

# Symbolic Execution

Emilio Coppa

[ecoppa.github.io](https://ecoppa.github.io)

# Why Symbolic Execution?

Real-world applications of this methodology:

- **(Software Testing)** Bug detection  
e.g., inputs that crash an application
- **(Security)** Exploit and backdoor identification  
e.g., inputs to bypass authentication code
- **(Security)** Malware analysis  
e.g., trigger malicious behaviors in evasive or dormant malware

# What is Symbolic Execution?

Originally introduced by **James C. King** in a **1976** paper as a static analysis technique for *software testing*.

Key ideas:

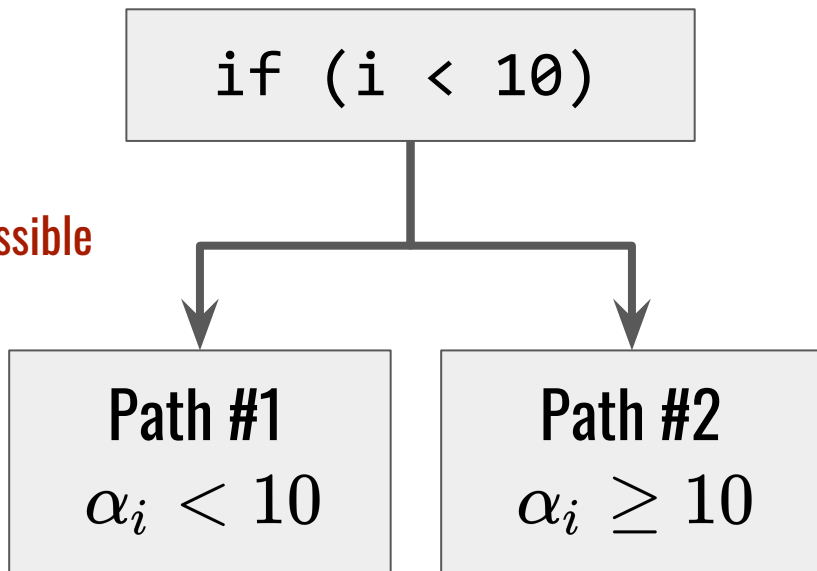
- Each *input* variable is associated to a *symbol*, e.g.,  $\text{int } i \mapsto \alpha_i$
- Each symbol represents a set of input values, e.g.,  $\alpha_i \in [0, 2^{32} - 1]$
- Program statements generate formulas over symbols, e.g.,

$$i * 2 + 5 \mapsto 2 \cdot \alpha_i + 5$$

**What about conditional branches?**

# Symbolic Execution: conditional branch

Fork the execution,  
considering each possible  
yet *realizable* path



From now on, we should consider these *path constraints*

## Example: find input values that make assertion fail

```
1.  void foobar(int a, int b) {  
2.      int x = 1, y = 0;  
3.      if (a != 0) {  
4.          y = 3+x;  
5.          if (b == 0)  
6.              x = 2*(a+b);  
7.      }  
8.      assert(x-y != 0);  
9.  }
```

Symbolic Execution can find **all** the inputs leading to the failure.

**Can you do the same?**

# Symbolic Execution: execution state

Execution state  $(\text{stmt}, \sigma, \pi)$ :

- $\text{stmt}$  is the next statement to evaluate
- $\sigma$  is the *symbolic store* that maps program variables to expressions over concrete values or symbols
- $\pi$  denotes the path constraints, i.e., assumptions made at branches taken in the execution to reach  $\text{stmt}$ . Initially,  $\pi = \text{true}$

In our example:

A	$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b\} \quad \pi = \text{true}$
	2. <code>int x = 1, y = 0</code>

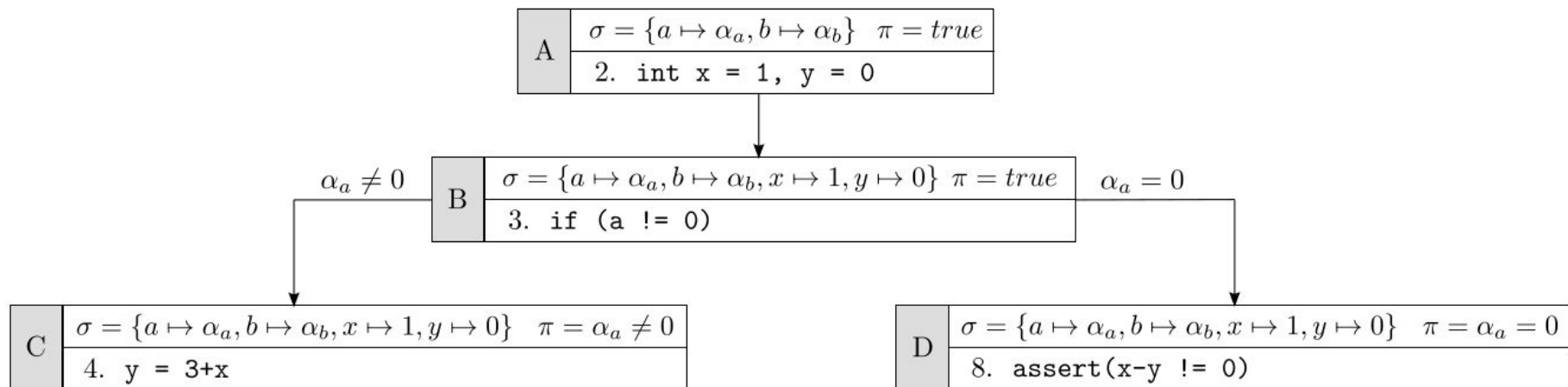
# Example: symbolic exploration (1)

A	$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b\} \quad \pi = true$
	2. <code>int x = 1, y = 0</code>



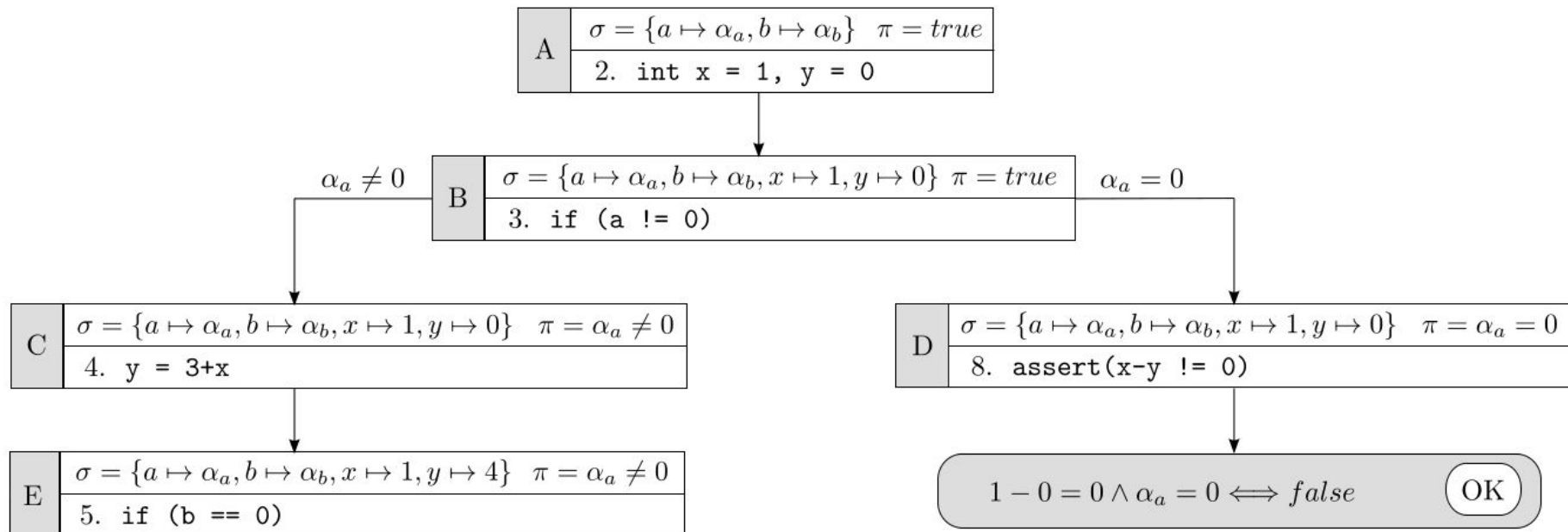
B	$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\} \quad \pi = true$
	3. <code>if (a != 0)</code>

# Example: symbolic exploration (2)

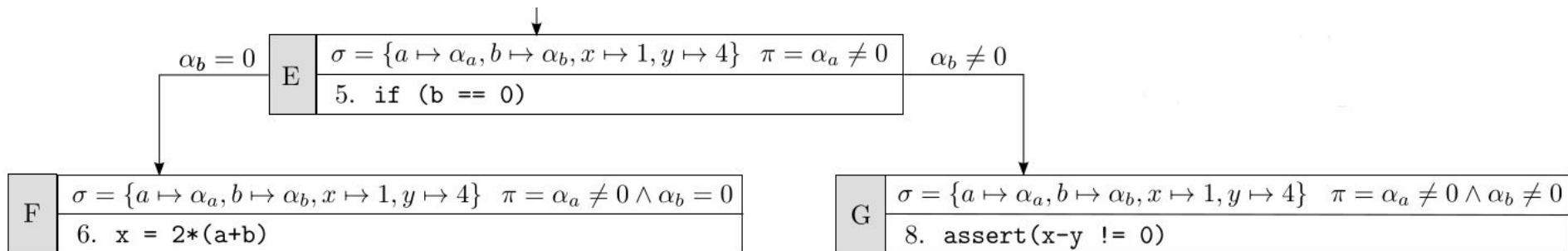




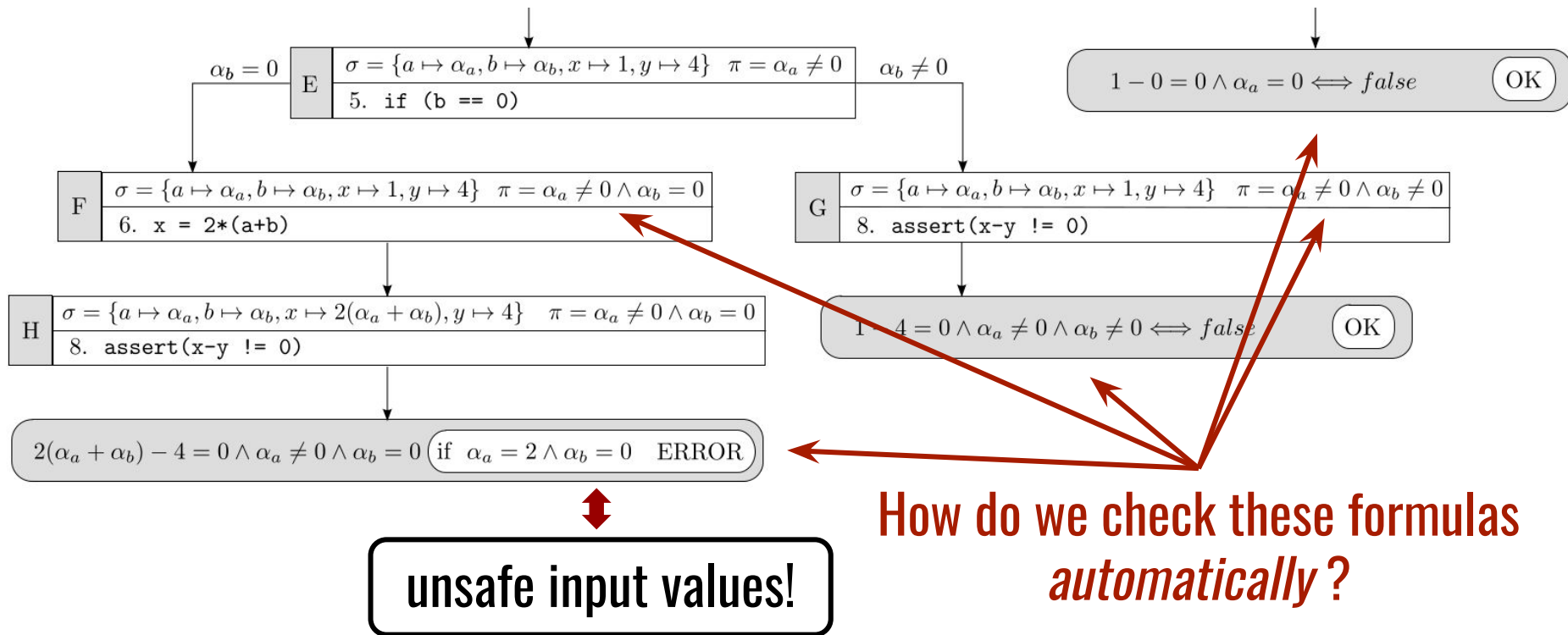
# Example: symbolic exploration (3)



# Example: symbolic exploration (4)

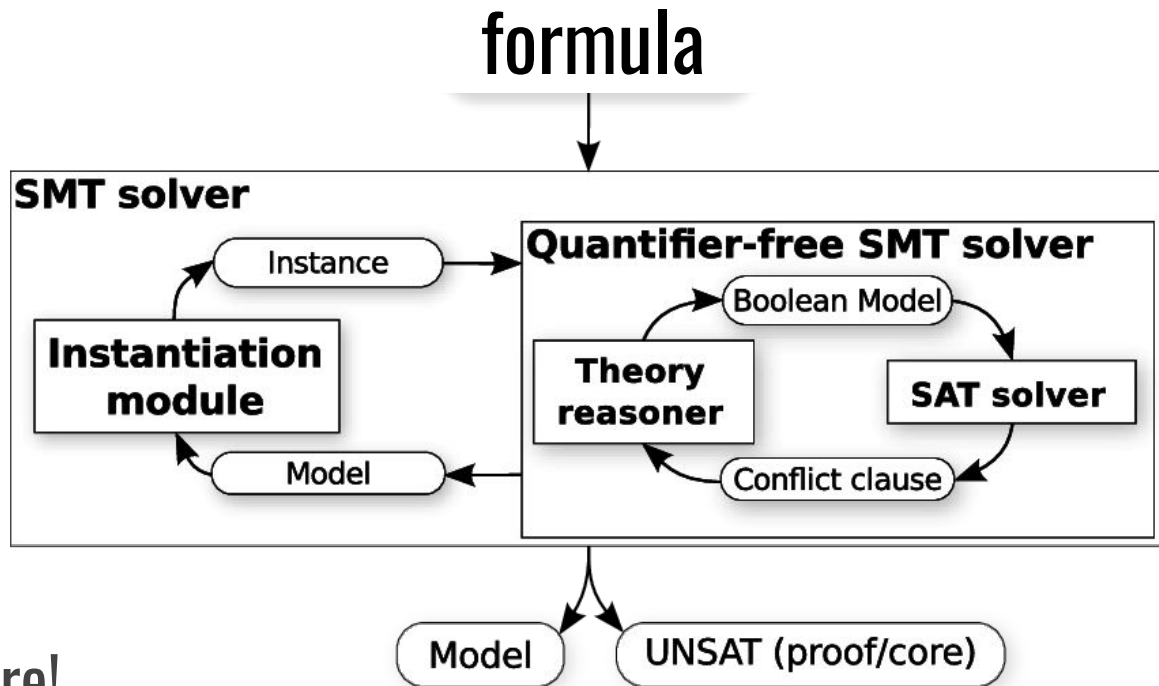


# Example: symbolic exploration (4)



# How to check a symbolic formula?

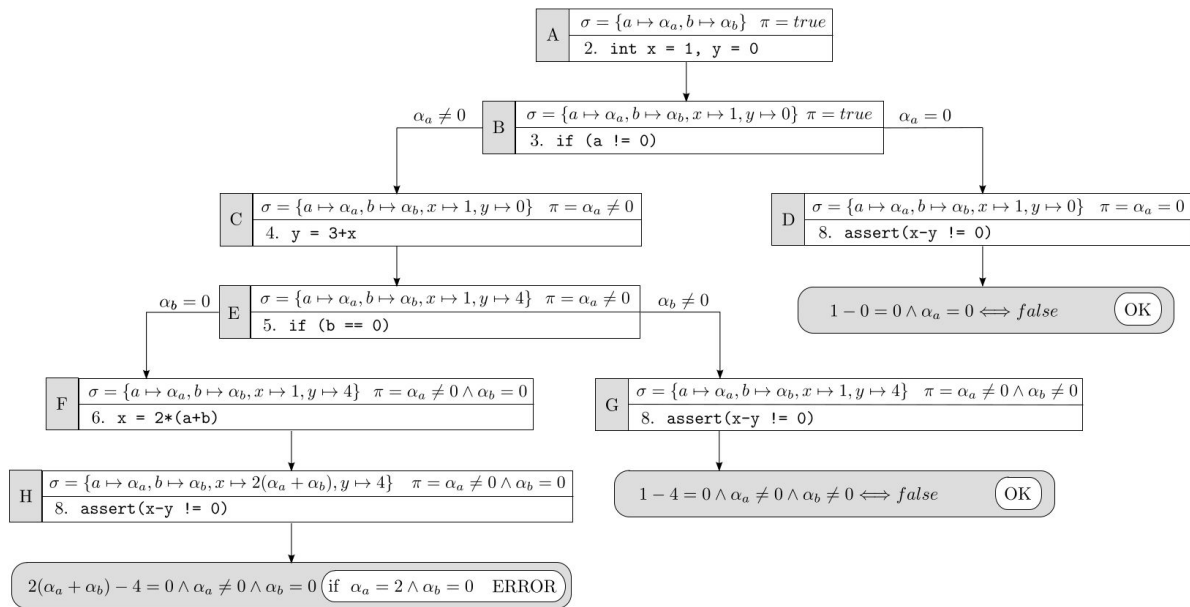
## SMT Solvers



SAT is everywhere!

([image credits](#))

# Example: symbolic exploration (5)



We checked **all** possible paths.  
Can we prove that no other crash may happen?

# Symbolic Execution: soundness and completeness

Can we prove that no other crash may happen?

In theory, **yes**. In practice, **no**.

When analyzing programs with Symbolic Execution:

- Constraints may be hard to solve
- Too many paths may be generated
- Tons of details play a role during an execution

# Symbolic Execution: is it actually useful?

A quote from “AEG: Automatic Exploit Generation” (T. Avgerinos et al.):

[...] KLEE is a state-of-the-art forward symbolic execution engine, but in practice is limited to small programs such as /bin/ls.

*Pure static symbolic execution hardly scales in practice.*

# Symbolic Execution: a success story [SAGE NDSS08]



SAGE: Whitebox Fuzzing  
for Security Testing

**SAGE has had a remarkable impact at Microsoft.**

Patrice Godefroid, Michael Y. Levin, David Molnar, Microsoft

Most *ACM Queue* readers might think of “program verification research” as mostly theoretical with little impact on the world at large. Think again. If you are reading these lines on a PC running some form of Windows (like 93-plus percent of PC users—that is, more than a billion people), then you have been affected by this line of work—without knowing it, which is precisely the way we want it to be.



# Symbolic Execution: a success story (2)

## IMPACT OF SAGE

Since 2007, SAGE has discovered many security-related bugs in many large Microsoft applications, including image processors, media players, file decoders, and document parsers. Notably, SAGE found roughly *one-third* of all the bugs discovered by file fuzzing during the development of Microsoft's Windows 7. Because SAGE is typically run last, those bugs were missed by everything else, including static program analysis and blackbox fuzzing.

Finding all these bugs has saved millions of dollars to Microsoft, as well as to the world in time and energy, by avoiding expensive security patches to more than 1 billion PCs. The software running on *your* PC has been affected by SAGE.

Since 2008, SAGE has been running 24/7 on an average of 100-plus machines/cores automatically fuzzing hundreds of applications in Microsoft security testing labs. This is more than 300 machine-years and the *largest computational usage ever for any SMT (Satisfiability Modulo Theories) solver*, with more than 1 billion constraints processed to date.

SAGE is so effective at finding bugs that, for the first time, we faced “bug triage” issues with dynamic test generation. We believe this effectiveness comes from being able to fuzz large applications (not just small units as previously done with dynamic test generation), which in turn allows us to find bugs resulting from problems across multiple components. SAGE is also easy to deploy, thanks to x86 binary analysis, and it is fully automatic. SAGE is now used daily in various groups at Microsoft.

# Symbolic execution: challenges

Some of the challenges that symbolic execution has to face:

1. memory model
2. path explosion
3. environment interaction
4. constraint solver

Depending on the goals, different choices and trade-offs must be made in order to effectively use symbolic execution.

# Symbolic Execution: memory model

How do we handle symbolic loads or symbolic writes?

```
int array [N] = { 0 };  
array [i] = 10;           // i symbolic  
assert(array[j] != 0);    // j symbolic
```

Approaches:

1. **fully symbolic:**  
consider any possible outcome
2. **fully concrete:**  
consider one possible outcome
3. **partial:** concretize writes,  
concretize loads when hard



N states  
accurate but not scale



1 state  
scale but not accurate



K states  
scale but (in)accurate

# Symbolic Execution: path explosion

When possible, pre-constrain your symbolic inputs:

- none
- known size
- known prefix
- concrete values

**Exploit knowledge from the application domain to limit path explosion!**

For instance: some fields of a packet in a protocol are constant,  
other fields can assume a restricted set of values, etc.

# Symbolic Execution: path explosion (2)

State scheduling to prioritize interesting paths. Common strategies:

- **depth first search**: minimize resource consumption
- **random path selection**: minimize starvation
- **coverage optimized search**: maximize coverage
- **buggy path first**: aims at vulnerability detection

The search heuristic depends on the goal that you are trying to achieve!

# Symbolic Execution: environment interaction

How to deal with library/system calls?

1. **fully symbolic**: symbolically execute OS
2. **fully concrete**: concretize & execute OS
3. **approximation**: API models describe effects on the state



accurate but not scale



scale but not accurate



scale but manpower  
costly and error-prone

Environment is often a source of input.  
Approximation is often used in practice.

# Symbolic Execution: concolic execution

**Concolic** execution: **con**crete + **sym**bol**ic**. A mixed static/dynamic approach.

Key idea:

1. Run code concretely and record instruction trace
2. Symbolically analyze trace
3. Choose one branch, negate it, and generate new input values
4. Restart from (1) using the new inputs

# Symbolic Execution: concolic execution (2)

Some of the benefits:

1. Symbolic execution is driven by a concrete execution
  - ➡ easier to detect false positives (divergences)
  - ➡ one solver query for each path
2. Exploit concrete values to solve hard constraints:

```
if (a == (b * b % 50))
```

Hard to solve when **a** and **b** are fully symbolic  
Easy to solve when **b** is concrete and **a** is symbolic



# Do you want to know more about Symbolic Execution?

Check out our survey!

Roberto Baldoni, Emilio Coppa, Daniele Cono  
D'Elia, Camil Demetrescu, Irene Finocchi. **A  
Survey of Symbolic Execution Techniques.** ACM  
Computing Surveys (ACM CSUR), 51(3), 2018.

DOI: [10.1145/3182657](https://doi.org/10.1145/3182657)

PDF: [\[URL\]](#)

## A Survey of Symbolic Execution Techniques

ROBERTO BALDONI, EMILIO COPPA, DANIELE CONO D'ELIA, CAMIL DEMETRESCU,  
and IRENE FINOCCHI, Sapienza University of Rome

Many security and software testing applications require checking whether certain properties of a program hold for any possible usage scenario. For instance, a tool for identifying software vulnerabilities may need to rule out the existence of any backdoor to bypass a program's authentication. One approach would be to test the program using different, possibly random inputs. As the backdoor may only be hit for very specific program workloads, automated exploration of the space of possible inputs is of the essence. Symbolic execution provides an elegant solution to the problem, by systematically exploring many possible execution paths at the same time without necessarily requiring concrete inputs. Rather than taking on fully specified input values, the technique abstractly represents them as symbols, resorting to constraint solvers to construct actual instances that would cause property violations. Symbolic execution has been incubated in dozens of tools developed over the last four decades, leading to major practical breakthroughs in a number of prominent software reliability applications. The goal of this survey is to provide an overview of the main ideas, challenges, and solutions developed in the area, distilling them for a broad audience.

CCS Concepts: • **Software and its engineering** → **Software verification**; *Software testing and debugging*; • **Security and privacy** → Software and application security;

Additional Key Words and Phrases: Symbolic execution, static analysis, concolic execution, software testing

### ACM Reference Format:

Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 0 (2018), 37 pages. <https://doi.org/10.1145/3182657>. 0000001

*"Sometimes you can't see how important something is in its moment, even if it seems kind of important. This is probably one of those times."*

(Cyber Grand Challenge highlights from DEF CON 24, August 6, 2016)

## 1 INTRODUCTION

Symbolic execution is a popular program analysis technique introduced in the mid '70s to test whether certain properties can be violated by a piece of software [16, 58, 67, 68]. Aspects of interest could be that no division by zero is ever performed, no NULL pointer is ever dereferenced, no backdoor exists that can bypass authentication, etc. While in general there is no automated way to decide some properties (e.g., the target of an indirect jump), heuristics and approximate analyses can prove useful in practice in a variety of settings, including mission-critical and security applications.

Author's addresses: R. Baldoni, E. Coppa, D.C. D'Elia, and C. Demetrescu, Department of Computer, Control, and Management Engineering, Sapienza University of Rome; I. Finocchi, Department of Computer Science, Sapienza University of Rome. E-mail addresses: {baldoni, coppa, delia, demetrescu}@diag.uniroma1.it, finocchi@di.uniroma1.it.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2018 Association for Computing Machinery.

0960-0900/2018-03-ART0 0000001  
<https://doi.org/10.1145/3182657>

ACM Computing Surveys, Vol. 51, No. 3, Article 0. Publication date: 2018.

# Hands-on Session

# **angr**: a python framework for analyzing binaries



<http://angr.io>

Support for:

- CFG reconstruction
- (Static) Symbolic Execution
- Static Binary Analysis
- ...a lot more...

Extremely easy to prototype new ideas  
(e.g., novel program analyses) with angr!



## angr @ DARPA CGC

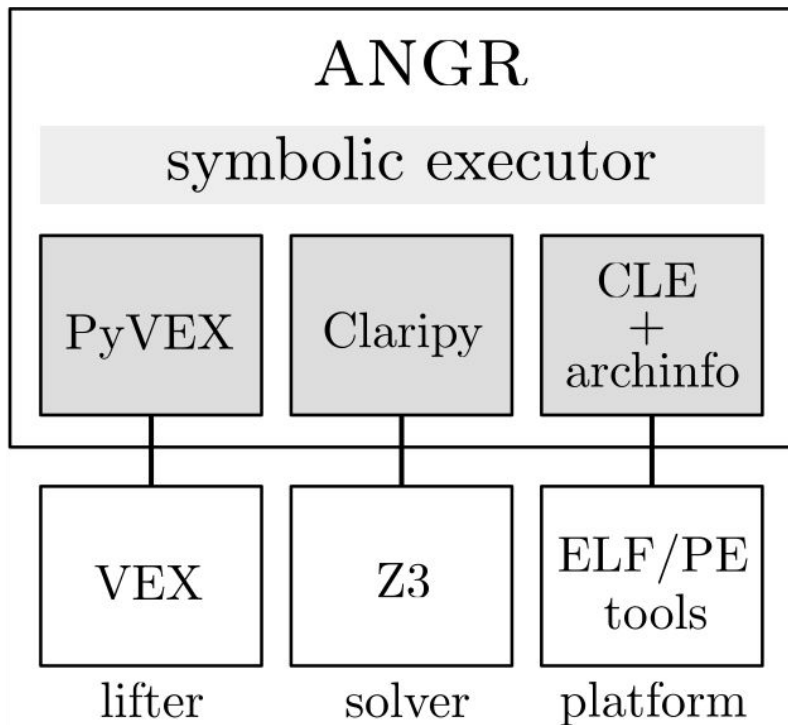
**angr** was the core of the MechaPhish CRS that played at the DARPA Cyber Grand Challenge.

This was a competition where **fully autonomous** systems have to detect, patch, and exploit software.

More details: [phrack.org/papers/cyber\\_grand\\_shellphish.html](http://phrack.org/papers/cyber_grand_shellphish.html)

**MechaPhish** and **angr** are developed by the Computer Security Lab @ UC Santa Barbara. This is the research group of Giovanni Vigna!

# angr: architecture



## Components:

- **PyVEX**: lifter based on VEX (valgrind) for reasoning on binary code. Makes angr architecture-independent!
- **Claripy**: solver abstraction. Support for Microsoft Z3 solver.
- **CLE/archinfo**: ELF/PE loaders. It handles platform details.

**angr:** let's start using it!

Installation on CC19-Lubuntu18.04:

```
> python3 -m pip install --user angr
```

# Setup on VM BIAR 1.6.5

Start the VM, open a terminal:

```
> cd ~/Desktop/  
> git clone https://tinyurl.com/SymbExec  
> cd SymbExec
```

Install angr and IDA:

```
> ./setup/install-angr.sh  
> ./setup/install-IDA.sh
```

**angr:** let's start using it!

Installation on LXLE-BIAR-1.6.5 VM:

```
> python2 -m pip install --upgrade --user pip  
> python2 -m pip install --user angr
```

angr has been recently ported to Python 3. It does not work correctly on the LXLE-BIAR VM.



# angr: how to use it?

To symbolic execute a binary, you need to:

1. choose exploration targets:
  - a. [start target] where (code address) start the execution
  - b. [find target] where (code address) stop the execution
  - c. [avoid targets] which paths (code addresses) ignore
2. Define the symbolic inputs

angr does not know what is symbolic and what is concrete. User has to mark it explicitly.

Unlikely that angr will reach the **find target** if the **start target** is the binary entry point!  
**Avoid targets** are needed to limit path explosion!

# Exercise #1: analyze our initial example [assertion fail]

```
> cd slide-example  
> ../setup/run-IDA.sh
```

Analyze with IDA the binary **slide-example/example**

If you want to use IDA on Windows, download the binary at:

<https://github.com/ercoppa/symbolic-execution-tutorial/raw/master/slide-example/example>

# Exercise #1: target addresses and symbolic inputs

**Start target:** 0x400576

first instruction of foobar

Target addresses may  
change if you re-compile  
the example.c source file!

**Find target:** 0x4005BC  
assert block

**Inputs:**  
function args

**Avoid target:**  
0x4005D5  
non assert block

```
0000000000400576
0000000000400576
0000000000400576 ; Attributes: bp-based frame
0000000000400576
0000000000400576 public foobar
0000000000400576 foobar proc near
0000000000400576
0000000000400576 var_18= dword ptr -18h
0000000000400576 var_14= dword ptr -14h
0000000000400576 var_8= dword ptr -8
0000000000400576 var_4= dword ptr -4
0000000000400576
0000000000400576 push rbp
0000000000400577 mov rbp, rsp
000000000040057A sub rsp, 20h
000000000040057E mov [rbp+var_14], edi
0000000000400581 mov [rbp+var_18], esi
0000000000400584 mov [rbp+var_8], 1
0000000000400588 mov [rbp+var_4], 0
0000000000400592 cmp [rbp+var_14], 0
0000000000400596 jz short loc_4005B4
```

```
0000000000400598 mov eax, [rbp+var_8]
000000000040059B add eax, 3
000000000040059E mov [rbp+var_4], eax
00000000004005A1 cmp [rbp+var_18], 0
00000000004005A5 jnz short loc_4005B4
```

```
00000000004005A7 mov edx, [rbp+var_14]
00000000004005AA mov eax, [rbp+var_18]
00000000004005AD add eax, edx
00000000004005AF add eax, eax
00000000004005B1 mov [rbp+var_8], eax
```

```
00000000004005B4
00000000004005B4 loc_4005B4:
00000000004005B4 mov eax, [rbp+var_8]
00000000004005B7 cmp eax, [rbp+var_4]
00000000004005BA jnz short loc_4005D5
```

```
00000000004005BC mov ecx, offset __PRETTY_FUNCTION___2822 ; "foobar"
00000000004005C1 mov edx, 0Ch ; line
00000000004005C6 mov esi, offset file ; "main.c"
00000000004005CB mov edi, offset assertion ; "x-y != 0"
00000000004005D0 call __assert_fail
```

```
00000000004005D5
00000000004005D5 loc_4005D5:
00000000004005D5 nop
00000000004005D6 leave
00000000004005D7 retn
00000000004005D7 foobar endp
00000000004005D7
```

# Exercise #1: angr script solve-example.py (1)

```
import angr
```

```
proj = angr.Project('example')
```

```
start = XXX      # foobar
```

```
avoid = [YYY]    # non assert block
```

```
end = ZZZ        # assert block
```

```
# blank_state since arbitrary point
```

```
state = proj.factory.blank_state(addr=start)
```

```
# arguments are inside registers in x86_64
```

```
a = state.regs.edi
```

```
b = state.regs.esi
```

# Exercise #1: angr script solve-example.py (2)

```
sm = proj.factory.simulation_manager(state)
while len(sm.active) > 0:
    print(sm) # info
    sm.explore(avoid=avoid, find=end, n=1)
    if len(sm.found) > 0: # Bazinga!
        print("\nReached the target\n")
        state = sm.found[0]
        print("%edi = " + str(state.se.eval_ upto(a, 10)))
        print("%esi = " + str(state.se.eval_ upto(b, 10)))

    break
```

# Exercise #1: run it!

```
> python3 solve-example.py
```

```
<SimulationManager with 1 active>
```

```
<SimulationManager with 2 active>
```

```
<SimulationManager with 3 active>
```

```
<SimulationManager with 3 active, 1 avoid>
```

```
Reached the target
```

```
%edi = [2L, 2147483650L]
```

```
%esi = [0]
```

**This solution was unexpected!  
Is it valid? Yes. But why?**



```
> ./example 2 0
```

```
example: example.c:12: foobar: Assertion `x-y != 0' failed.  
Aborted
```

```
> ./example 2147483650 0
```

```
example: example.c:12: foobar: Assertion `x-y != 0'  
failed.Aborted
```

# angr: useful API

## Expressions:

- **symbolic value:** `state.solver.BVS(label, size_in_bits)`
- **concrete value:** `state.solver.BVV(value, size_in_bits)`
- **operators:** `state.solver.{Concat, Extract, Or, And, Not}`  
`{+, -, *, /, ==, <, <=, >, >=, !=}`  
`state.solver.Reverse(expr) // swaps bytes to`  
`// fix endianness`

# angr: useful API (2)

## Exploration:

- create project: `p = angr.Project('binary')`
- blank state: `state = p.factory.blank_state(addr=start)`
- entry state: `state = p.factory.entry_state()`
- simulation engine:  
  
`sm = proj.factory.simulation_manager(state) # create it`  
`sm.explore(avoid=avoid, find=end, n=1) # explore 1 BB`  
`sm.explore(avoid=avoid, find=end)`  
`sm.active # list of active states`  
`sm.found # list of found states`  
`sm.avoid # list of avoided states`



# angr: useful API (3)

## Registers:

- get expr from a register: `expr = state.regs.eax`
- put expr into a register: `state.regs.eax = expr`

## Memory: **keep in mind the endianness!**

- read bytes from memory: `expr = state.memory.load(0xABC, 4)`
- store expr into a register: `state.memory.store(0xABC, expr, 4)`

Stack pop expr: `expr = state.stack_pop()`

Stack push expr: `state.stack_push(expr)`

# angr: useful API (4)

## Constraint solver:

- add a constraint: `state.add_constraints(expr)`
- get a numeric solution: `val = state.solver.eval(expr)`
- get K numeric solutions: `v = state.solver.eval_upto(expr, K)`
- get a solution as string:  
`S = state.solver.eval(arg, cast_to=bytes)`  
`S = S.split(b'\x00')[0].decode('ascii')`

# angr: useful API (5)

## Hook:

```
def custom_hook(state):  
    state.regs.eax = state.solver.BVS('input', 32)  
    state.add_constraints(  
        state.solver.And(state.regs.eax > 0, state.regs.eax < 10))  
  
proj.hook(0x4011BE, hook=custom_hook, length=5)
```

**custom\_hook** will be executed when instruction pointer is at 0x4011BE  
Since **length=5**, angr will skip 5 bytes from the code when executing the hook,  
recovering execution from 0x4011BE+0x5.

Hooks helps bypassing complex functions/instrs or OS interactions.

To write robust API models, angr offers **SimProcedures** (see documentation)

## Exercise #2: logic bomb

It's the same logic bomb that you have already solved in a prev training lecture. You can run it on Windows or Linux [after installing wine (not possible on VM BLAR): `sudo apt install wine`].

Binary: bomb/bomb.exe

Logic behind each phase:



# Exercise #2: logic bomb

A logic bomb from CMU.

Binary: linux-bomb/bomb

Logic behind each phase:



## Exercise #2: phase 1

```
> cd linux-bomb
```

```
> ./bomb
```

```
Welcome to my fiendish little bomb. You have 6 phases with  
which to blow yourself up. Have a nice day!
```

```
TEST_TEST_TEST
```

typed the wrong  
solution



```
BOOM!!!
```

```
The bomb has blown up.
```

## Exercise #2: phase 1

```
> cd ../bomb
```

```
> wine ./bomb
```

```
Welcome to my fiendish little bomb. You have 6 phases with  
which to blow yourself up. Have a nice day!
```

```
TEST_TEST_TEST
```

typed the wrong  
solution



```
BOOM!!!
```

```
The bomb has blown up.
```

# Exercise #2: target addresses and input

**start target:** begin of phase\_1()

**Input:** first argument, pointer to a string

**avoid target:** block calling bomb\_explode()

**find target:** block returning to main function

```
0000000000400EE0
0000000000400EE0
0000000000400EE0
0000000000400EE0 public phase_1
0000000000400EE0 phase_1 proc near
0000000000400EE0 sub     rsp, 8
0000000000400EE4 mov     esi, offset aBorderRelation ; "Border relations with Canada have never"...
0000000000400EE9 call    strings_not_equal
0000000000400EEE test    eax, eax
0000000000400EF0 jz      short loc_400EF7
```

```
0000000000400EF2 call    explode_bomb
```

```
0000000000400EF7
0000000000400EF7 loc_400EF7:
0000000000400EF7 add     rsp, 8
0000000000400EFB retn
0000000000400EFB phase_1 endp
0000000000400EFB
```



# Exercise #2: target addresses and input

**start target:** begin of phase\_1()

**Input:** first argument, pointer to a string

**avoid target:** block calling bomb\_explode()

**find target:** block returning to main function

```
; Attributes: bp-based frame

sub_401190 proc near

arg_0= dword ptr 8

push    ebp
mov     ebp, esp
push    offset aPublicSpeaking ; "Public speaking is very easy."
mov     eax, [ebp+8]
push    eax
call    sub_401740
add     esp, 8
test    eax, eax
jz      short loc_4011AD
```

```
call    sub_401960
```

```
loc_4011AD:
pop     ebp
retn
sub_401190 endp
```

## Exercise #2: discussion script of phase #1 (solution)

Exploration targets:

```
start = 0x400ee0      # addr of phase_1()
avoid = [0x40143a]    # addr of explode_bomb()
end = [0x400ef7]      # last block of
                      phase_1 (before ret)
```

## Exercise #2: discussion script of phase #1 (solution)

Exploration targets:

```
start = 0x401190      # addr of phase_1()
avoid = [0x401960]    # addr of explode_bomb()
end = [0x4011AD]      # last block of
                      phase_1 (before ret)
```

## Exercise #2: new angr bits from phase-1.py

```
# a symbolic input string with a length up to 128 bytes
```

```
arg = state.solver.BVS("input_string", 8 * 128)
```

```
# an ending byte
```

```
arg_end = state.solver.BVV(0x0, 8)
```

Alternative approach:

```
arg_end = state.se.BVS("end_input_string", 8)
```

```
state.add_constraints(arg_end == 0x0)
```

```
# concat arg and arg_end
```

```
arg = state.solver.Concat(arg, arg_end)
```

```
bind_addr = 0x603780 # read_line() is storing here the input string
```

```
# store the symbolic string at this address
```

```
state.memory.store(bind_addr, arg)
```

```
# set arg
```

```
state.regs.rdi = bind_addr
```

## Exercise #2: new angr bits from phase-1.py

```
# a symbolic input string with a length up to 128 bytes
```

```
arg = state.solver.BVS("input_string", 8 * 128)
```

```
# an ending byte
```

```
arg_end = state.solver.BVV(0x0, 8)
```

Alternative approach:

```
arg_end = state.se.BVS("end_input_string", 8)
```

```
state.add_constraints(arg_end == 0x0)
```

```
# concat arg and arg_end
```

```
arg = state.solver.Concat(arg, arg_end)
```

```
bind_addr = 0x603780 # read_line() is storing here the input string
```

```
# store the symbolic string at this address
```

```
state.memory.store(bind_addr, arg)
```

```
# push string address into the stack
```

```
state.stack_push(state.se.BVV(bind_addr, 32))
```

```
state.stack_push(state.se.BVV(0x0, 32))
```

## Exercise #2: phase #1 (execution)

```
> python3 phase-1.py  
<SimulationManager with 1 active>  
[...] // after 2 minutes
```

Reached the target

```
<SimulationManager with 1 found, 35 active, 101 avoid>
```

Solution: **Public speaking is very easy.**

```
> ./bomb.exe # you can run it on Linux using wine  
Welcome to my fiendish little bomb. You have 6 phases with  
which to blow yourself up. Have a nice day!
```

**Border relations with Canada have never been better.**

**Phase 1 defused.** How about the next one? [...]

## Exercise #2: now move on to other phases!

### Hints:

1. use phase-1.py as a template to edit for other phases
2. check previous slides on angr API
3. detect start/find/avoid target addresses
4. try to avoid OS interactions
5. understand the execution context of each phase: where the input is expected to be found, what assumptions can be made on it to help SE

**Good luck!**

**Ask me questions!**