at the start of the function passing a symbolic username and use a SimulationManager to explore until the return!

```
def username(p):
    name = claripy.BVS("name", 12 *8)
```

create a "name" bit vector (symbol) of 12 bytes

```
    st = p.factory.call_state(
            0x00400cdd,
            angr.PointerWrapper(name, buffer=True))
```

create a call_state passing the name as a pointer

```
    simgr = p.factory.simgr(st)
    simgr.explore(find=0x400d90, avoid=0x400d8b)
```

explore with the simgr

```
    return simgr.found[0].solver.eval(name,
                                      cast_to=bytes)
```

```
name = username(p)
name
>>> b'catalyst_ceo'
```

```
00400d8b e8 c0 f9 ff ff    CALL    <EXTERNAL>::exit
```

```
Decompile: check_username_co...
void check_username_content(uint *username)

ulong uVar1;
ulong uVar2;

uVar1 = (ulong)username[1];
uVar2 = (ulong)username[2];
if (((uVar2 + (*username - uVar1) == 0x5c664b56) &&
    ((*username + uVar2) * 3 + uVar1 == 0x2e700c7b2)) && (uVar1 * uVar2 == 0x32ac30689a6ad314)) {
  return;
}
puts("invalid username or password");
                    /* WARNING:
exit(0);
```

We want to reach this program point

```
00400d90 90        NOP
00400d91 c9        LEAVE
00400d92 c3        RET
```

The angr binary analysis platform

Hooks and SimProcedures

Some functions may be too complex to be symbolically executed efficiently

We can replace such functions with their *symbolic summary*, a function written in Python that interacts directly with the SimState and is executed instead of the real function during symbolic execution

`SimProcedure`

Represents a simulated procedure or function summary

`run()`	Execute the procedure
`state`	The SimState where the procedure is running

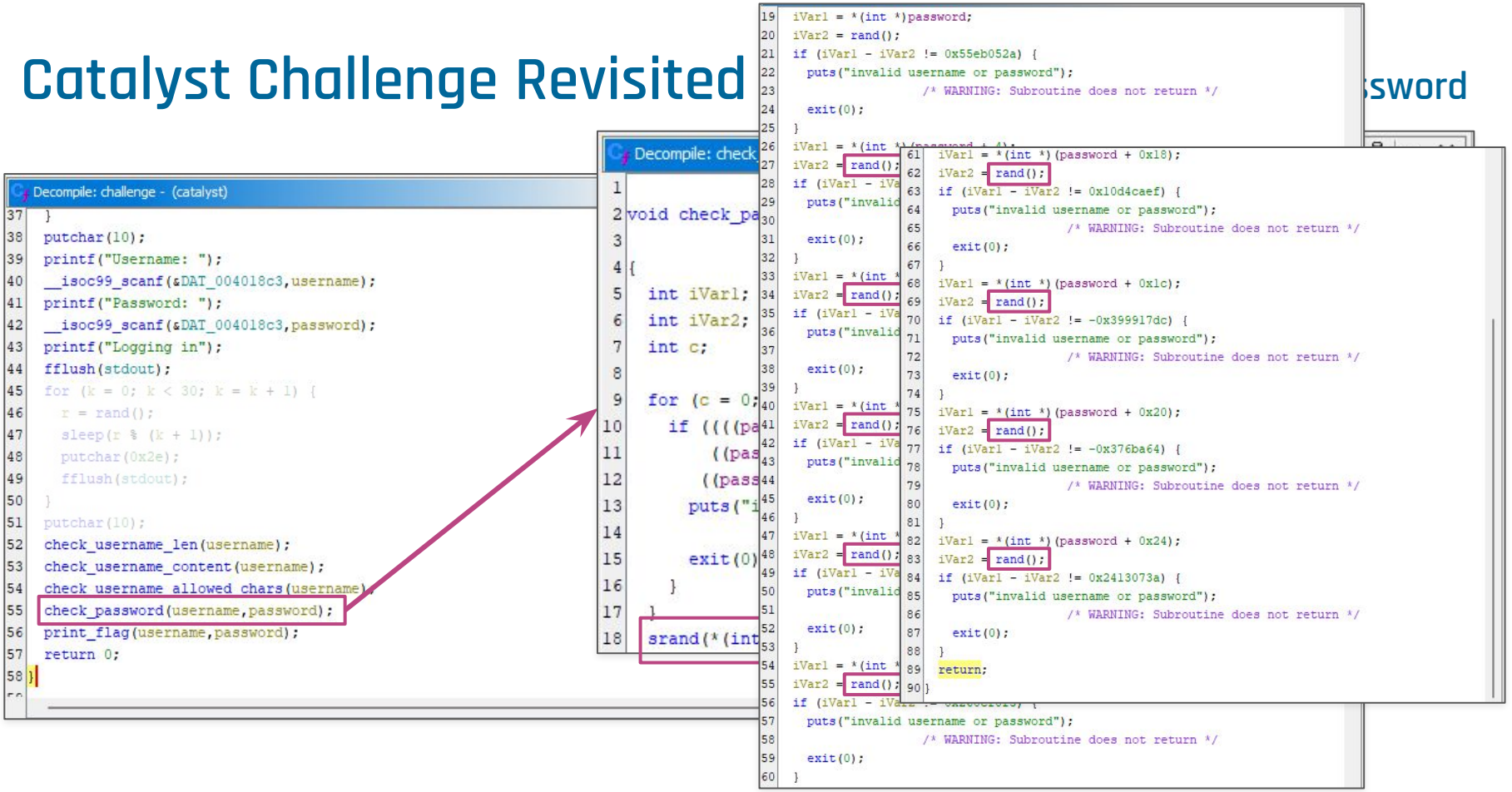
We can replace any arbitrary sequence of bytes in the binary with a SimProcedure:

```
p.hook(ADDR, hook=SIMPROCEDURE, length=N)
```

Or Replace entire functions:

```
p.hook_symbol(FN_NAME, SIMPROCEDURE)
```

Catalyst Challenge Revisited



Catalyst Challenge Revisited

Finding the password

```
class NotVeryRand(angr.SimProcedure):
   def run(self, return_values):
       rand_idx = self.state.globals.get('rand_idx', 0) % len(return_values)
       out = return_values[rand_idx]
       self.state.globals['rand_idx'] = rand_idx + 1
       return out
```

state.globals is copied to the new state after every step

```
import ctypes
libc = ctypes.cdll.LoadLibrary("libc.so.6")
libc.srand(
  sum(struct.unpack("I", name[i:i+4])[0] for i in range(0, len(name), 4))
)
```

We generate the seed by adding together the 3 pieces of the username

```
p.hook_symbol('rand', NotVeryRand(return_values=[libc.rand() for _ in range(10)]))
p.hook_symbol('srand', lambda _:None)
```

Now rand is a procedure that always returns the same values

We don't care about srand so we replace it with an empty function

Catalyst Challenge Revisited

Finding the password

```
def password(p):
      passwd = claripy.BVS("passwd", 40 * 8)
      st = p.factory.call_state(0x00400977,
                                angr.PointerWrapper(name, buffer=True),
                                angr.PointerWrapper(passwd, buffer=True))

      # angr doesn't like this loop, so let's skip it :)
      p.hook(0x00400a46, hook=lambda _:None, length=6)

      simgr = p.factory.simgr(st)
      simgr.explore(find=0x400c40)
      return simgr.found[0].solver.eval(passwd, cast_to=bytes)
```

```
passwd = password(p)
print(passwd)
>>> b'sLSVpQ4vK3cGWyW86AiZhggwLHBjmx9CRspVGggj'
```

The angr binary analysis platform

Files and SimFiles

angr can also reason about file I/O

`SimFile`

Represents an open file

`seek(where)`	Seek to a position in the file
`read_from(len)`	Read some (symbolic) data from the current position in the file

`SimState.posix`

Information about the operating system

`files`	List of open file descriptors
`dumps(fd)`	Returns the concrete content for a file descriptor

Catalyst Challenge Revisited

Printing the Flag

Symbolic execution is "execution" so we can run the get_flag function and read the stdout!

```
def get_flag(p, username, passwd):
  st = p.factory.call_state(0x00400876,
                            angr.PointerWrapper(name, buffer=True),
                            angr.PointerWrapper(passwd, buffer=True))
  simgr = p.factory.simgr(st)
  simgr.explore(find=0x004008ef)
  return simgr.found[0].posix.dumps(1).split(b'\n')[0]
```

```
print( get_flag(p, name, passwd) )

>>> b'your flag is: ALEXCTF{1_t41d_y0u_y0u_ar3__gr34t__reverser__s33}'
```

Thank You!
Q&A

Lorenzo **Veronese** <lorenzo.Veronese@tuwien.ac.at>