Symbolic Execution

Prof. Mathy Vanhoef
DistriNet – KU Leuven – Belgium

The downside of (semi-)random fuzzing

Can the following bug be found by random testing?

- Generate random or mutated inputs (like in AFL)
- Execute the program on those concrete inputs

```
void test_me(int x) {
    if (x == 514983144) {
        // crash the pogram
        assert(0);
    }
}
```

Problem: probability of reaching this bug is extremely small

- Probability of $2^{-32} \approx 0.00000023\%$ in every random test

Alternative: symbolic execution

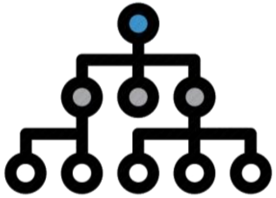
Use **symbolic values** for input

- Run program with symbolic input(s)
- At each conditional instruction, follow both branches

```
void test_me(int x) {
  if (x == 514983144) {
    assert(0);
  }
}
```

Idea is to analyze every possible execution path

- Holy grail of achieving complete path coverage…
- Number of paths grows exponentially: $\approx 2^{|branches|}$



Symbolic Execution

First: what do we mean with symbolic input?

A symbolic input (i.e., a symbol) is like a variable **x** in math

- At the start of the program, symbolic input is unconstrained
- The **execution path** that will be followed will add constraints
- For instance, when an if-test depends on a symbolic input

$$x < 0$$
$$x^2 + 2x + 4 = 4$$

- Can now find possible solutions for the symbolic variable x

Symbolic execution: intuitive example

Mark data as symbolic

```
void recv(data, len) {
  if (data[0] != 1)
    return
  if (data[1] != len)
    return

  int num = len/data[2]
  ...
}
```

Symbolic branch

Symbolic execution: intuitive example

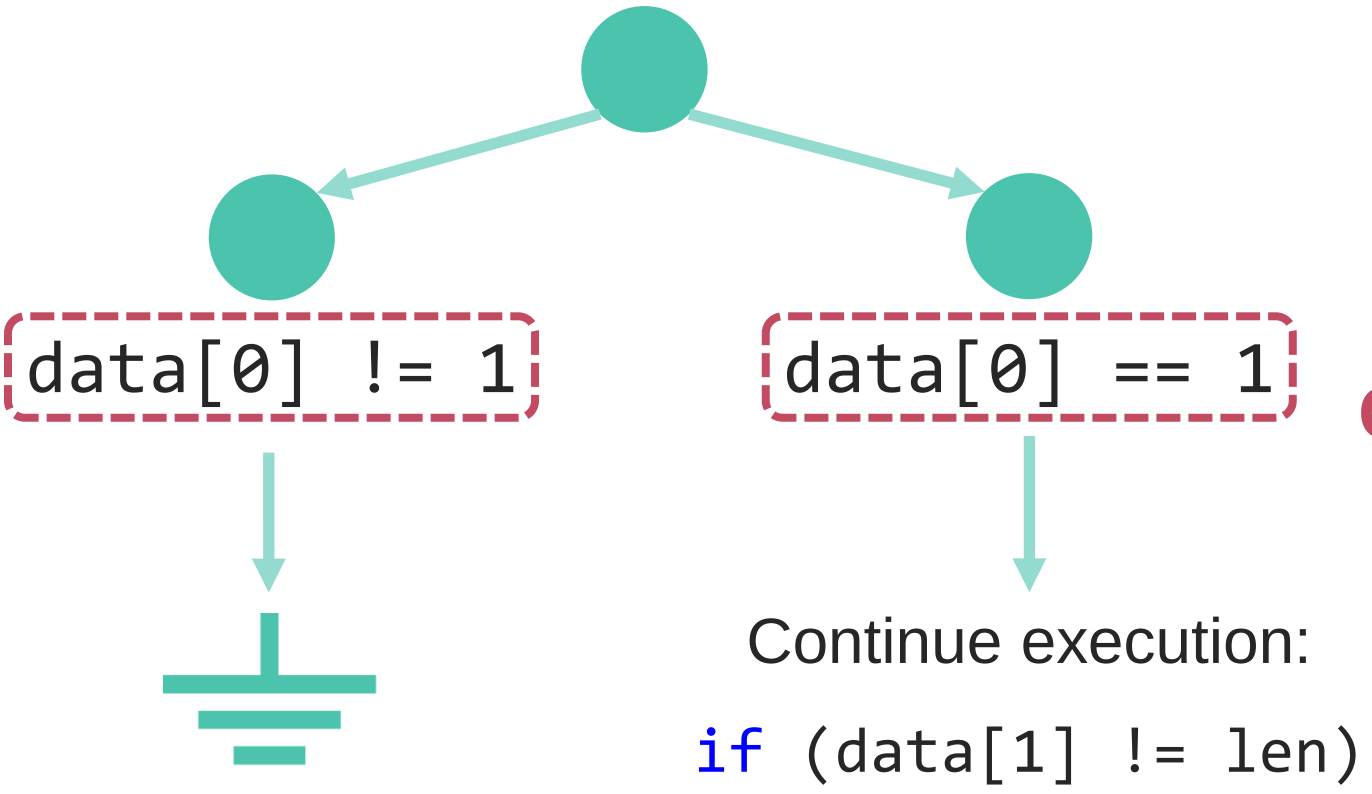
data[0] != 1

```
void recv(data, len) {
  if (data[0] != 1)
    return
  if (data[1] != len)
    return

  int num = len/data[2]
  ...
}
```

data[0] == 1

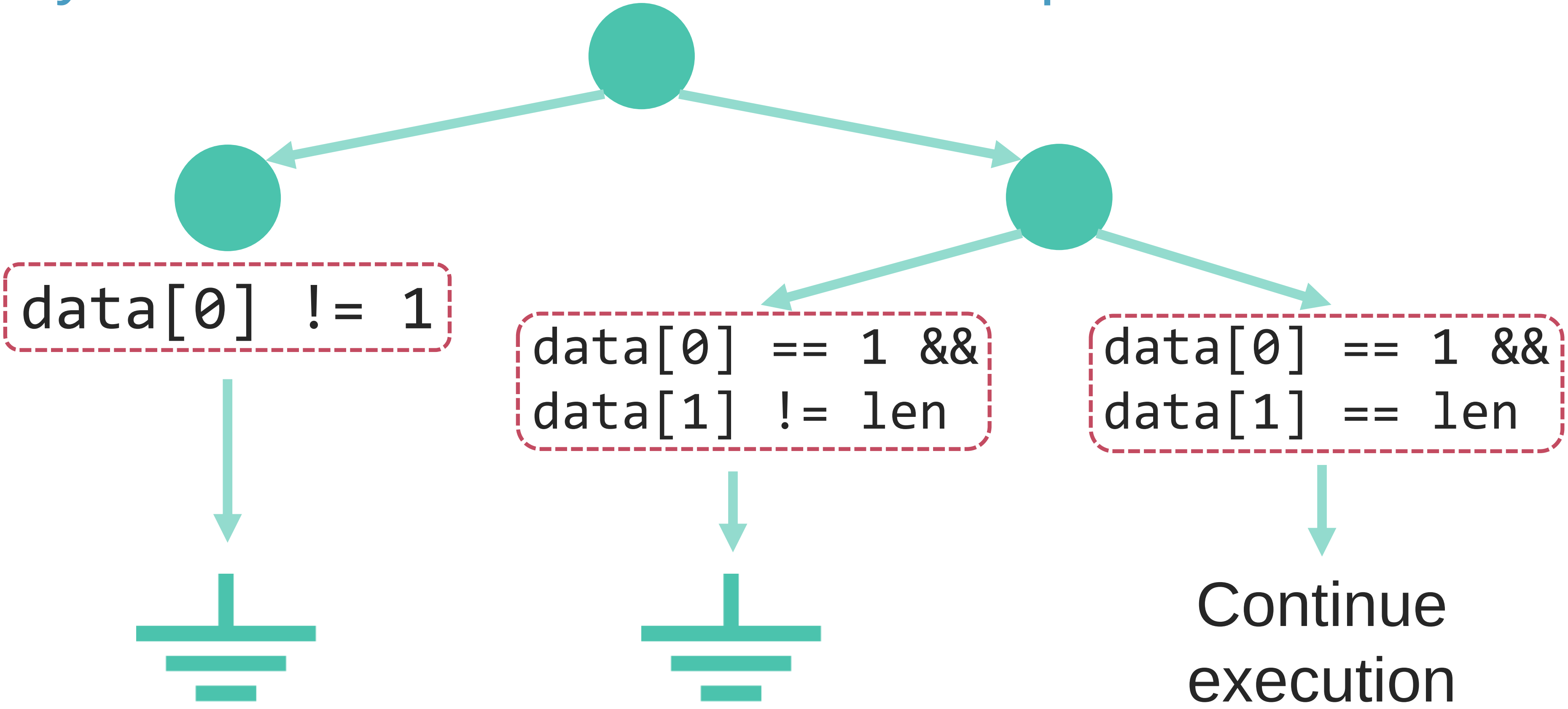
```
void recv(data, len) {
  if (data[0] != 1)
    return
  if (data[1] != len)
    return

  int num = len/data[2]
  ...
}
```

Symbolic execution: intuitive example



PC = Path Constraint

Symbolic execution: intuitive example



Symbolic execution: intuitive example

```
data[0] == 1 &&
data[1] == len
-------------------------------
void recv(data, len) {
  if (data[0] != 1)
    return
  if (data[1] != len)
    return

  int num = len/data[2]
  ...
```

Can data[2] equal zero under the current PC?

Use an SMT solver
(more on this later)

Yes! Bug detected!

How to symbolic execute a program?

Goal is to explore all execution paths in the program

- Collect constraints & generate a matching concrete input.

Symbolically run a program by maintaining a state $(stmt, \sigma, \pi)$

- $stmt$: the **instructions** to be executed
- σ: a **symbolic store** that maps variables to expressions over either concrete values or symbolic input values α_i
- π: a **path constraint (PC)** that records which constraints on the symbolic input variables α_i lead to the current $stmt$

How to symbolic execute a program?

Three main types of instructions. They are handled as follows:

- An assignment "**x = e**": update the symbolic store σ by associating x with a new symbolic expression e_s
- A branch "**if e then s_t else s_f**": execution is forked by creating two new execution states:
  - True branch: $(s_t, \sigma, \pi \wedge e)$
  - False branch: $(s_f, \sigma, \pi \wedge \neg e)$

  The path constraint (PC) is updated on every fork/branch
- A jump "**goto s**": update state by advancing to statement s

Example from “Practical Binary Analysis”

```
x = int(argv[0])
y = int(argv[1])
z = x + y
if(x >= 5) {
   foo(x, y, z)
   y = y + z
   if(y < x)
       baz(x, y, z)
   else
       qux(x, y, z)
} else {
   bar(x, y, z)
}
```

Goal of this specific example:

- Find inputs so that all the four functions `foo`, `baz`, `bux`, and `bar` get called.

Example from “Practical Binary Analysis”

```
x = int(argv[0])
y = int(argv[1])
z = x + y
if(x >= 5) {
   foo(x, y, z)
    y = y + z
    if(y < x)
        baz(x, y, z)
    else
        qux(x, y, z)
} else {
   bar(x, y, z)
}
```

`x = int(argv[0])`

$\pi = \top$

$\sigma = \emptyset$

↓

`y = int(argv[1])`

$\pi = \top$

$\sigma = \{x \mapsto \alpha_1\}$

Example from “Practical Binary Analysis”

```
x = int(argv[0])
y = int(argv[1])
z = x + y
if(x >= 5) {
   foo(x, y, z)
   y = y + z
   if(y < x)
      baz(x, y, z)
   else
      qux(x, y, z)
} else {
   bar(x, y, z)
}
```

…

…

z = x + y

$\pi = T$

$\sigma = \{x \mapsto \alpha_1, y \mapsto \alpha_2\}$

Example from "Practical Binary Analysis"

```
x = int(argv[0])
y = int(argv[1])
z = x + y
if(x >= 5) {
   foo(x, y, z)
```

`if (x >= 5)`

$\pi = \mathrm{T}$
$\sigma = \{x \mapsto \alpha_1, \mathrm{y} \mapsto \alpha_2, \mathrm{z} \mapsto \alpha_1 + \alpha_2\}$

false

true

`bar(x, y, z)`

$\pi = \alpha_1 < 5$
$\sigma = \{x \mapsto \alpha_1, \mathrm{y} \mapsto \alpha_2, \mathrm{z} \mapsto \alpha_1 + \alpha_2\}$

`foo(x, y, z)`

$\pi = \alpha_1 \geq 5$
$\sigma = \{x \mapsto \alpha_1, \mathrm{y} \mapsto \alpha_2, \mathrm{z} \mapsto \alpha_1 + \alpha_2\}$

Example from "Practical Binary Analysis"

```
x = int(argv[0])
y = int(argv[1])
z = x + y
if(x >= 5) {
    foo(x, y, z)
    y = y + z
    if(y < x)
        baz(x, y, z)
    else
        qux(x, y, z)
} else {
    bar(x, y, z)
}
```

```
y = y + z
```

$\pi = \alpha_1 \geq 5$

$\sigma = \{x \mapsto \alpha_1, \mathrm{y} \mapsto \alpha_2, \mathrm{z} \mapsto \alpha_1 + \alpha_2\}$

$\downarrow$

```
if(y < x)
```

$\pi = \alpha_1 \geq 5$

$\sigma = \{x \mapsto \alpha_1, \mathrm{y} \mapsto \alpha_2 + \alpha_1 + \alpha_2, \mathrm{z} \mapsto \alpha_1 + \alpha_2\}$

Example from “Practical Binary Analysis”

```
x = int(argv[0])
y = int(argv[1])
z = x + y
if(x >= 5) {
   foo(x, y, z)
    y = y + z
    if(y < x)
        baz(x, y, z)
    else
        qux(x, y, z)
} else {
   bar(x, y, z)
}
```

Note: the book “Practical Binary Analysis” uses slightly different notation, but it’s semantically equivalent:

```
if(y < x)
```

$\pi = \phi_1 \geq 5$

$\sigma = \{\phi_1 = \alpha_1, \phi_2 = \alpha_2, \phi_3 = \phi_1 + \phi_2, \phi_4 = \phi_2 + \phi_3\}$

$x \mapsto \phi_1$

$y \mapsto \phi_4$

$\mathrm{z} \mapsto \phi_2$

What about loops?

Impact of loops (or recursion) in the analyzed code?

- Results in infinitely many and/or arbitrarily long paths.
- Need to set a bound on the number of iterations, depth of recursion, and/or size of PCs.

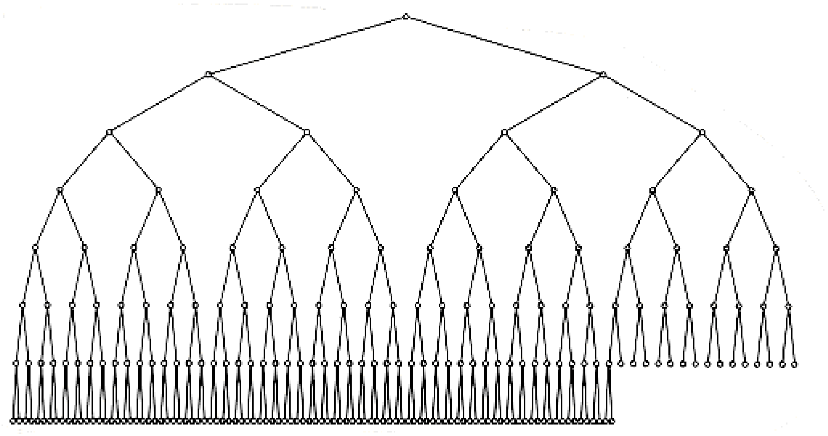
More general, path explosion is a big obstacle

- Heuristics to prioritize exploration of “interesting” branches
- How do determine which branches are interesting?

How to schedule which branches to explore?

Various heuristics are possible:

- Select at random
- Select based on code coverage
- Prioritize based on distance to "interesting" code sections



Can also prune branches by checking if the PC is satisfiable

- Means no input exists that follows a particular branch

Examples of symbolic execution engines

KLEE (symbolic execution for C, built on LLVM)

- Found many bugs in open-source GNU Coreutils tools
- Open-source & well-maintained: https://klee.github.io

And many others:

- PathFinder (Java bytecode)
- Otter (C)
- RubyX (Ruby)
- SymDroid (Dalvik bytecode)
- SymDroid (Dalvik bytecode)
- Mayhem (binaries)
- angr (binaries)
- SymJS (Java Script)

Limitations

- Path explosion
- Memory modeling: pointers, aliasing, pointer arithmetic
- Cumulative cost of solving path constraints
- Environment modeling: dealing with calls to native / system / library functions
- …

Libraries and native code

- At some point, symbolic execution reaches the “edges” of the program being executed
  - That is, when interacting with the environment / OS
  - Library, system, or assembly code calls
- If all arguments are concrete, could just execute the function
- We could symbolically execute this “environment” code
  - For instance, symbolically execute the standard libc library…
  - …but due to its complexity, symbolic execution will easily get stuck
  - Could use a simpler libc library, such as uclibc

Libraries and native code

- If arguments are symbolic, use a **model of the environment**
  - For instance, implement a (simplified) symbolic file system
  - This is a lot of work. And model may differ for every OS.
- Last resort is to **concretize symbolic arguments**
  - For instance, a system call is being called with a symbolic argument
  - The symbolic execution engine asks the SMT solver for a concrete variable assignment that satisfies the current path constraint
  - Perform the system call using the concrete value
  - Downside: the analysis is no longer *complete*, that is, we may miss possible executions (and hence bugs)

Example of concretizing symbolic variables

```
void fun(int arg) {
  if (arg == 0)
    return
  if (arg > 100)
    return

  r=syscall(arg+1);
  return r
}
```

Mark `arg` as equal to symbolic variable α

Example of concretizing symbolic variables

```
void fun(int arg) {
  if (arg == 0)
    return
  if (arg > 100)
    return

  r=syscall(arg+1);
  return r
}
```

Path constraint: $\alpha \neq 0 \land \alpha \leq 100$

- Syscall argument: $\alpha + 1$
- Can't symbolically execute syscall
- Ask SMT solver for concrete value for $\alpha + 1$ given that $\alpha \neq 0 \land \alpha \leq 100$
- Execute syscall(101)

→ Info that the syscall may behave differently on other arguments is lost.

Note: there are multiple trade-offs when interacting with uninstrumented code. See https://github.com/klee/klee/issues/833 for some considerations

How are path constraints specified & evaluated?

SMT: Satisfiability Modulo Theories

A generalization of SAT (Boolean satisfiability problem):

- SAT: is there an assignment that satisfies a Boolean formula?

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge \neg x_1$$

- SAT is an NP-Complete problem, so hard to efficiently solve.

We don’t want to write Boolean formulas. We instead use SMT:

- Write formulas using more “high-level constructs”
- These constructs can be more powerful than SAT formulas (i.e., they can model more problems)

“High level constructs”? We call them theories.

Supported theories depend on solver. Common theories are:

› **Integer arithmetic**: models +, –, <, ≤,… with usual meanings

› **Uninterpreted functions**: models functions while ignoring their internals. For instance, models that $x = y \Rightarrow f(x) = f(y)$

› **Bit vectors**: models machine integers that can over/underflow

› **Arrays**: models (infinite-size) arrays to which we can store values and select elements from. Indices can be “symbolic”.

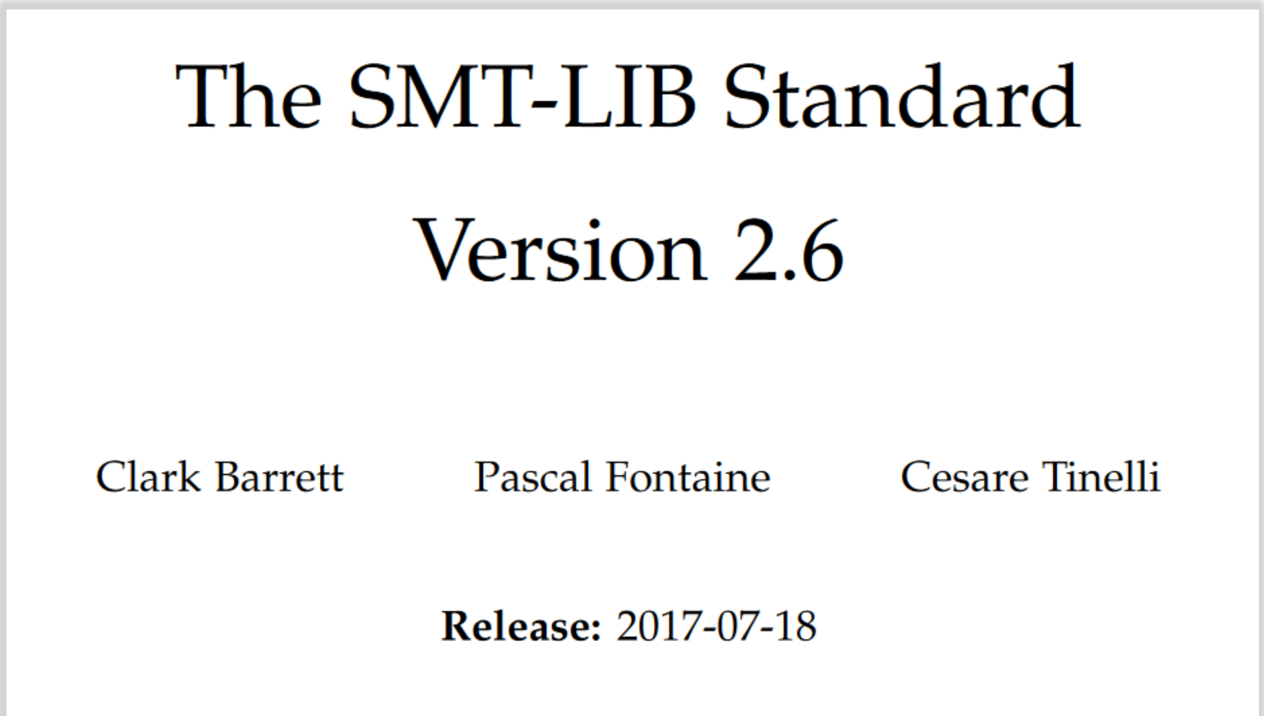
› …

More optional info about SMT theories be found on https://microsoft.github.io/z3guide/docs/theories/Arithmetic and on:

- SMT: Equality Logic With Uninterpreted Functions: https://www21.in.tum.de/teaching/sar/SS20/6.pdf
- SMT - Bit Vectors: https://www21.in.tum.de/teaching/sar/SS20/7.pdf

The SMT-LIB standard

SMT solvers should be interchangeable!



The SMT-LIB Standard

Version 2.6

Clark Barrett Pascal Fontaine Cesare Tinelli

Release: 2017-07-18

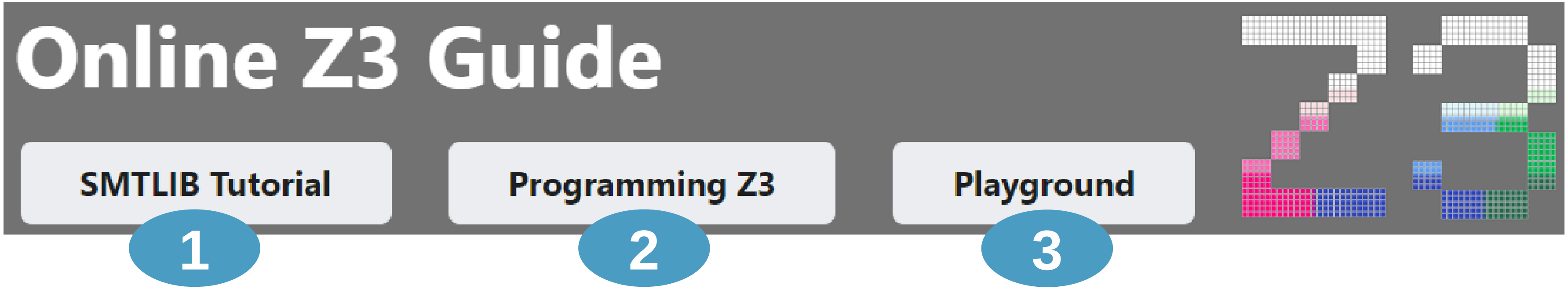
An example SMT-LIB problem

```
; Variable declarations
(declare-const a Int)
(declare-const b Int)

; Constraints
(assert (> a 0))
(assert (> b 0))
(assert (= (+ a b) 7))

; Solve
(check-sat)
(get-model)
```

Try to solve →

```
; Output of Z3
sat
(
  (define-fun b () Int
    1)
  (define-fun a () Int
    6)
)
```

Z3 solver of Microsoft: microsoft.github.io/z3guide/



1. Introduction to SMT-LIB to define SMT problems
2. Interactive tutorial on using SMT-LIB and Z3
3. Interactive playground to define and solve problems ☺

SMT: Satisfiability Modulo Theories

Solving SMT problems is typically NP-hard, and for many useful theories it is undecidable

- Undecidable = there's no algorithm that is correct and always terminates with a sat or unsat result
- Example: nonlinear integer arithmetic is undecidable*

Linear:	multiply x by 3
Non-linear:	multiply x by z

* Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für mathematik und physik*, *38*(1), 173-198.

SMT: Satisfiability Modulo Theories

Solving SMT problems is typically NP-hard, and for many useful theories it is undecidable

- Another example are hard to solve constraints:
  - E.g., find value for x such that $a^x \, mod \, p = b$ (with a, b, and p known)
  - This is the discrete logarithm problem!
- Being able to efficiently solve all constraints would imply that most of crypto is broken…

* Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für mathematik und physik*, *38*(1), 173-198.

How are undecidable theories handled?

- SMT solver may give up, or loop forever
- In many particular cases, it does work!
- Decidable fragments of undecidable theories

In practice, **queries to the SMT may timeout**

- Meaning we can’t decide if a branch is feasible
- Several ways to handle this. One **is concretizing symbolic variables** and then querying the SMT solver again.
- But concretizing symbolic variables may fail…

Practically: modelling fixed-width integers

- Binaries operate on fixed-width integers, e.g., 32-bit integers
- This differs from typical arbitrary length Int datatypes:
  - $2 * 2^{31} = 0 \qquad\qquad 0 - 1 = 2^{32} - 1$
- Fixed-width integers are modelled by **bitvectors**:
  - `(declare-const x (_ BitVec 32))`
  - `#xDEADBEEF`
- Dedicated operators mirror all primitive mathematical operations:
  - `(= y (bvadd x #x10)))`
  - `(bvmul #x2 #x3)`

References

Required reading:

- Chapter 12 “Principles of Symbolic Execution” from the book “Practical Binary Analysis” by Dennis Andriesse.

Optional reading:

- “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)” by Schwartz, IEEE Symposium on Security and Privacy (2010).
- “A Survey of Symbolic Execution Techniques” by Baldoni et al., ACM Computing Surveys (2018).