Validation of Abstract Side-Channel Models for Computer Architectures

Roberto Guanciale	KTH
Pablo Buiras	KTH
Andreas Lindner	KTH
Hamed Nemati	CISPA

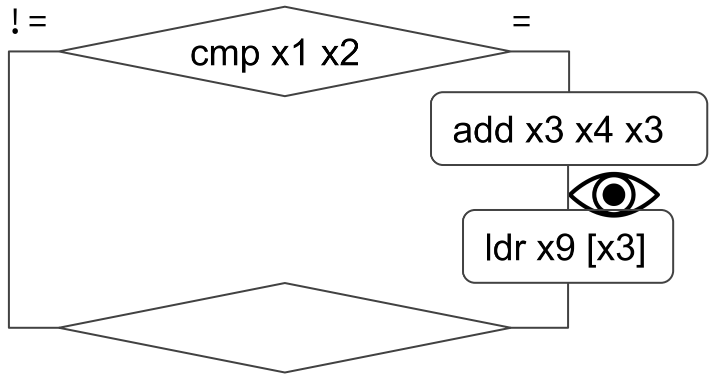
Side channel attacks

- Power consumption, Electromagnetic radiation, Sound, Temperature, Timing (caches)
- Difficult to audit

Unfeasible to have precise deterministic models of these channels.

- Too complex
- Many undocumented features (e.g. cache replacement policies)
- Different processors / same ISA / different channels

Observational models



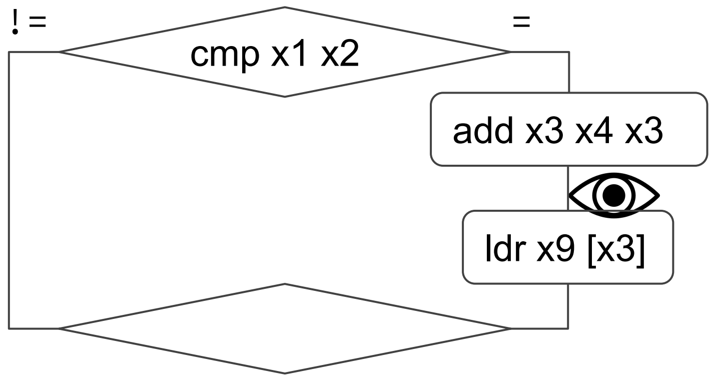
Verify absence of side-channel leakage

- Via constant-time (observation) programming policy (J.B. Almeida et al.)

Require abstract attacker observations

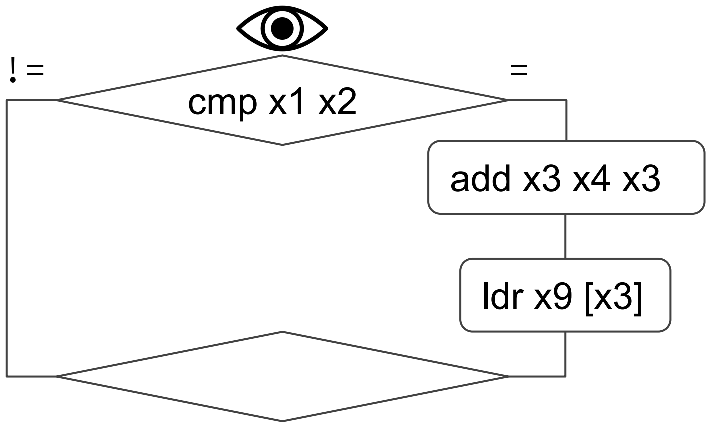
- I.e. a model of what an attacker may see (over approximation)

Observation equivalence classes



x1 != x2	x1 == x2 /\ x4+x3 = 0
	x1 == x2 /\ x4+x3 = 1
	...
	x1 == x2 /\ x4+x3 = 2^64

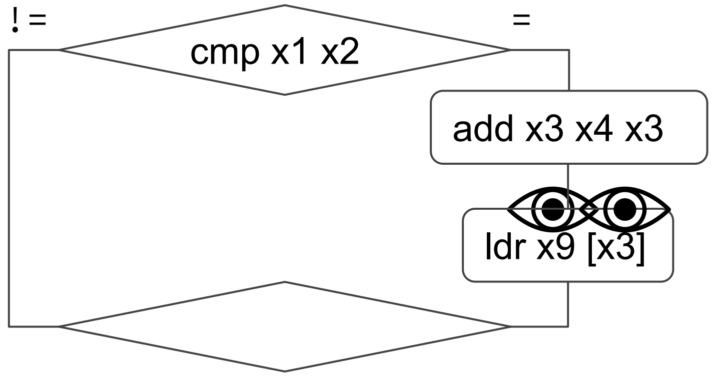
Observation equivalence classes



x1 != x2	x1 == x2 /\ x4+x3 = 0
	x1 == x2 /\ x4+x3 = 1
	...
	x1 == x2 /\ x4+x3 = 2^64

x1 != x2	x1 == x2

Observation equivalence classes

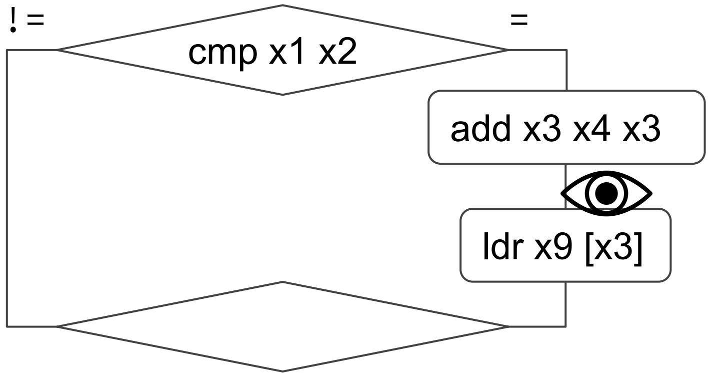


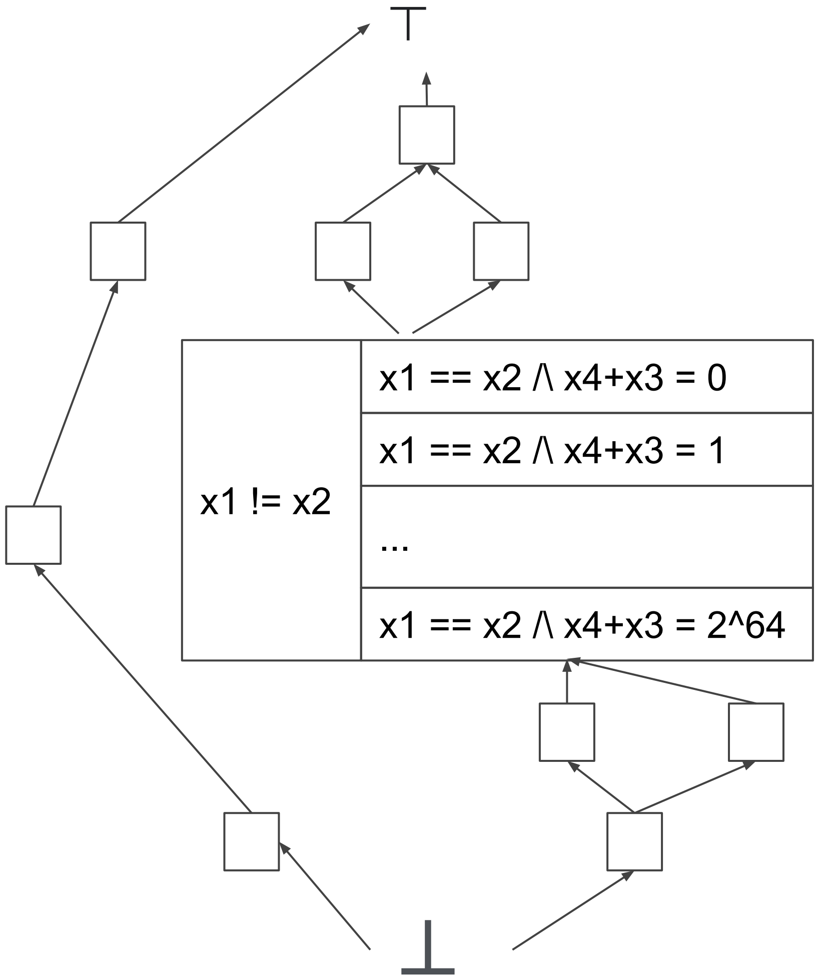
x1 != x2	M[0]=0	M[0]=1	...
	...		

x1 != x2	x1 == x2 /\ x4+x3 = 0
	x1 == x2 /\ x4+x3 = 1
	...
	x1 == x2 /\ x4+x3 = 2^64

x1 != x2	x1 == x2

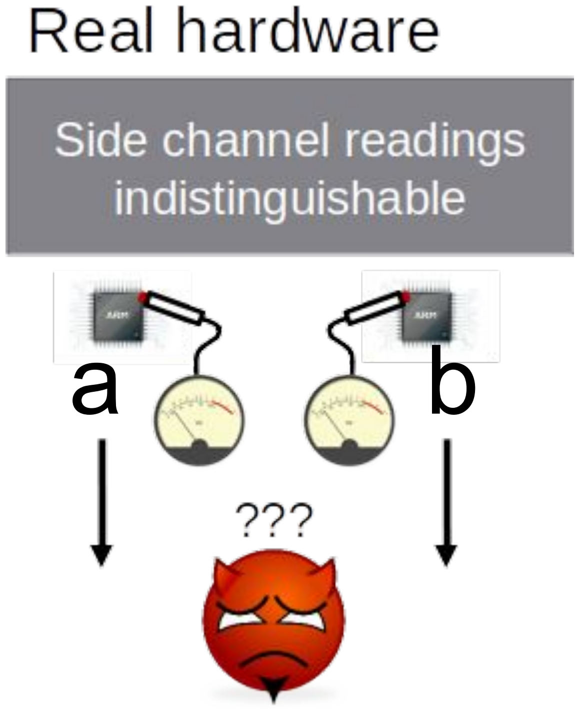
Model Lattice





Side Channel Abstract Model Validation

P(a) ~👁 P(b) ==>



- Are the existing models sound?
- Program may be wrongly considered secure

Are the existing models sound? No, Spectre! (P. Kocher et al.)

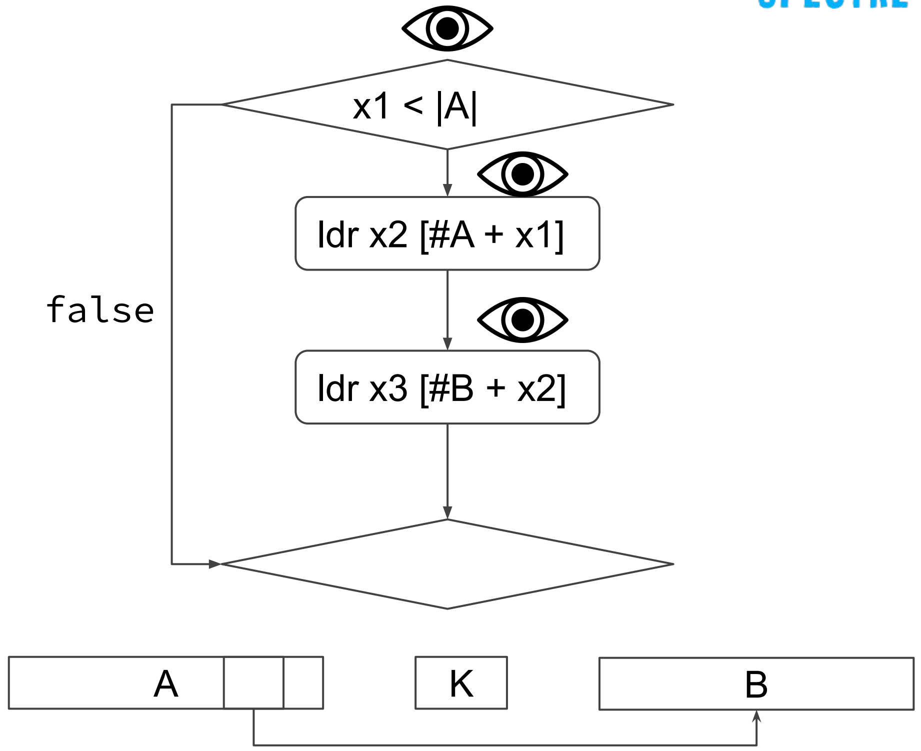
Assume for every every i

- 0 <= A[i] < |B|

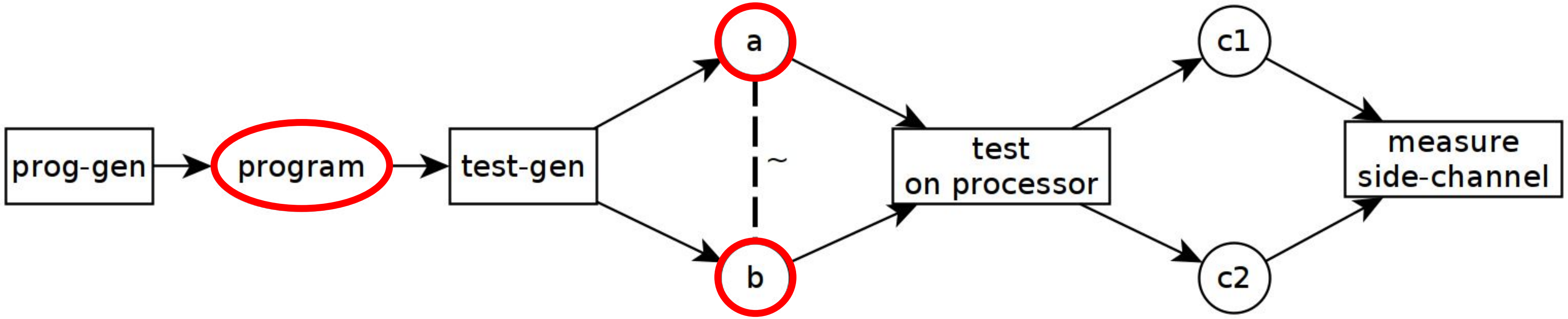
Assumptions and condition ensure no out-of-bound memory read.

Observations depend on

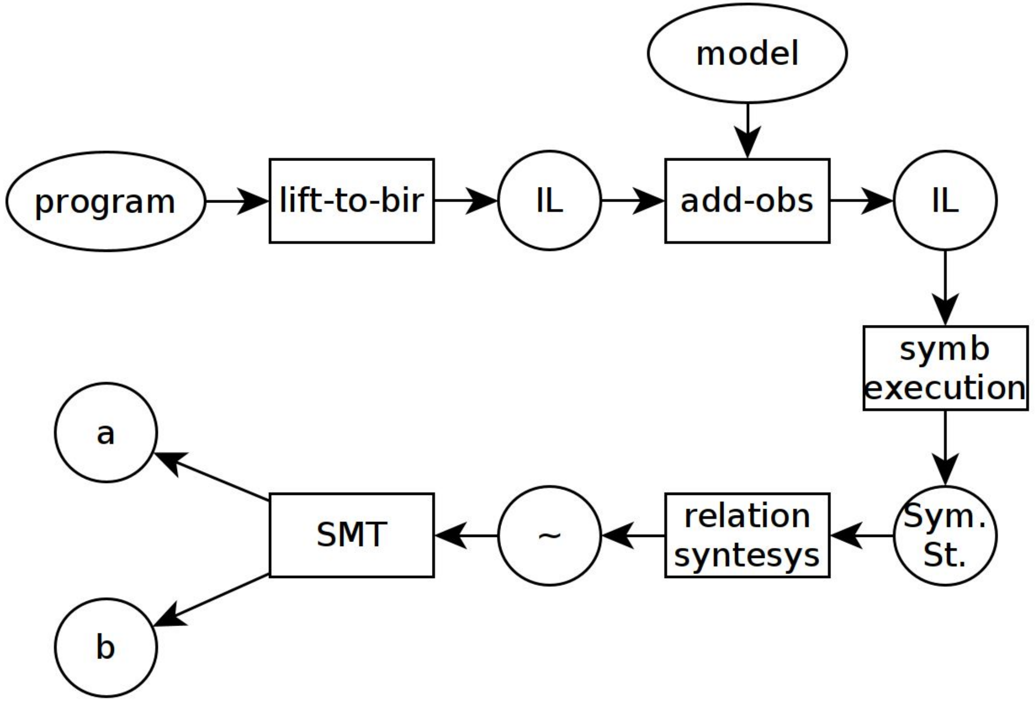
- Position of A and B
- Size of A and B
- Content of A and B
- Input x1



SCAM-V: Validation via testing

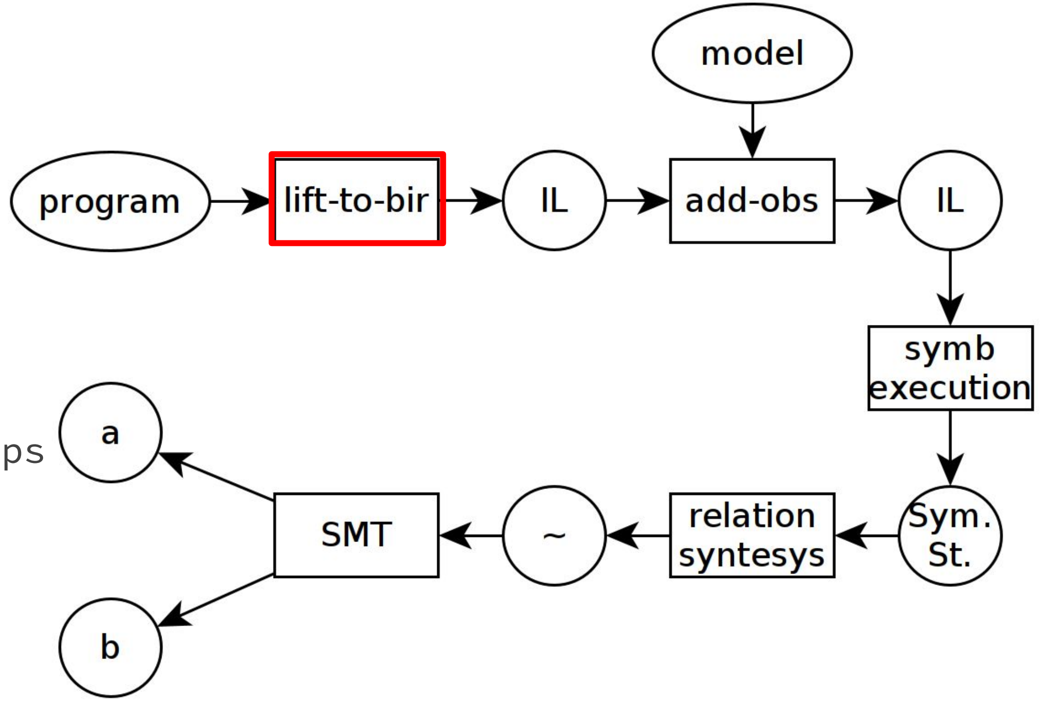


test-gen


model
program
lift-to-bir
IL
add-obs
IL
symb execution
Sym. St.
relation syntesys
~
SMT
a
b

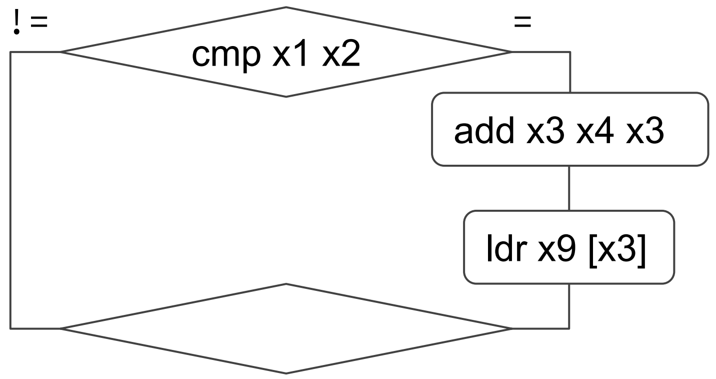
BIR

- Abstract Assembly Language
- Infinite number of register variables
- Assignments, jumps, cond. jumps



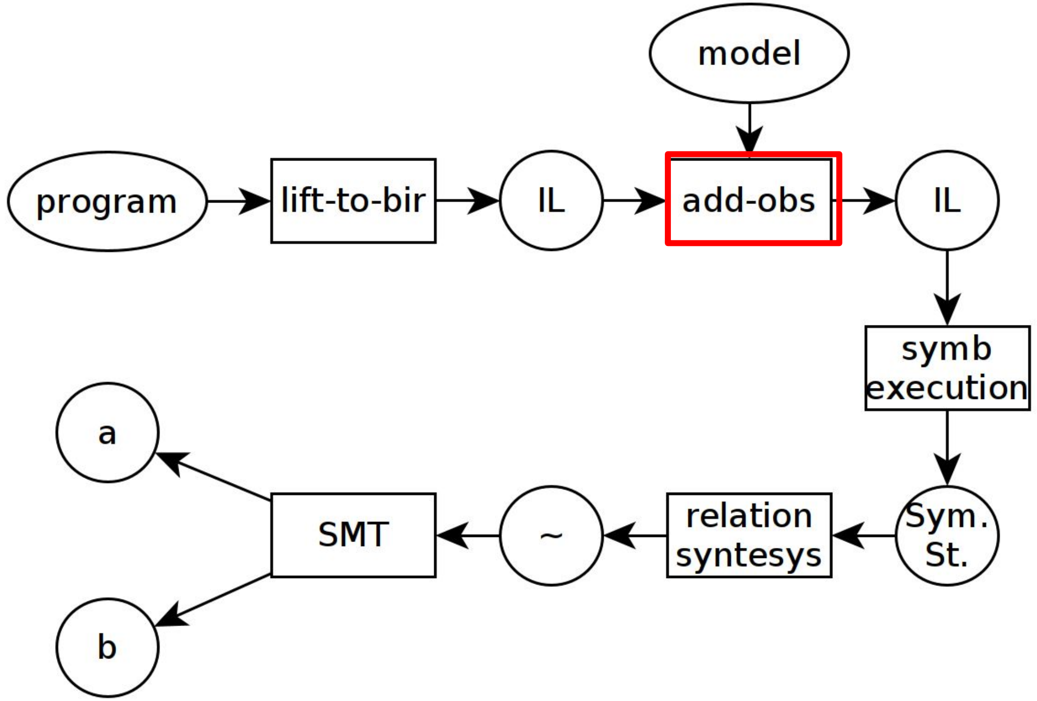
BIR

```
0 : CJMP x1=x2 4 12

4: x3 = x4 + x3; JMP 8

8 : x9 = MEM[x3]; JMP 12

12 : HALT
```



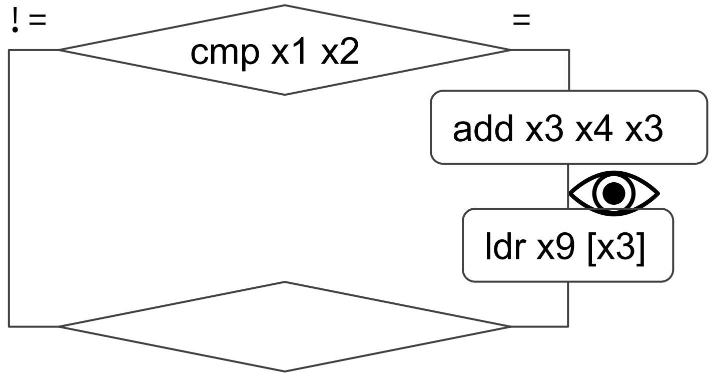
Add observations

- Program transformation that inlines observations
- Different transformations for different models



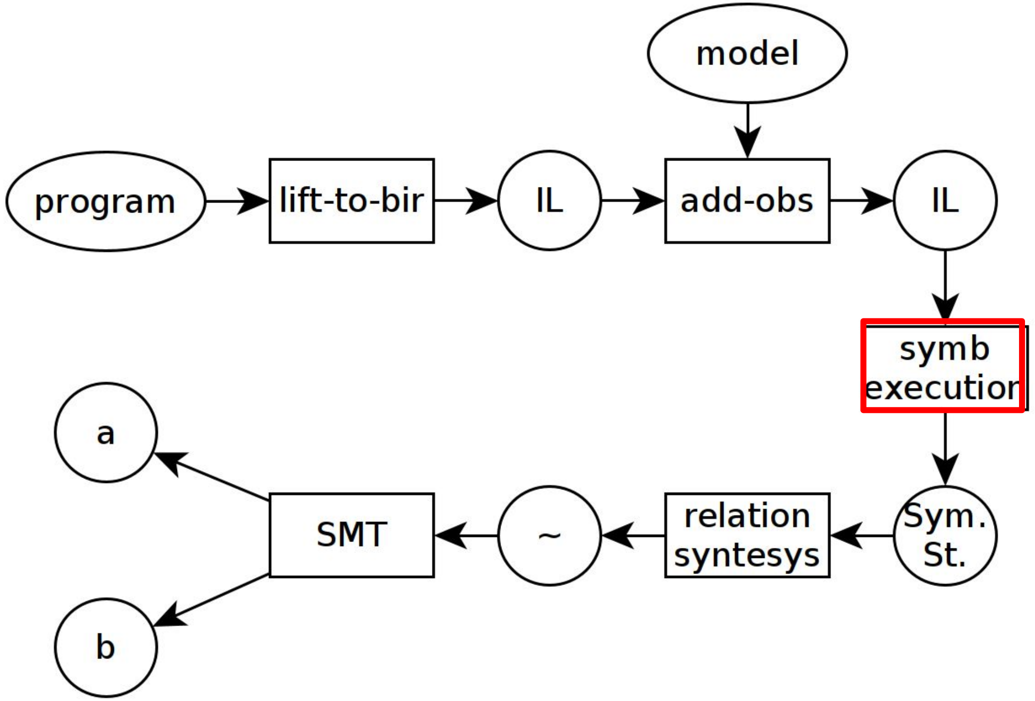
BIR

```
0 : CJMP x1=x2 4 12

4: x3 = x4 + x3; JMP 8

8 : Obs(x3); x9 = MEM[x3]; JMP 12

12 : HALT
```

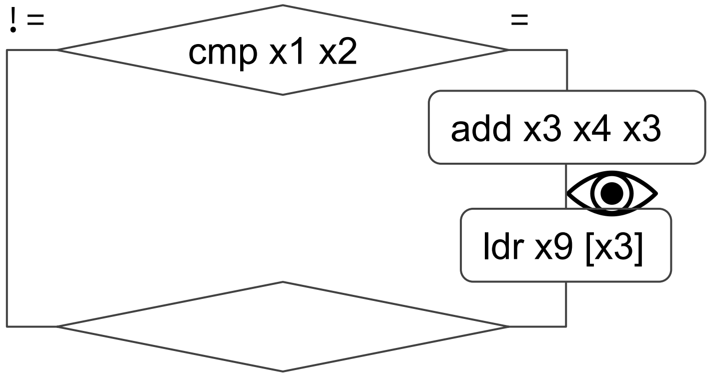


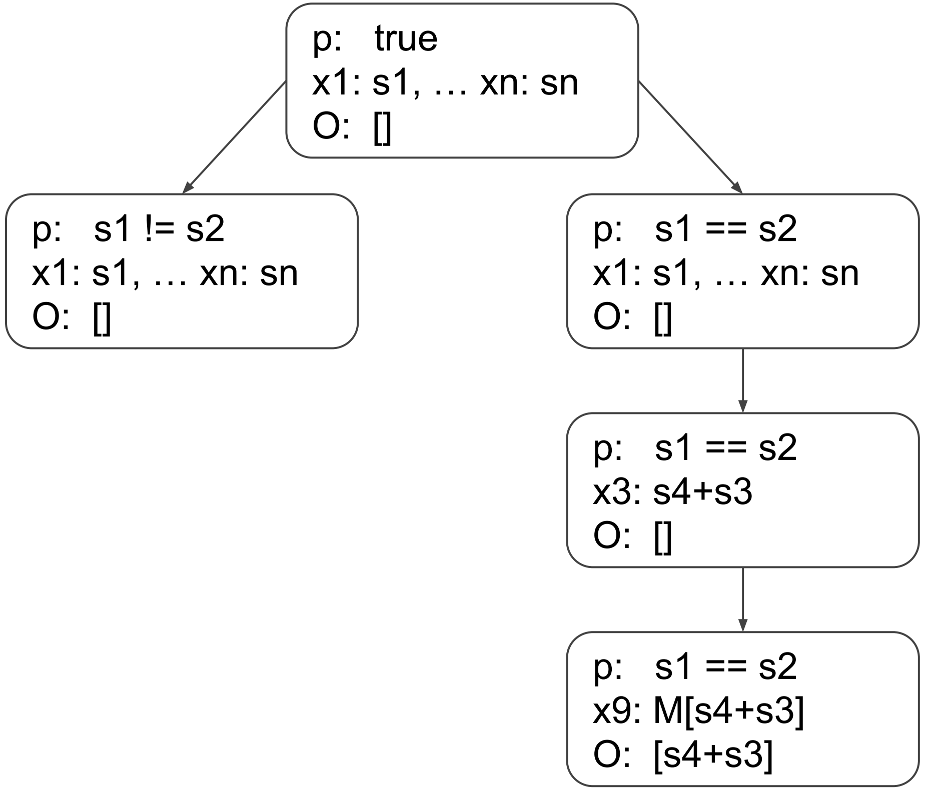
Symbolic execution

- States have symbolic path and symbolic assignments
- Plus a symbolic observation list



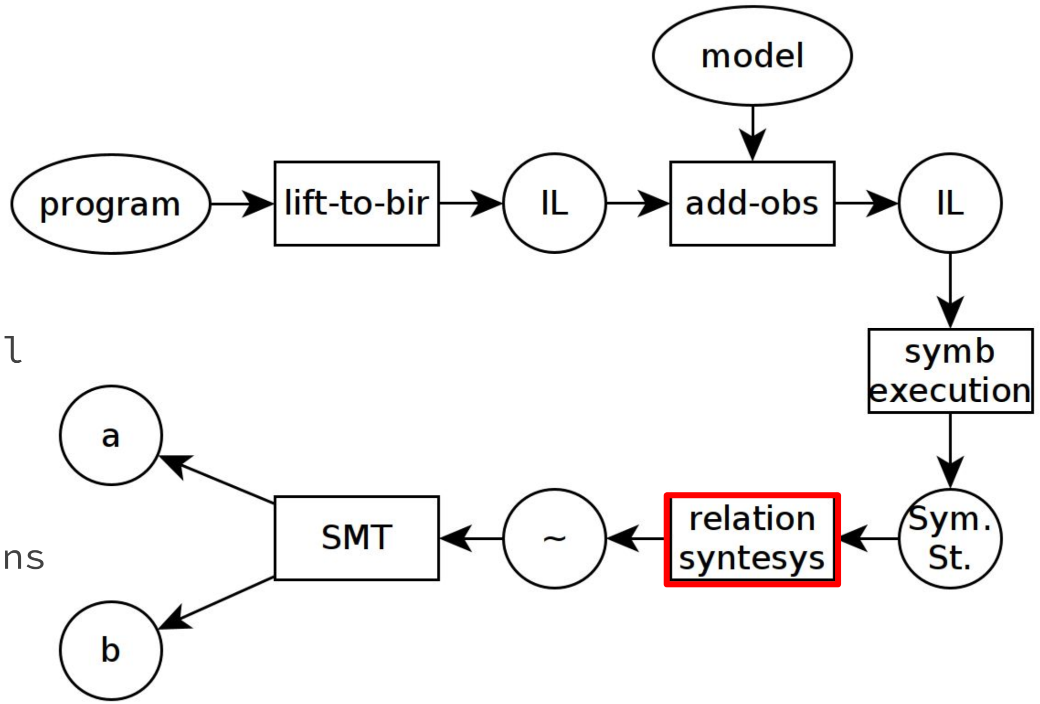
Symbolic execution





Relation synthesis

- Self composition
- Cartesian product of the final symbolic states (i.e. all possible pair of execution paths)
- Impose equality of observations



Relation synthesis

p: s1 == s2
...
O: [s3+s4]

p: s1 != s2
...
O: []

p: s1 == s2
...
O: [s3+s4]

(a.x1 == a.x2)/\(b.x1 == b.x2)
==>
[a.x3+a.x4] == [b.x3+b.x4]

(a.x1 == a.x2) /\ (b.x1 != b.x2)
==>
[a.x3+a.x4] == []

$\wedge$

p: s1 != s2
...
O: []

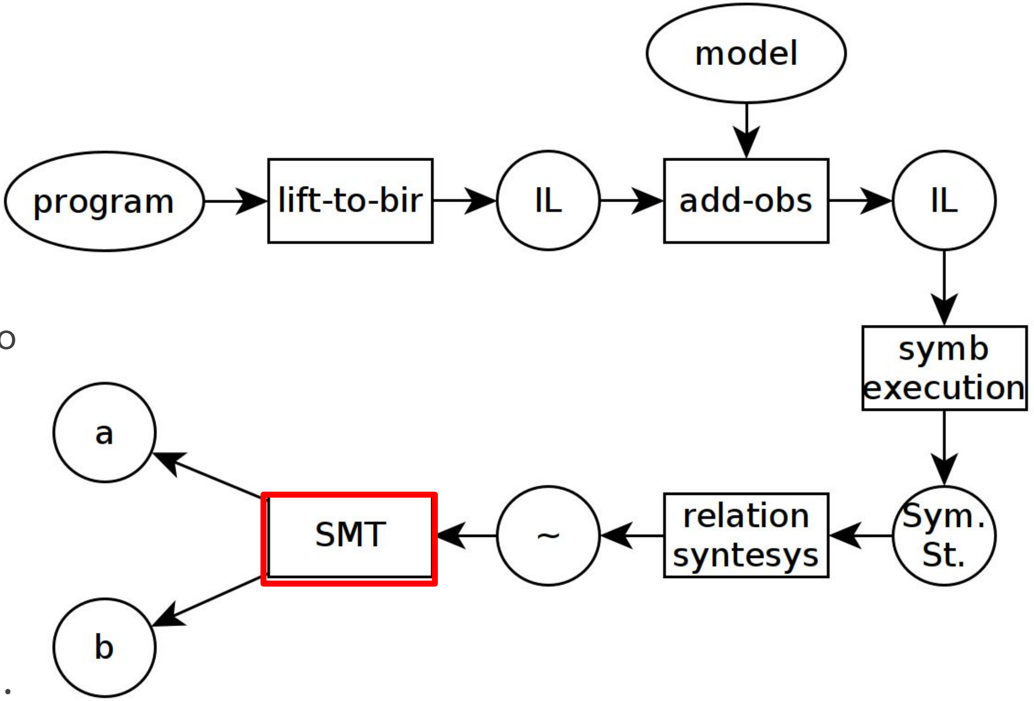
...

(a.x1 != a.x2) /\ (b.x1 != b.x2)
==>
[] == []

SMT Solver

- Z3 results in assignments to relevant variables for the two states

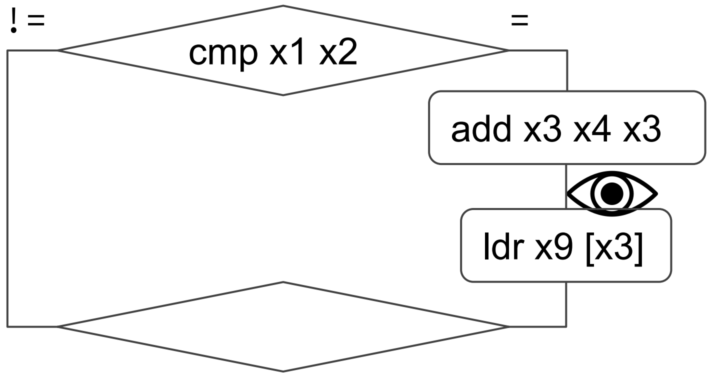
- Some additional constraints enforced by the lifter
  - E.g. only a part of memory is accessible for tests
  - Result in BIR assertions (e.g. for each memory load/store)
  - Stricter path conditions that lead to only executable test cases



How about spectre?

Raspberry Pi3 is claimed to be immune.

1) Program template
2) We must train the branch predictor
3) It is really hard to find counterexamples



(a.x1 == a.x2) /\ (b.x1 == b.x2) ⇒ (a.x3+a.x4=b.x3+b.x4)

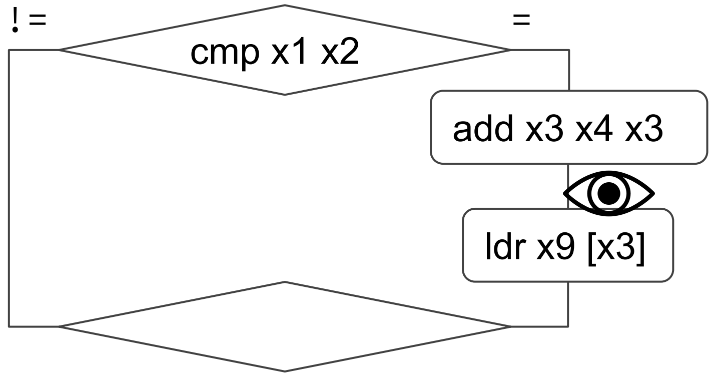
…

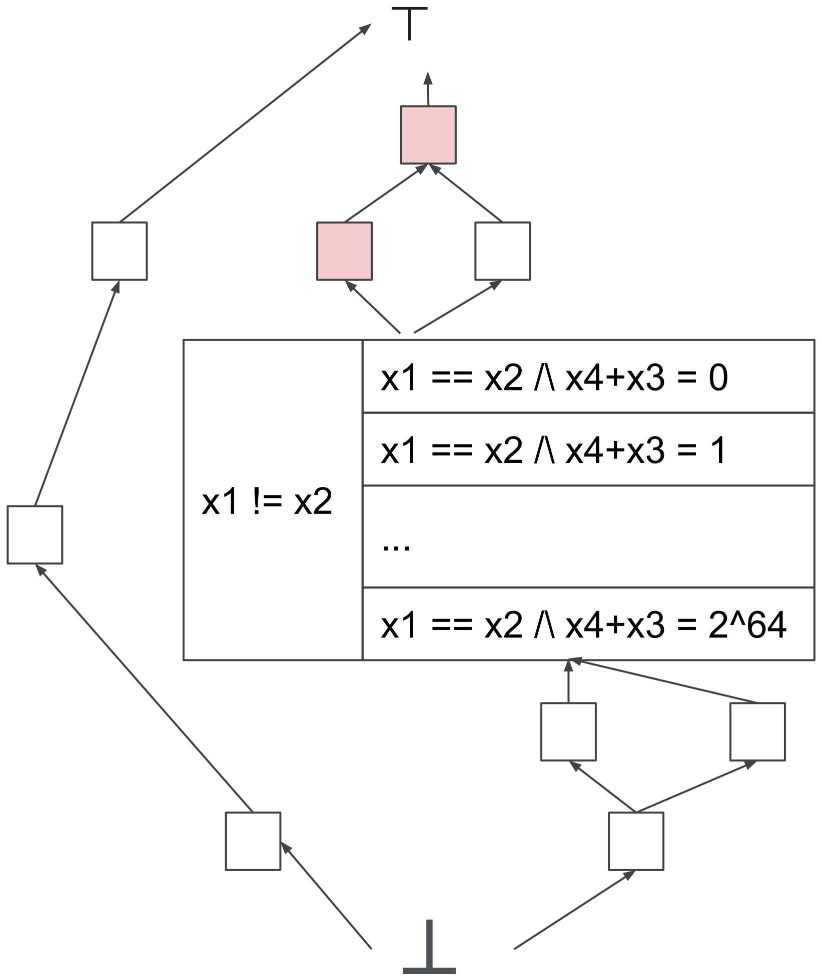
$\wedge$

…

(a.x1 != a.x2) /\ (b.x1 != b.x2) ⇒ true

Model Lattice





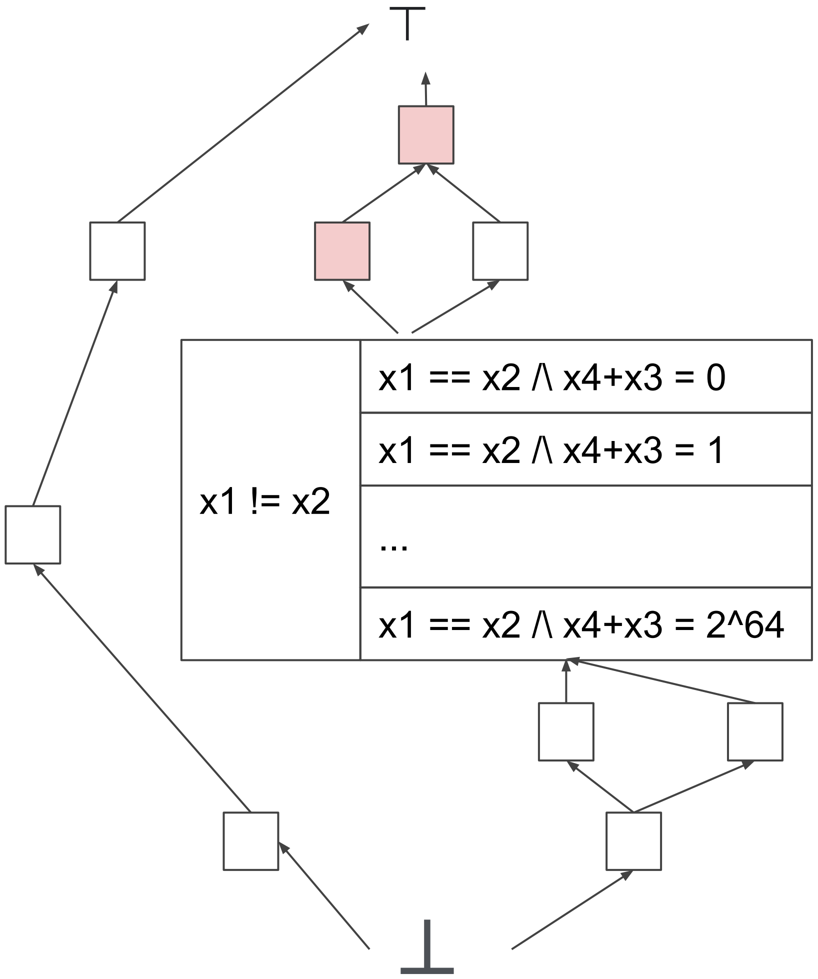
Model Lattice Driven Test

$\sim_1$ is model undel validation

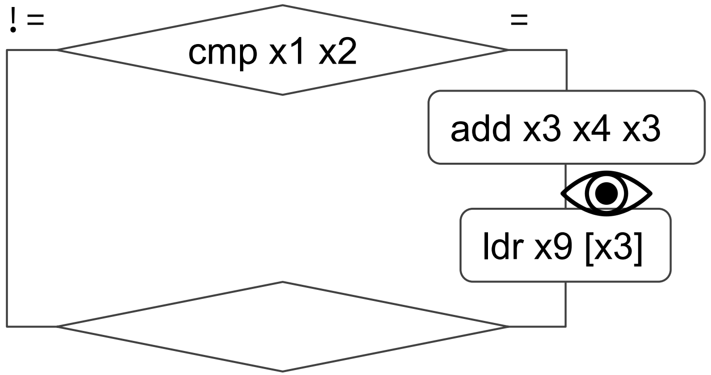
$\sim_2$ is refined model

Generate a,b such that

a $\sim_1$ b and not (a $\sim_2$ b)



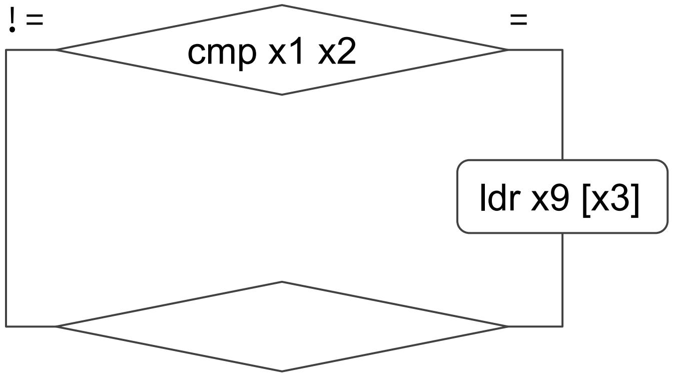
BIR



```
0 : CJMP x1=x2 4 12

4: x3 = x4 + x3; JMP 8

8 : x9 = MEM[x3]; JMP 12

12 : HALT
```

```
0 : CJMP x1=x2 4’ 8’

4’: JMP 4
4:  x3 = x4 + x3; JMP 8

8 : Obs_1(x3); x9 = MEM[x3]; JMP 12

12’ : x3’ = x4’ + x3’;
     Obs_1(x3’); x9’ = MEM[x3’];
     JMP 12


12: HALT
```

Refined Observations



```
0 : CJMP x1=x2 4 8

4 : Obs(x3);
     x9 = MEM[x3]; JMP 8

8: HALT
```

```
0 : CJMP x1=x2 4’ 8’

4’: JMP 4
4 : Obs_1(x3);
     x9 = MEM[x3]; JMP 8

8’: Obs_2(x3);
     x9’ = MEM[x3]; JMP 8

8: HALT
```

(a.x1 == a.x2) /\ (b.x1 == b.x2) ⇒
    (a.x3+a.x4 = b.x3+b.x4) /\
    (a.x3+a.x4 != b.x3+b.x4)

…

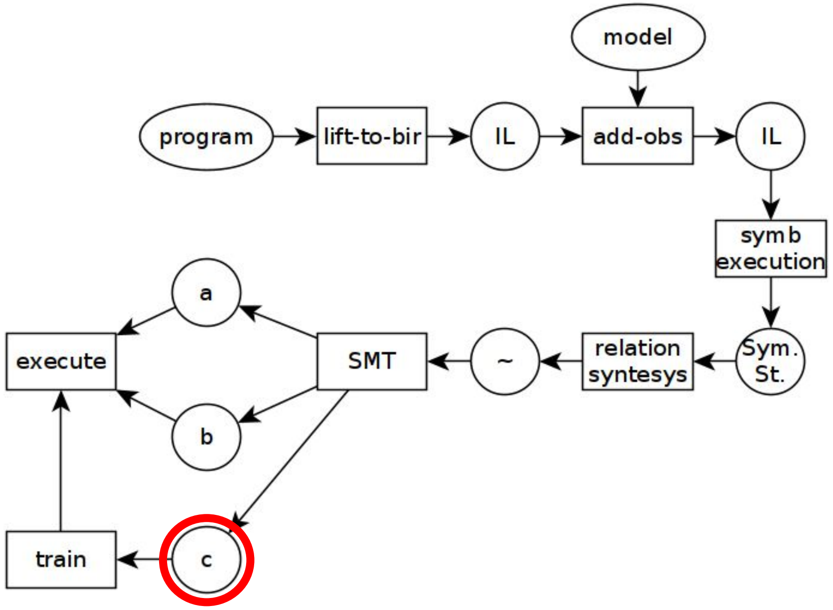
$\wedge$

…

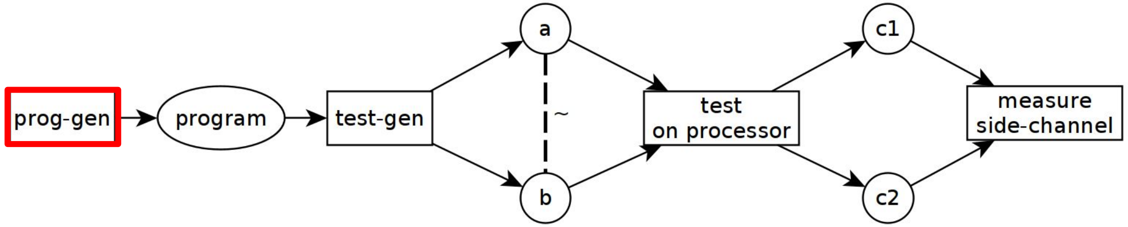
(a.x1 != a.x2) /\ (b.x1 != b.x2) ⇒
    true /\
    (a.x3+a.x4 != b.x3+b.x4)

Train branch predictor

- interesting cases:
    - a and b follows the same execution path
- use SMT to generate a state c that follows the wrong path
    - E.g. if (a.x1 != a.x2) /\ (b.x1 != b.x2) then (c.x1 == c.x2)
    - Use c to train the branch-predictor

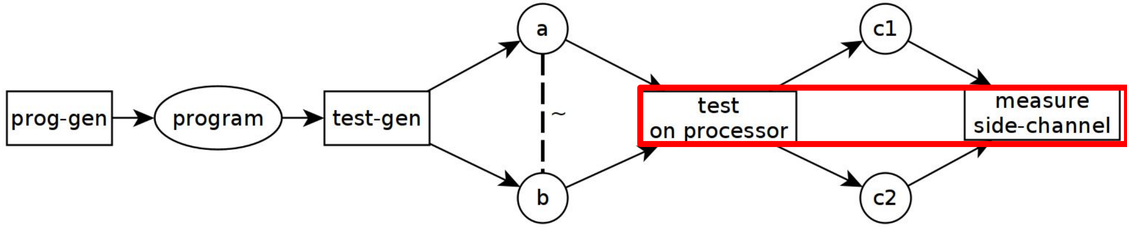


Program generation



- Generators can be composed
- Generators are randomly instantiated

Test and measure



- ARMv8 (Raspberry Pi 3)
- Execution coordinated by a server
- Bootloader
  - load program
  - sets-up environment (e.g. page tables)
  - sets-up state a
  - exec from state a
  - inspect cache
  - clean state
  - repeat for state b
- Implemented in TrustZone
  - execution bare to metal => no noise
  - instructions for cache inspection

How about spectre?

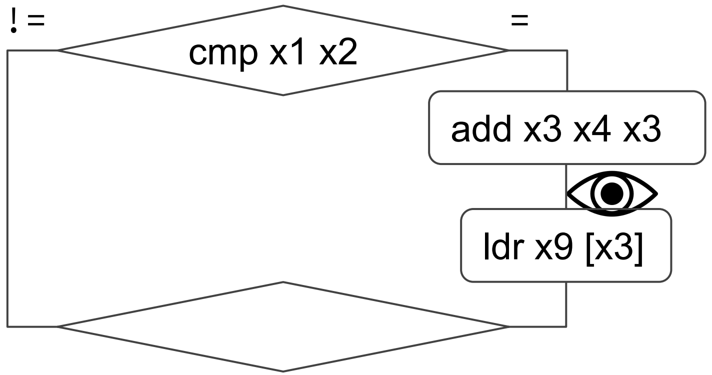
Raspberry Pi3 is claimed to be immune.

NO!

It seems to be immune to the original Spectre, but it is affected by other speculative leakage (i.e. the first load in the mispredicted branch can leak data)

16000 tests, 600 programs

- Without refinement 2 counterexamples
- With refinement 3971 counterexamples



Concluding remarks

Summary:

- Existing observational model often overlook some information flows
- Without sound models we cannot reason about SW security
- Observation equivalence and Model refinement provide a good strategy to drive tests

New challenges:

- Automatic model repair/improvement
- Hardware coverage