Security Verification and Bug-finding

José Santos Ana Matos

Bug-finding

- Testing and code reviews is widely used to find bugs in code
  - But, each test only explores one possible execution ⇨ some bugs are still missed (complex features, rare circumstances…)
- Static analysis can analyze all possible runs of a program in a tractable way
  - But, abstraction introduces conservatism ⇨ expertise is required to manage errors

Syntact Analysis

- EmPoWeb* found often than 100 leaks in JS browser extensions
- It uses a simple syntactic analysis that explicitly looks for expressions:
  - chrome.cookies.getAll
  - chrome.cookies.remove
- But there are multiple other ways to access these values that would not be caught by this analysis

Obfuscated JS leak

- Implicit flows can be very subtil

```
function getBit(x) {
    var a = [ "a", "b", "c" ];
    if ((x % 2) === 0) {
        Object.defineProperty(a, "1", {value: "d", configurable: false} )
    }
    a.length = 1;
    return (a.length === 1) ? 1 : 0;
}
```

Changes first element of a to “d”, and makes it non-configurable.

- Bit by bit we can leak all the bits

```
function getBits(x) {
    var bits = [];
    while (x > 0) {
        bits.push(getBit(x));
        x = x >> 1;
    }
    return (bits.length === 0) ? 0 : bits.reverse();
}
```

Obfuscated JS leak

- Malicious code can exploit corner case behaviors of the JavaScript semantics to encode sophisticated information flow
- And now we can access chrome.cookies.remove

```
var x = chrome;
var y1 = "cook";
var y2 = "ies";
var z1  = "rem";
var z2  = "ove";
var w = getBits(x[y1+y2][z1+z2])
```

- Sometimes we need stronger guarantees than simple syntactic analysis!

Symbolic Execution

- Middle ground: Symbolic execution generalizes testing by computing over unknown symbolic variables
- If execution path depends on unknown, conceptually fork symbolic executor
- Each symbolic execution path stands for runs whose concrete values satisfy the path condition

Class Outline

- Program Properties and Noninterference
  - Trace-properties vs. Hyper-properties
- Verification and Bug-finding for Noninterference
  - Self-composition + Symbolic Execution

Class Outline

- Program Properties and Noninterference
  - Trace properties vs. Hyper properties
- Verification and Bug-finding for Noninterference
  - Self-composition + Symbolic Execution

Verification vs. Bug-finding

- Verification: verifying a program S with respect to a property $\mathcal{P}$ means proving that all the behaviors $[\![S]\!]$ of S are contained in $\mathcal{P}$:

  $[\![S]\!] \subseteq \mathcal{P}$

- Bug Finding: debugging a program S with respect to a property $\mathcal{P}$ means finding a behavior of S that is not in $\mathcal{P}$:

  $[\![S]\!] \cap \neg\mathcal{P} \neq \varnothing$

Verification vs. Bug-finding

- Verification
- $[\![S]\!] \subseteq \mathcal{P}$
- Harder: "for all"

- Bug Finding
- $[\![S]\!] \cap \neg\mathcal{P} \neq \emptyset$
- Easier: "exists"

Program properties

- Program property $\mathcal{P}$ = set of "behaviors"
- How do we define behavior?
- Depending on how we define the set of allowed behaviors, we get different classes of properties.

Trace

- Complete trace - a sequence of configurations (cfg) that represents a complete execution according to the small-step semantics
- Finite: $[cfg_0, \ldots, cfg_n]$ such that
  $cfg_0 = \langle S_0, \rho_0 \rangle$ and cfg_n "final" and
  for all $0 \leq i < n$ $cfg_i \Rightarrow cfg_{i+1}$
- Infinite: $[cfg_0, \ldots]$ such that
  $cfg_0 = \langle S_0, \rho_0 \rangle$ and for all $0 \leq i < n$ $cfg_i \Rightarrow cfg_{i+1}$

Traces Properties

$[\![S_0]\!]$ (set of "behaviors" of S_0)

= set of complete traces (both finite and infinite)

$= \{[cfg_0, \ldots, cfg_n]\} \cup \{[cfg_0, \ldots]\}$

remember:
$\{[cfg_0, \ldots, cfg_n]\}$ abbreviates the set of all such that
$cfg_0 = \langle S_0, \rho_0 \rangle$ and $cfg_n = \rho_n$ and for all $0 \leq i < n$ $cfg_i \Rightarrow cfg_{i+1}$

and $[cfg_0', \ldots]$ abbreviates the set of all infinite traces

Safety and Liveness

- Safety Properties: Something bad never happens. Examples:
  - Type Safety
  - Memory Safety (no null pointer exceptions)
- Liveness Properties: Something good will eventually happen. Examples:
  - Termination
  - Absence of memory leaks

WHILE + Errors

Assignment:

$$\frac{\mathcal{A}[\![a]\!]\rho = c}{< x{:=}a, \rho> \Rightarrow \rho[x \mapsto c]} \qquad \frac{\mathcal{A}[\![a]\!]\rho = \lightning}{< x{:=}a, \rho> \Rightarrow \lightning}$$

Skip: $< \text{skip}, \rho> \Rightarrow \rho$

Sequential composition:

$$\frac{<S_1, \rho> \Rightarrow \rho'}{< S_1;S_2, \rho> \Rightarrow <S_2,\rho'>} \qquad \frac{<S_1, \rho> \Rightarrow <S_1', \rho'>}{< S_1;S_2, \rho> \Rightarrow <S_1';S_2, \rho'>'}$$

Conditional test:

$$\frac{\mathcal{B}[\![t]\!]\rho = \lightning}{<\text{if t then } S_1 \text{ else } S_2, \rho> \rightarrow \lightning}$$

$$\frac{\mathcal{B}[\![t]\!]\rho = \text{true}}{<\text{if t then } S_1 \text{ else } S_2, \rho> \Rightarrow <S_1,\rho>} \qquad \frac{\mathcal{B}[\![t]\!]\rho = \text{false}}{<\text{if t then } S_1 \text{ else } S_2, \rho> \Rightarrow <S_2,\rho>}$$

While loop:

$$\frac{\mathcal{B}[\![t]\!]\rho = \lightning}{<\text{while t do S}, \rho> \Rightarrow \lightning}$$

$$\frac{\mathcal{B}[\![t]\!]\rho = \text{true}}{<\text{while t do S}, \rho> \Rightarrow <\text{S;while t do S} , \rho>} \qquad \frac{\mathcal{B}[\![t]\!]\rho = \text{false}}{<\text{while t do S}, \rho> \Rightarrow \rho}$$

Safety property - NoDivZero

"No program execution performs a division by zero"

$\mathcal{NoDivZero} = \{[cfg_0, \ldots, cfg_n] \mid cfg_n \neq \lightning\} \cup \{[cfg_0, \ldots]\}$

Verification of that S satisfies the property:

$[\![S]\!] \subseteq \mathcal{NoDivZero}$

Verification of that S satisfies *the property*:

$[\![S]\!] \subseteq \mathcal{NoDivZero}$

Liveness property - Termination

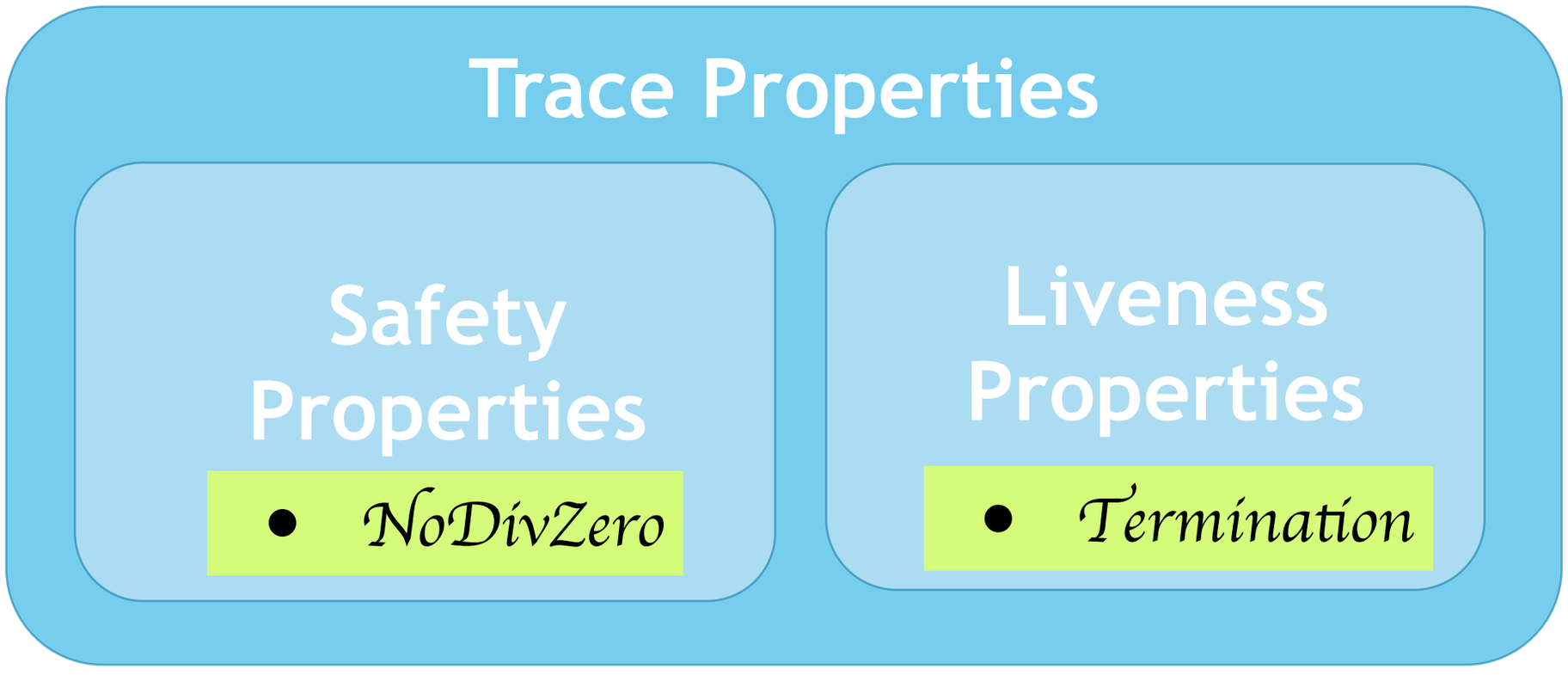
"All program executions eventually terminate correctly"

$\mathcal{T}ermination = \{[cfg_0, \ldots, cfg_n] \mid cfg_n = \rho\}$

Verification of that S satisfies the property:

$[\![S]\!] \subseteq \mathcal{T}ermination$

Trace Properties



Is Noninterference a Trace Property?

- **Deterministic Input-Output Noninterference** -
  A program S is secure if for every security level L and for all pairs of memories ρ_1 and ρ_2 such that $\rho_1 \sim_L \rho_2$, we have that
  $\langle S, \rho_1 \rangle \rightarrow \rho_1'$ and $\langle S, \rho_2 \rangle \rightarrow \rho_2'$ implies $\rho_1' \sim_L \rho_2'$.

2-Traces Property

$[\![S_0]\!]$ (set of "behaviors of S_0) = set of pairs of traces

$= \{([cfg_0, \ldots, cfg_n],[cfg_0', \ldots, cfg_m'])\}$

$\cup \{([cfg_0, \ldots, cfg_n],[cfg_0', \ldots])\}$

$\cup \{([cfg_0, \ldots],[cfg_0', \ldots,cfg_m'])\}$

$\cup \{([cfg_0, \ldots],[cfg_0', \ldots])\}$

Includes all pairs of traces, finite or infinite.

Noninterference: a 2-Trace Property

"No pair of program executions reveals an information leak"

Include all pairs of finite traces witch preserve low memory indistinguishability…

Noninterference =

$\{([cfg_0, \ldots, cfg_n],[cfg_0', \ldots, cfg_m']) \mid \rho_0 \sim_L \rho_0' \text{ or } \rho_n \sim_L \rho_m'\}$

$\cup \{([cfg_0, \ldots, cfg_n],[cfg_0', \ldots])\}$
$\cup \{([cfg_0, \ldots],[cfg_0', \ldots, cfg_m'])\}$
$\cup \{([cfg_0, \ldots],[cfg_0', \ldots])\}$

… and all pairs of traces that are not both finite.

Noninterference Verification vs. Bug-finding

"No pair of program executions reveals an information leak"

- Verification

  $[\![S]\!] \subseteq \mathcal{N}oninterference$

  All pairs of traces satisfy the property

- Bug-finding

  $[\![S]\!] \cap \neg\, \mathcal{N}oninterference \neq \emptyset$

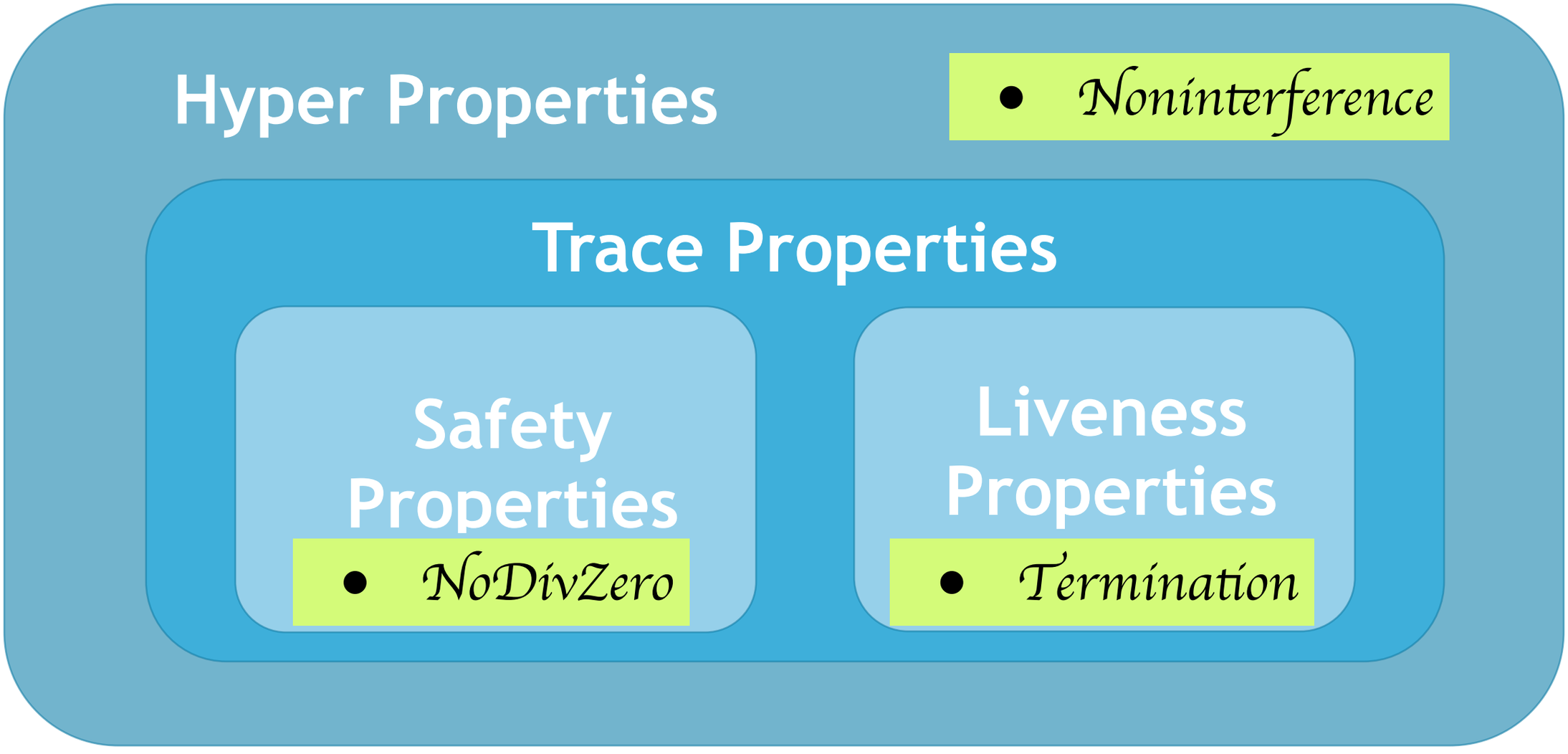
  A pair of traces that does not satisfy the property

A program with an information flow bug

- Deterministic Input-Output Noninterference -
  A program S is secure if for every security level L and for all pairs of memories ρ_1 and ρ_2 such that $\rho_1 \sim_L \rho_2$, we have that
  $\langle S, \rho_1 \rangle \rightarrow \rho_1'$ and $\langle S, \rho_2 \rangle \rightarrow \rho_2'$ implies $\rho_1' \sim_L \rho_2'$.

- S does not satisfy *Noninterference*
  = S has a *Noninterference* bug if
  there exist ρ_1 and ρ_2 such that $\rho_1 \sim_L \rho_2$,
  $\langle S, \rho_1 \rangle \rightarrow \rho_1'$ and $\langle S, \rho_2 \rangle \rightarrow \rho_2'$ but $\rho_1' \not\sim_L \rho_2'$.

Trace Properties



Class Outline

- Program Properties and Noninterference
  - Trace-properties vs. Hyper-properties
- Verification and Bug-finding for Noninterference
  - Self-composition + Symbolic Execution

Verification tools

- How to verify Noninterference?
- Existing verification tools are mainly for trace properties
- Scalable program analysis are hard to design and implement, especially when targeting real world languages
- If we can reduce Noninterference to a trace property, we can use an existing analysis to check it instead of building a new one from scratch!

Verifying Noninterference by Self-composition

$[\![S]\!] \subseteq$ *Noninterference* (2-trace property)

iff

$[\![C(S)]\!] \subseteq$ *SafetyNoninterference ???* (trace property)

Idea: Self-Composition

- How to perform two runs of S in only one go?
  - Create a "copy" $\underline{S}$ of program S using fresh variables from $\underline{\mathcal{V}ar}$. Then execute S;$\underline{S}$.
- How to assume and assert same "low" variables in the encoded "two runs"?
  - Use a memory $\rho+\underline{\rho}$ where ρ and $\underline{\rho}$ don't overlap but between which we can map variables

✓ To automatize: Define a source-to-source compiler

$$C(S) = \underline{S}$$

A program with an information flow bug

- Deterministic Input-Output Noninterference -
  A program S is secure if for every security level L and for all pairs of memories ρ_1 and ρ_2 such that $\rho_1 \sim_L \rho_2$, we have that
  $\langle S, \rho_1 \rangle \rightarrow \rho_1'$ and $\langle S, \rho_2 \rangle \rightarrow \rho_2'$ implies $\rho_1' \sim_L \rho_2'$.

- S does not satisfy *SafetyNoninterference*
  = S has a *SafetyNoninterference* bug if
  there exist ρ_1 and ρ_2 such that $\rho_1 \sim_L \rho_2$,
  $\langle S, \rho_1 \rangle \rightarrow \rho_1'$ and $\langle S, \rho_2 \rangle \rightarrow \rho_2'$ but $\rho_1' \not\sim_L \rho_2'$.

Safety Noninterference

Deterministic Input-Output Safety Noninterference - A program S is secure if, *for S;S obtained by self composition of S*, and for every observation level L and for all memories $\rho+\underline{\rho}$ such that $\rho \sim_L \underline{\rho}$, we have that $\langle S;\underline{S}, \rho+\underline{\rho}\rangle \rightarrow \rho'+\underline{\rho}'$ implies $\rho' \sim_L \underline{\rho}'$.

$\rho \nsim_L \underline{\rho}$ iff

For all x such that $\Gamma(x) \leq L$, then $\rho(x) = \underline{\rho}(\underline{x})$

Safety Noninterference

Deterministic Input-Output Safety Noninterference - A program S is secure if, *for S;S obtained by self composition of S*, and for every observation level L and for all memories $\rho+\underline{\rho}$ such that $\rho \sim_L \underline{\rho}$, we have that $\langle S;\underline{S}, \rho+\underline{\rho}\rangle \rightarrow \rho'+\underline{\rho}'$ implies $\rho' \sim_L \underline{\rho}'$.

$\rho \nsim_L \underline{\rho}$ iff

For all x such that $\Gamma(x) \leq L$, then $\rho(x) = \underline{\rho}(\underline{x})$

Noninterference as a Safety Property

"No self-composed execution reveals an information leak"

SafetyNoninterference =

$\{([cfg_0, \ldots, cfg_n]) \mid \rho_0 \underline{\not\sim_L} \underline{\rho_0}$ or $\rho_n \underline{\sim_L} \underline{\rho_n}\} \cup \{[cfg_0, \ldots]\}$

Include all finite traces witch preserve low memory indistinguishability…

… and infinite traces.

Symbolic Execution for Noninterference

To prepare a program S for verification/bug finding for Noninterference by symbolic execution:

1. Produce a program S;$\underline{S}$ by self-composition
2. For each low variable x that appears in S:
   - add assumptions and assertions x=$\underline{x}$, at the beginning and end of the program, respectively.

Self-composition for Symbolic Execution

- $x_L := y_H$

```
assume(x = x̲);
x := y;
assert(x = x̲)
```

- if (y_H) then $x_L := 1$ else skip

```
assume(x = x̲);
if (y) then x:=1 else skip;
if (y̲) then x̲:=1 else skip;
assert(x = x̲)
```

Self-composition for Symbolic Execution

$z_L := l;$
if (h_H) then $x_L := l_L$ else skip;
if (!h_H) then $x_L := z_L$ else skip;
$l_L := x_L + y_L$

```
assume(l=l̲ ∧ x=x̲ ∧ y=y̲ ∧ z=z̲)
z:=1;
if(h) then x:=1 else skip;
if(!h) then x:=z else skip;
l:=x+y;
z̲:=1;
if (h̲) then x̲:=1 else skip;
if (!h̲) then x̲:=z̲ else skip;
l̲:=x̲+y̲;
assert(l=l̲ ∧ x=x̲ ∧ y=y̲ ∧ z=z̲)
```

Does the **assertion** hold?

Symbolic Execution: Example 1

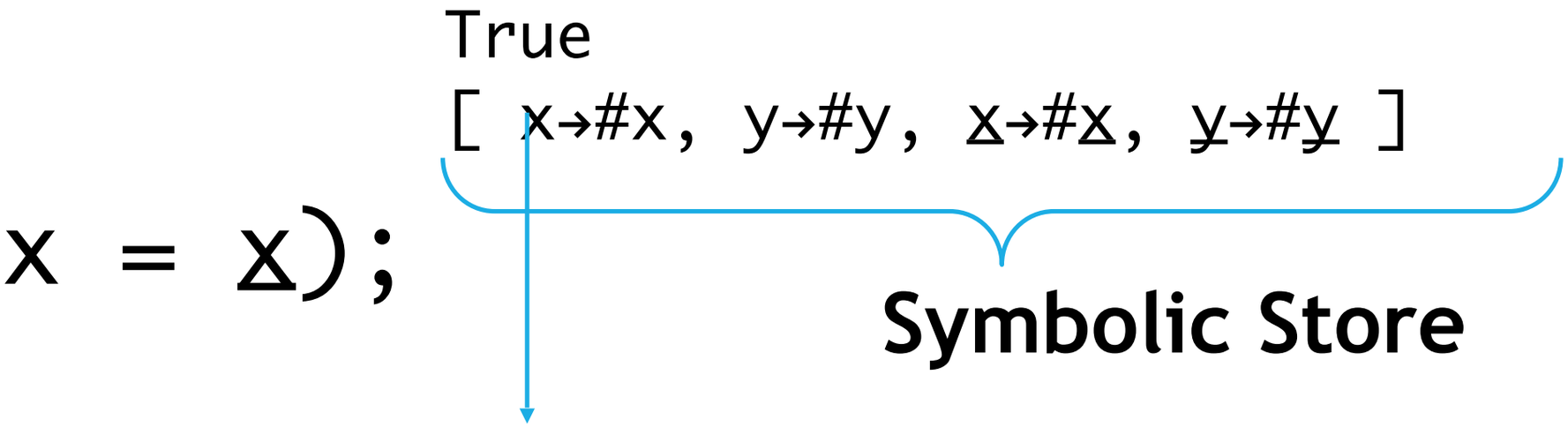
```
assume(x = x);
x := y;
x := y;
assert(x = x)
```

We will execute the generated program symbolically

Instead of using concrete values, we'll use **symbolic variables #x, #y, …**

Self-Composition: Example 1

```
assume(x = x̲);
x := y;
x̲ := y̲;
assert(x = x̲)
```



Path Condition: conjunction of all the expressions on which the execution has branched before reaching the current execution point

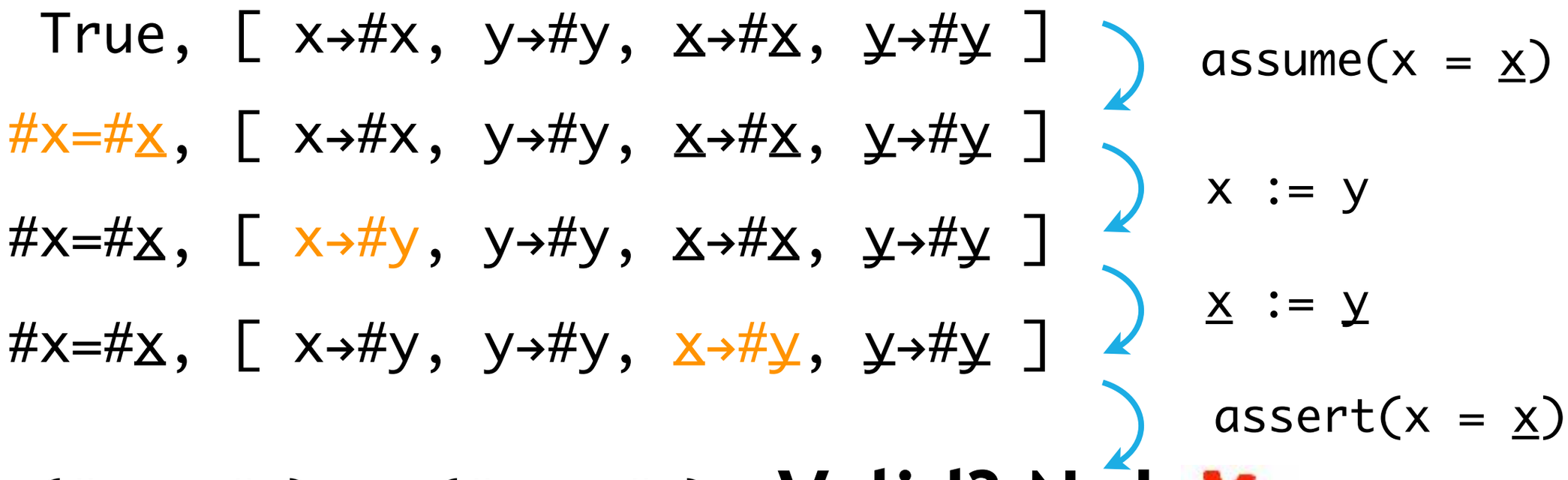
Bug-finding for Noninterference

To search for a Noninterference bug in S:

1. Prepare S for verification/bug finding for Noninterference by symbolic execution
2. Perform a symbolic execution of the new program
    - collect all assumptions over the symbolic variables in the path condition
3. See if path condition implies assertions for all possible symbolic values
    - (equivalently) see if negation of the implication is satisfiable for any symbolic values

Example 1 (bug)

```
assume(x = x̲);
x := y;
x̲ := y̲;
assert(x = x̲)
```

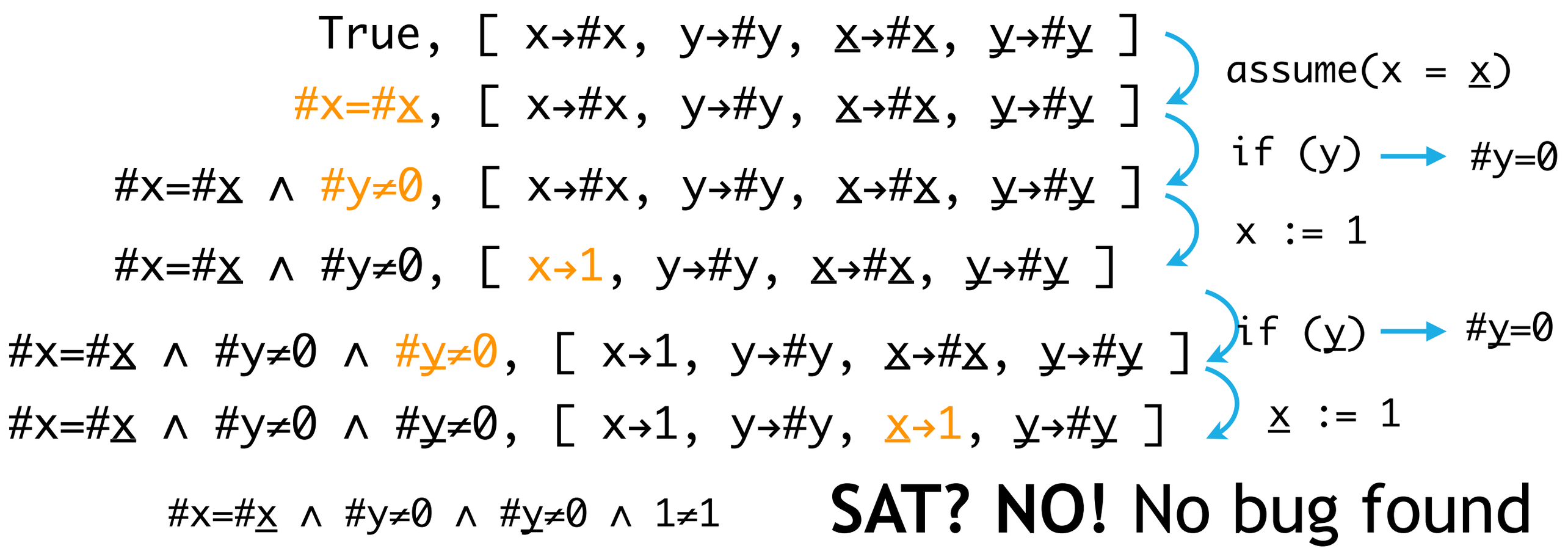


$(\#x = \#\underline{x}) \Rightarrow (\#y = \#\underline{y})$ **Valid? No!** ✗

$(\#x = \#\underline{x}) \wedge (\#y \neq \#\underline{y})$ **Is its negation SAT? Yes!**

Example 2 (no bug)

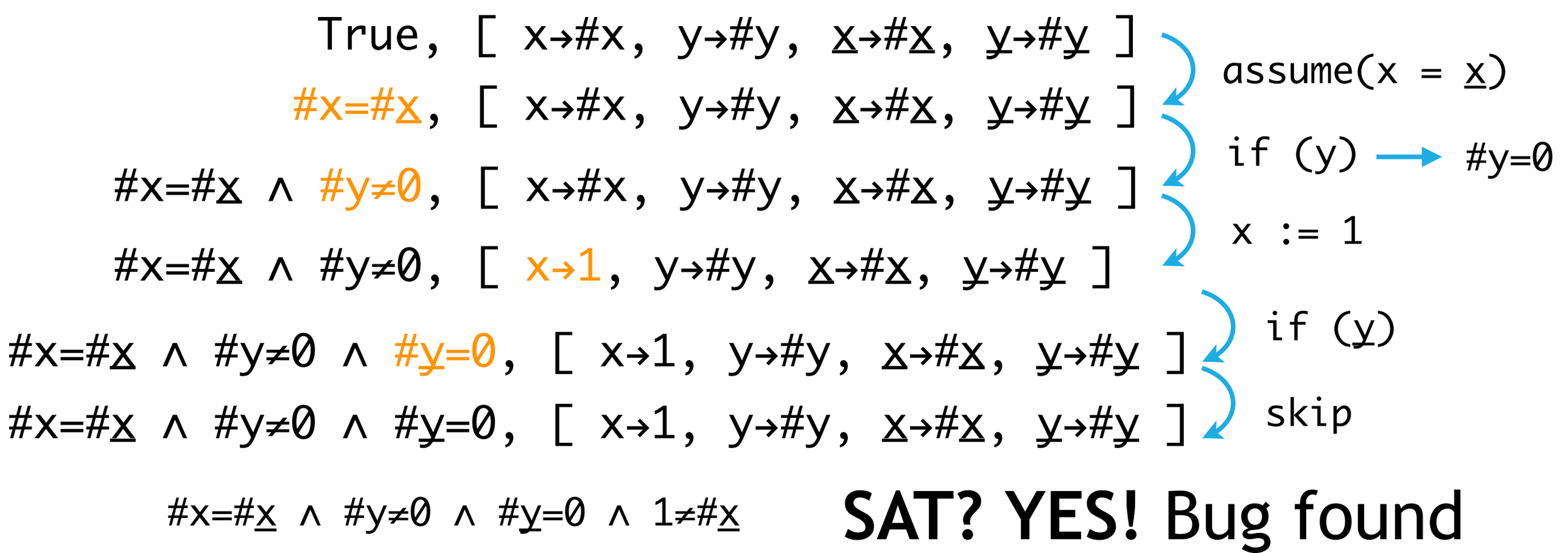
```
assume(x = x̲);
if (y) then x := 1 else skip;
if (y̲) then x̲ := 1 else skip;
assert(x = x̲)
```



SAT? NO! No bug found

Example 2 (finds bug)

```
assume(x = x̲);
if (y) then x := 1 else skip;
if (y̲) then x̲ := 1 else skip;
assert(x = x̲)
```

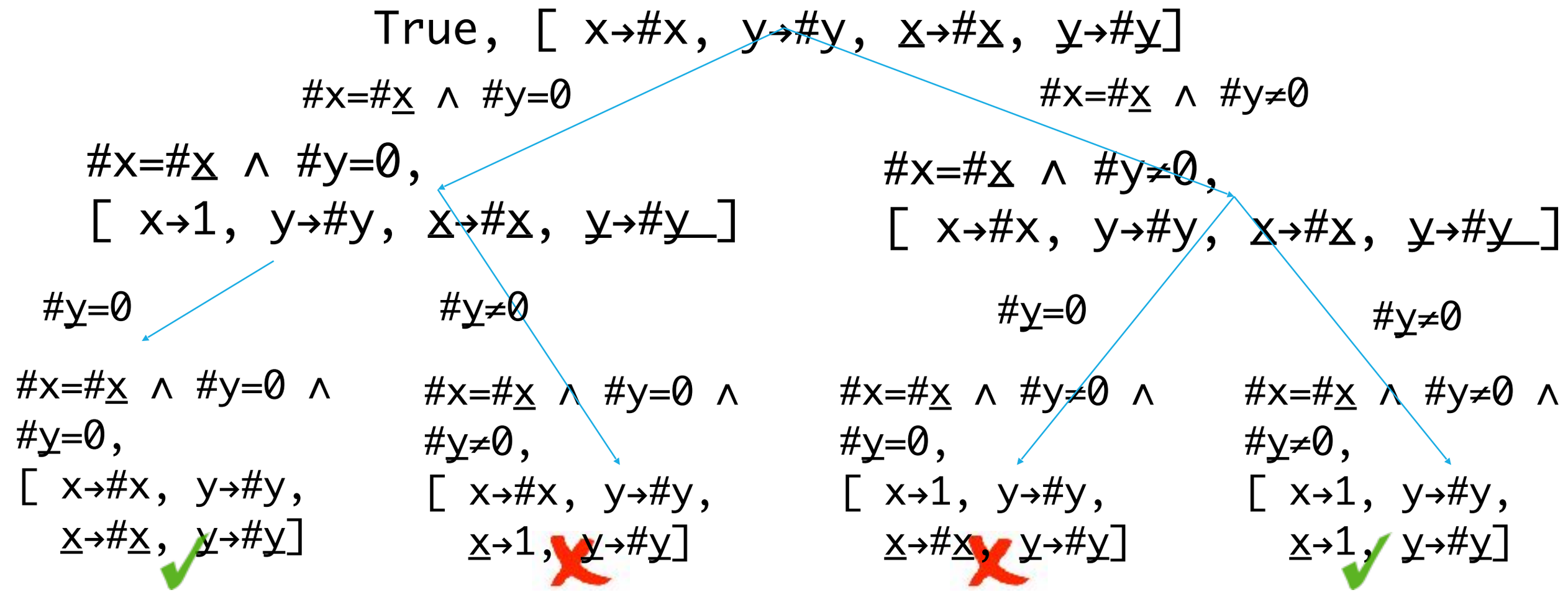


#x=#x̲ ∧ #y≠0 ∧ #y̲=0 ∧ 1≠#x̲

SAT? YES! Bug found

Example 2 (all paths)

```
assume(x = x̲);
if (y) then x := 1 else skip;
if (y̲) then x̲ := 1 else skip;
assert(x = x̲)
```



Verification for Noninterference

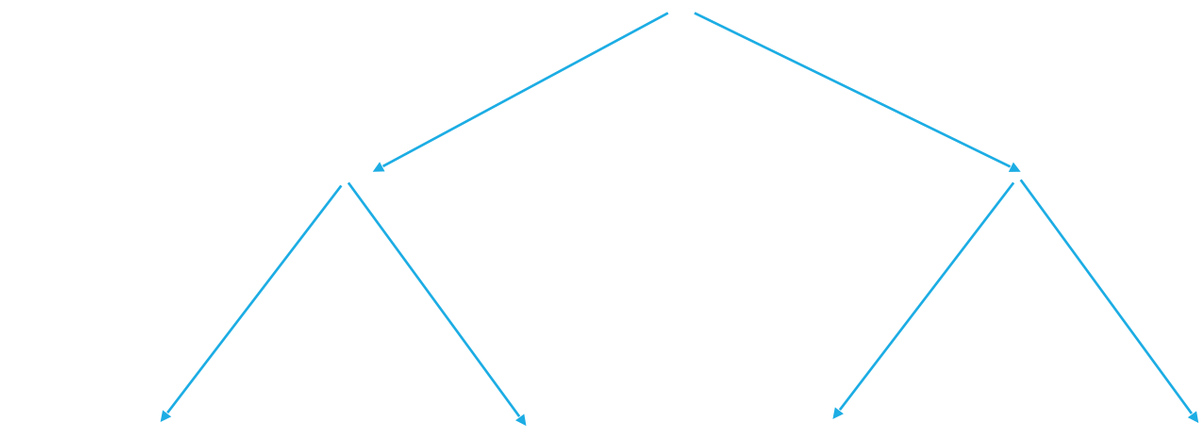
To verify absence of Noninterference bugs in S:

1. Prepare S for verification/bug finding for Noninterference by symbolic execution
2. For each low variable x that appears in S:
   - add assumptions and assertions $x=\underline{x}$, at the beginning end end of the program, respectively.
3. For each symbolic execution path of the new program
   - search for a Noninterference bug
4. S satisfies Noninterference iff no bug is found

Example: Self-Composition

If we find a bug, we know that the program is **not** secure

The program is secure if we covered **all the possible** execution paths without finding a bug.



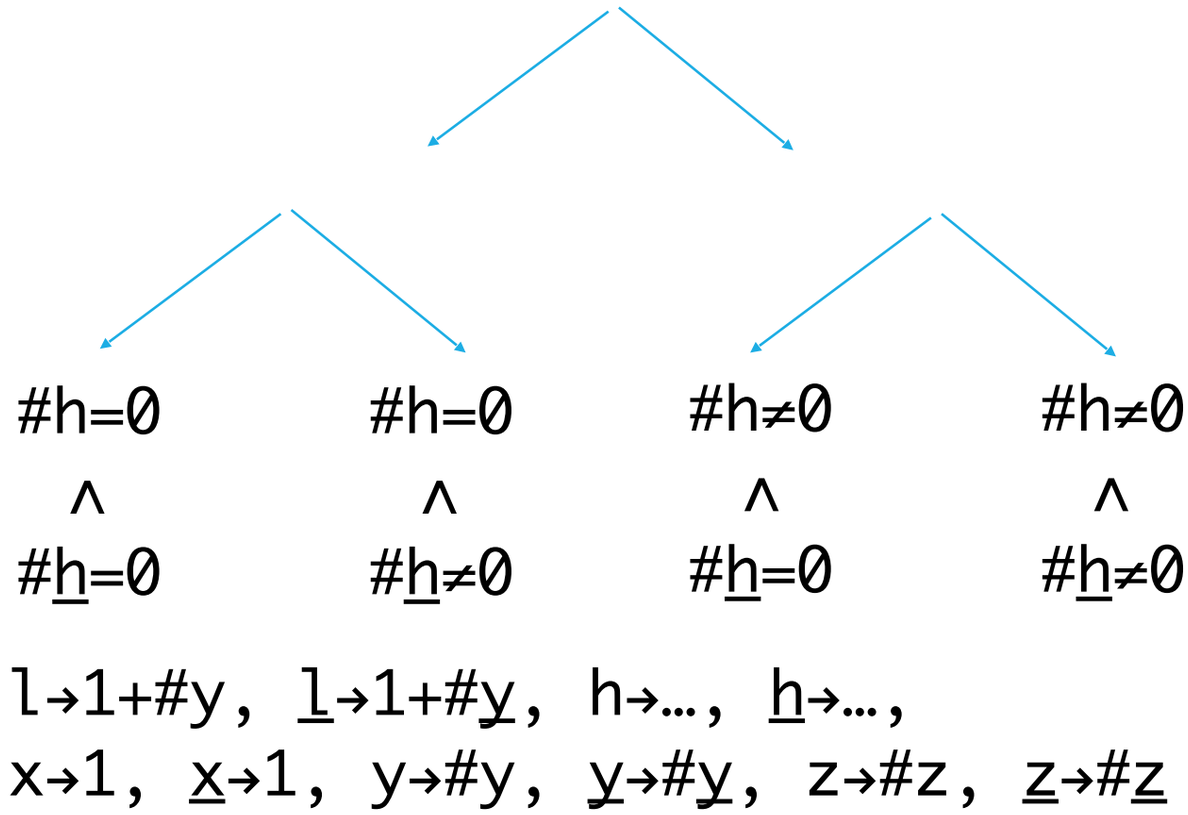
Path condition ⇏ Assertions **V** Path condition ⇏ Assertions **V** Path condition ⇏ Assertions **V** Path condition ⇏ Assertions

Bug found! Program does NOT satisfy Noninterference!

= True

Example 3

```
assume(l=l̲ ∧ x=x̲ ∧ y=y̲ ∧ z=z̲)
z:=1;
if(h) then x:=1 else skip;
if(!h) then x:=z else skip;
l:=x+y;
z:=1;
if (h̲) then x̲:=1 else skip;
if (!h̲) then x̲:=z̲ else skip;
l̲:=x̲+y̲;
assert(l=l̲ ∧ x=x̲ ∧ y=y̲ ∧ z=z̲)
```



The final symbolic store is always the same!

What is the SAT query?

Conclusions (this class)

- Important **security properties** cannot be expressed over individual execution traces of a system but rather as properties of a system as a whole.
  - information flow (confidentiality, integrity)
  - another example: stipulating a bound on mean response time (availability)
- It can still be possible to use trace-based tools such as for symbolic execution to find **security** bugs in programs

Conclusions (part II of course)

- We have studied how to use different language-based techniques for designing mechanisms that enforce a security property over different kinds of programming languages.
- We used as a running example the fundamental security problem of controlling information flows.
- We have seen how different techniques enable to attain and reason about the guarantees offered by a given enforcement mechanism.

CONCOLIC SYMBOLIC EXECUTION – THE MAIN IDEA

Idea: Execute the program concretely **and** symbolically at the same time

P. Godefroid

Why?

- Symbolic execution is often **too** expensive…
- Back-end constraint solvers sometimes (often!) cannot find the answer : **UNKNOWN**

CONCOLIC SYMBOLIC EXECUTION – THE MAIN IDEA

Concolic Testing: Main Algorithm

$Input_0$ = Pick random
Coverage = False
i = 0
While ($Input_i \neq NULL$) {
  (RES_i, PC_i)= Run Program with $Input_i$
  Coverage = Coverage $\vee PC_i$
  $Input_{i+1} \in Models(\neg$ Coverage)
  i = i+1

Concolic Symbolic Execution – the main idea

```
z := 2*y;
if (z = x) {
  if (x > y+10)
{

assert(false)
  } else { skip
};
} else { skip }
```

Step 1:

$Inputs_0$ = [x→22,y→7]

(RES_1, PC_1) = (OK, (x ≠ 2*y))

Coverage = (x ≠ 2*y)

$Inputs_1$ = [x→2,y→1]

Concolic Symbolic Execution – the main idea

```
z := 2*y;
if (z = x) {
  if (x > y+10)
{

assert(false)
  } else { skip
};
} else { skip }
```

Step 2:

$Inputs_1$ = [x→2,y→1]

(RES_2, PC_2) = (OK,(x = 2*y)∧(x<= y+10))

Coverage = (x ≠ 2*y) ∨ ((x = 2*y)∧(x<= y+10))

$Inputs_2$ = [x→30,y→15]

Concolic Symbolic Execution – the main idea

```
z := 2*y;
if (z = x) {
  if (x > y+10)
{

assert(false)
  } else { skip
};
} else { skip }
```

Step 3:

$Inputs_2$ = [x→30,y→15]

(RES_3, PC_3) = (FAIL,(x = 2*y)∧(x>y+10))

Coverage = (x ≠ 2*y) ∨ ((x = 2*y)∧(x<=y+10))∨ ((x = 2*y)∧(x>y+10))

$Inputs_3$ = NULL

Concolic Symbolic Execution – the main idea

```
z := 2*y;
if (z = x) {
  if (x > y+10)
{

assert(false)
  } else { skip
};
} else { skip }
```

Step 3:

$Inputs_2$ = [x→30,y→15]

(RES_3, PC_3) = (FAIL,(x = 2*y)∧(x>y+10))

Coverage = (x ≠ 2*y) ∨ ((x = 2*y)∧(x<=y+10))∨ ((x = 2*y)∧(x>y+10))

$Inputs_3$ = NULL

A LOT MORE TO COVER…

1. Symbolic execution with **data structures**
   - **Lazy-Initialization**
   - **Declassification**
   - **Data structure unfolding**
2.
3. Other Security Properties: **Confinement**
4. **Invariants and verification**