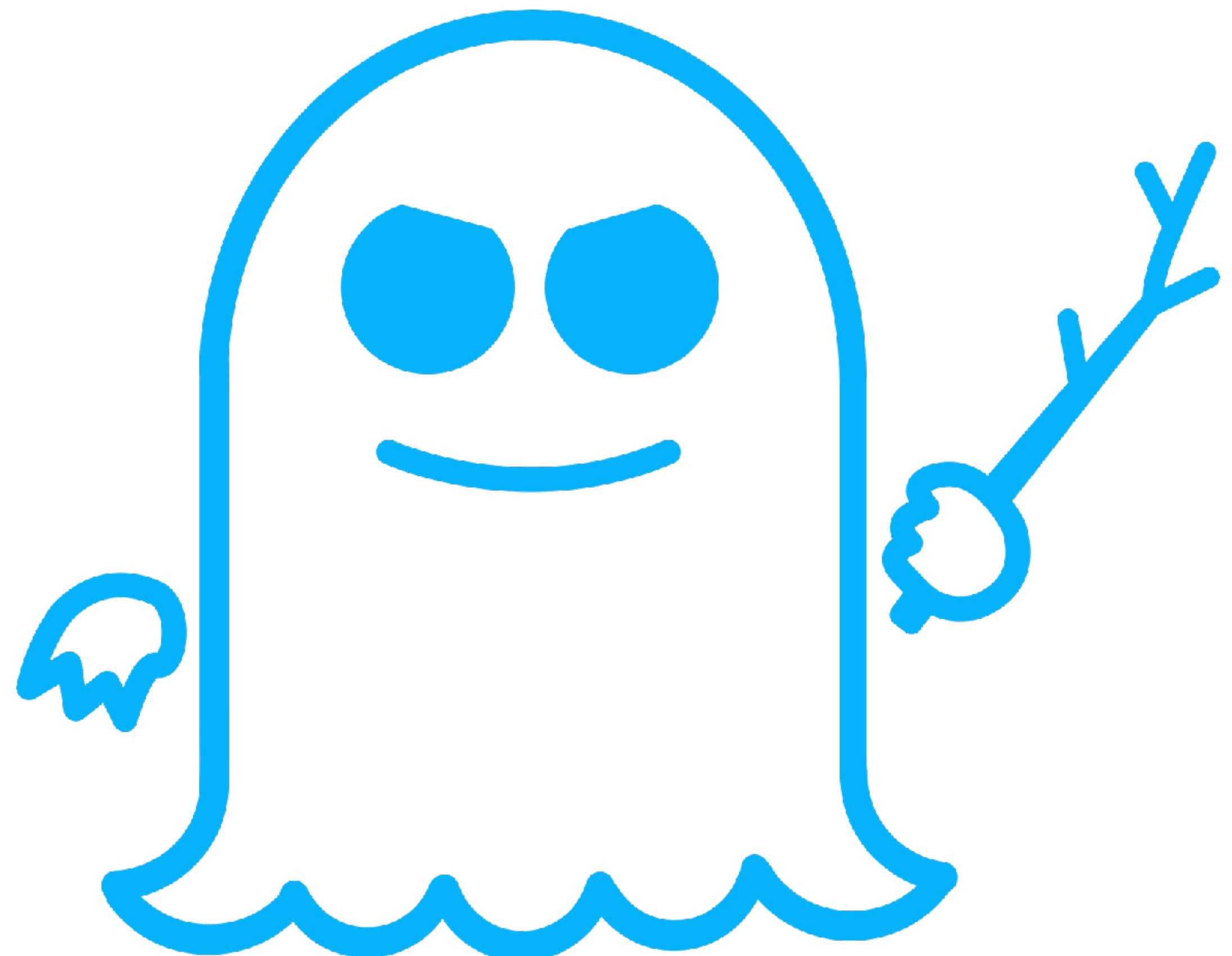


# Spectector: Principled detection of speculative information flows

Marco Guarnieri  
IMDEA Software Institute

*Joint work with*

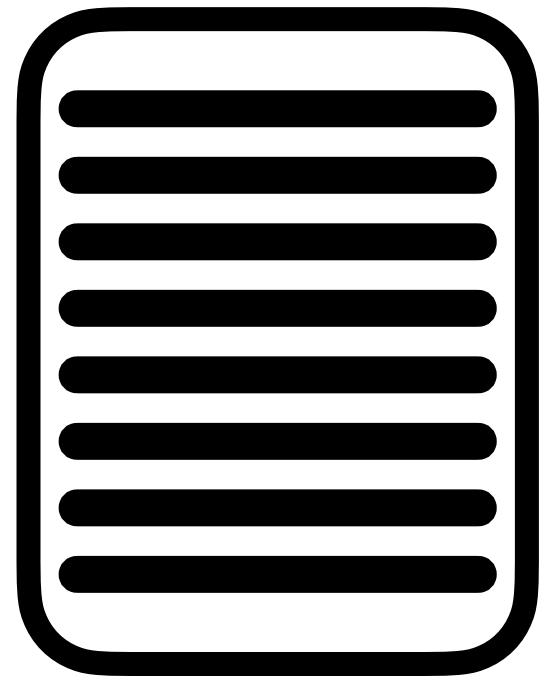
José F. Morales, Andrés Sánchez @ IMDEA Software Institute  
Boris Köpf @ Microsoft Research  
Jan Reineke @ Saarland University



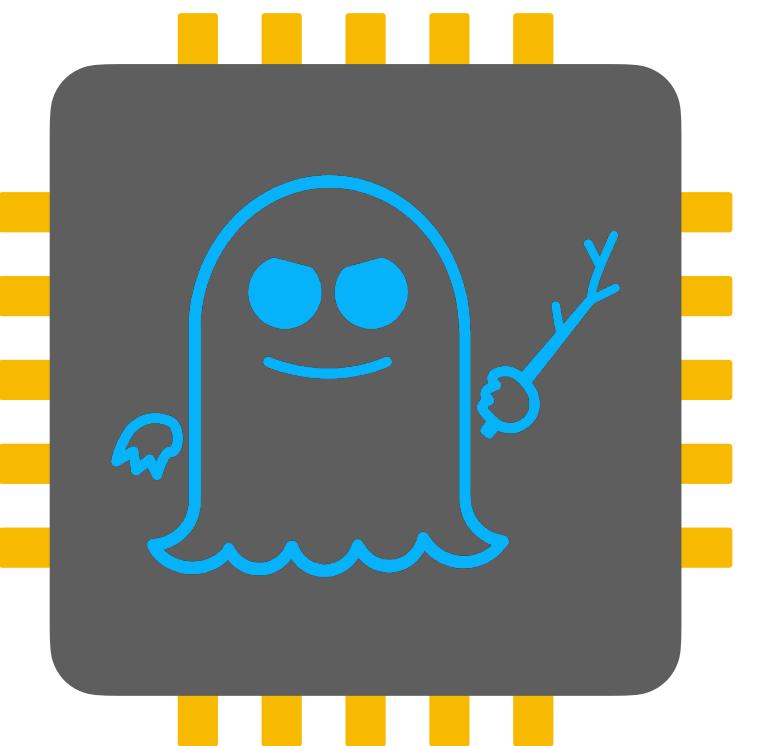
# SPECTRE

Exploits ***speculative execution***

Almost ***all*** modern ***CPUs*** are ***affected***



+

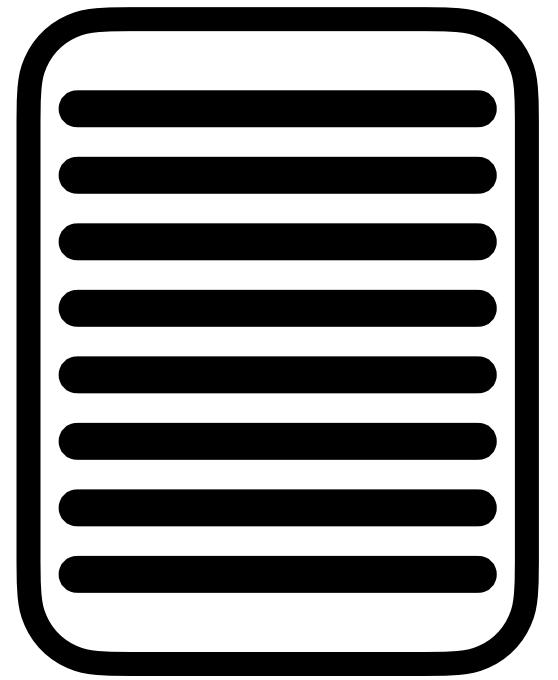


= Secure?

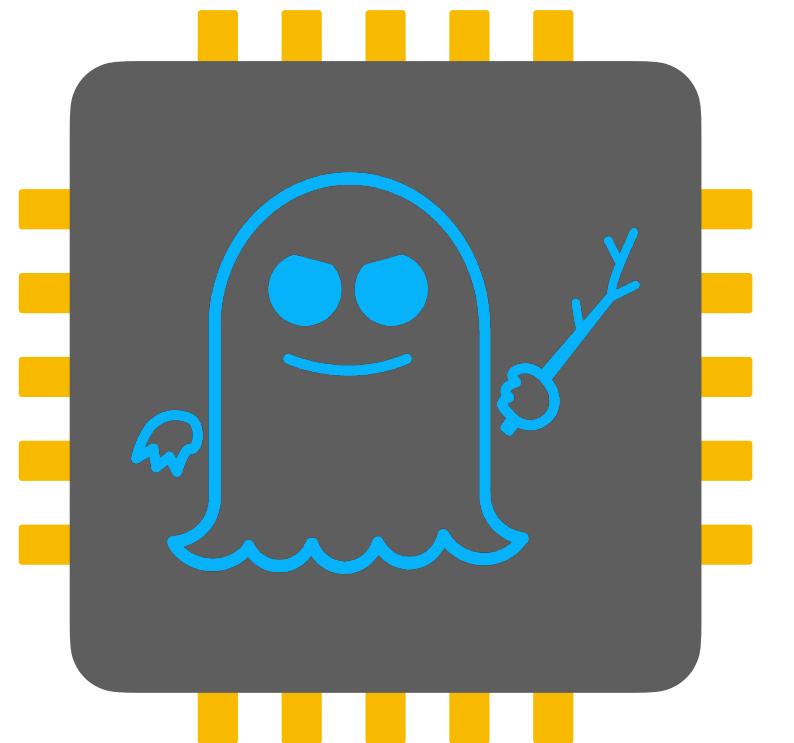


Program

CPU with ***speculative  
execution***



+



= Secure?



Program

CPU with ***speculative execution***

## In this talk..

1. ***Semantic notion*** of ***security*** against ***speculative execution attacks***

2. Analysis to ***detect vulnerability*** or ***prove security***

# Speculative execution attacks 101

# Speculative execution + branch prediction

Size of array **A**

```
if (x < A_size)
    y = B[A[x]]
```

# Speculative execution + branch prediction

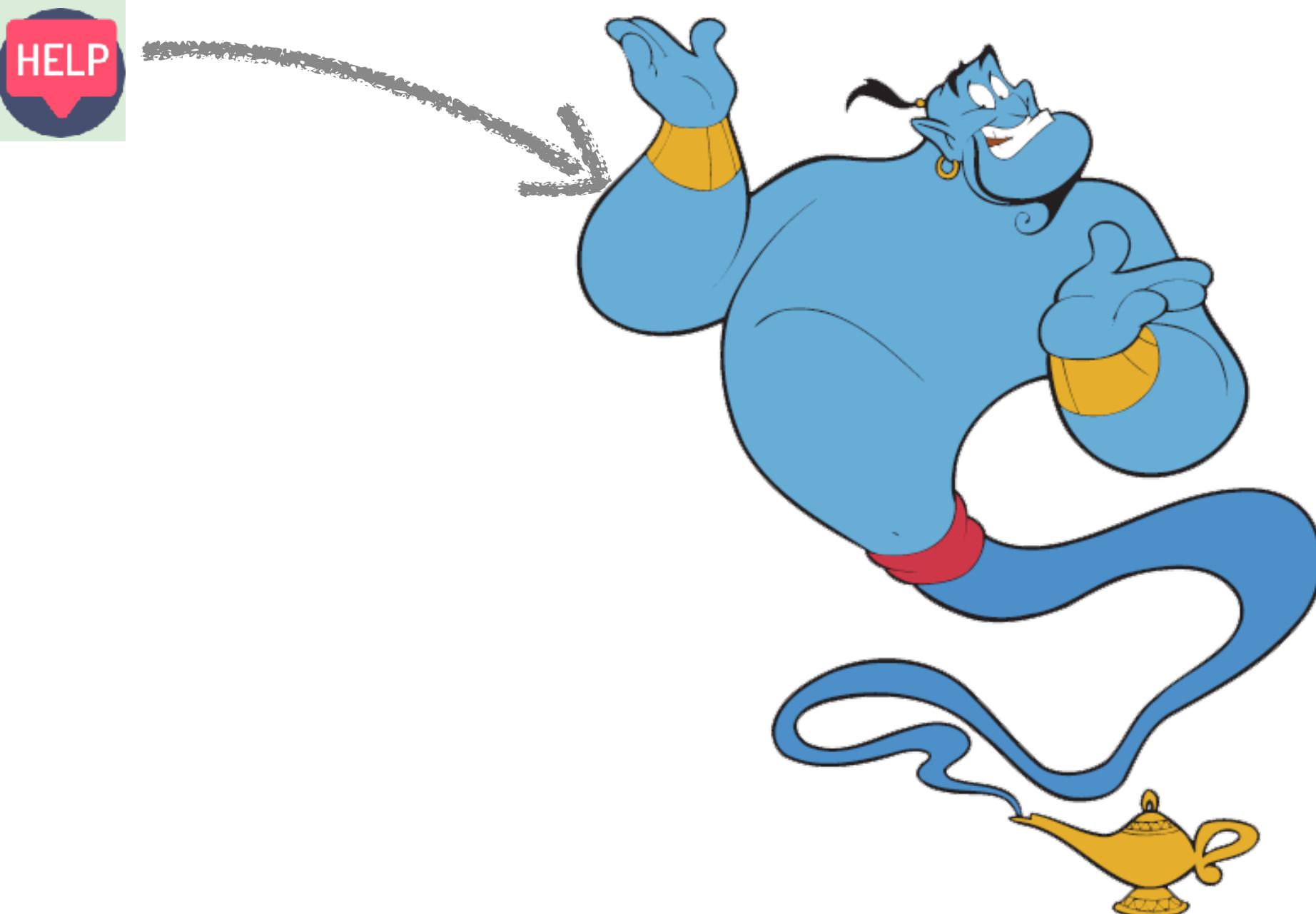
Size of array **A**

```
if (x < A_size)
    y = B[A[x]]
```

# Speculative execution + branch prediction

Size of array **A**

```
if (x < A_size)  
    y = B[A[x]]
```



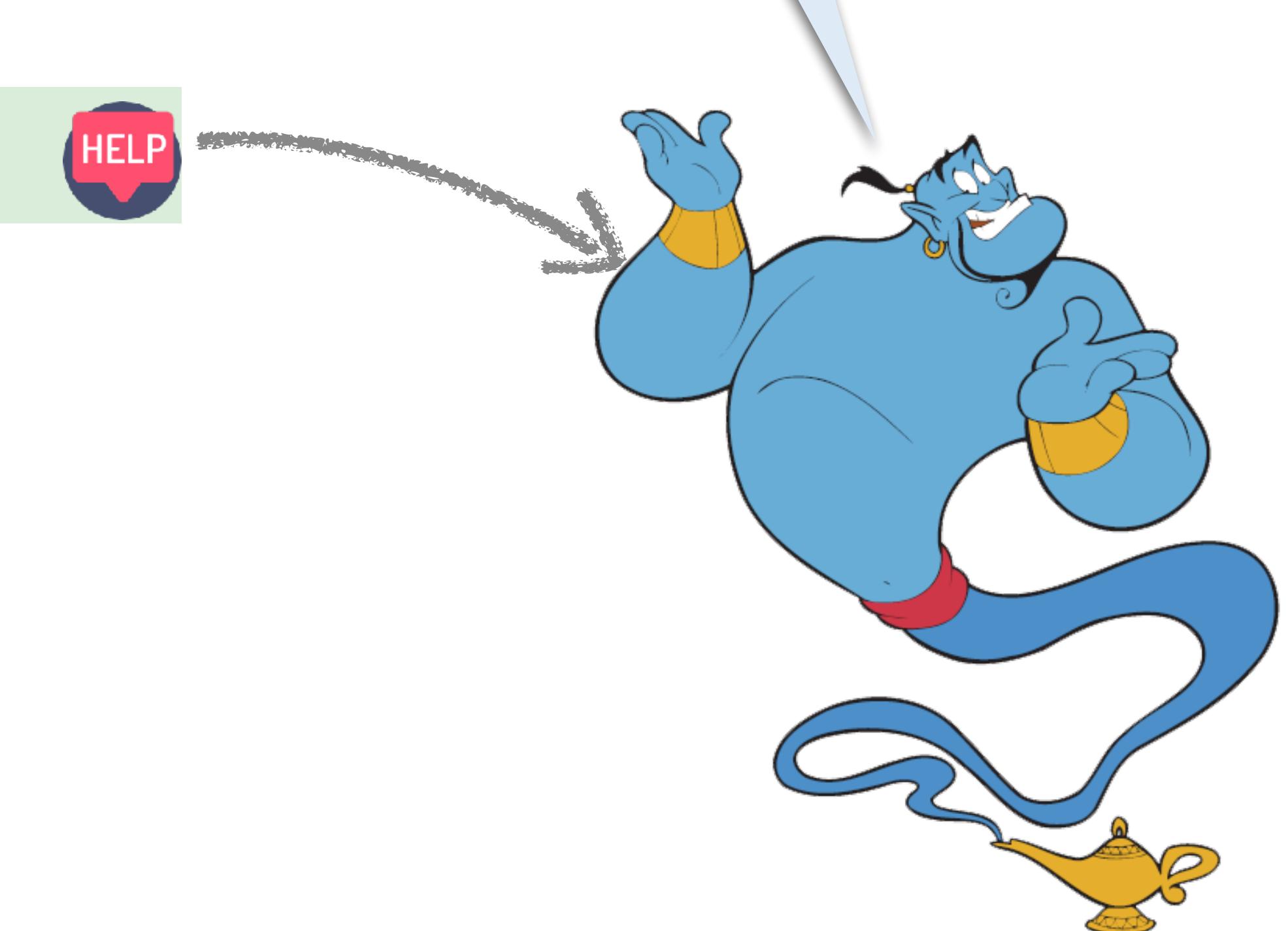
Branch predictor

# Speculative execution + branch prediction

Prediction based on **branch history** & **program structure**

Size of array **A**

```
if (x < A_size) HELP  
    y = B[A[x]]
```



Branch predictor

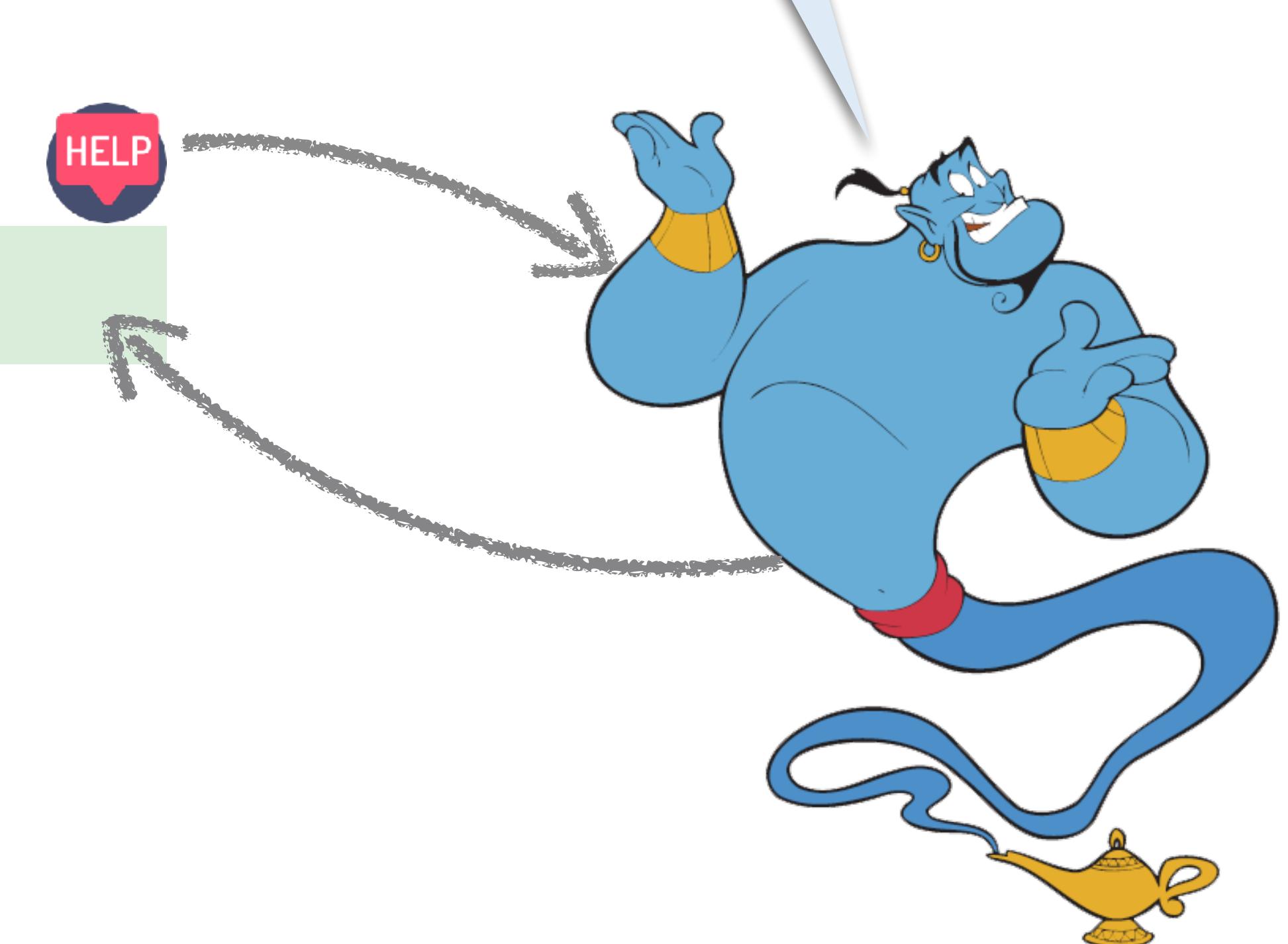
# Speculative execution + branch prediction

Prediction based on **branch history** & **program structure**

Size of array **A**

```
if (x < A_size)  
    y = B[A[x]]
```

HELP



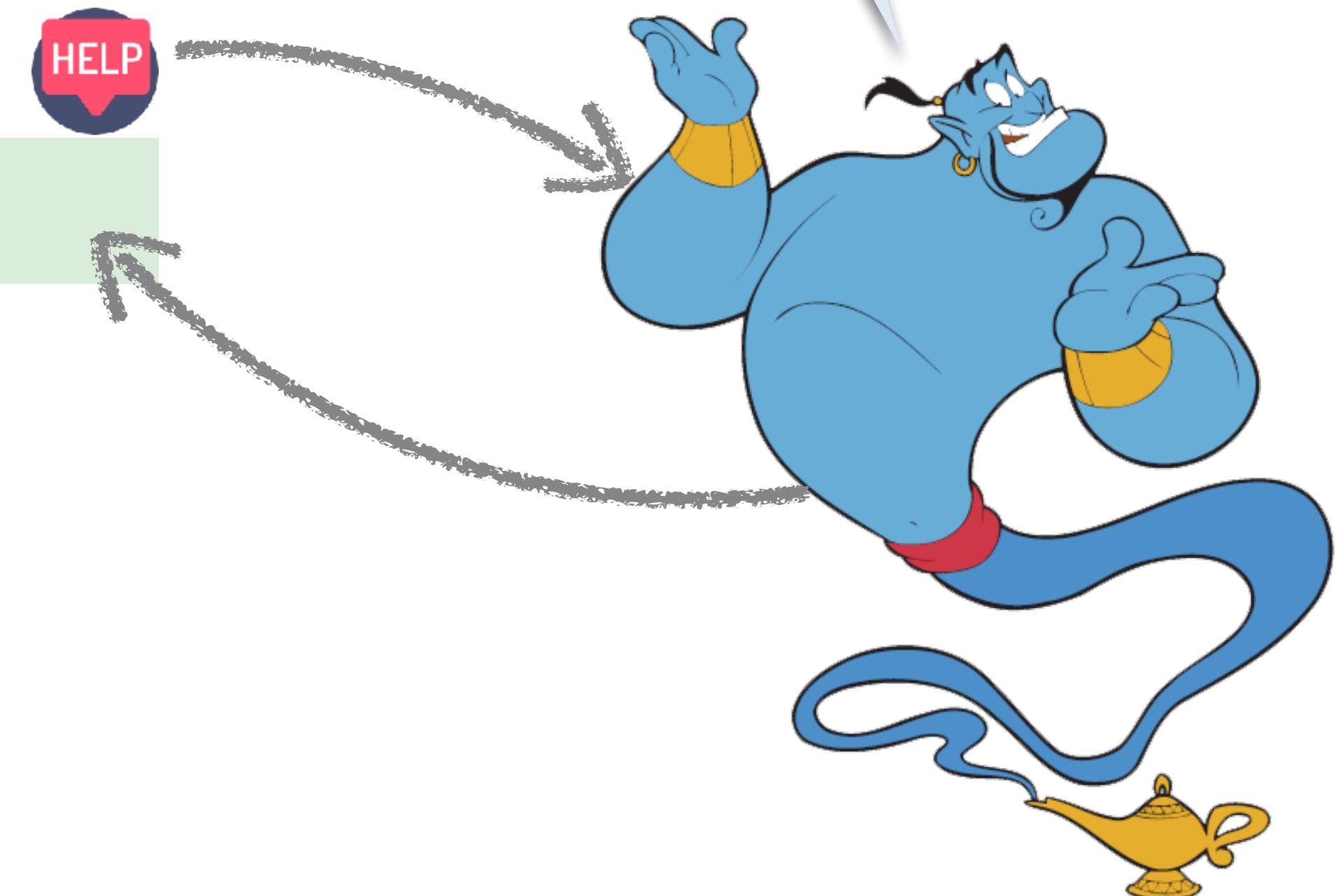
Branch predictor

# Speculative execution + branch prediction

Prediction based on **branch history** & **program structure**

Size of array **A**

```
if (x < A_size)  
    y = B[A[x]]
```



Wrong prediction? **Rollback changes!**



Architectural (ISA) state



Microarchitectural state

Branch predictor

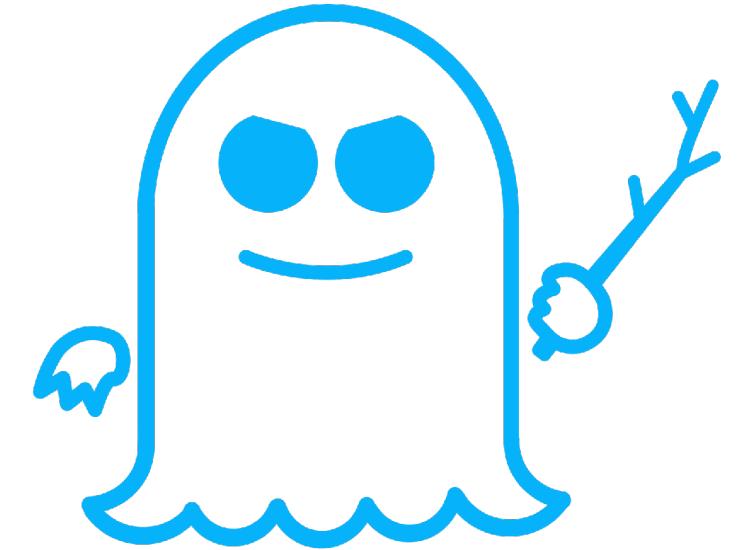
# Spectre V1

# Spectre V1

```
void f(int x)
  if (x < A_size)
    y = B[A[x]]
```



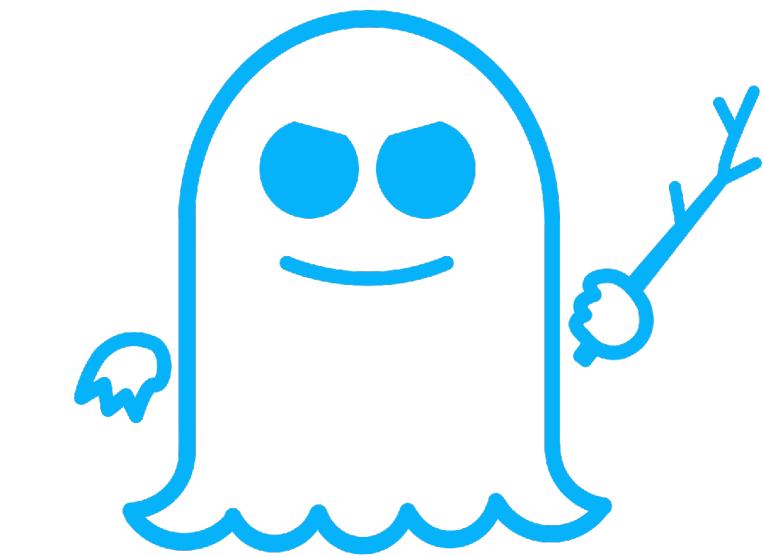
# Spectre V1



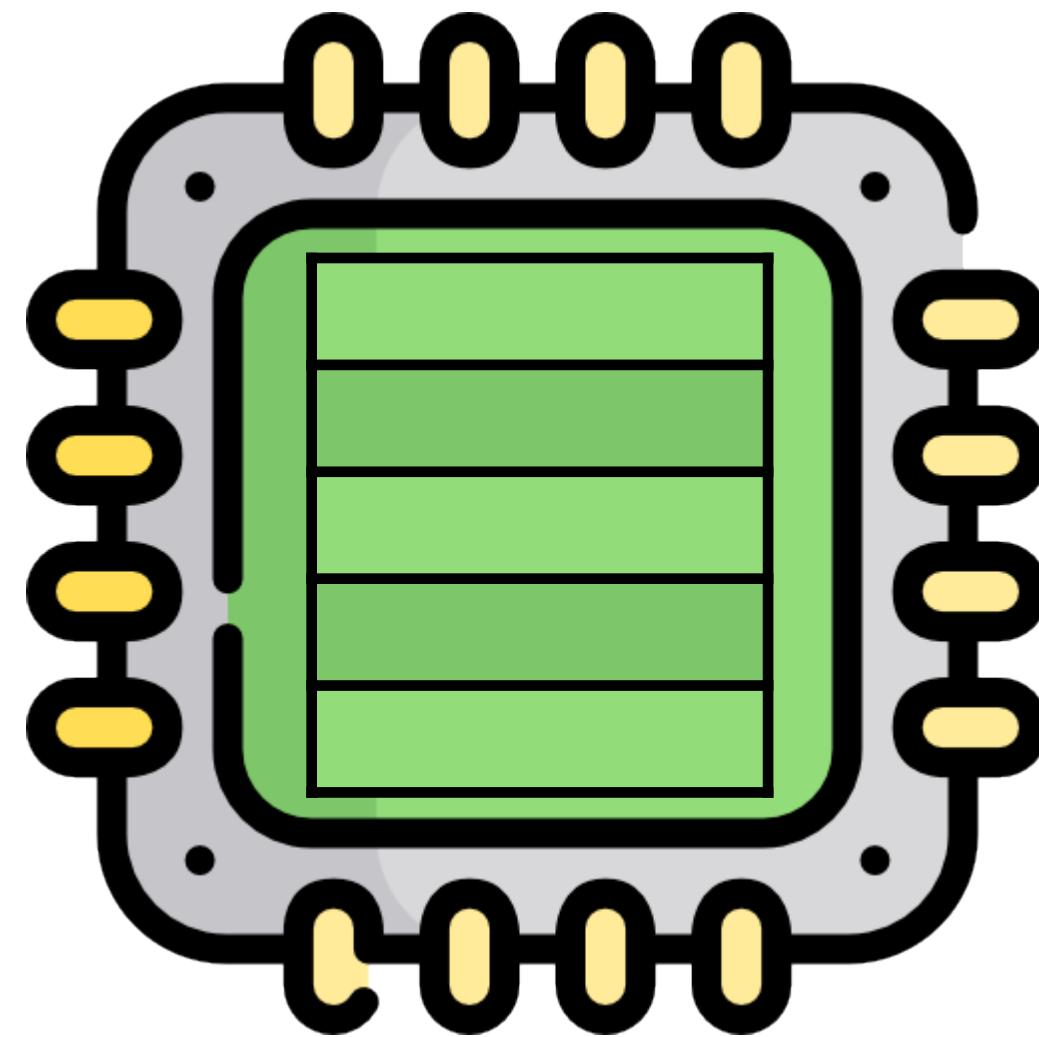
```
void f(int x)
  if (x < A_size)
    y = B[A[x]]
```



# Spectre V1

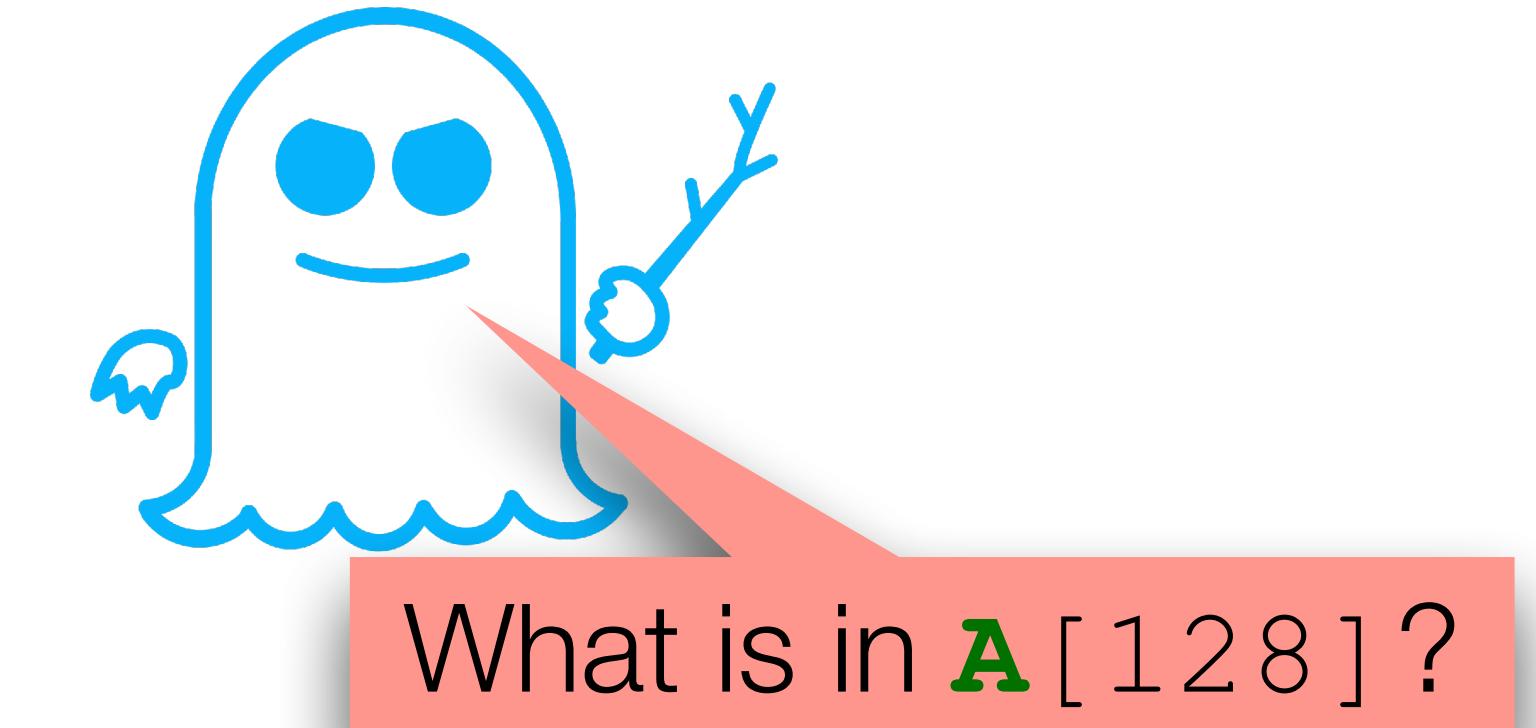


```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```

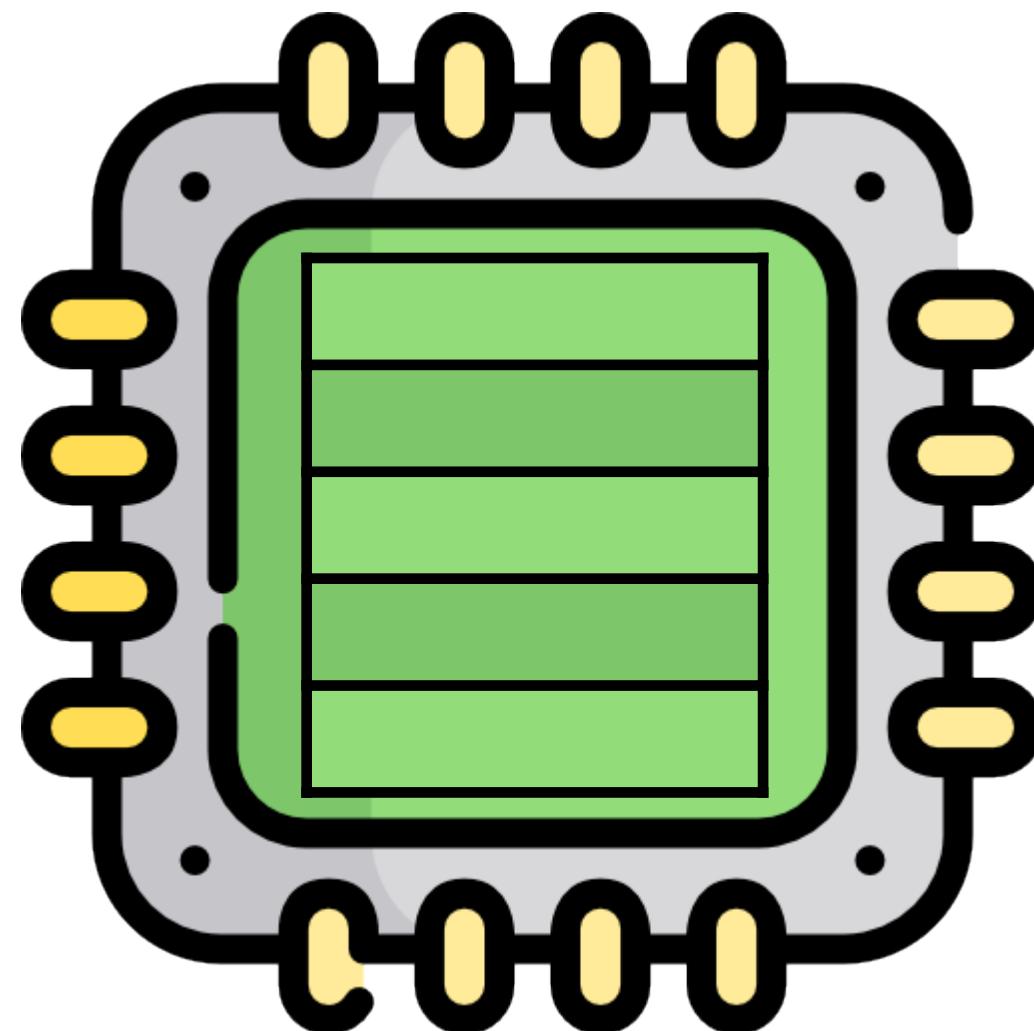


# Spectre V1

```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```

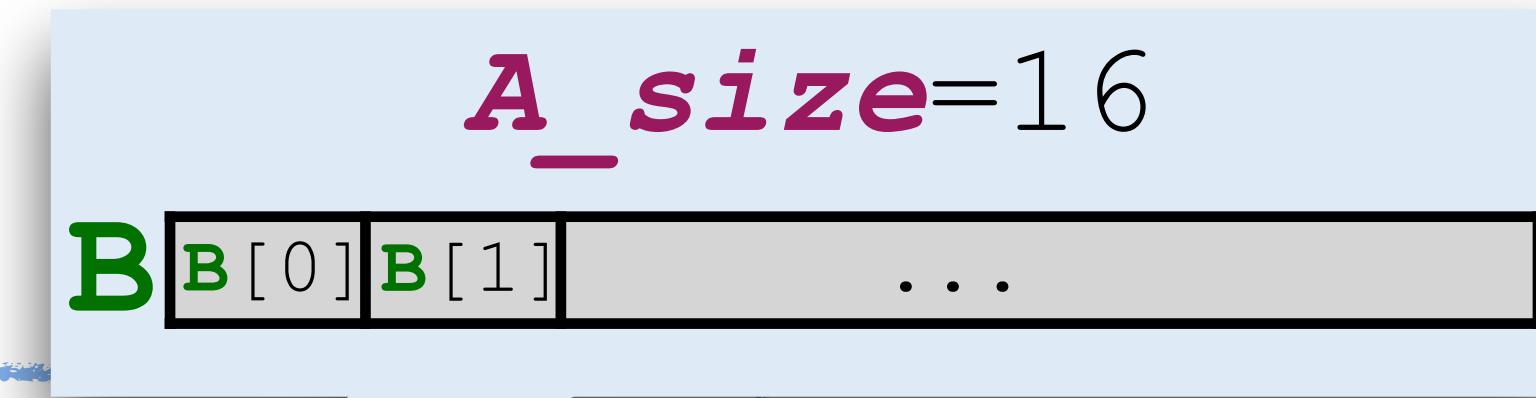


What is in **A**[128]?



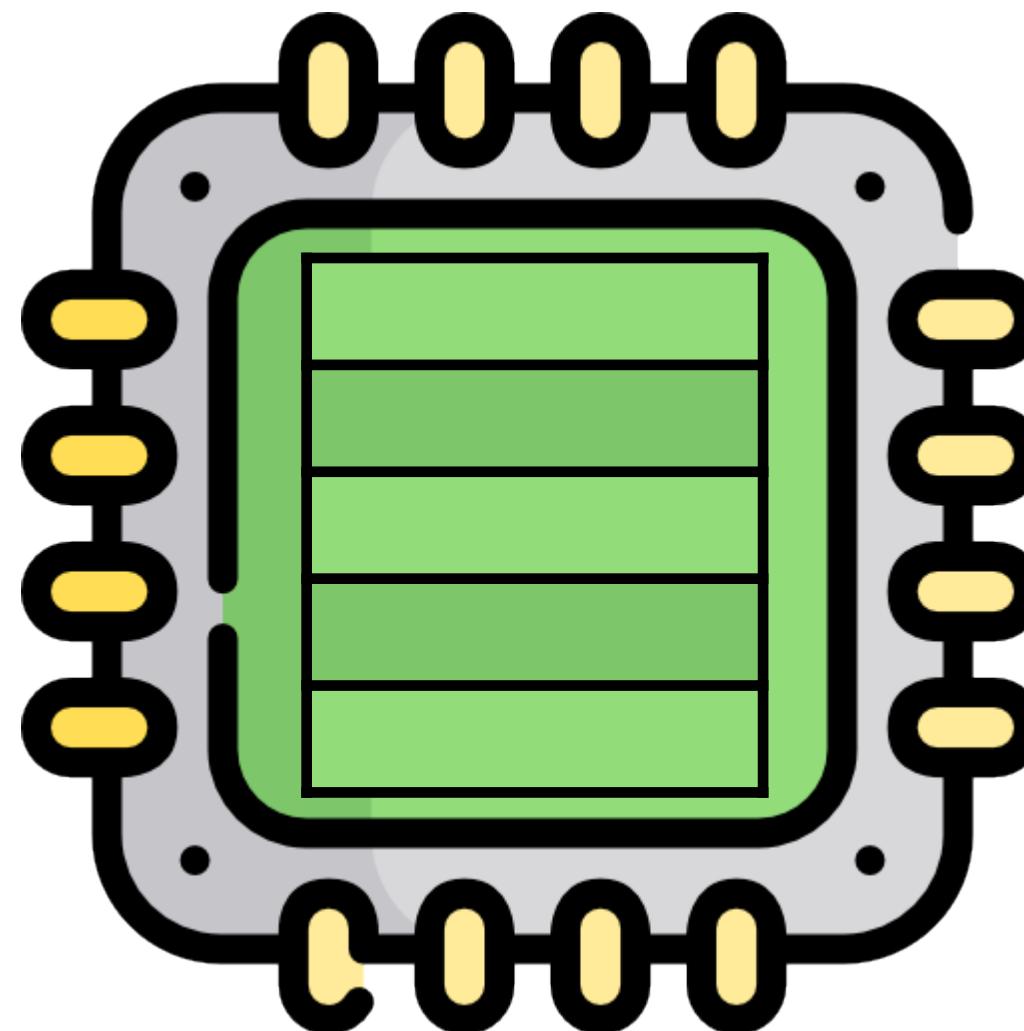
# Spectre V1

```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



What is in  $A[128]$ ?

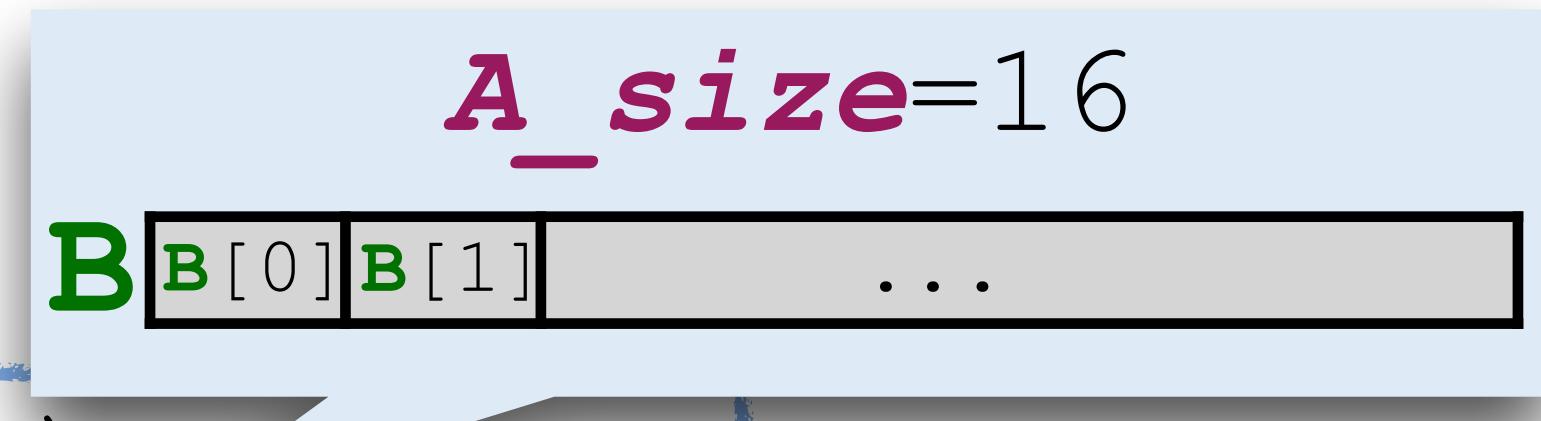
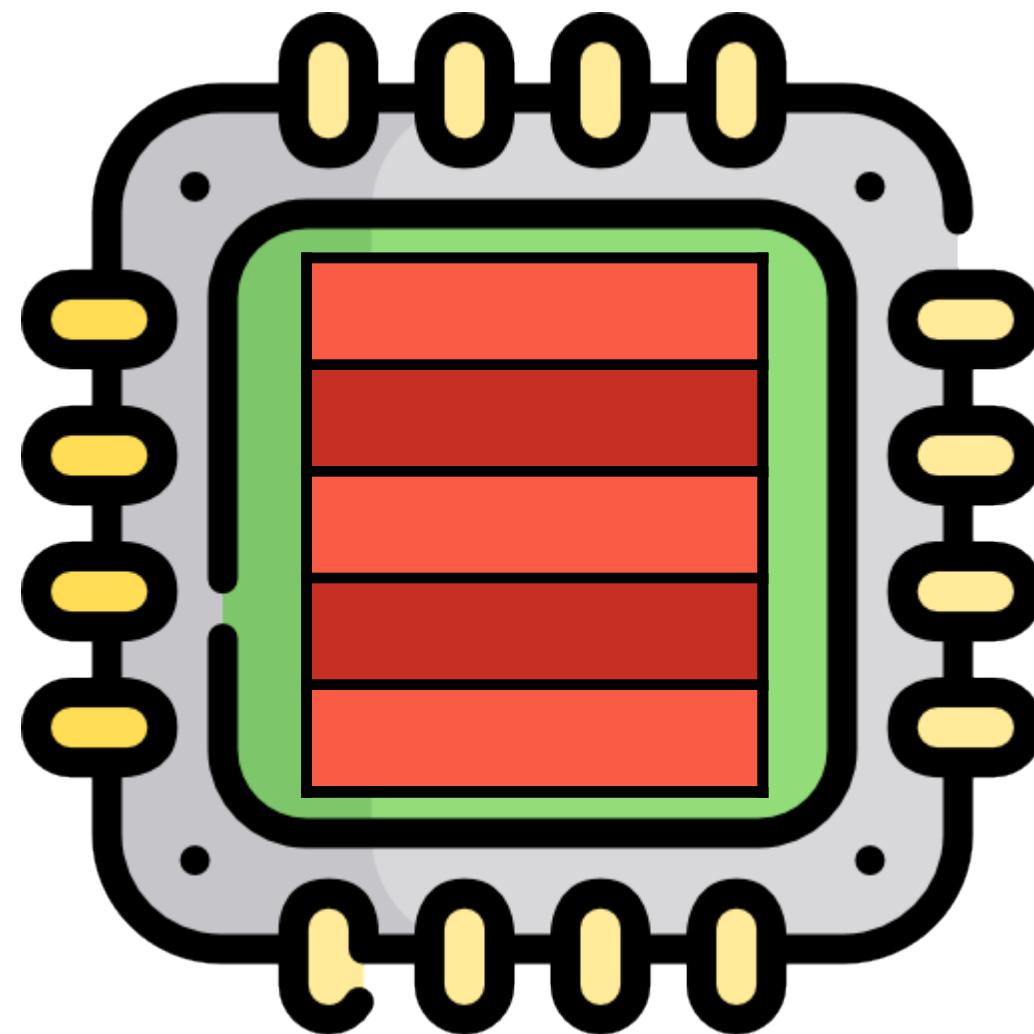
1) Train branch predictor



# Spectre V1



```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



What is in A[128]?

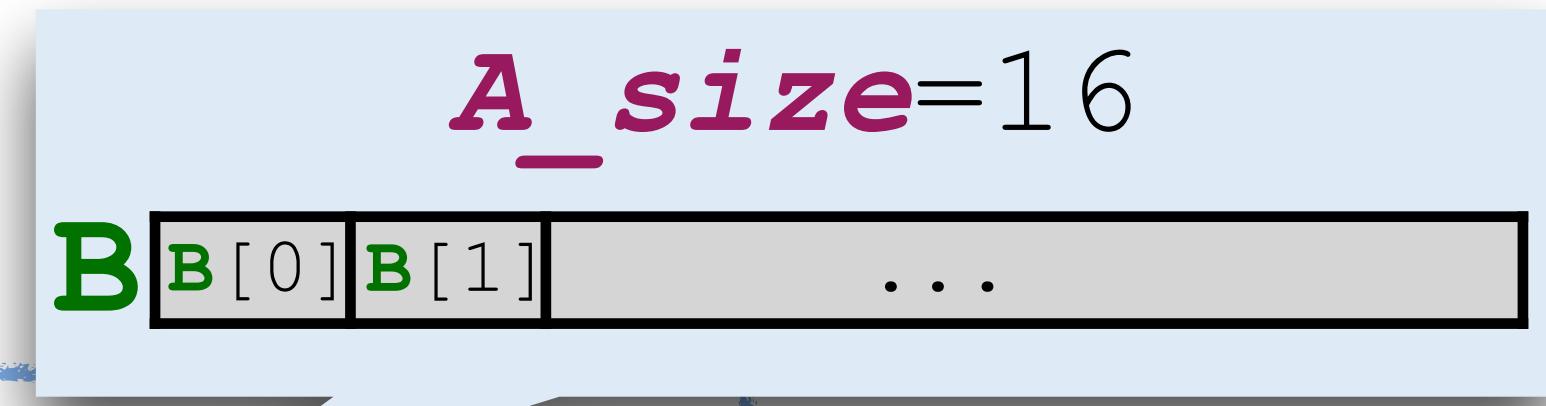
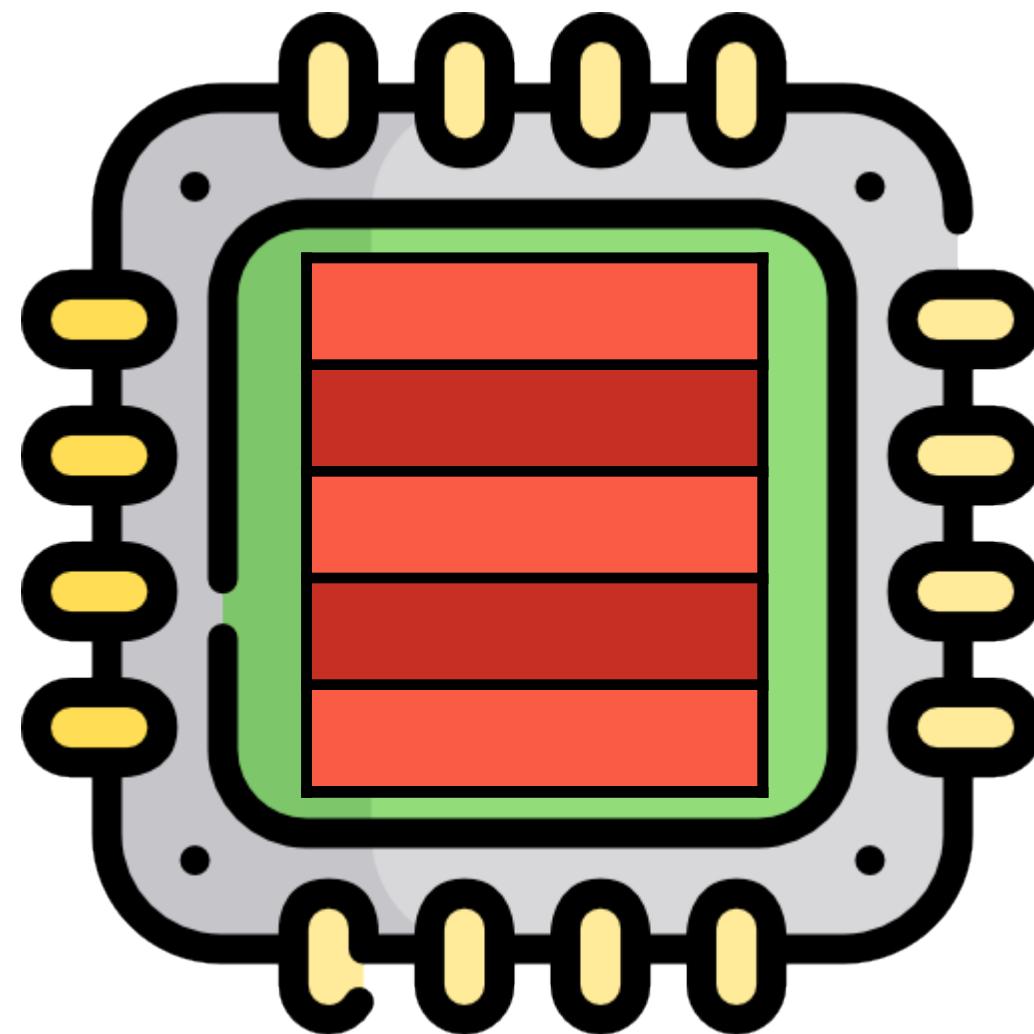
1) Train branch predictor

2) Prepare cache

# Spectre V1



```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



What is in A[128]?

1) Train branch predictor

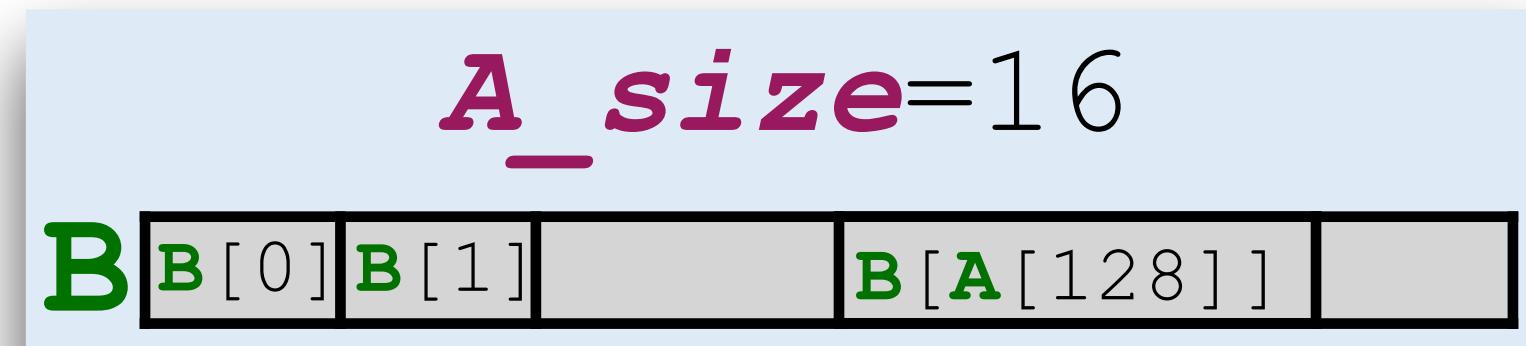
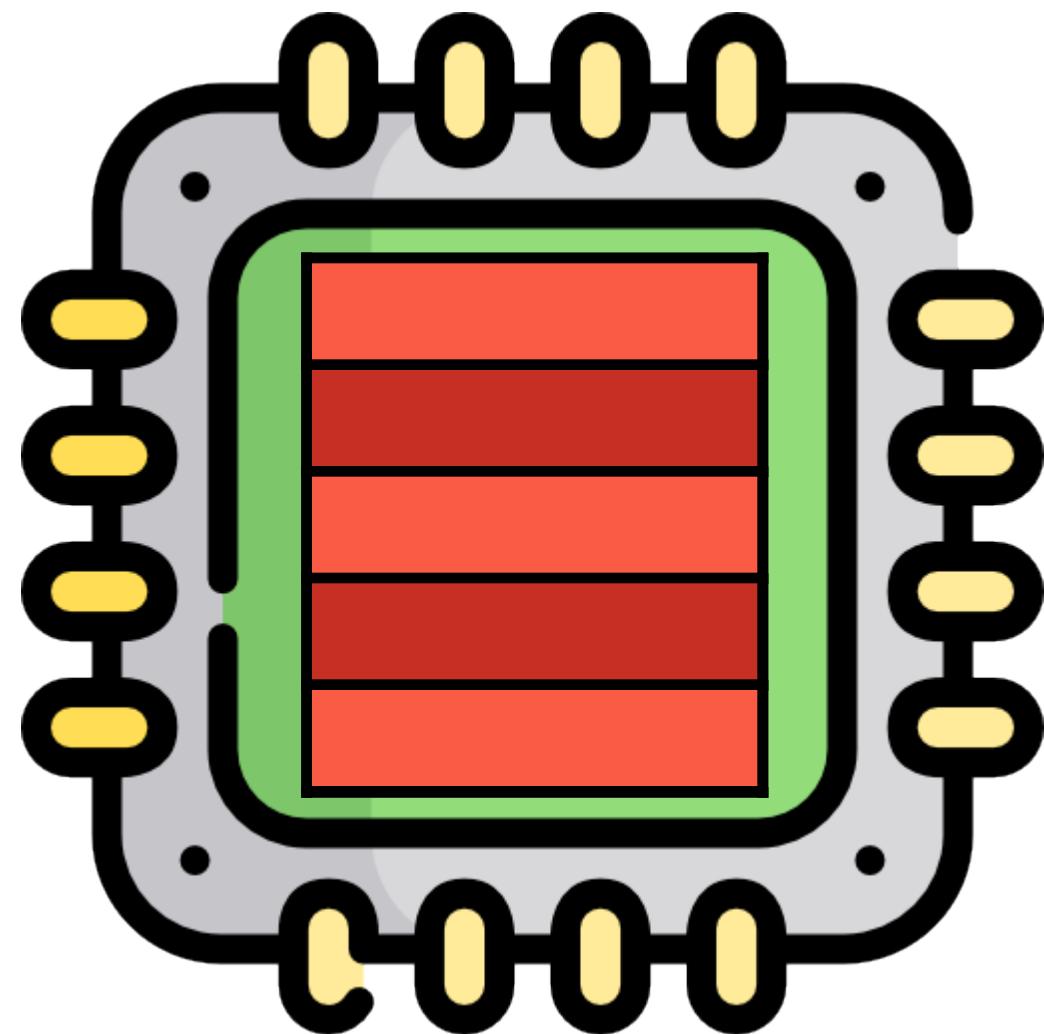
2) Prepare cache

3) Run with  $x = 128$

# Spectre V1



```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



What is in  $A[128]$ ?

1) Train branch predictor

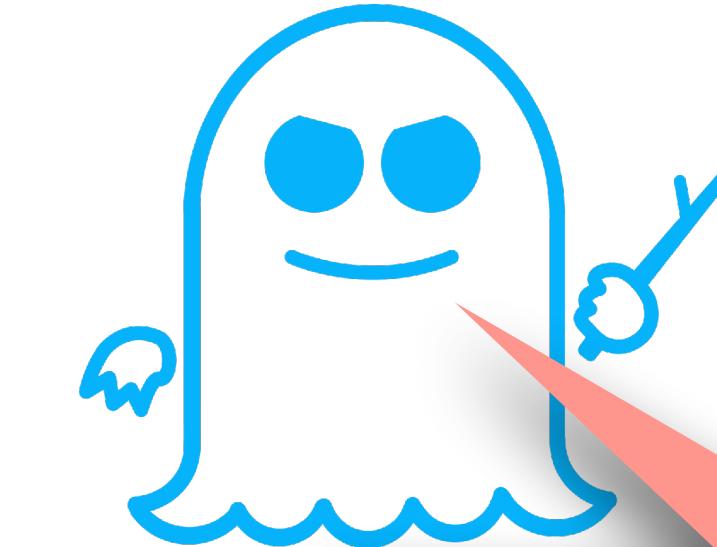
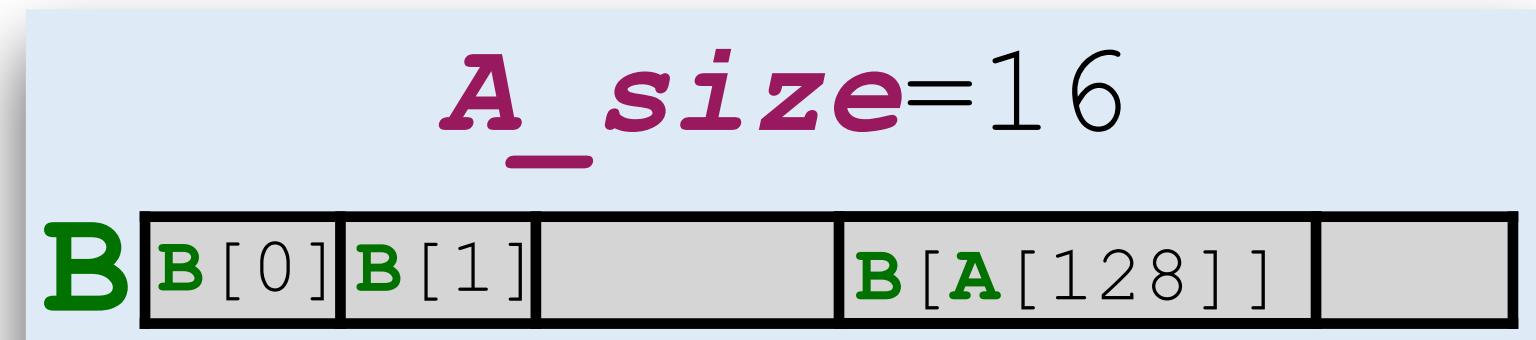
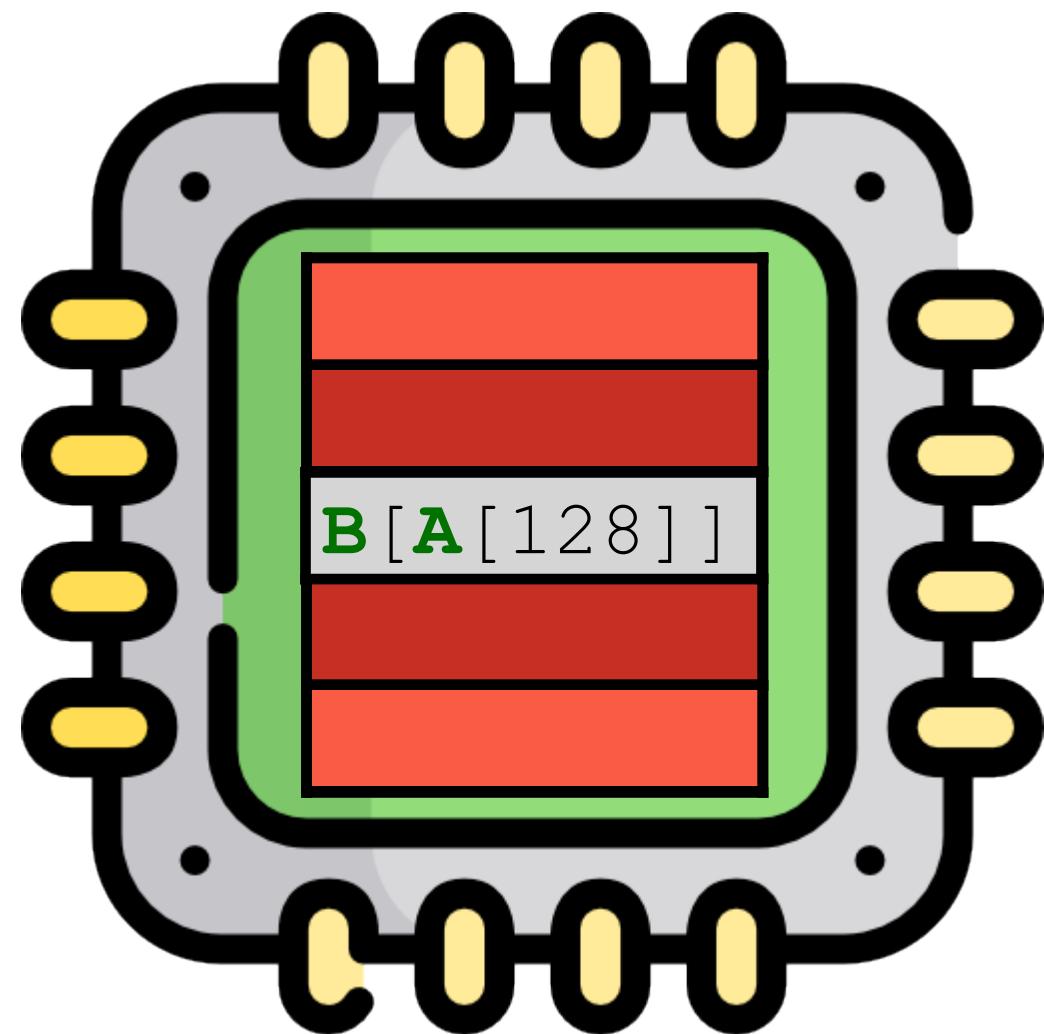
2) Prepare cache

3) Run with  $x = 128$

# Spectre V1



```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



What is in  $A[128]$ ?

1) Train branch predictor

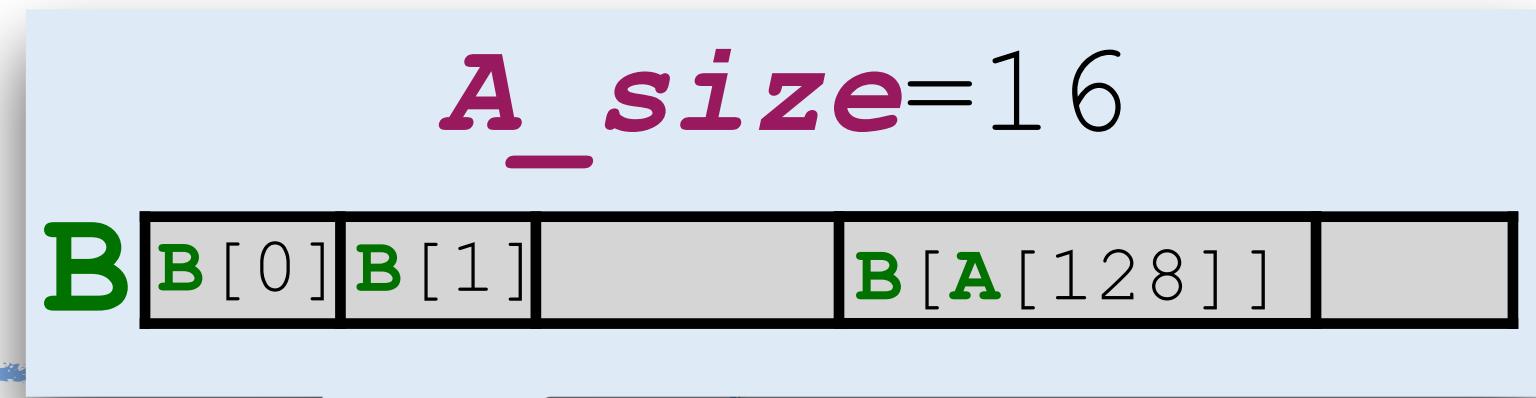
2) Prepare cache

3) Run with  $x = 128$

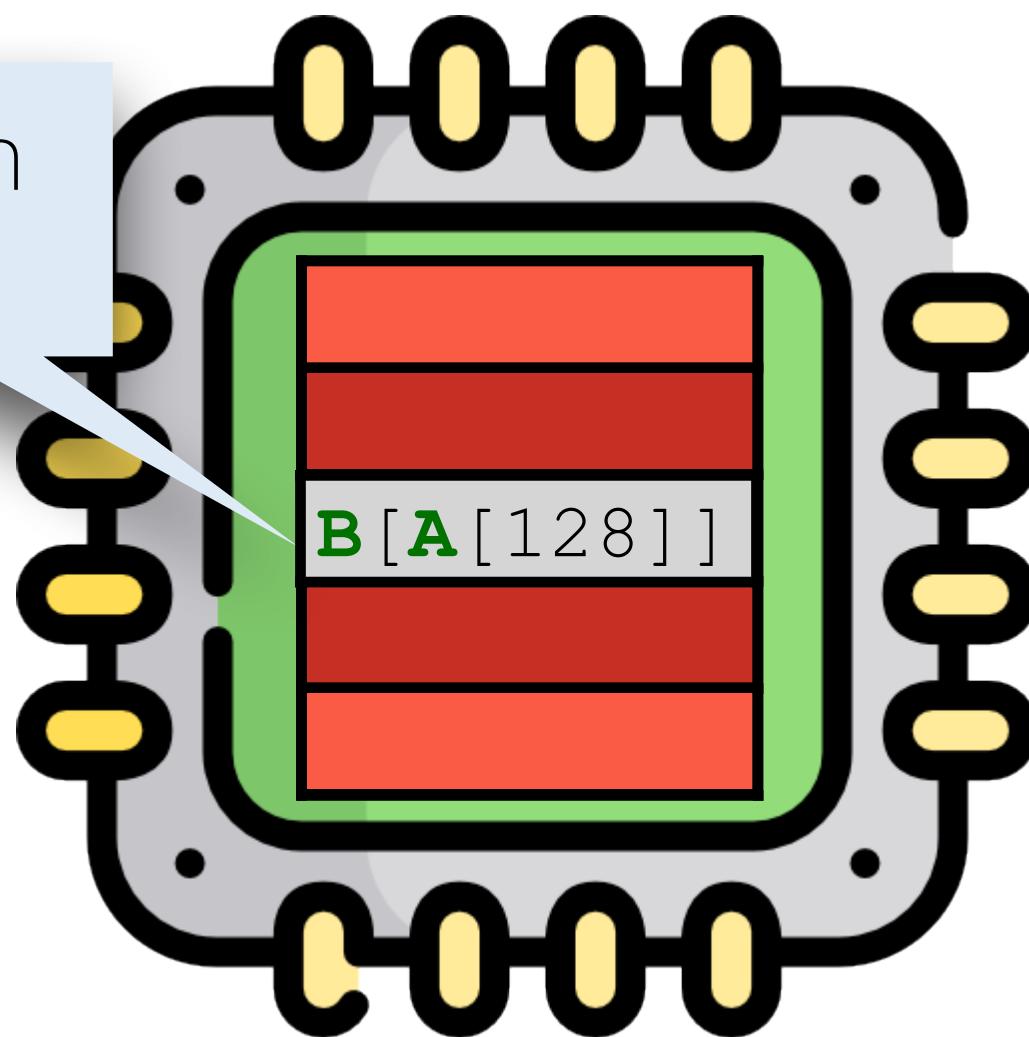
# Spectre V1



```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



What is in  $A[128]$ ?



Depends on  
 $A[128]$

1) Train branch predictor

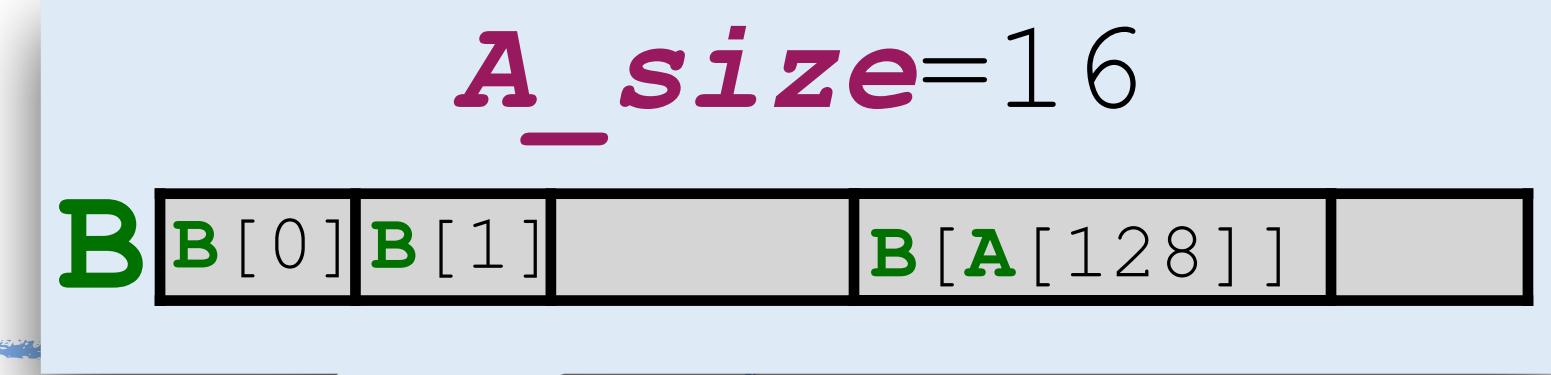
2) Prepare cache

3) Run with  $x = 128$

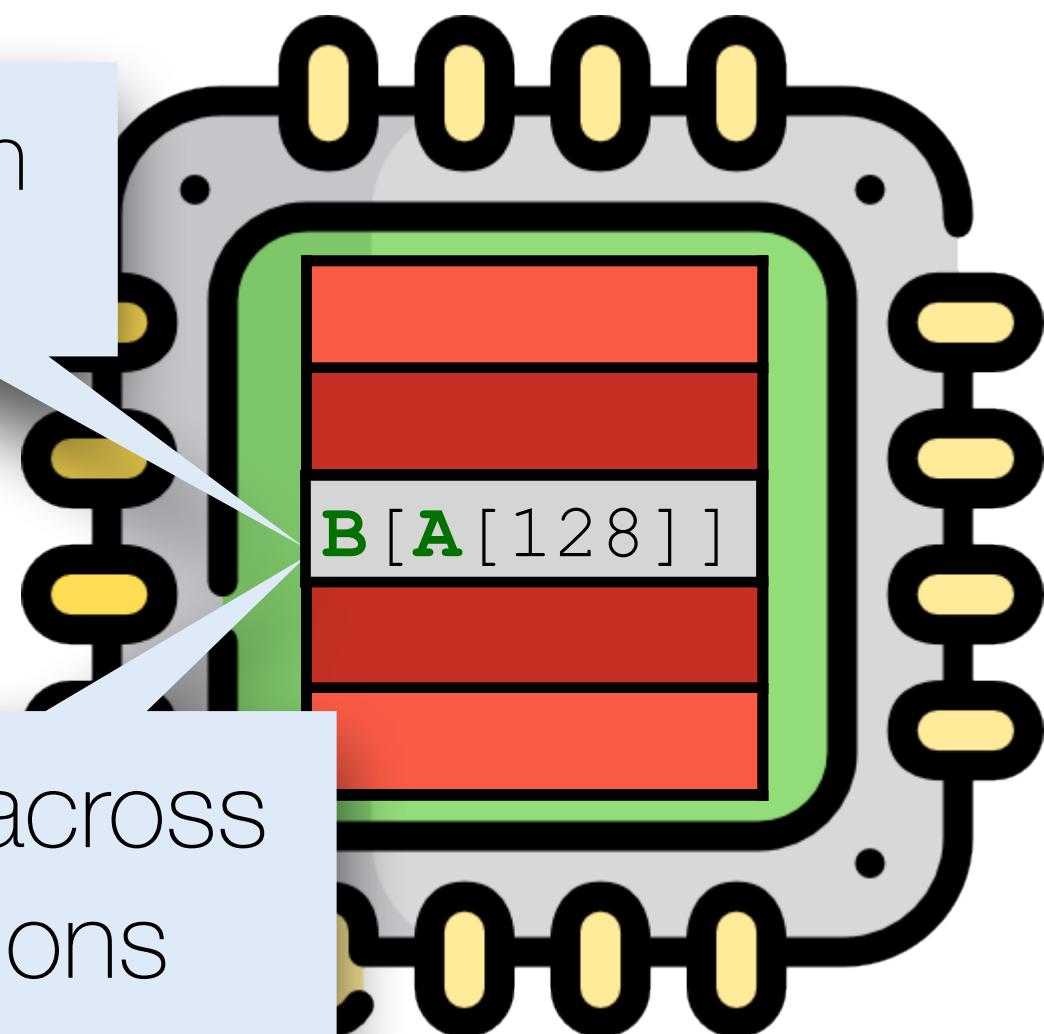
# Spectre V1



```
void f(int x)
if (x < A_size)
y = B[A[x]]
```



Depends on  
 $A[128]$



Persistent across  
speculations



What is in  $A[128]$ ?

1) Train branch predictor

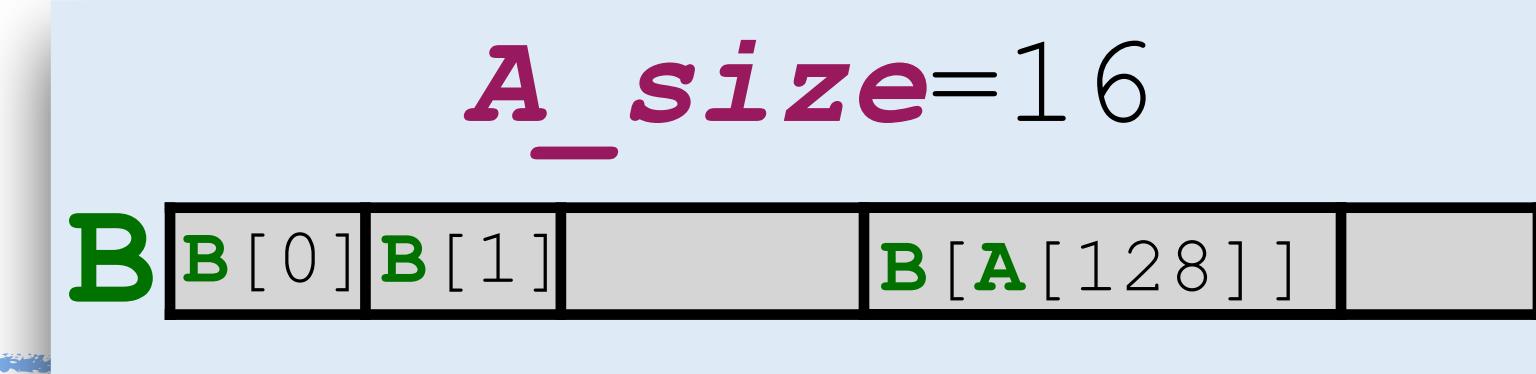
2) Prepare cache

3) Run with  $x = 128$

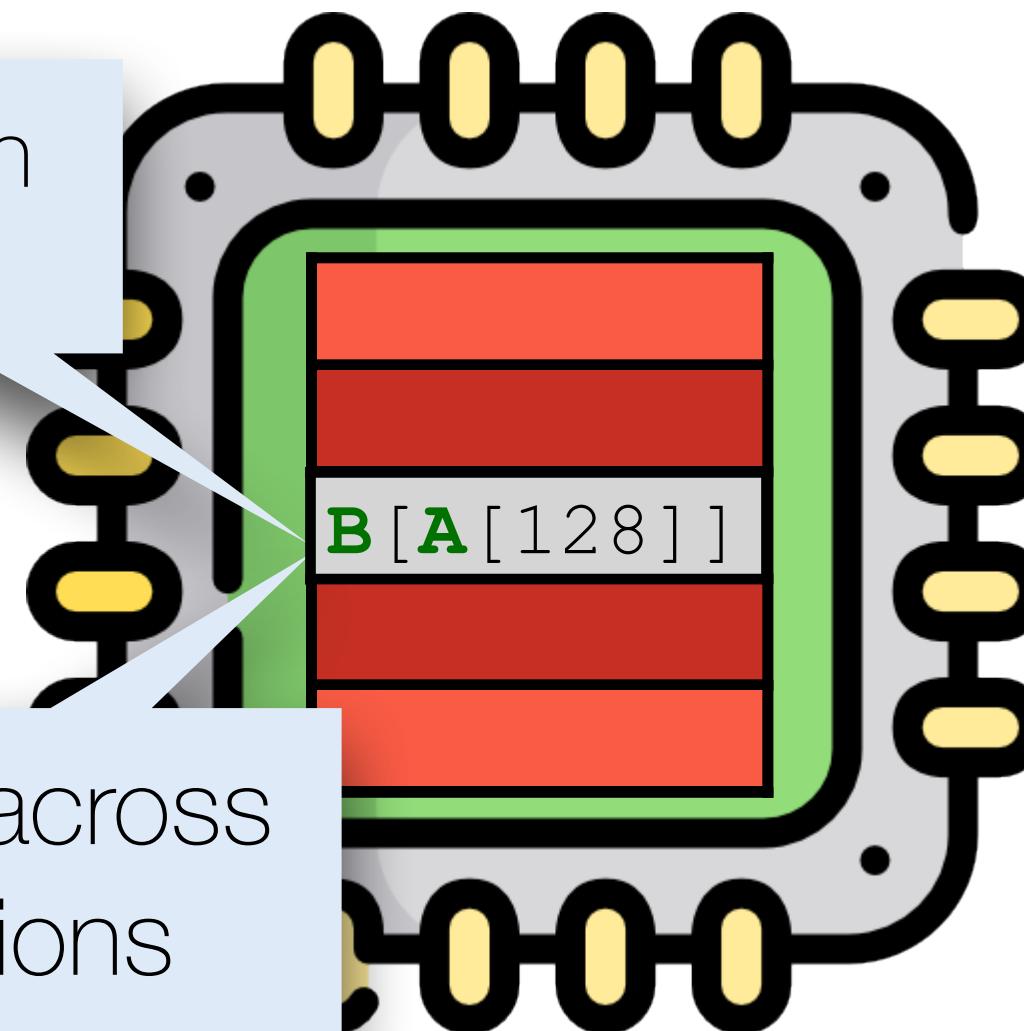
# Spectre V1



```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



Depends on  
 $A[128]$



Persistent across  
speculations



What is in  $A[128]$ ?

1) Train branch predictor

2) Prepare cache

3) Run with  $x = 128$

4) Extract from cache

# Speculative non-interference

# Speculative non-interference

Program  $\mathbf{P}$  is **speculatively non-interferent** if

$$\text{Leakage}(\mathbf{P}, \boxed{\text{chip}}) = \text{Leakage}(\mathbf{P}, \boxed{\text{ghost}})$$

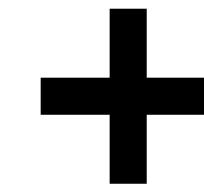
Information leaked by  
executing  $\mathbf{P}$  **without**  
speculative execution

Information leaked by  
executing  $\mathbf{P}$  **with**  
speculative execution

# How to capture leakage?

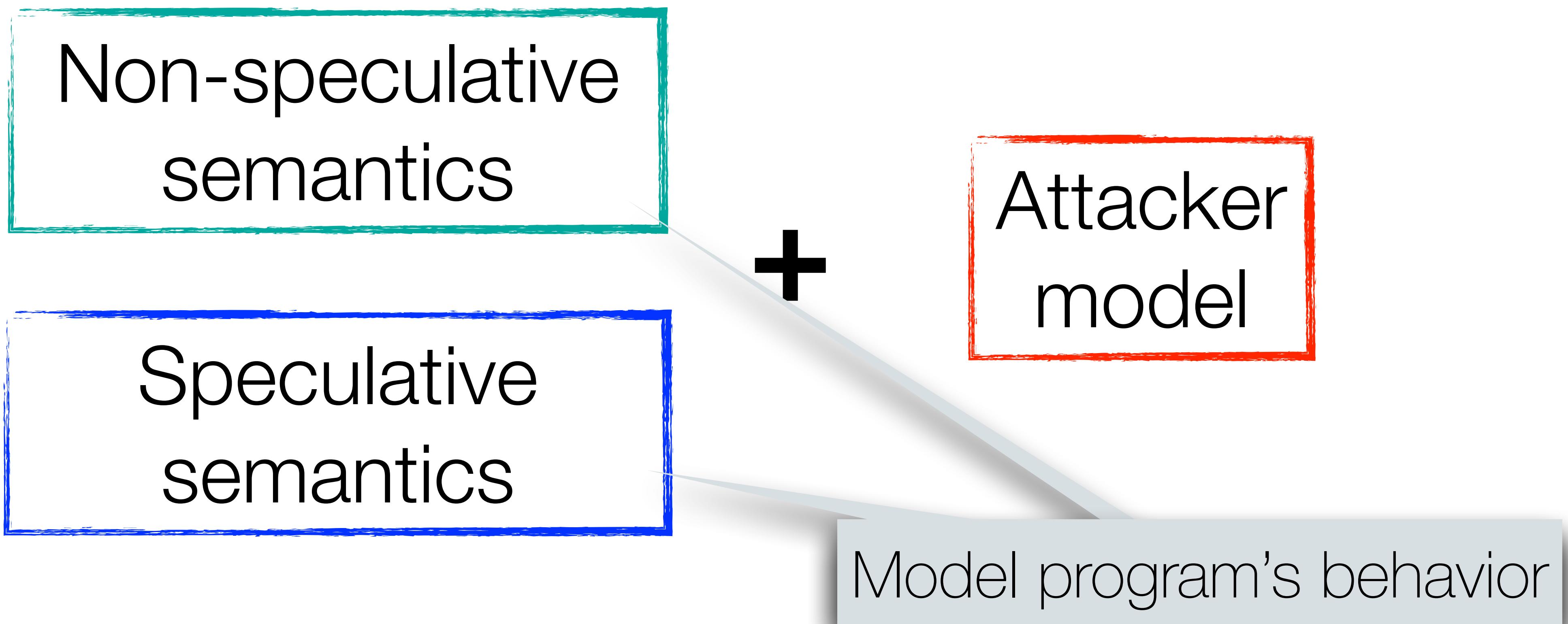
Non-speculative  
semantics

Speculative  
semantics

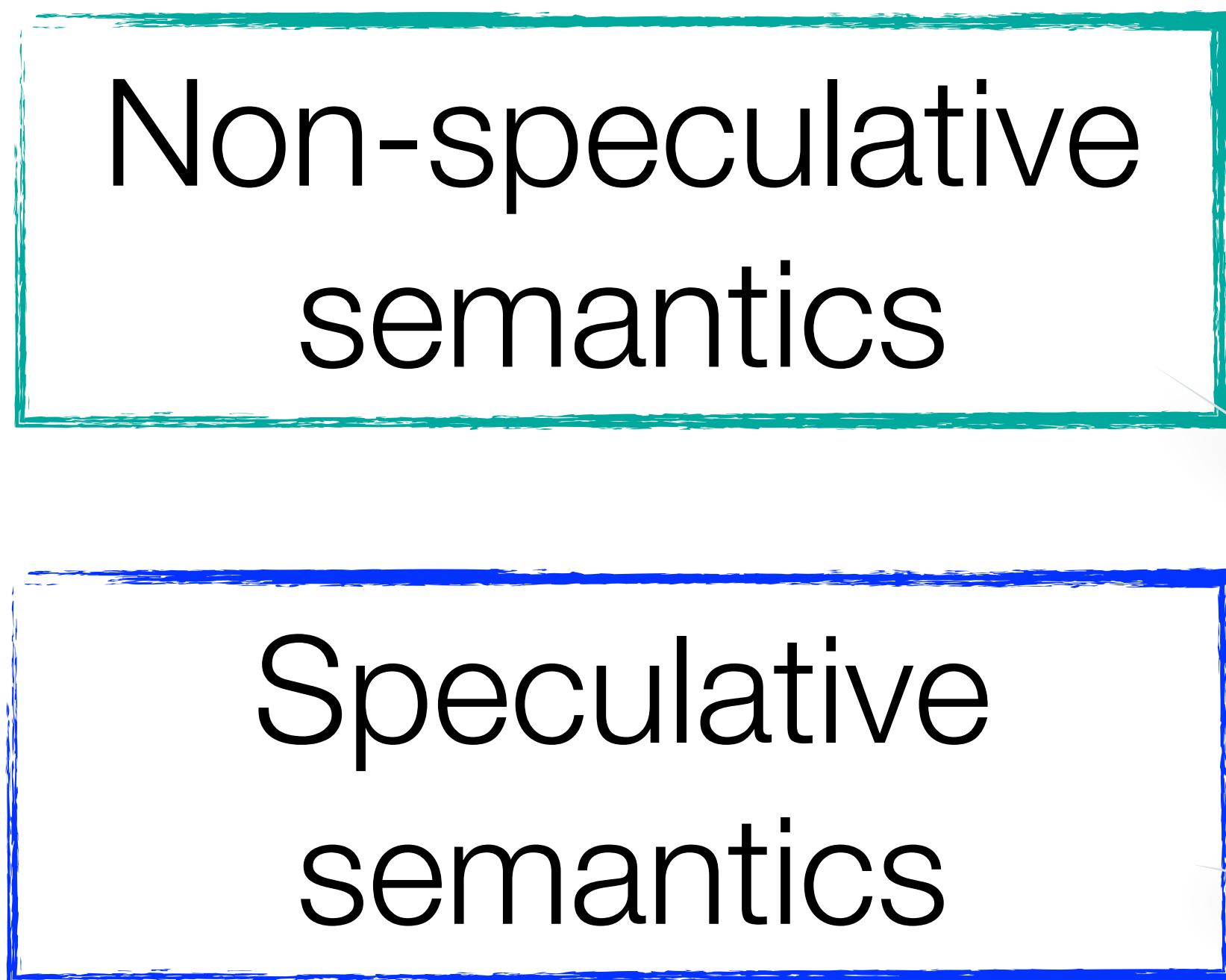


Attacker  
model

# How to capture leakage?



# How to capture leakage?



Attacker  
model

Capture attacker's  
observational power

Model program's behavior

# How to capture leakage?

Non-speculative  
semantics

Speculative  
semantics



Attacker  
model

Capture attacker's  
observational power

Model program's behavior

# How to capture leakage?

Standard *in-order* semantics

Non-speculative  
semantics

Speculative  
semantics

Capture attacker's  
observational power

Attacker  
model

Model program's behavior



# How to capture leakage?

Non-speculative  
semantics

Speculative  
semantics



Attacker  
model

Capture attacker's  
observational power

Model program's behavior

# How to capture leakage?

***Prediction Oracle  $O$ :***

branch prediction + length of  
speculative window

Speculative  
semantics



Attacker  
model

Capture attacker's  
observational power

Model program's behavior

# How to capture leakage?

Capture attacker's observational power

***Prediction Oracle O***:

branch prediction + length of speculative window

Speculative semantics

Starts ***speculative transactions*** upon branch instructions

- ***Committed*** upon correct speculation
- ***Rolled-back*** upon misprediction

Model program's behavior

# How to capture leakage?

Non-speculative  
semantics

Speculative  
semantics



Attacker  
model

Capture attacker's  
observational power

Model program's behavior

# How to capture leakage?

Non-speculative  
semantics

Speculative  
semantics

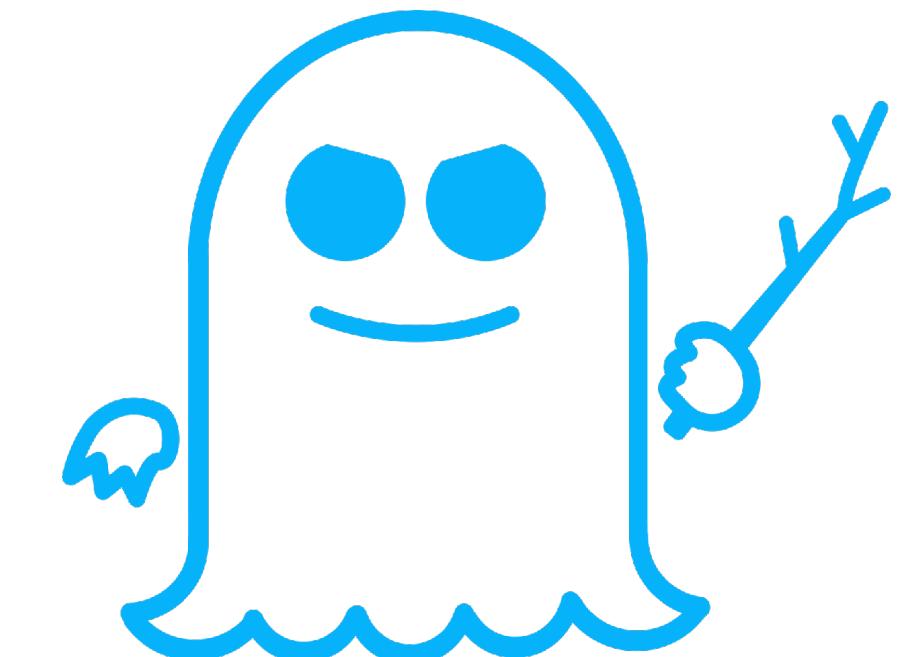


Attacker  
model

- Attack can observe:
- locations of **memory accesses**
  - **branch/jump** targets
  - **start/end** speculative execution

# How to capture leakage?

```
1. if (x < A_size)
2.     y = A [x]
3.     z = B [y]
4. end
```

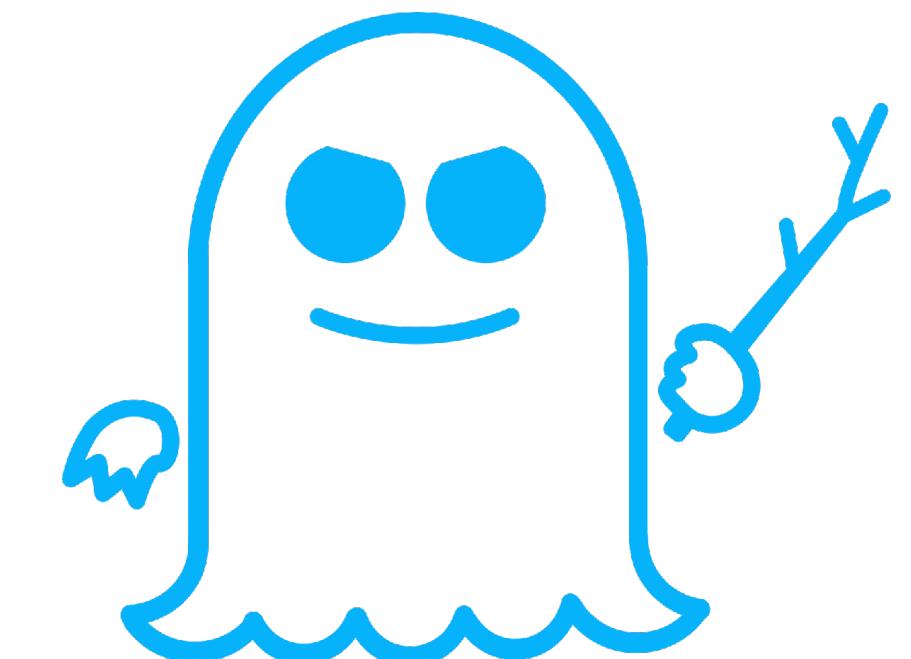


# How to capture leakage?

```
1. if (x < A_size)
2.   y = A[x]
3.   z = B[y]
4. end
```

**x** > **A\_size**

**x** < **A\_size** predicted as satisfied

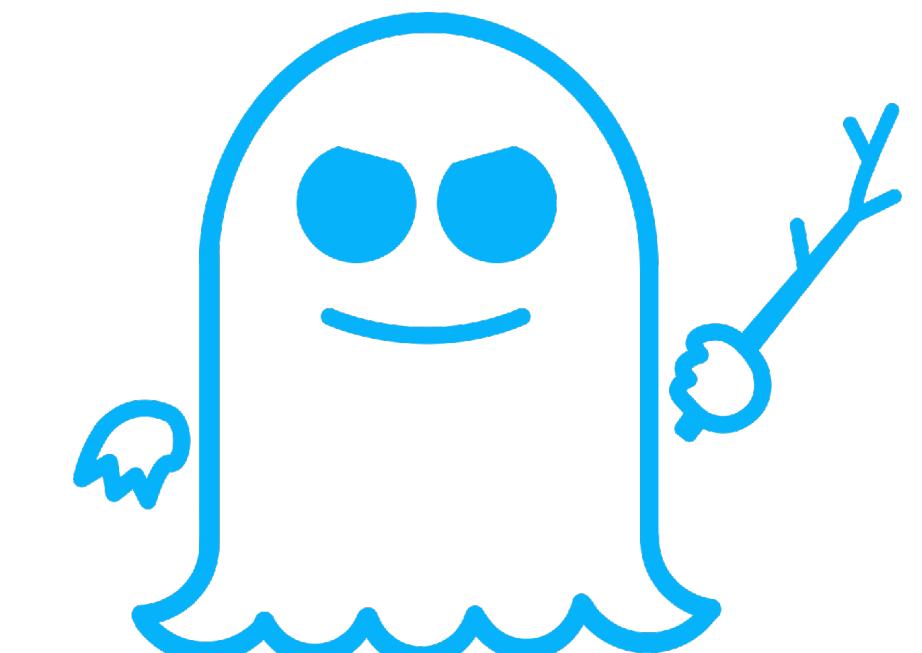


# How to capture leakage?

```
1. if (x < A_size)  
2.     y = A [x]  
3.     z = B [y]  
4. end
```

**x** > A\_size

**x** < A\_size predicted as satisfied



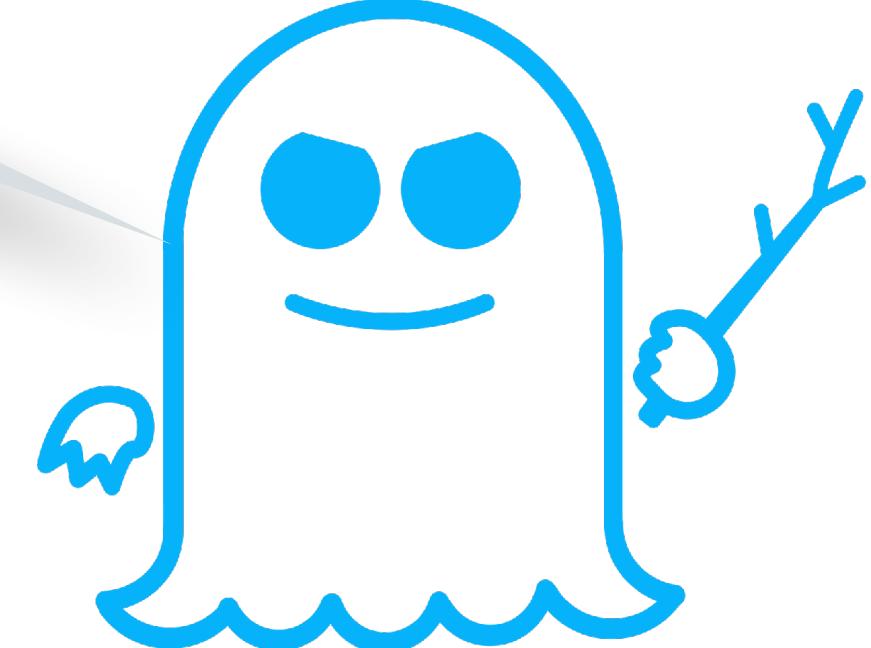
# How to capture leakage?

```
1. if (x < A_size)  
2.     y = A[x]  
3.     z = B[y]  
4. end
```

**x** > A\_size

**x** < A\_size predicted as satisfied

start  
pc 2

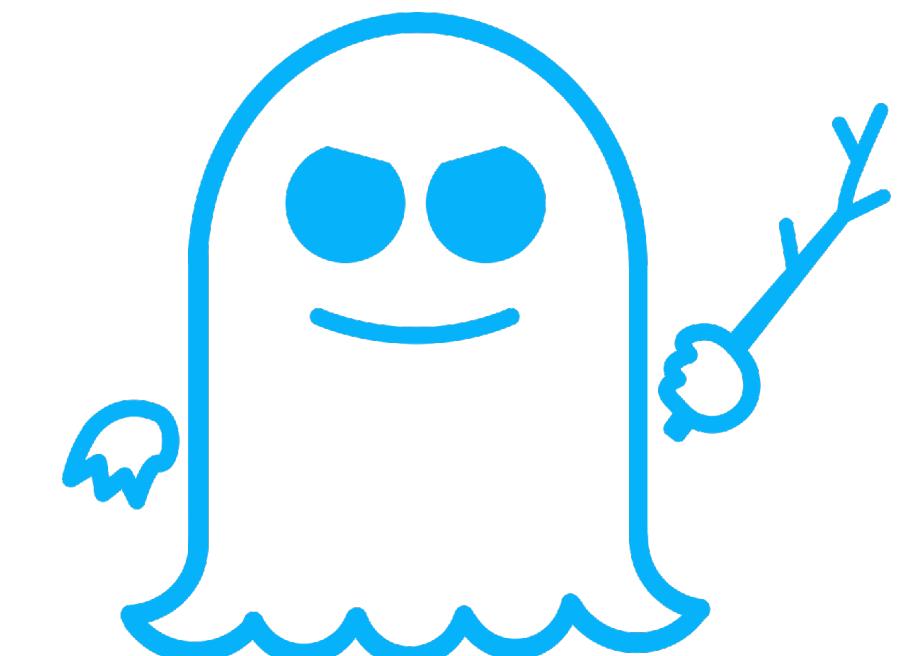
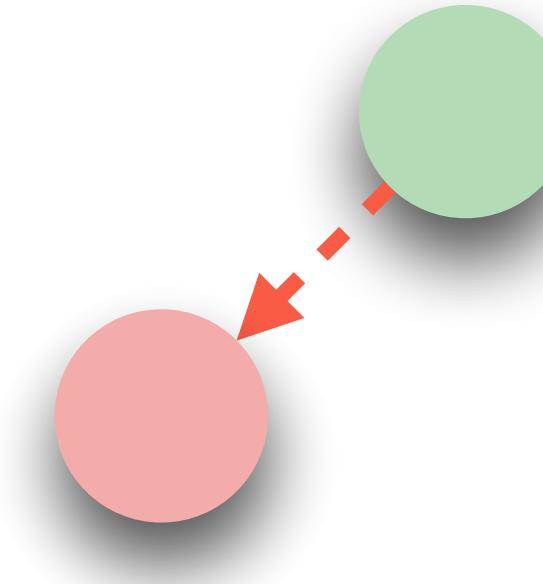


# How to capture leakage?

```
1. if (x < A_size)
2.     y = A [x]
3.     z = B [y]
4. end
```

**x** > A\_size

**x** < A\_size predicted as satisfied

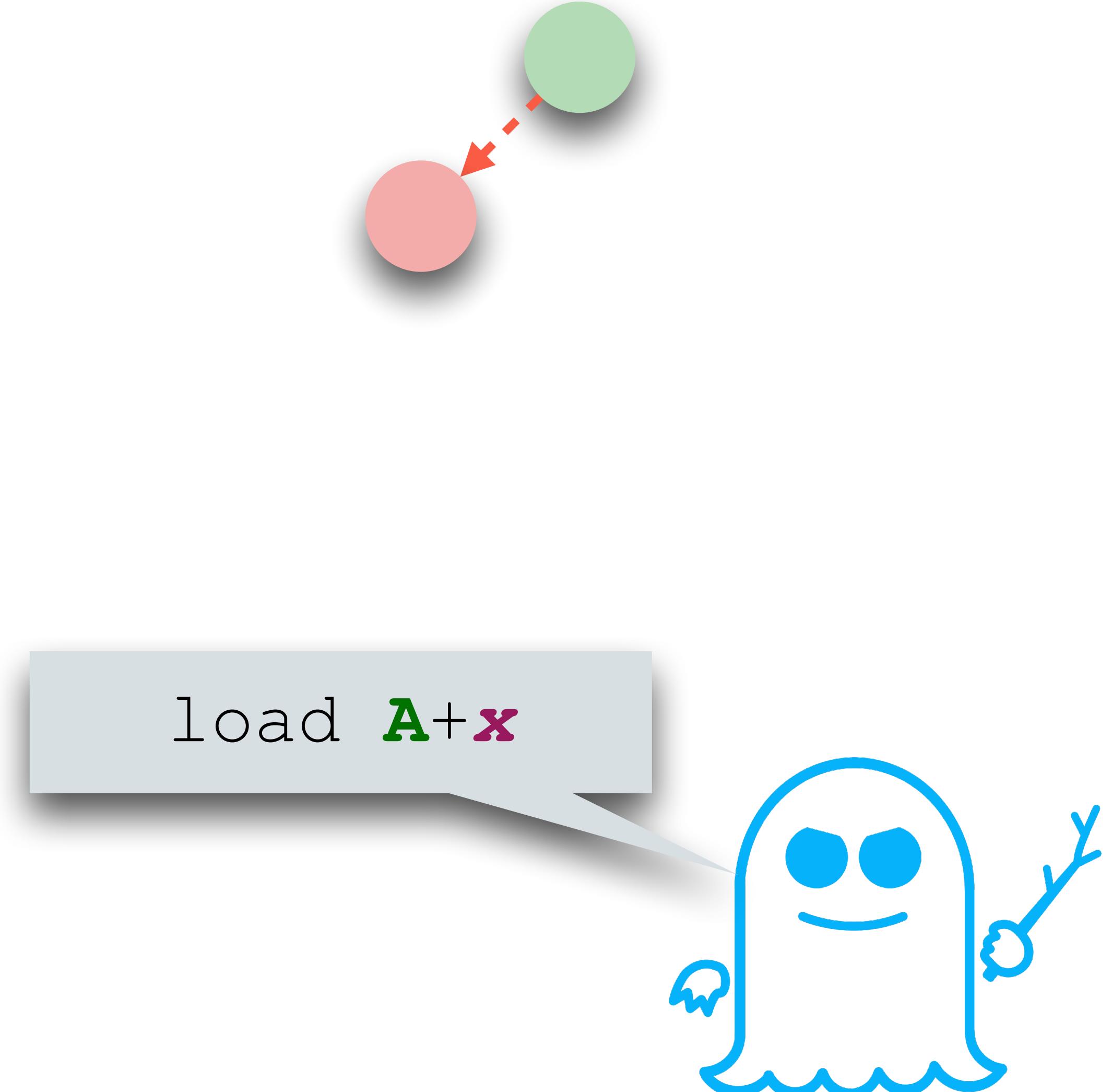


# How to capture leakage?

```
1. if (x < A_size)
2.     y = A[x]
3.     z = B[y]
4. end
```

**x** > A\_size

**x** < A\_size predicted as satisfied

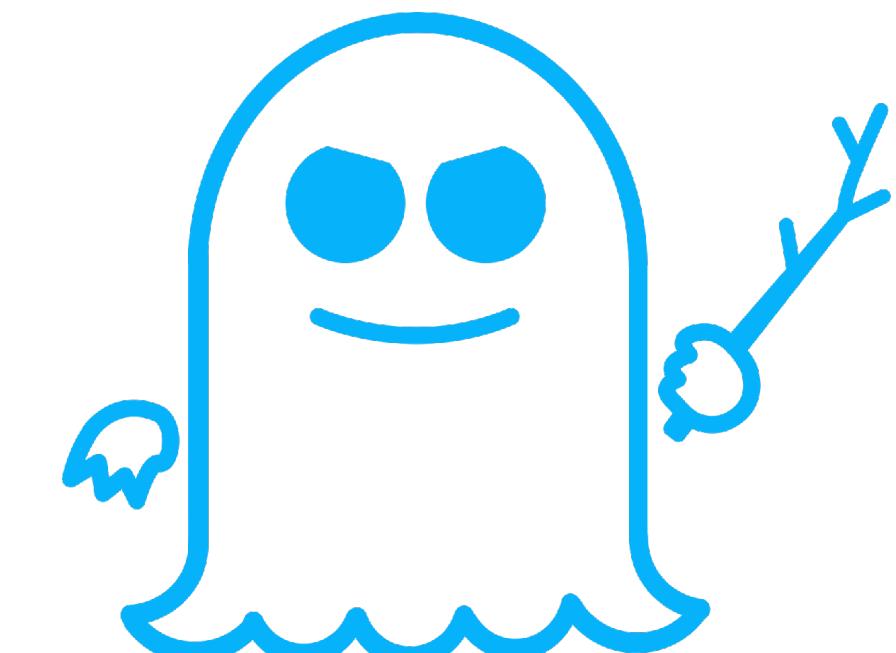
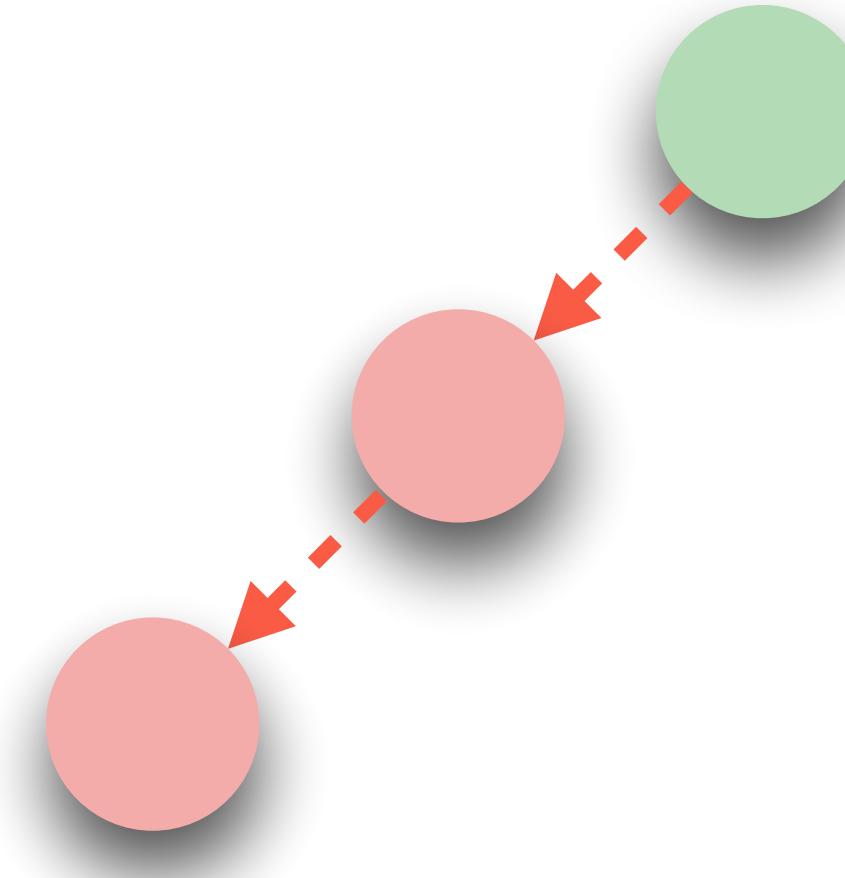


# How to capture leakage?

```
1. if (x < A_size)
2.     y = A[x]
3.     z = B[y]
4. end
```

**x** > **A\_size**

**x** < **A\_size** predicted as satisfied

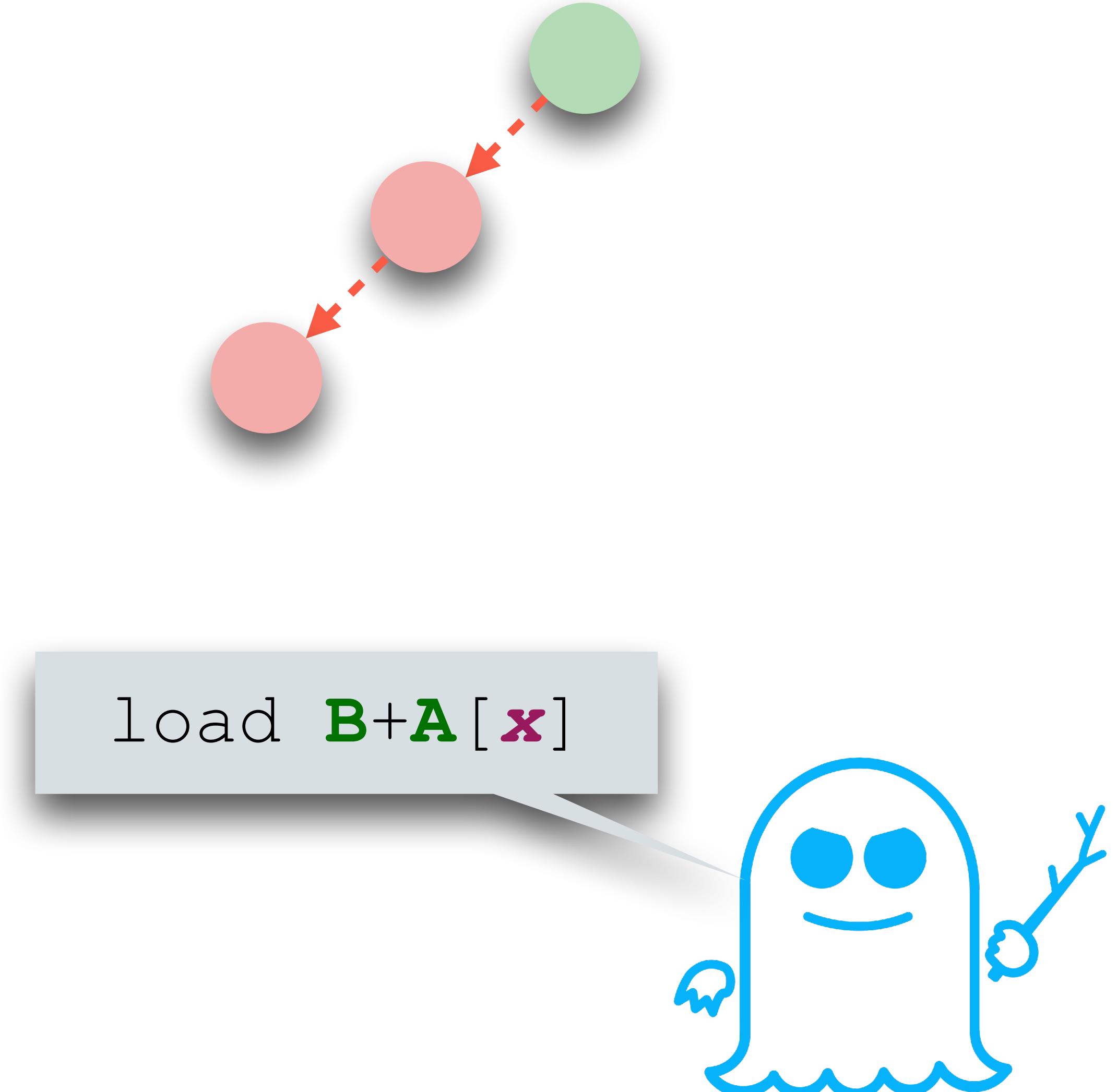


# How to capture leakage?

```
1. if (x < A_size)
2.   y = A[x]
3.   z = B[y]
4. end
```

**x** > **A\_size**

**x** < **A\_size** predicted as satisfied

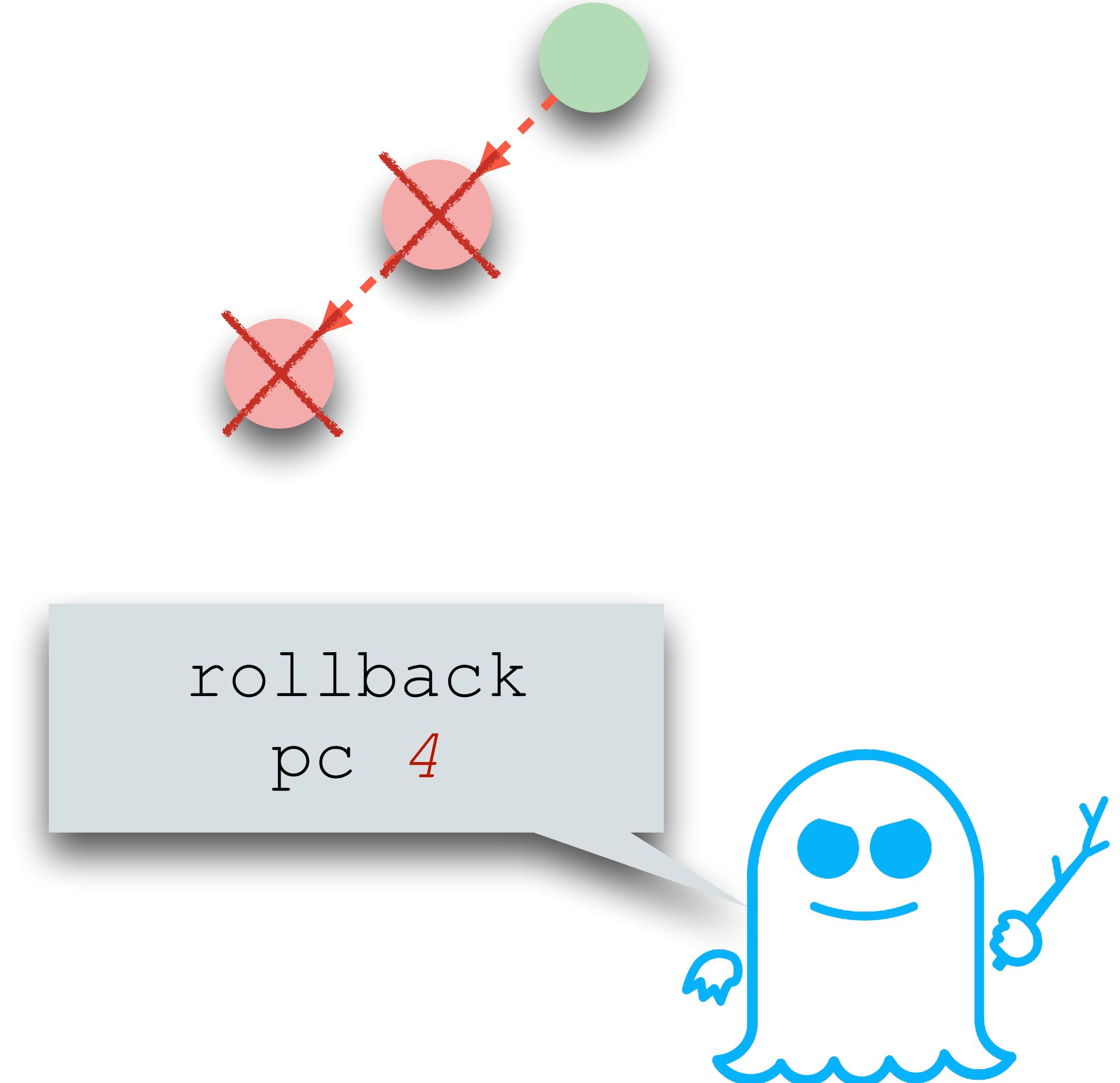


# How to capture leakage?

```
1. if (x < A_size)
2.   y = A[x]
3.   z = B[y]
4. end
```

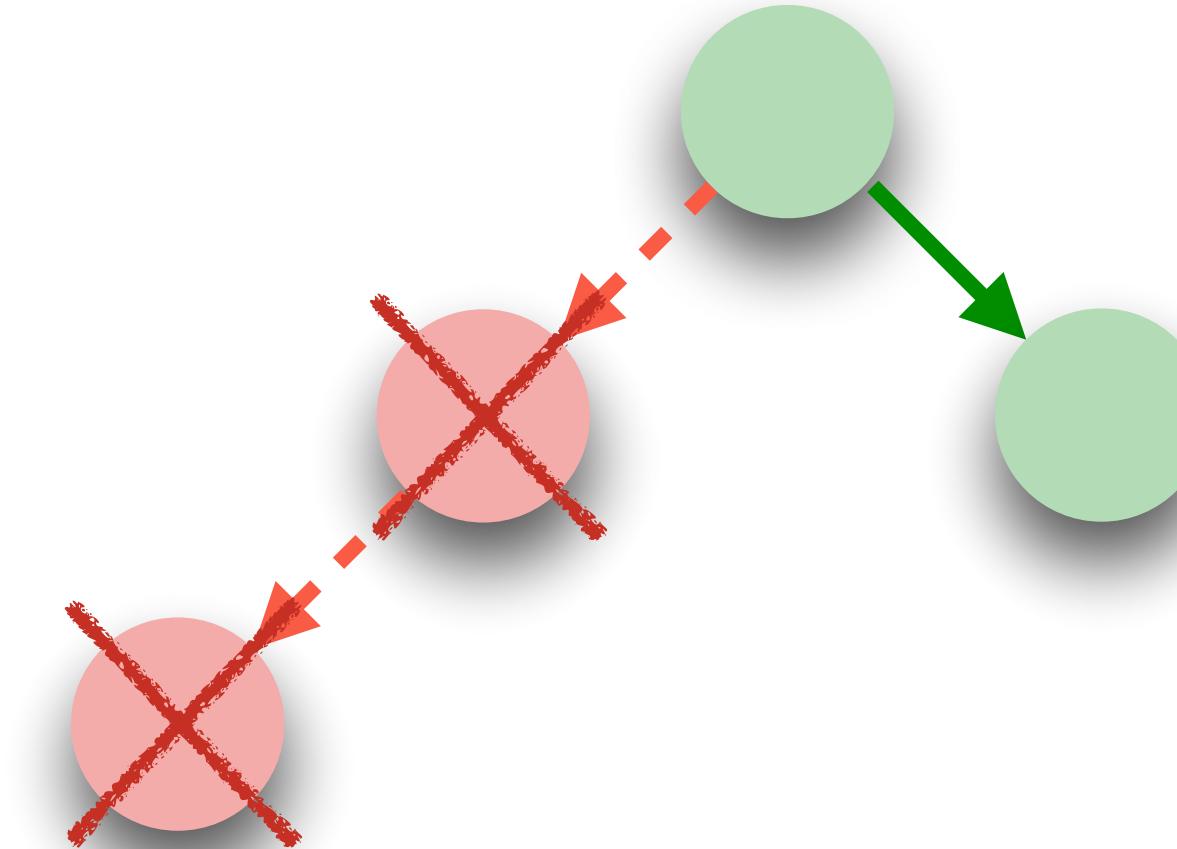
**x** > **A\_size**

**x** < **A\_size** predicted as satisfied



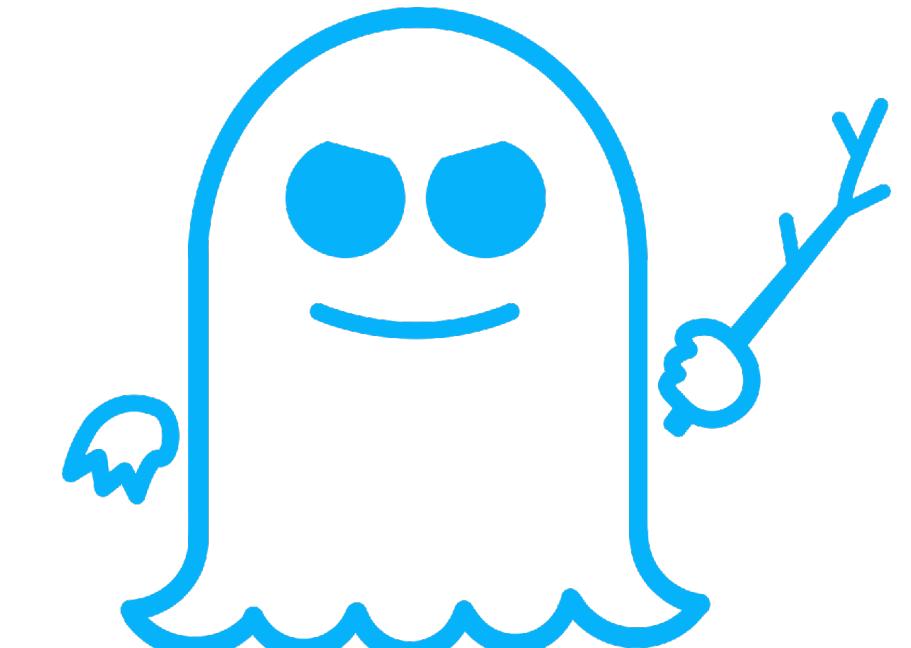
# How to capture leakage?

```
1. if (x < A_size)
2.   y = A[x]
3.   z = B[y]
4. end
```



**x** > **A\_size**

**x** < **A\_size** predicted as satisfied



# Speculative non-interference

Formally!

# Speculative non-interference

Formally!

Program **P** is **speculatively non-interferent** for prediction oracle **O** if

# Speculative non-interference

Formally!

Program  $\mathbf{P}$  is **speculatively non-interferent** for prediction oracle  $\mathbf{O}$  if

For all program states  $s$  and  $s'$ :

# Speculative non-interference

Formally!

Program  $\mathbf{P}$  is **speculatively non-interferent** for prediction oracle  $\mathbf{O}$  if

For all program states  $s$  and  $s'$ :

$$\mathbf{P}_{\text{non-spec}}(s) = \mathbf{P}_{\text{non-spec}}(s')$$

# Speculative non-interference

Formally!

Program  $\mathbf{P}$  is **speculatively non-interferent** for prediction oracle  $\mathbf{O}$  if

For all program states  $\mathbf{s}$  and  $\mathbf{s}'$ :

$$\begin{aligned} \mathbf{P}_{\text{non-spec}}(\mathbf{s}) &= \mathbf{P}_{\text{non-spec}}(\mathbf{s}') \\ \Rightarrow \mathbf{P}_{\text{spec}}(\mathbf{s}, \mathbf{O}) &= \mathbf{P}_{\text{spec}}(\mathbf{s}', \mathbf{O}) \end{aligned}$$

# Speculative non-interference

Formally!

Program  $\mathbf{P}$  is **speculatively non-interferent** for prediction oracle  $\mathbf{O}$  if

For all program states  $\mathbf{s}$  and  $\mathbf{s}'$ :

$$\begin{aligned} \mathbf{P}_{\text{non-spec}}(\mathbf{s}) &= \mathbf{P}_{\text{non-spec}}(\mathbf{s}') \\ \Rightarrow \mathbf{P}_{\text{spec}}(\mathbf{s}, \mathbf{O}) &= \mathbf{P}_{\text{spec}}(\mathbf{s}', \mathbf{O}) \end{aligned}$$

*See paper for:* reasoning about **arbitrary prediction oracles**

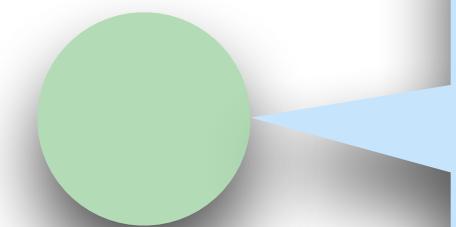
# Speculative non-interference

```
1. if (x < A_size)
2.     y = A [x]
3.     z = B [y]
4. end
```

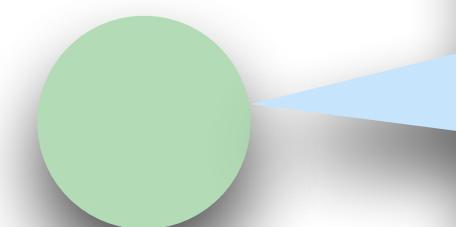
# Speculative non-interference

```
1. if (x < A_size)
2.   y = A[x]
3.   z = B[y]
4. end
```

**x** < **A\_size** predicted as satisfied



**x**=128  
**A\_size**=16  
**A**[128]=0

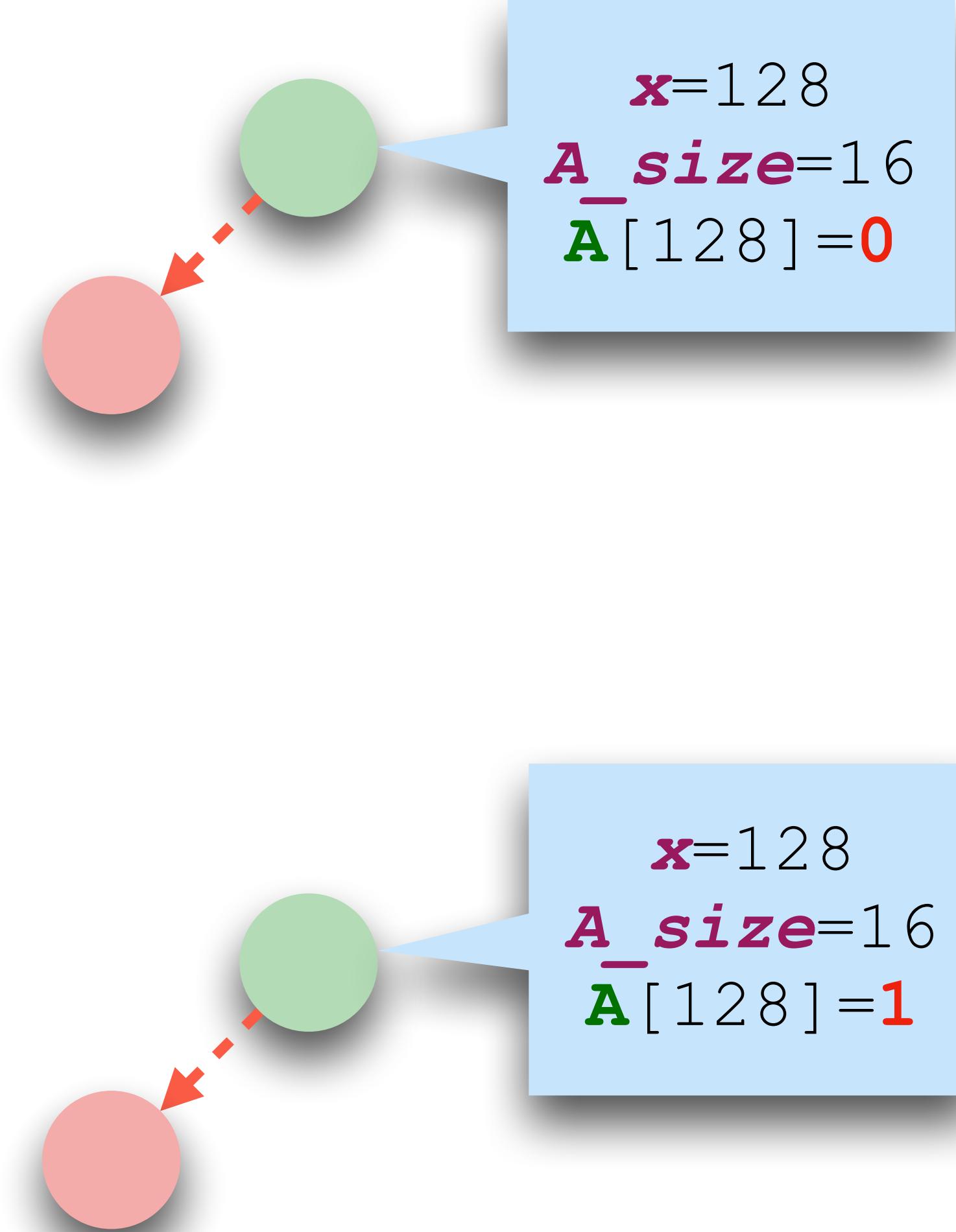


**x**=128  
**A\_size**=16  
**A**[128]=1

# Speculative non-interference

```
1. if (x < A_size)
2.     y = A[x]
3.     z = B[y]
4. end
```

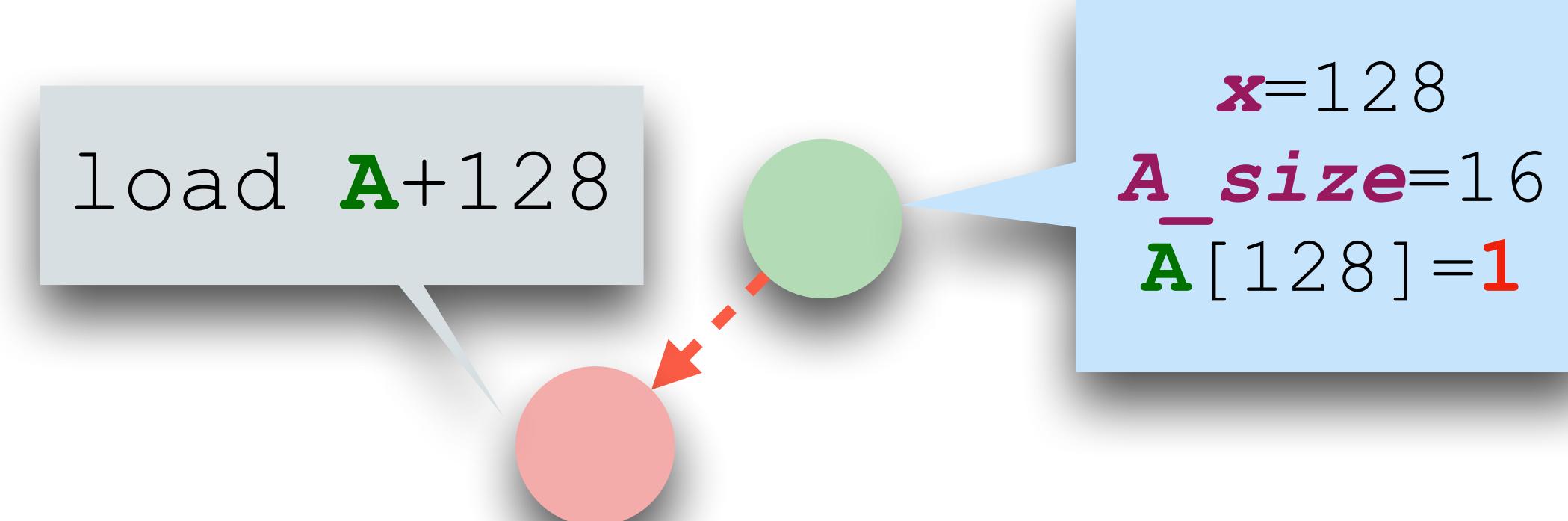
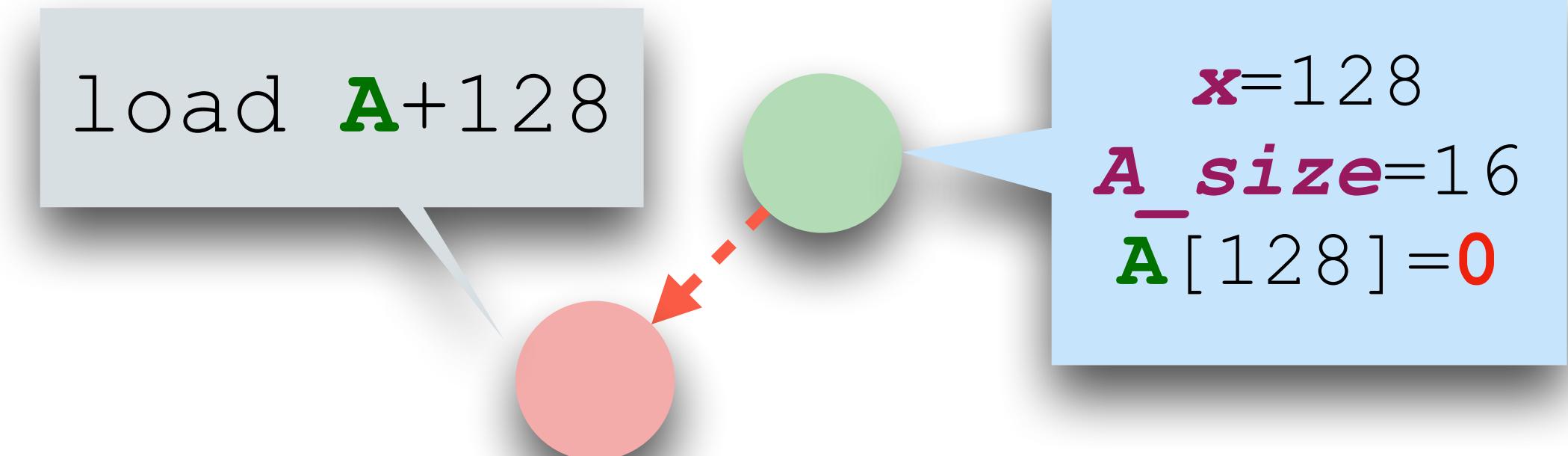
**x** < **A\_size** predicted as satisfied



# Speculative non-interference

```
1. if (x < A_size)
2.   y = A[x]
3.   z = B[y]
4. end
```

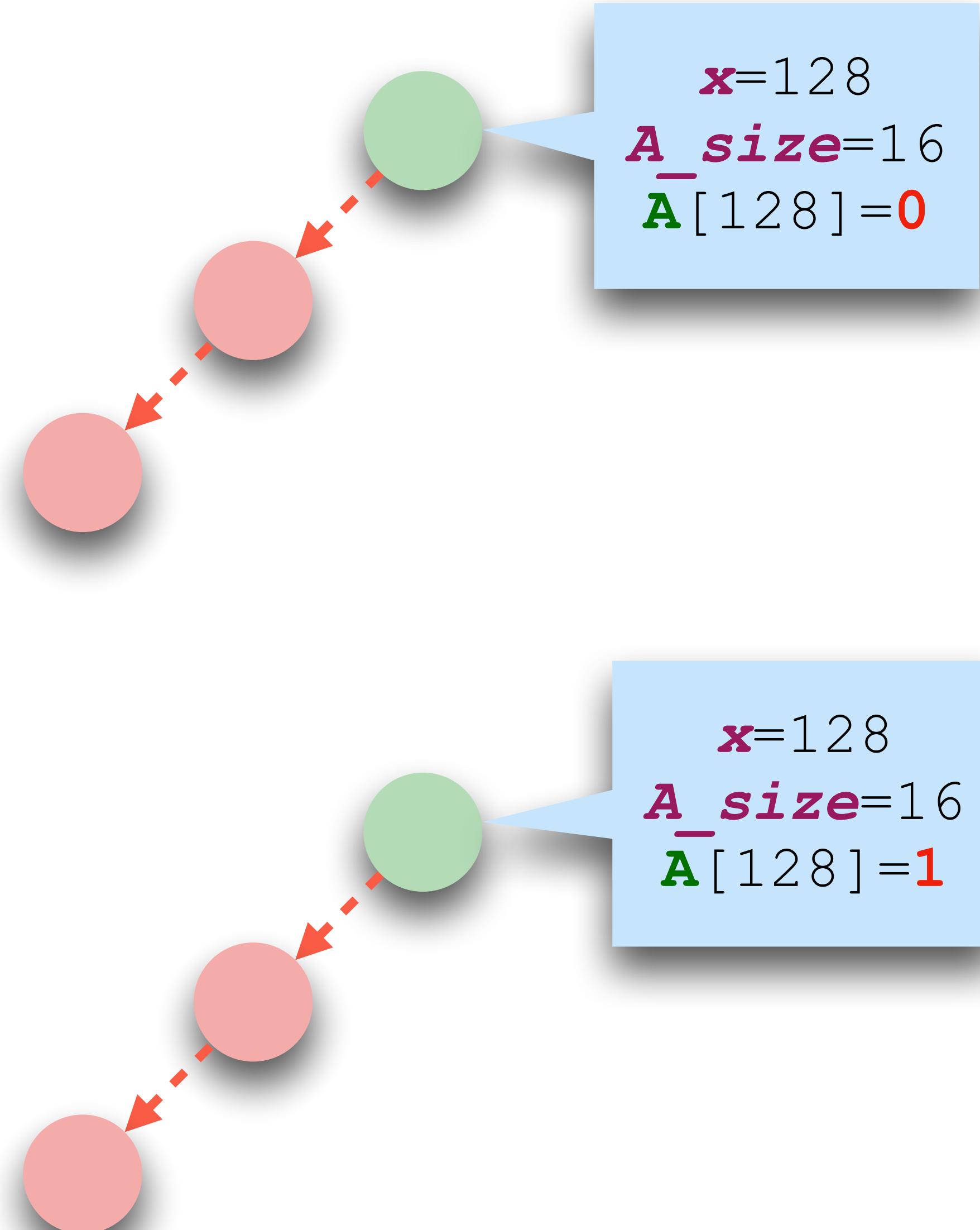
**x** < **A\_size** predicted as satisfied



# Speculative non-interference

```
1. if (x < A_size)
2.   y = A[x]
3.   z = B[y]
4. end
```

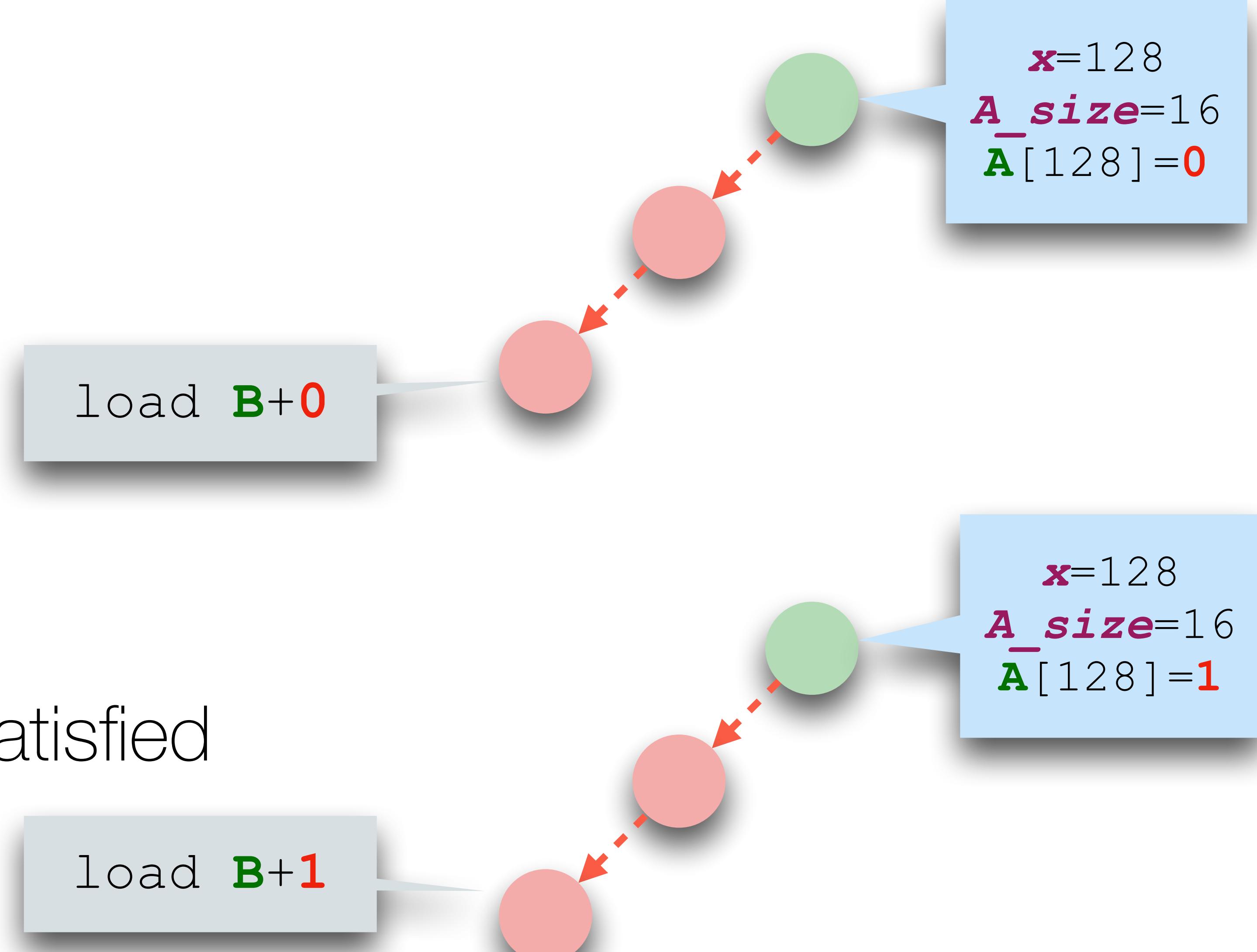
**x** < **A\_size** predicted as satisfied



# Speculative non-interference

```
1. if (x < A_size)
2.   y = A[x]
3.   z = B[y]
4. end
```

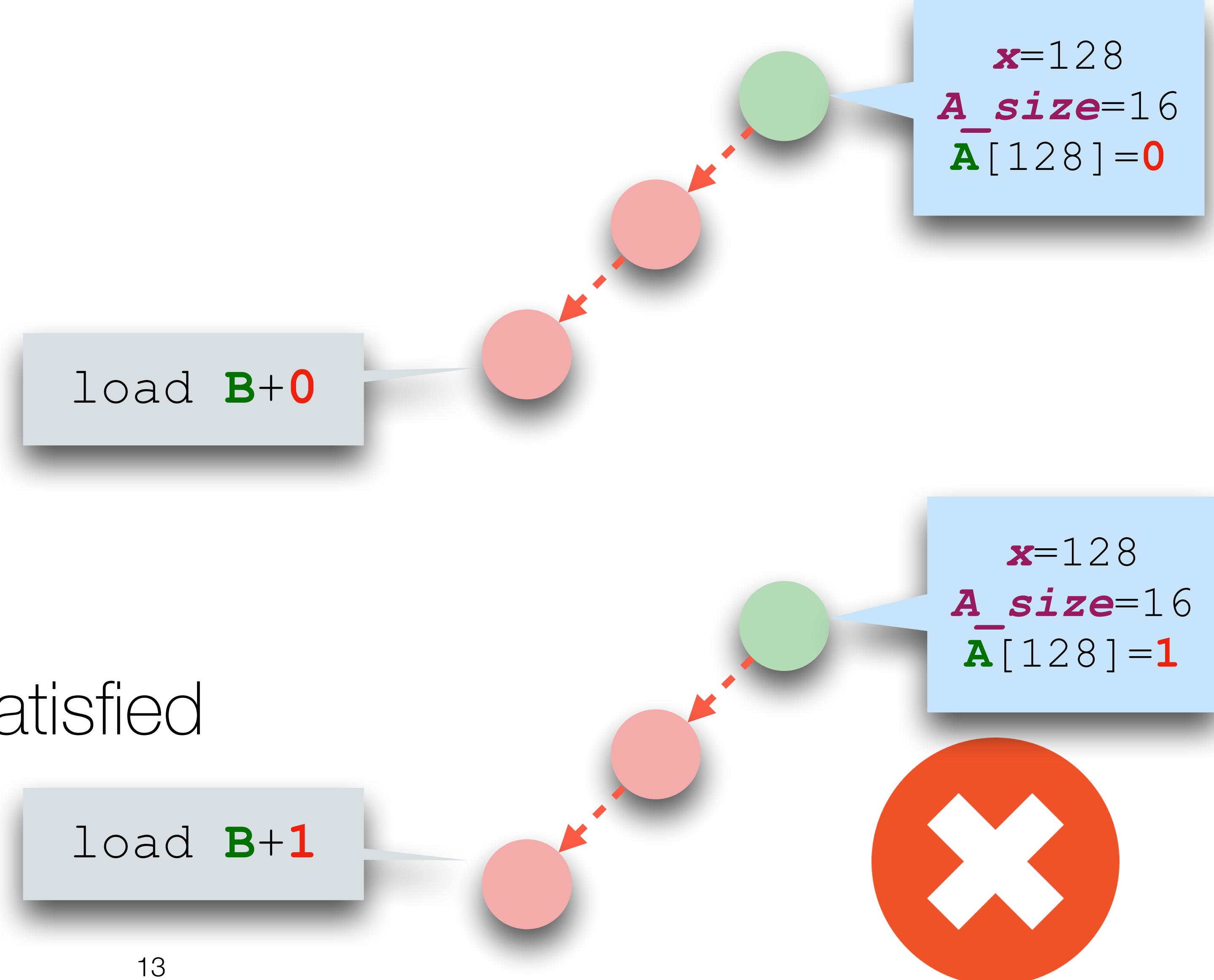
**x** < **A\_size** predicted as satisfied



# Speculative non-interference

```
1. if (x < A_size)
2.   y = A[x]
3.   z = B[y]
4. end
```

**x** < **A\_size** predicted as satisfied

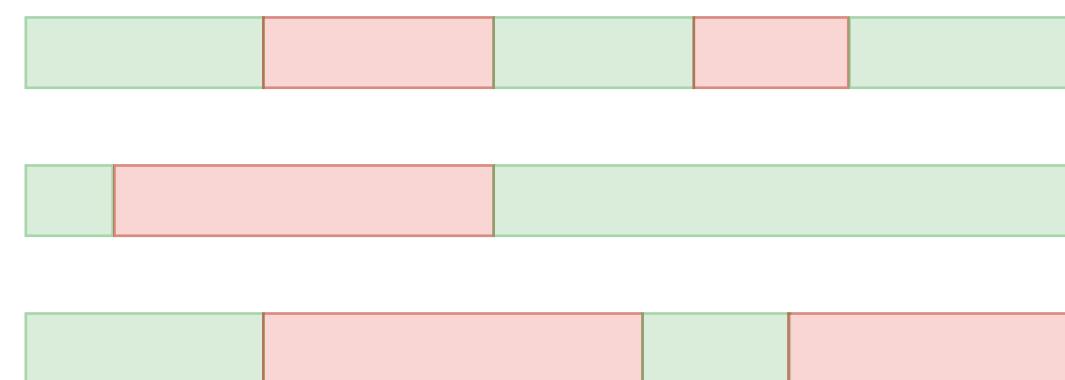


# Detecting speculative leaks

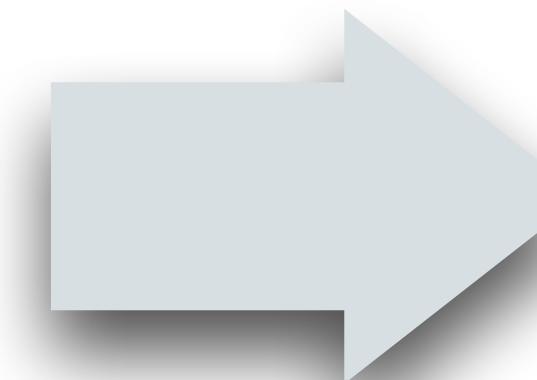
# Detecting speculative leaks

```
1. if (x < A_size)  
2.     y = A[x]  
3.     z = B[y]  
4. end
```

Symbolic  
execution



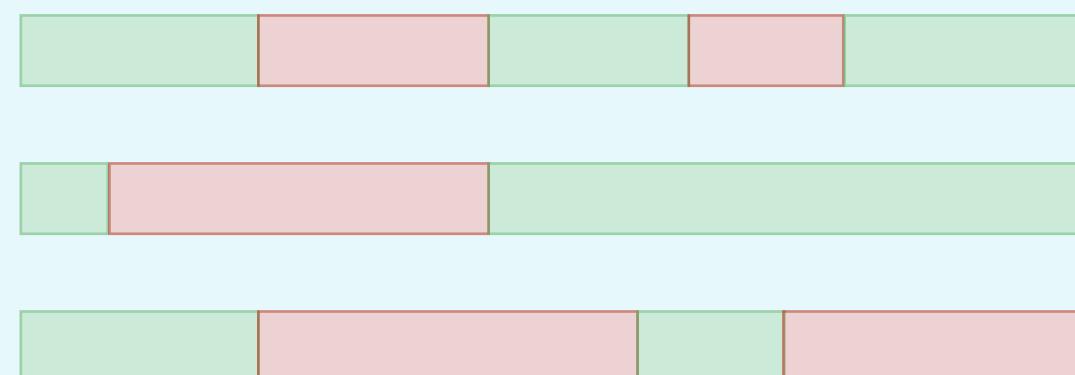
Detect leaks



# Detecting speculative leaks

```
1. if (x < A_size)  
2.     y = A[x]  
3.     z = B[y]  
4. end
```

Symbolic  
execution



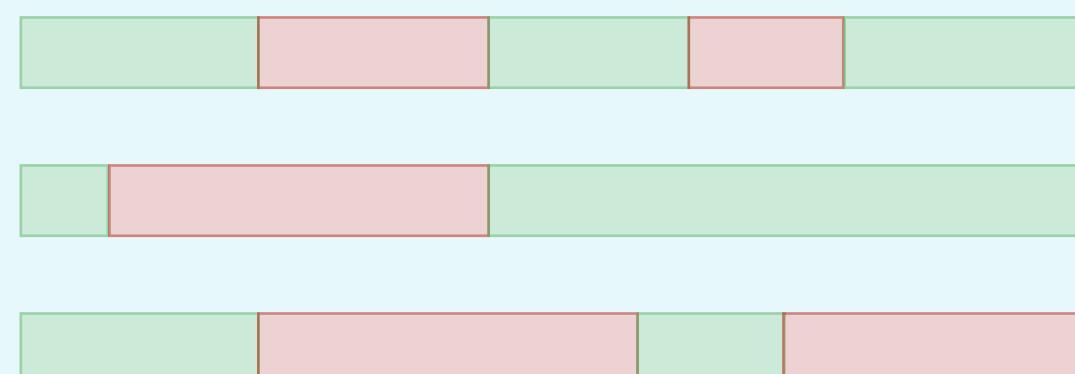
Detect leaks



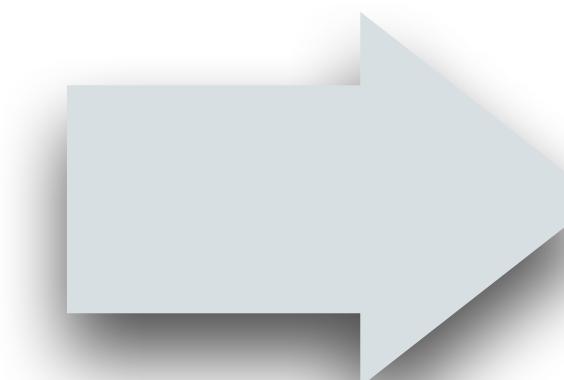
# Detecting speculative leaks

```
1. if (x < A_size)  
2.     y = A[x]  
3.     z = B[y]  
4. end
```

Symbolic  
execution



Detect leaks



**Symbolic trace:** path condition +  
observations along the symbolic path

# Symbolic execution

```
1. if (x < A_size)
2.     y = A[x]
3.     z = B[y]
4. end
```

# Symbolic execution

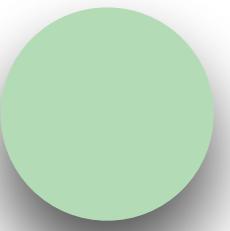
```
1. if (x < A_size)  
2.     y = A[x]  
3.     z = B[y]  
4. end
```



*Always mispredict*  
branch instructions

# Symbolic execution

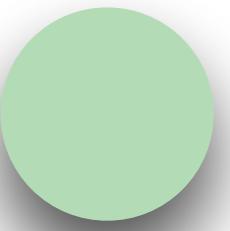
```
1. if (x < A_size)           true  
2.   y = A[x]  
3.   z = B[y]  
4. end
```



*Always mispredict*  
branch instructions

# Symbolic execution

```
1. if (x < A_size)           true  
2.   y = A [x]  
3.   z = B [y]  
4. end
```

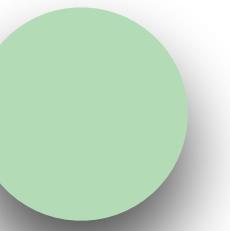


*Always mispredict*  
branch instructions

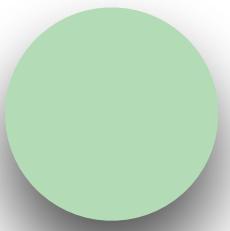
# Symbolic execution

```
1. if ( $x < A\_size$ )  
2.      $y = A[x]$   
3.      $z = B[y]$   
4. end
```

$x \geq A\_size$



$x < A\_size$



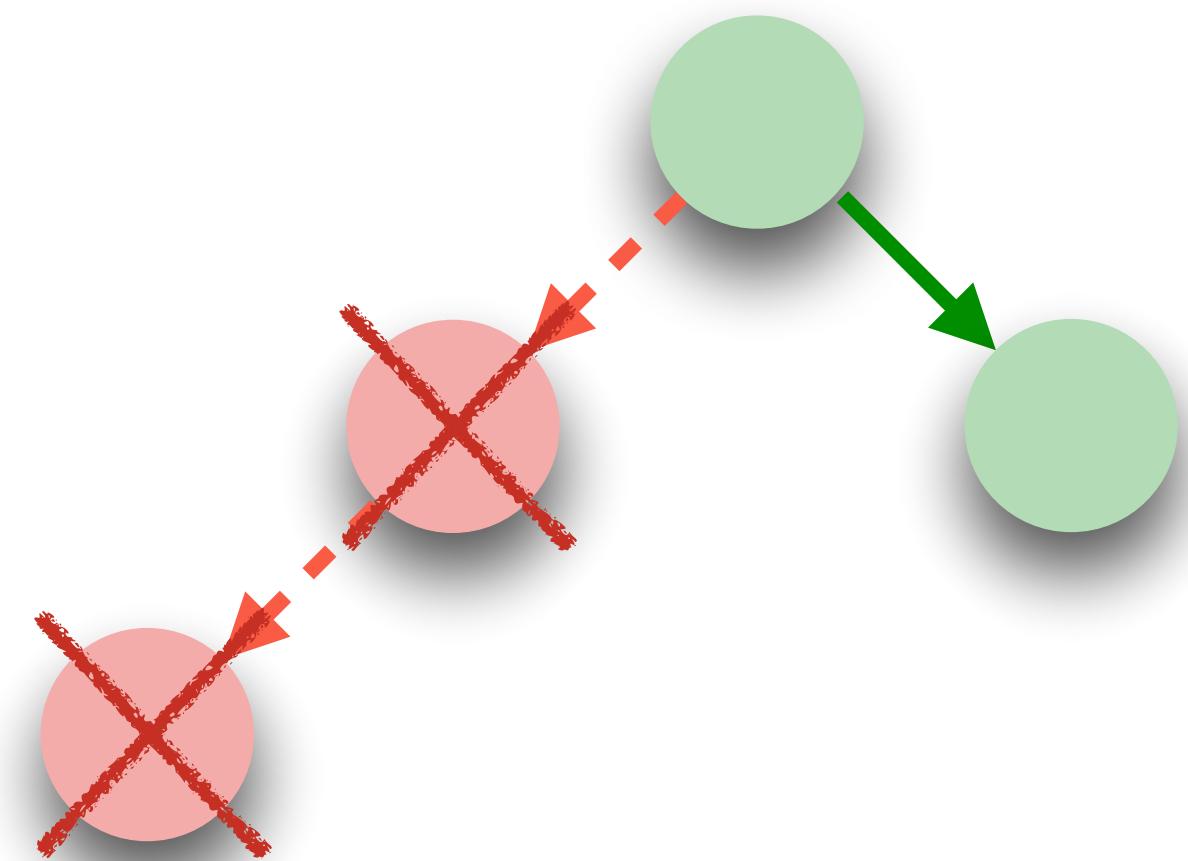
*Always mispredict*  
branch instructions

# Symbolic execution

```
1. if ( $x < A\_size$ )
2.    $y = A[x]$ 
3.    $z = B[y]$ 
4. end
```

$x \geq A\_size$

$x < A\_size$

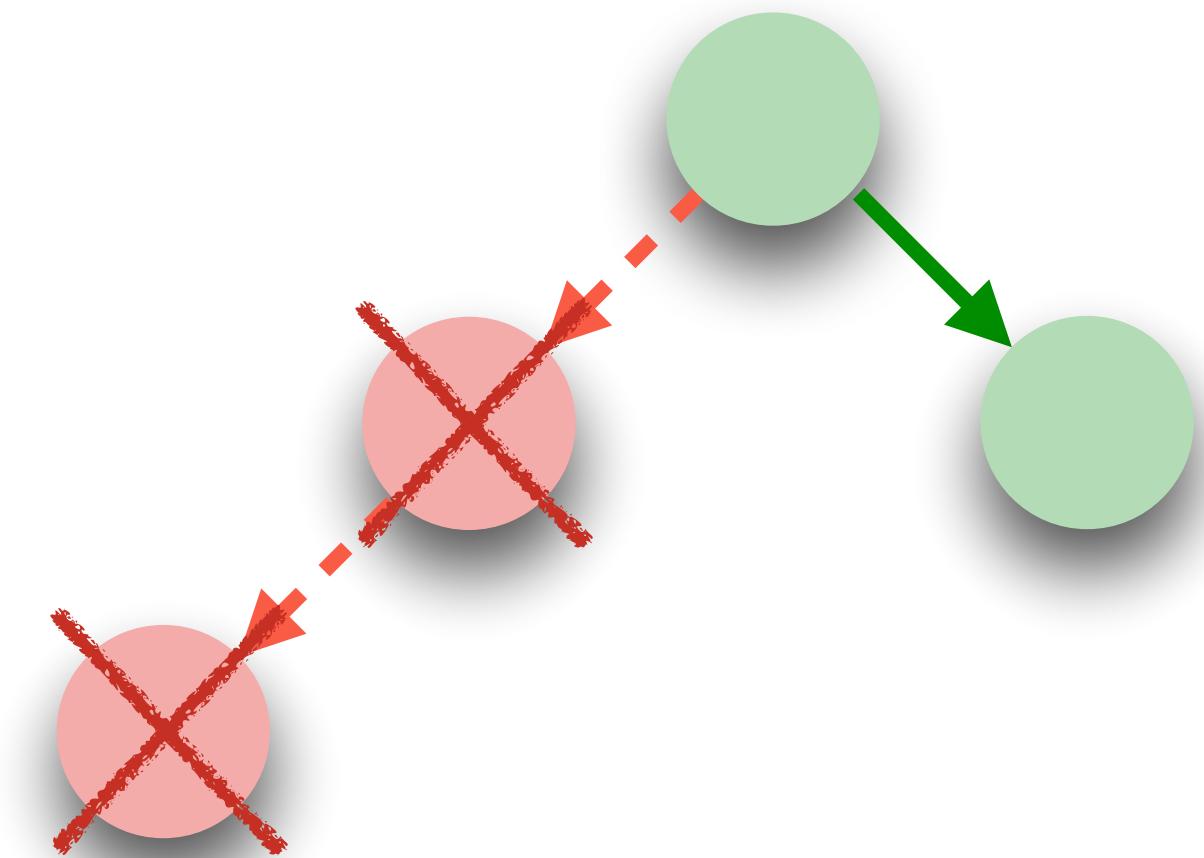


*Always mispredict*  
branch instructions

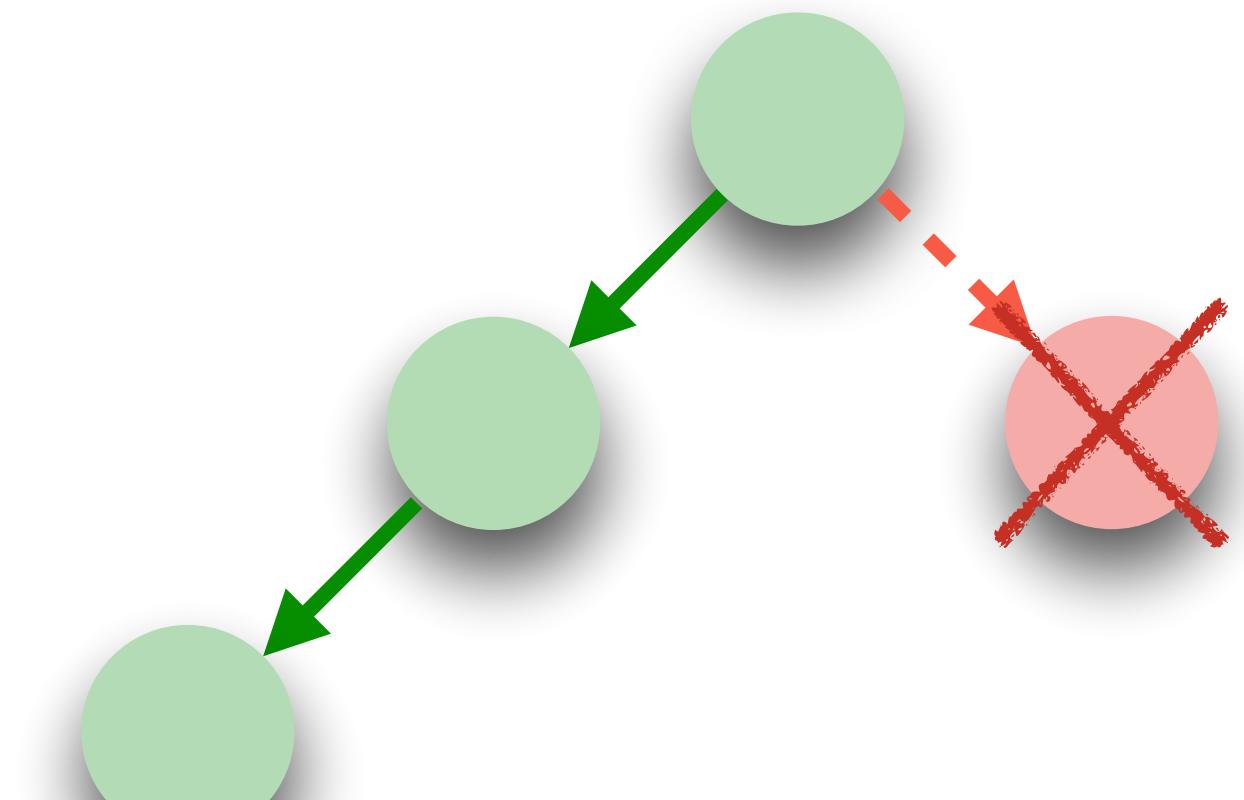
# Symbolic execution

```
1. if ( $x < A\_size$ )
2.    $y = A[x]$ 
3.    $z = B[y]$ 
4. end
```

$x \geq A\_size$



$x < A\_size$



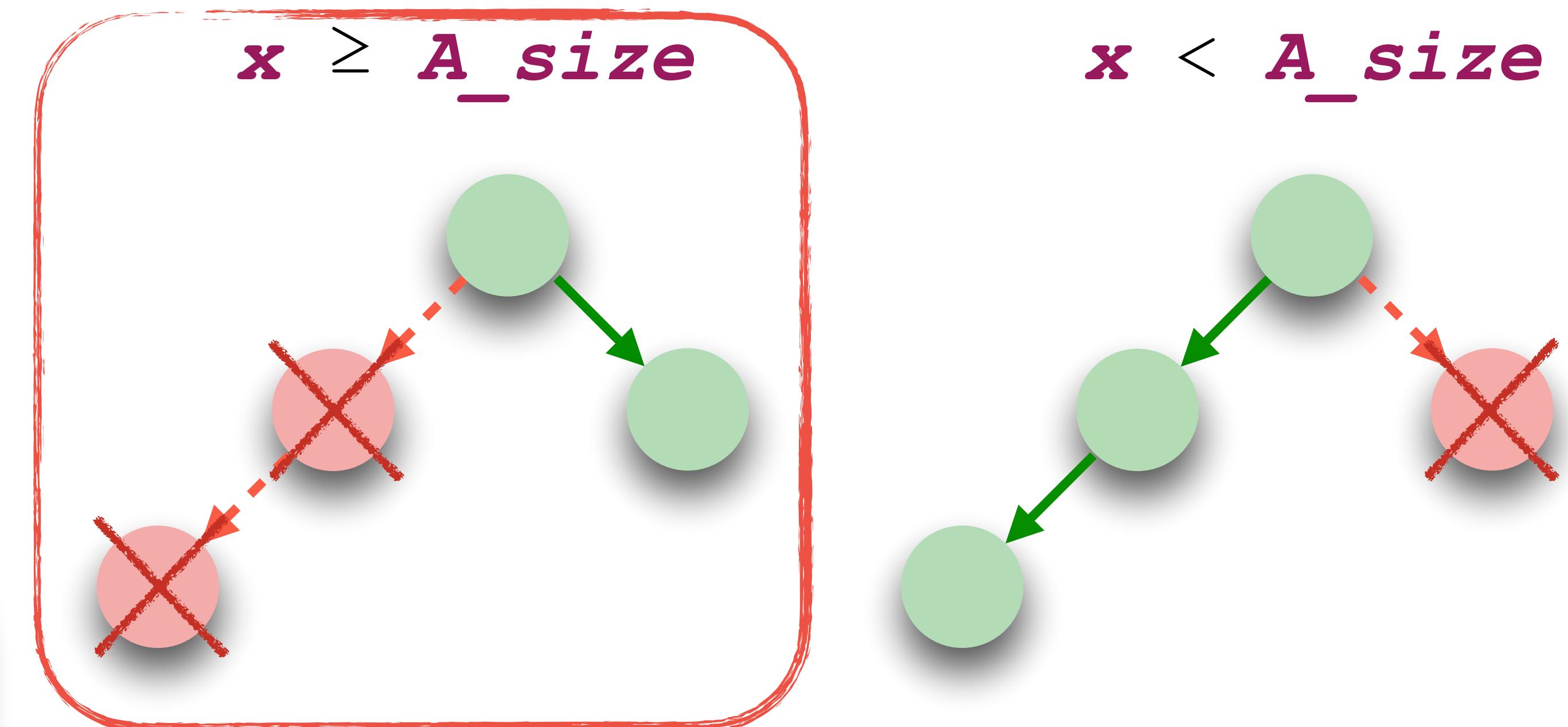
*Always mispredict*  
branch instructions

# Symbolic execution

```
1. if ( $x < A\_size$ )  
2.      $y = A[x]$   
3.      $z = B[y]$   
4. end
```



*Always mispredict*  
branch instructions

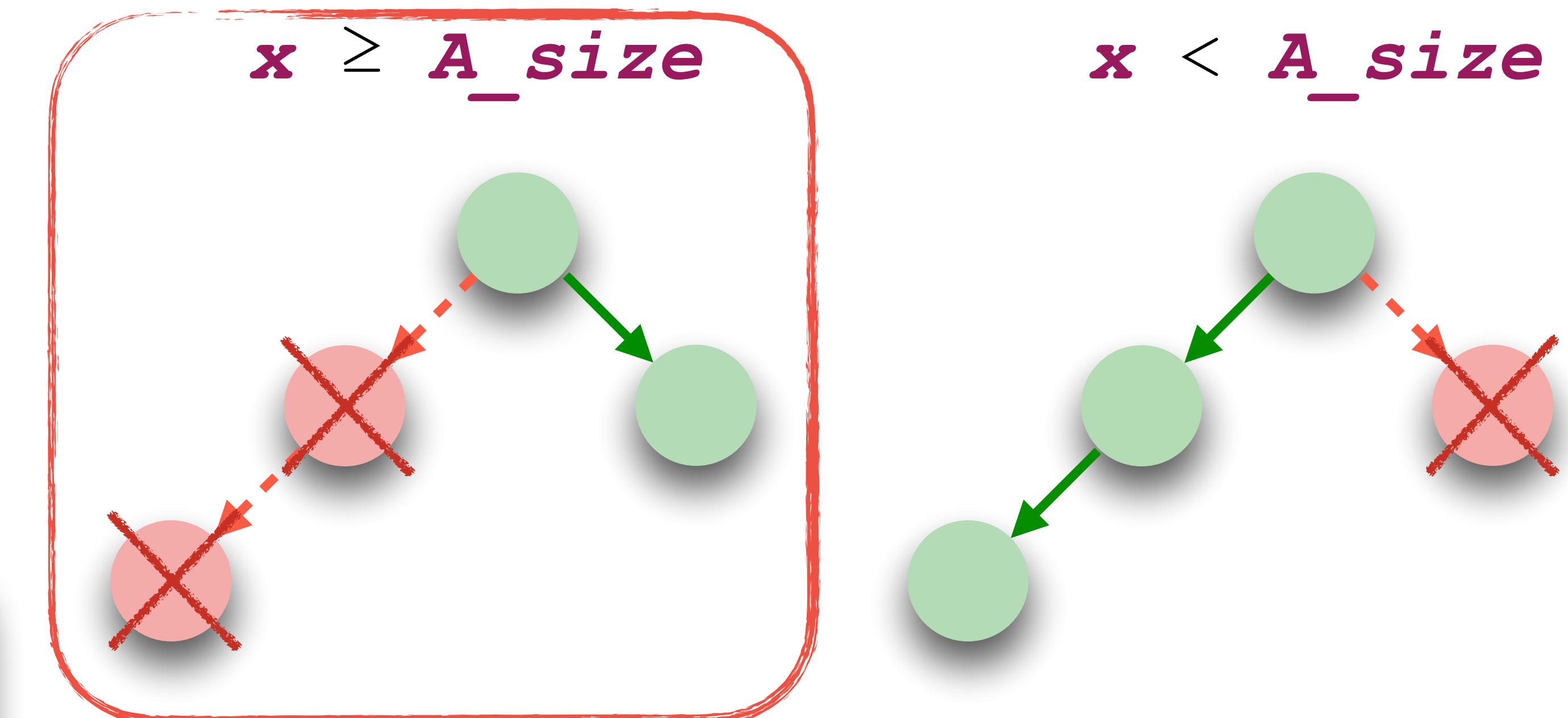


# Symbolic execution

```
1. if ( $x < A\_size$ )  
2.    $y = A[x]$   
3.    $z = B[y]$   
4. end
```



*Always mispredict*  
branch instructions



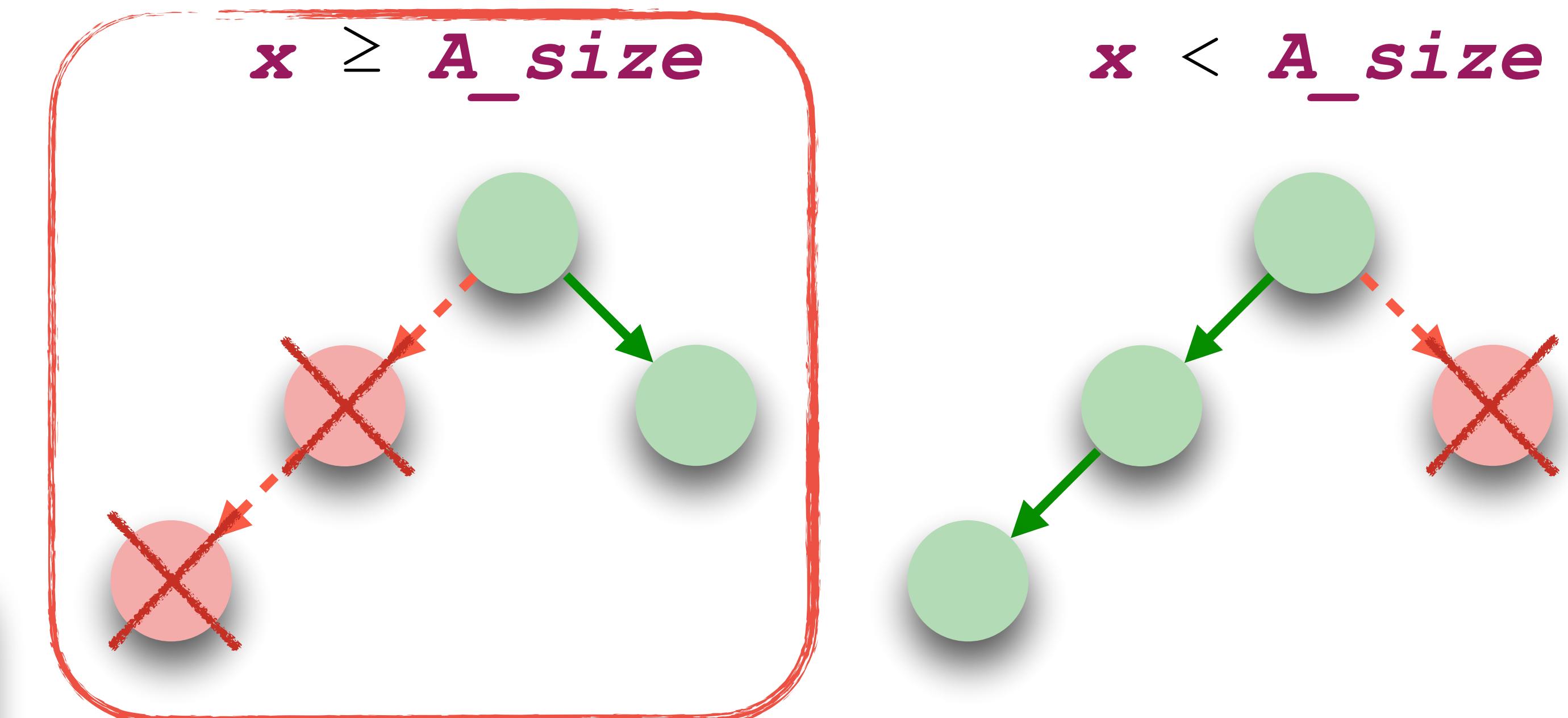
start pc 2 load  $A+x$  load  $B+A[x]$  rollback pc 4

# Symbolic execution

```
1. if ( $x < A\_size$ )  
2.    $y = A[x]$   
3.    $z = B[y]$   
4. end
```



*Always mispredict*  
branch instructions



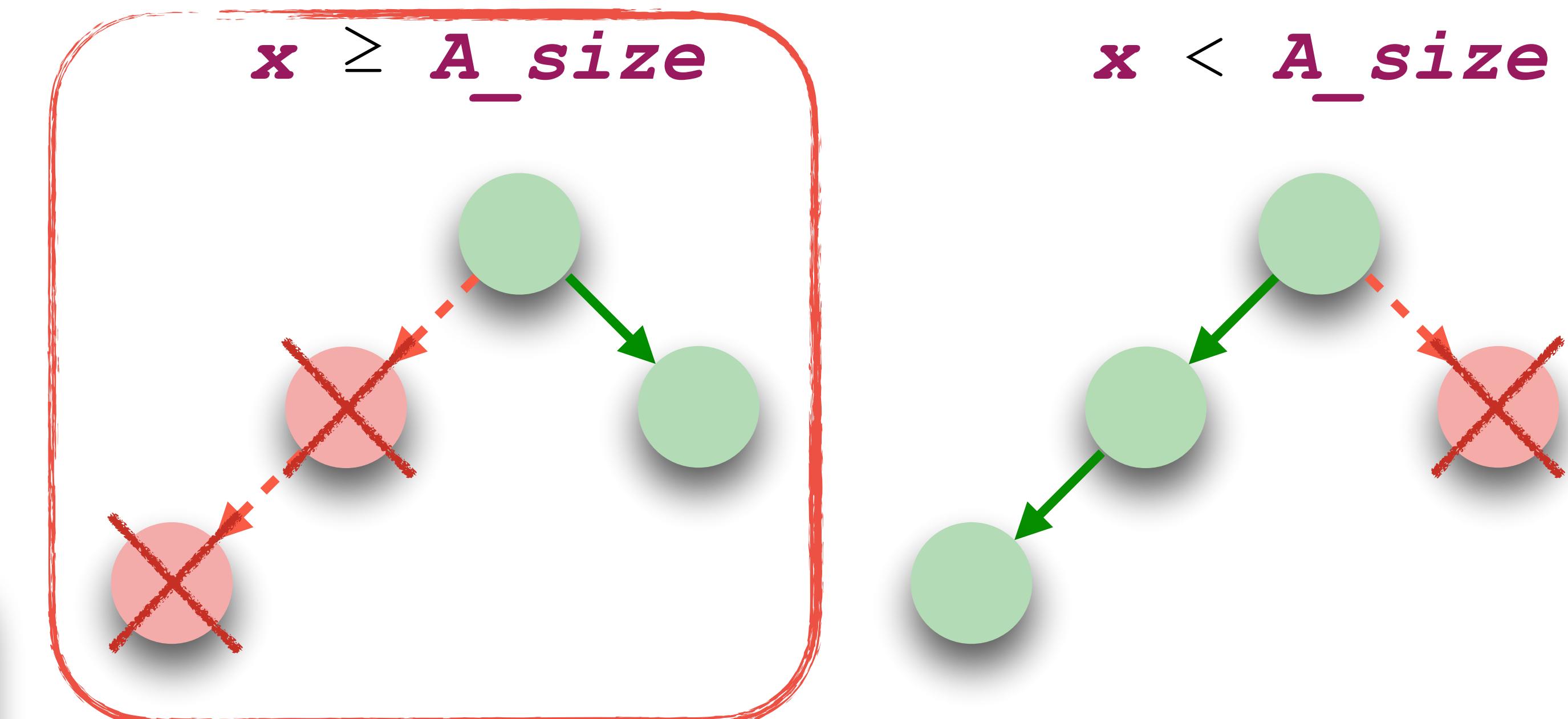
start pc 2 load  $A+x$  load  $B+A[x]$  rollback pc 4

# Symbolic execution

```
1. if ( $x < A\_size$ )  
2.      $y = A[x]$   
3.      $z = B[y]$   
4. end
```



*Always mispredict*  
branch instructions

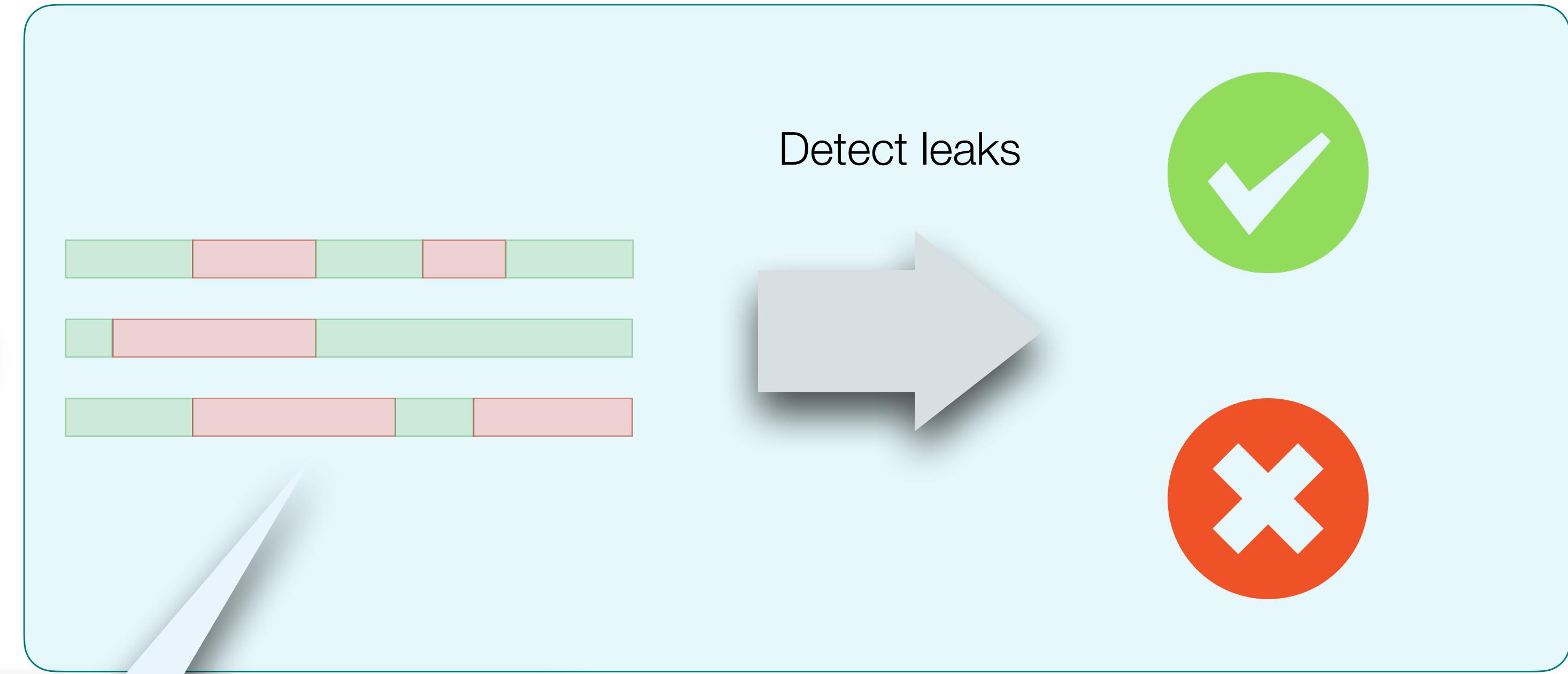


start	pc 2	load $A+x$	load $B+A[x]$	rollback	pc 4
-------	------	------------	---------------	----------	------

# Detecting speculative leaks

```
rax <- A_size  
rcx <- x  
jmp rcx>=rax, END  
L1:    load rax, A + rcx  
       load rax, B + rax  
END:
```

Symbolic execution



**Symbolic trace:** path condition +  
observations along the symbolic path

# Detecting speculative leaks

For each symbolic trace  $\tau \in traces(prg)$

```
if MemLeak( $\tau$ ) then  
    return INSECURE  
if CtrlLeak( $\tau$ ) then  
    return INSECURE  
return SECURE
```

L1:

END:

rax  
rcx  
jmp  
load  
load



# Detecting speculative leaks

For each symbolic trace  $\tau \in traces(prg)$

**if**  $MemLeak(\tau)$  **then**

**return** *INSECURE*

**if**  $CtrlLeak(\tau)$  **then**

**return** *INSECURE*

**return** *SECURE*

rax  
rcx  
jmp  
load  
load

*L1:*

*END:*



# Memory leaks

Speculative memory accesses ***must be fully determined*** by ***non-speculative*** observations

# Memory leaks

Speculative memory accesses ***must be fully determined*** by ***non-speculative*** observations

$\tau$



# Memory leaks

Speculative memory accesses ***must be fully determined*** by ***non-speculative*** observations

$\tau$



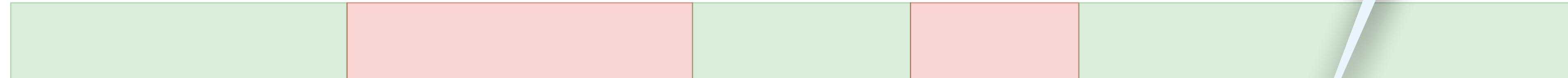
$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

# Memory leaks

Speculative memory accesses ***must be fully determined*** by ***non-speculative*** observations

Check with self-composition

$\tau$



$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

# Memory leaks

Speculative memory accesses **must be fully determined** by **non-speculative** observations

Check with self-composition

$\tau$



$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

$s_1$

$s_2$

# Memory leaks

Speculative memory accesses **must be fully determined** by **non-speculative** observations

Check with self-composition

$\tau$



$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$

$$s_1 \models \varphi$$

$$s_2 \models \varphi$$

# Memory leaks

Speculative memory accesses **must be fully determined** by **non-speculative** observations

Check with self-composition

$\tau$



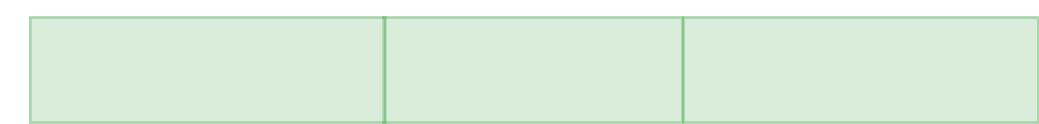
$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$

$s_1 \models \varphi$



||

$s_2 \models \varphi$



# Memory leaks

Speculative memory accesses **must be fully determined** by **non-speculative** observations

$\tau$



Check with self-composition

$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

$$s_1 \models \varphi$$



$$s_2 \models \varphi$$



# Spectector + Case studies

# Spectector



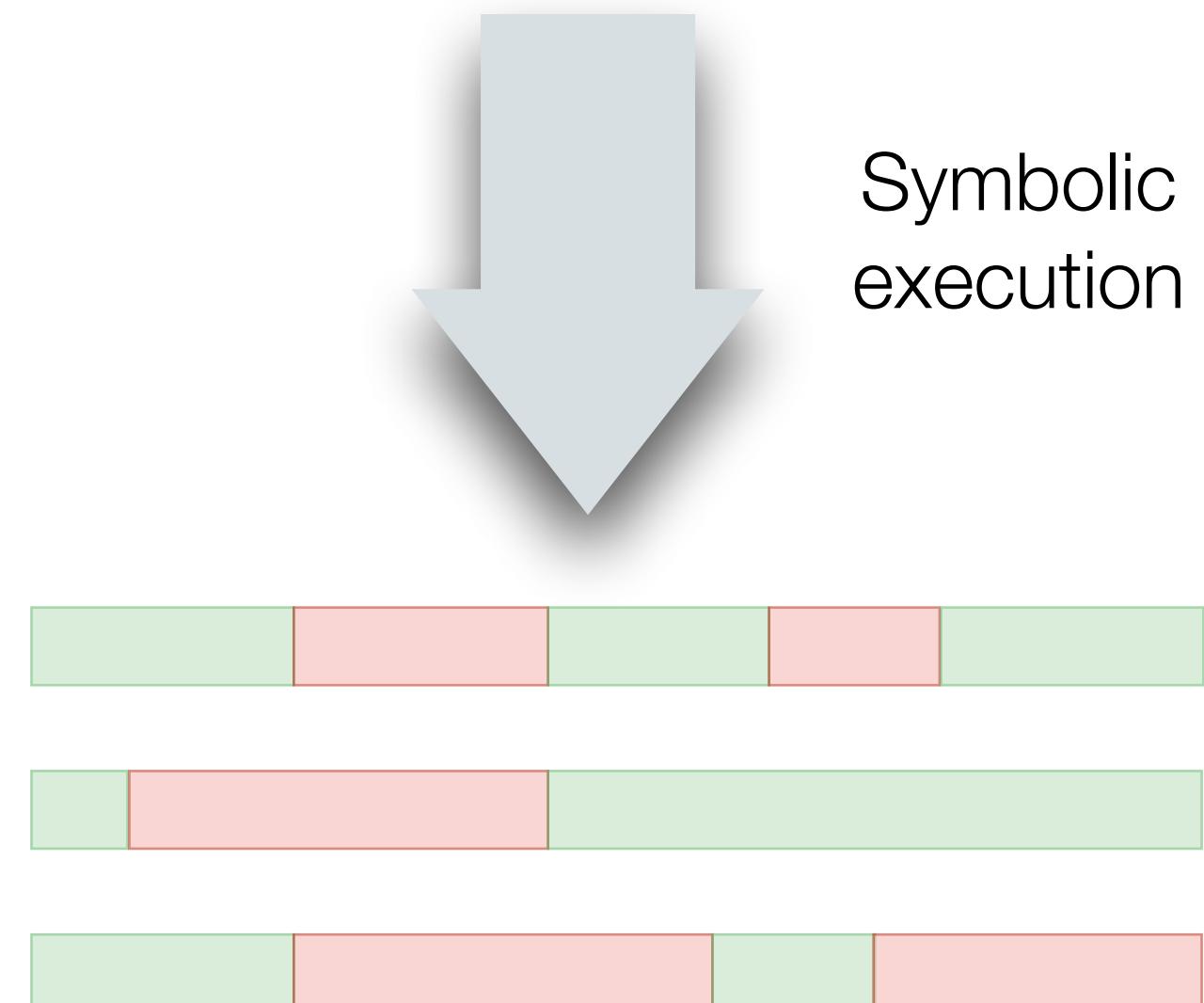
```
mov    rax, A_size
mov    rcx, x
cmp    rcx, rax
END
L1:   mov    rax, A[rcx]
      mov    rax, B[rax]
```

x64 to μASM

```
L1:   rax <- A_size
        rcx <- x
        jmp  rcx≥rax, END
END:  load  rax, A + rcx
        load  rax, B + rax
```



Check for speculative leaks



# Spectector



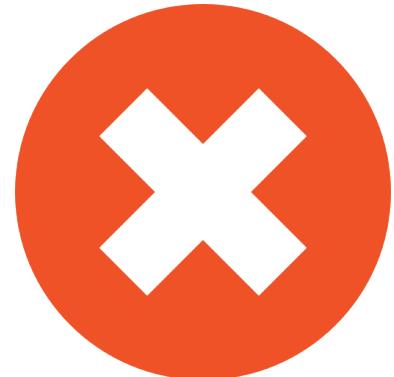
```
mov      rax, A_size  
mov      rcx, x  
cmp      rcx, rax  
jae      END  
L1: mov      rax, A  
       mov      rax, B
```

```
x64 to μASM  
mov      rax, A_size  
mov      rcx, x  
cmp      rcx, rax  
jae      END  
rax, A  
rax, B
```

## More details



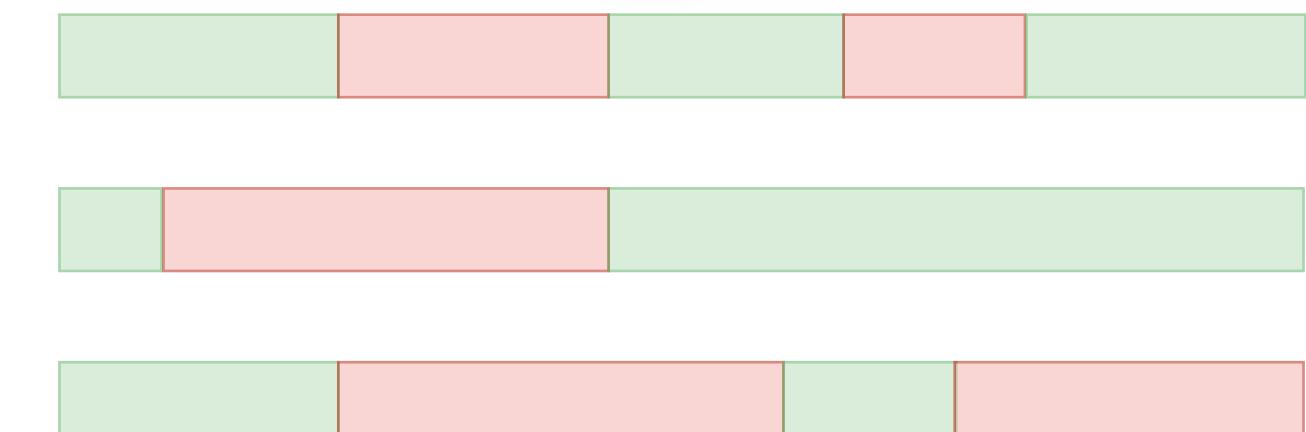
- Built in Ciao Prolog
- **Z3** for symbolic execution and leak detection



Check for speculative leaks

```
rax <- A_size  
rcx <- x  
jmp rcx>=rax, END  
load rax, A + rcx  
load rax, B + rax
```

Symbolic  
execution



# Case study: compiler mitigations

## *Target:*

- 15 variants of Spectre V1 by Paul Kocher\*
- Compiled with Microsoft Visual C++, Intel ICC, and Clang with different mitigations and optimization levels
- 240 assembly programs of up to 200 instructions each

## *How:*

- Use Spectector to prove security or detect leaks

\* Paul Kocher - Spectre Mitigations in Microsoft C/C++ Compiler – <https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html>

# Results

Ex.	VCC						ICC						CLANG					
	UNP		FEN 19.15		FEN 19.20		UNP		FEN		UNP		FEN		UNP		FEN	
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02
01	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●
02	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●
03	○	○	●	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
04	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
05	○	○	●	○	●	○	○	○	●	●	○	○	●	●	●	●	●	●
06	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
07	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
08	○	●	○	●	○	●	○	●	●	●	○	●	●	●	●	●	●	●
09	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
10	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	○
11	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
12	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
13	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
14	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
15	○	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	○	●

# Results

Ex.	VCC						ICC						CLANG					
	UNP		FEN 19.15		FEN 19.20		UNP		FEN		UNP		FEN		UNP		FEN	
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02
01	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●
02	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●
03	○	○	●	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
04	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
05	○	○	●	○	●	○	○	○	●	●	○	○	●	●	●	●	●	●
06	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
07	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
08	○	●	○	●	○	●	○	●	●	●	○	●	●	●	●	●	●	●
09	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
10	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	○
11	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
12	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
13	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
14	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
15	○	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	○	●

# Results

Ex.	VCC				ICC				CLANG					
	UNP		FEN 19.15		FEN 19.20		UNP		FEN		UNP		FEN	
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02
01	○	○	●	●	●	●	○	○	●	●	○	○	●	●
02	○	○	●	●	●	●	○	○	●	●	○	○	●	●
03	○	○	●	○	●	●	○	○	●	●	○	○	●	●
04	○	○	○	○	●	●	○	○	●	●	○	○	●	●
05	○	○	●	○	●	○	○	○	●	●	○	○	●	●
06	○	○	○	○	○	○	○	○	●	●	○	○	●	●
07	○	○	○	○	○	○	○	○	●	●	○	○	●	●
08	○	●	○	●	○	●	○	●	●	●	○	●	●	●
09	○	○	○	○	○	○	○	○	●	●	○	○	●	●
10	○	○	○	○	○	○	○	○	●	●	○	○	●	●
11	○	○	○	○	○	○	○	○	●	●	○	○	●	●
12	○	○	○	○	●	●	○	○	●	●	○	○	●	●
13	○	○	○	○	○	○	○	○	●	●	○	○	●	●
14	○	○	○	○	●	●	○	○	●	●	○	○	●	●
15	○	○	○	○	○	○	○	○	●	●	○	○	●	●

# Results

Ex.	VCC						ICC						CLANG					
	UNP		FEN 19.15		FEN 19.20		UNP		FEN		UNP		FEN		SLH			
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	
01	o	o	●	●	●	●	o	o	●	●	o	o	●	●	●	●	●	
02	o	o	●	●	●	●	o	o	●	●	o	o	●	●	●	●	●	
03	o	o	●	o	●	●	o	o	●	●	o	o	●	●	●	●	●	
04	o	o	o	o	●	●	o	o	●	●	o	o	●	●	●	●	●	
05	o	o	●	o	●	o	o	o	●	●	o	o	●	●	●	●	●	
06	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
07	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
08	o	●	o	●	o	●	o	●	●	●	o	●	●	●	●	●	●	
09	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
10	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	o	
11	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
12	o	o	o	o	●	●	o	o	●	●	o	o	●	●	●	●	●	
13	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
14	o	o	o	o	●	●	o	o	●	●	o	o	●	●	●	●	●	
15	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	o	●	

# Results

No countermeasures

Ex.	VCC						ICC						CLANG					
	UNP	FEN	19.15	FEN	19.20	UNP	FEN	UNP	FEN	UNP	FEN	SLH	-00	-02	-00	-02	-00	-02
01	○	○	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●	●
02	○	○	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●	●
03	○	○	●	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
04	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
05	○	○	●	○	●	○	○	●	●	○	○	●	●	●	●	●	●	●
06	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●	●
07	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●	●
08	○	●	○	●	○	●	○	●	●	○	○	●	●	●	●	●	●	●
09	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●	●
10	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●	○
11	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●	●
12	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
13	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●	●
14	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
15	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	○	●

# Results

Automated insertion of fences

Ex.	VCC					ICC					CLANG				
	UNP	FEN 19.15	FEN 19.20	UNP	FEN	UNP	FEN	UNP	FEN	SLH					
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	
01	o	o	●	●	●	o	o	●	●	●	●	●	●	●	●
02	o	o	●	●	●	o	o	●	●	o	o	●	●	●	●
03	o	o	●	o	●	●	o	●	●	o	o	●	●	●	●
04	o	o	o	o	●	●	o	●	●	o	o	●	●	●	●
05	o	o	●	o	●	o	o	●	●	o	o	●	●	●	●
06	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●
07	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●
08	o	●	o	●	o	●	o	●	●	o	●	●	●	●	●
09	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●
10	o	o	o	o	o	o	o	●	●	o	o	●	●	●	o
11	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●
12	o	o	o	o	●	●	o	●	●	o	o	●	●	●	●
13	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●
14	o	o	o	o	●	●	o	o	●	o	o	●	●	●	●
15	o	o	o	o	o	o	o	o	●	●	o	●	●	o	●

# Results

Speculative load  
hardening

Ex.	VCC						ICC						CLANG					
	UNP	FEN	19.15	FEN	19.20	UNP	FEN	UNP	FEN	UNP	FEN	UNP	FEN	SLH				
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02
01	○	○	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●	●
02	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●
03	○	○	●	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
04	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
05	○	○	●	○	●	○	○	○	●	●	○	○	●	●	●	●	●	●
06	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
07	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
08	○	●	○	●	○	●	○	●	●	●	○	●	●	●	●	●	●	●
09	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
10	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	○
11	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
12	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
13	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
14	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
15	○	○	○	○	○	○	○	○	○	●	●	○	●	●	●	●	○	●

# Results

Ex.	VCC						ICC						CLANG					
	UNP		FEN 19.15		FEN 19.20		UNP		FEN		UNP		FEN		SLH			
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	
01	o	o	●	●	●	●	o	o	●	●	o	o	●	●	●	●	●	
02	o	o	●	●	●	●	o	o	●	●	o	o	●	●	●	●	●	
03	o	o	●	o	●	●	o	o	●	●	o	o	●	●	●	●	●	
04	o	o	o	o	●	●	o	o	●	●	o	o	●	●	●	●	●	
05	o	o	●	o	●	o	o	o	●	●	o	o	●	●	●	●	●	
06	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
07	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
08	o	●	o	●	o	●	o	●	●	●	o	●	●	●	●	●	●	
09	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
10	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	o	
11	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
12	o	o	o	o	●	●	o	o	●	●	o	o	●	●	●	●	●	
13	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
14	o	o	o	o	●	●	o	o	●	●	o	o	●	●	●	●	●	
15	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	o	●	

# Results

Ex.	VCC						ICC						CLANG					
	UNP		FEN 19.15		FEN 19.20		UNP		FEN		UNP		FEN		UNP		FEN	
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02
01	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●
02	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●
03	○	○	●	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
04	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
05	○	○	●	○	●	○	○	○	●	●	○	○	●	●	●	●	●	●
06	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
07	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
08	○	●	○	●	○	●	○	●	●	●	○	●	●	●	●	●	●	●
09	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
10	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	○
11	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
12	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
13	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
14	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
15	○	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	○	●

# Results

Ex.	VCC			ICC			CLANG		
	UNP		FEN	19.15			UNP	FEN	SLH
	-00	-02							
01	o	o							•
02	o	o							•
03	o	o							•
04	o	o							•
05	o	o							•
06	o	o							•
07	o	o							•
08	o	•							•
09	o	o							•
10	o	o							•
11	o	o							•
12	o	o							•
13	o	o							•
14	o	o	o	o	•	•	o	o	•
15	o	o	o	o	o	o	o	o	•

## Summary

- Leaks in all unprotected programs (except example #08 with optimizations)
- Confirm all vulnerabilities in VCC pointed out by Paul Kocher
- Programs with fences (ICC and Clang) are secure
- Unnecessary fences
- Programs with SLH are secure except #10 and #15

# Case study: scalability

**Target:** Xen hypervisors

***Main challenges for scalability:***

- Policy definition
- ISA coverage
- Path explosion

**How:**

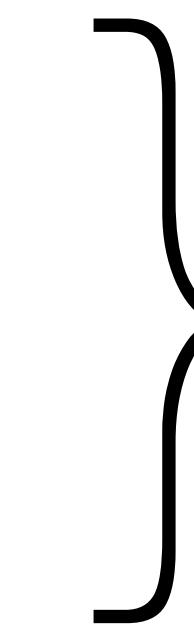
- Analyze scalability of checking SNI ***relative to*** symbolic execution
  - 24'000 symbolic paths of < 10'000 instructions (from ~ 4'000 functions)

# Case study: scalability

**Target:** Xen hypervisors

## ***Main challenges for scalability:***

- Policy definition
- ISA coverage
- Path explosion

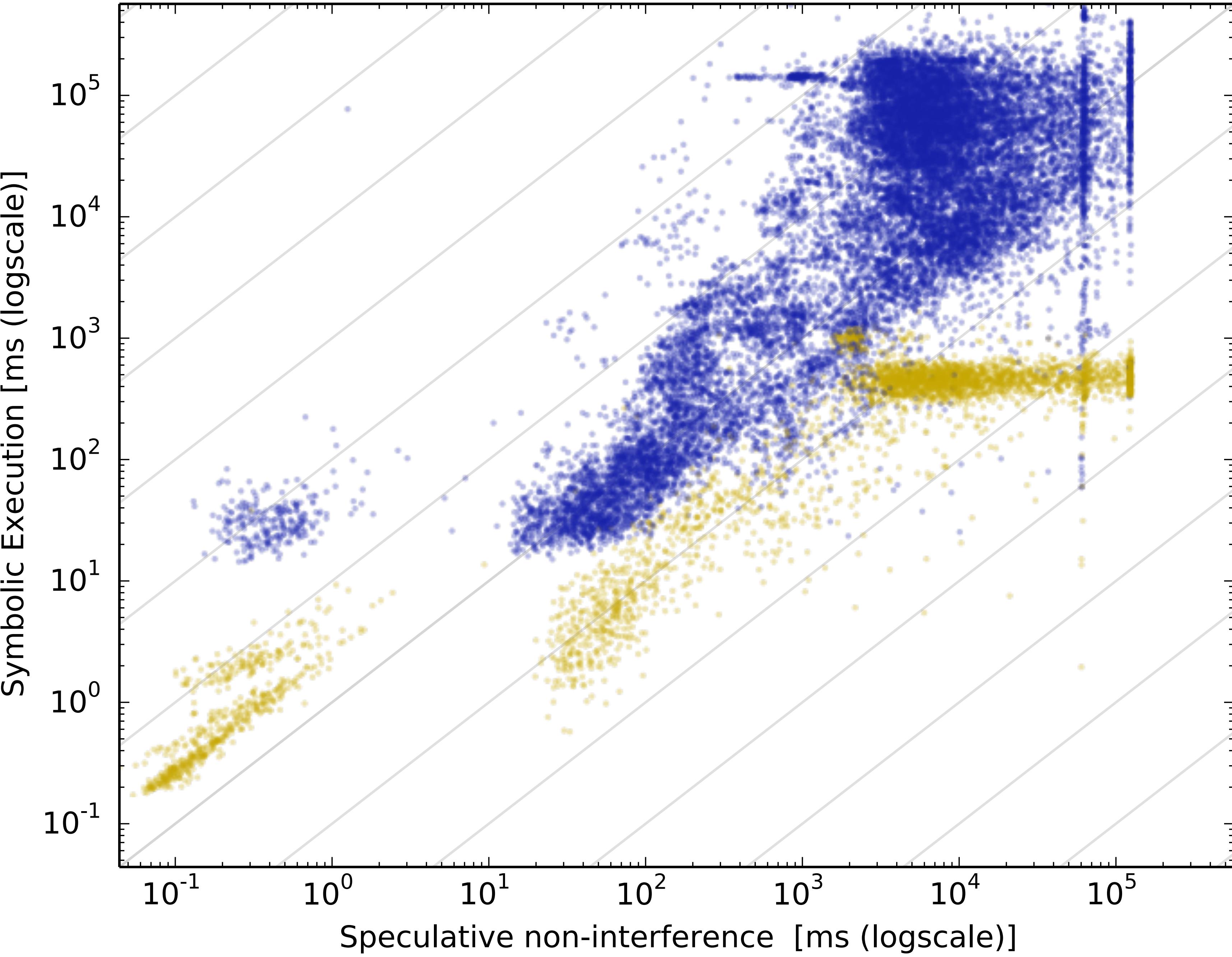


Trade-offs affect analysis soundness and completeness

## ***How:***

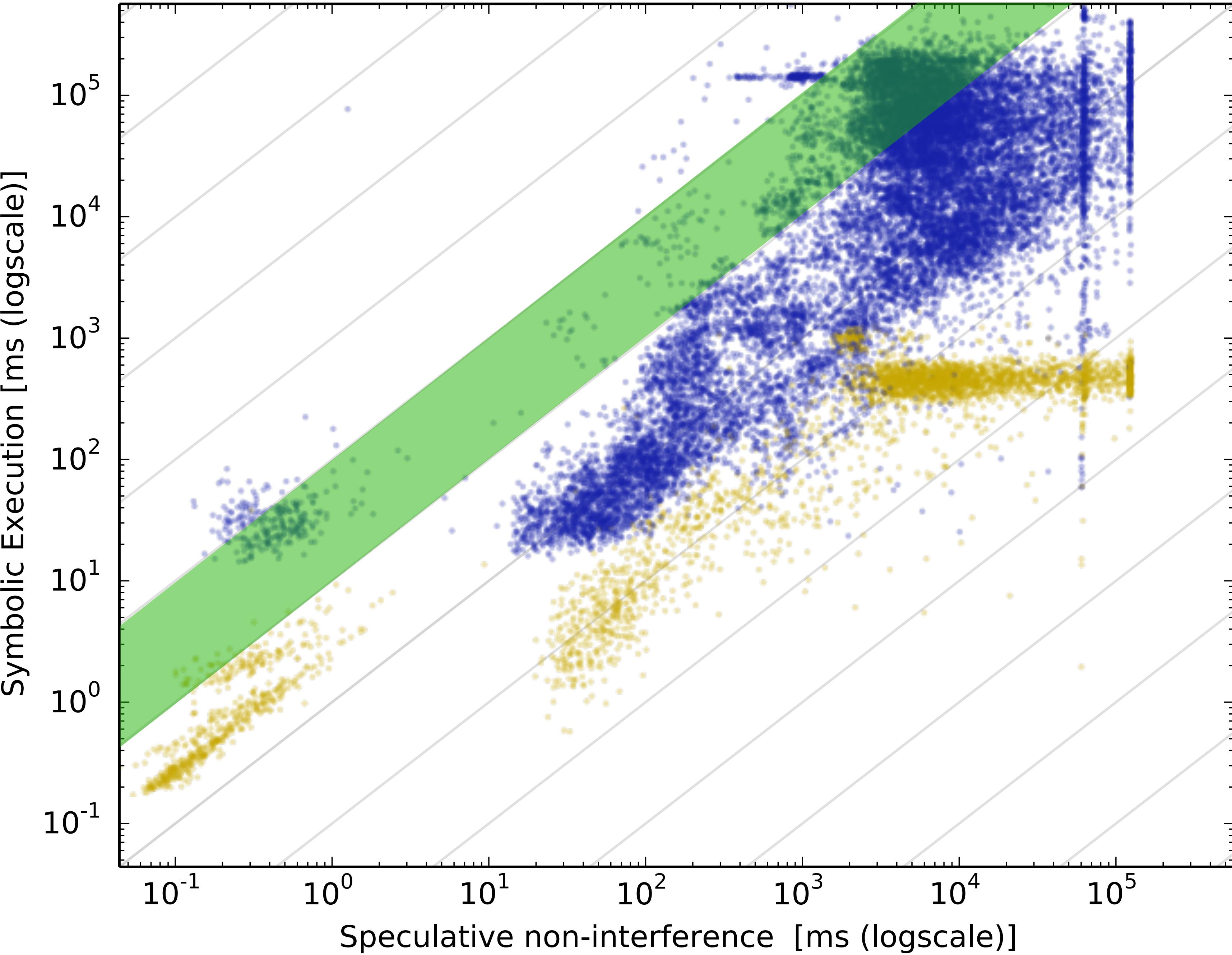
- Analyze scalability of checking SNI ***relative to*** symbolic execution
  - 24'000 symbolic paths of < 10'000 instructions (from ~ 4'000 functions)

# Results



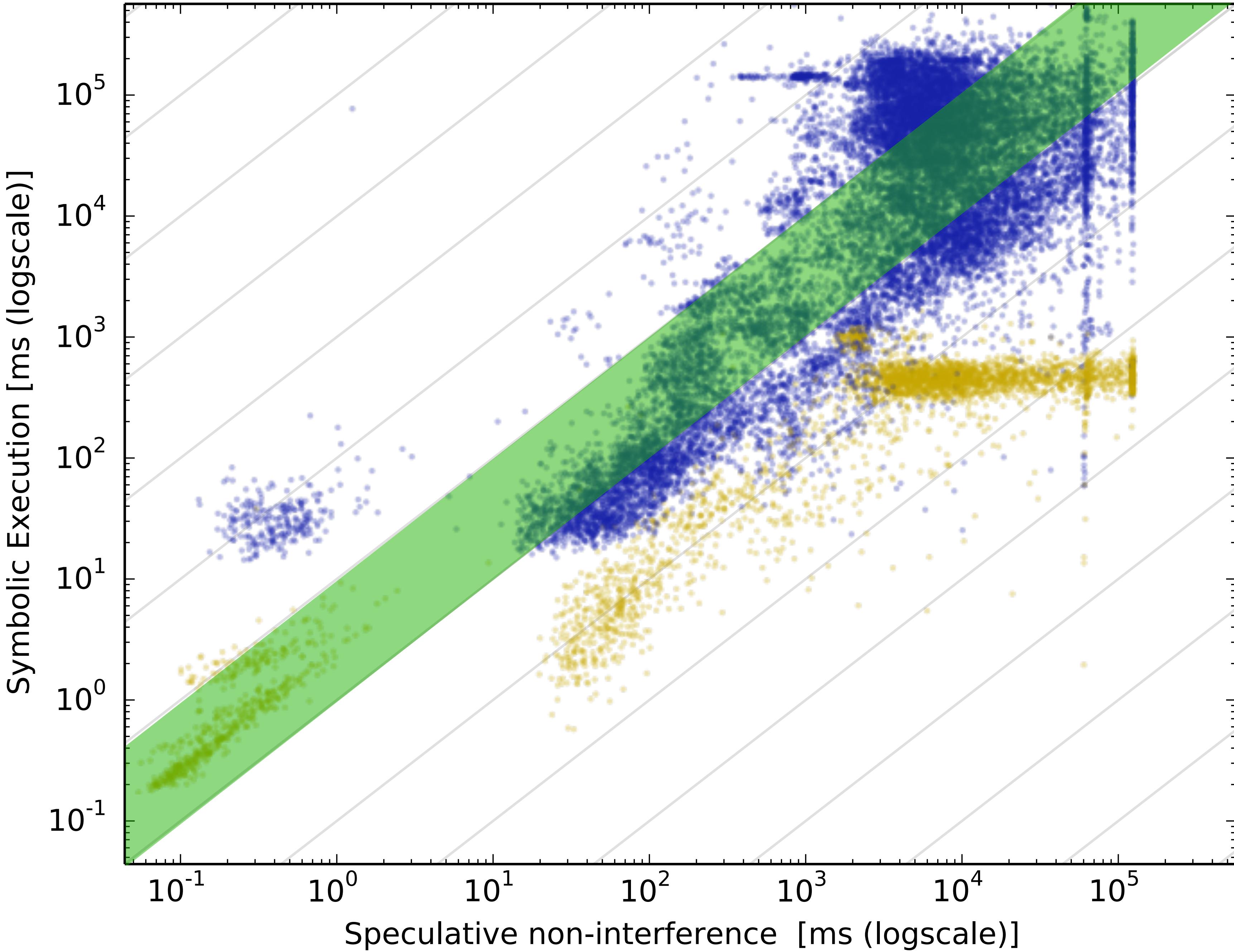
# Results

- SNI 10x-100x faster
  - 20.2% traces



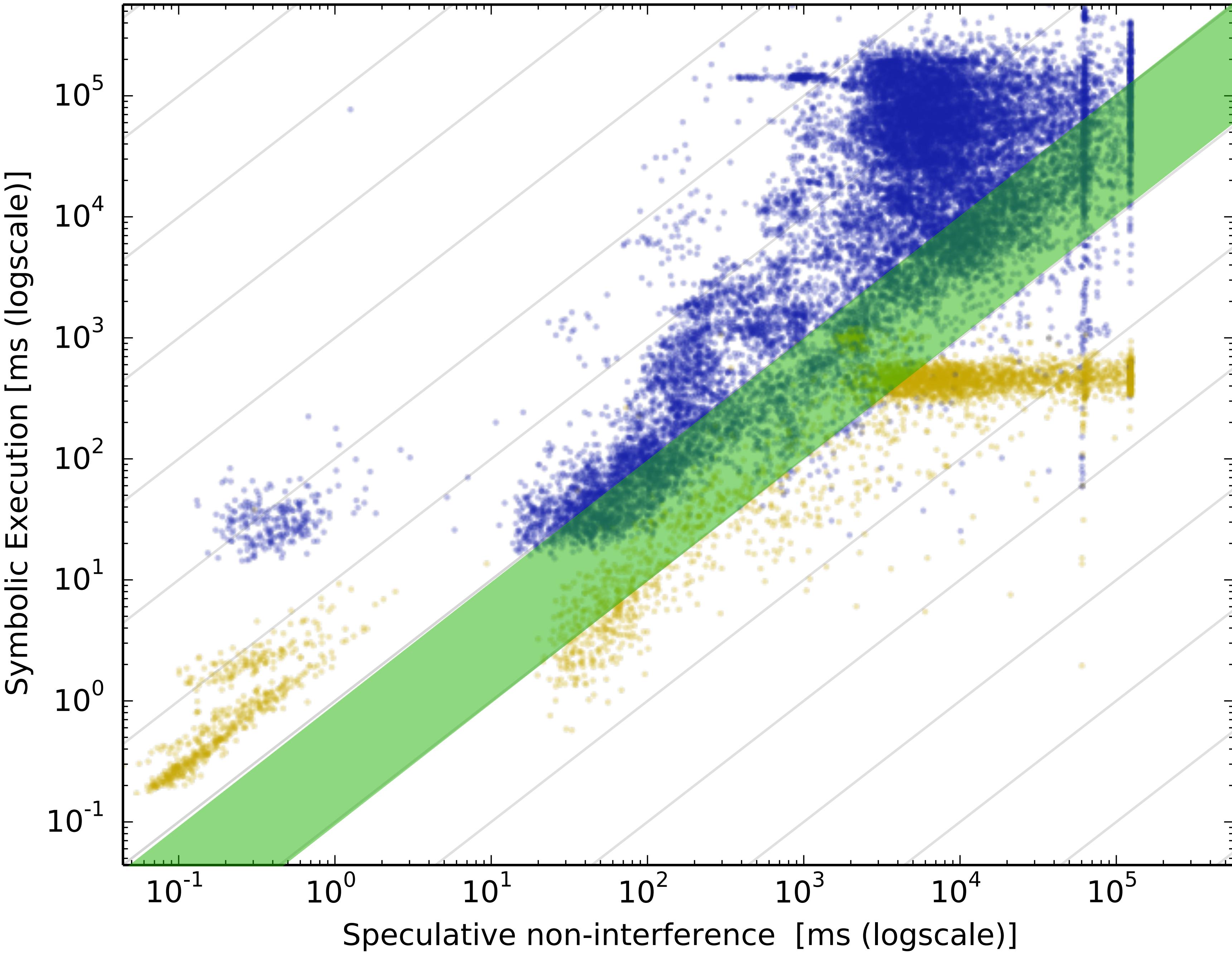
# Results

- SNI 10x-100x faster
  - 20.2% traces
- SNI  $\leq$ 10x faster
  - 41.9% traces



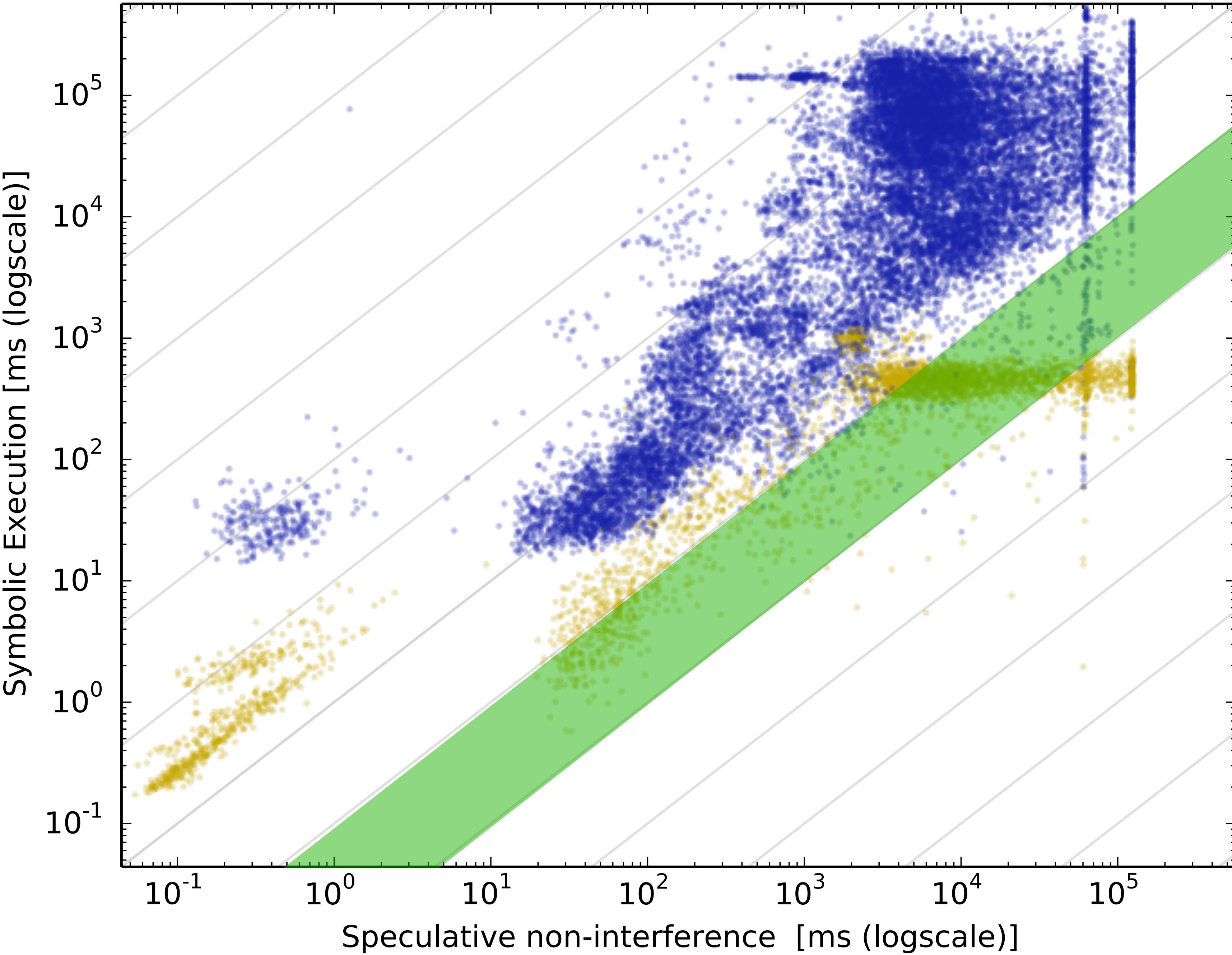
# Results

- SNI 10x-100x faster
  - 20.2% traces
- SNI  $\leq$ 10x faster
  - 41.9% traces
- SNI  $\leq$ 10x slower
  - 26.9% traces



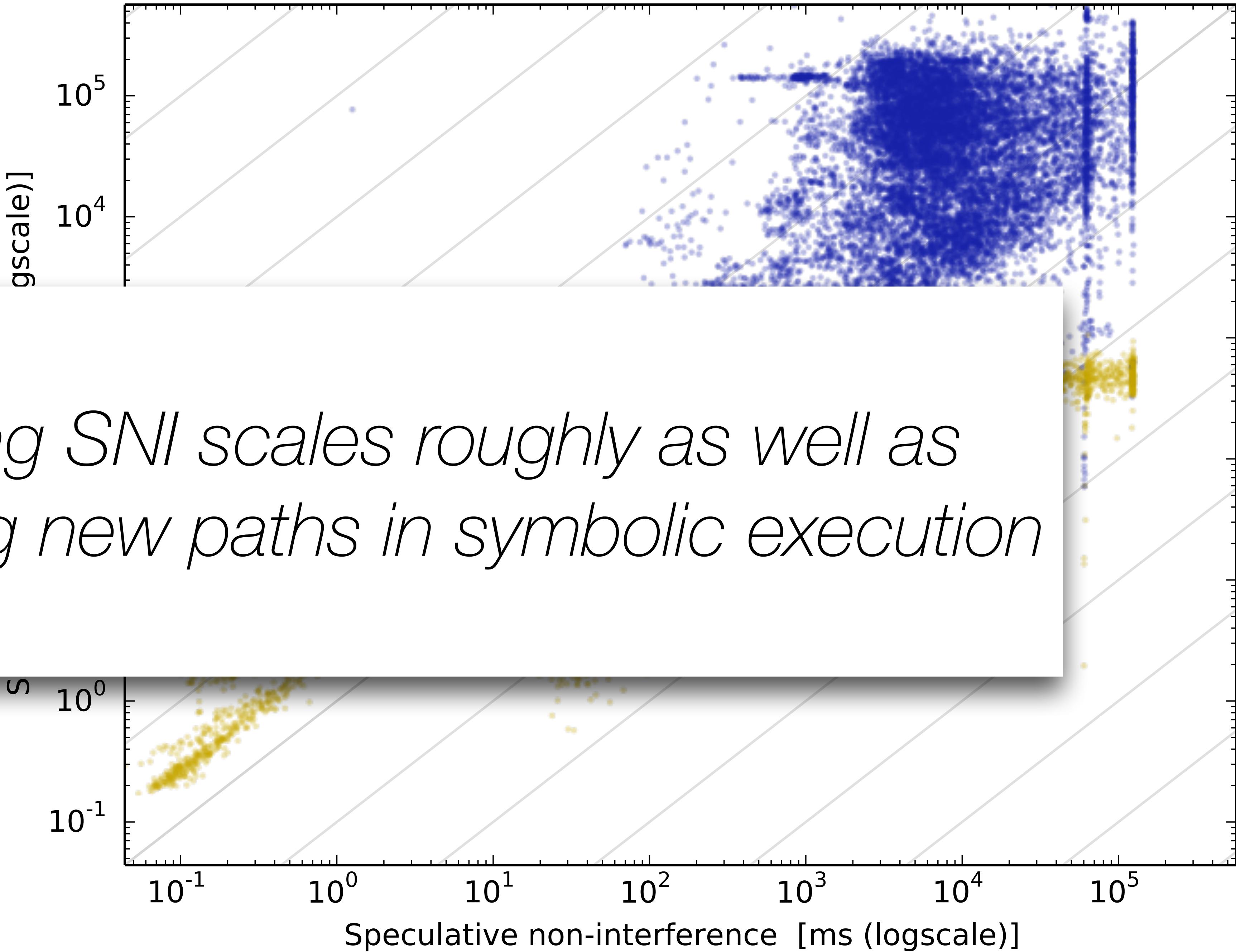
# Results

- SNI 10x-100x faster
  - 20.2% traces
- SNI  $\leq$ 10x faster
  - 41.9% traces
- SNI  $\leq$ 10x slower
  - 26.9% traces
- SNI 10x-100x slower
  - 7.9% traces



# Results

- SNI 10x-100x faster
- 20.9% traces
- SNI scales roughly as well as discovering new paths in symbolic execution
- 41.1% traces
- SNI 10x-100x slower
- 7.9% traces



# Conclusion

# Speculative non-interference

Formally!

Program **P** is **speculatively non-interferent** for prediction oracle **O** if

For all program states  $s$  and  $s'$ :

$$\begin{aligned} P_{\text{non-spec}}(s) &= P_{\text{non-spec}}(s') \\ \Rightarrow P_{\text{spec}}(s, O) &= P_{\text{spec}}(s', O) \end{aligned}$$

See paper for: reasoning about *arbitrary prediction oracles*

## Results

Ex.	VCC				ICC				CLANG							
	UNP		FEN 19.15		FEN 19.20		UNP		FEN		UNP		FEN		SLH	
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02
01	o	o	•	•	•	•	o	o	•	•	o	o	•	•	•	•
02	o	o	•	•	•	•	o	o	•	•	o	o	•	•	•	•
03	o	o	•	o	•	•	o	o	•	•	o	o	•	•	•	•
04	o	o	o	o	•	•	o	o	•	•	o	o	•	•	•	•
05	o	o	•	o	•	•	o	o	•	•	o	o	•	•	•	•
06	o	o	o	o	o	o	o	o	•	•	o	o	•	•	•	•
07	o	o	o	o	o	o	o	o	•	•	o	o	•	•	•	•
08	o	•	o	•	o	•	o	•	•	•	o	•	•	•	•	•
09	o	o	o	o	o	o	o	o	•	•	o	o	•	•	•	•
10	o	o	o	o	o	o	o	o	•	•	o	o	•	•	•	o
11	o	o	o	o	o	o	o	o	•	•	o	o	•	•	•	•
12	o	o	o	o	•	•	o	o	•	•	o	o	•	•	•	•
13	o	o	o	o	o	o	o	o	•	•	o	o	•	•	•	•
14	o	o	o	o	•	•	o	o	•	•	o	o	•	•	•	•
15	o	o	o	o	o	o	o	o	•	•	o	o	•	•	o	•

# Spectector



```

mov    rax, A_size
mov    rcx, x
cmp    rcx, rax
jae    END
L1:   mov    rax, A[rcx]
      mov    rax, B[rax]
  
```

x64 to µASM

L1:

END:

```

rax <- A_size
rcx <- x
jmp rcx>=rax, END
load rax, A + rcx
load rax, B + rax
  
```

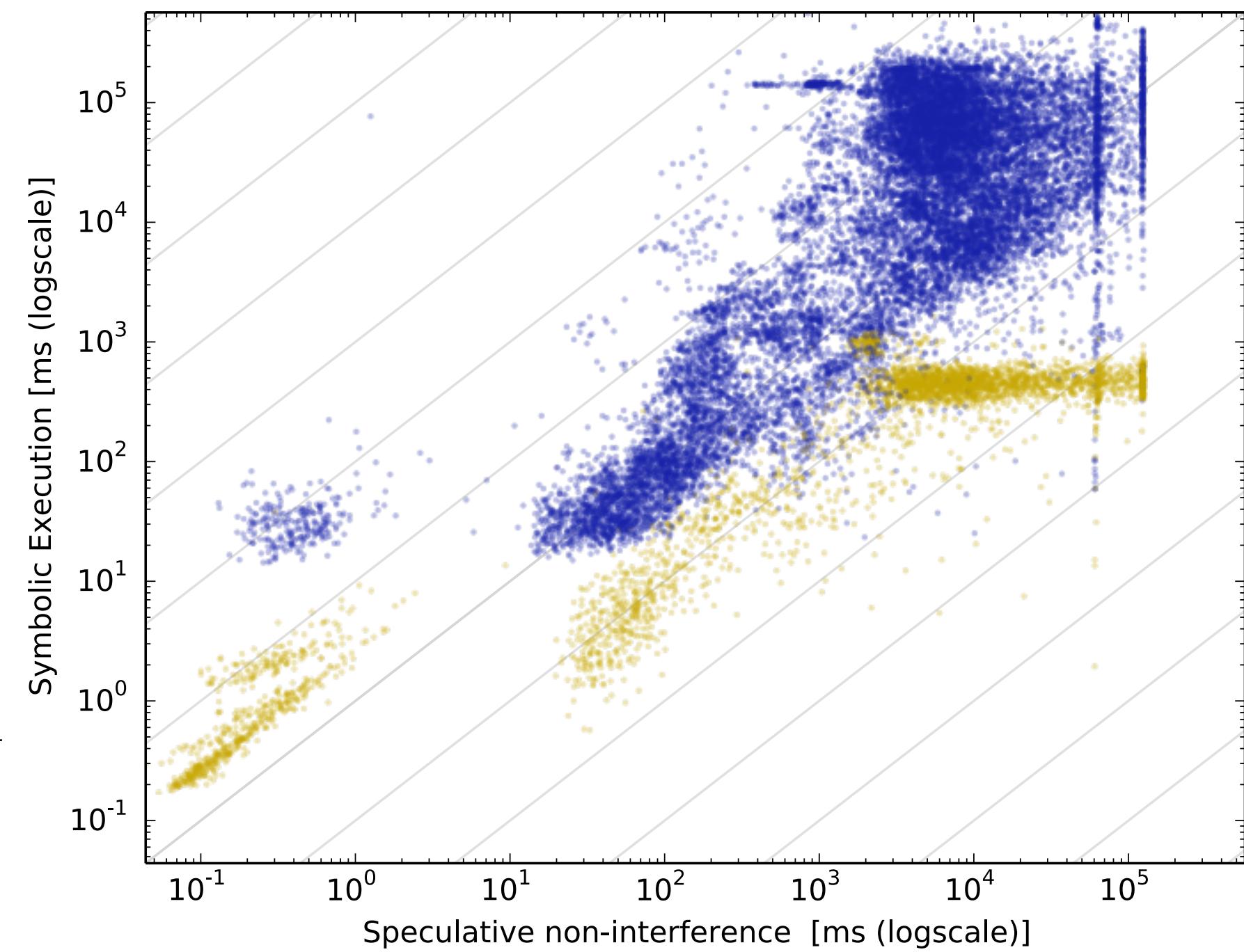
Symbolic  
execution



Check for speculative leaks

## Results

- SNI 10x-100x faster
  - 20.2% traces
- SNI  $\leq 10x$  faster
  - 41.9% traces
- SNI  $\leq 10x$  slower
  - 26.9% traces
- SNI 10x-100x slower
  - 7.9% traces



# Speculative non-interference

Program **P** is **speculatively non-interferent** for prediction oracle **O** if

Formally!

For all

$P_{nc}$

See paper for: [https://arxiv.org/abs/1802.07433](#)

## Results

Ex.	VCC			
	UNP	FEN	19.15	
-00	-02	-00	-02	
01	o	o	•	•
02	o	o	•	•
03	o	o	•	o
04	o	o	o	o
05	o	o	•	o
06	o	o	o	o
07	o	o	o	o
08	o	•	o	•
09	o	o	o	o
10	o	o	o	o
11	o	o	o	o
12	o	o	o	o
13	o	o	o	o
14	o	o	o	•
15	o	o	o	o



# Spectector

```
mov    rax, A_size
mov    rcx, x
cmp    rcx, rax
jae    END
L1: mov    rax, A[rcx]
```

x64 to µASM

*L1:*

*END*.

```
rax <- A_size
rcx <- x
jmp rcx>=rax, END
load rax, A + rcx
load rax, B + rax
```

# Spectector



<https://spectector.github.io>

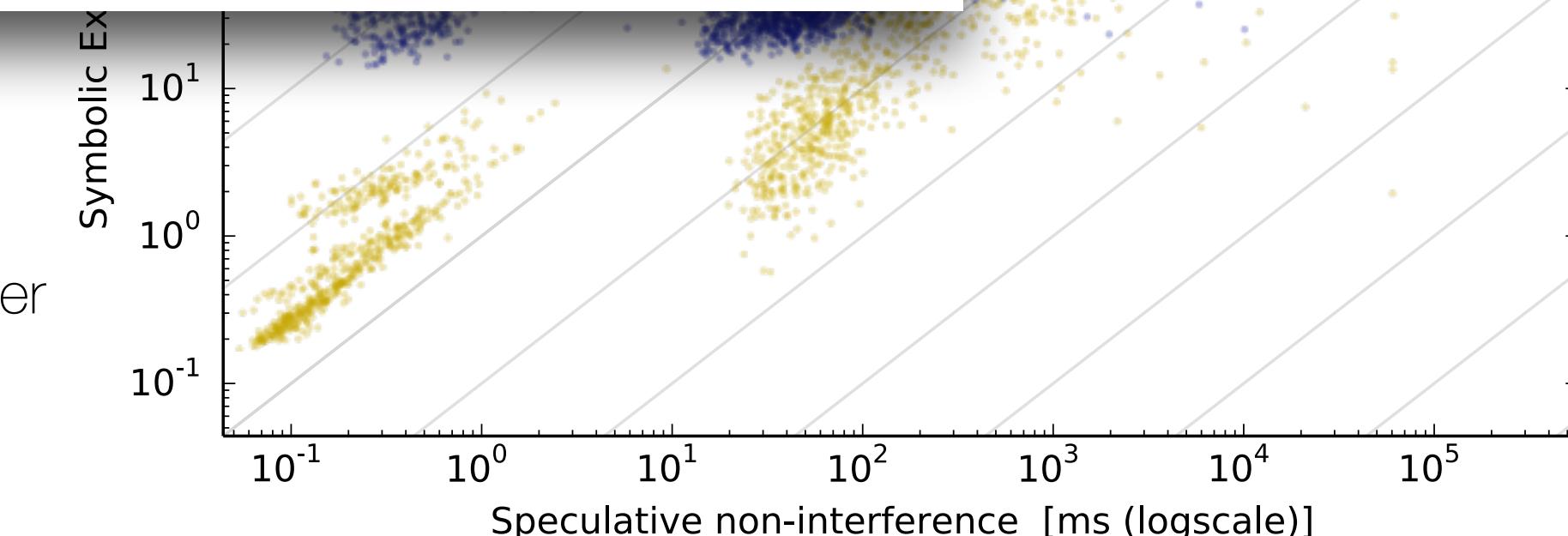


marco.guarnieri@imdea.org

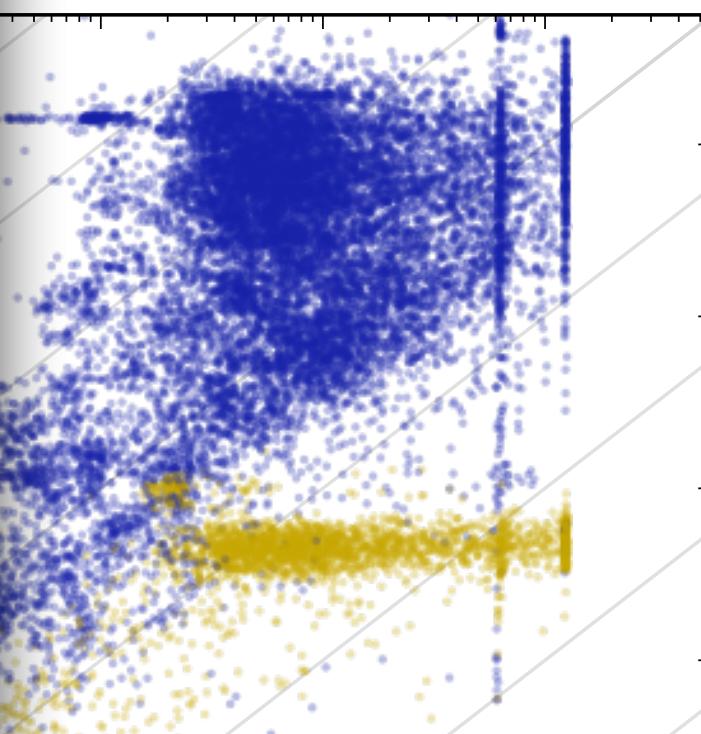
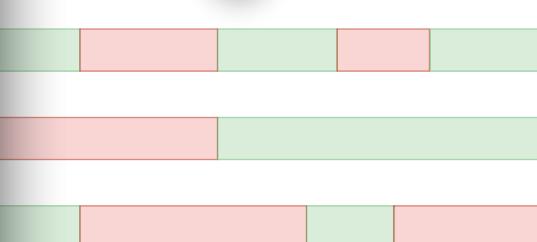


@MarcoGuarnier1

- SNI  $\leq 10x$  slower
- 26.9% traces
- SNI 10x-100x slower
- 7.9% traces



Symbolic execution



# Backup

# Speculative non-interference

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```

# Speculative non-interference

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



# Speculative non-interference

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```

**x**=128  
**A\_size**=16  
**A**[128]=**0**

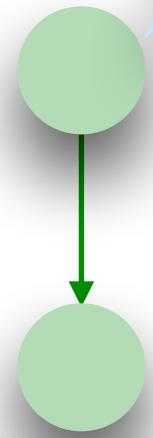
**x**=128  
**A\_size**=16  
**A**[128]=**1**

# Speculative non-interference

$x=128$   
 $A\_size=16$   
 $\mathbf{A}[128]=1$

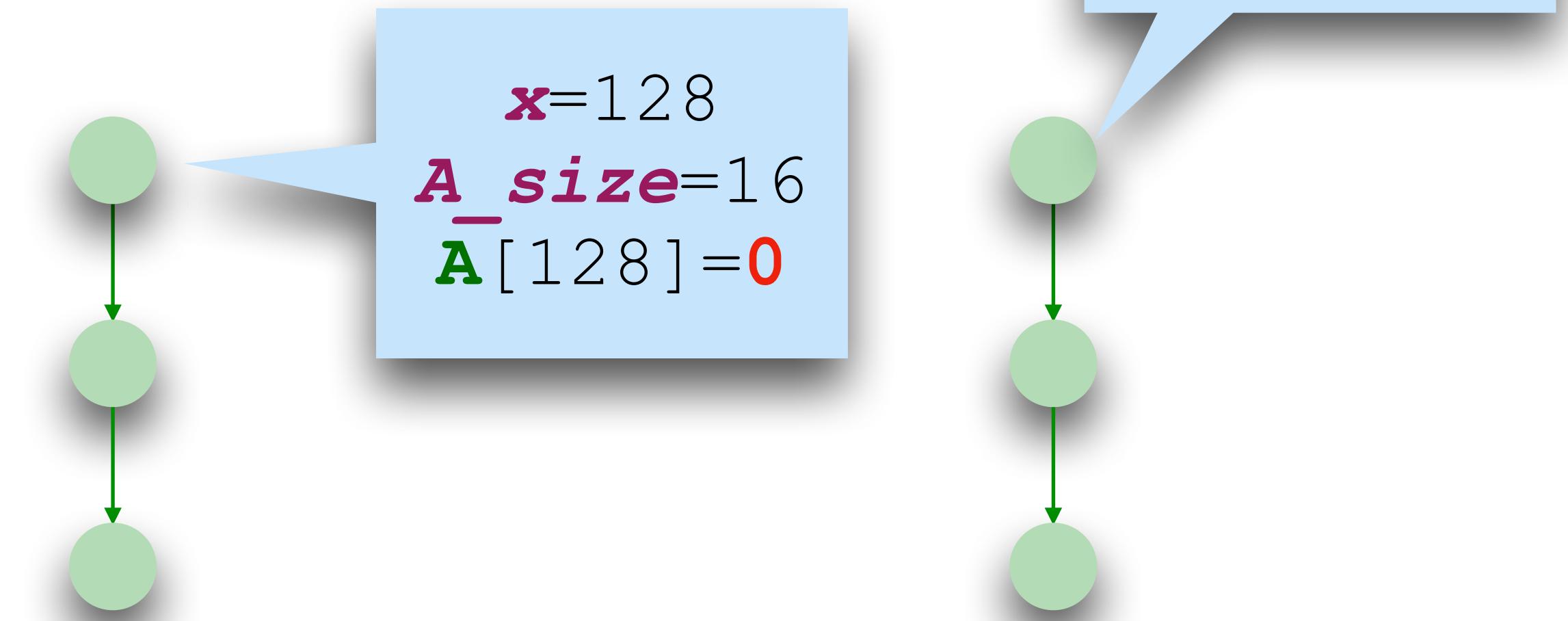
```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
     load rax, B + rax
END:
```

$x=128$   
 $A\_size=16$   
 $\mathbf{A}[128]=0$



# Speculative non-interference

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



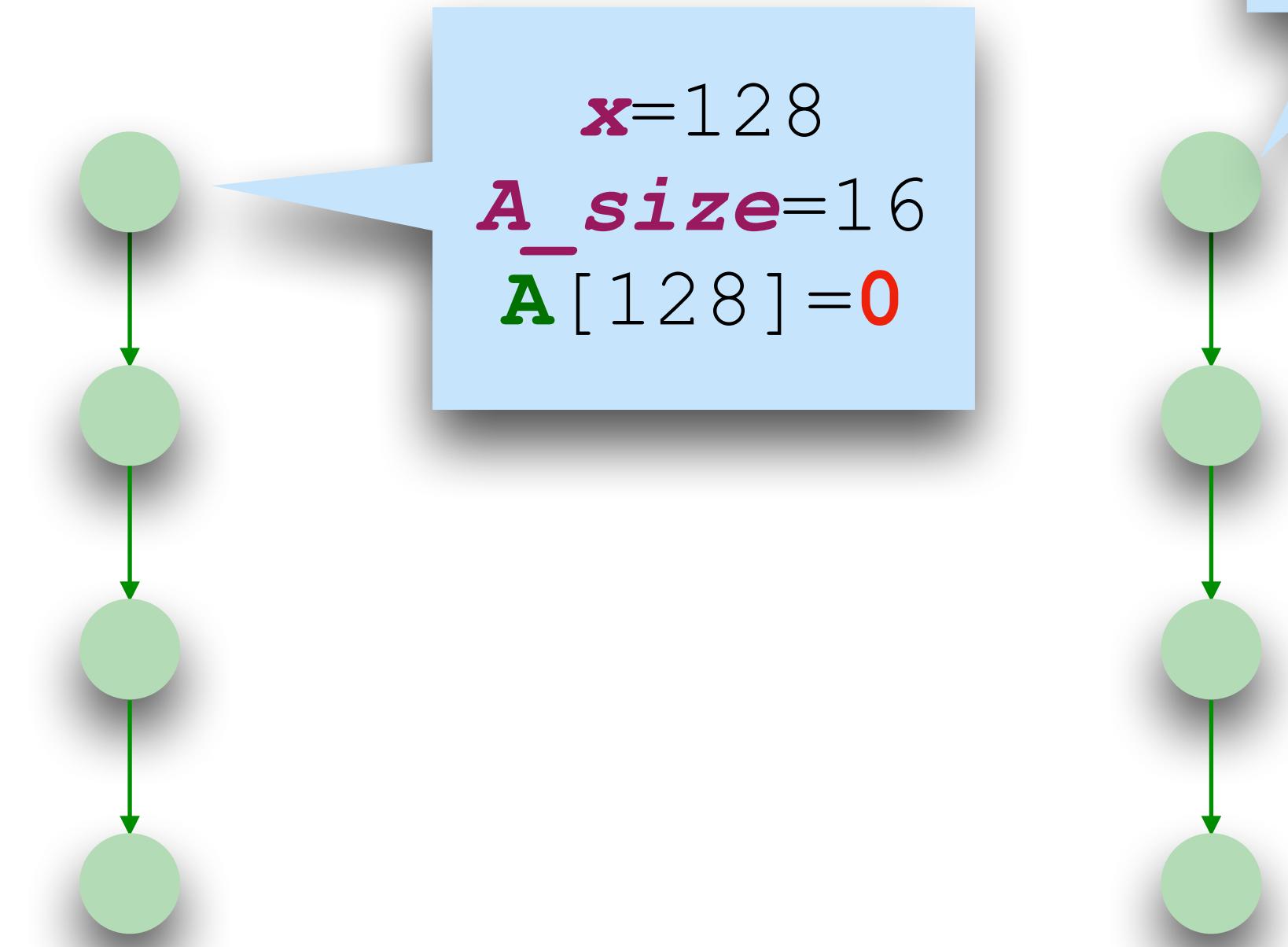
# Speculative non-interference

$x=128$   
 $A\_size=16$   
 $\mathbf{A}[128]=1$

```
rax <- A_size  
rcx <- x  
jmp rcx≥rax, END
```

L1: load rax, **A** + rcx  
load rax, **B** + rax

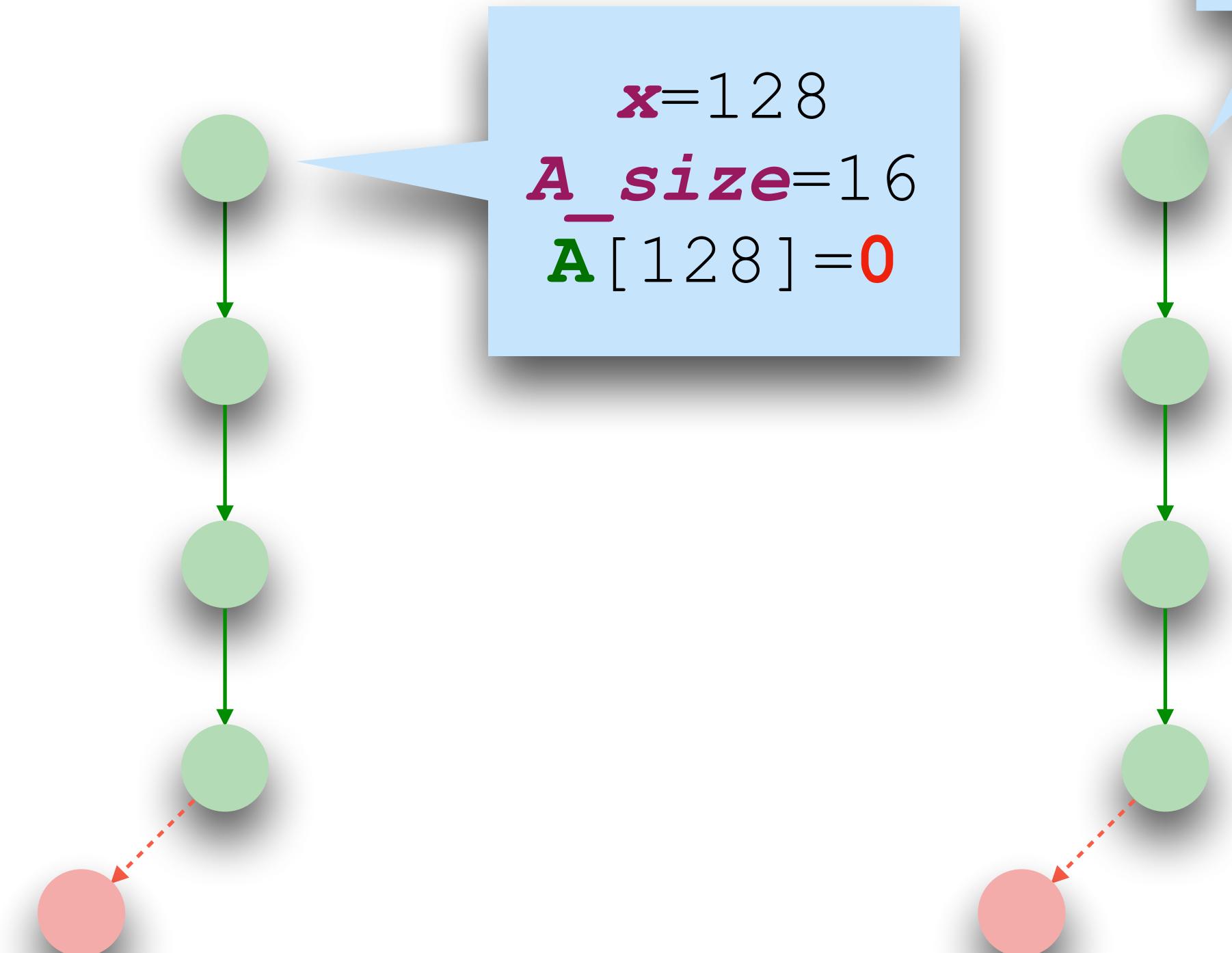
END:



# Speculative non-interference

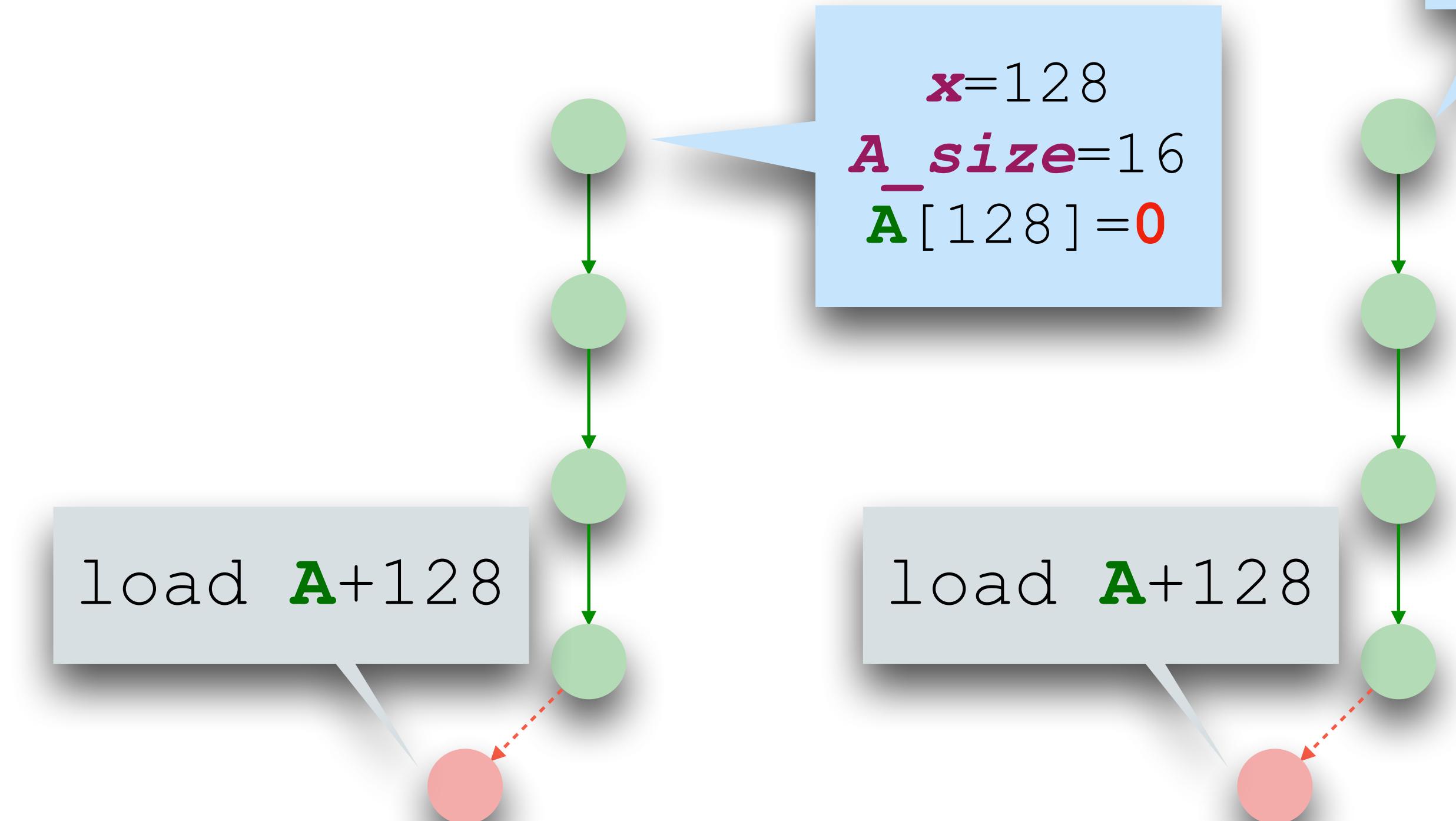
$x=128$   
 $A\_size=16$   
 $\mathbf{A}[128]=1$

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



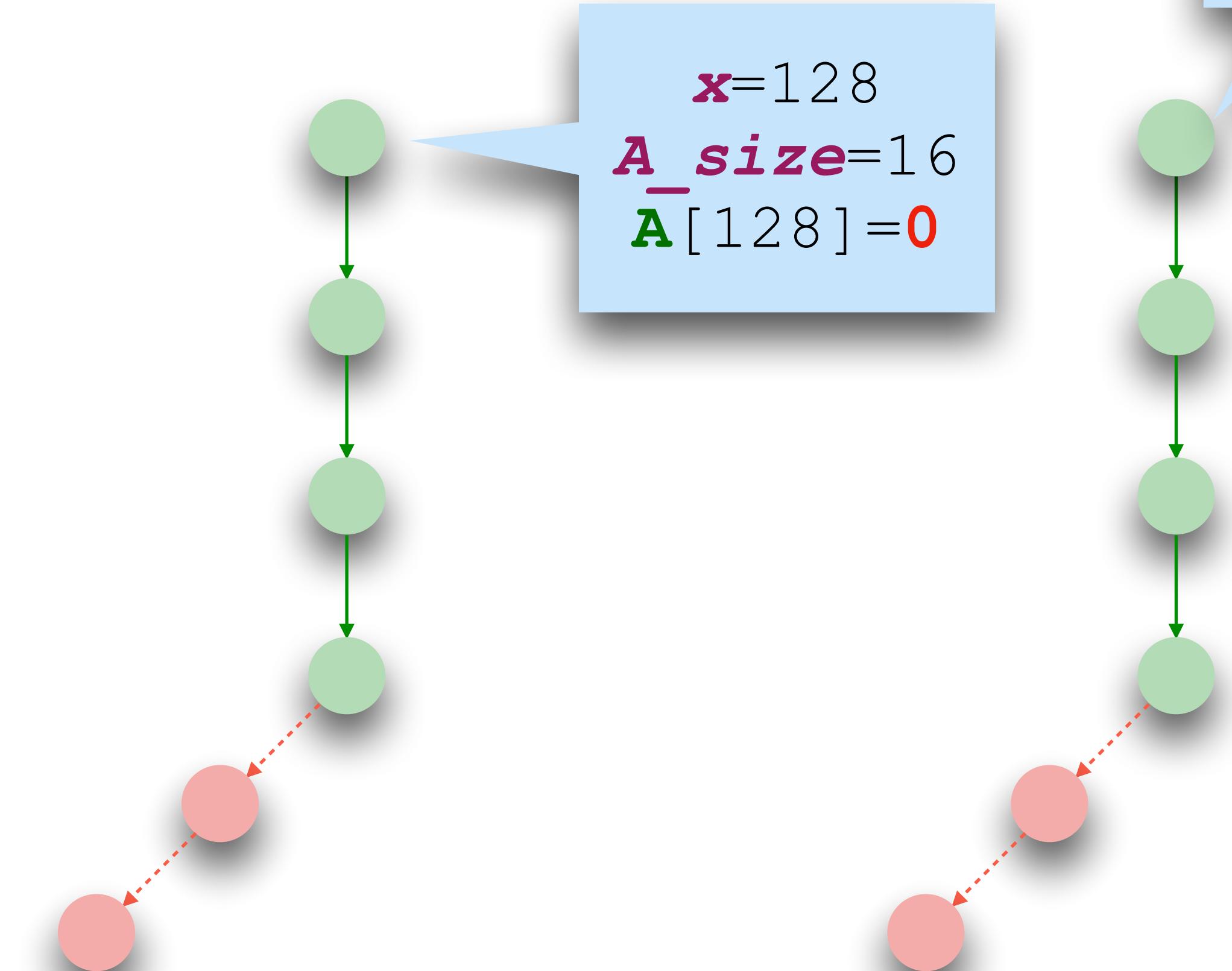
# Speculative non-interference

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



# Speculative non-interference

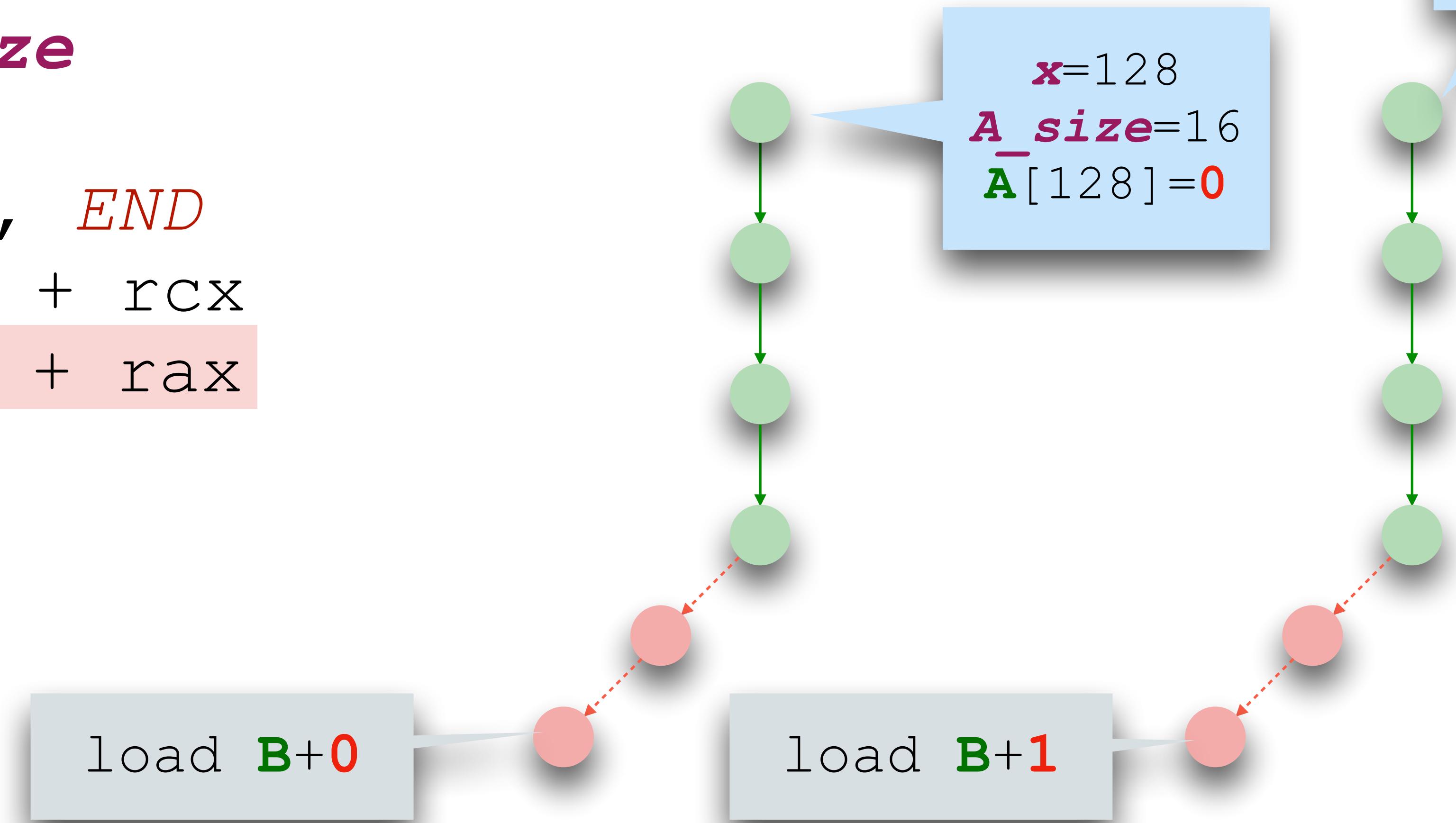
```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



# Speculative non-interference

$x=128$   
 $A\_size=16$   
 $\mathbf{A}[128]=1$

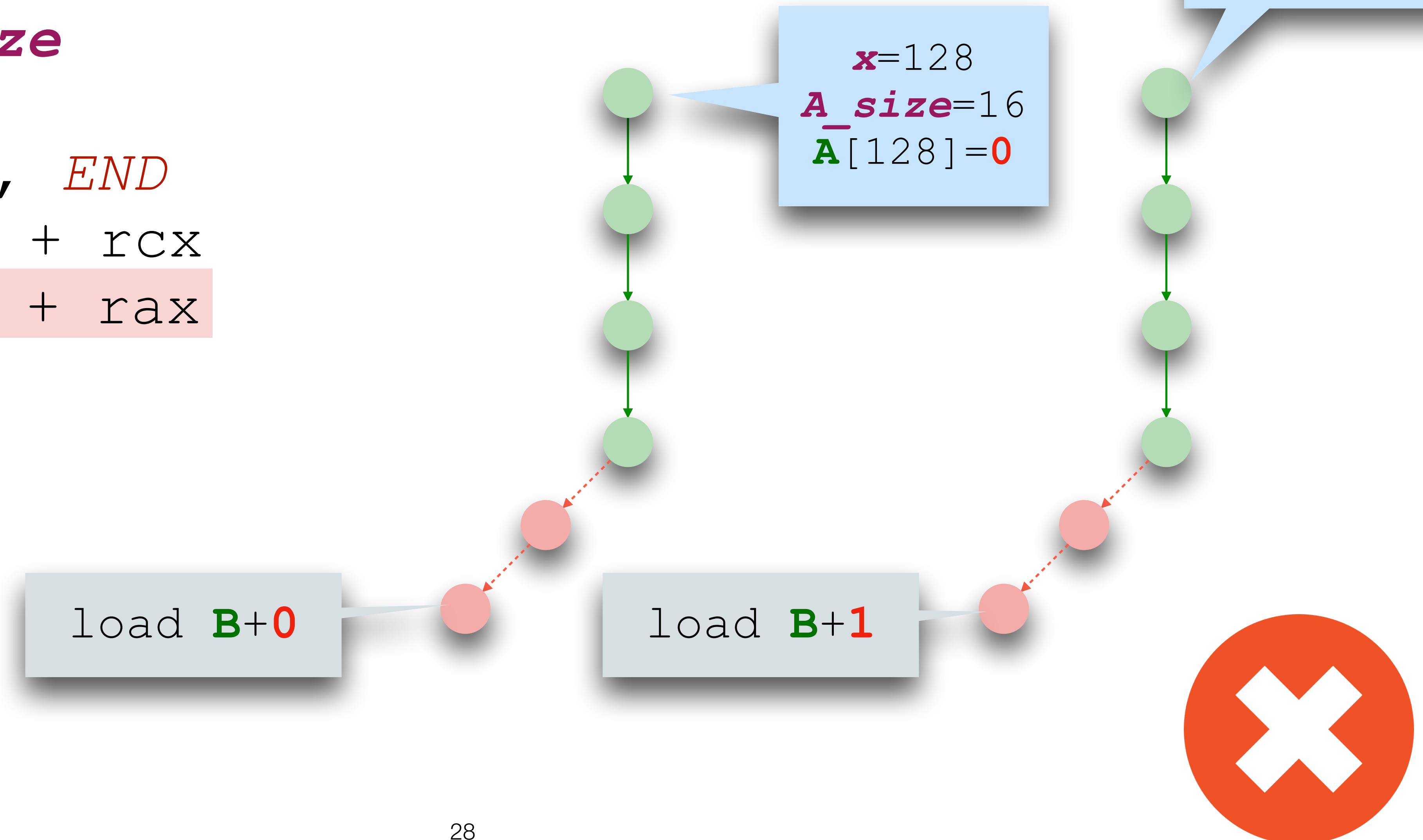
```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



# Speculative non-interference

$x=128$   
 $A\_size=16$   
 $\mathbf{A}[128]=1$

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



# Symbolic execution

# Symbolic execution

- Program analysis technique

Experiments with a symbolic evaluation system

JAM E. HOWDEN  
California at San Diego

**Programming Languages**  
**Symbolic Execution and Program Testing**  
James C. King  
IBM Thomas J. Watson Research Center  
B. Webbrait  
Editor

This paper describes the symbolic execution of programs. Instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values. The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols. This difficult, yet interesting issues arise during the symbolic execution of conditional branch type statements. A particular system called EFFIGY which provides symbolic execution for program testing and debugging is also described. It interpretively executes programs written in a simple PL/I style programming language. It includes many standard debugging features, the ability to manage and prove things about symbolic expressions, a simple and brief discussion of the relationship between symbolic program testing manager, and a program verifier. A execution and program verification is also included. Key Words and Phrases: symbolic expressions, symbolic program testing, program debugging, program execution, program verification, symbolic interpretation, program proving, CR Categories: 4.13, 5.21, 5.24

Copyright © 1976 Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that the source is made to the publication, its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. P.O. Box 218, Yorktown Heights, N.Y. 10598.

July 1976  
Volume 19  
Number 7  
Communications  
of the ACM

385

1. Introduction

The large-scale production of reliable programs is one of the fundamental requirements for applying computers to today's challenging problems. Several techniques are used in practice; others are the focus of current research. The work reported in this paper is directed at assuring that a program meets its requirements when formal specifications are not given. The current technology in this area is basically a testing technology. That is, some small sample of the data that a testing technology expects to handle is presented to the program. If the program is judged to produce correct results, the sample, it is assumed to be correct. Much current work [11] focuses on the question of how to choose this sample.

Recent work on proving the correctness of programs by formal analysis [5] shows the great promise and appears to be the ultimate technique for producing reliable programs. However, the practical accomplishments in this area fall short of a tool for routine use. Fundamental problems in reducing the theory to practice are not likely to be solved by the time the technology is available.

Program testing and program proving are considered as extreme alternatives. While test runs work correctly by carefully checking the results, the correct execution for inputs not in the sample is still in doubt. Alternatively, in program proving the programmer formulates his specification and proves it correct for all executions without being required to execute the program at all. To do this he gives a precise specification for the correct program behavior and then follows the method hinging on the care and accuracy employed in both the creation of the proof steps, as well as on the attention to machine-dependent issues such as overflow, rounding errors, etc.

This paper describes a practical approach between these two extremes. From one simple view, it is enhanced testing technique. Instead of executing a program on a set of sample inputs, a program is "symbolically" executed for a set of classes of inputs. That is, each symbolic execution result may be equivalent to a large number of normal test cases. These results can be checked against the programmer's expectations for correctness either formally or informally.

The class of inputs characterized by each symbolic execution is determined by its dependence on the program's control flow on its inputs. If the control flow of the program is completely independent of the input variables, a single symbolic execution will suffice to check all possible executions of the program. If the control flow of the program is dependent on the inputs, one must resort to a case analysis. Often the set of input

ables into the expression. The resulting expression is then simplified algebraically. All operators are evaluated as actual as opposed to symbolic operands. The resulting expression in the normal way. The original expression, Figure 1 contains a program for carrying out polynomial interpolation. The documentation for carrying out polynomial interpolation consists of four segments, each describes it as consisting of the interpolation process. Figure 1 is given in terms of  $y_i$  and  $A_j$ . The first segment is given as:

$\frac{a_1}{a_2}$

$\frac{a_2}{a_3}$

$\frac{a_3}{a_4}$

$\frac{a_4}{a_5}$

$\frac{a_5}{a_6}$

$\frac{a_6}{a_7}$

$\frac{a_7}{a_8}$

$\frac{a_8}{a_9}$

$\frac{a_9}{a_{10}}$

$\frac{a_{10}}{a_{11}}$

$\frac{a_{11}}{a_{12}}$

$\frac{a_{12}}{a_{13}}$

$\frac{a_{13}}{a_{14}}$

$\frac{a_{14}}{a_{15}}$

$\frac{a_{15}}{a_{16}}$

$\frac{a_{16}}{a_{17}}$

$\frac{a_{17}}{a_{18}}$

$\frac{a_{18}}{a_{19}}$

$\frac{a_{19}}{a_{20}}$

$\frac{a_{20}}{a_{21}}$

$\frac{a_{21}}{a_{22}}$

$\frac{a_{22}}{a_{23}}$

$\frac{a_{23}}{a_{24}}$

$\frac{a_{24}}{a_{25}}$

$\frac{a_{25}}{a_{26}}$

$\frac{a_{26}}{a_{27}}$

$\frac{a_{27}}{a_{28}}$

$\frac{a_{28}}{a_{29}}$

$\frac{a_{29}}{a_{30}}$

$\frac{a_{30}}{a_{31}}$

$\frac{a_{31}}{a_{32}}$

$\frac{a_{32}}{a_{33}}$

$\frac{a_{33}}{a_{34}}$

$\frac{a_{34}}{a_{35}}$

$\frac{a_{35}}{a_{36}}$

$\frac{a_{36}}{a_{37}}$

$\frac{a_{37}}{a_{38}}$

$\frac{a_{38}}{a_{39}}$

$\frac{a_{39}}{a_{40}}$

$\frac{a_{40}}{a_{41}}$

$\frac{a_{41}}{a_{42}}$

$\frac{a_{42}}{a_{43}}$

$\frac{a_{43}}{a_{44}}$

$\frac{a_{44}}{a_{45}}$

$\frac{a_{45}}{a_{46}}$

$\frac{a_{46}}{a_{47}}$

$\frac{a_{47}}{a_{48}}$

$\frac{a_{48}}{a_{49}}$

$\frac{a_{49}}{a_{50}}$

$\frac{a_{50}}{a_{51}}$

$\frac{a_{51}}{a_{52}}$

$\frac{a_{52}}{a_{53}}$

$\frac{a_{53}}{a_{54}}$

$\frac{a_{54}}{a_{55}}$

$\frac{a_{55}}{a_{56}}$

$\frac{a_{56}}{a_{57}}$

$\frac{a_{57}}{a_{58}}$

$\frac{a_{58}}{a_{59}}$

$\frac{a_{59}}{a_{60}}$

$\frac{a_{60}}{a_{61}}$

$\frac{a_{61}}{a_{62}}$

$\frac{a_{62}}{a_{63}}$

$\frac{a_{63}}{a_{64}}$

$\frac{a_{64}}{a_{65}}$

$\frac{a_{65}}{a_{66}}$

$\frac{a_{66}}{a_{67}}$

$\frac{a_{67}}{a_{68}}$

$\frac{a_{68}}{a_{69}}$

$\frac{a_{69}}{a_{70}}$

$\frac{a_{70}}{a_{71}}$

$\frac{a_{71}}{a_{72}}$

$\frac{a_{72}}{a_{73}}$

$\frac{a_{73}}{a_{74}}$

$\frac{a_{74}}{a_{75}}$

$\frac{a_{75}}{a_{76}}$

$\frac{a_{76}}{a_{77}}$

$\frac{a_{77}}{a_{78}}$

$\frac{a_{78}}{a_{79}}$

$\frac{a_{79}}{a_{80}}$

$\frac{a_{80}}{a_{81}}$

$\frac{a_{81}}{a_{82}}$

$\frac{a_{82}}{a_{83}}$

$\frac{a_{83}}{a_{84}}$

$\frac{a_{84}}{a_{85}}$

$\frac{a_{85}}{a_{86}}$

$\frac{a_{86}}{a_{87}}$

$\frac{a_{87}}{a_{88}}$

$\frac{a_{88}}{a_{89}}$

$\frac{a_{89}}{a_{90}}$

$\frac{a_{90}}{a_{91}}$

$\frac{a_{91}}{a_{92}}$

$\frac{a_{92}}{a_{93}}$

$\frac{a_{93}}{a_{94}}$

$\frac{a_{94}}{a_{95}}$

$\frac{a_{95}}{a_{96}}$

$\frac{a_{96}}{a_{97}}$

$\frac{a_{97}}{a_{98}}$

$\frac{a_{98}}{a_{99}}$

$\frac{a_{99}}{a_{100}}$

$\frac{a_{100}}{a_{101}}$

$\frac{a_{101}}{a_{102}}$

$\frac{a_{102}}{a_{103}}$

$\frac{a_{103}}{a_{104}}$

$\frac{a_{104}}{a_{105}}$

$\frac{a_{105}}{a_{106}}$

$\frac{a_{106}}{a_{107}}$

$\frac{a_{107}}{a_{108}}$

$\frac{a_{108}}{a_{109}}$

$\frac{a_{109}}{a_{110}}$

$\frac{a_{110}}{a_{111}}$

$\frac{a_{111}}{a_{112}}$

$\frac{a_{112}}{a_{113}}$

$\frac{a_{113}}{a_{114}}$

$\frac{a_{114}}{a_{115}}$

$\frac{a_{115}}{a_{116}}$

$\frac{a_{116}}{a_{117}}$

$\frac{a_{117}}{a_{118}}$

$\frac{a_{118}}{a_{119}}$

$\frac{a_{119}}{a_{120}}$

$\frac{a_{120}}{a_{121}}$

$\frac{a_{121}}{a_{122}}$

$\frac{a_{122}}{a_{123}}$

$\frac{a_{123}}{a_{124}}$

$\frac{a_{124}}{a_{125}}$

$\frac{a_{125}}{a_{126}}$

$\frac{a_{126}}{a_{127}}$

$\frac{a_{127}}{a_{128}}$

$\frac{a_{128}}{a_{129}}$

$\frac{a_{129}}{a_{130}}$

$\frac{a_{130}}{a_{131}}$

$\frac{a_{131}}{a_{132}}$

$\frac{a_{132}}{a_{133}}$

$\frac{a_{133}}{a_{134}}$

$\frac{a_{134}}{a_{135}}$

$\frac{a_{135}}{a_{136}}$

$\frac{a_{136}}{a_{137}}$

$\frac{a_{137}}{a_{138}}$

$\frac{a_{138}}{a_{139}}$

$\frac{a_{139}}{a_{140}}$

$\frac{a_{140}}{a_{141}}$

$\frac{a_{141}}{a_{142}}$

$\frac{a_{142}}{a_{143}}$

$\frac{a_{143}}{a_{144}}$

$\frac{a_{144}}{a_{145}}$

$\frac{a_{145}}{a_{146}}$

$\frac{a_{146}}{a_{147}}$

$\frac{a_{147}}{a_{148}}$

$\frac{a_{148}}{a_{149}}$

$\frac{a_{149}}{a_{150}}$

$\frac{a_{150}}{a_{151}}$

$\frac{a_{151}}{a_{152}}$

$\frac{a_{152}}{a_{153}}$

$\frac{a_{153}}{a_{154}}$

$\frac{a_{154}}{a_{155}}$

$\frac{a_{155}}{a_{156}}$

$\frac{a_{156}}{a_{157}}$

$\frac{a_{157}}{a_{158}}$

$\frac{a_{158}}{a_{159}}$

$\frac{a_{159}}{a_{160}}$

$\frac{a_{160}}{a_{161}}$

$\frac{a_{161}}{a_{162}}$

$\frac{a_{162}}{a_{163}}$

$\frac{a_{163}}{a_{164}}$

$\frac{a_{164}}{a_{165}}$

$\frac{a_{165}}{a_{166}}$

$\frac{a_{166}}{a_{167}}$

$\frac{a_{167}}{a_{168}}$

$\frac{a_{168}}{a_{169}}$

$\frac{a_{169}}{a_{170}}$

$\frac{a_{170}}{a_{171}}$

$\frac{a_{171}}{a_{172}}$

$\frac{a_{172}}{a_{173}}$

$\frac{a_{173}}{a_{174}}$

$\frac{a_{174}}{a_{175}}$

$\frac{a_{175}}{a_{176}}$

$\frac{a_{176}}{a_{177}}$

$\frac{a_{177}}{a_{178}}$

$\frac{a_{178}}{a_{179}}$

$\frac{a_{179}}{a_{180}}$

$\frac{a_{180}}{a_{181}}$

$\frac{a_{181}}{a_{182}}$

$\frac{a_{182}}{a_{183}}$

$\frac{a_{183}}{a_{184}}$

$\frac{a_{184}}{a_{185}}$

$\frac{a_{185}}{a_{186}}$

$\frac{a_{186}}{a_{187}}$

$\frac{a_{187}}{a_{188}}$

$\frac{a_{188}}{a_{189}}$

$\frac{a_{189}}{a_{190}}$

$\frac{a_{190}}{a_{191}}$

$\frac{a_{191}}{a_{192}}$

$\frac{a_{192}}{a_{193}}$

$\frac{a_{193}}{a_{194}}$

$\frac{a_{194}}{a_{195}}$

$\frac{a_{195}}{a_{196}}$

$\frac{a_{196}}{a_{197}}$

$\frac{a_{197}}{a_{198}}$

$\frac{a_{198}}{a_{199}}$

$\frac{a_{199}}{a_{200}}$

$\frac{a_{200}}{a_{201}}$

$\frac{a_{201}}{a_{202}}$

$\frac{a_{202}}{a_{203}}$

$\frac{a_{203}}{a_{204}}$

$\frac{a_{204}}{a_{205}}$

$\frac{a_{205}}{a_{206}}$

$\frac{a_{206}}{a_{207}}$

$\frac{a_{207}}{a_{208}}$

$\frac{a_{208}}{a_{209}}$

$\frac{a_{209}}{a_{210}}$

$\frac{a_{210}}{a_{211}}$

$\frac{a_{211}}{a_{212}}$

$\frac{a_{212}}{a_{213}}$

$\frac{a_{213}}{a_{214}}$

$\frac{a_{214}}{a_{215}}$

$\frac{a_{215}}{a_{216}}$

$\frac{a_{216}}{a_{217}}$

$\frac{a_{217}}{a_{218}}$

$\frac{a_{218}}{a_{219}}$

$\frac{a_{219}}{a_{220}}$

$\frac{a_{220}}{a_{221}}$

$\frac{a_{221}}{a_{222}}$

$\frac{a_{222}}{a_{223}}$

$\frac{a_{223}}{a_{224}}$

$\frac{a_{224}}{a_{225}}$

$\frac{a_{225}}{a_{226}}$

$\frac{a_{226}}{a_{227}}$

$\frac{a_{227}}{a_{228}}$

$\frac{a_{228}}{a_{229}}$

$\frac{a_{229}}{a_{230}}$

$\frac{a_{230}}{a_{231}}$

$\frac{a_{231}}{a_{232}}$

$\frac{a_{232}}{a_{233}}$

$\frac{a_{233}}{a_{234}}$

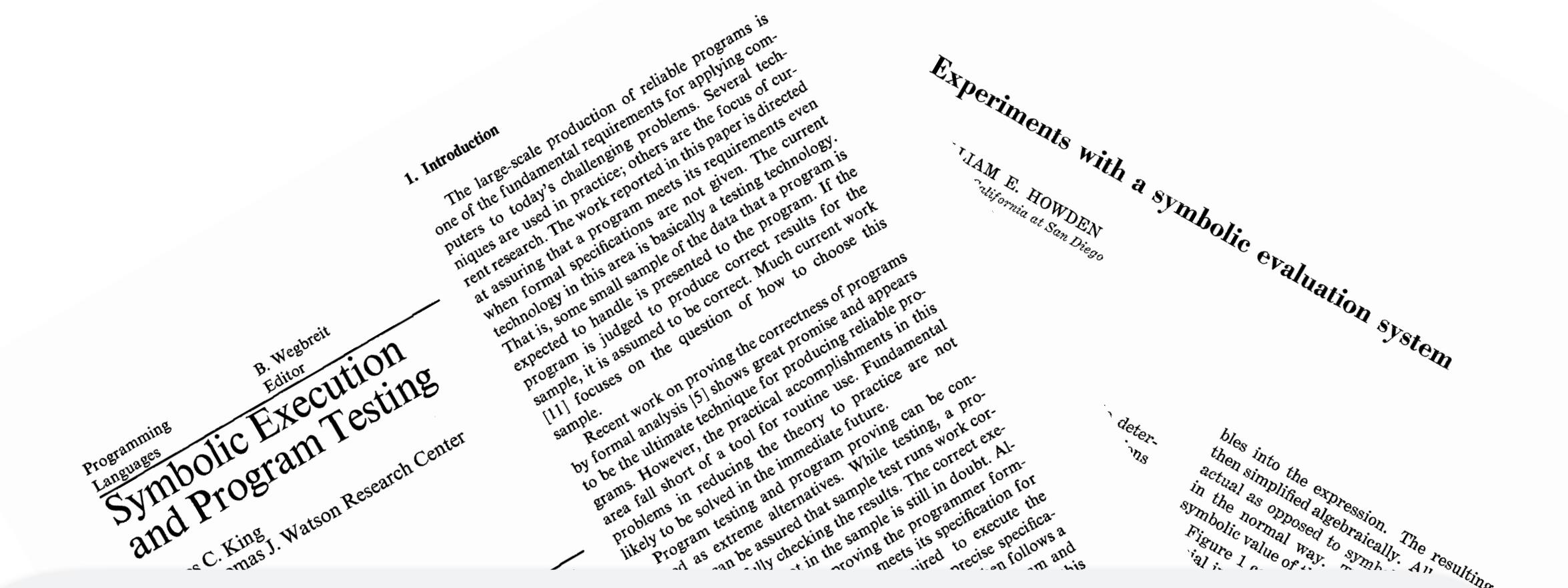
$\frac{a_{234}}{a_{235}}$

$\frac{a_{235}}{a_{236}}$

$\frac{a_{236}}$

# Symbolic execution

- Program analysis technique
- Execute programs over symbolic values



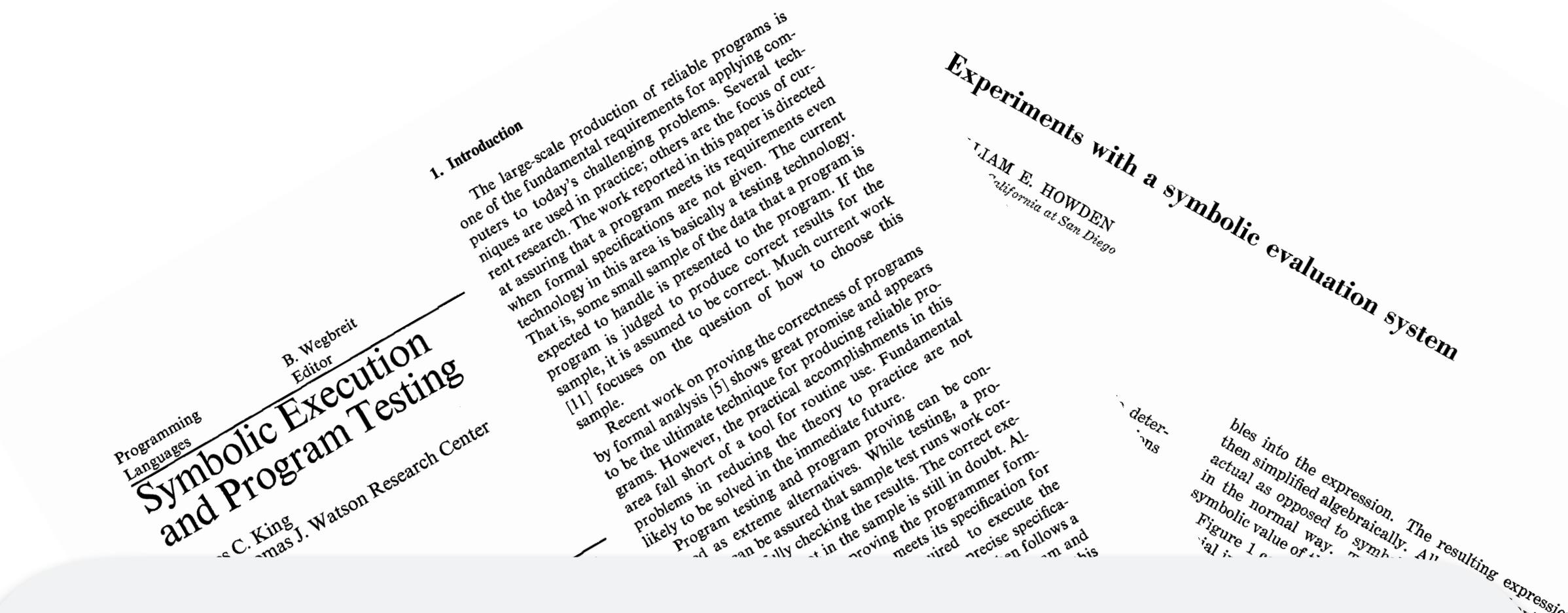
“The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols”  
— James C. King

Copyright © 1976 Association for Computing Machinery, Inc. General permission to republish, otherwise, is granted provided that 385  
is given and to the reference that is made to the journal. by permission of the Association for Computing Machinery, P.O. Box 218, Yorktown Heights, N.Y. 10598.

# Symbolic execution

- Program analysis technique
  - Execute programs over symbolic values
    - Explore all paths, each with its own path constraint

“The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols”

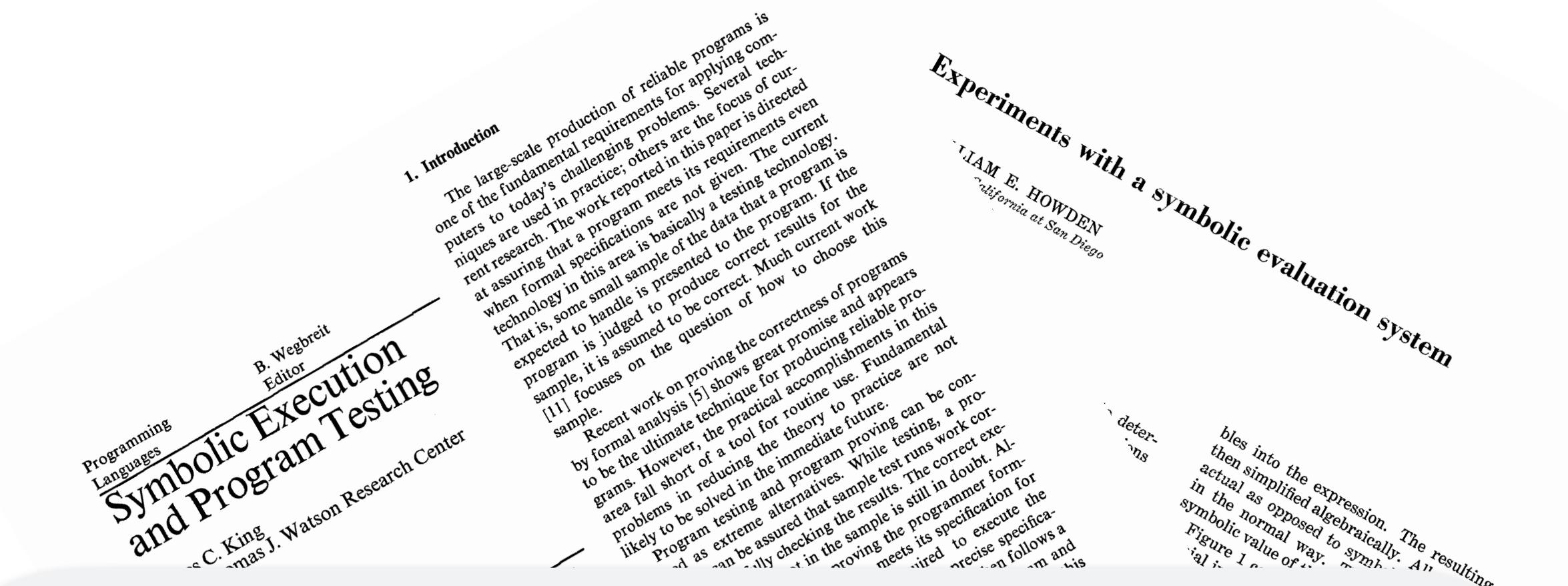


Copyright © 1976, Association  
of Computing Machinery,  
Inc. All rights reserved.  
General permission is granted to  
reproduce, but not to  
sell, this publication  
in whole or in part, without  
prior written permission  
from the publisher.  
Author's address: IBM Thomas J. Watson Research  
Center, P.O. Box 218, Yorktown Heights, N.Y. 10598.

Symbolic variables which  
are used to generate  
different programs  
symbolic evaluation  
used as the validation  
present in an automated  
order for the system to  
listic programs. The user

# Symbolic execution

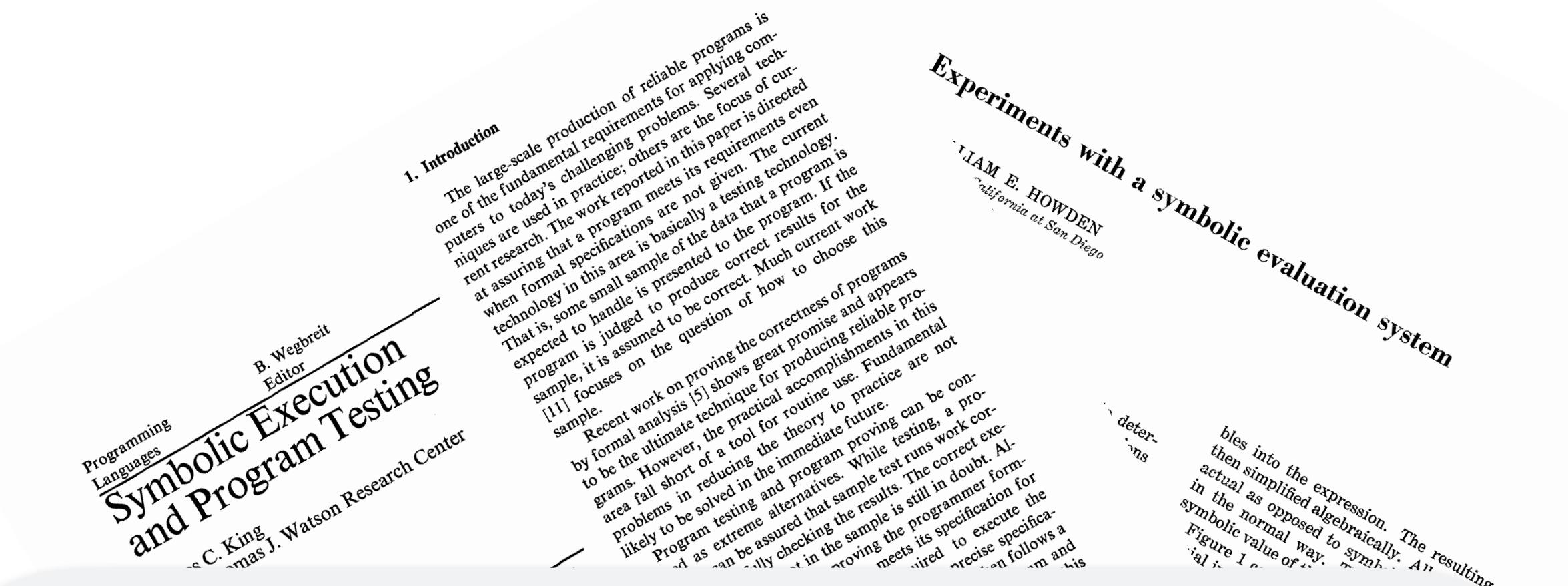
- Program analysis technique
- Execute programs over symbolic values
  - Explore all paths, each with its own path constraint
  - Each path represents all concrete executions satisfying the constraint



“The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols”  
— James C. King

# Symbolic execution

- Program analysis technique
- Execute programs over symbolic values
  - Explore all paths, each with its own path constraint
  - Each path represents all concrete executions satisfying the constraint
  - Branch and jump instructions: fork paths and update path constraint



“The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols”  
— James C. King

Copyright © 1976 Association  
of Computing Machinery, Inc.  
General permission to republish  
material from this journal is granted  
by the publisher, provided that  
the base of issue and to the reference  
is given and to the fact that reprinting  
privileges are granted by the publisher.  
P.O. Box 218, Yorktown Heights, N.Y. 10598.

385

# Symbolic execution

```
1. if (x < A_size)
2.     y = A[x]
3.     z = B[y]
4. end
```

# Symbolic execution

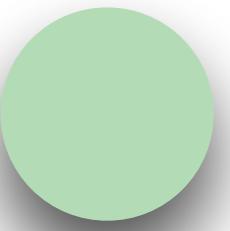
```
1. if (x < A_size)  
2.     y = A[x]  
3.     z = B[y]  
4. end
```



*Always mispredict*  
branch instructions

# Symbolic execution

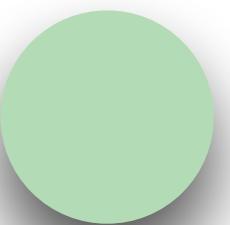
```
1. if (x < A_size)           true  
2.   y = A[x]  
3.   z = B[y]  
4. end
```



*Always mispredict*  
branch instructions

# Symbolic execution

```
1. if (x < A_size)           true  
2.   y = A [x]  
3.   z = B [y]  
4. end
```

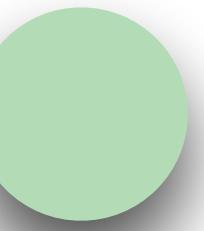


*Always mispredict*  
branch instructions

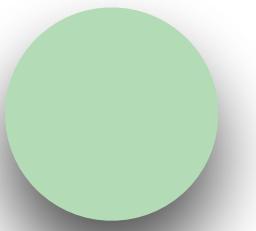
# Symbolic execution

```
1. if ( $x < A\_size$ )  
2.      $y = A[x]$   
3.      $z = B[y]$   
4. end
```

$x \geq A\_size$



$x < A\_size$



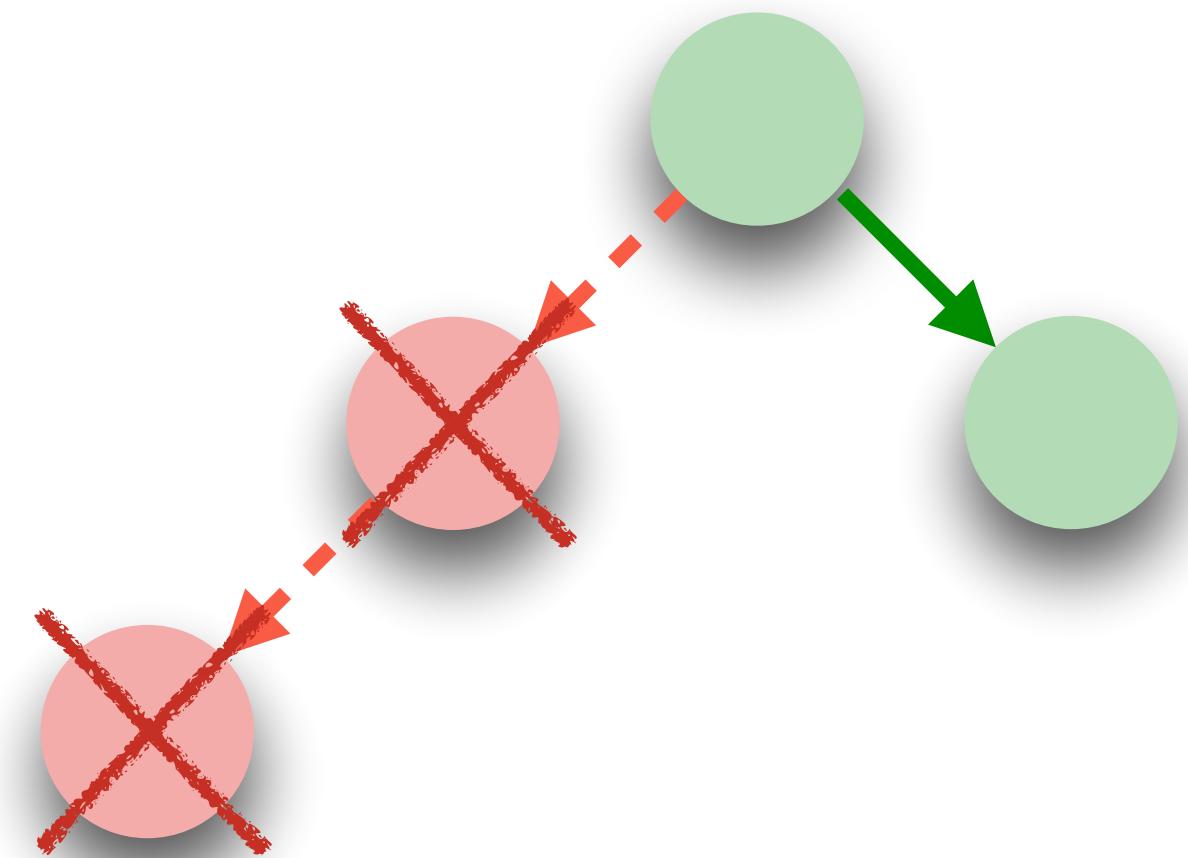
*Always mispredict*  
branch instructions

# Symbolic execution

```
1. if ( $x < A\_size$ )
2.    $y = A[x]$ 
3.    $z = B[y]$ 
4. end
```

$x \geq A\_size$

$x < A\_size$



*Always mispredict*  
branch instructions

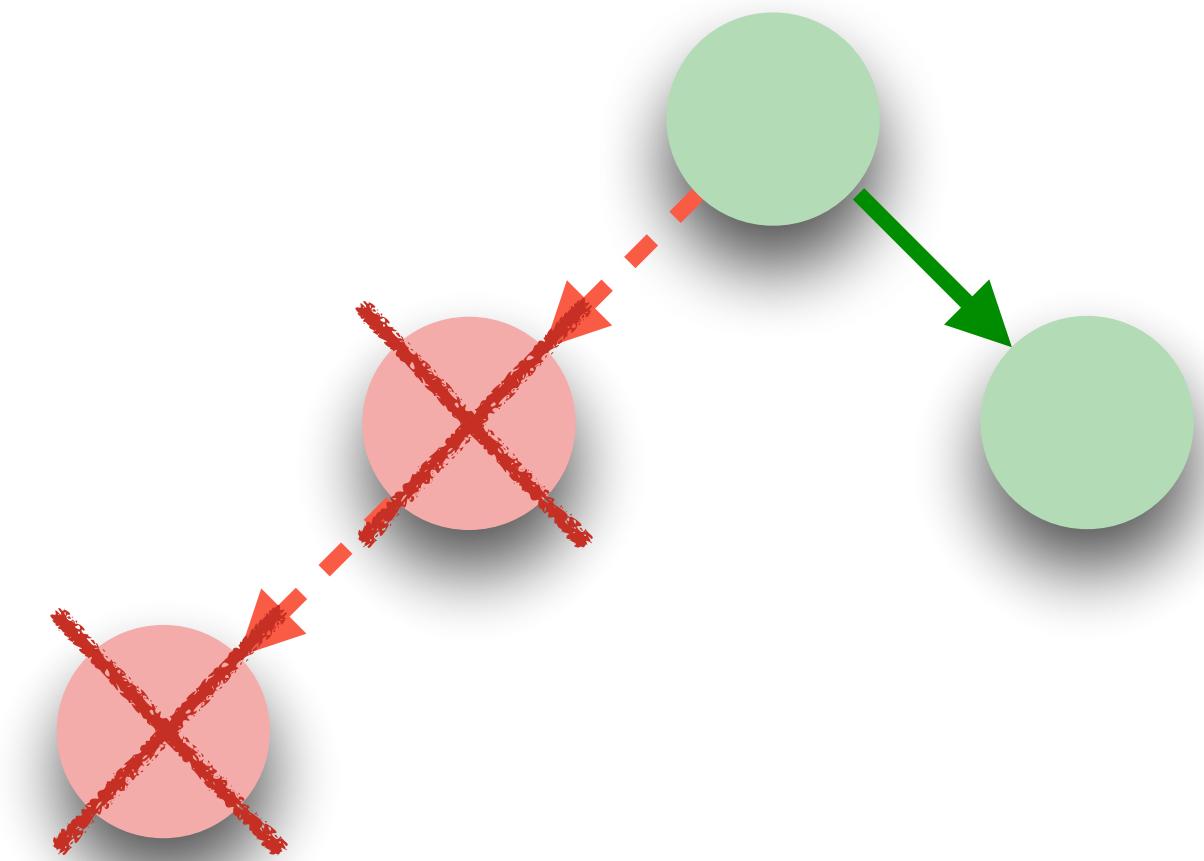
# Symbolic execution

```
1. if ( $x < A\_size$ )
2.    $y = A[x]$ 
3.    $z = B[y]$ 
4. end
```

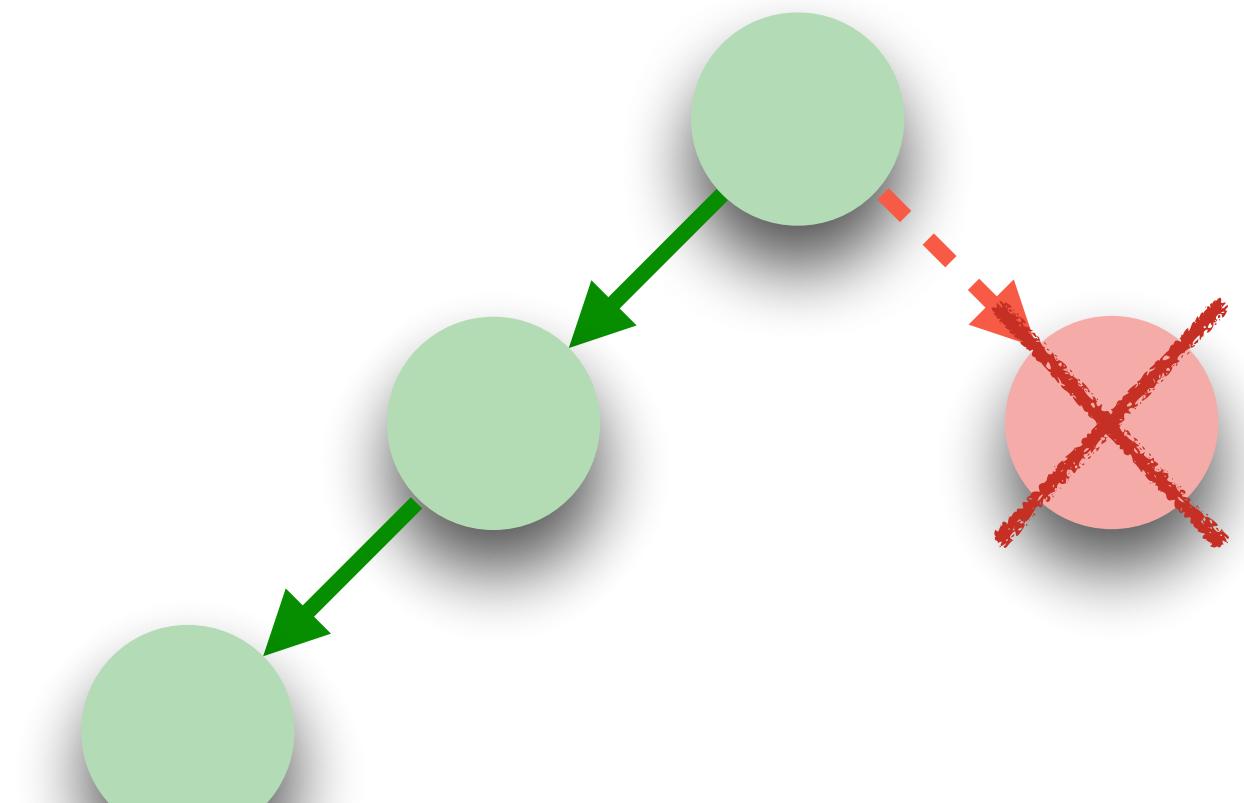


*Always mispredict*  
branch instructions

$x \geq A\_size$



$x < A\_size$

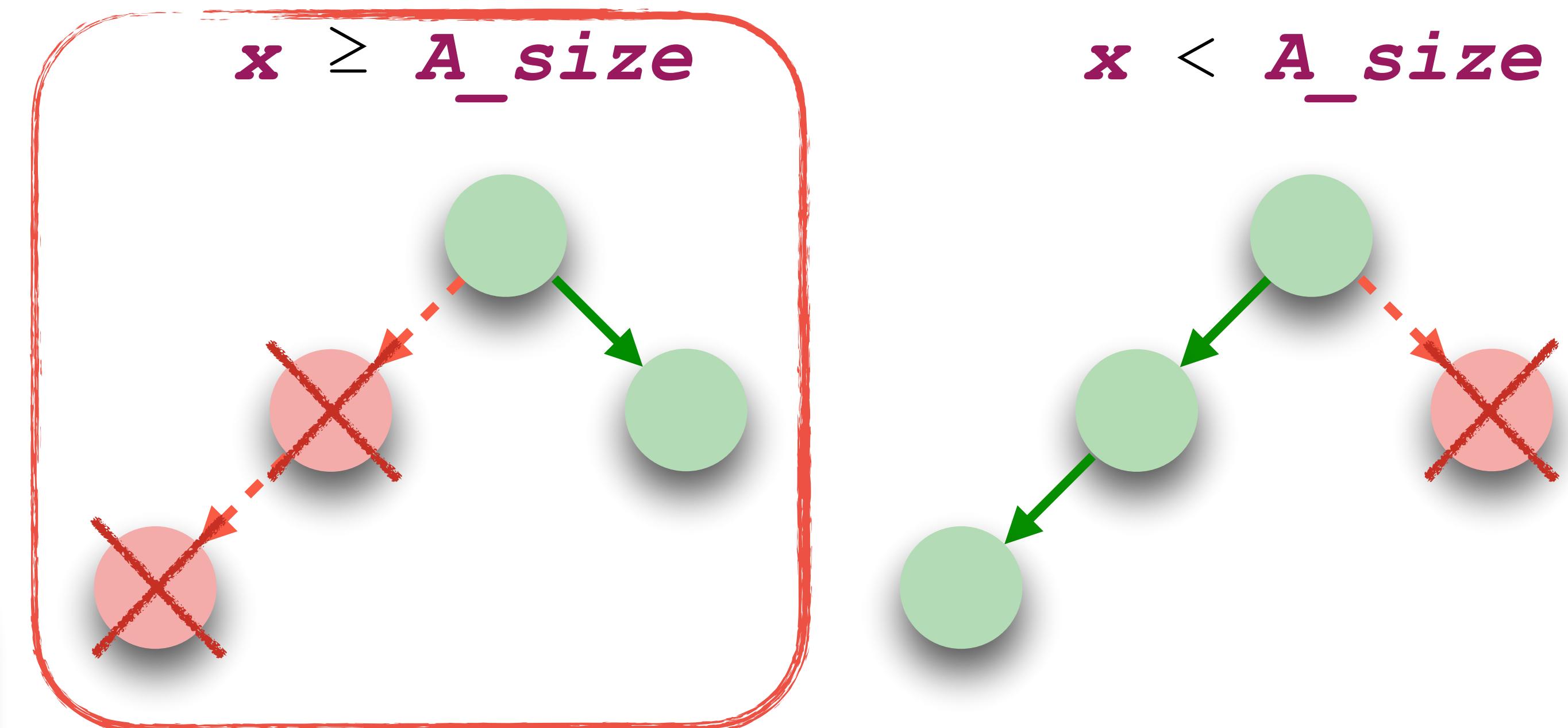


# Symbolic execution

```
1. if ( $x < A\_size$ )  
2.      $y = A[x]$   
3.      $z = B[y]$   
4. end
```



*Always mispredict*  
branch instructions

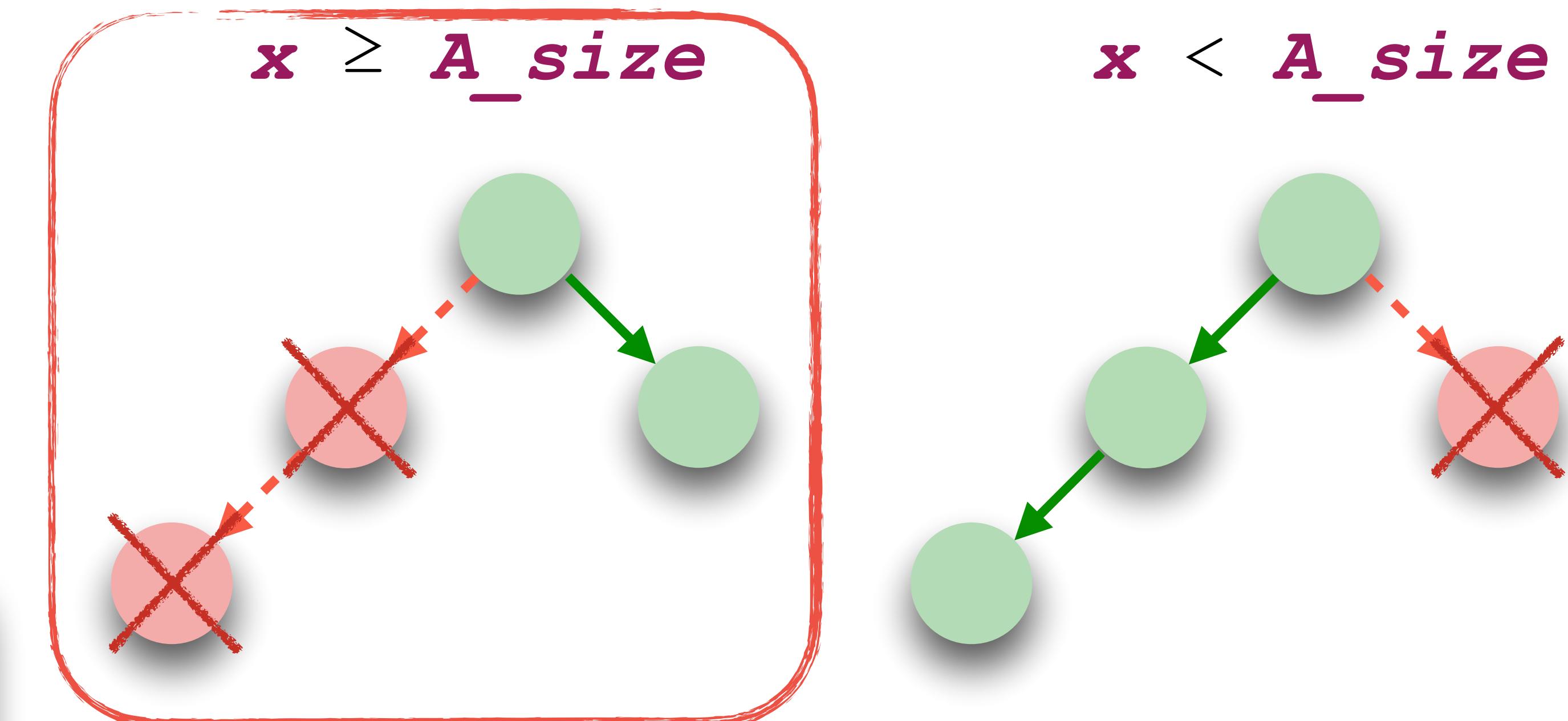


# Symbolic execution

```
1. if ( $x < A\_size$ )  
2.    $y = A[x]$   
3.    $z = B[y]$   
4. end
```



*Always mispredict*  
branch instructions



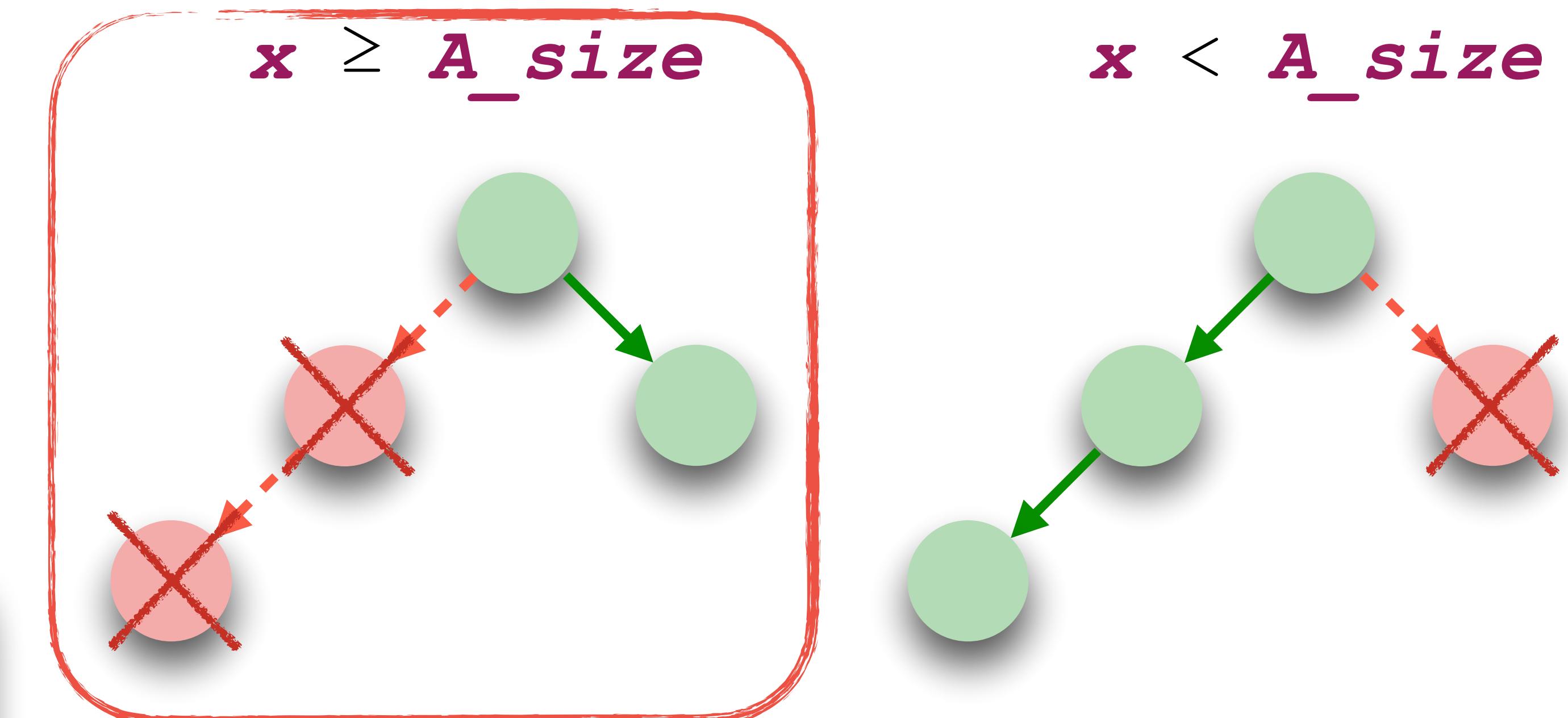
start pc 2 load  $A+x$  load  $B+A[x]$  rollback pc 4

# Symbolic execution

```
1. if ( $x < A\_size$ )  
2.    $y = A[x]$   
3.    $z = B[y]$   
4. end
```



*Always mispredict*  
branch instructions



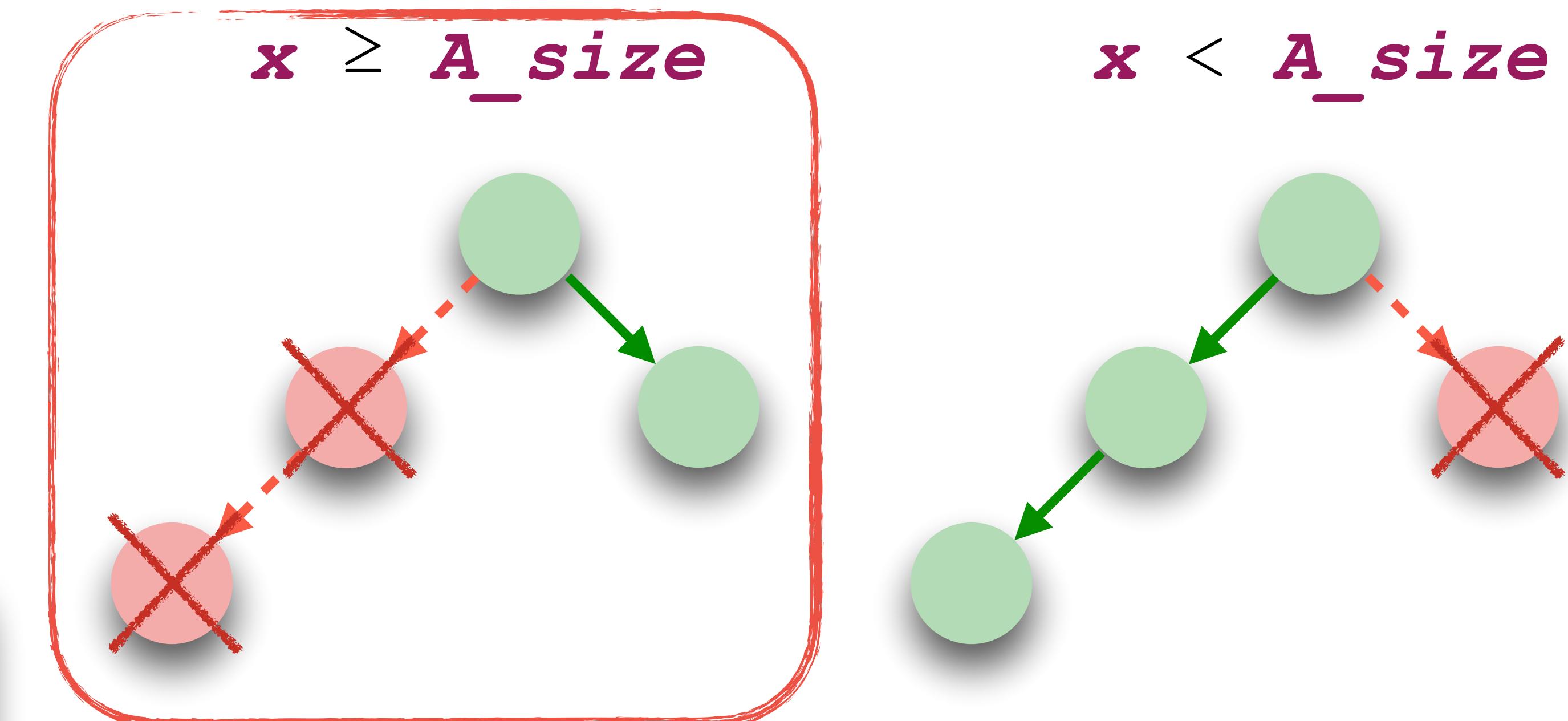
start pc 2 load  $A+x$  load  $B+A[x]$  rollback pc 4

# Symbolic execution

```
1. if ( $x < A\_size$ )  
2.      $y = A[x]$   
3.      $z = B[y]$   
4. end
```



*Always mispredict*  
branch instructions

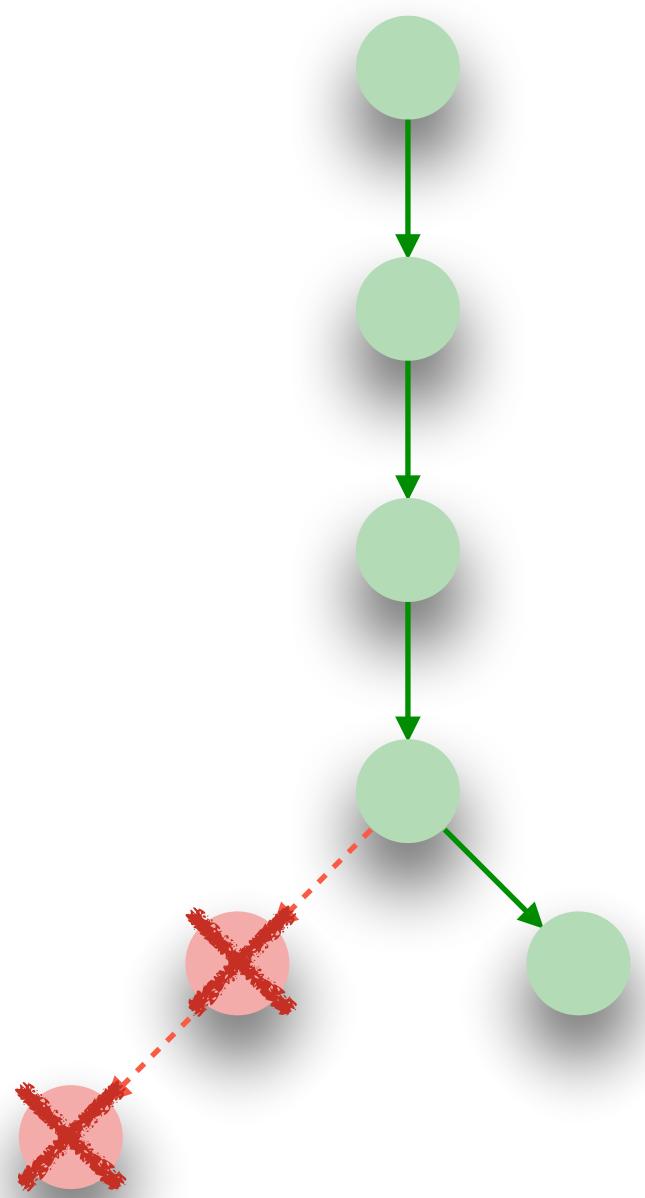


start	pc	2	load	$A+x$	load	$B+A[x]$	rollback	pc	4
-------	----	---	------	-------	------	----------	----------	----	---

# Memory leaks

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
```

*END:*



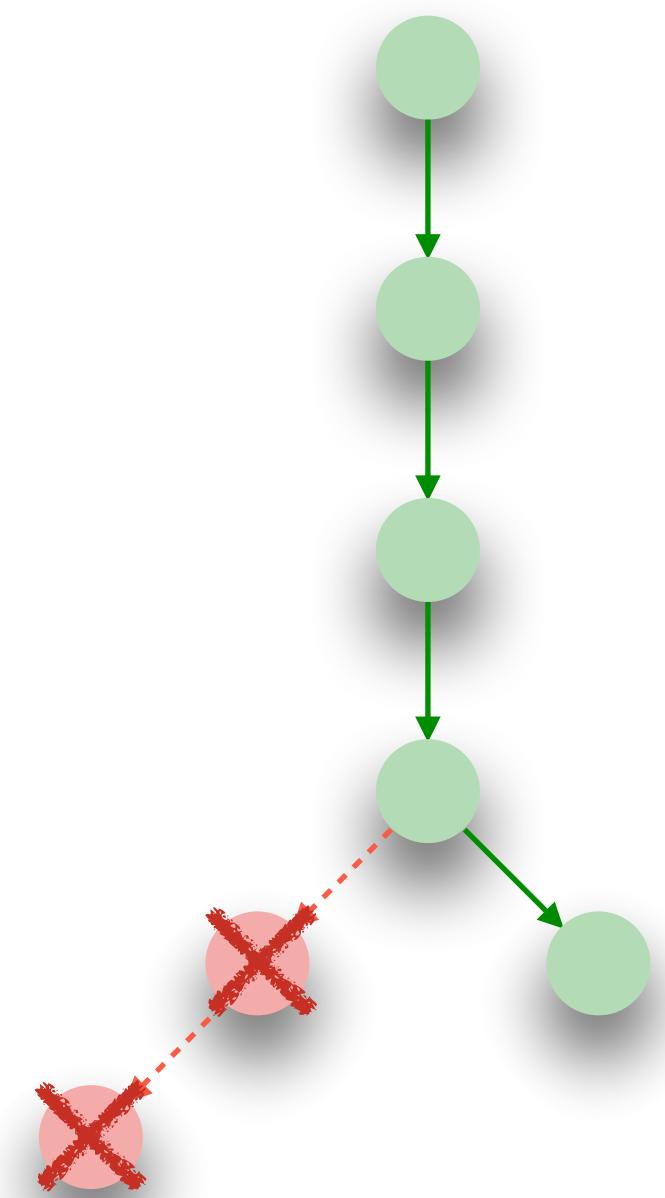
**Policy**  
**x, A\_size, A, B**  
are public

start pc <i>L1</i>	load <b>A+x</b>	load <b>B+A[x]</b>	rollback	pc <i>END</i>
--------------------	-----------------	--------------------	----------	---------------

# Memory leaks

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
```

*END:*



**Policy**  
**x, A\_size, A, B**  
are public

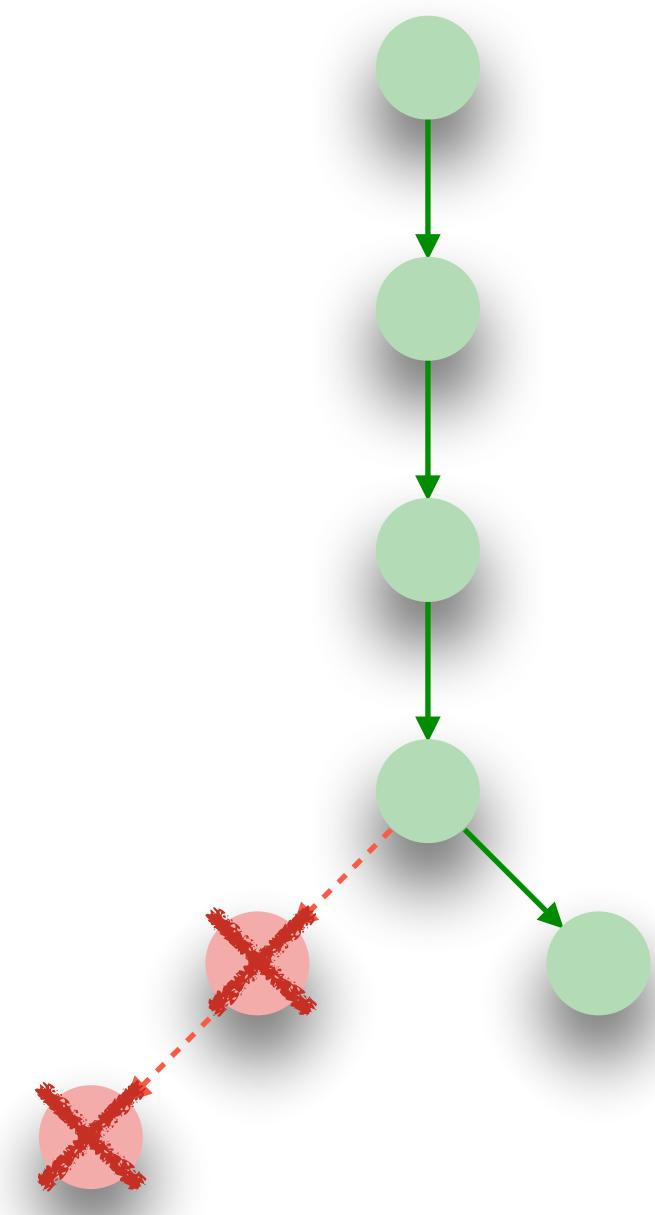
start pc <i>L1</i> load <b>A+x</b> load <b>B+A[x]</b> rollback	pc <i>END</i>
--	---------------

$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

# Memory leaks

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
```

*END:*



**Policy**  
**x, A\_size, A, B**  
are public

start pc <i>L1</i>	load <b>A+x</b>	load <b>B+A[x]</b>	rollback	pc <i>END</i>
--------------------	-----------------	--------------------	----------	---------------

$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

*s<sub>1</sub>*

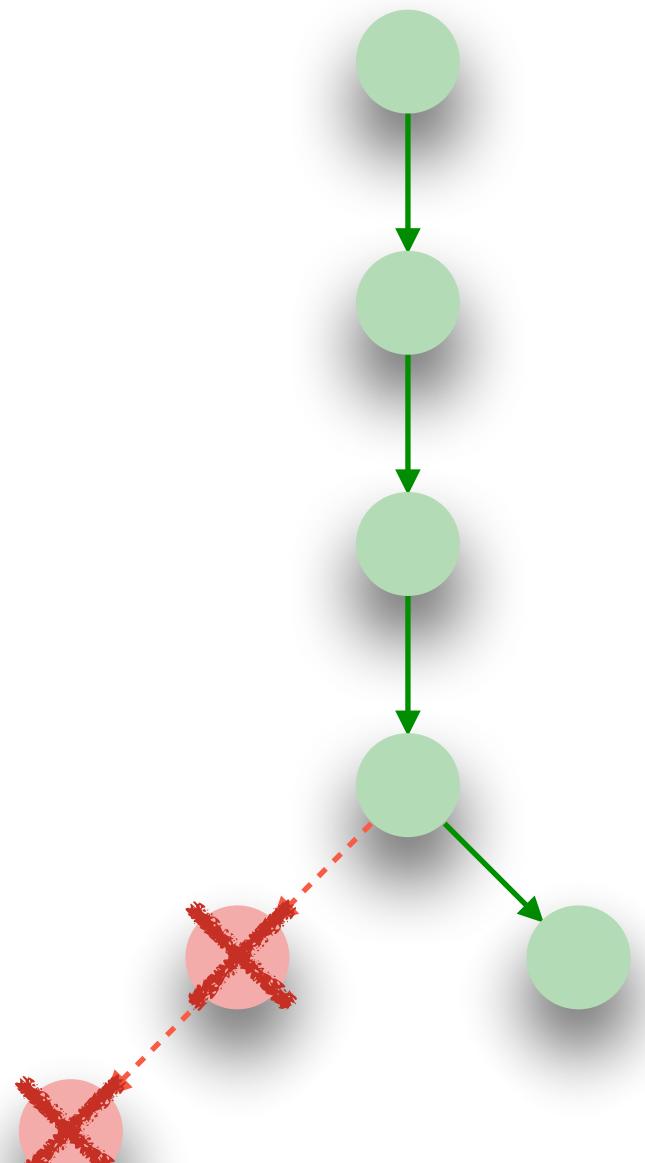
*s<sub>2</sub>*

# Memory leaks



```

    rax <- A_size
    rcx <- x
    jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
  
```



**Policy**  
 $\mathbf{x}, \mathbf{A\_size}, \mathbf{A}, \mathbf{B}$   
 are public

start pc <i>L1</i>	load <b>A+x</b>	load <b>B+A[x]</b>	rollback	pc <i>END</i>
--------------------	-----------------	--------------------	----------	---------------

$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

$s_1$

$s_2$

$$\mathbf{x}_1 = \mathbf{x}_2 \wedge \mathbf{A\_size}_1 = \mathbf{A\_size}_2 \wedge \mathbf{A}_1 = \mathbf{A}_2 \wedge \mathbf{B}_1 = \mathbf{B}_2$$

# Memory leaks

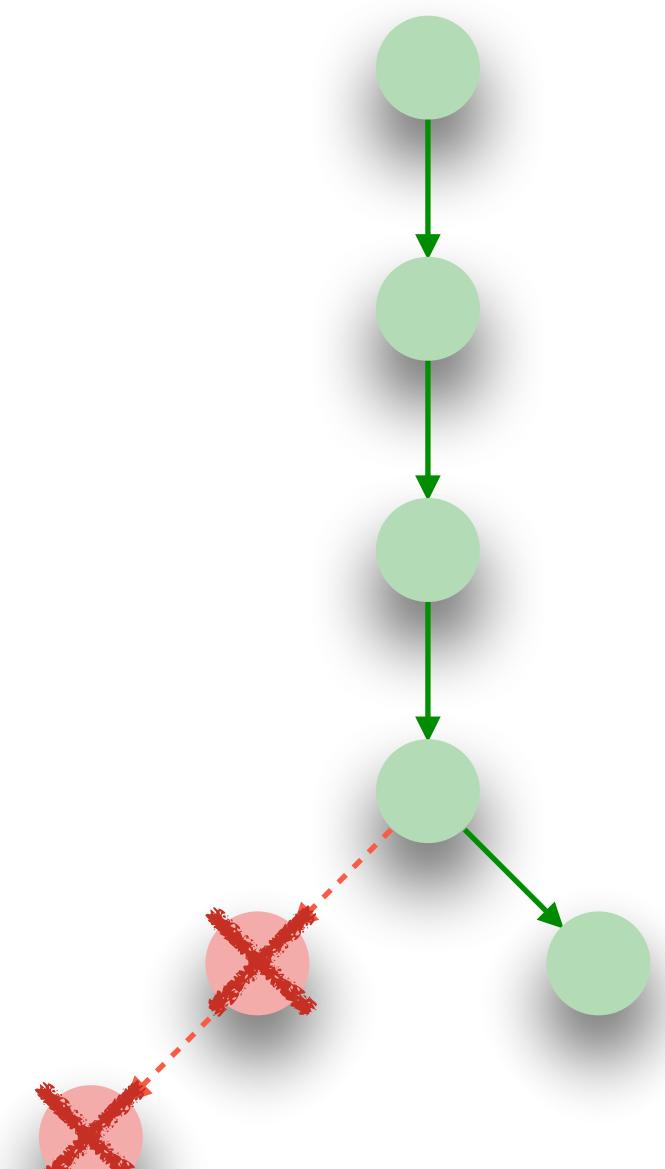


```

    rax <- A_size
    rcx <- x
    jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax

```

*END:*



**Policy**  
 $\mathbf{x}, \mathbf{A\_size}, \mathbf{A}, \mathbf{B}$   
 are public

start pc <i>L1</i>	load <b>A+x</b>	load <b>B+A[x]</b>	rollback	pc <i>END</i>
--------------------	-----------------	--------------------	----------	---------------

$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

$S_1 \models \mathbf{x}_1 \geq \mathbf{A\_size}_1$

$S_2 \models \mathbf{x}_2 \geq \mathbf{A\_size}_2$

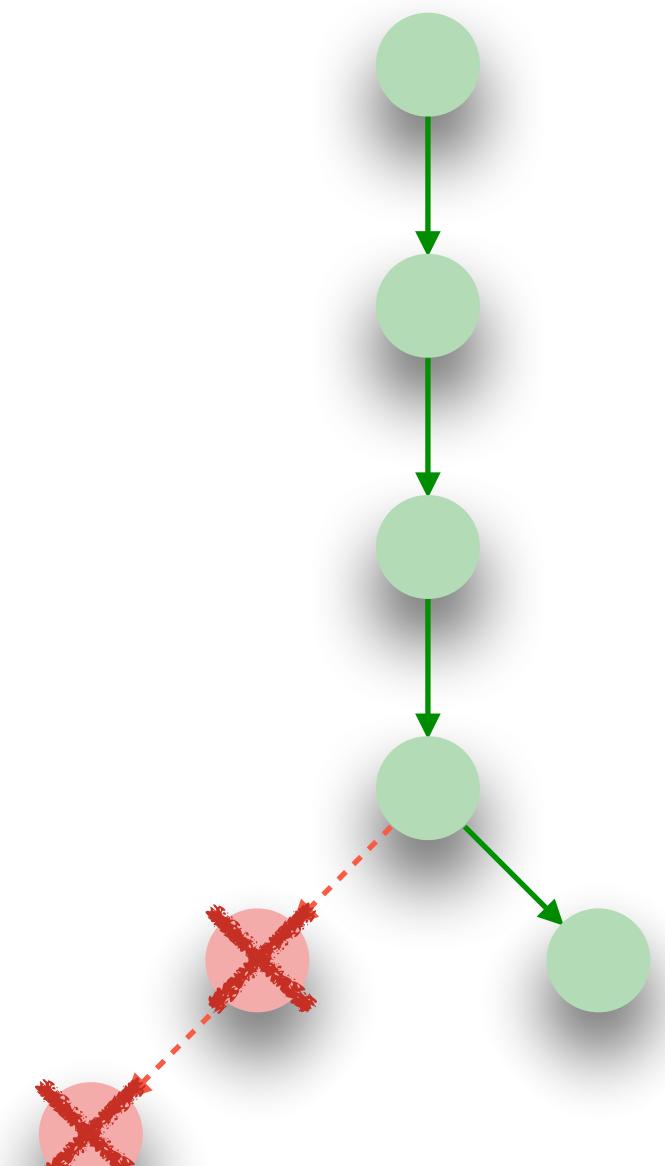
$$\mathbf{x}_1 = \mathbf{x}_2 \wedge \mathbf{A\_size}_1 = \mathbf{A\_size}_2 \wedge \mathbf{A}_1 = \mathbf{A}_2 \wedge \mathbf{B}_1 = \mathbf{B}_2$$

# Memory leaks



```

    rax <- A_size
    rcx <- x
    jmp rcx≥rax, END
L1: load rax, A + rcx
          load rax, B + rax
END:
  
```



**Policy**  
 $\mathbf{x}, \mathbf{A\_size}, \mathbf{A}, \mathbf{B}$   
 are public

start pc <i>L1</i>	load <b>A+x</b>	load <b>B+A[x]</b>	rollback	pc <i>END</i>
--------------------	-----------------	--------------------	----------	---------------

$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$

$S_1 \models \mathbf{x}_1 \geq \mathbf{A\_size}_1$

$S_2 \models \mathbf{x}_2 \geq \mathbf{A\_size}_2$

$\mathbf{x}_1 = \mathbf{x}_2 \wedge \mathbf{A\_size}_1 = \mathbf{A\_size}_2 \wedge \mathbf{A}_1 = \mathbf{A}_2 \wedge \mathbf{B}_1 = \mathbf{B}_2$

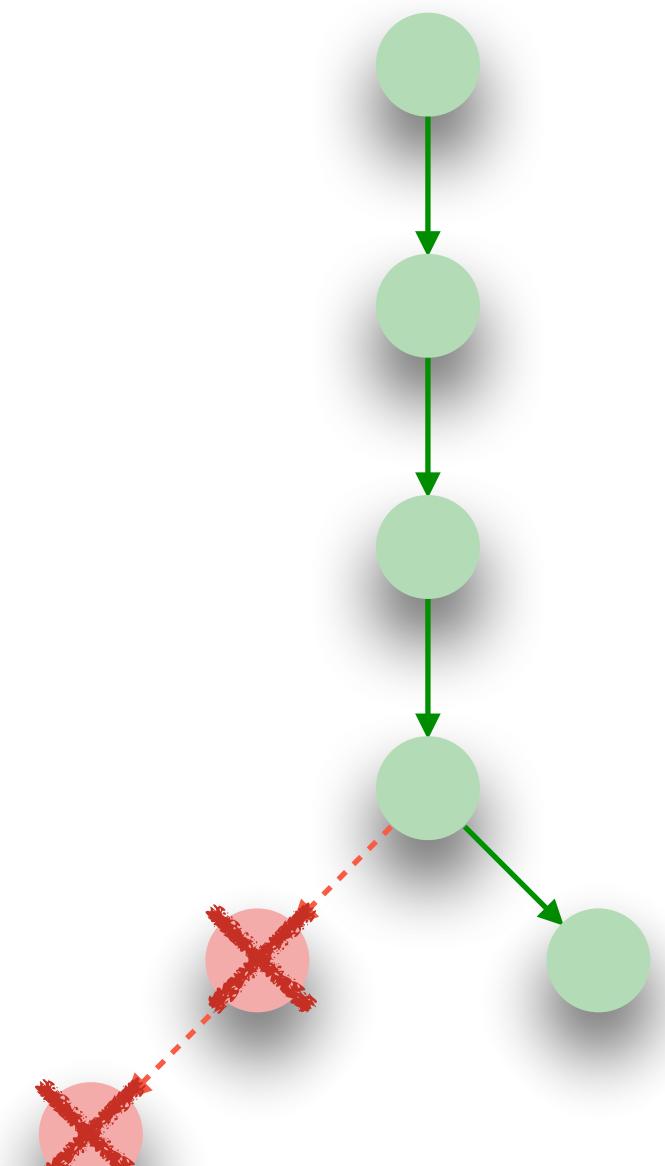
Always true!

# Memory leaks



```

    rax <- A_size
    rcx <- x
    jmp rcx≥rax, END
L1: load rax, A + rcx
          load rax, B + rax
END:
  
```



**Policy**  
 $\mathbf{x}, \mathbf{A\_size}, \mathbf{A}, \mathbf{B}$   
 are public

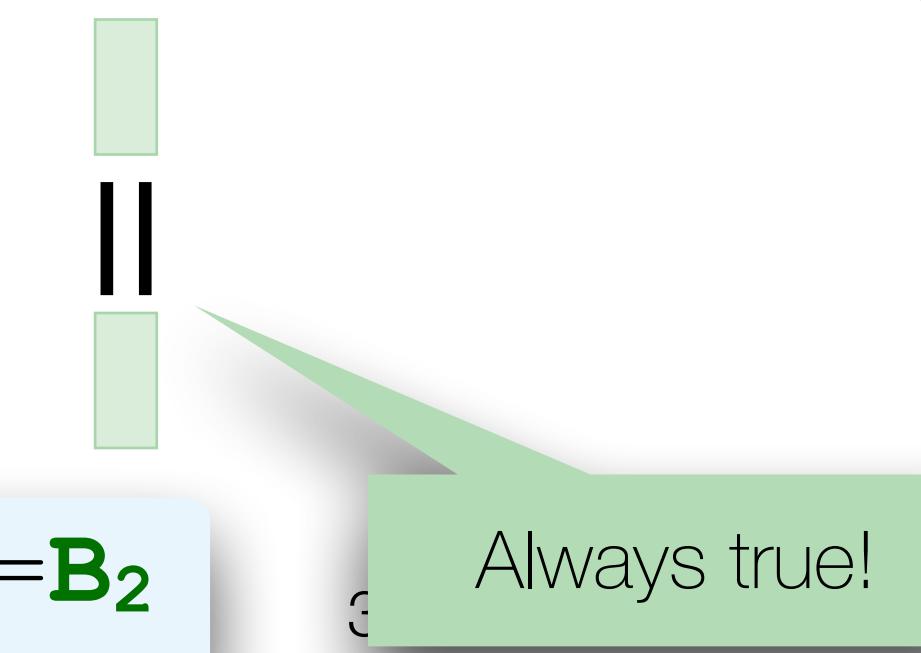
start pc <i>L1</i>	load <b>A+x</b>	load <b>B+A[x]</b>	rollback	pc <i>END</i>
--------------------	-----------------	--------------------	----------	---------------

$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

$$S_1 \models \mathbf{x}_1 \geq \mathbf{A\_size}_1$$

$$S_2 \models \mathbf{x}_2 \geq \mathbf{A\_size}_2$$

$$\mathbf{x}_1 = \mathbf{x}_2 \wedge \mathbf{A\_size}_1 = \mathbf{A\_size}_2 \wedge \mathbf{A}_1 = \mathbf{A}_2 \wedge \mathbf{B}_1 = \mathbf{B}_2$$

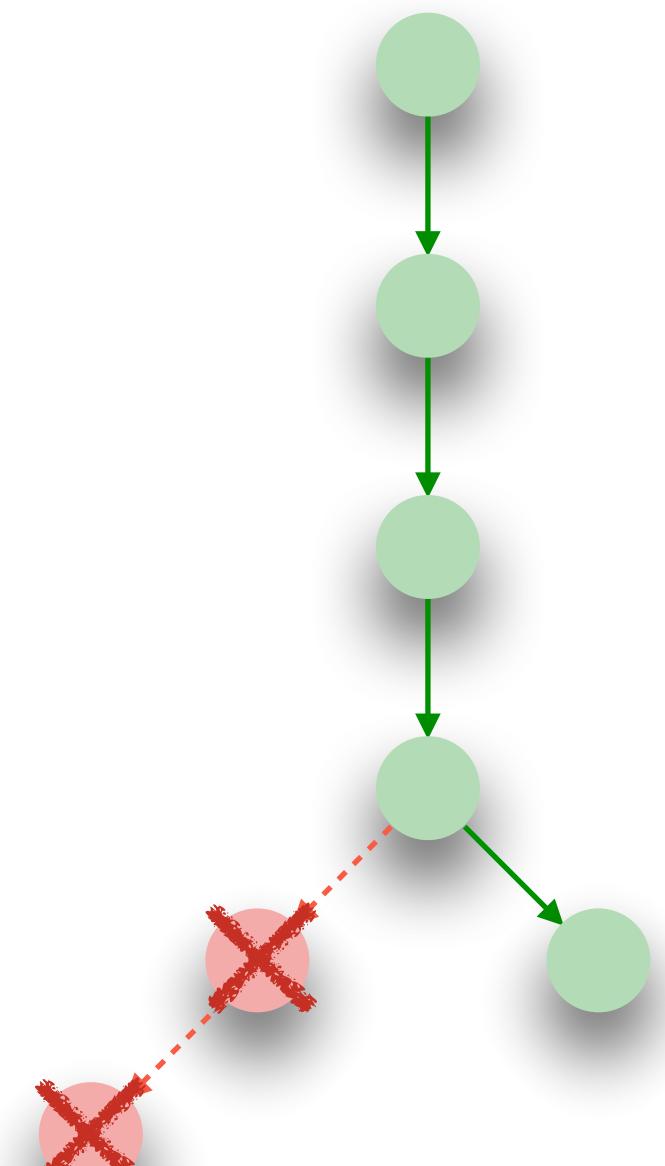


# Memory leaks



```

    rax <- A_size
    rcx <- x
    jmp rcx≥rax, END
L1: load rax, A + rcx
          load rax, B + rax
END:
  
```



**Policy**  
 $\mathbf{x}, \mathbf{A\_size}, \mathbf{A}, \mathbf{B}$   
 are public

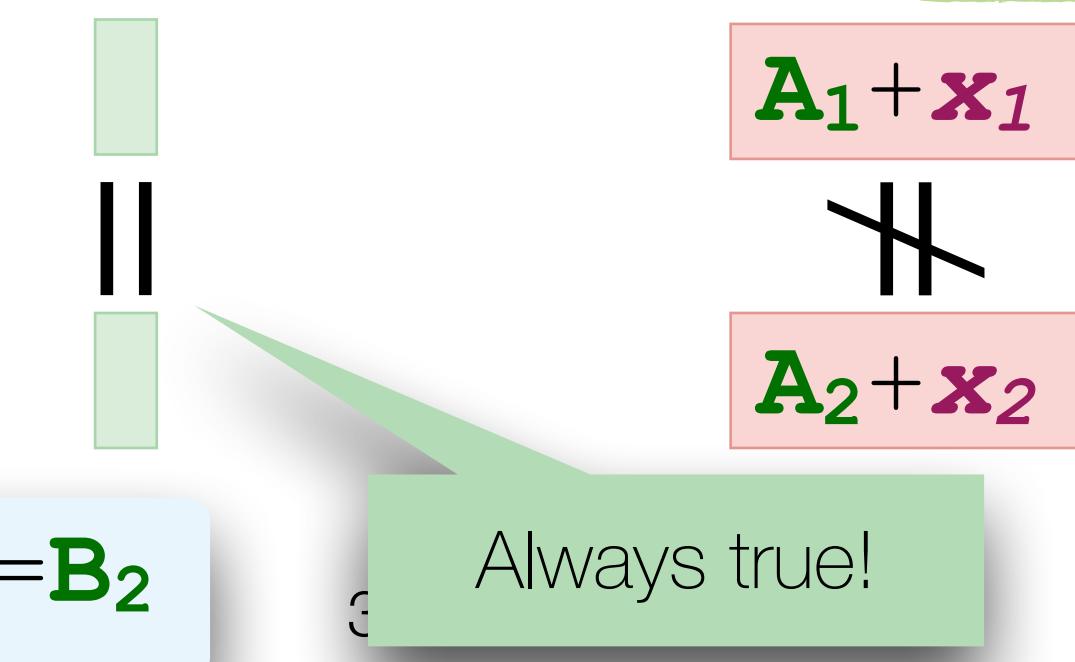
start pc <i>L1</i>	load <b>A+x</b>	load <b>B+A[x]</b>	rollback	pc <i>END</i>
--------------------	-----------------	--------------------	----------	---------------

$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

$$S_1 \models \mathbf{x}_1 \geq \mathbf{A\_size}_1$$

$$S_2 \models \mathbf{x}_2 \geq \mathbf{A\_size}_2$$

$$\mathbf{x}_1 = \mathbf{x}_2 \wedge \mathbf{A\_size}_1 = \mathbf{A\_size}_2 \wedge \mathbf{A}_1 = \mathbf{A}_2 \wedge \mathbf{B}_1 = \mathbf{B}_2$$

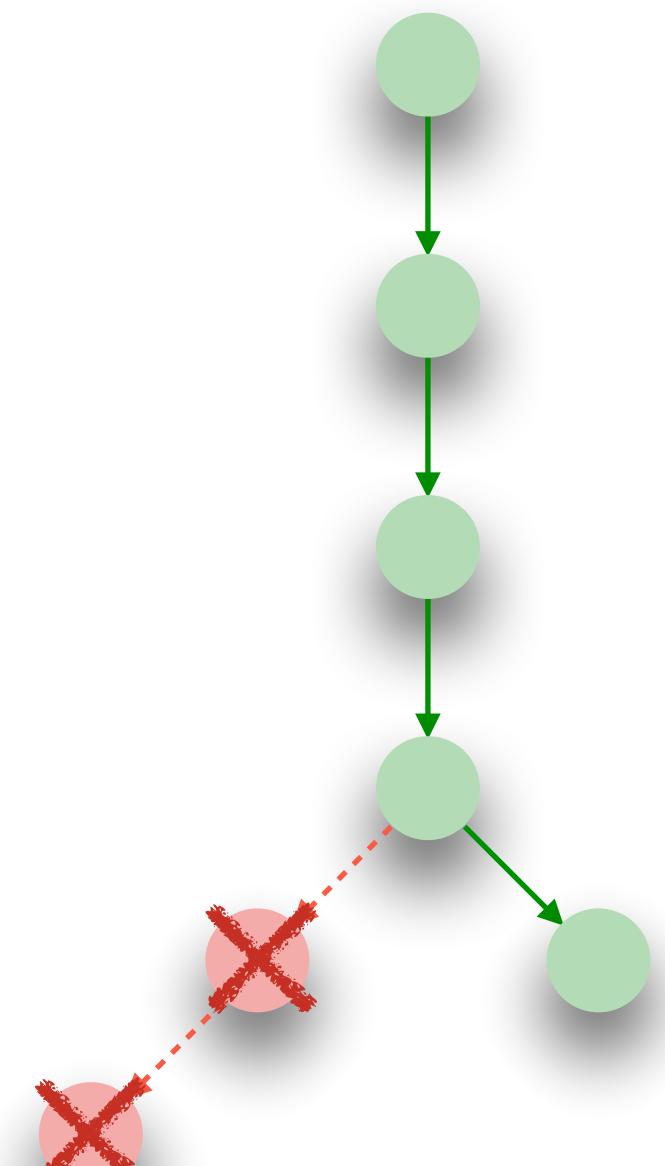


# Memory leaks



```

    rax <- A_size
    rcx <- x
    jmp rcx≥rax, END
L1: load rax, A + rcx
          load rax, B + rax
END:
  
```



**Policy**  
 $\mathbf{x}, \mathbf{A\_size}, \mathbf{A}, \mathbf{B}$   
 are public

start pc	<i>L1</i>	load <b>A+x</b>	load <b>B+A[x]</b>	rollback	pc	<i>END</i>
----------	-----------	-----------------	--------------------	----------	----	------------

$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

$$S_1 \models \mathbf{x}_1 \geq \mathbf{A\_size}_1$$

$$S_2 \models \mathbf{x}_2 \geq \mathbf{A\_size}_2$$

$$\mathbf{x}_1 = \mathbf{x}_2 \wedge \mathbf{A\_size}_1 = \mathbf{A\_size}_2 \wedge \mathbf{A}_1 = \mathbf{A}_2 \wedge \mathbf{B}_1 = \mathbf{B}_2$$

$$\begin{array}{c}
 \boxed{\mathbf{A}_1 + \mathbf{x}_1} \\
 \times \quad \vee \\
 \boxed{\mathbf{A}_2 + \mathbf{x}_2}
 \end{array}$$

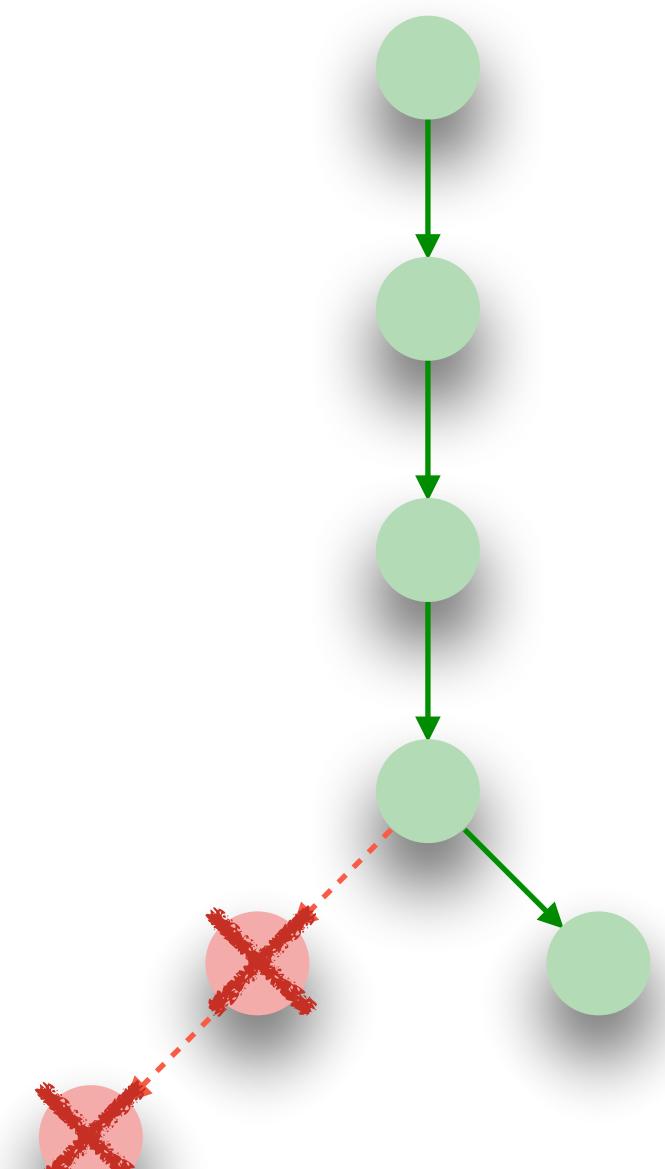
Always true!

# Memory leaks

```

    rax <- A_size
    rcx <- x
    jmp rcx≥rax, END
L1: load rax, A + rcx
          load rax, B + rax
END:

```



**Policy**  
 $\mathbf{x}, \mathbf{A\_size}, \mathbf{A}, \mathbf{B}$   
are public

start pc	<i>L1</i>	load <b>A+x</b>	load <b>B+A[x]</b>	rollback	pc	<i>END</i>
----------	-----------	-----------------	--------------------	----------	----	------------

$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

$$S_1 \models \mathbf{x}_1 \geq \mathbf{A\_size}_1$$

$$S_2 \models \mathbf{x}_2 \geq \mathbf{A\_size}_2$$

$$\mathbf{x}_1 = \mathbf{x}_2 \wedge \mathbf{A\_size}_1 = \mathbf{A\_size}_2 \wedge \mathbf{A}_1 = \mathbf{A}_2 \wedge \mathbf{B}_1 = \mathbf{B}_2$$

$$\begin{array}{c}
\boxed{\mathbf{A}_1 + \mathbf{x}_1} \quad \boxed{\mathbf{B}_1 + \mathbf{A}_1 [\mathbf{x}_1]} \\
\not\models \quad \vee \quad \not\models \\
\boxed{\mathbf{A}_2 + \mathbf{x}_2} \quad \boxed{\mathbf{B}_2 + \mathbf{A}_2 [\mathbf{x}_2]}
\end{array}$$

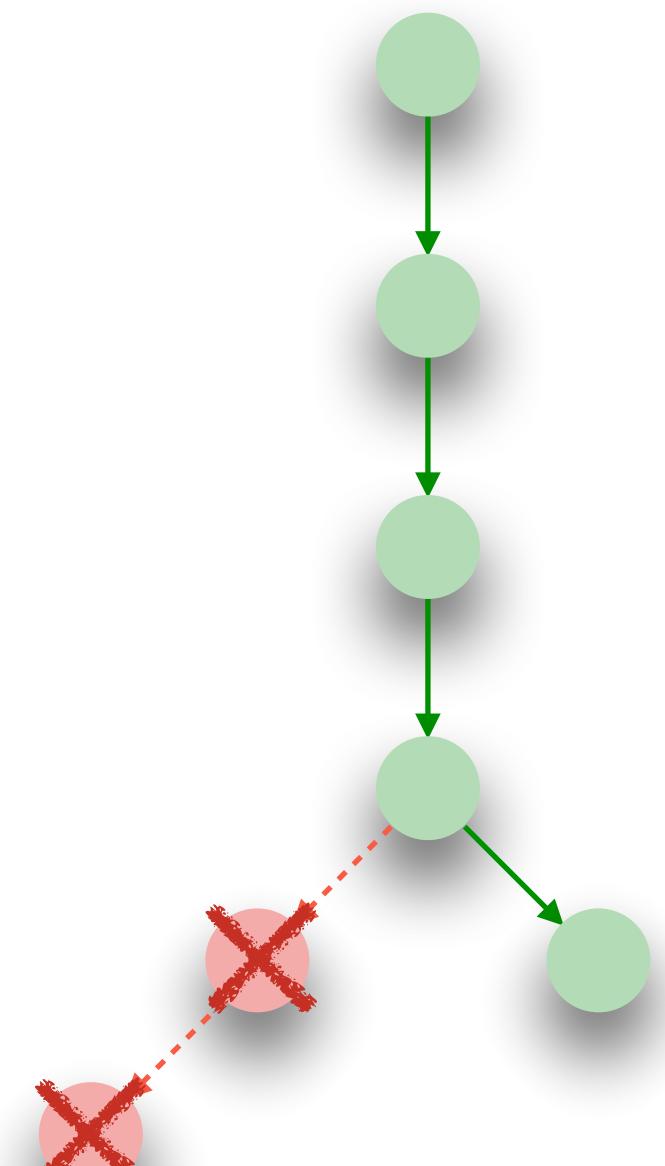
Always true!

# Memory leaks



```

    rax <- A_size
    rcx <- x
    jmp rcx≥rax, END
L1: load rax, A + rcx
          load rax, B + rax
END:
  
```



**Policy**  
 $\mathbf{x}, \mathbf{A\_size}, \mathbf{A}, \mathbf{B}$   
 are public

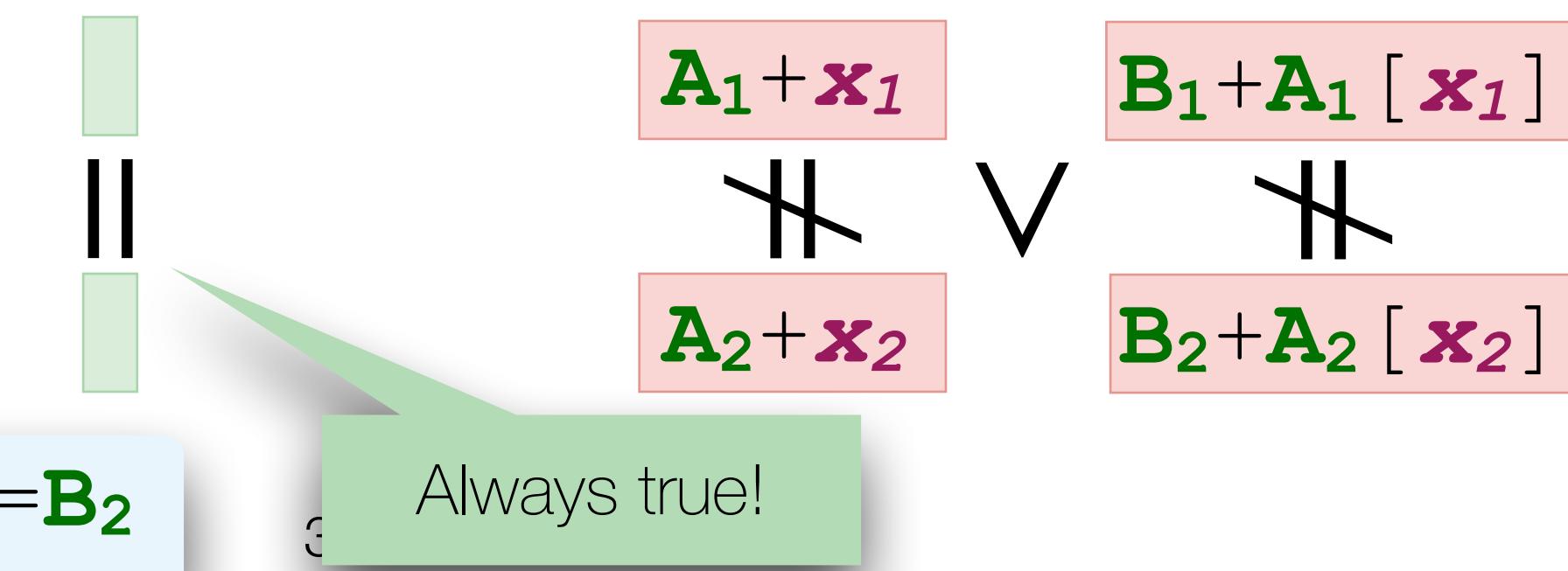
start pc <i>L1</i>	load <b>A+x</b>	load <b>B+A[x]</b>	rollback	pc <i>END</i>
--------------------	-----------------	--------------------	----------	---------------

$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

$$S_1 \models \mathbf{x}_1 \geq \mathbf{A\_size}_1$$

$$S_2 \models \mathbf{x}_2 \geq \mathbf{A\_size}_2$$

$$\mathbf{x}_1 = \mathbf{x}_2 \wedge \mathbf{A\_size}_1 = \mathbf{A\_size}_2 \wedge \mathbf{A}_1 = \mathbf{A}_2 \wedge \mathbf{B}_1 = \mathbf{B}_2$$



# Reasoning about arbitrary oracles

# Reasoning about arbitrary oracles

Always-mispredict  
speculative semantics

Mispredict ***all*** branch  
instructions

Fixed speculative window

Rollback of every transaction

# Reasoning about arbitrary oracles

Always-mispredict  
speculative semantics

Mispredict ***all*** branch  
instructions

Fixed speculative window

Rollback of every transaction

Always-mispredict is **worst-case**

$$P_{am}(s) = P_{am}(s') \iff$$

$$\forall O. P_{spec}(s, O) = P_{spec}(s', O)$$

# Reasoning about arbitrary oracles

Always-mispredict  
speculative semantics

Mispredict ***all*** branch  
instructions

Fixed speculative window

Rollback of every transaction

Always-mispredict is **worst-case**

$$P_{am}(s) = P_{am}(s') \iff$$

$$\forall O. P_{spec}(s, O) = P_{spec}(s', O)$$

If program **P** satisfies

$$\begin{aligned} \forall s, s'. P_{\text{non-spec}}(s) &= P_{\text{non-spec}}(s') \\ \Rightarrow P_{am}(s) &= P_{am}(s') \end{aligned}$$

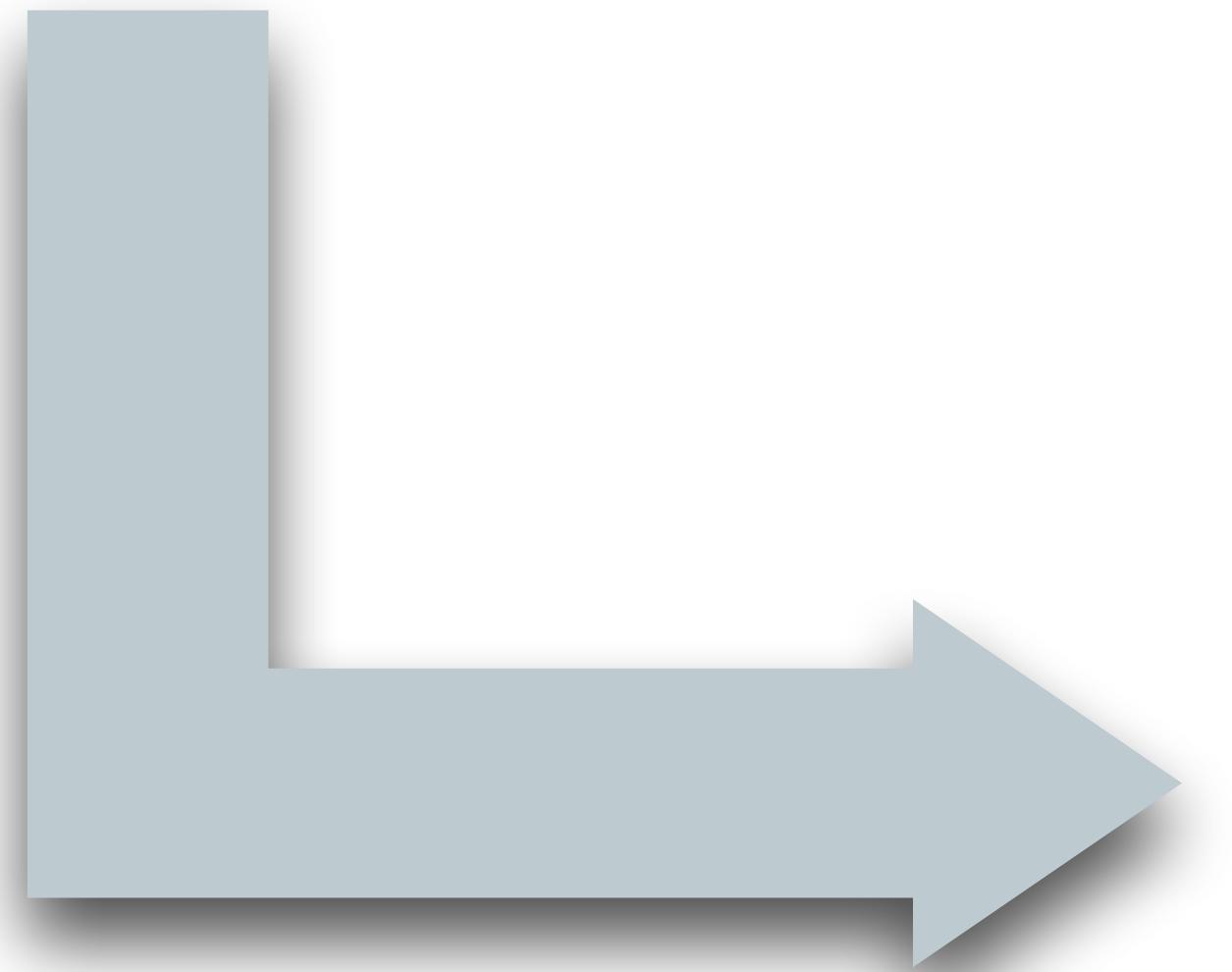
then **P** satisfies **SNI** w.r.t. all **O**

# Example #01 - SLH

```
if (x < A_size)  
y = B[A[x] * 512]
```

# Example #01 - SLH

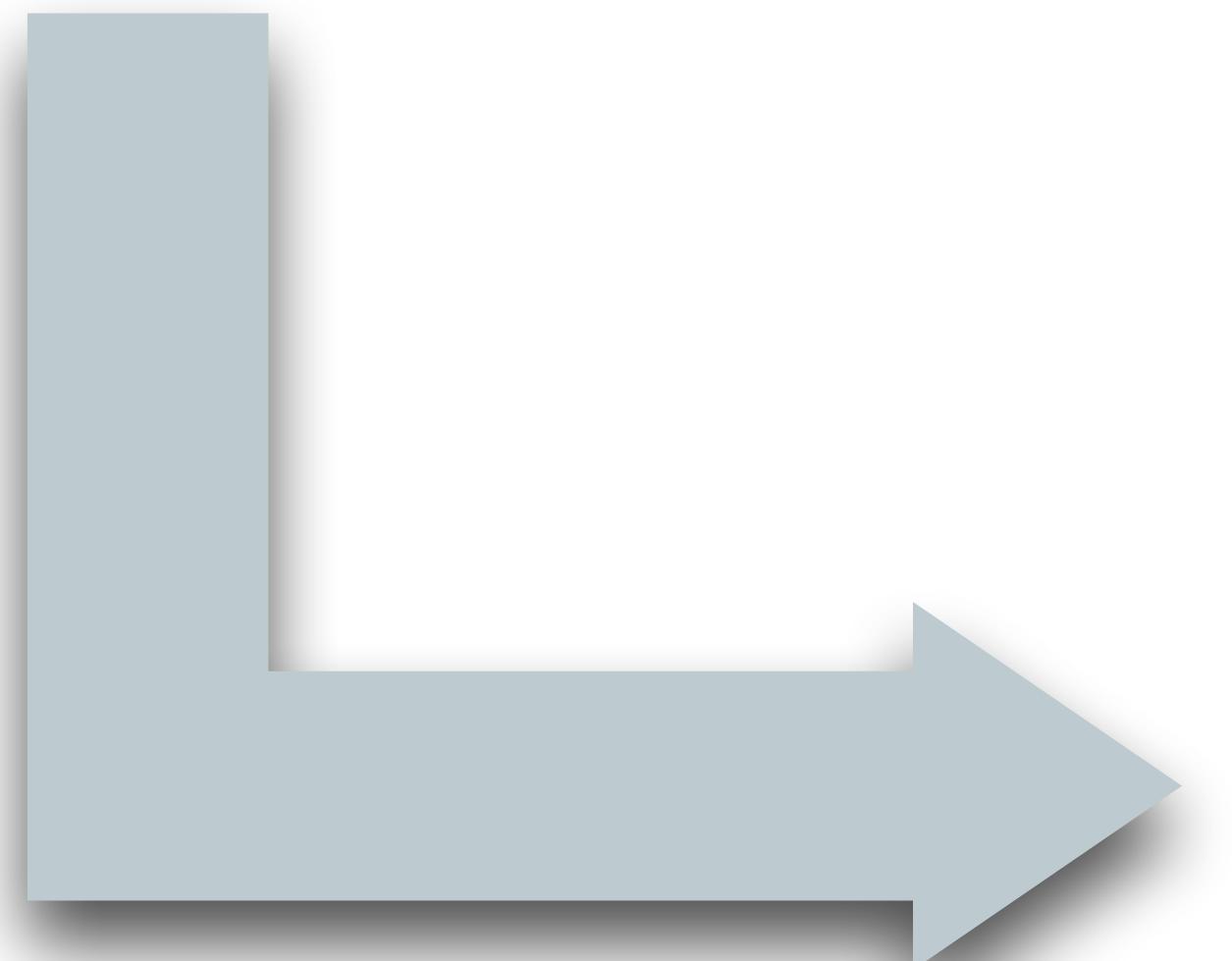
```
if (x < A_size)
    y = B[A[x] * 512]
```



mov	rax, A_size
mov	rcx, x
mov	rdx, 0
cmp	rcx, rax
jae	END
cmoveae	-1, rdx
mov	rax, A[rcx]
shl	rax, 9
or	rax, rdx
mov	rax, B[rax]

# Example #01 - SLH

```
if (x < A_size)
    y = B[A[x] * 512]
```



rax is -1 whenever **x** ≥ **A\_size**  
We can prove security

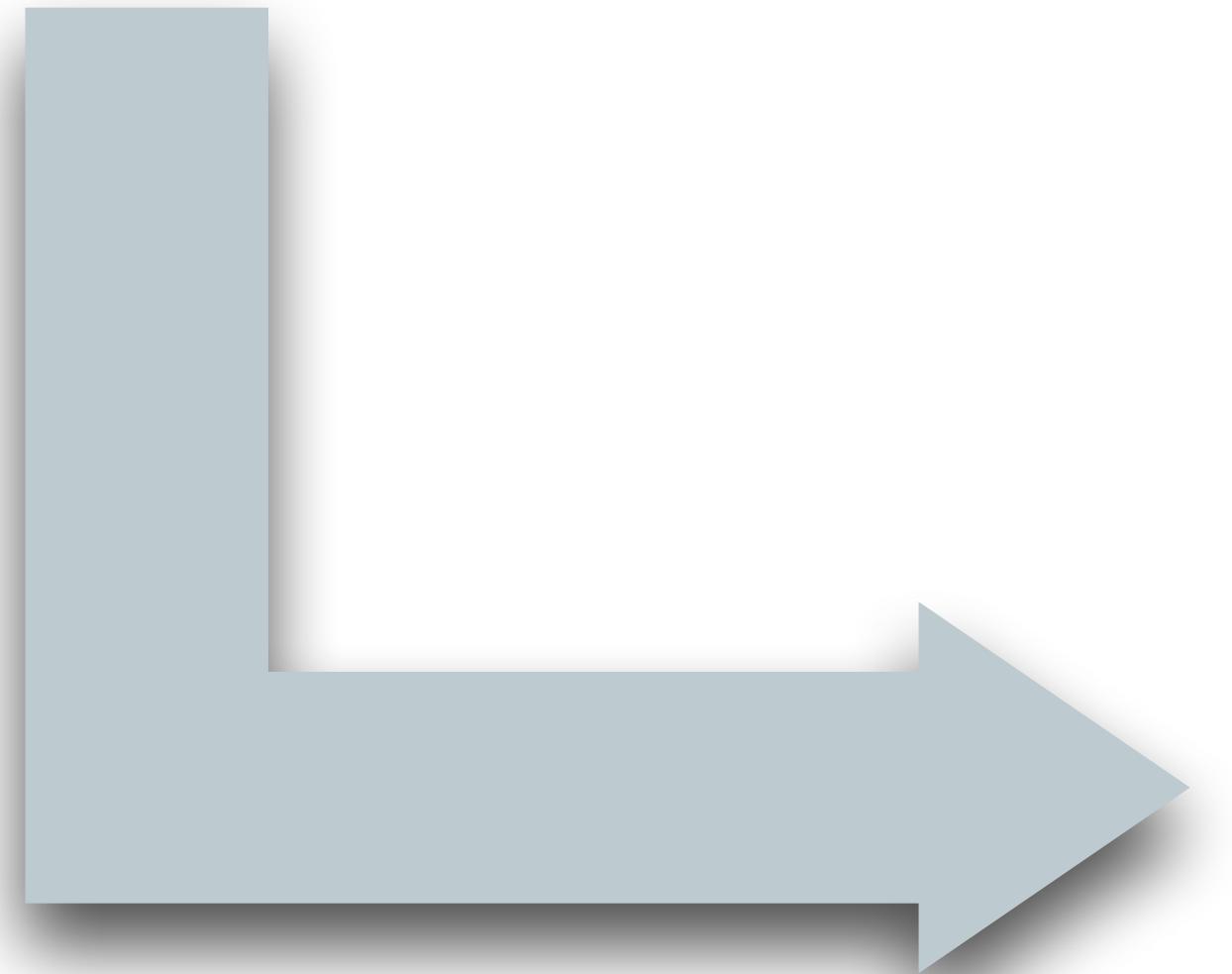
mov	rax, <b>A_size</b>
mov	rcx, <b>x</b>
mov	rdx, 0
cmp	rcx, rax
jae	<i>END</i>
cmove	-1, rdx
mov	rax, <b>A[rcx]</b>
shl	rax, 9
or	rax, rdx
mov	rax, <b>B[rax]</b>

# Example #10 - SLH

```
if (x < A_size)
    if (A[x]==k)
        y = B[0]
```

# Example #10 - SLH

```
if (x < A_size)
  if (A[x]==k)
    y = B[0]
```



mov	rax, A_size
mov	rcx, x
mov	rdx, 0
cmp	rcx, rax
jae	END
cmovae	-1, rdx
mov	rax, A[rcx]
jne	rax, END
cmovne	-1, rdx
mov	rax, [B]

# Example #10 - SLH

```
if (x < A_size)
    if (A[x]==k)
        y = B[0]
```

Leaks A[x]==0 via  
control-flow  
We detect the leak!

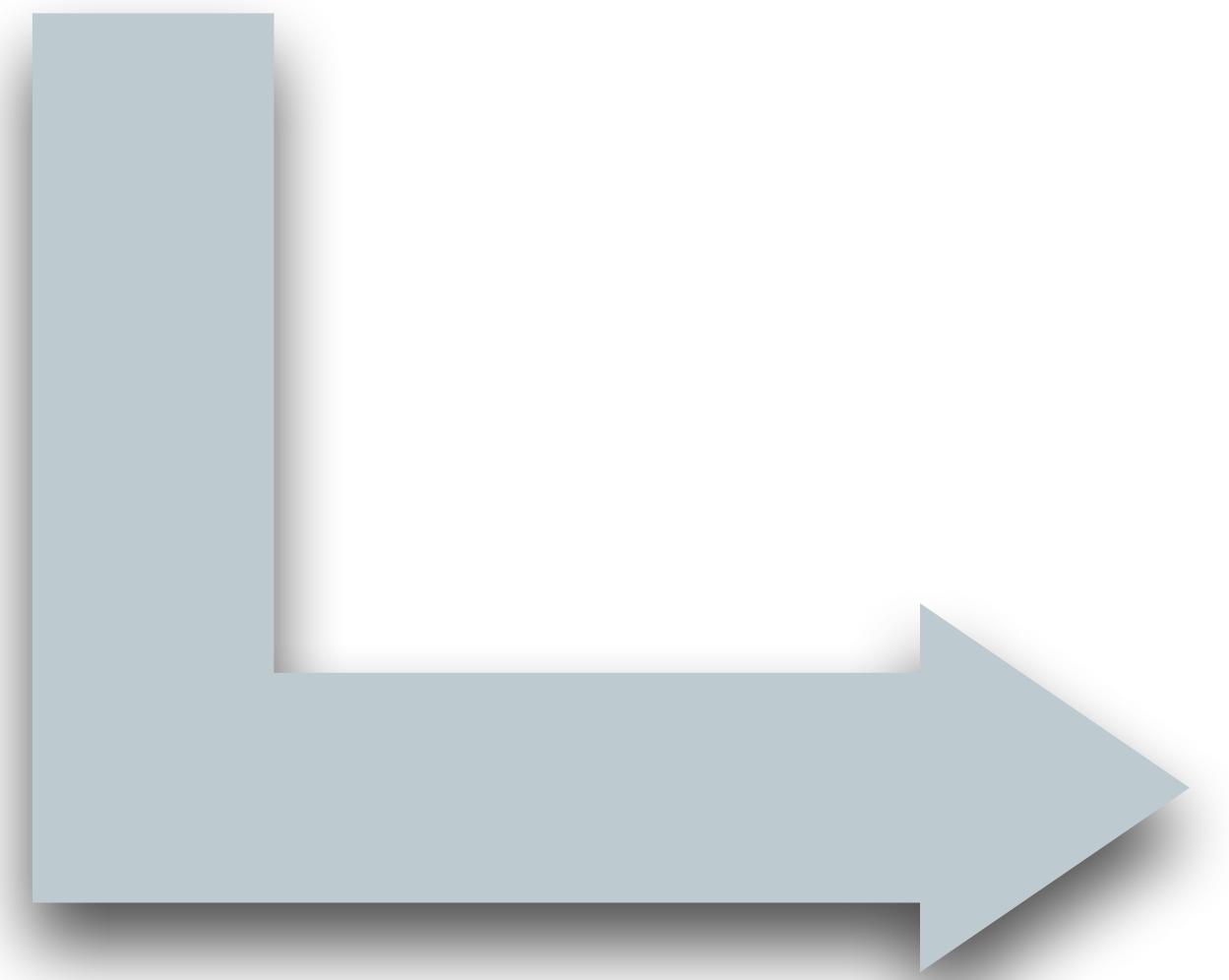
mov	rax, A_size
mov	rcx, x
mov	rdx, 0
cmp	rcx, rax
jae	END
cmovae	-1, rdx
mov	rax, A[rcx]
jne	rax, END
cmovne	-1, rdx
mov	rax, [B]

# Example #08 - FEN

```
y = B[A[x < A_size ? (x+1) : 0] * 512]
```

# Example #08 - FEN

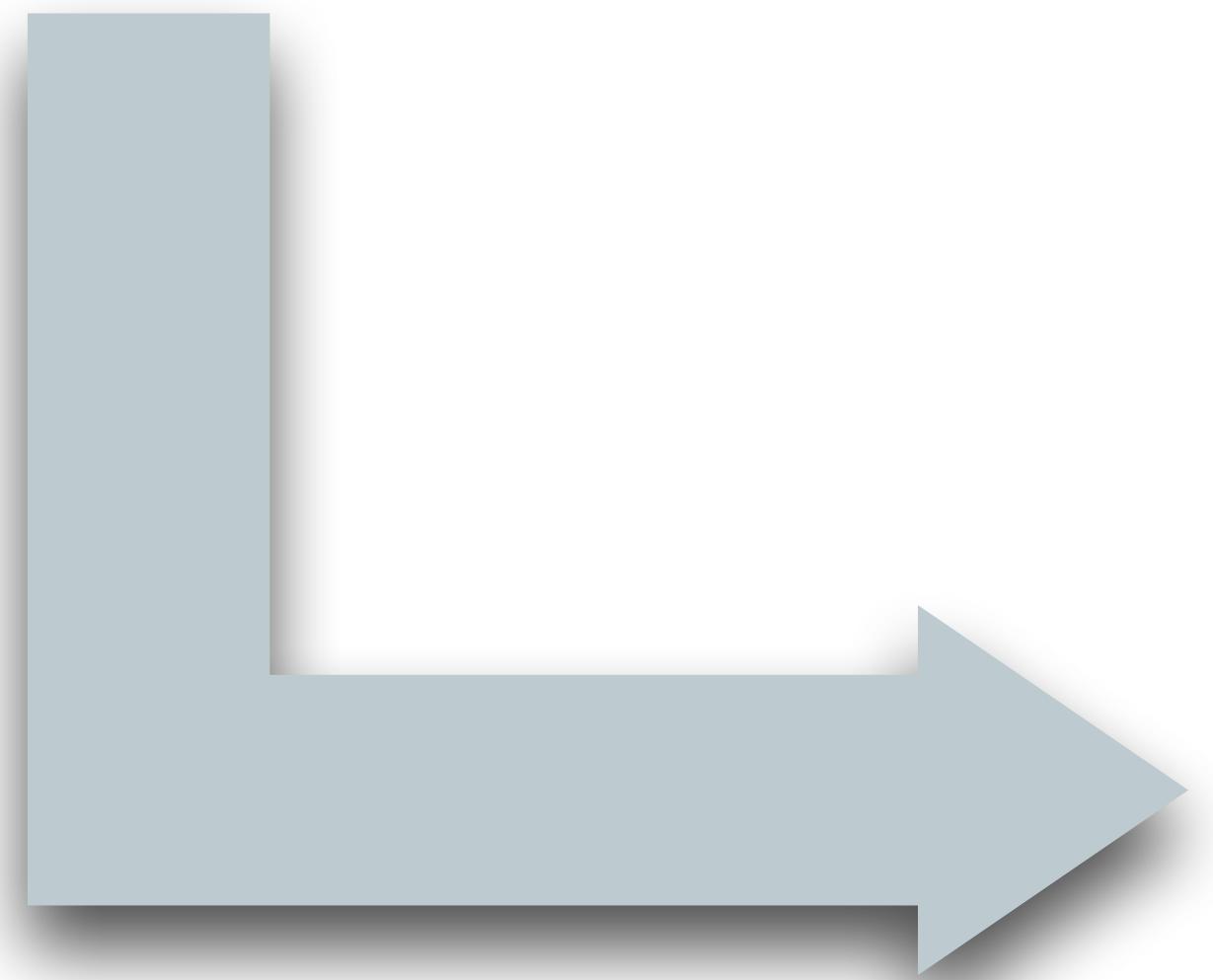
```
y = B[A[x < A_size ? (x+1) : 0] * 512]
```



mov	rax, <b>A_size</b>
mov	rcx, <b>x</b>
lea	rcx, [rcx+1]
xor	rdx, rdx
cmp	rcx, rax
cmoveae	rdx, rcx
mov	rax, <b>A[rdx]</b>
shl	rax, 9
lfence	
mov	rax, <b>B[rax]</b>

# Example #08 - FEN

```
y = B[A[x < A_size ? (x+1) : 0] * 512]
```



lfence is unnecessary

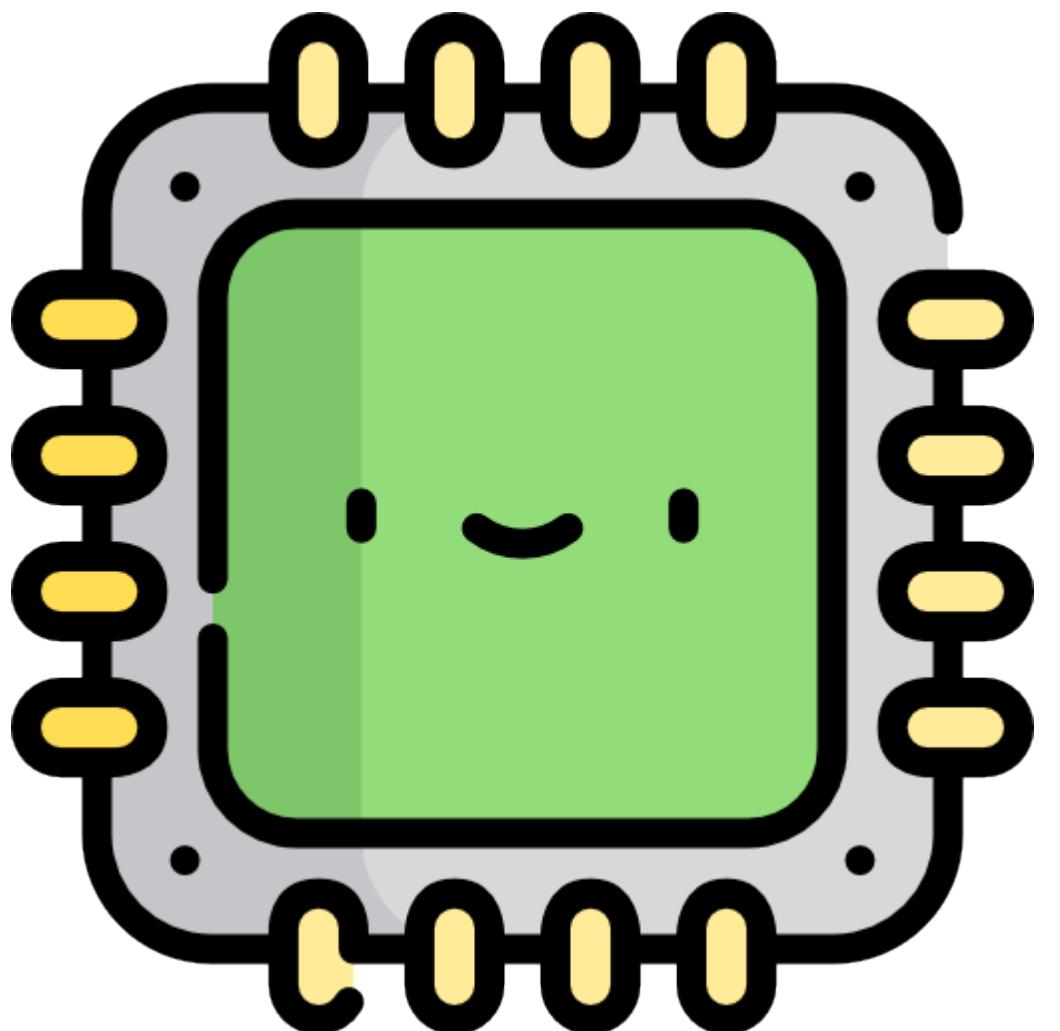
mov	rax, <b>A_size</b>
mov	rcx, <b>x</b>
lea	rcx, [rcx+1]
xor	rdx, rdx
cmp	rcx, rax
cmovae	rdx, rcx
mov	rax, <b>A[rdx]</b>
shl	rax, 9
lfence	
mov	rax, <b>B[rax]</b>

# Spectre V1

```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```

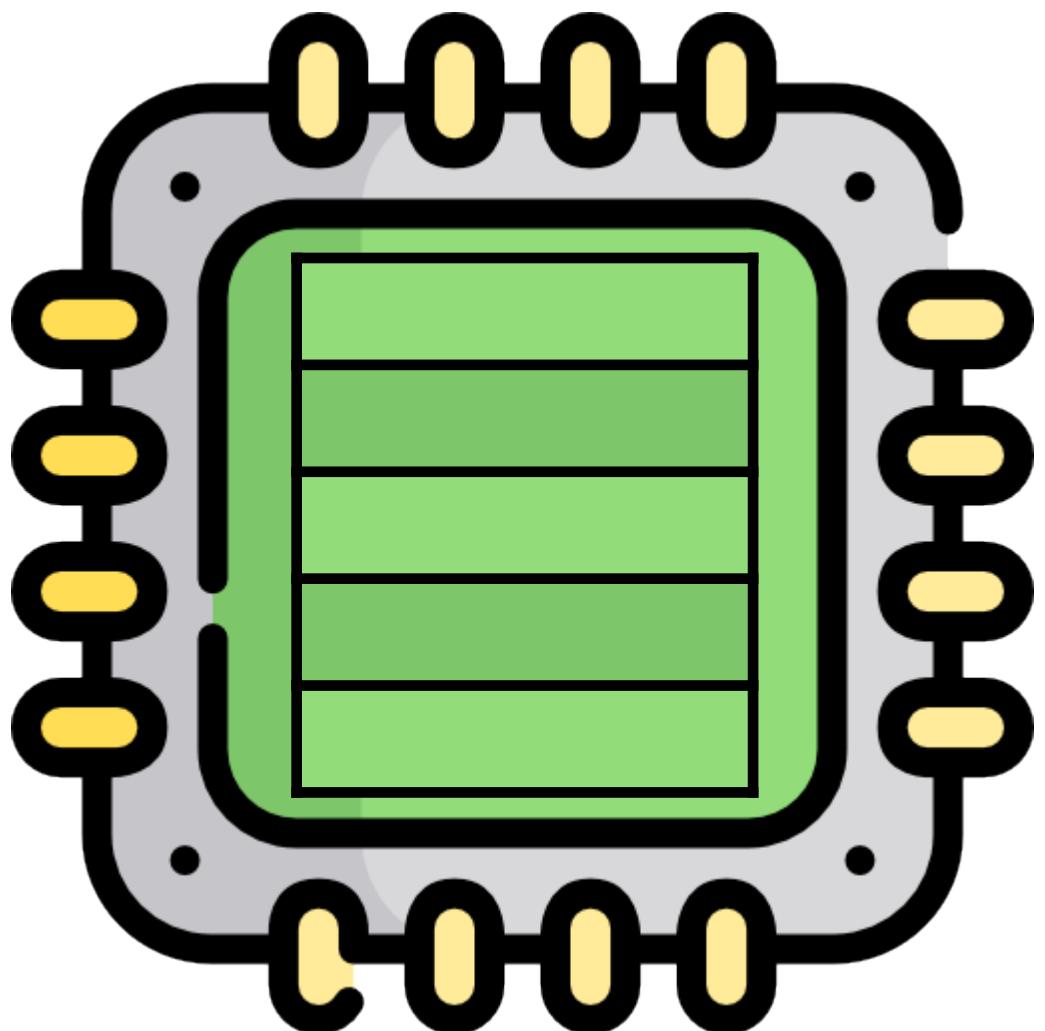
# Spectre V1

```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



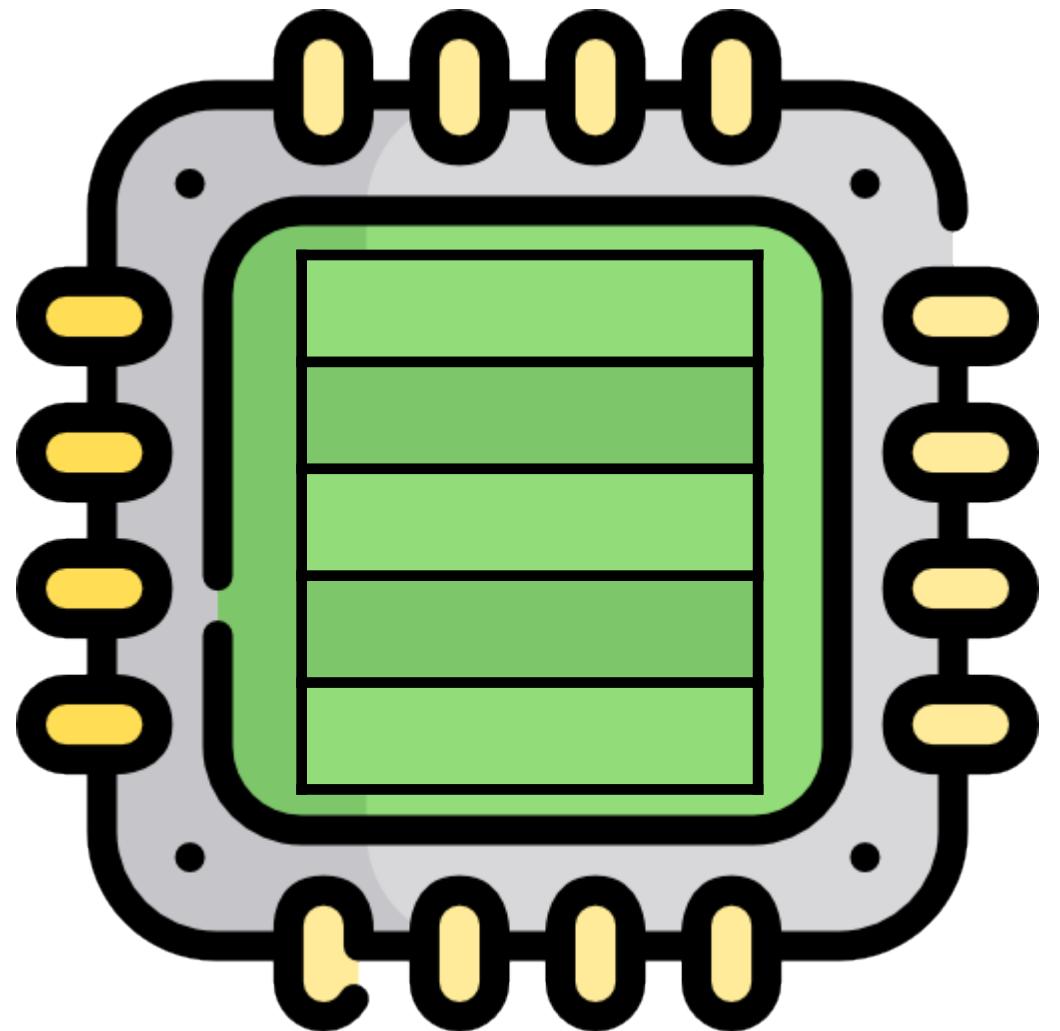
# Spectre V1

```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```

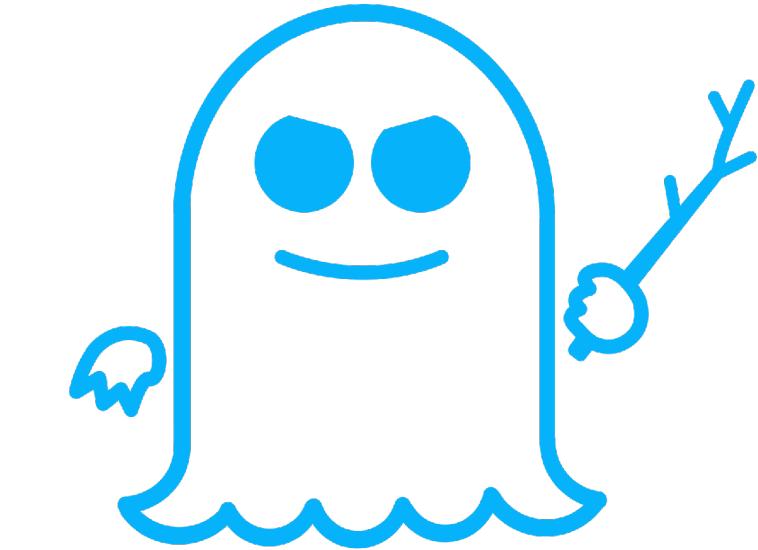


# Spectre V1

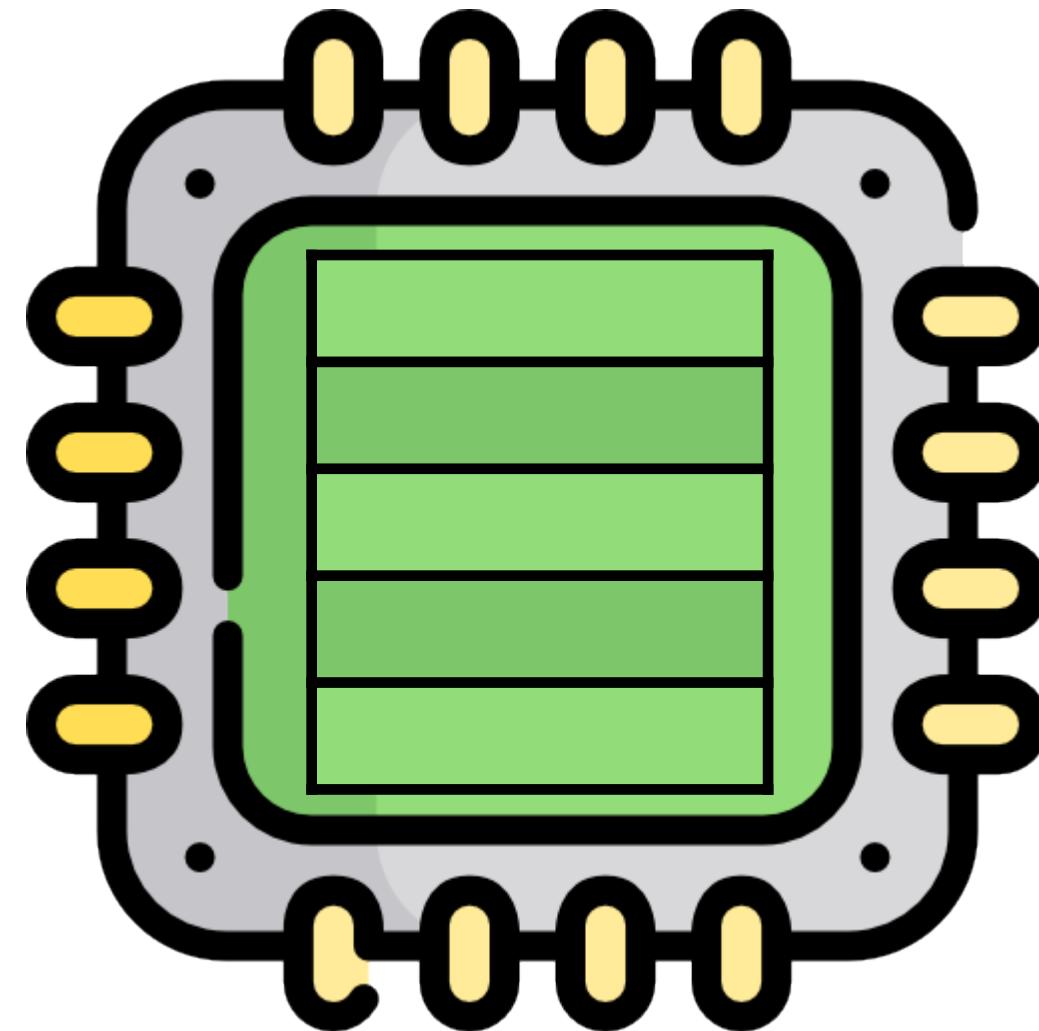
```
void f(int x)
  if (x < A_size)
    y = B[A[x]]
```



# Spectre V1

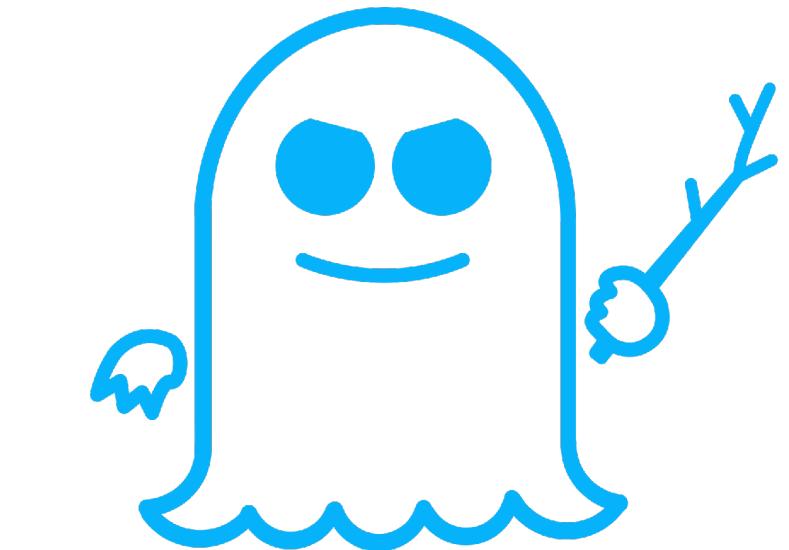
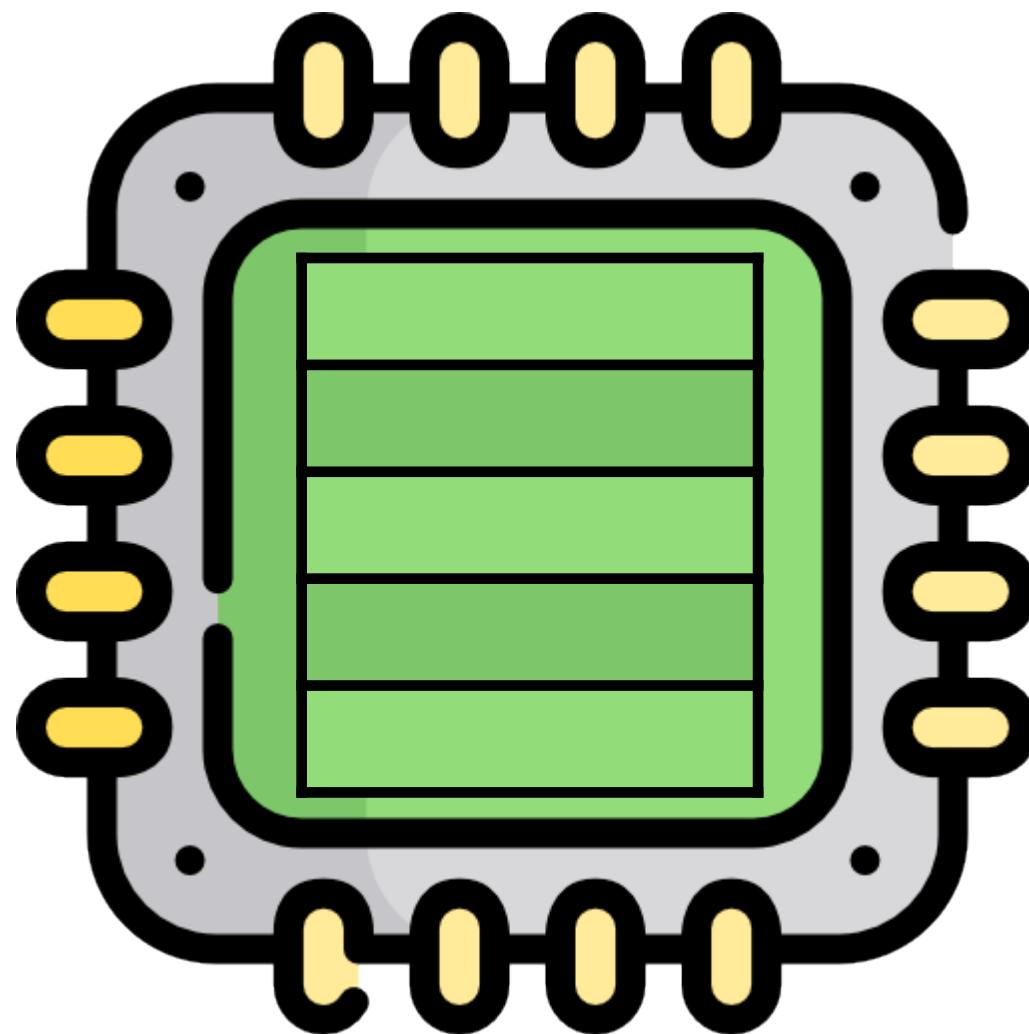


```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



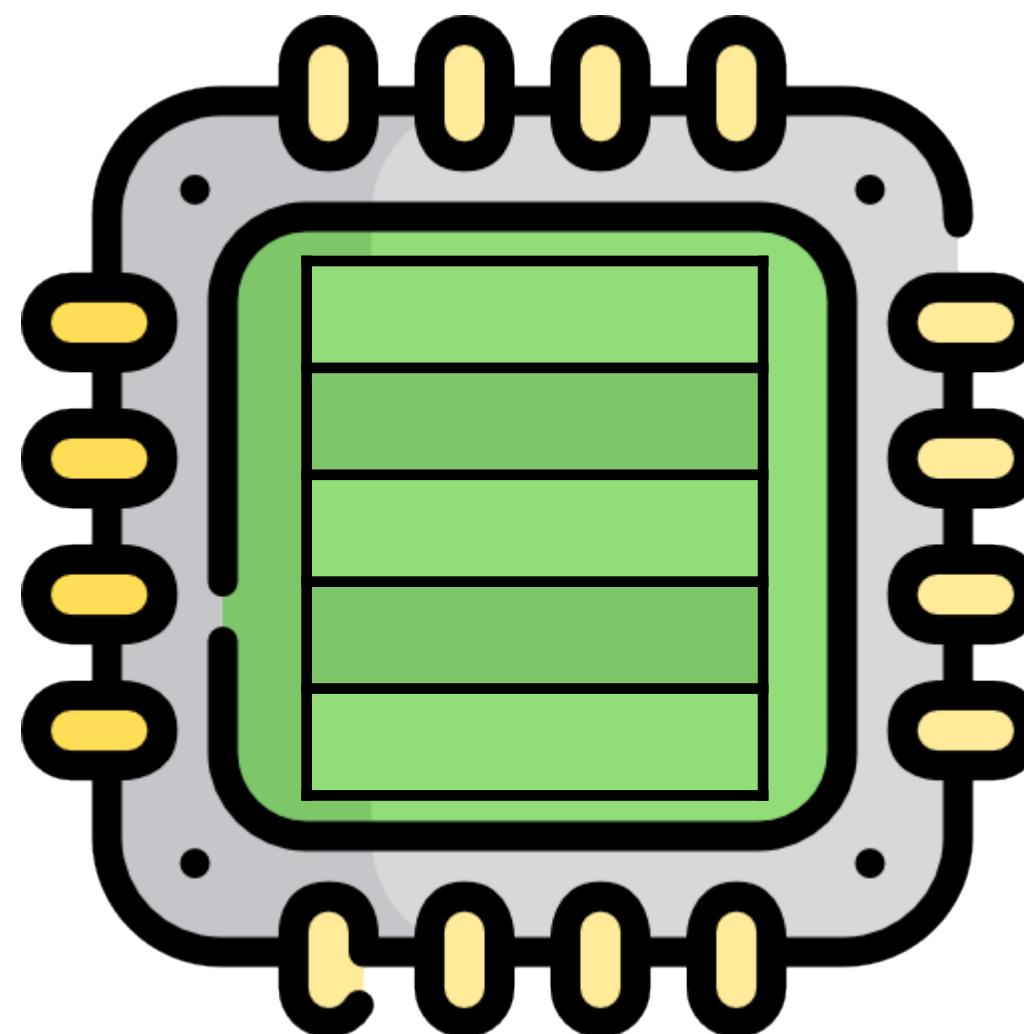
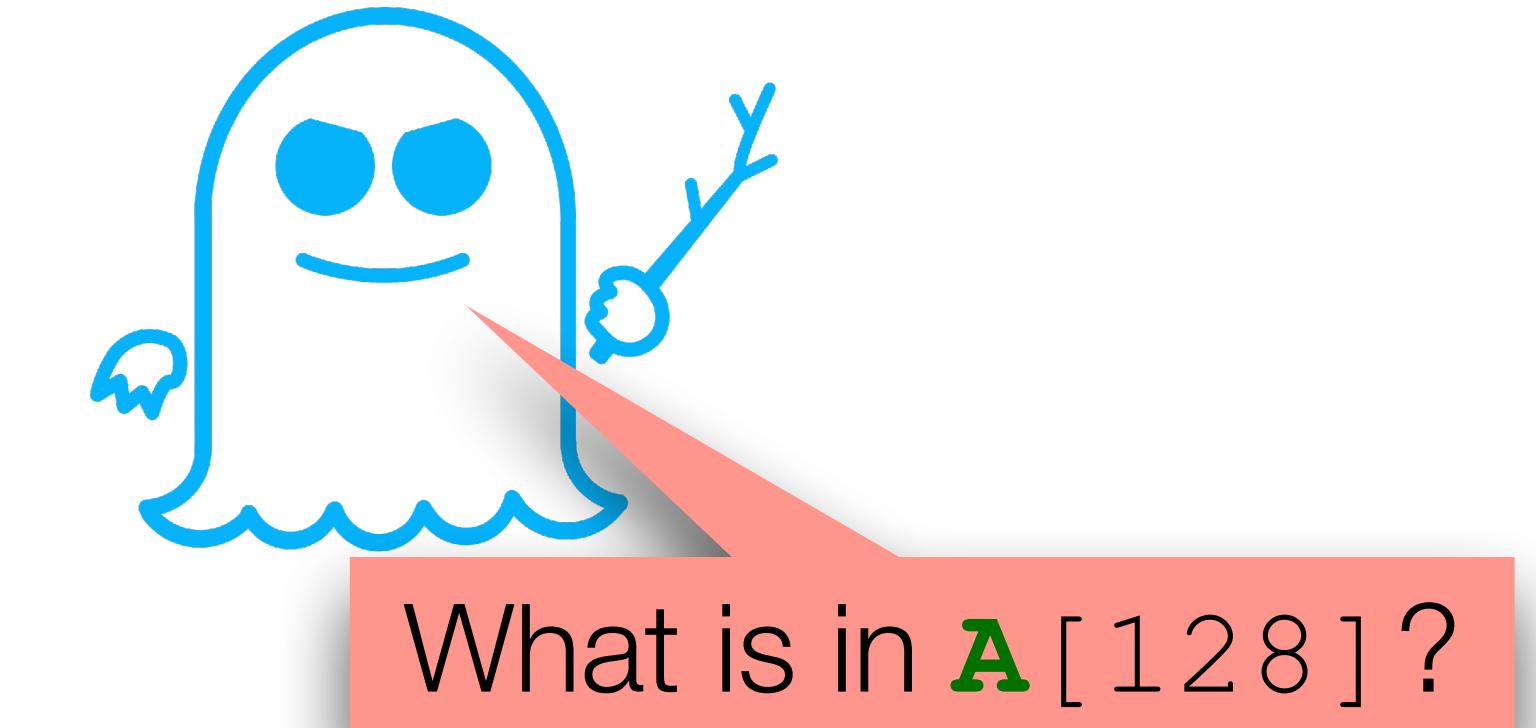
# Spectre V1

```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



# Spectre V1

```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



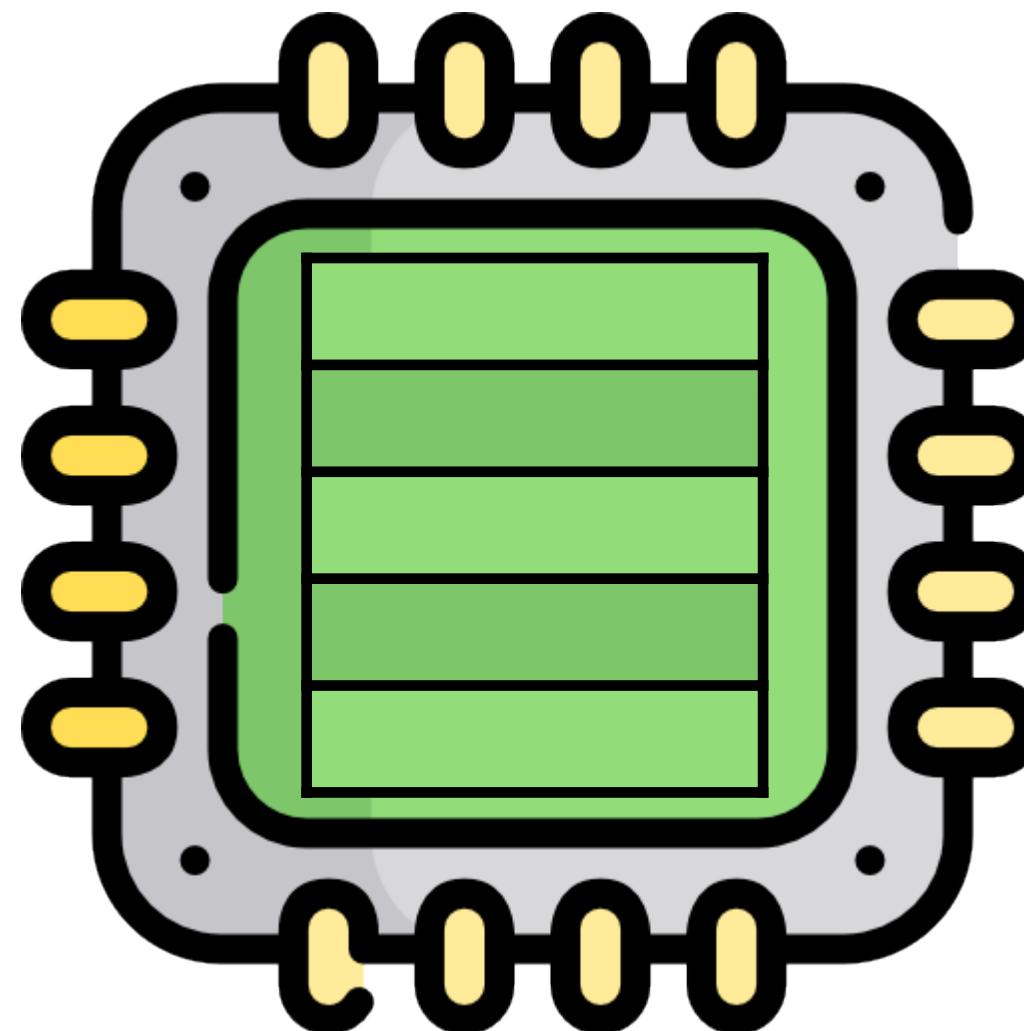
# Spectre V1

```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



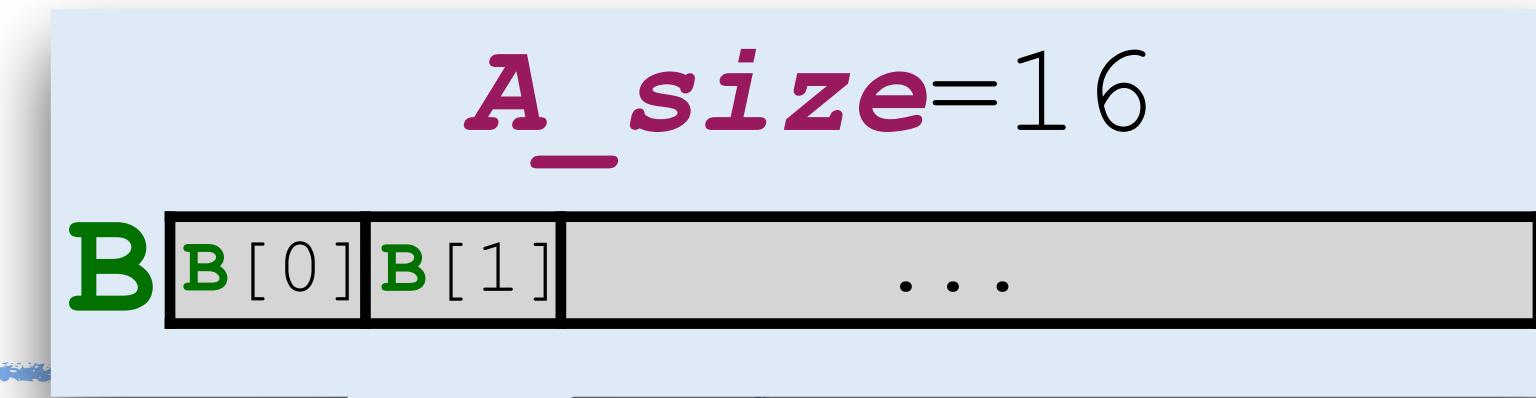
What is in  $A[128]$ ?

1) Training



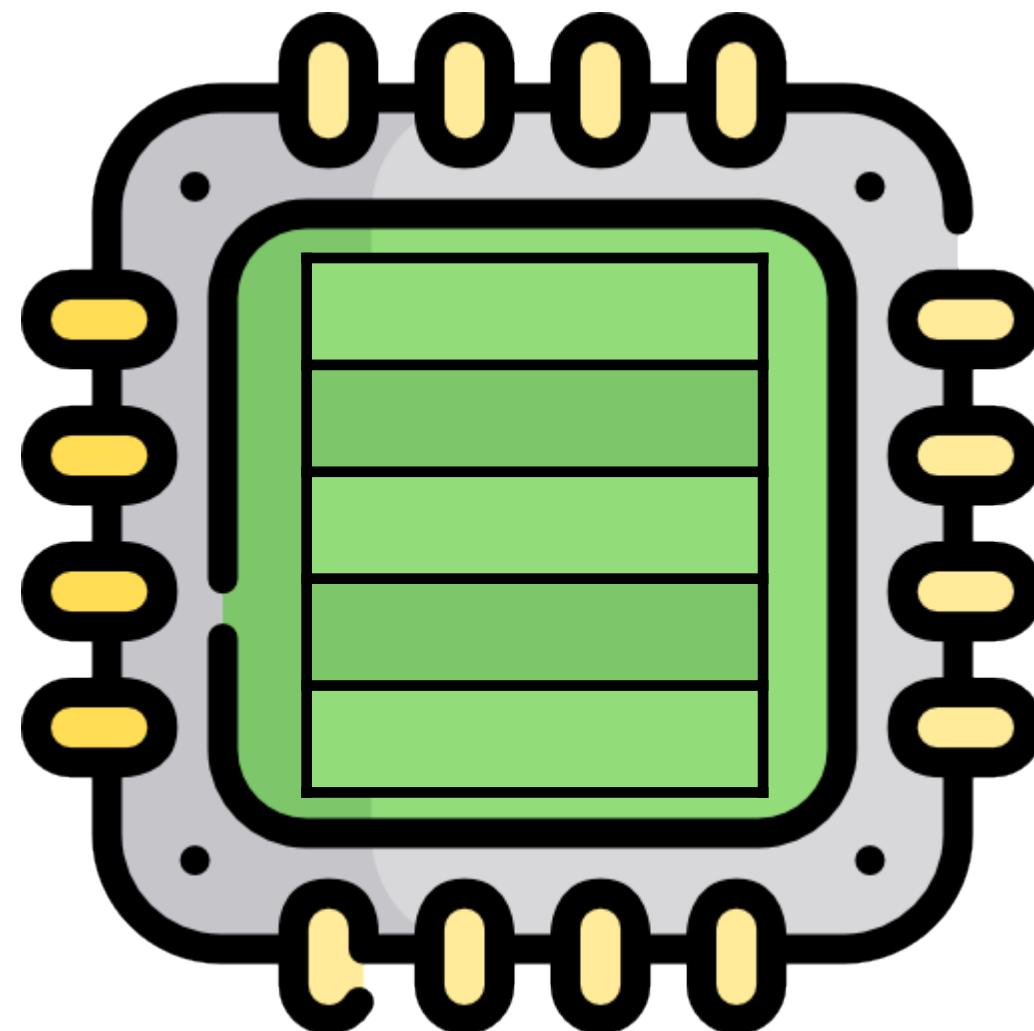
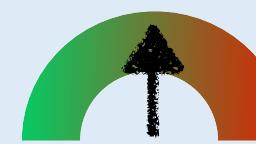
# Spectre V1

```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



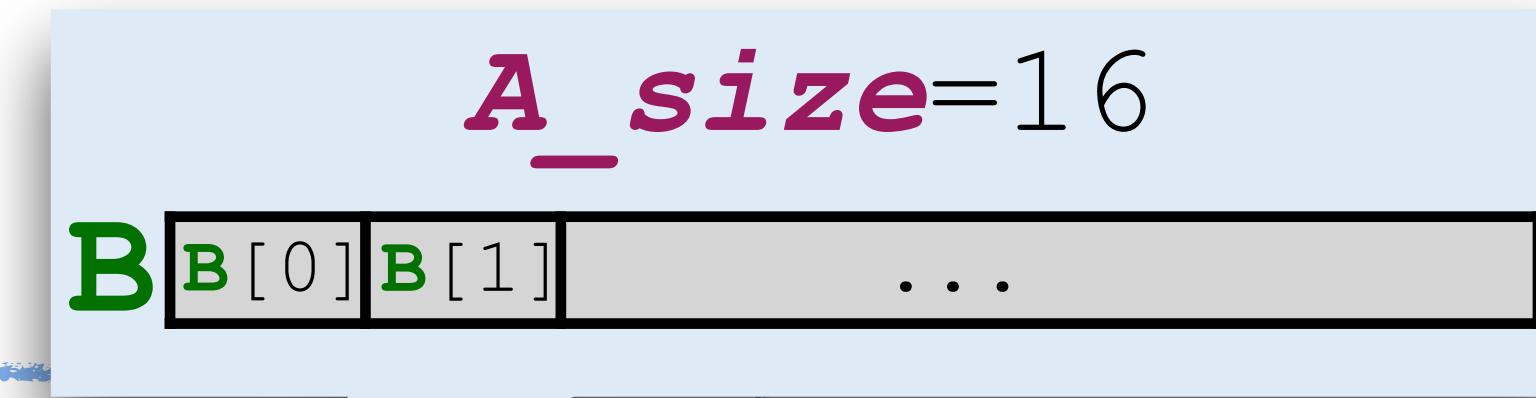
What is in  $A[128]$ ?

1) Training

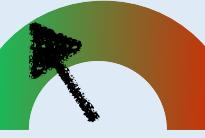


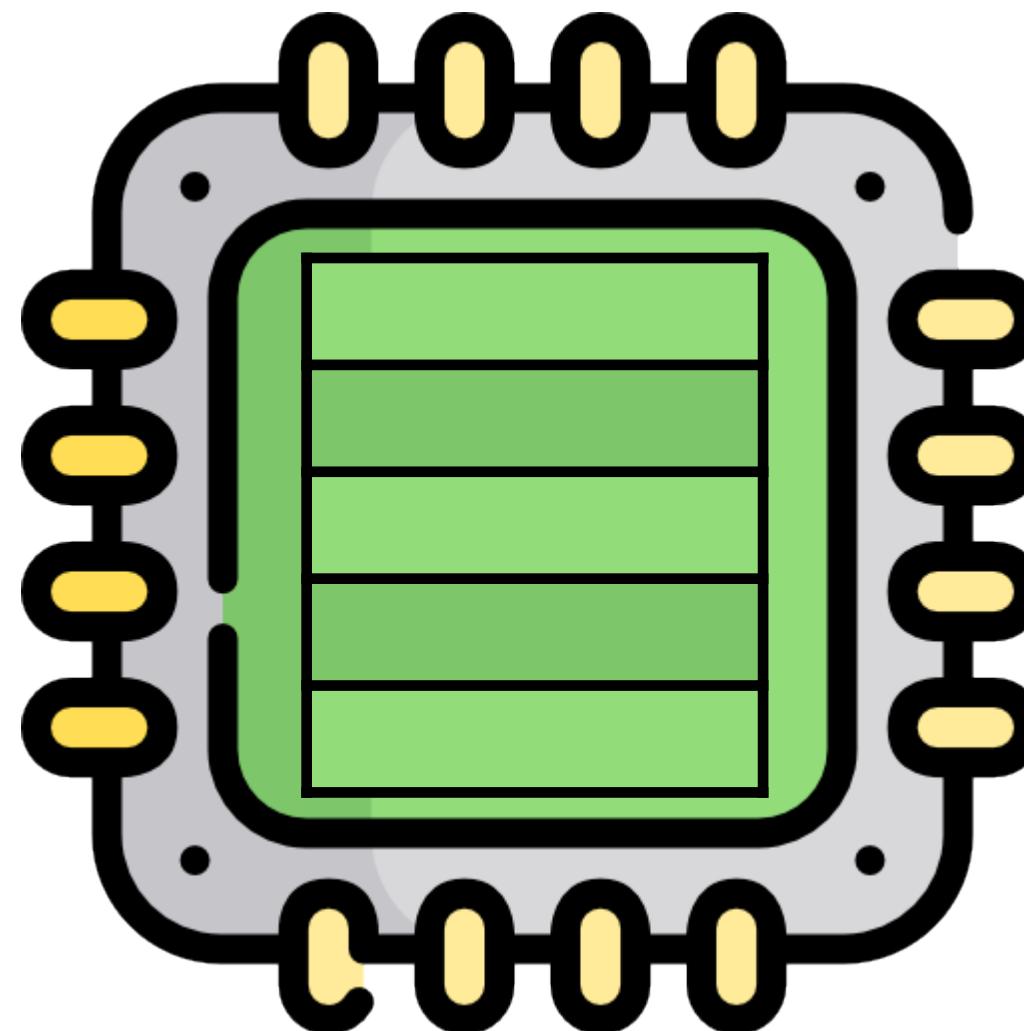
# Spectre V1

```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



What is in A[128]?

1) Training  f(0);



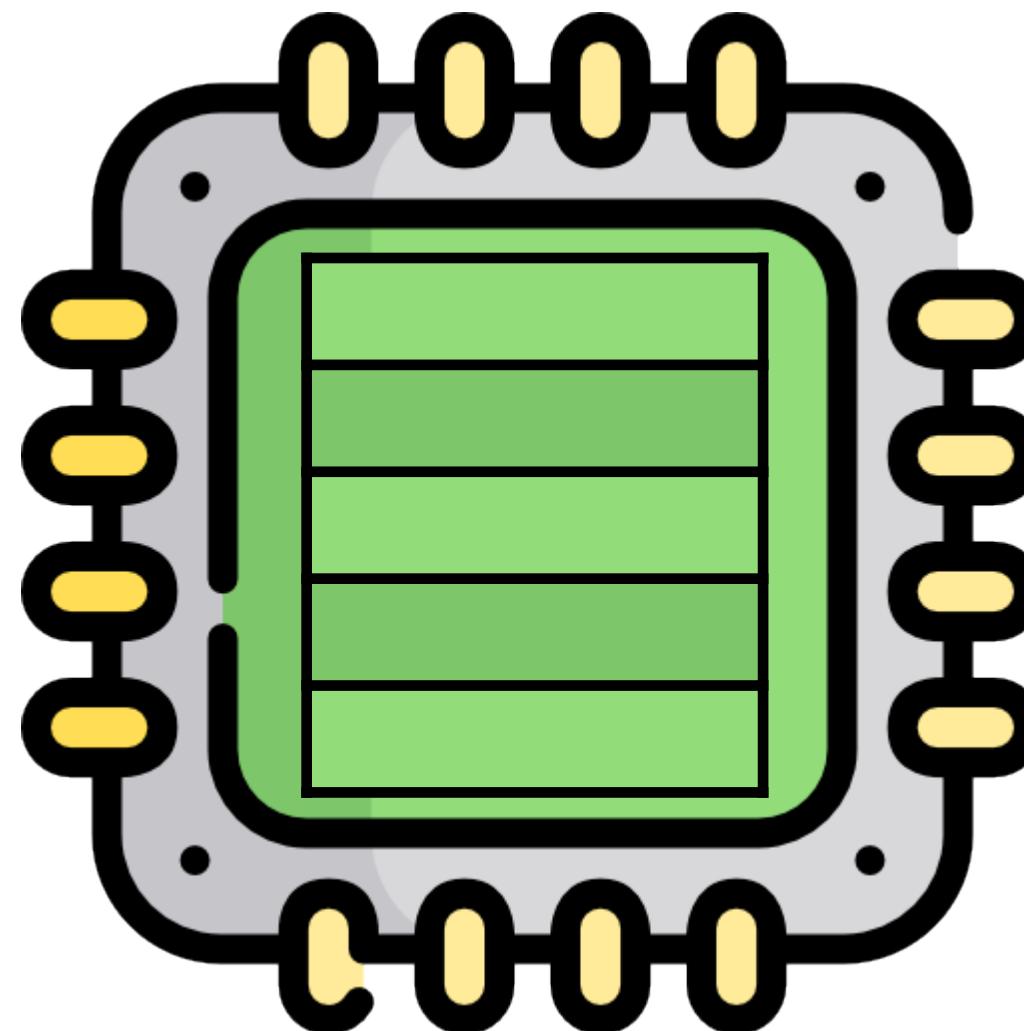
# Spectre V1

```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



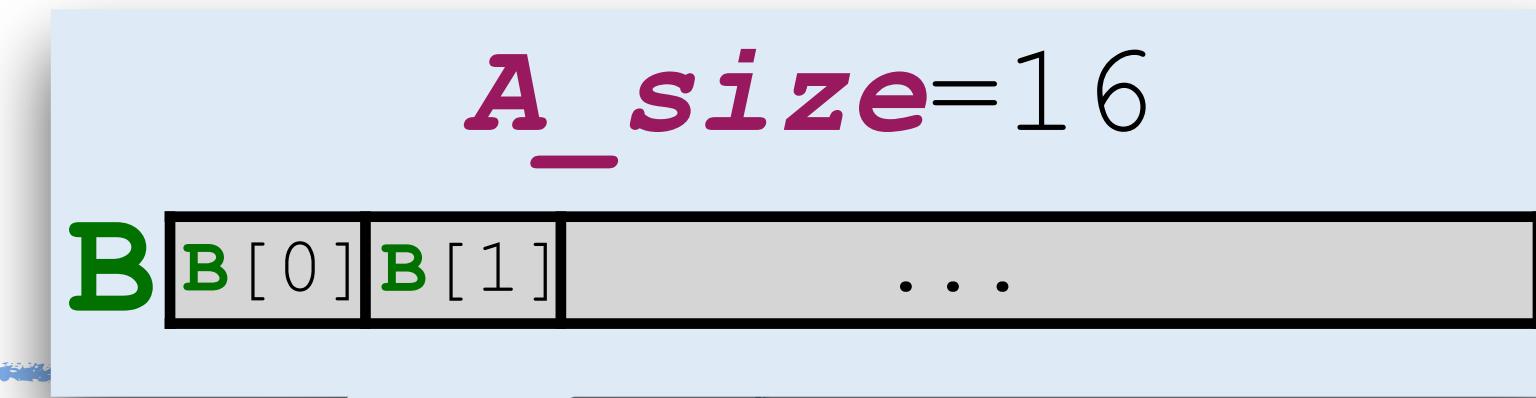
What is in  $A[128]$ ?

1) Training  $f(0); f(1);$



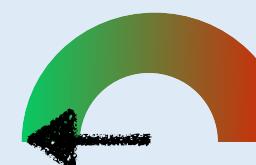
# Spectre V1

```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```

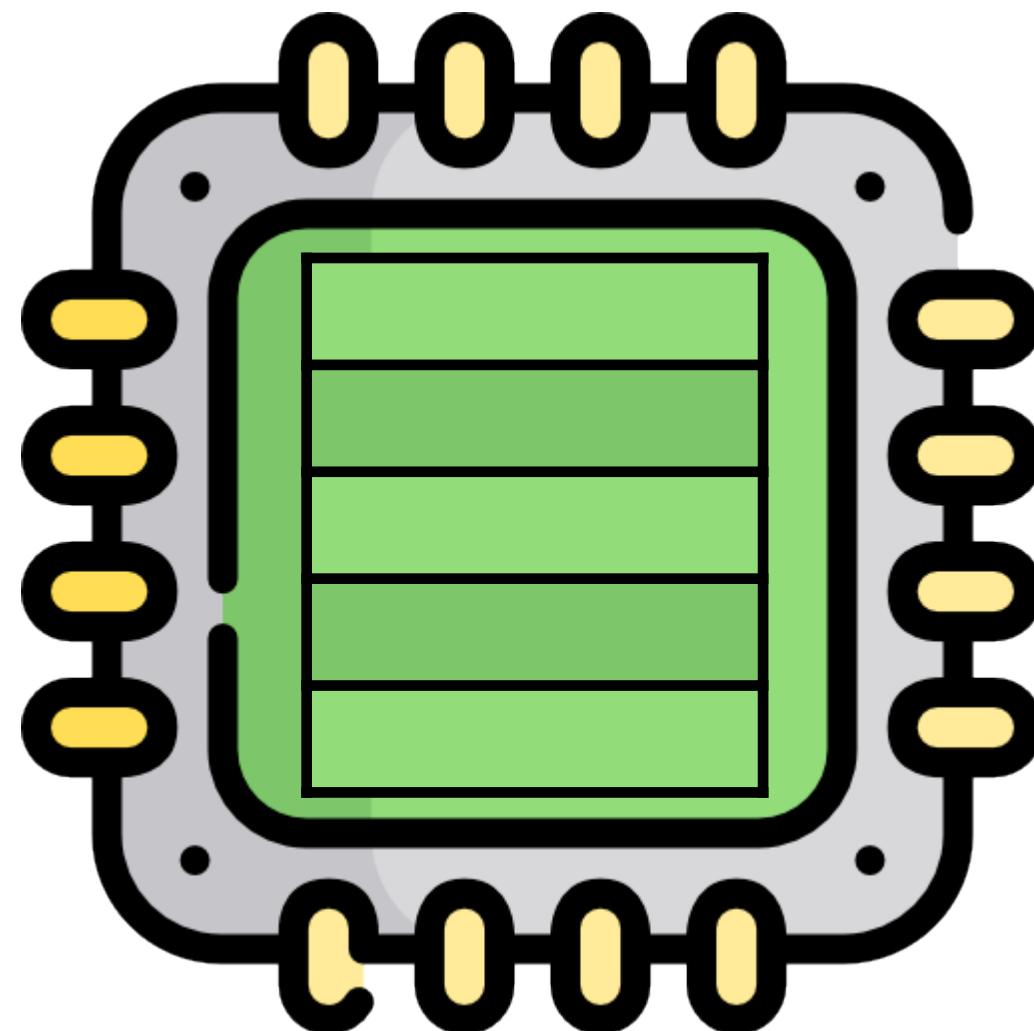


What is in  $A[128]$ ?

1) Training



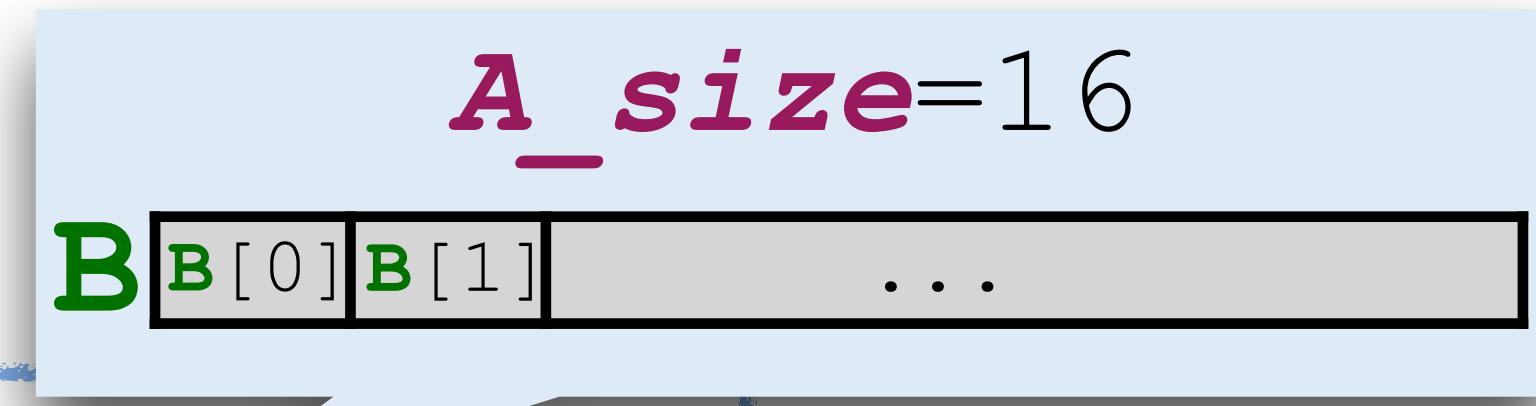
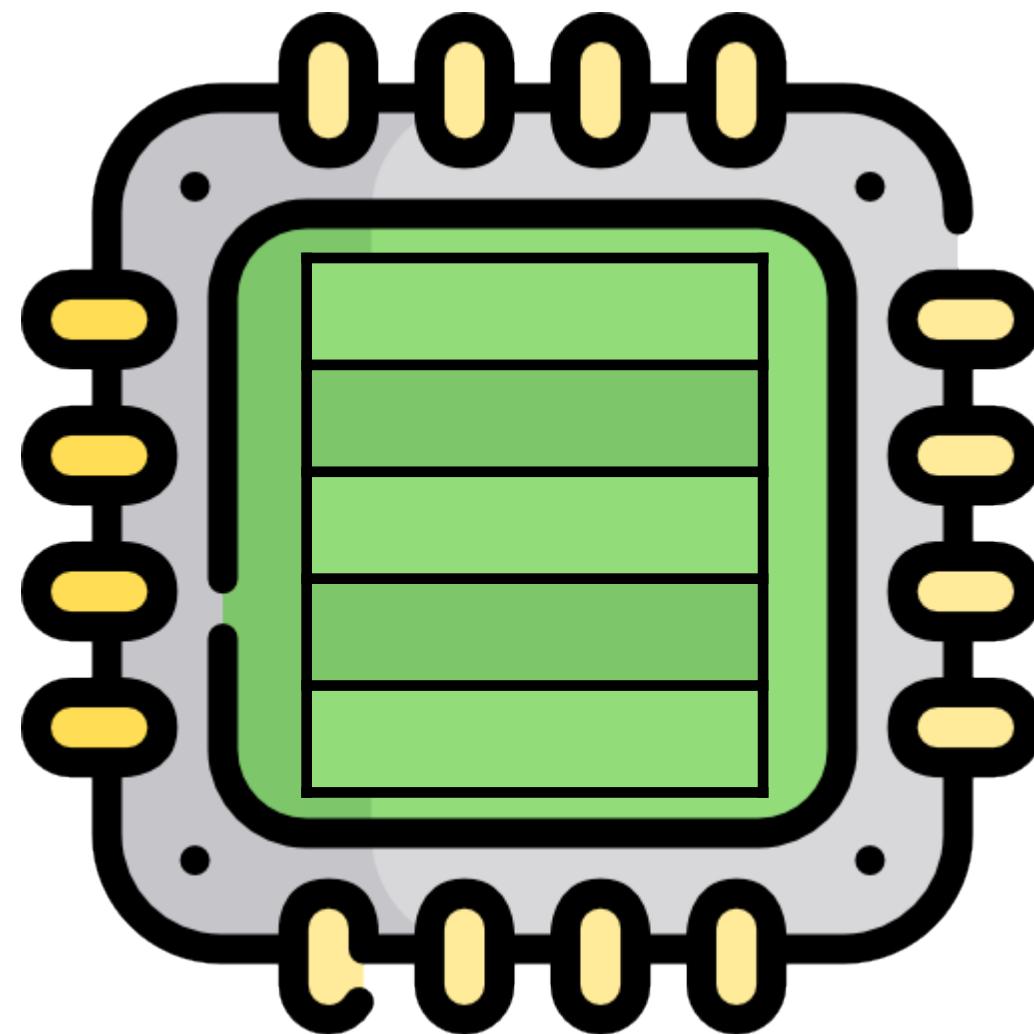
f(0); f(1); f(2); ...



# Spectre V1

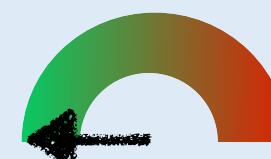


```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



What is in A[128]?

1) Training



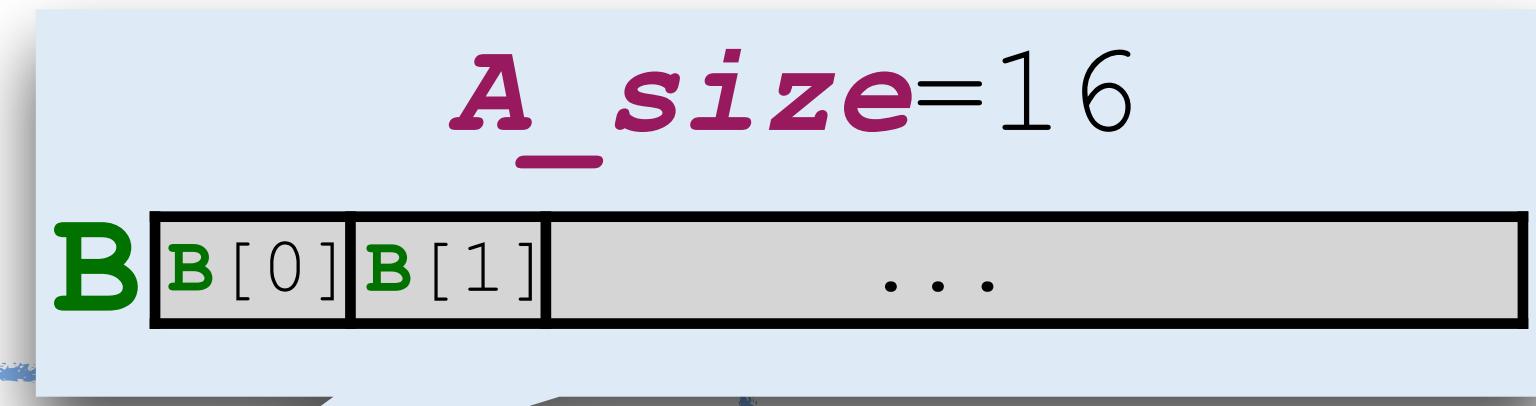
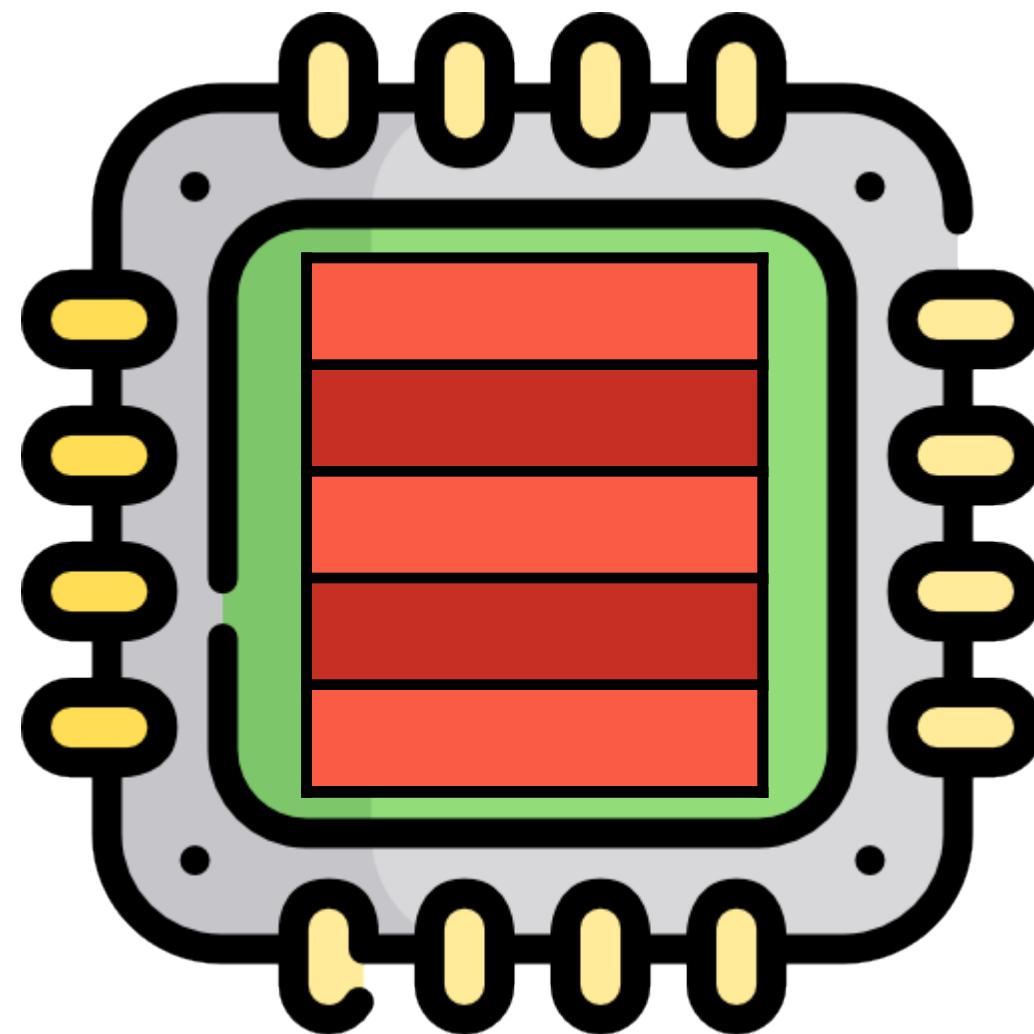
f(0); f(1); f(2); ...

2) Prepare cache

# Spectre V1

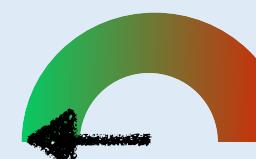


```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



What is in A[128]?

1) Training



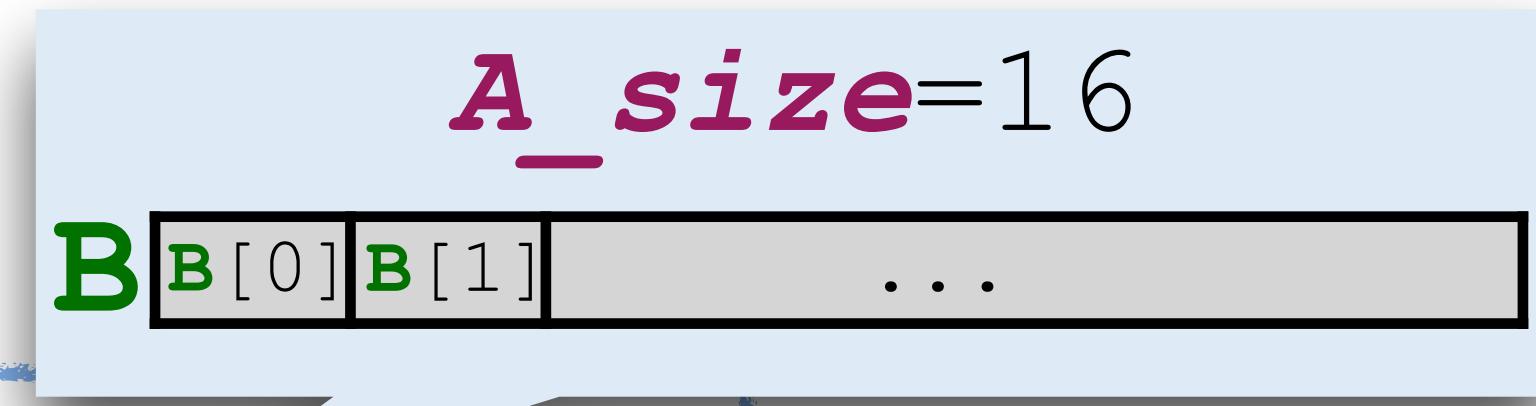
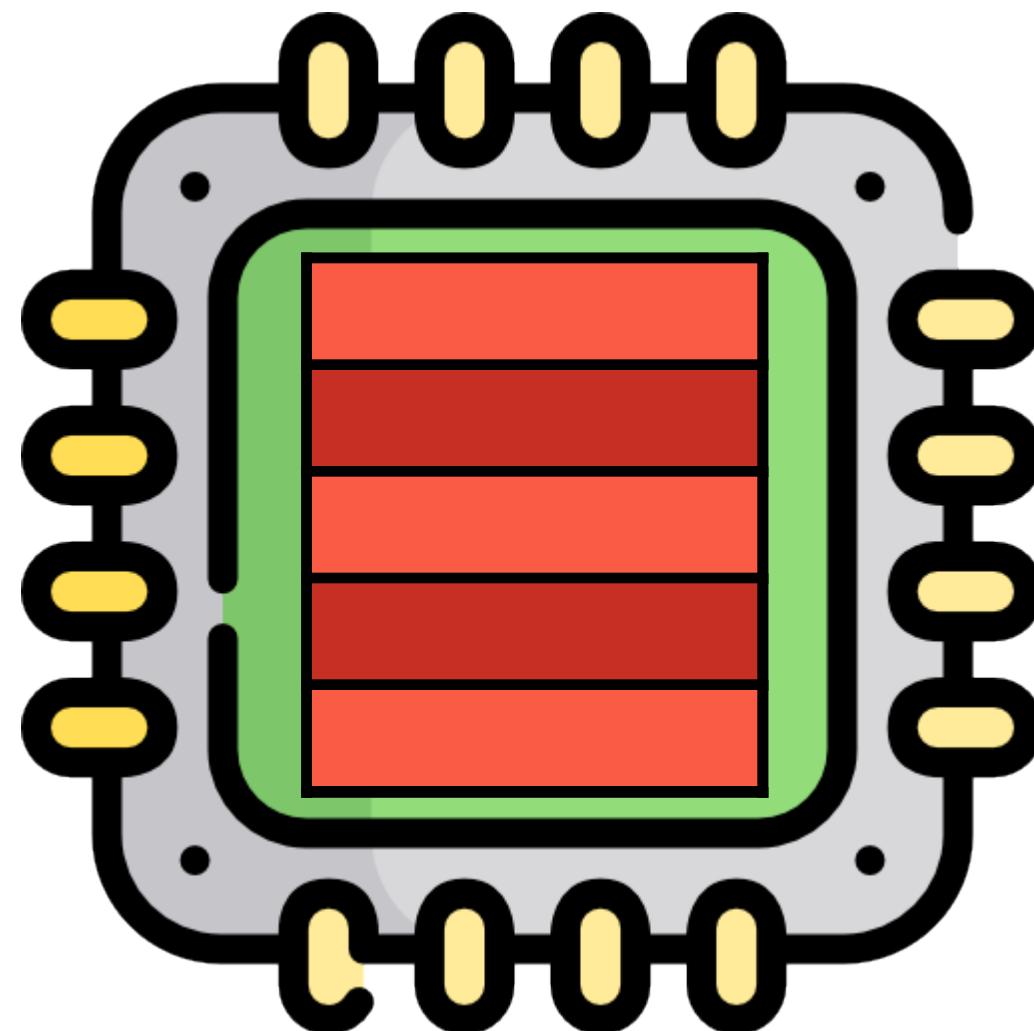
f(0); f(1); f(2); ...

2) Prepare cache

# Spectre V1

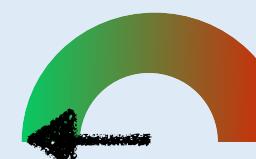


```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



What is in A[128]?

1) Training



f(0); f(1); f(2); ...

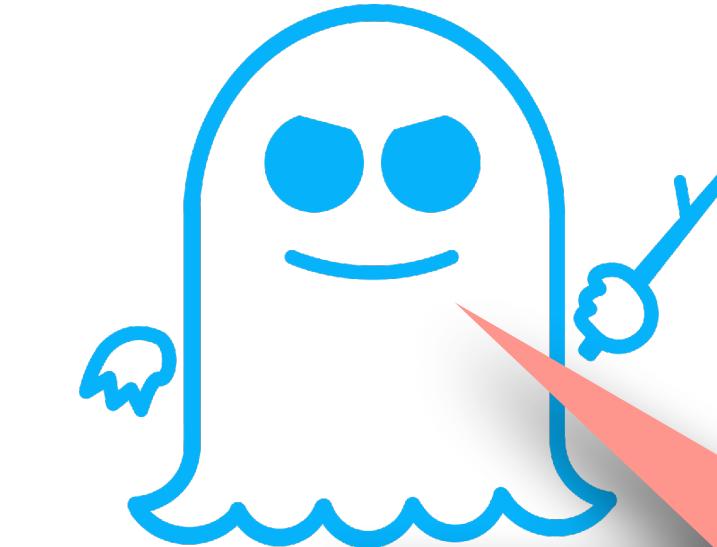
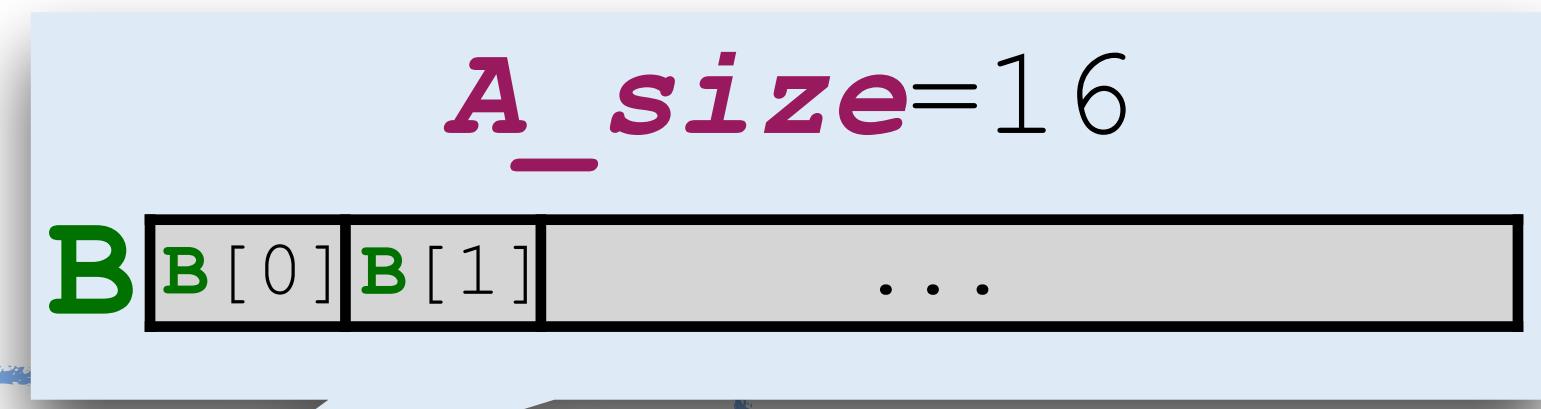
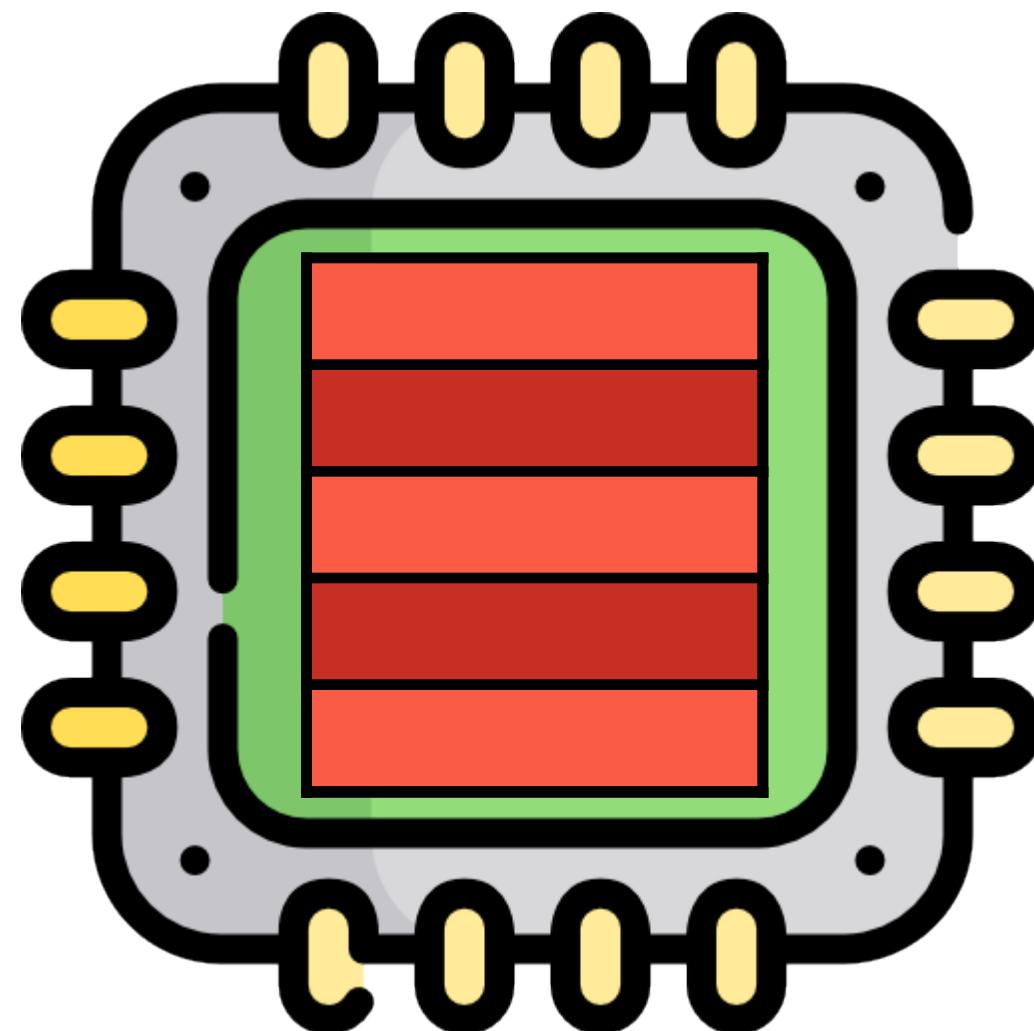
2) Prepare cache

3) Run with  $x = 128$

# Spectre V1

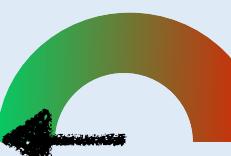


```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



What is in A[128]?

1) Training



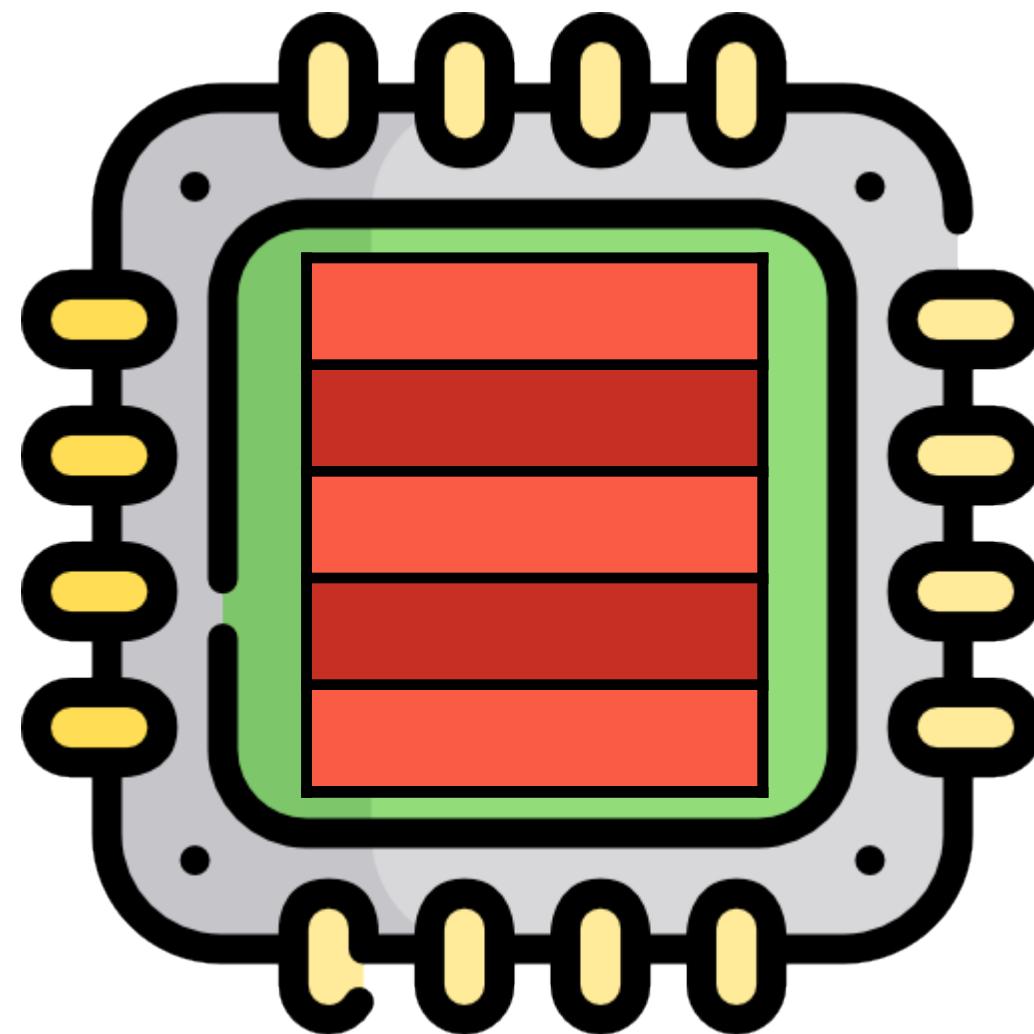
f(0); f(1); f(2); ...

2) Prepare cache

3) Run with  $x = 128$

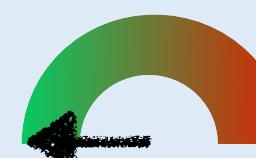
# Spectre V1

```
void f(int x)
if (x < A_size)
y = B[A[x]]
```



What is in A[128]?

1) Training



f(0); f(1); f(2); ...

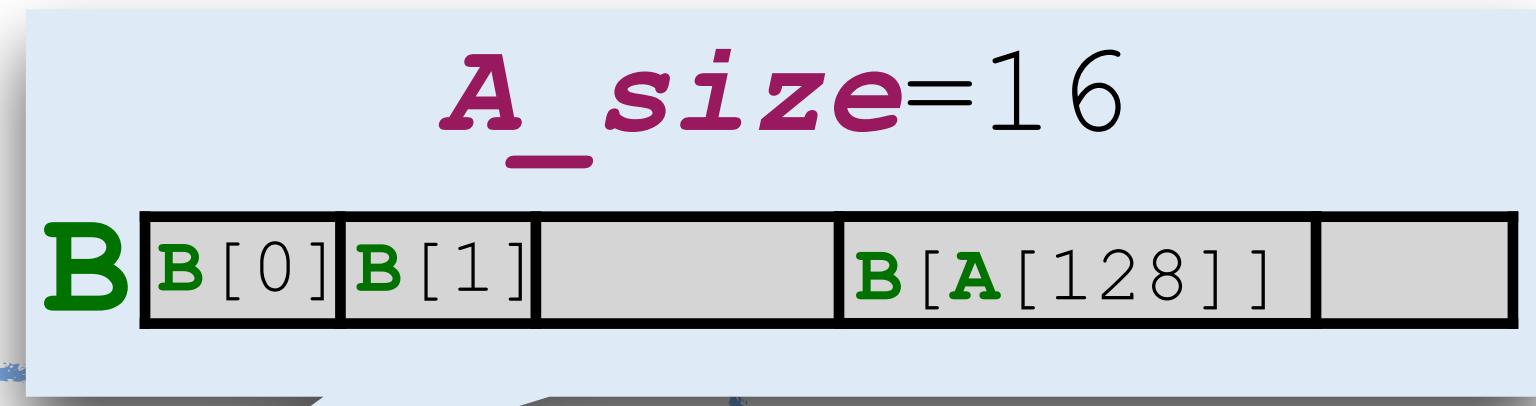
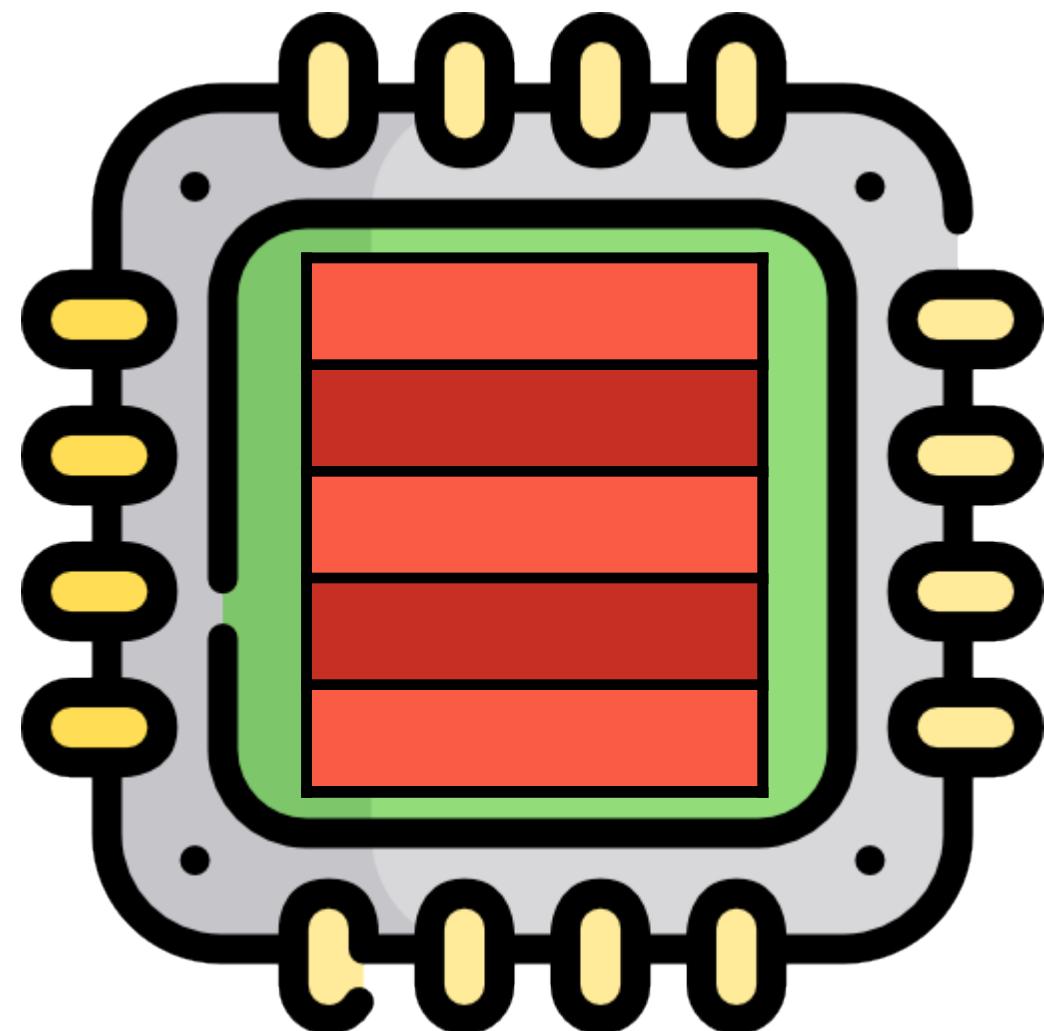
2) Prepare cache

3) Run with  $x = 128$

# Spectre V1

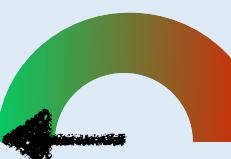


```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



What is in  $A[128]$ ?

1) Training



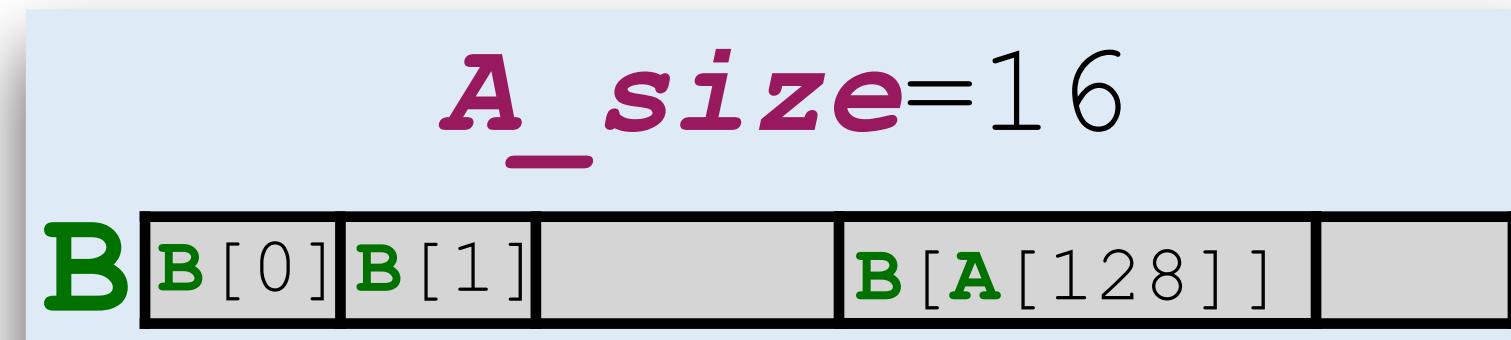
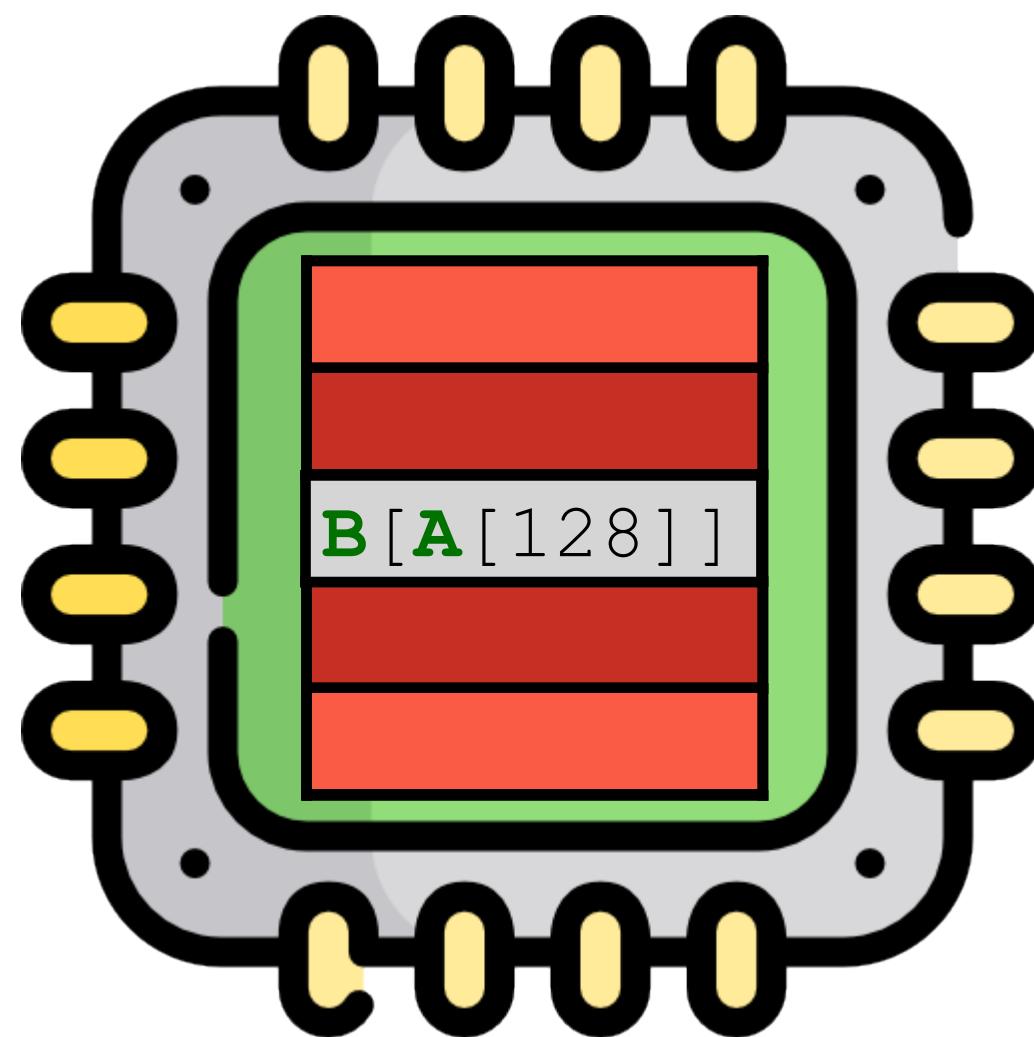
f(0); f(1); f(2); ...

2) Prepare cache

3) Run with  $x = 128$

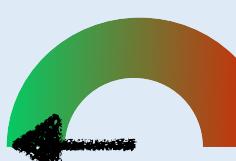
# Spectre V1

```
void f(int x)
if (x < A_size)
y = B[A[x]]
```



What is in  $A[128]$ ?

1) Training



f(0); f(1); f(2); ...

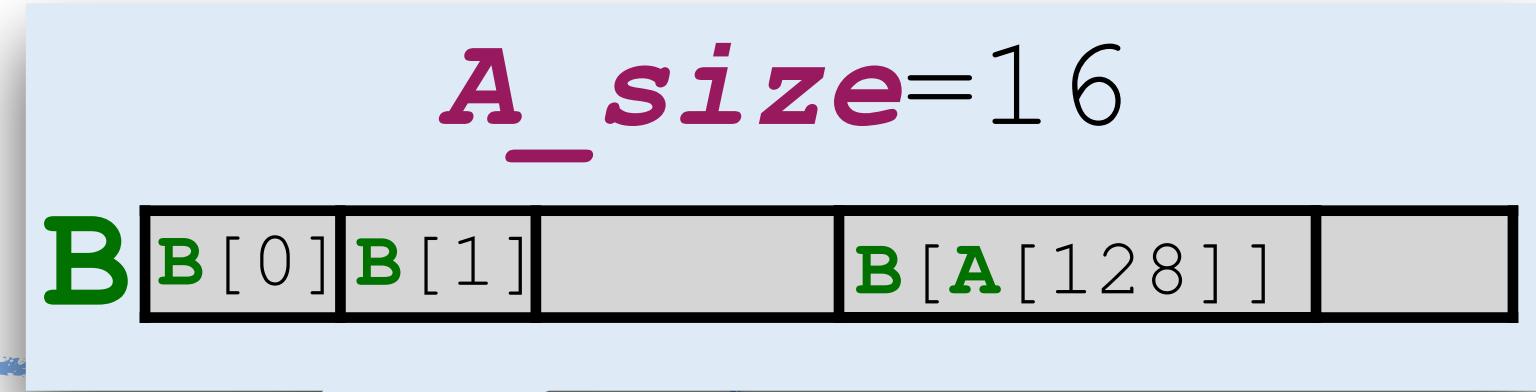
2) Prepare cache

3) Run with  $x = 128$

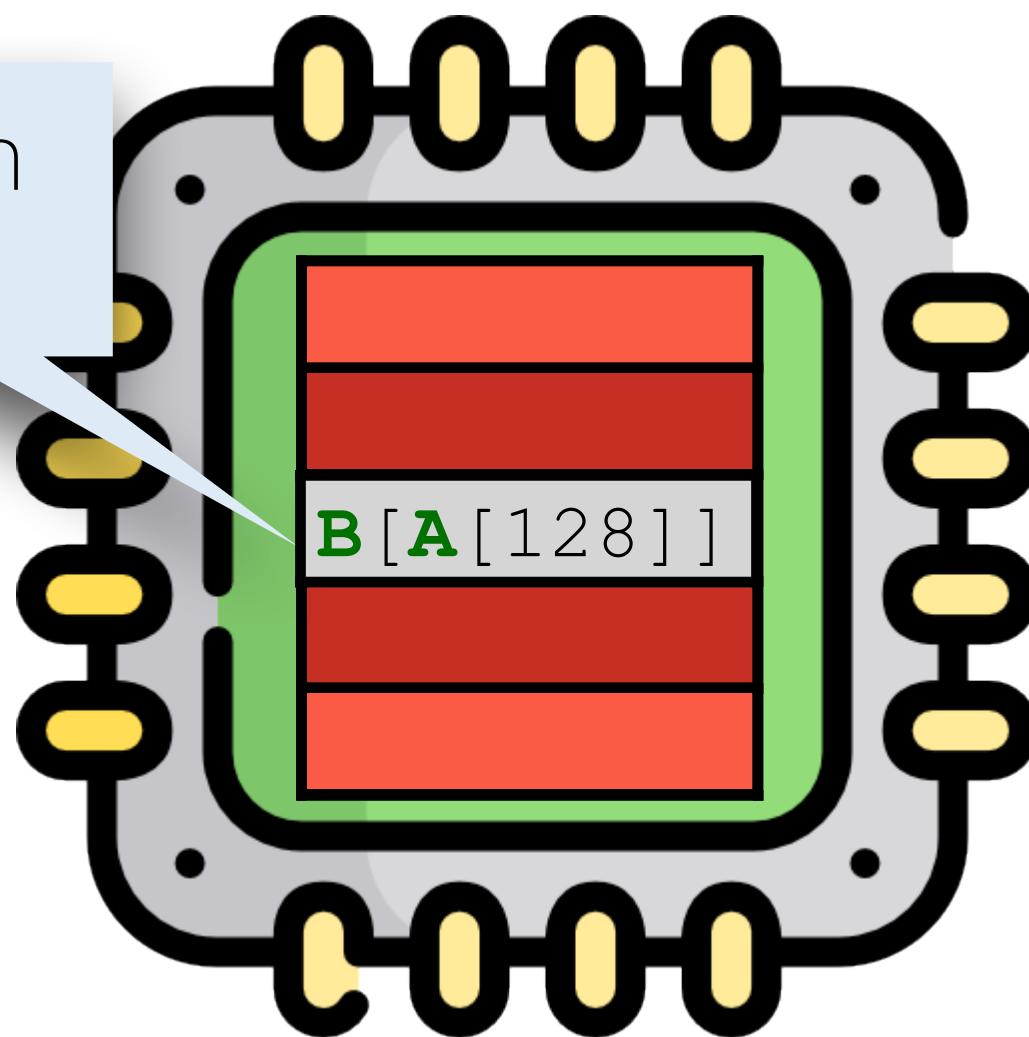
# Spectre V1



```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```

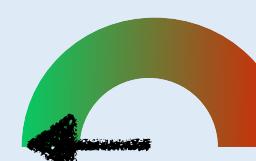


What is in  $A[128]$ ?



Depends on  
 $A[128]$

1) Training



$f(0); f(1); f(2); \dots$

2) Prepare cache

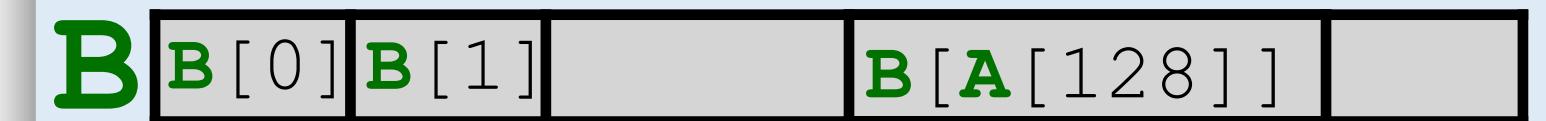
3) Run with  $x = 128$

# Spectre V1

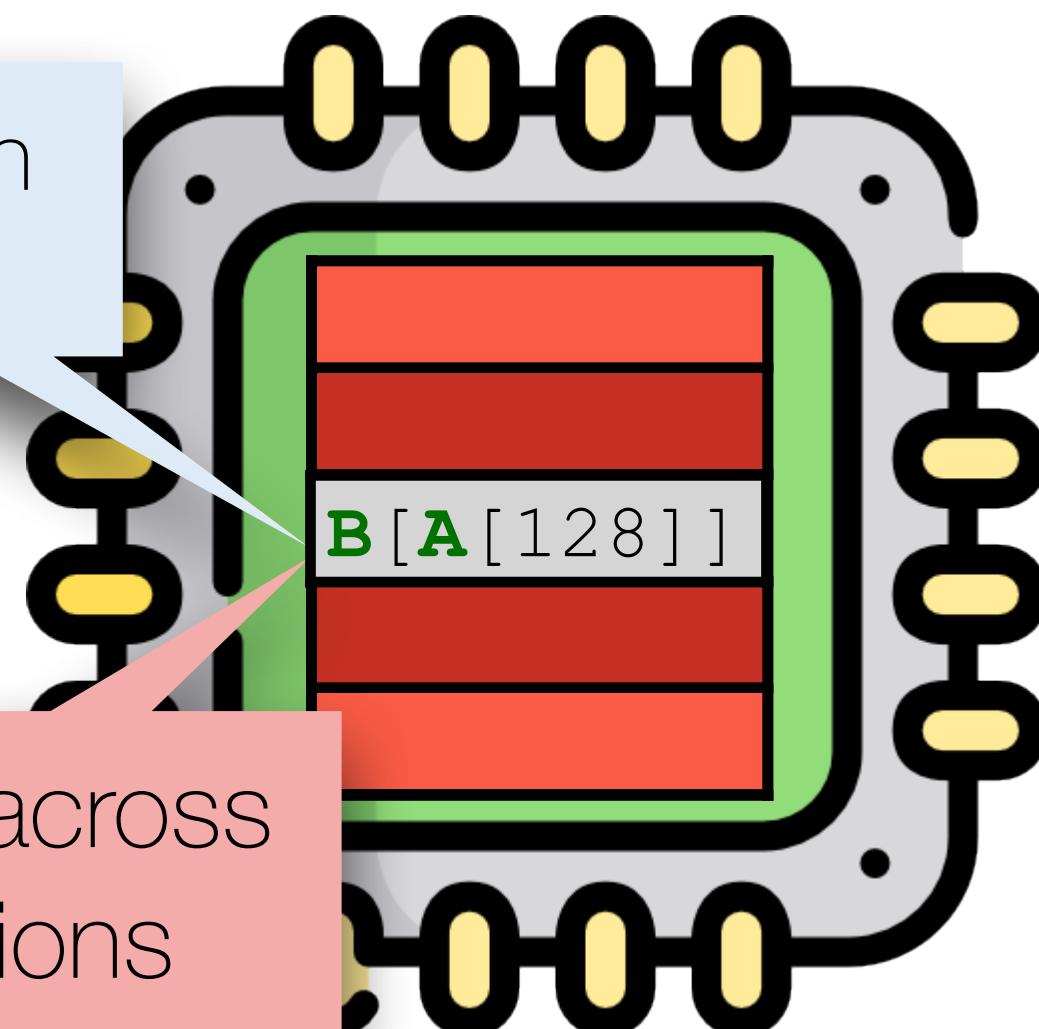


```
void f(int x)
if (x < A_size)
y = B[A[x]]
```

$A\_size=16$

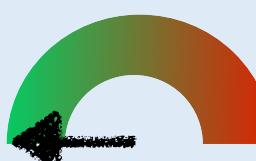


What is in  $A[128]$ ?



Persistent across  
speculations

1) Training



$f(0); f(1); f(2); \dots$

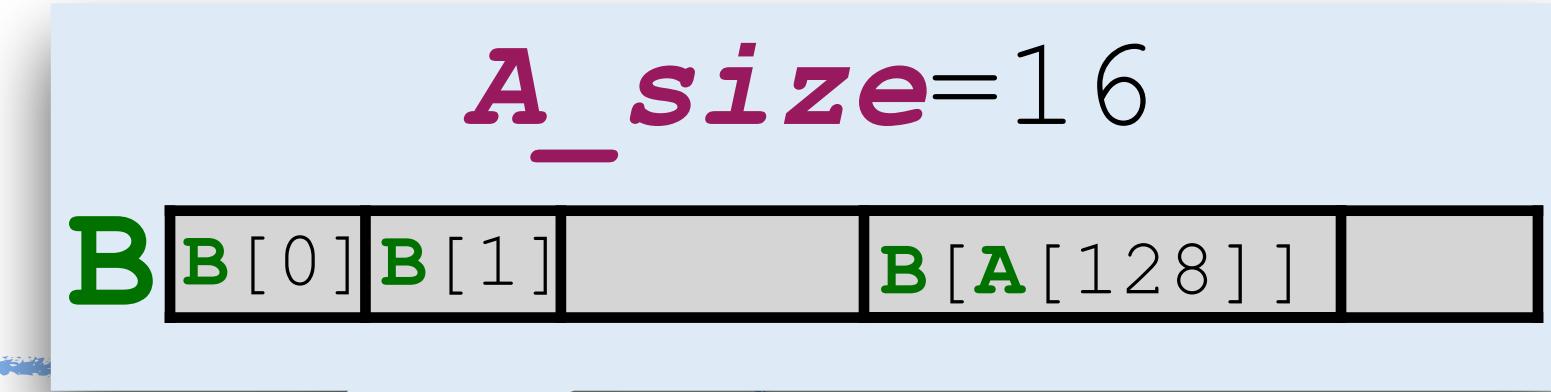
2) Prepare cache

3) Run with  $x = 128$

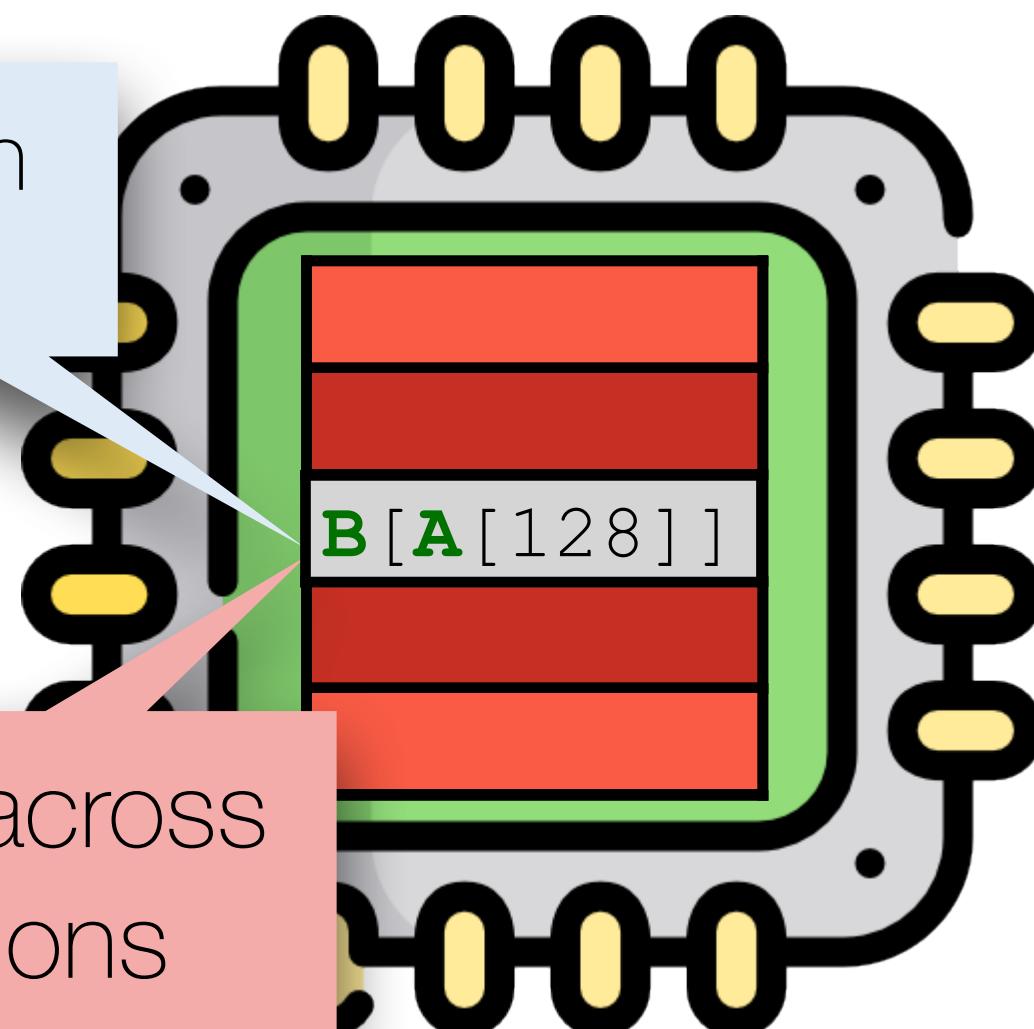
# Spectre V1



```
void f(int x)
if (x < A_size)
y = B[A[x]]
```



Depends on  
 $A[128]$

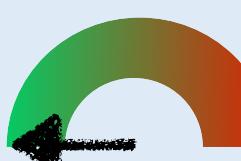


Persistent across  
speculations



What is in  $A[128]$ ?

1) Training



f(0); f(1); f(2); ...

2) Prepare cache

3) Run with  $x = 128$

4) Extract from cache

# Countermeasures

# Countermeasures

***Long Term:*** Co-design of software and hardware countermeasures

# Countermeasures

***Long Term:*** Co-design of software and hardware countermeasures

***Short and Mid Term:*** Software countermeasures

## Compiler-level countermeasures

- Example: insert LFENCE to selectively stop speculative execution
- Implemented in major compilers (Microsoft Visual C++, Intel ICC, Clang)

# Countermeasures

***Long Term:*** Co-design of software and hardware countermeasures

***Short and Mid Term:*** Software countermeasures

## Compiler-level countermeasures

- Example: insert LFENCE to selectively stop speculative execution
- Implemented in major compilers (Microsoft Visual C++, Intel ICC, Clang)

PROBLEM  
SOLVED?

# Compiler-level countermeasures

# Compiler-level countermeasures

## Spectre Mitigations in Microsoft's C/C++ Compiler

Paul Kocher

February 13, 2018

<https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html>

# Compiler-level countermeasures

## Spectre Mitigations in Microsoft's C/C++ Compiler

Paul Kocher

February 13, 2018

<https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html>

---

“compiler [...] produces **unsafe code** when the static analyzer is unable to determine whether a code pattern will be exploitable”

# Compiler-level countermeasures

## Spectre Mitigations in Microsoft's C/C++ Compiler

Paul Kocher

February 13, 2018

<https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html>

“compiler [...] produces **unsafe code** when the static analyzer is unable to determine whether a code pattern will be exploitable”

“there is **no guarantee** that all possible instances of [Spectre] will be instrumented”

# Compiler-level countermeasures

## Spectre Mitigations in Microsoft's C/C++ Compiler

Paul Kocher

February 13, 2018

<https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html>

“compiler [...] produces **unsafe code** when the static analyzer is unable to determine whether a code pattern will be exploitable”

“there is **no guarantee** that all possible instances of [Spectre] will be instrumented”

Bottom line: No guarantees!

# Outline

1. Speculative execution attacks 101
2. Speculative non-interference
3. Detecting speculative leaks
4. Spectector + Case studies

# Speculative execution attacks 101

# Speculative execution + branch prediction

# Speculative execution + branch prediction

Size of array **A**

```
if (x < A_size)
    y = B[A[x]]
```

# Speculative execution + branch prediction

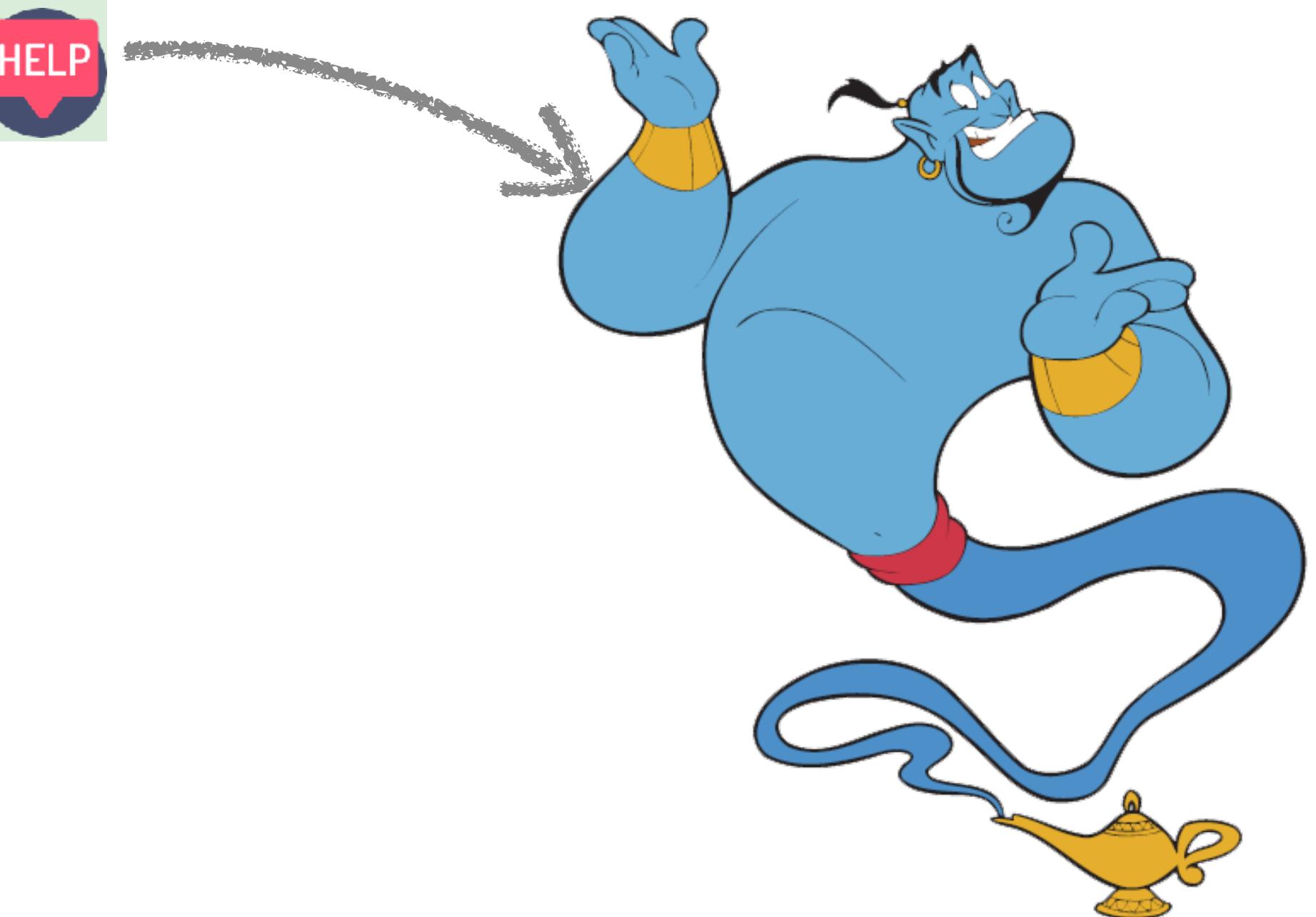
Size of array **A**

```
if (x < A_size)
    y = B[A[x]]
```

# Speculative execution + branch prediction

Size of array **A**

```
if (x < A_size)  
    y = B[A[x]]
```



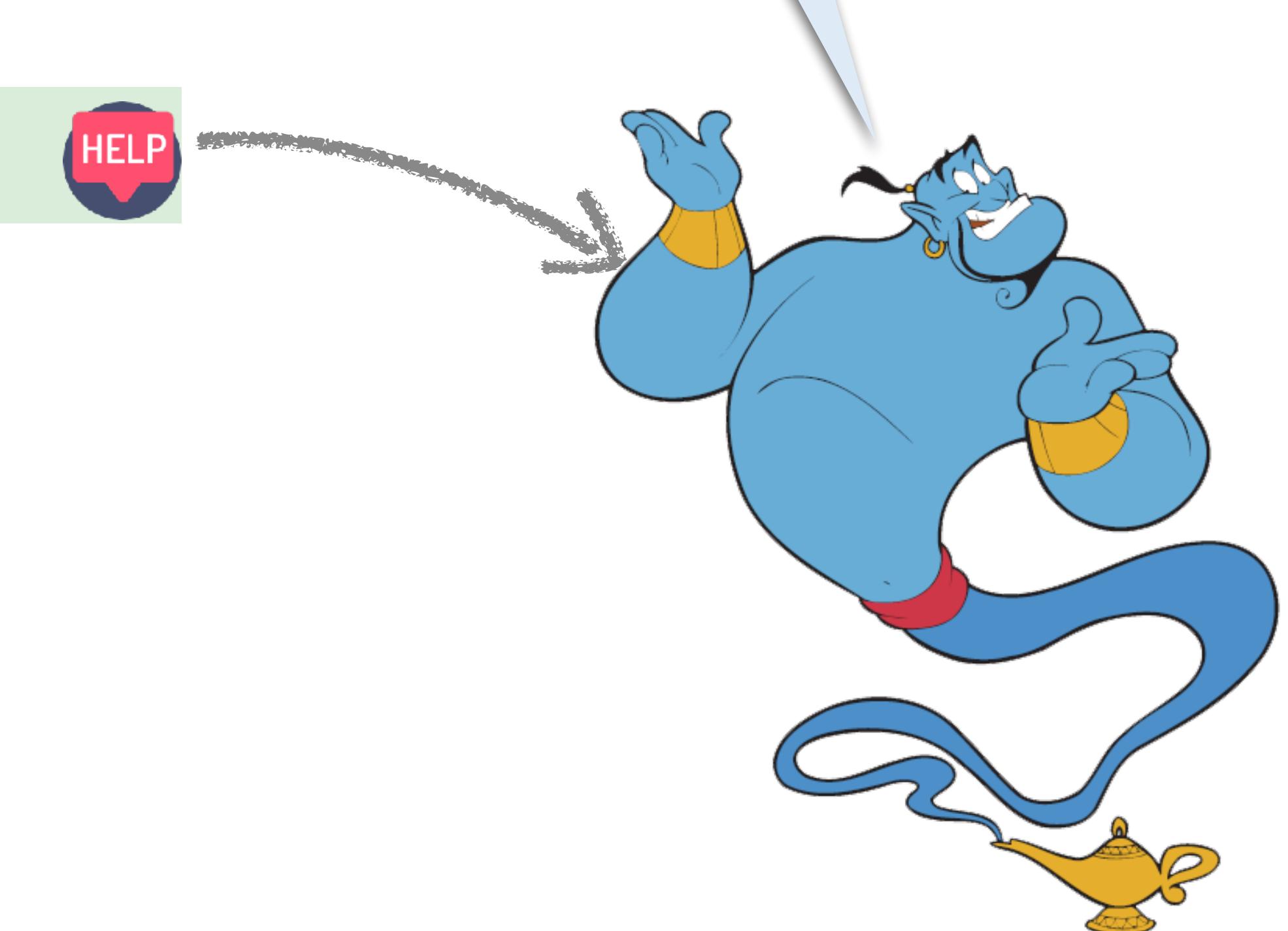
Branch predictor

# Speculative execution + branch prediction

Prediction based on **branch history** & **program structure**

Size of array **A**

```
if (x < A_size) HELP  
    y = B[A[x]]
```



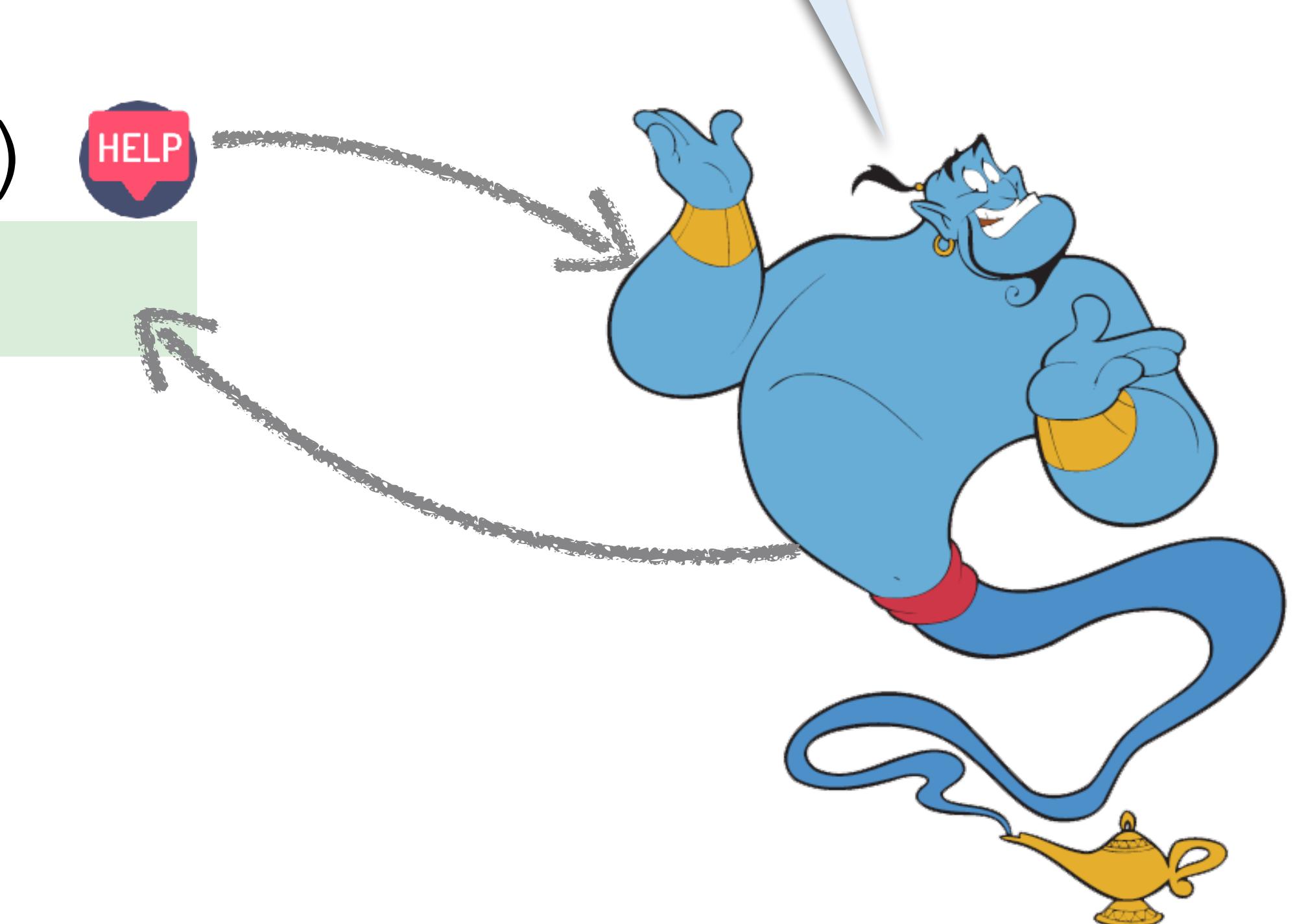
Branch predictor

# Speculative execution + branch prediction

Prediction based on **branch history** & **program structure**

Size of array **A**

```
if (x < A_size)  
    y = B[A[x]]
```



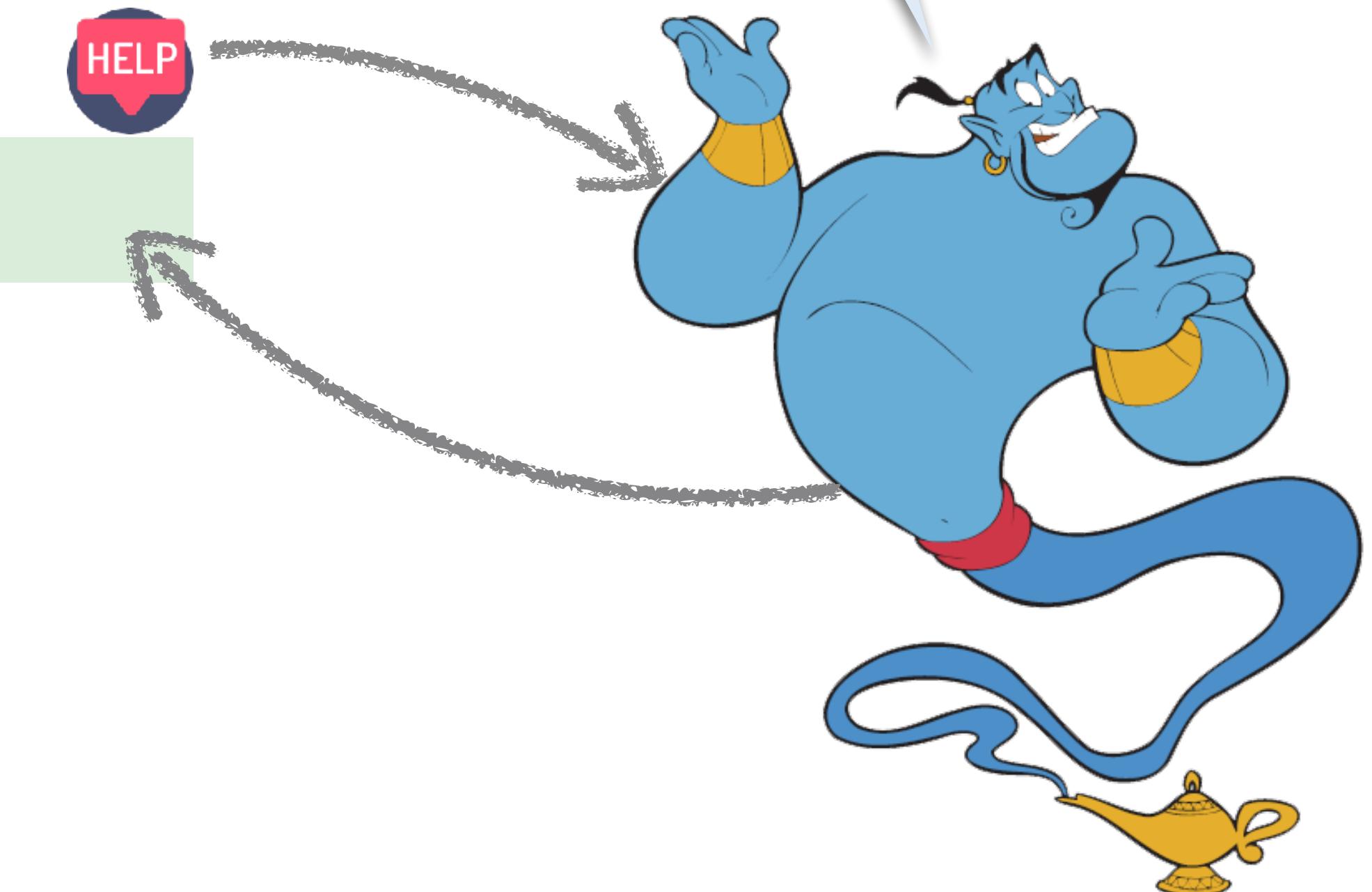
Branch predictor

# Speculative execution + branch prediction

Prediction based on **branch history** & **program structure**

Size of array **A**

```
if (x < A_size)  
    y = B[A[x]]
```



Wrong predicton? **Rollback changes!**



Architectural (ISA) state



Microarchitectural state

Branch predictor

# Speculative non-interference

# Speculative non-interference

# Speculative non-interference

Program **P** is **speculatively non-interferent** if

# Speculative non-interference

Program  $P$  is **speculatively non-interferent** if

*Informally:*

Leakage of  $P$  in  
***non-speculative***  
execution

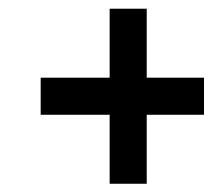
=

Leakage of  $P$  in  
***speculative***  
execution

# How to capture leakage?

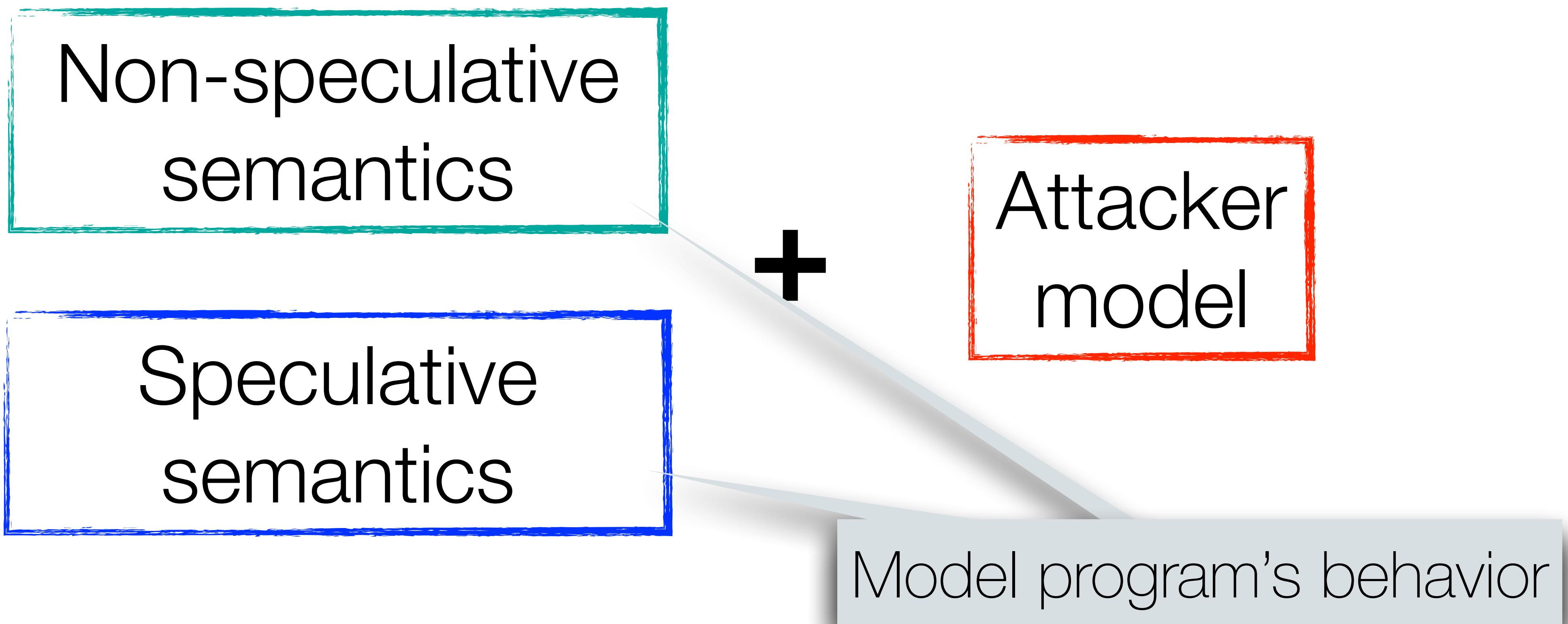
Non-speculative  
semantics

Speculative  
semantics

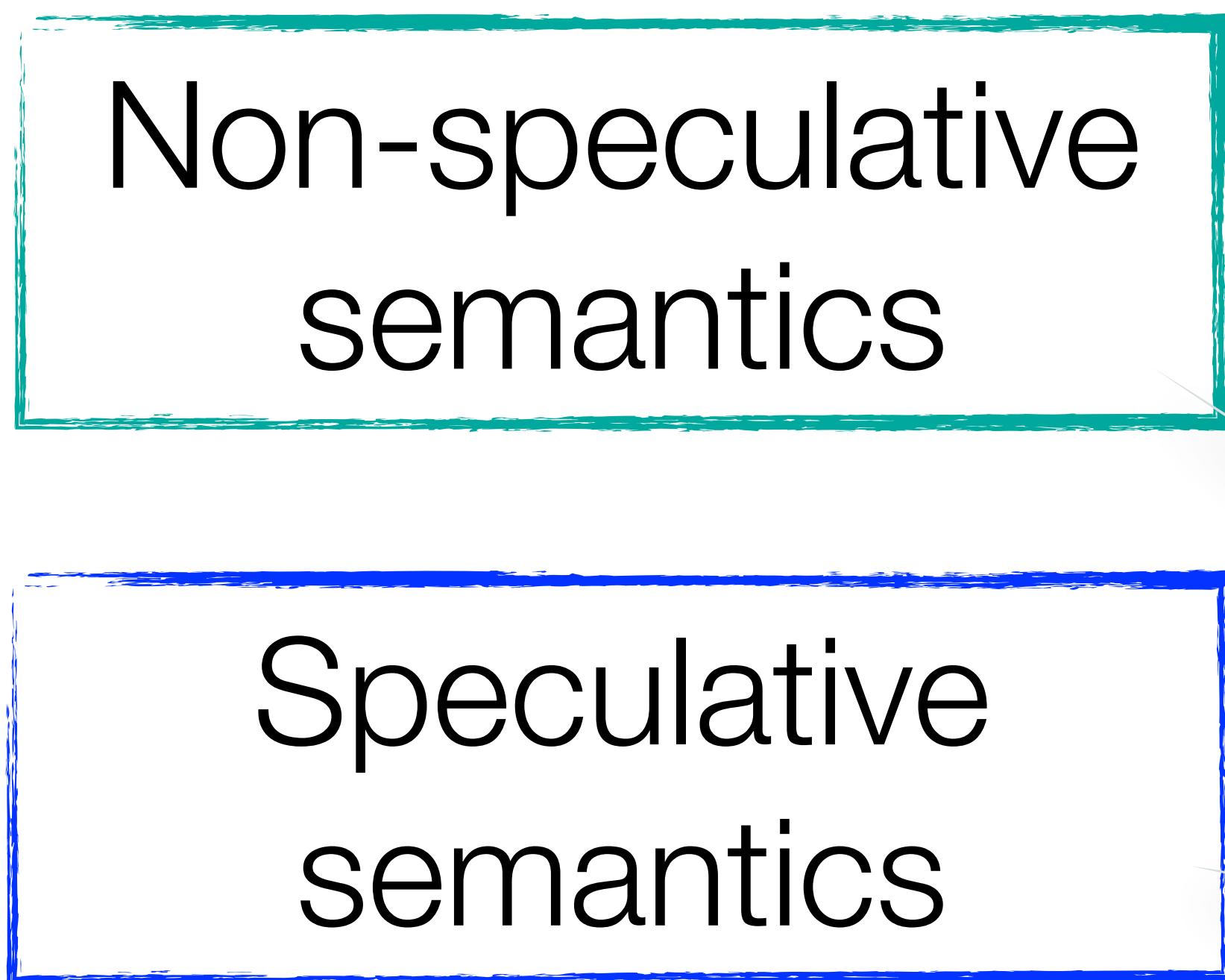


Attacker  
model

# How to capture leakage?



# How to capture leakage?



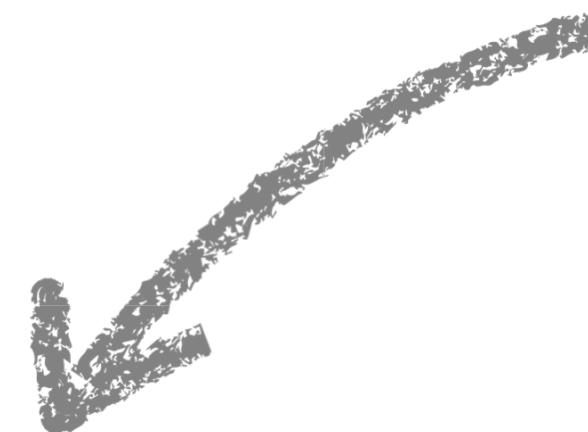
Attacker  
model

Capture attacker's  
observational power

Model program's behavior

# $\mu$ Assembly + non-speculative semantics

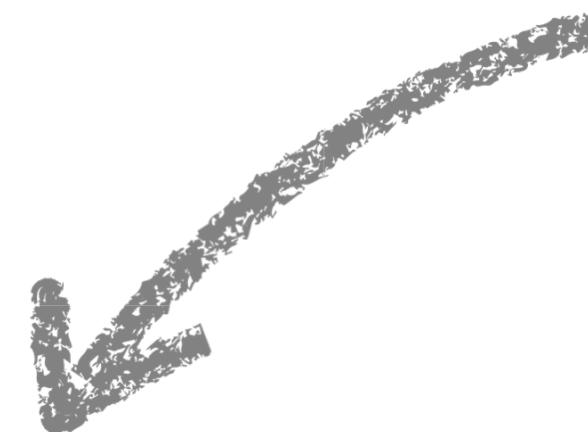
if ( $x < A\_size$ )  
y = B[A[x]]



```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```

# $\mu$ Assembly + non-speculative semantics

if ( $x < A\_size$ )  
y = B[A[x]]



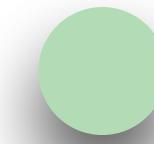
rax <-  $A\_size$

rcx <-  $x$

jmp rcx≥rax, END

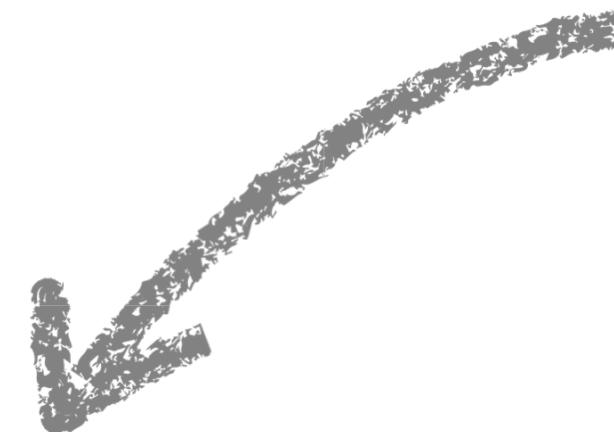
L1: load rax, A + rcx  
load rax, B + rax

END:

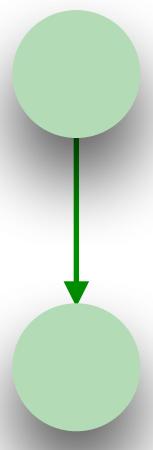


# $\mu$ Assembly + non-speculative semantics

if ( $x < A\_size$ )  
y = B[A[x]]

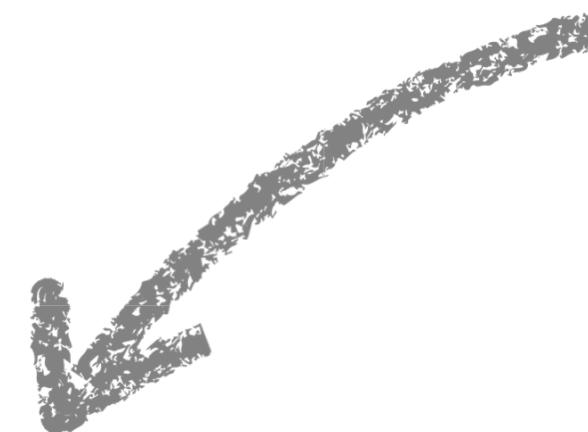


```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



# $\mu$ Assembly + non-speculative semantics

if ( $x < A\_size$ )  
y = B[A[x]]



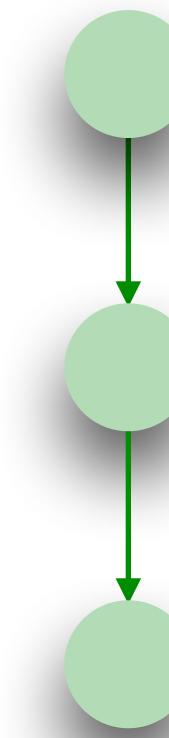
rax <-  $A\_size$

rcx <-  $x$

jmp rcx≥rax, END

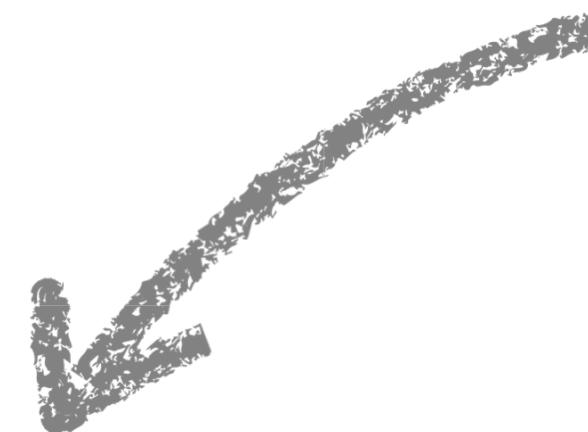
L1: load rax, A + rcx  
load rax, B + rax

END:



# $\mu$ Assembly + non-speculative semantics

if ( $x < A\_size$ )  
y = B[A[x]]



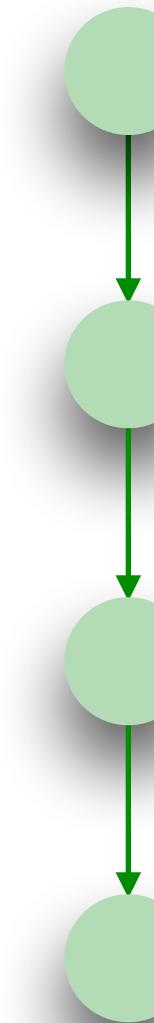
rax <-  $A\_size$

rcx <-  $x$

jmp rcx≥rax, END

L1: load rax, A + rcx  
load rax, B + rax

END:



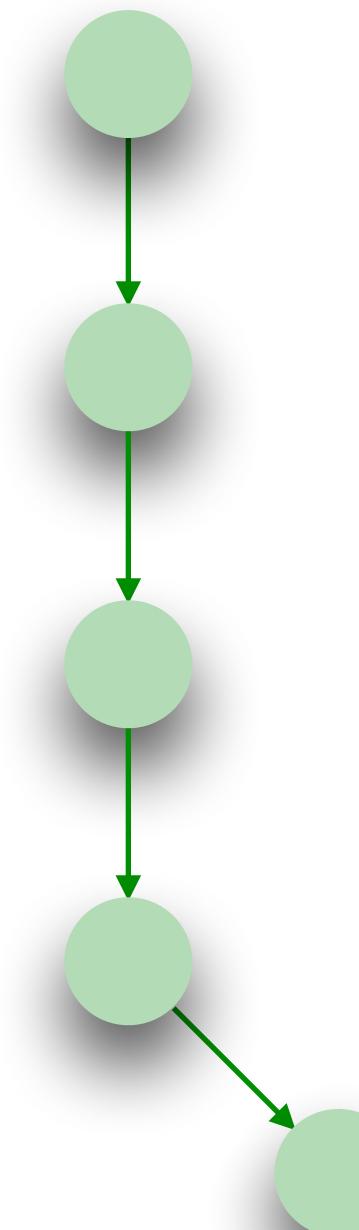
# $\mu$ Assembly + non-speculative semantics

if ( $x < A\_size$ )  
y = B[A[x]]



```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
```

END:



# Speculative semantics

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```

# Speculative semantics

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```

**Prediction Oracle**  $O$ : branch prediction + length of speculative window

# Speculative semantics

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```

Starts ***speculative transactions***  
upon branch instructions

**Prediction Oracle**  $O$ : branch prediction + length of speculative window

# Speculative semantics

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rcx
END:
```

Starts ***speculative transactions***  
upon branch instructions

Committed upon  
correct speculation

**Prediction Oracle  $O$** : branch prediction + length of speculative window

# Speculative semantics

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rcx
END:
```

Starts ***speculative transactions***  
upon branch instructions

Committed upon  
correct speculation

Rolled back upon misspeculation

**Prediction Oracle  $O$** : branch prediction + length of speculative window

# Speculative semantics

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rcx
END:
```

Starts ***speculative transactions***  
upon branch instructions

Committed upon  
correct speculation

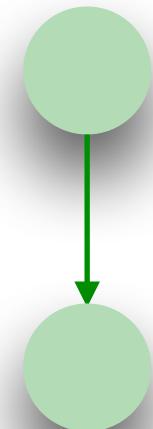
Rolled back upon misspeculation

**Prediction Oracle  $O$** : branch prediction + length of speculative window

# Speculative semantics

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rcx
```

*END:*



Starts ***speculative transactions***  
upon branch instructions

Committed upon  
correct speculation

Rolled back upon misspeculation

**Prediction Oracle  $O$ :** branch prediction + length of speculative window

# Speculative semantics

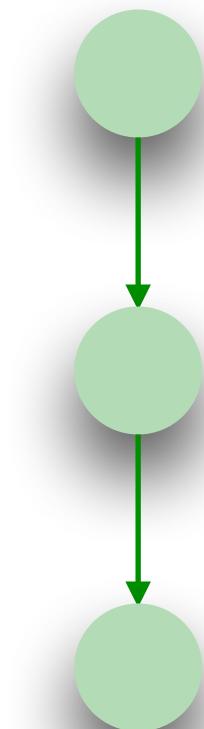
```
rax <- A_size
```

```
rcx <- x
```

```
jmp rcx≥rax, END
```

*L1:* load rax, **A** + rcx  
load rax, **B** + rax

*END:*



Starts ***speculative transactions***  
upon branch instructions

Committed upon  
correct speculation

Rolled back upon misspeculation

**Prediction Oracle  $O$ :** branch prediction + length of speculative window

# Speculative semantics

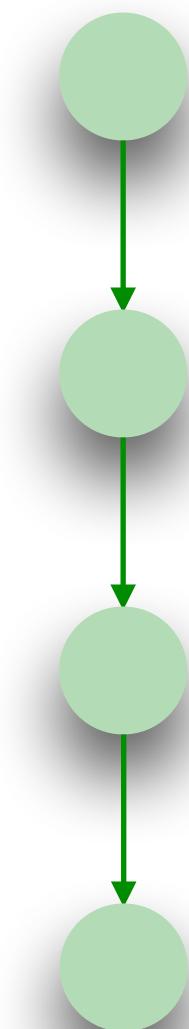
```
rax <- A_size
```

```
rcx <- x
```

```
jmp rcx≥rax, END
```

```
L1: load rax, A + rcx  
load rax, B + rax
```

```
END:
```



Starts **speculative transactions**  
upon branch instructions

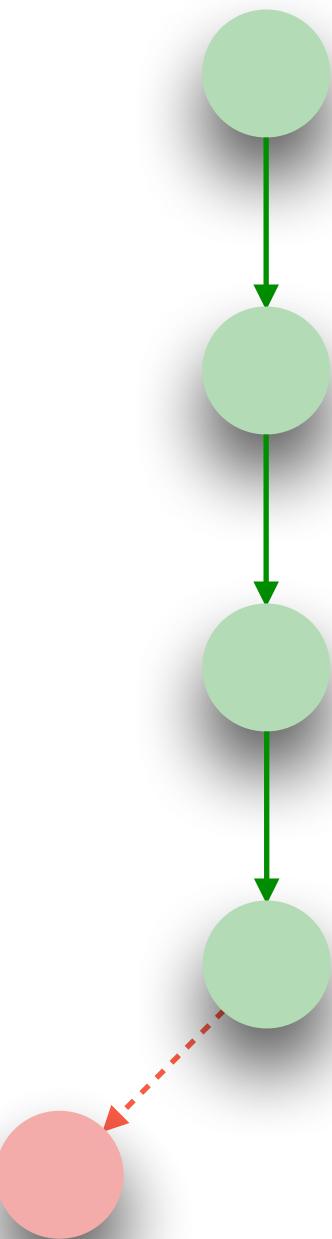
Committed upon  
correct speculation

Rolled back upon misspeculation

**Prediction Oracle**  $O$ : branch prediction + length of speculative window

# Speculative semantics

```
rax <- A_size  
rcx <- x  
jmp rcx≥rax, END  
L1: load rax, A + rcx  
     load rax, B + rax  
END:
```



Starts ***speculative transactions***  
upon branch instructions

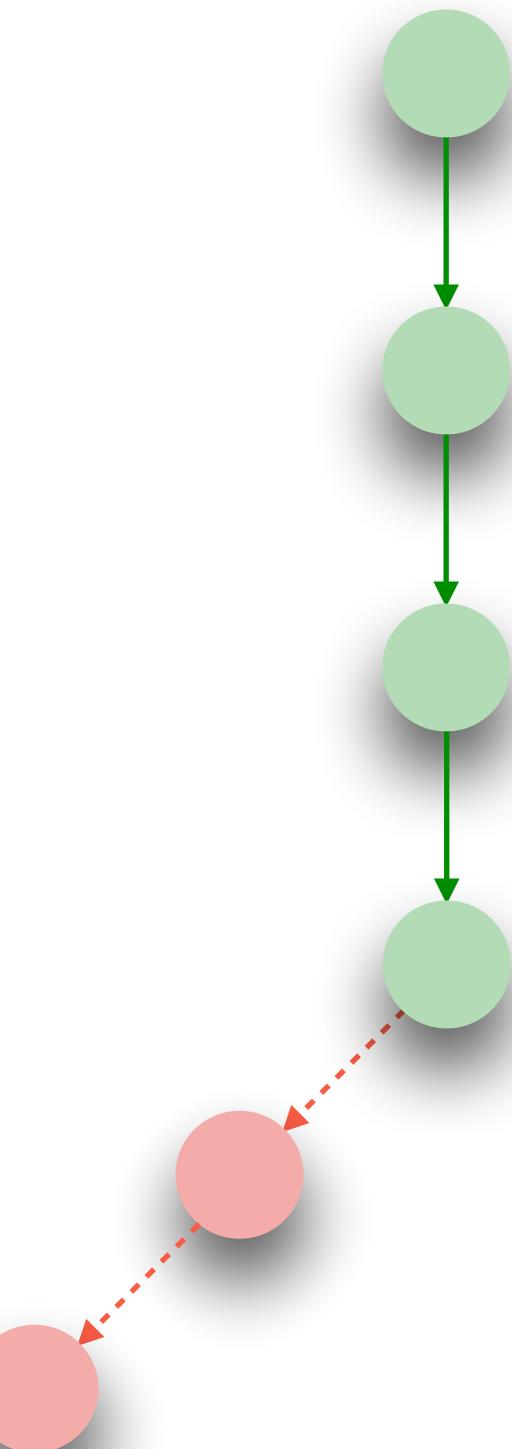
Committed upon  
correct speculation

Rolled back upon misspeculation

**Prediction Oracle  $O$** : branch prediction + length of speculative window

# Speculative semantics

```
rax <- A_size  
rcx <- x  
jmp rcx≥rax, END  
L1: load rax, A + rcx  
     load rax, B + rax  
END:
```



Starts ***speculative transactions***  
upon branch instructions

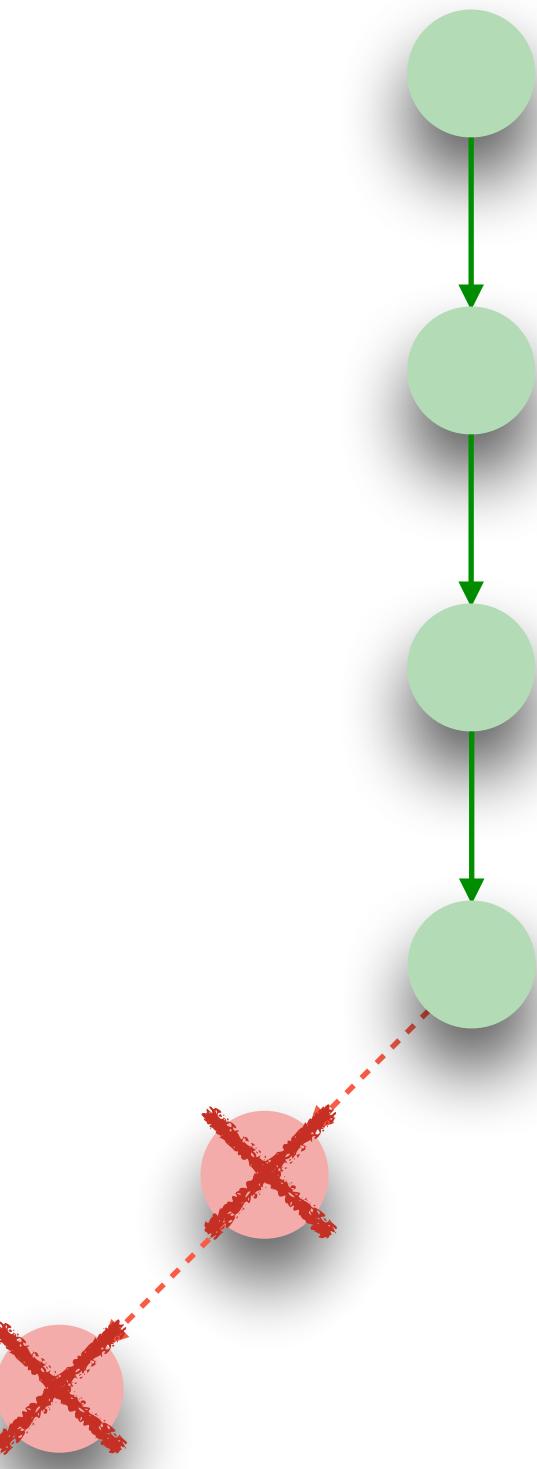
Committed upon  
correct speculation

Rolled back upon misspeculation

**Prediction Oracle  $O$** : branch prediction + length of speculative window

# Speculative semantics

```
rax <- A_size  
rcx <- x  
jmp rcx≥rax, END  
L1: load rax, A + rcx  
     load rax, B + rax  
END:
```



Starts ***speculative transactions***  
upon branch instructions

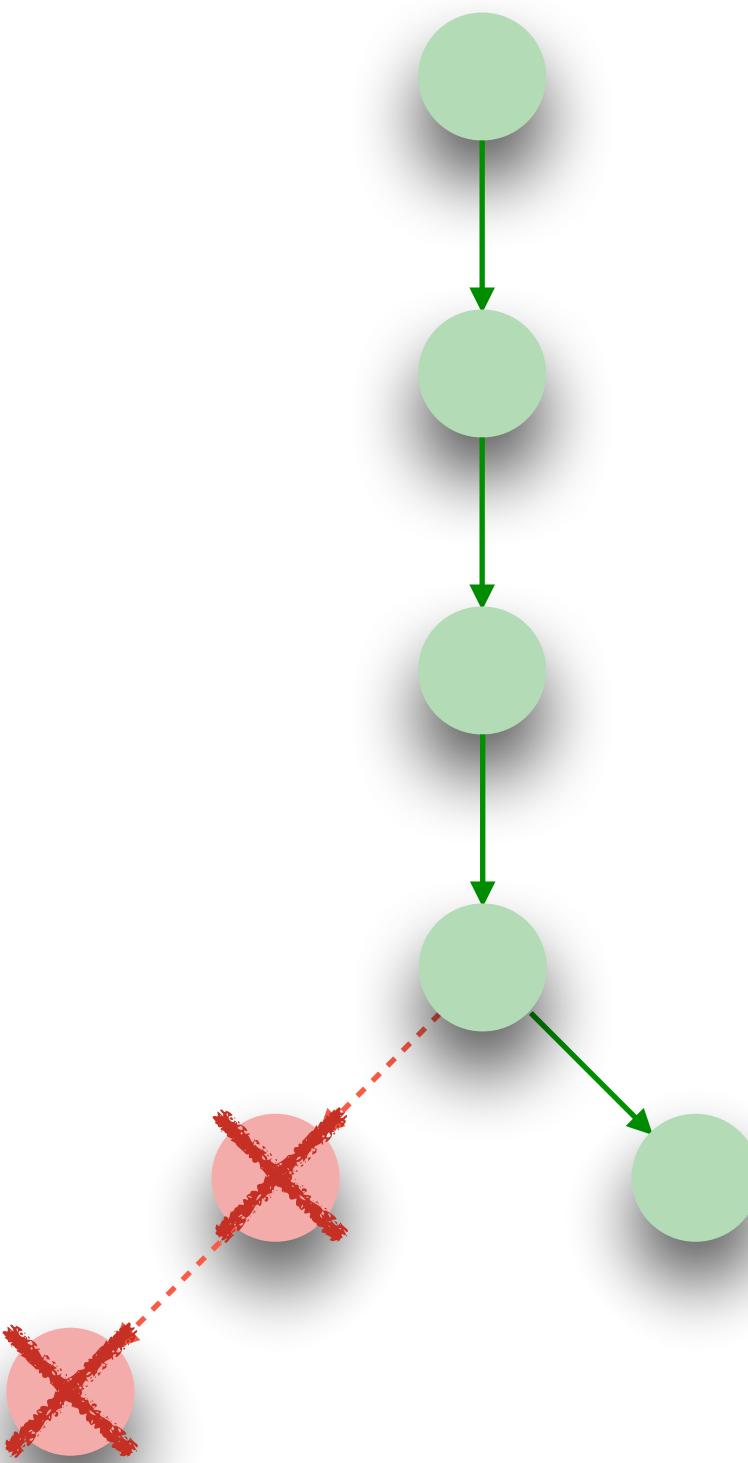
Committed upon  
correct speculation

Rolled back upon misspeculation

**Prediction Oracle  $O$** : branch prediction + length of speculative window

# Speculative semantics

```
rax <- A_size  
rcx <- x  
jmp rcx≥rax, END  
L1: load rax, A + rcx  
     load rax, B + rax  
END:
```



Starts ***speculative transactions***  
upon branch instructions

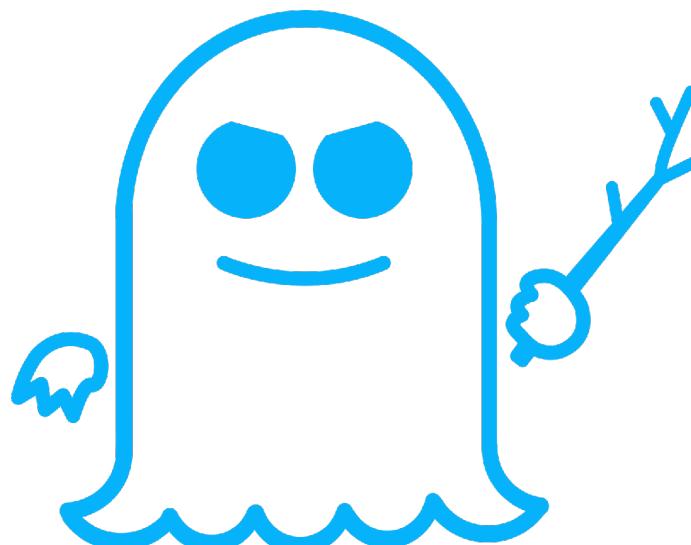
Committed upon  
correct speculation

Rolled back upon misspeculation

**Prediction Oracle  $O$** : branch prediction + length of speculative window

# Leakage into μarchitecture

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```

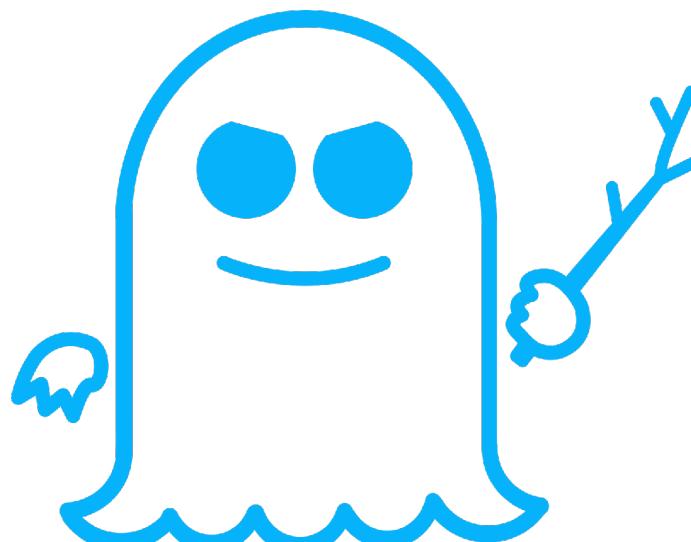


# Leakage into μarchitecture

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```

Attacker can observe:

- locations of **memory accesses**
- **branch/jump** targets
- **start/end** speculative execution



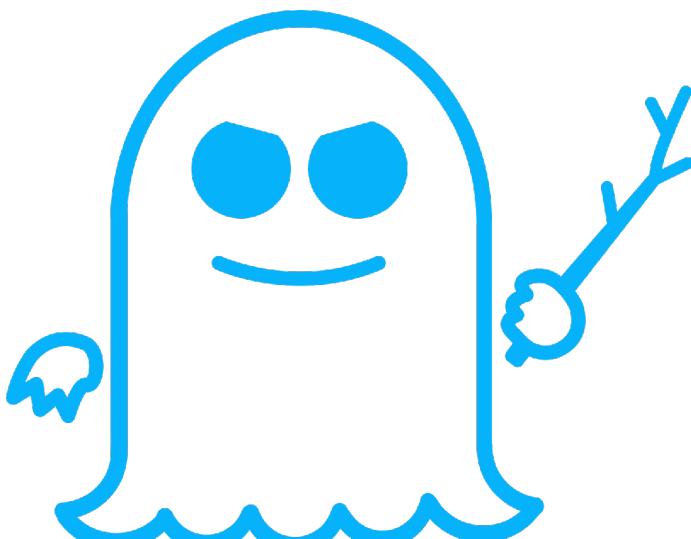
# Leakage into μarchitecture

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```

Attacker can observe:

- locations of **memory accesses**
- **branch/jump** targets
- **start/end** speculative execution

Inspired by “constant-time” rqmts



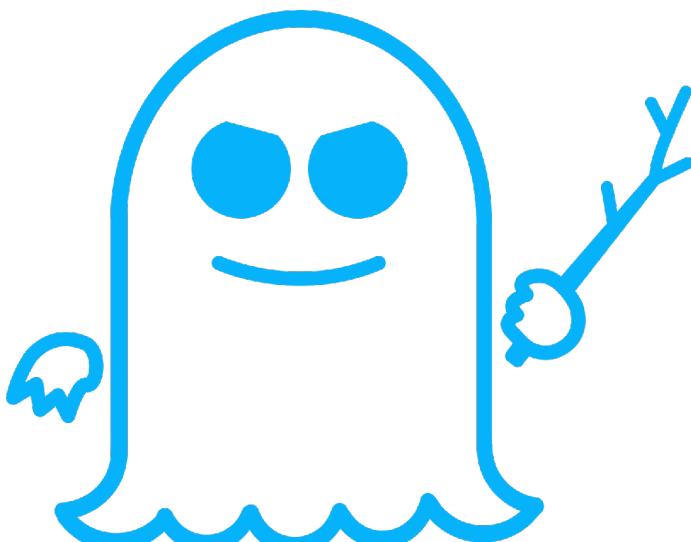
# Leakage into μarchitecture

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```

Attacker can observe:

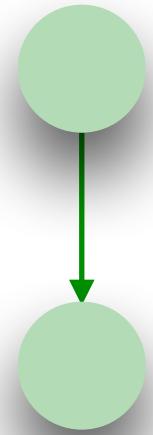
- locations of **memory accesses**
- **branch/jump** targets
- **start/end** speculative execution

Inspired by “constant-time” rqmts



# Leakage into μarchitecture

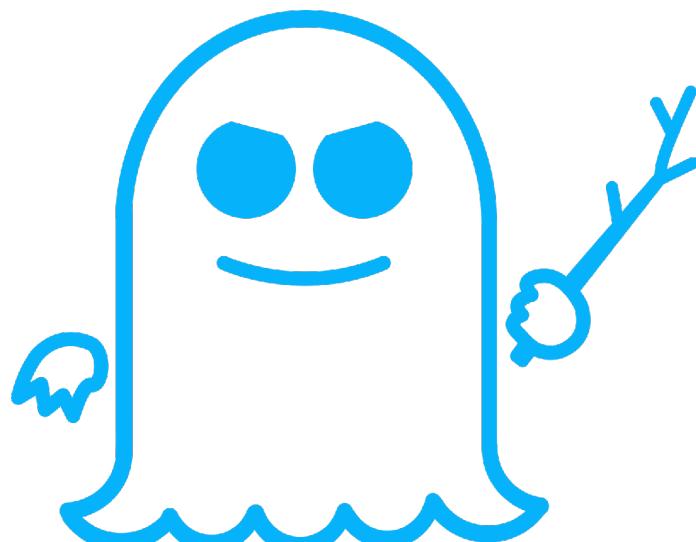
```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
    load rax, B + rax
END:
```



Attacker can observe:

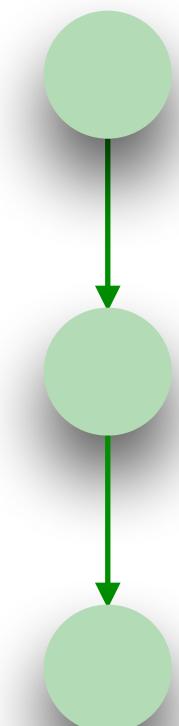
- locations of **memory accesses**
- **branch/jump** targets
- **start/end** speculative execution

Inspired by “constant-time” rqmts



# Leakage into μarchitecture

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
    load rax, B + rax
END:
```



Attacker can observe:

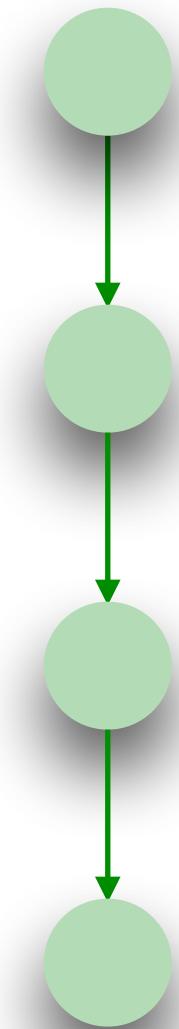
- locations of **memory accesses**
- **branch/jump** targets
- **start/end** speculative execution

Inspired by “constant-time” rqmts



# Leakage into μarchitecture

```
rax <- A_size  
rcx <- x  
jmp rcx≥rax, END  
L1: load rax, A + rcx  
     load rax, B + rax  
END:
```



Attacker can observe:

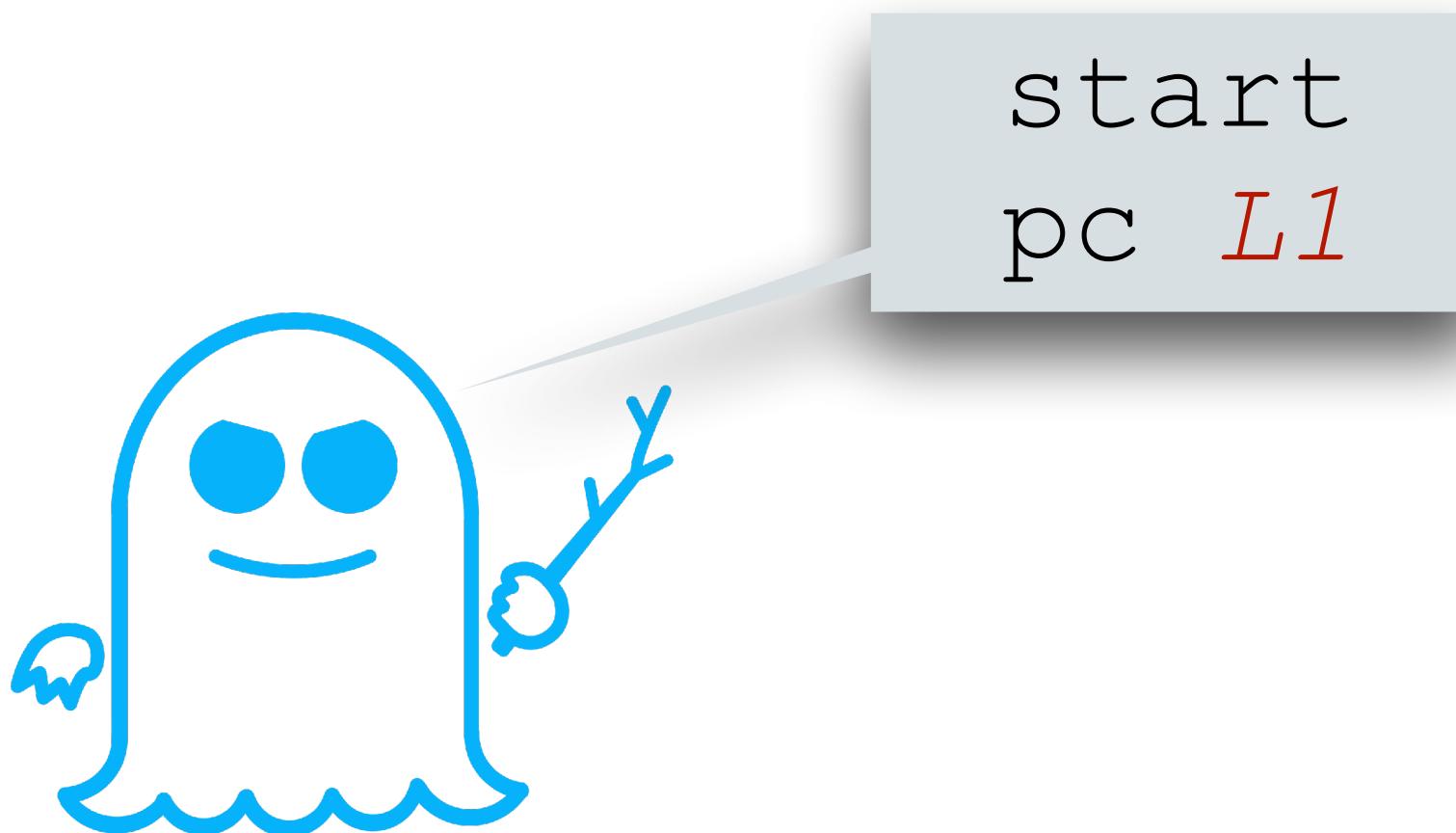
- locations of **memory accesses**
- **branch/jump** targets
- **start/end** speculative execution

Inspired by “constant-time” rqmts

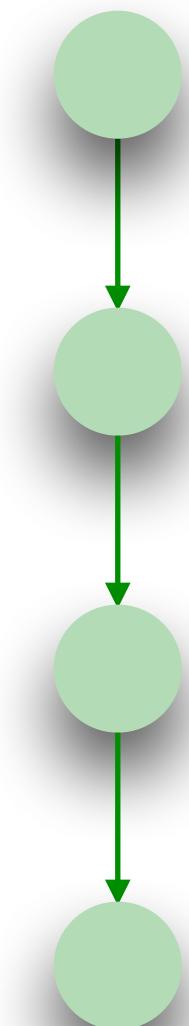


# Leakage into μarchitecture

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



start  
pc *L1*



Attacker can observe:

- locations of **memory accesses**
- **branch/jump** targets
- **start/end** speculative execution

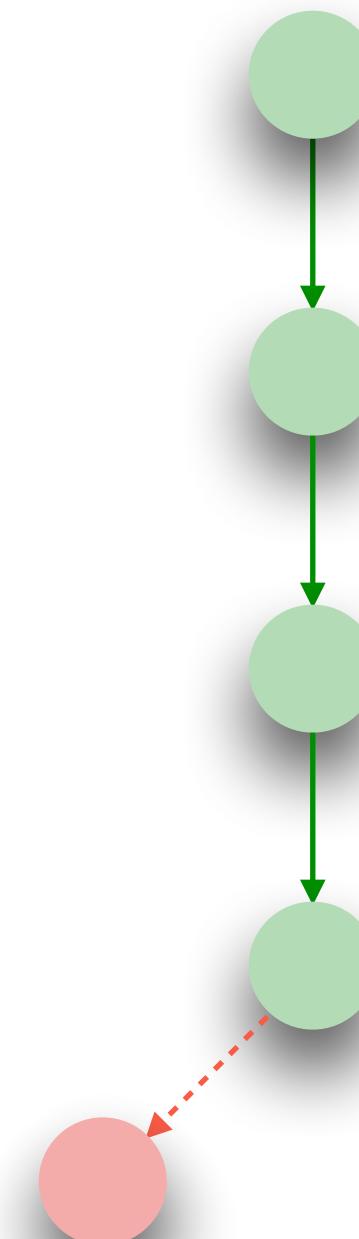
Inspired by “constant-time” rqmts

# Leakage into μarchitecture

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



load **A+x**



Attacker can observe:

- locations of **memory accesses**
- **branch/jump** targets
- **start/end** speculative execution

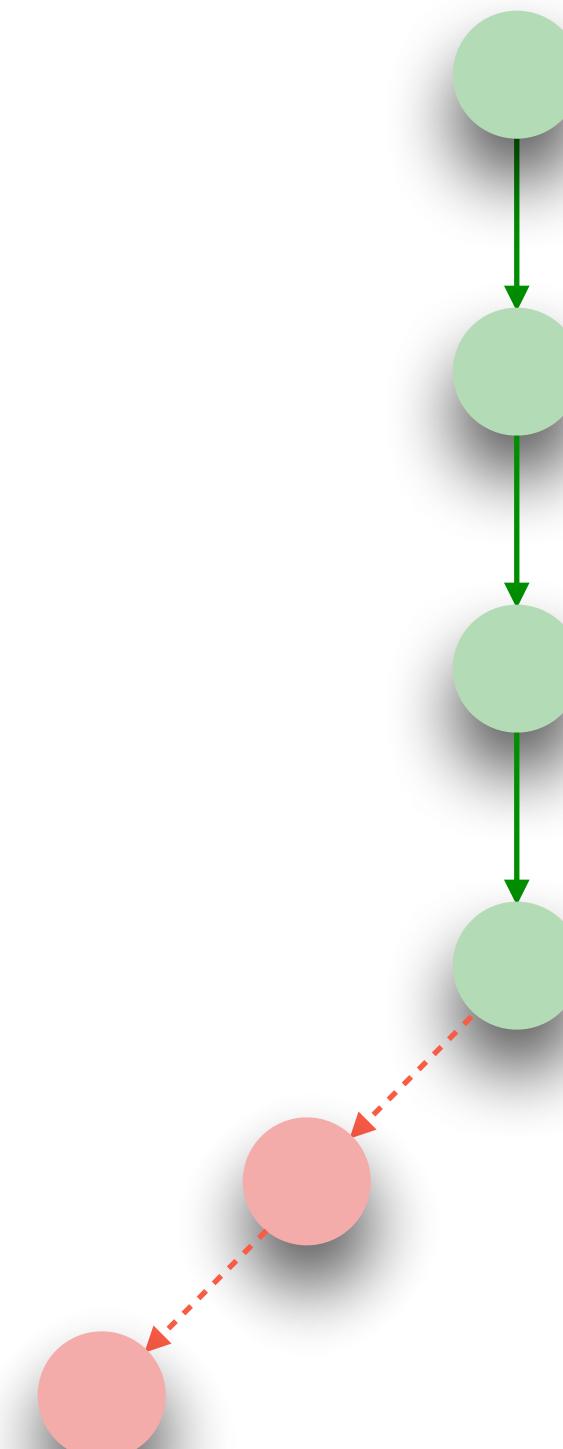
Inspired by “constant-time” rqmts

# Leakage into μarchitecture

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
    load rax, B + rax
END:
```



load **B+A** [**x**]



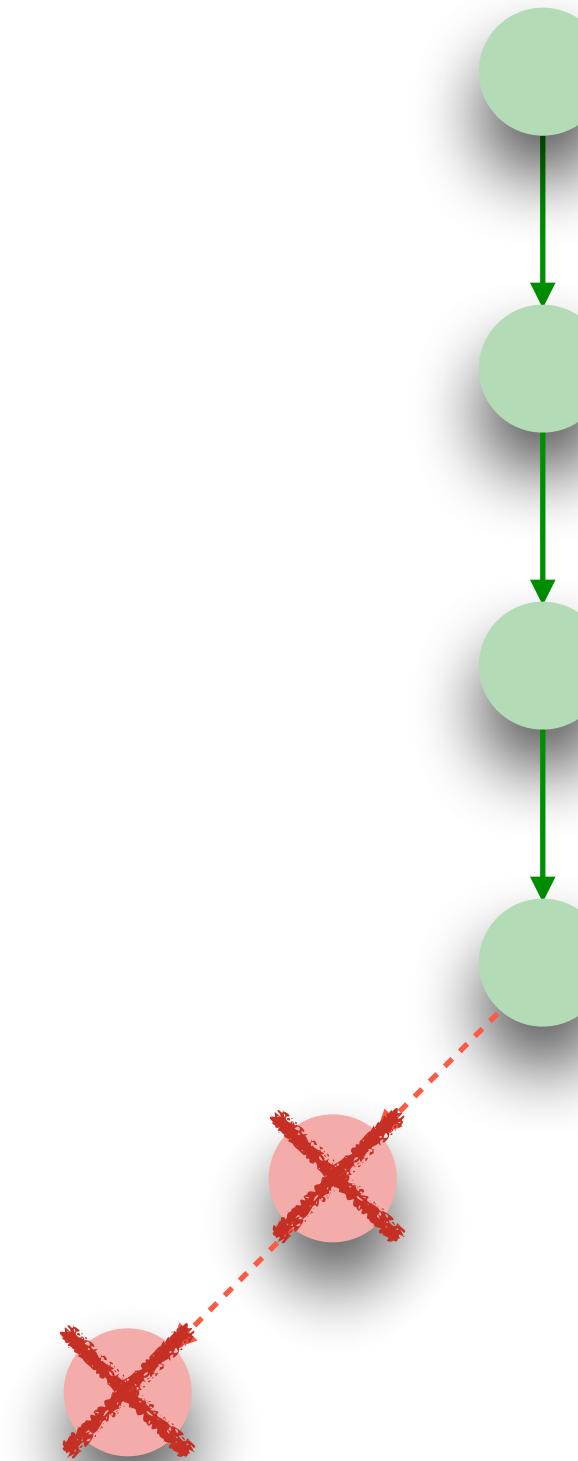
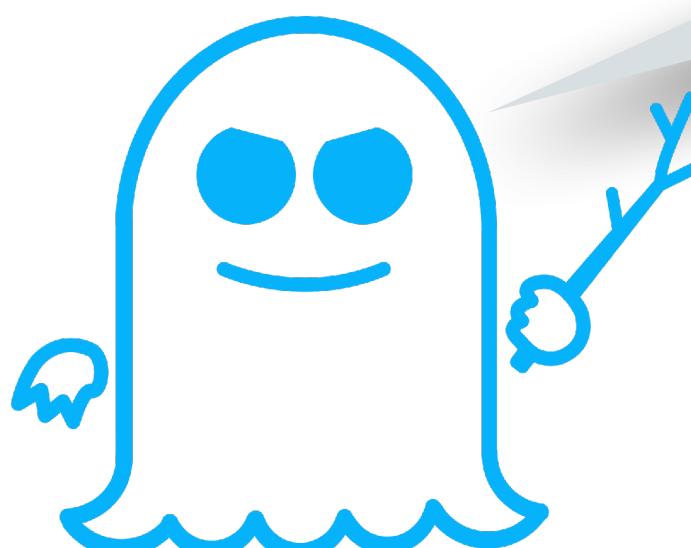
Attacker can observe:

- locations of **memory accesses**
- **branch/jump** targets
- **start/end** speculative execution

Inspired by “constant-time” rqmts

# Leakage into μarchitecture

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



rollback  
pc *END*

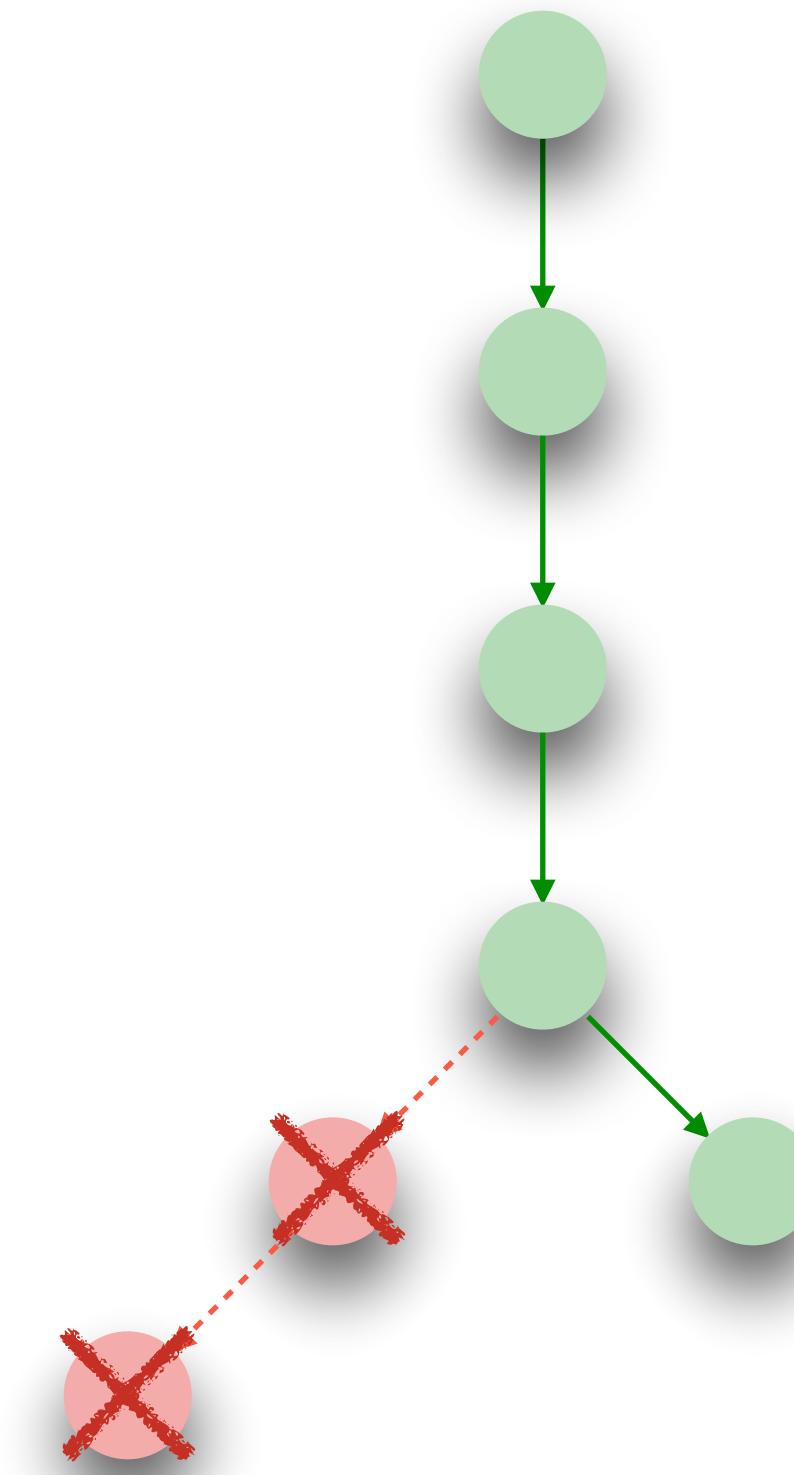
Attacker can observe:

- locations of **memory accesses**
- **branch/jump** targets
- **start/end** speculative execution

Inspired by “constant-time” rqmts

# Leakage into μarchitecture

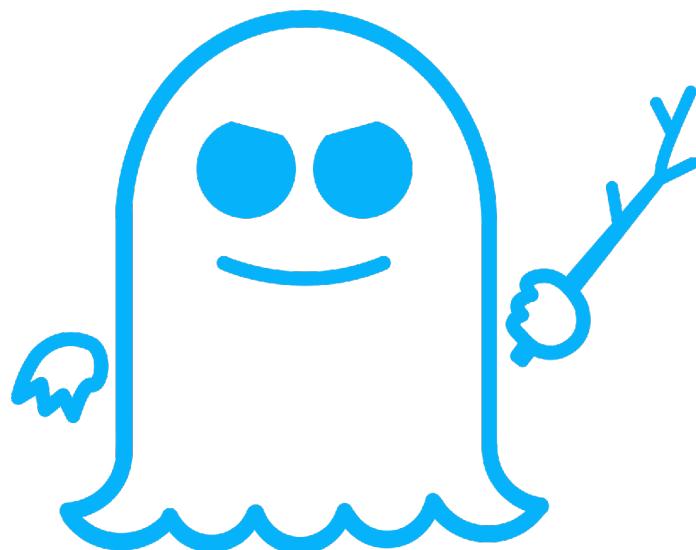
```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



Attacker can observe:

- locations of **memory accesses**
- **branch/jump** targets
- **start/end** speculative execution

Inspired by “constant-time” rqmts



# Speculative non-interference

Formally!

# Speculative non-interference

Formally!

Program **P** is **speculatively non-interferent** for prediction oracle **O** if

# Speculative non-interference

Formally!

Program  $\mathbf{P}$  is **speculatively non-interferent** for prediction oracle  $\mathbf{O}$  if

For all program states  $s$  and  $s'$ :

# Speculative non-interference

Formally!

Program  $\mathbf{P}$  is **speculatively non-interferent** for prediction oracle  $\mathbf{O}$  if

For all program states  $s$  and  $s'$ :

$$\mathbf{P}_{\text{non-spec}}(s) = \mathbf{P}_{\text{non-spec}}(s')$$

# Speculative non-interference

Formally!

Program  $\mathbf{P}$  is **speculatively non-interferent** for prediction oracle  $\mathbf{O}$  if

For all program states  $\mathbf{s}$  and  $\mathbf{s}'$ :

$$\begin{aligned} \mathbf{P}_{\text{non-spec}}(\mathbf{s}) &= \mathbf{P}_{\text{non-spec}}(\mathbf{s}') \\ \Rightarrow \mathbf{P}_{\text{spec}}(\mathbf{s}, \mathbf{O}) &= \mathbf{P}_{\text{spec}}(\mathbf{s}', \mathbf{O}) \end{aligned}$$

# Reasoning about arbitrary oracles

# Reasoning about arbitrary oracles

Always-mispredict  
speculative semantics

Mispredict ***all*** branch  
instructions

Fixed speculative window

Rollback of every transaction

# Reasoning about arbitrary oracles

Always-mispredict  
speculative semantics

Mispredict ***all*** branch  
instructions

Fixed speculative window

Rollback of every transaction

Always-mispredict is **worst-case**

$$P_{\text{am}}(\mathbf{s}) = P_{\text{am}}(\mathbf{s}') \iff$$

$$\forall \mathbf{o}. P_{\text{spec}}(\mathbf{s}, \mathbf{o}) = P_{\text{spec}}(\mathbf{s}', \mathbf{o})$$

# Reasoning about arbitrary oracles

Always-mispredict  
speculative semantics

Mispredict ***all*** branch  
instructions

Fixed speculative window

Rollback of every transaction

Always-mispredict is **worst-case**

$$P_{am}(s) = P_{am}(s') \iff$$

$$\forall O. P_{spec}(s, O) = P_{spec}(s', O)$$

If program **P** satisfies

$$\begin{aligned} \forall s, s'. P_{\text{non-spec}}(s) &= P_{\text{non-spec}}(s') \\ \Rightarrow P_{am}(s) &= P_{am}(s') \end{aligned}$$

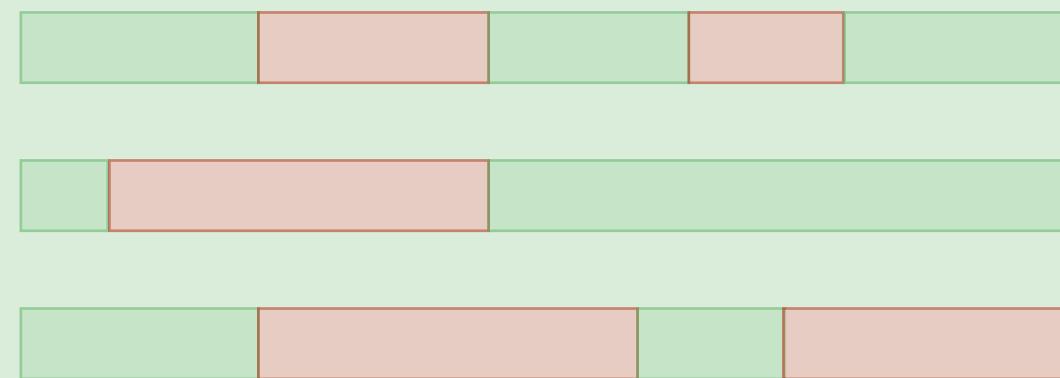
then **P** satisfies **SNI** w.r.t. all **O**

# Detecting speculative leaks

# Detecting speculative leaks

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1:    load rax, A + rcx
        load rax, B + rax
END:
```

Symbolic  
execution



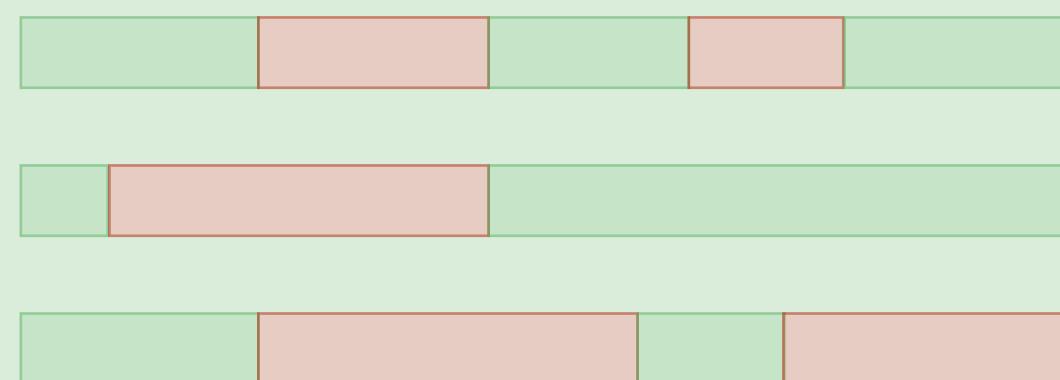
Detect leaks



# Detecting speculative leaks

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```

Symbolic  
execution



Detect leaks



**Symbolic trace:** path condition +  
observations along the symbolic path

# Symbolic execution

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```

# Symbolic execution

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



*Always mispredict*  
branch instructions

# Symbolic execution

```
rax <- A_size          true  
rcx <- x  
jmp rcx≥rax, END  
L1: load rax, A + rcx  
      load rax, B + rax  
END:
```

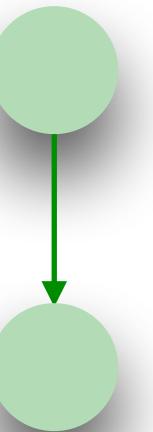


*Always mispredict*  
branch instructions

# Symbolic execution

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```

true

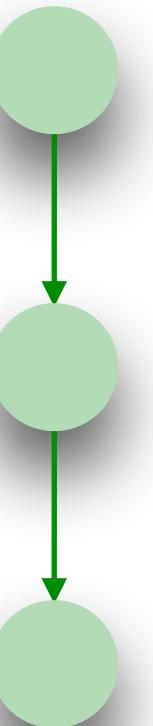


*Always mispredict*  
branch instructions

# Symbolic execution

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```

true



*Always mispredict*  
branch instructions

# Symbolic execution

```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
```

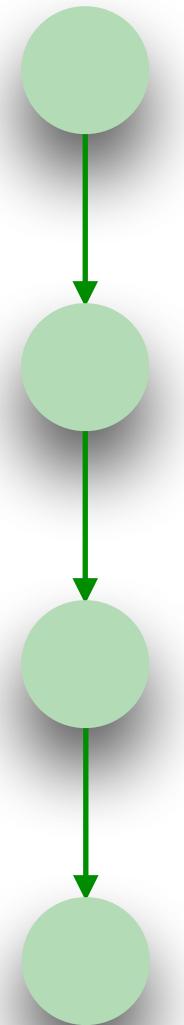
*L1:* load rax, **A** + rcx  
load rax, **B** + rax

*END:*

*Always mispredict*  
branch instructions



true



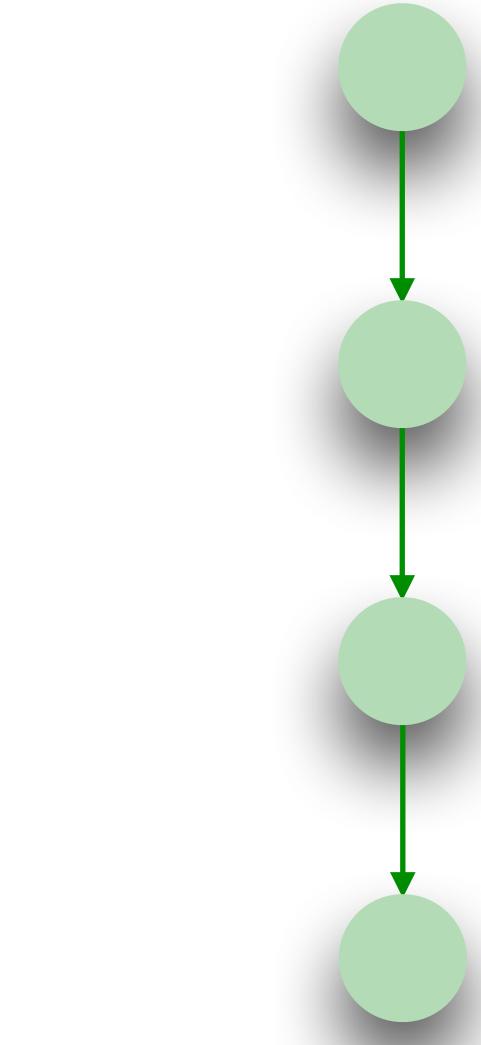
# Symbolic execution

```
rax <- A_size  
rcx <- x  
jmp rcx≥rax, END  
L1: load rax, A + rcx  
     load rax, B + rax  
END:
```

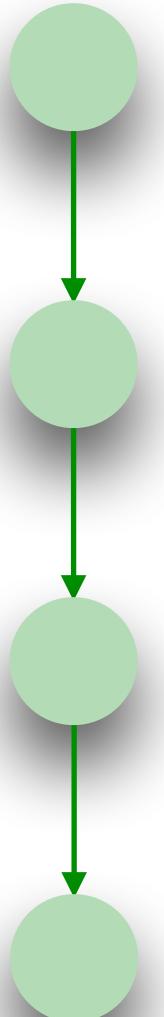


*Always mispredict*  
branch instructions

$x \geq A\_size$

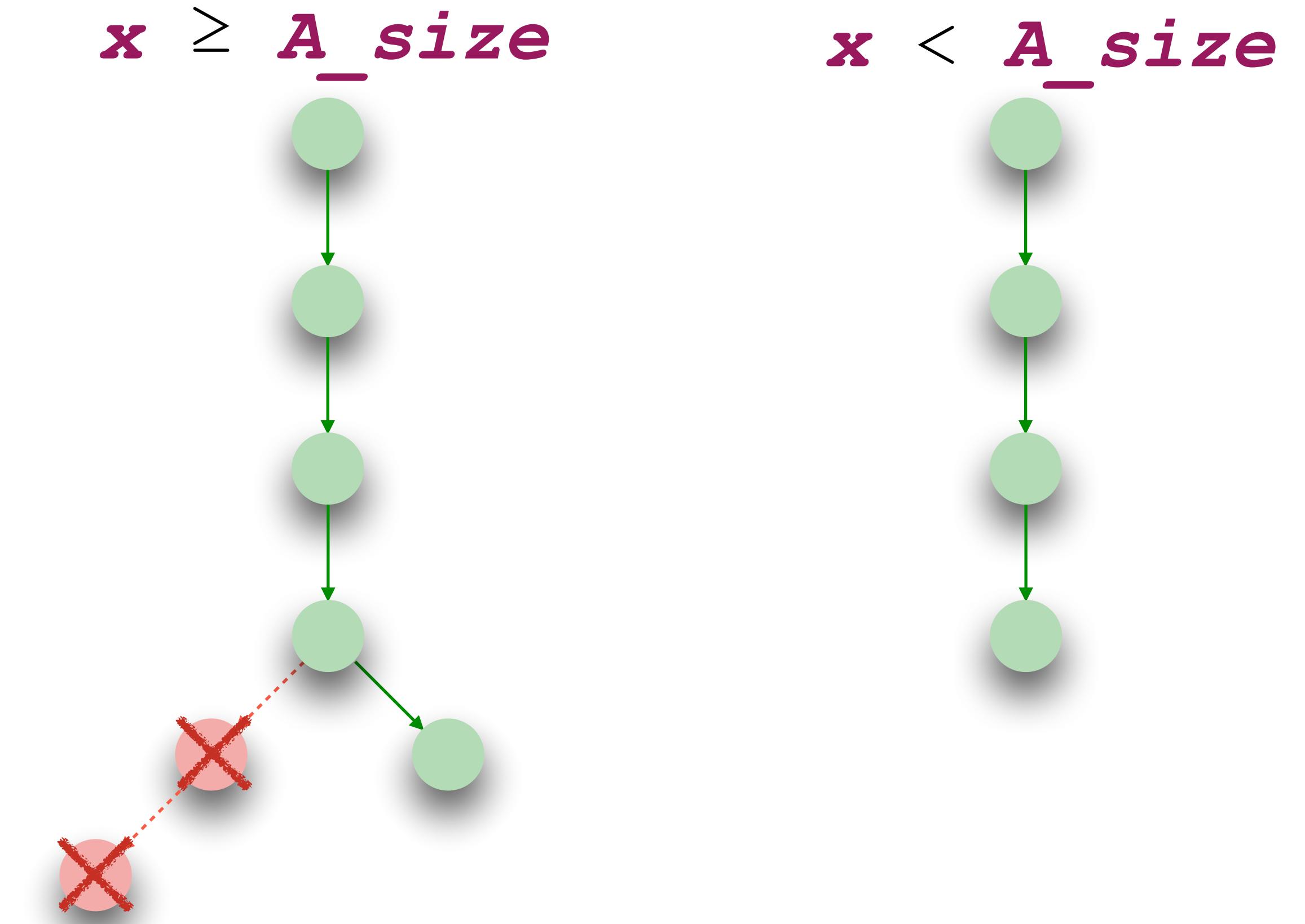


$x < A\_size$



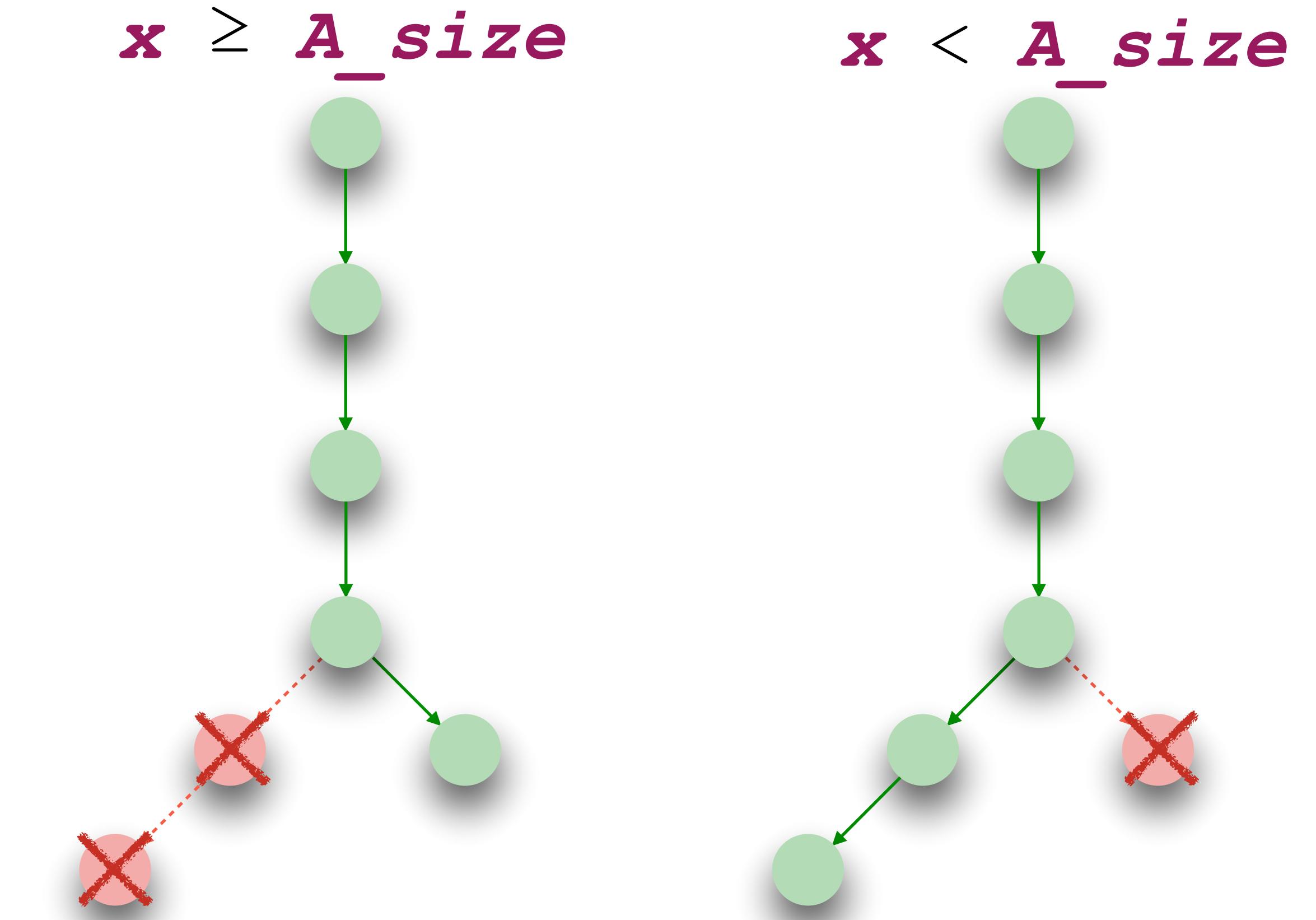
# Symbolic execution

```
    rax <- A_size
    rcx <- x
    jmp rcx≥rax, END
L1: load rax, A + rcx
    load rax, B + rax
END:
```



# Symbolic execution

```
rax <- A_size  
rcx <- x  
jmp rcx≥rax, END  
L1: load rax, A + rcx  
     load rax, B + rax  
END:
```

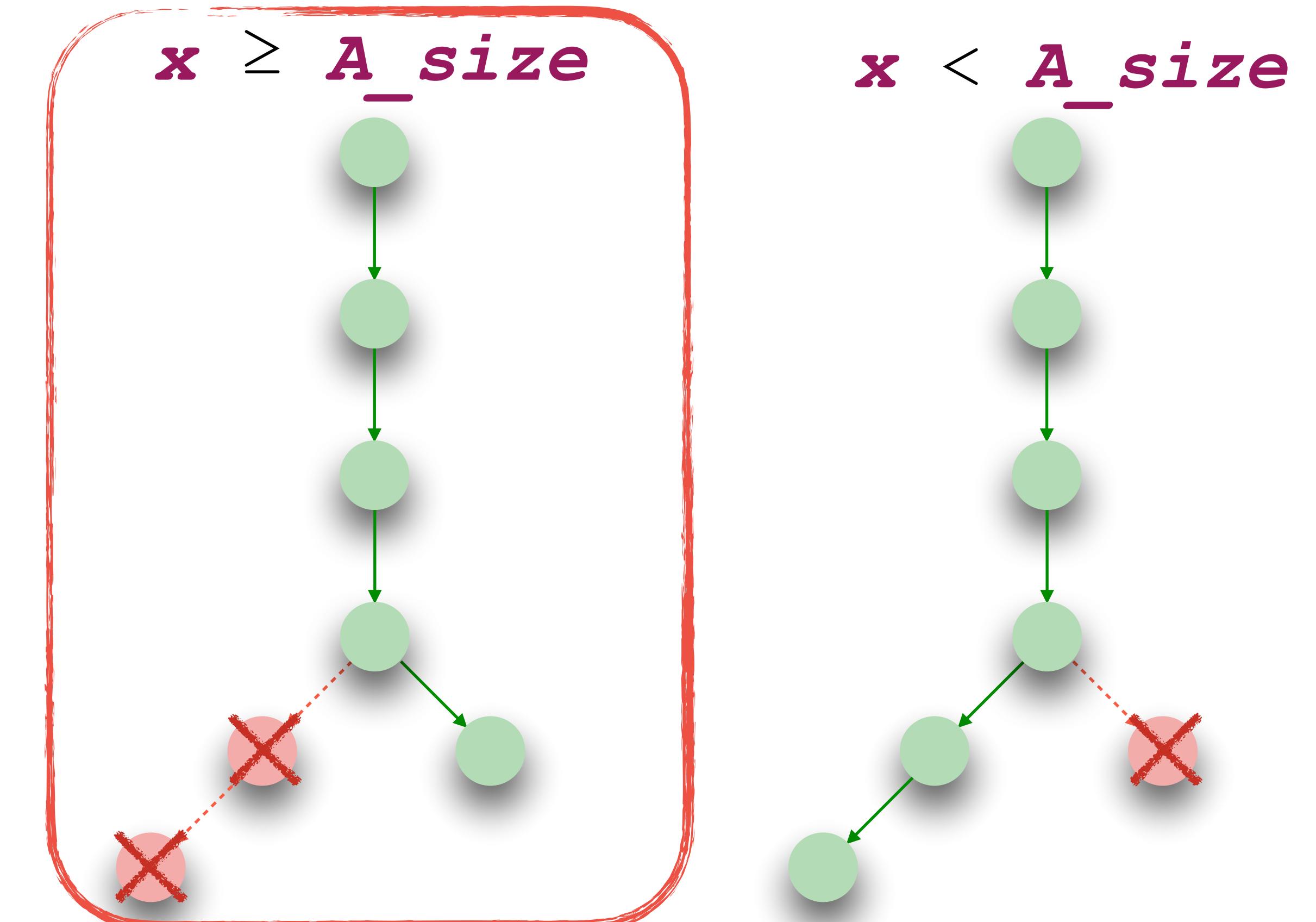


# Symbolic execution

```
rax <- A_size  
rcx <- x  
jmp rcx≥rax, END  
L1: load rax, A + rcx  
     load rax, B + rax  
END:
```



*Always mispredict*  
branch instructions

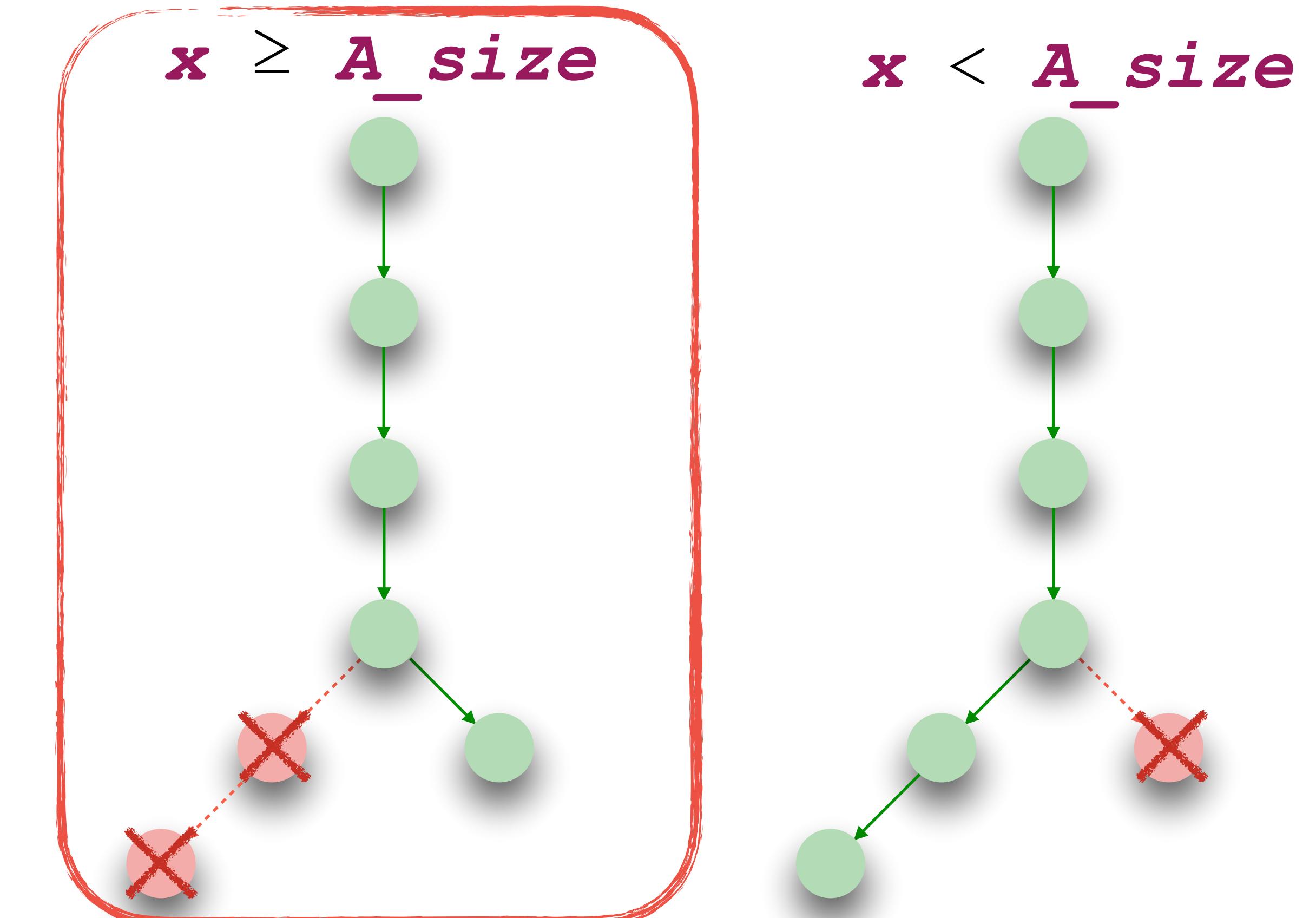


# Symbolic execution

```
rax <- A_size  
rcx <- x  
jmp rcx≥rax, END  
L1: load rax, A + rcx  
     load rax, B + rax  
END:
```



*Always mispredict*  
branch instructions



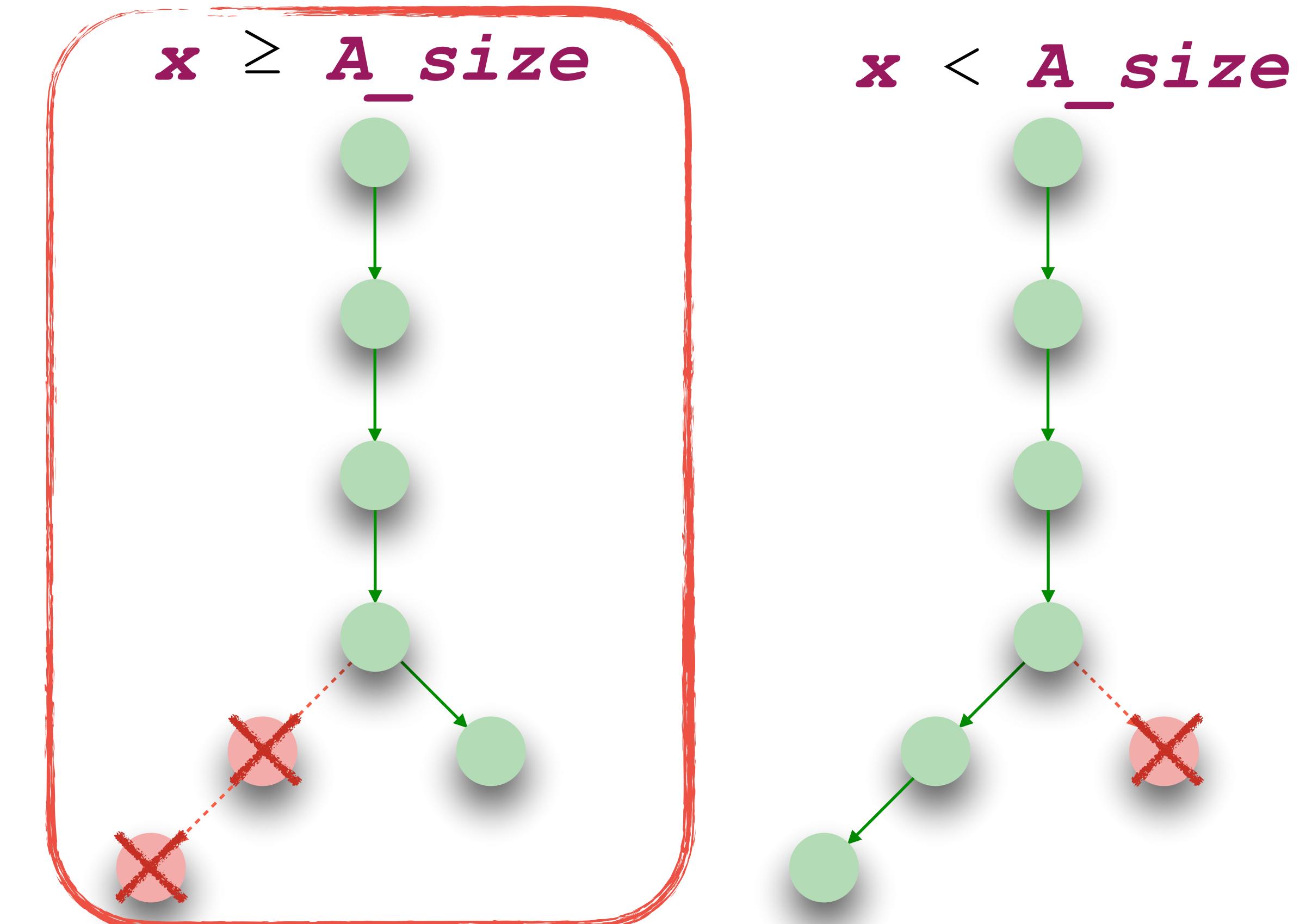
start pc L1 load A+x load B+A[x] rollback pc END

# Symbolic execution

```
rax <- A_size  
rcx <- x  
jmp rcx≥rax, END  
L1: load rax, A + rcx  
     load rax, B + rax  
END:
```



*Always mispredict*  
branch instructions



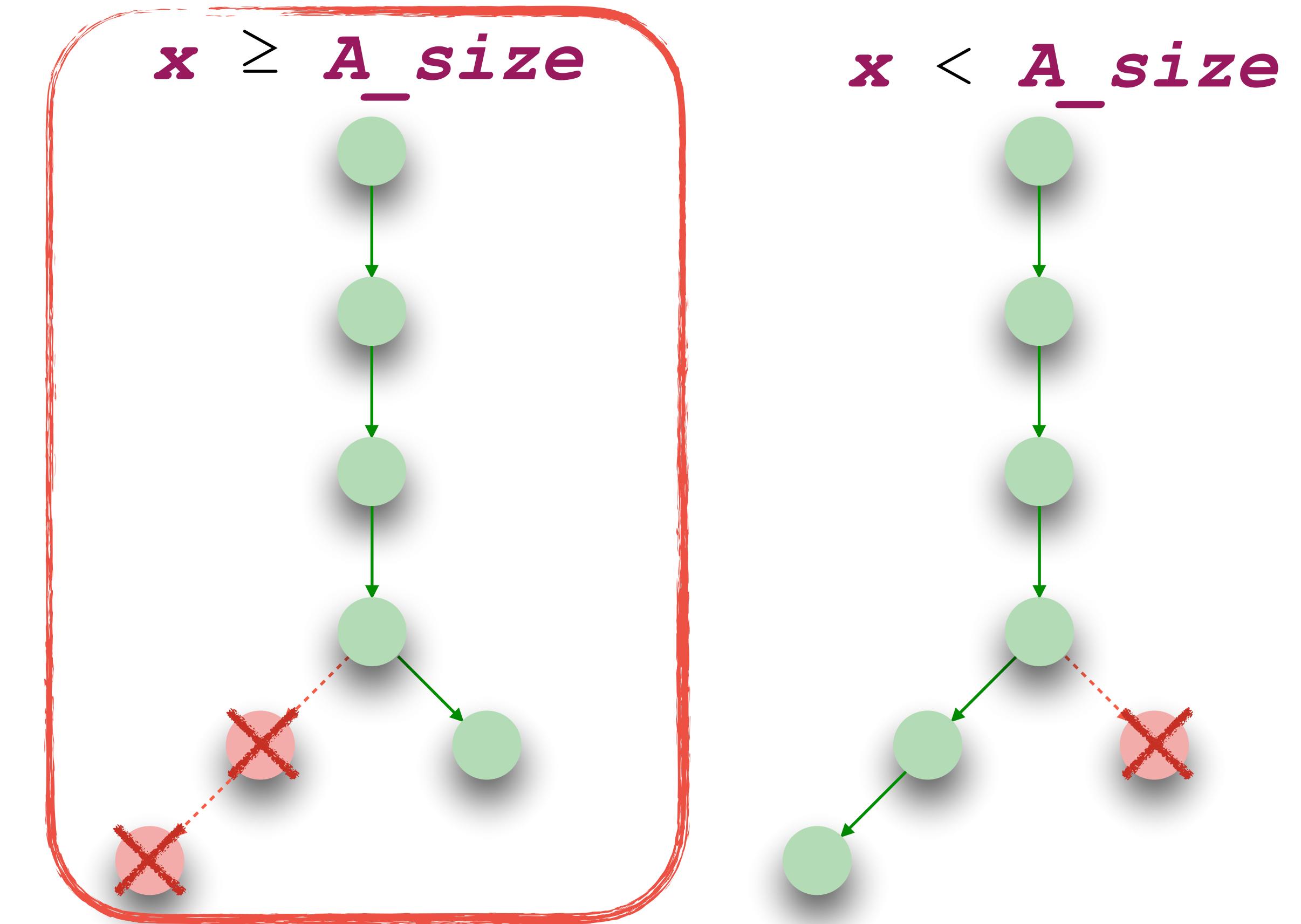
start pc L1 load A+x load B+A[x] rollback pc END

# Symbolic execution

```
rax <- A_size  
rcx <- x  
jmp rcx≥rax, END  
L1: load rax, A + rcx  
     load rax, B + rax  
END:
```



*Always mispredict*  
branch instructions

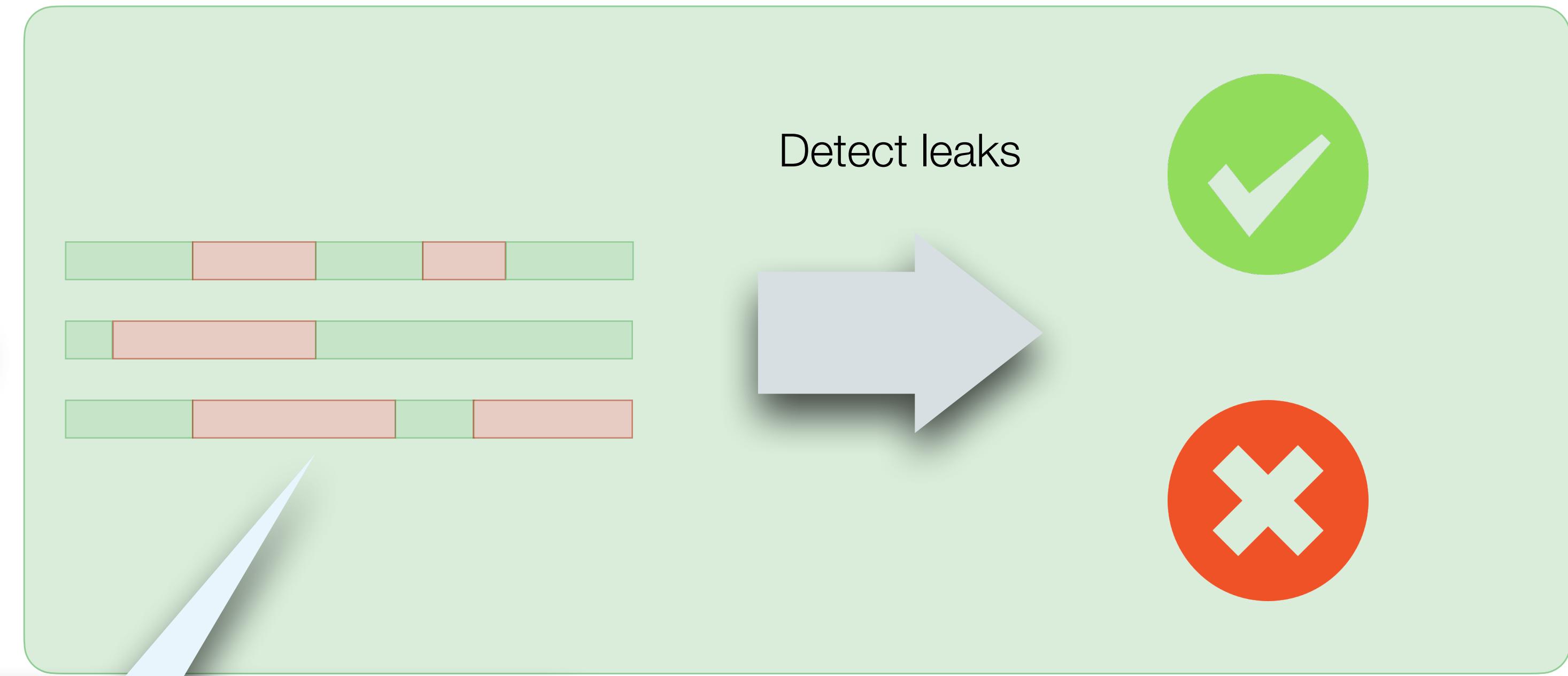


start	pc	L1	load A+x	load B+A[x]	rollback	pc	END
-------	----	----	----------	-------------	----------	----	-----

# Detecting speculative leaks

```
rax <- A_size  
rcx <- x  
jmp rcx>=rax, END  
L1:    load rax, A + rcx  
       load rax, B + rax  
END:
```

Symbolic execution



**Symbolic trace:** path condition +  
observations along the symbolic path

# Detecting speculative leaks

For each symbolic trace  $\tau \in traces(prg)$

```
if MemLeak( $\tau$ ) then  
    return INSECURE  
if CtrlLeak( $\tau$ ) then  
    return INSECURE  
return SECURE
```

L1:

END:

rax  
rcx  
jmp  
load  
load

# Detecting speculative leaks

For each symbolic trace  $\tau \in traces(prg)$

```
if MemLeak( $\tau$ ) then  
    return INSECURE  
if CtrlLeak( $\tau$ ) then  
    return INSECURE  
return SECURE
```

rax  
rcx  
jmp  
load  
load

L1:

END:



# Memory leaks

Speculative memory accesses **must** depend only on

- Non-sensitive information
- Non-speculative observations

# Memory leaks

Speculative memory accesses **must** depend only on

- Non-sensitive information
- Non-speculative observations

$\tau$



# Memory leaks

Speculative memory accesses **must** depend only on

- Non-sensitive information
- Non-speculative observations

$\tau$



$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

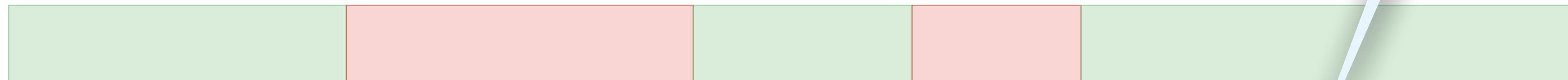
# Memory leaks

Speculative memory accesses **must** depend only on

- Non-sensitive information
- Non-speculative observations

Check with self-composition

$\tau$



$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

# Memory leaks

Speculative memory accesses **must** depend only on

- Non-sensitive information
- Non-speculative observations

Check with self-composition

$\tau$



$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

$s_1$

$s_2$

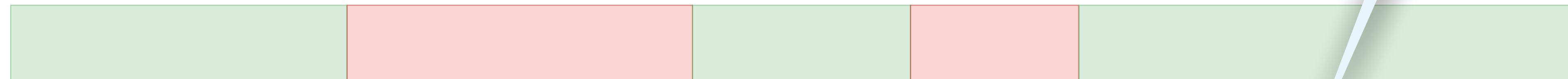
# Memory leaks

Speculative memory accesses **must** depend only on

- Non-sensitive information
- Non-speculative observations

Check with self-composition

$\tau$



$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

Equivalent  
wrt **policy**

$s_1$

$s_2$

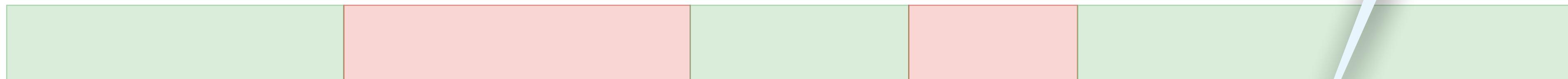
# Memory leaks

Speculative memory accesses **must** depend only on

- Non-sensitive information
- Non-speculative observations

Check with self-composition

$\tau$



$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$

Equivalent  
wrt **policy**

$$s_1 \models \varphi$$

$$s_2 \models \varphi$$

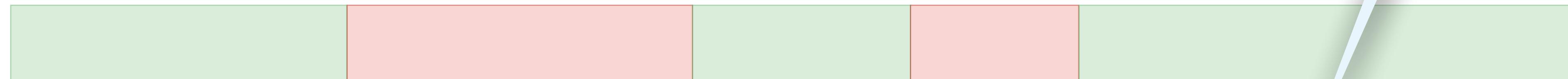
# Memory leaks

Speculative memory accesses **must** depend only on

- Non-sensitive information
- Non-speculative observations

Check with self-composition

$\tau$



$$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$$

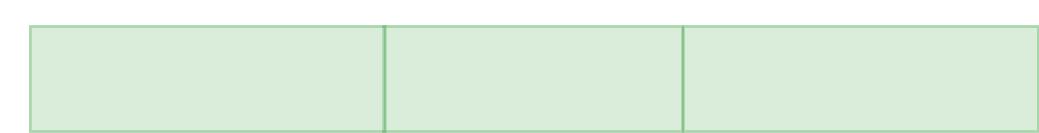
Equivalent  
wrt **policy**

$$s_1 \models \varphi$$



||

$$s_2 \models \varphi$$



# Memory leaks

Speculative memory accesses **must** depend only on

- Non-sensitive information
- Non-speculative observations

Check with self-composition

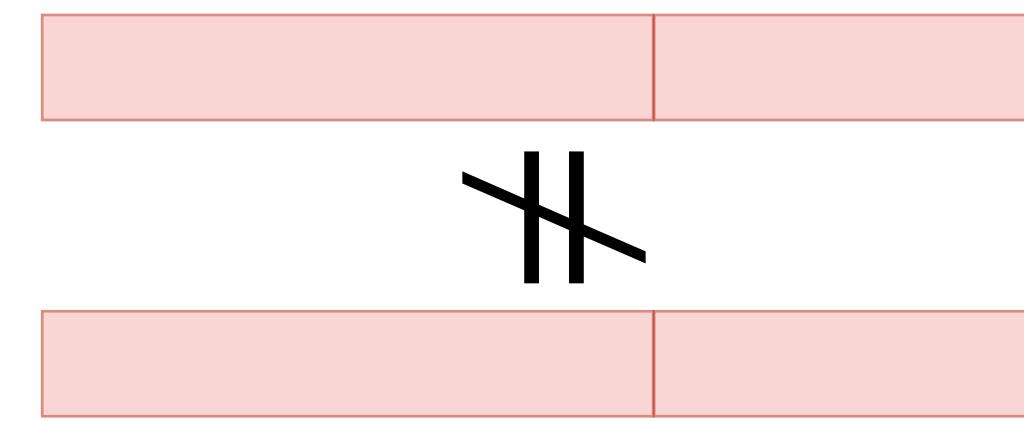
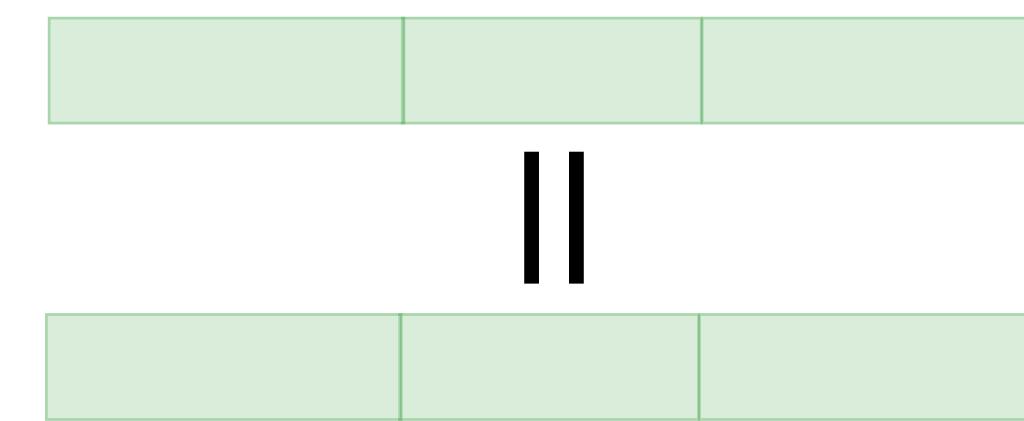
$\tau$



$pathCnd(\tau) \wedge obsEqv(\tau|_{non-spec}) \wedge \neg obsEqv(\tau|_{spec})$

Equivalent  
wrt **policy**

$s_1 \models \varphi$



# Spectector + Case studies

# Spectector



```
mov    rax, A_size  
mov    rcx, x  
cmp    rcx, rax  
jae    END  
L1:   mov    rax, A[rcx]  
      mov    rax, B[rax]
```

x64 to μASM

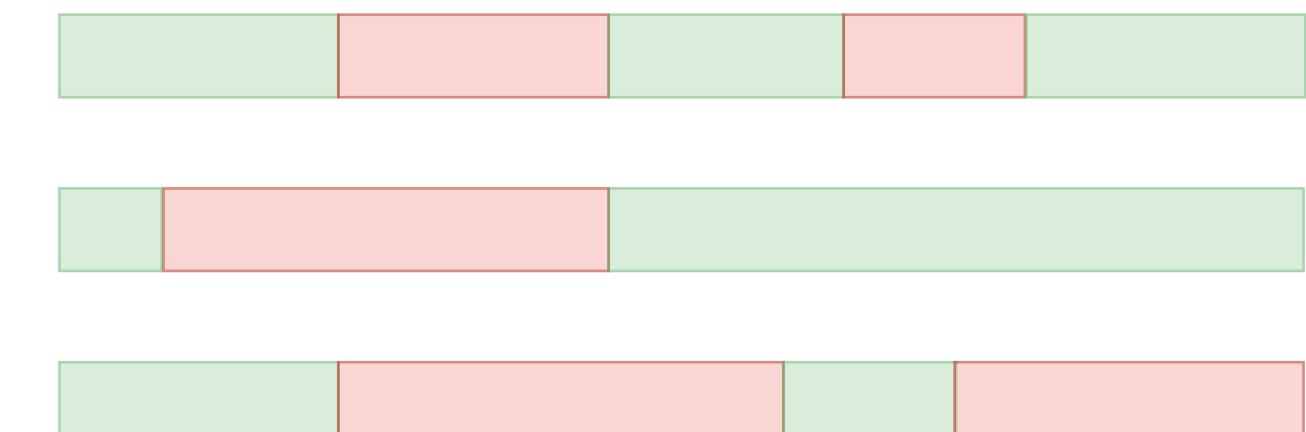
```
L1:  
END:  
      rax <- A_size  
      rcx <- x  
      jmp rcx>=rax, END  
      load rax, A + rcx  
      load rax, B + rax
```



Check for speculative leaks



Symbolic  
execution



# Spectector



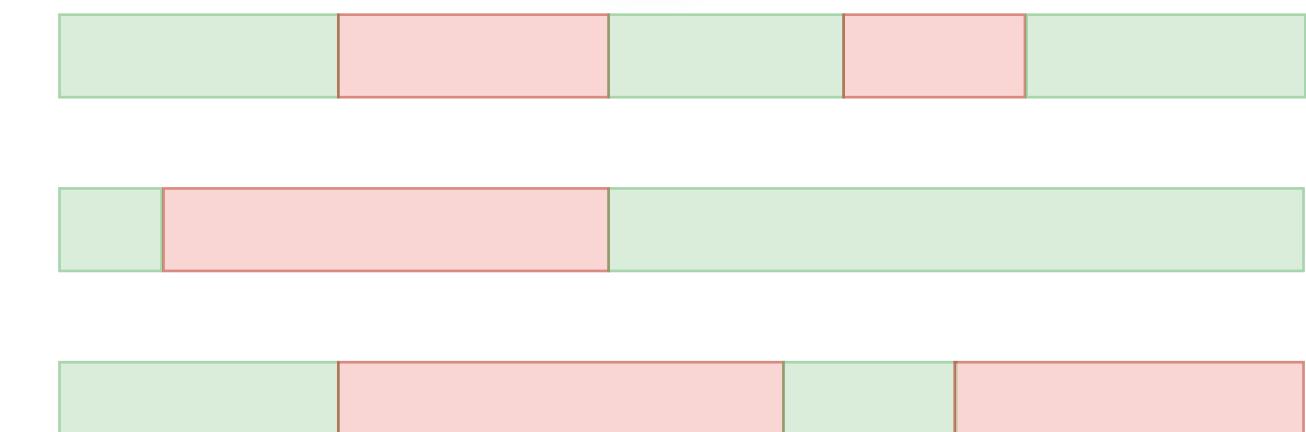
```
L1: mov    rax, A_size  
      mov    rcx, x  
      cmp    rcx, rax  
      jae    END  
      mov    rax, A  
      mov    rax, B
```

```
mov    rax, A_size  
mov    rcx, x  
cmp    rcx, rax  
END  
rax, A  
rax, B
```

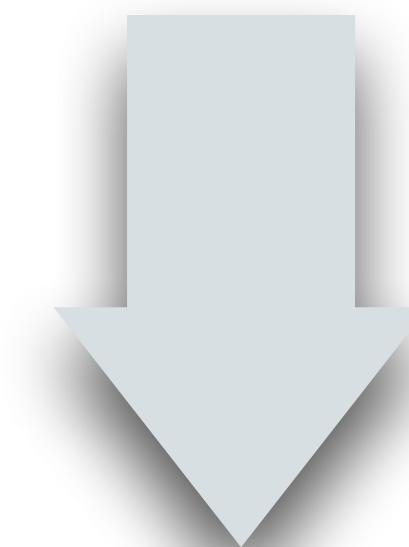
x64 to µASM



Check for speculative leaks



```
rax <- A_size  
rcx <- x  
jmp rcx>=rax, END  
load rax, A + rcx  
load rax, B + rax
```



Symbolic  
execution

# Case study: compiler mitigations

## *Target:*

- 15 variants of Spectre V1 by Paul Kocher\*
- Compiled with Microsoft Visual C++, Intel ICC, and Clang with different mitigations and optimization levels
- 240 assembly programs of up to 200 instructions each

## *How:*

- Use Spectector to prove security or detect leaks

\* Paul Kocher - Spectre Mitigations in Microsoft C/C++ Compiler – <https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html>

# Results

Ex.	VCC						ICC						CLANG					
	UNP		FEN 19.15		FEN 19.20		UNP		FEN		UNP		FEN		UNP		FEN	
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02
01	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●
02	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●
03	○	○	●	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
04	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
05	○	○	●	○	●	○	○	○	●	●	○	○	●	●	●	●	●	●
06	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
07	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
08	○	●	○	●	○	●	○	●	●	●	○	●	●	●	●	●	●	●
09	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
10	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	○
11	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
12	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
13	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
14	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
15	○	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	○	●

# Results

Ex.	VCC						ICC						CLANG					
	UNP		FEN 19.15		FEN 19.20		UNP		FEN		UNP		FEN		UNP		FEN	
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02
01	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●
02	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●
03	○	○	●	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
04	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
05	○	○	●	○	●	○	○	○	●	●	○	○	●	●	●	●	●	●
06	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
07	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
08	○	●	○	●	○	●	○	●	●	●	○	●	●	●	●	●	●	●
09	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
10	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	○
11	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
12	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
13	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
14	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
15	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	○	●

# Results

Ex.	VCC				ICC				CLANG					
	UNP		FEN 19.15		FEN 19.20		UNP		FEN		UNP		FEN	
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02
01	○	○	●	●	●	●	○	○	●	●	○	○	●	●
02	○	○	●	●	●	●	○	○	●	●	○	○	●	●
03	○	○	●	○	●	●	○	○	●	●	○	○	●	●
04	○	○	○	○	●	●	○	○	●	●	○	○	●	●
05	○	○	●	○	●	○	○	○	●	●	○	○	●	●
06	○	○	○	○	○	○	○	○	●	●	○	○	●	●
07	○	○	○	○	○	○	○	○	●	●	○	○	●	●
08	○	●	○	●	○	●	○	●	●	●	○	●	●	●
09	○	○	○	○	○	○	○	○	●	●	○	○	●	●
10	○	○	○	○	○	○	○	○	●	●	○	○	●	●
11	○	○	○	○	○	○	○	○	●	●	○	○	●	●
12	○	○	○	○	●	●	○	○	●	●	○	○	●	●
13	○	○	○	○	○	○	○	○	●	●	○	○	●	●
14	○	○	○	○	●	●	○	○	●	●	○	○	●	●
15	○	○	○	○	○	○	○	○	○	●	○	○	●	●

# Results

Ex.	VCC						ICC						CLANG					
	UNP		FEN 19.15		FEN 19.20		UNP		FEN		UNP		FEN		SLH			
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	
01	o	o	●	●	●	●	o	o	●	●	o	o	●	●	●	●	●	
02	o	o	●	●	●	●	o	o	●	●	o	o	●	●	●	●	●	
03	o	o	●	o	●	●	o	o	●	●	o	o	●	●	●	●	●	
04	o	o	o	o	●	●	o	o	●	●	o	o	●	●	●	●	●	
05	o	o	●	o	●	o	o	o	●	●	o	o	●	●	●	●	●	
06	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
07	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
08	o	●	o	●	o	●	o	●	●	●	o	●	●	●	●	●	●	
09	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
10	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	o	
11	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
12	o	o	o	o	●	●	o	o	●	●	o	o	●	●	●	●	●	
13	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
14	o	o	o	o	●	●	o	o	●	●	o	o	●	●	●	●	●	
15	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	o	●	

# Results

No countermeasures

Ex.	VCC						ICC						CLANG					
	UNP	FEN	19.15	FEN	19.20	UNP	FEN	UNP	FEN	UNP	FEN	SLH	-00	-02	-00	-02	-00	-02
01	○	○	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●	●
02	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●
03	○	○	●	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
04	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
05	○	○	●	○	●	○	○	○	●	●	○	○	●	●	●	●	●	●
06	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
07	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
08	○	●	○	●	○	●	○	●	●	●	○	●	●	●	●	●	●	●
09	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
10	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	○
11	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
12	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
13	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
14	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
15	○	○	○	○	○	○	○	○	○	●	●	○	●	●	●	●	○	●

# Results

Automated insertion of fences

Ex.	VCC					ICC					CLANG				
	UNP	FEN 19.15	FEN 19.20	UNP	FEN	UNP	FEN	UNP	FEN	SLH					
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	
01	o	o	●	●	●	o	o	●	●	●	●	●	●	●	●
02	o	o	●	●	●	o	o	●	●	o	o	●	●	●	●
03	o	o	●	o	●	●	o	●	●	o	o	●	●	●	●
04	o	o	o	o	●	●	o	●	●	o	o	●	●	●	●
05	o	o	●	o	●	o	o	●	●	o	o	●	●	●	●
06	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●
07	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●
08	o	●	o	●	o	●	o	●	●	o	●	●	●	●	●
09	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●
10	o	o	o	o	o	o	o	●	●	o	o	●	●	●	o
11	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●
12	o	o	o	o	●	●	o	●	●	o	o	●	●	●	●
13	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●
14	o	o	o	o	●	●	o	o	●	o	o	●	●	●	●
15	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●

# Results

Speculative load  
hardening

Ex.	VCC						ICC						CLANG					
	UNP	FEN	19.15	FEN	19.20	UNP	FEN	UNP	FEN	UNP	FEN	UNP	FEN	SLH				
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02
01	○	○	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●	●
02	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●
03	○	○	●	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
04	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
05	○	○	●	○	●	○	○	○	●	●	○	○	●	●	●	●	●	●
06	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
07	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
08	○	●	○	●	○	●	○	●	●	●	○	●	●	●	●	●	●	●
09	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
10	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	○
11	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
12	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
13	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
14	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
15	○	○	○	○	○	○	○	○	○	●	●	○	●	●	●	●	○	●

# Results

Ex.	VCC						ICC						CLANG					
	UNP		FEN 19.15		FEN 19.20		UNP		FEN		UNP		FEN		SLH			
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	
01	o	o	●	●	●	●	o	o	●	●	o	o	●	●	●	●	●	
02	o	o	●	●	●	●	o	o	●	●	o	o	●	●	●	●	●	
03	o	o	●	o	●	●	o	o	●	●	o	o	●	●	●	●	●	
04	o	o	o	o	●	●	o	o	●	●	o	o	●	●	●	●	●	
05	o	o	●	o	●	o	o	o	●	●	o	o	●	●	●	●	●	
06	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
07	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
08	o	●	o	●	o	●	o	●	●	●	o	●	●	●	●	●	●	
09	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
10	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	o	
11	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
12	o	o	o	o	●	●	o	o	●	●	o	o	●	●	●	●	●	
13	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	●	●	
14	o	o	o	o	●	●	o	o	●	●	o	o	●	●	●	●	●	
15	o	o	o	o	o	o	o	o	●	●	o	o	●	●	●	o	●	

# Results

Ex.	VCC						ICC						CLANG					
	UNP		FEN 19.15		FEN 19.20		UNP		FEN		UNP		FEN		UNP		FEN	
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02
01	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●
02	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●	●	●
03	○	○	●	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
04	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
05	○	○	●	○	●	○	○	○	●	●	○	○	●	●	●	●	●	●
06	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
07	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
08	○	●	○	●	○	●	○	●	●	●	○	●	●	●	●	●	●	●
09	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
10	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	○
11	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
12	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
13	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●	●	●
14	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●	●	●
15	○	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	○	●

# Results

Ex.	VCC			ICC			CLANG		
	UNP		FEN	19.15			UNP	FEN	SLH
	-00	-02							
01	o	o							•
02	o	o							•
03	o	o							•
04	o	o							•
05	o	o							•
06	o	o							•
07	o	o							•
08	o	•							•
09	o	o							•
10	o	o							•
11	o	o							•
12	o	o							•
13	o	o							•
14	o	o	o	o	•	•	o	o	•
15	o	o	o	o	o	o	o	o	•

## Summary

- Leaks in all unprotected programs (except example #08 with optimizations)
- Confirm all vulnerabilities in VCC pointed out by Paul Kocher
- Programs with fences (ICC and Clang) are secure
- Unnecessary fences
- Programs with SLH are secure except #10 and #15

# Case study: scalability

**Target:** Xen hypervisors

***Main challenges for scalability:***

- Policy definition
- ISA coverage
- Path explosion

**How:**

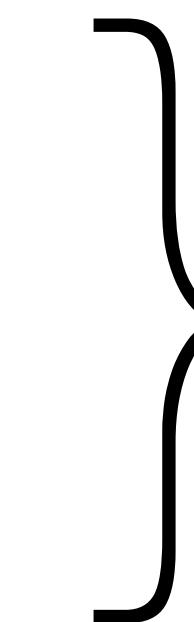
- Analyze scalability of checking SNI ***relative to*** symbolic execution
  - 24'000 symbolic paths of < 10'000 instructions (from ~ 4'000 functions)

# Case study: scalability

**Target:** Xen hypervisors

## ***Main challenges for scalability:***

- Policy definition
- ISA coverage
- Path explosion

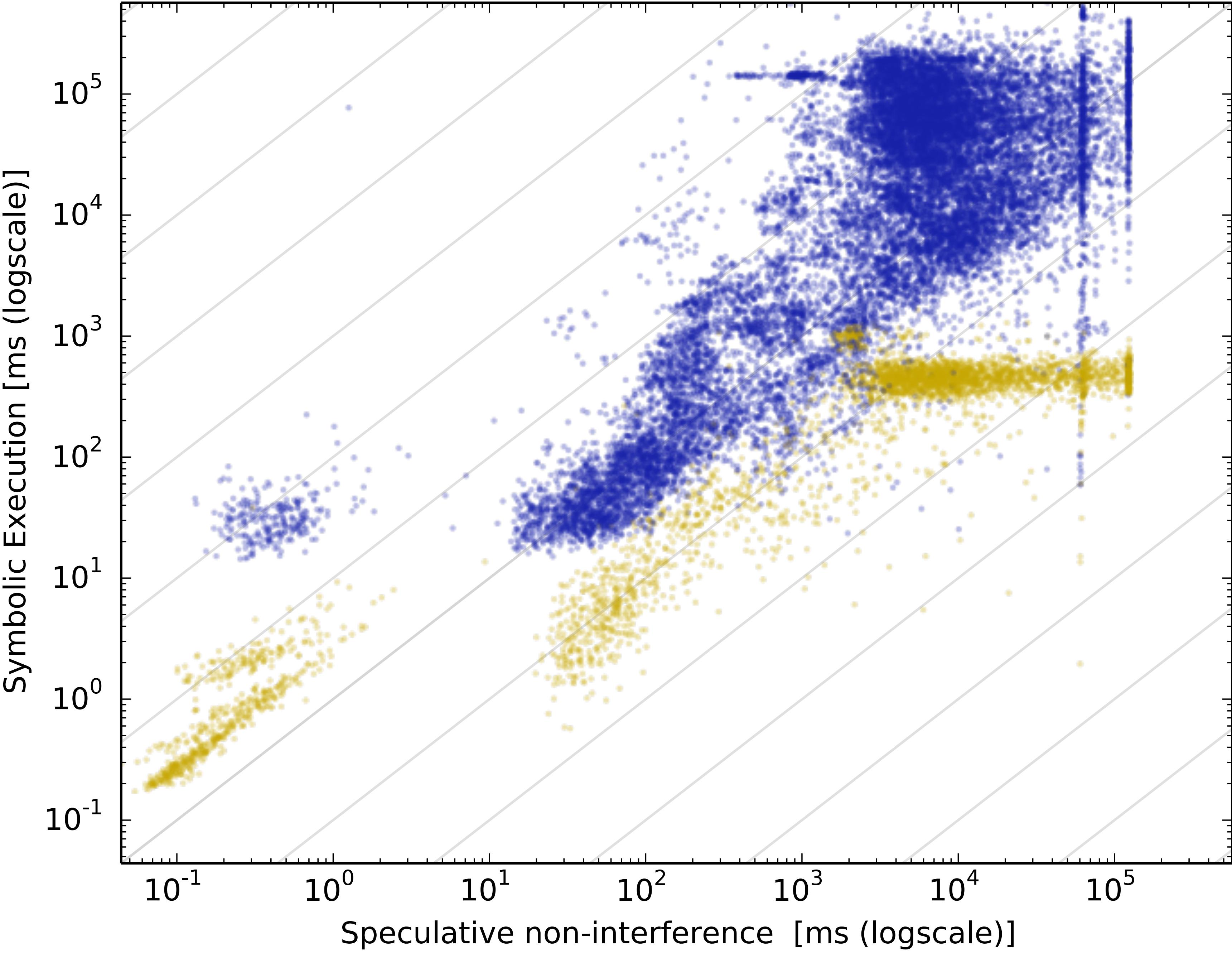


Trade-offs affect analysis soundness and completeness

## ***How:***

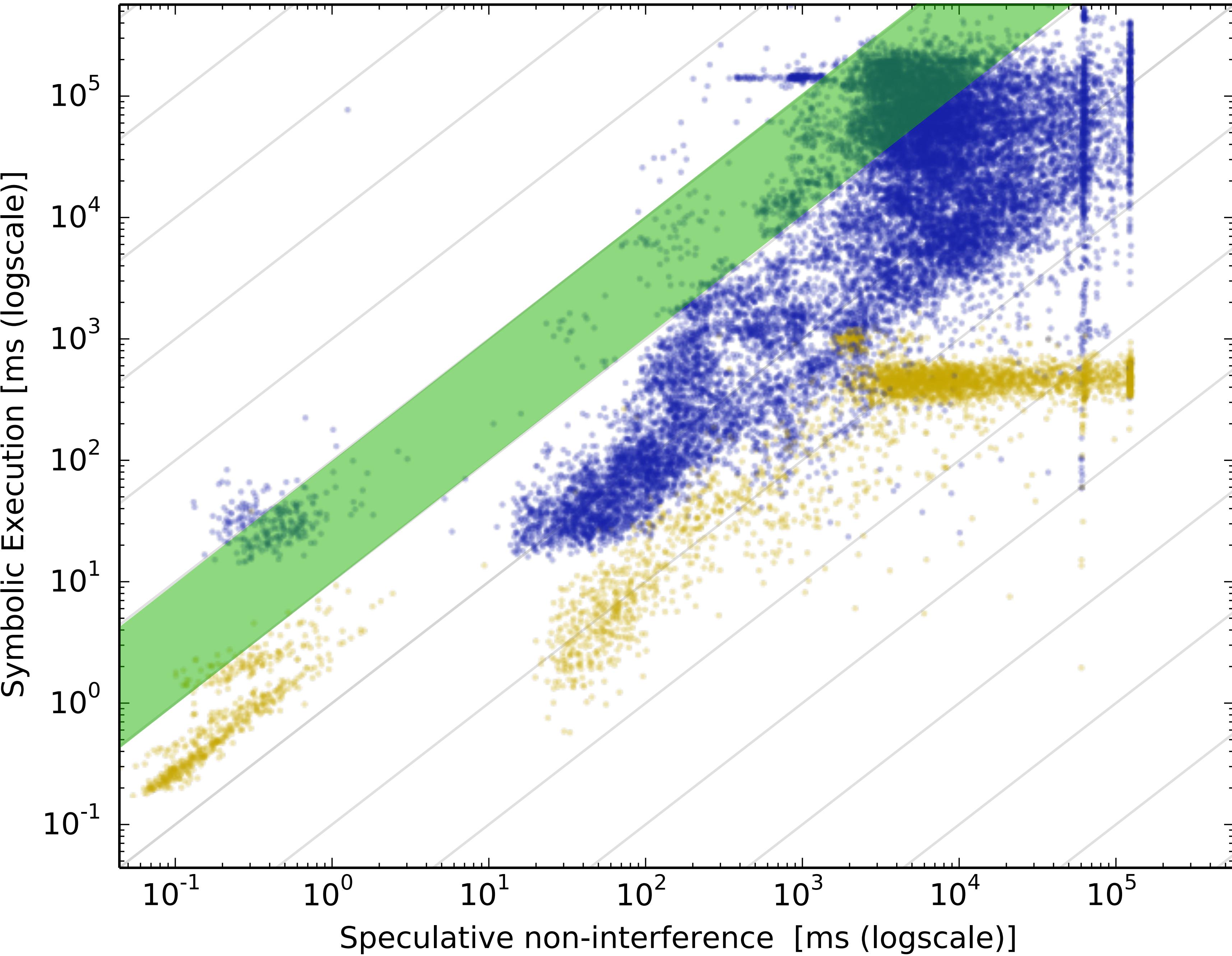
- Analyze scalability of checking SNI ***relative to*** symbolic execution
  - 24'000 symbolic paths of < 10'000 instructions (from ~ 4'000 functions)

# Results



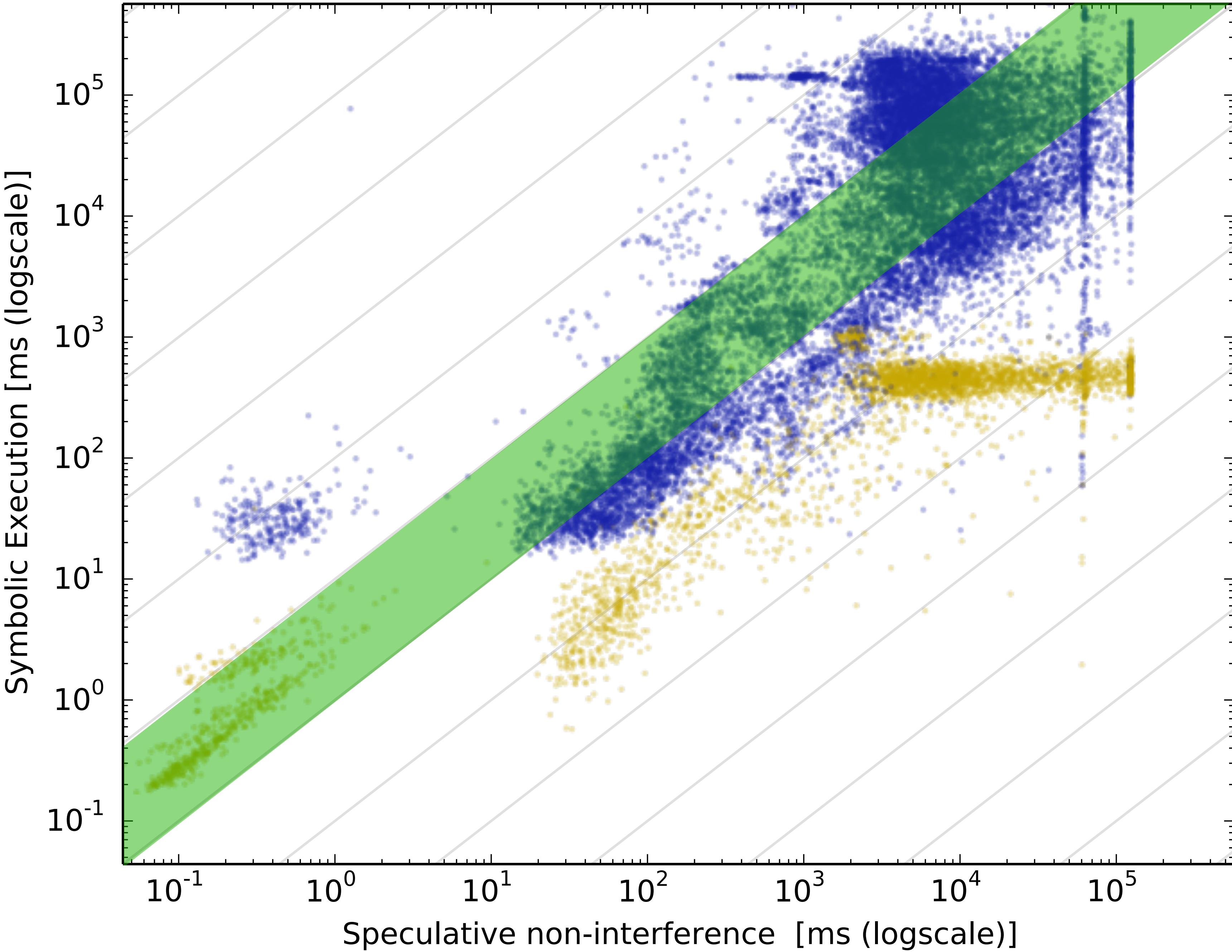
# Results

- SNI 10x-100x faster
  - 20.2% traces



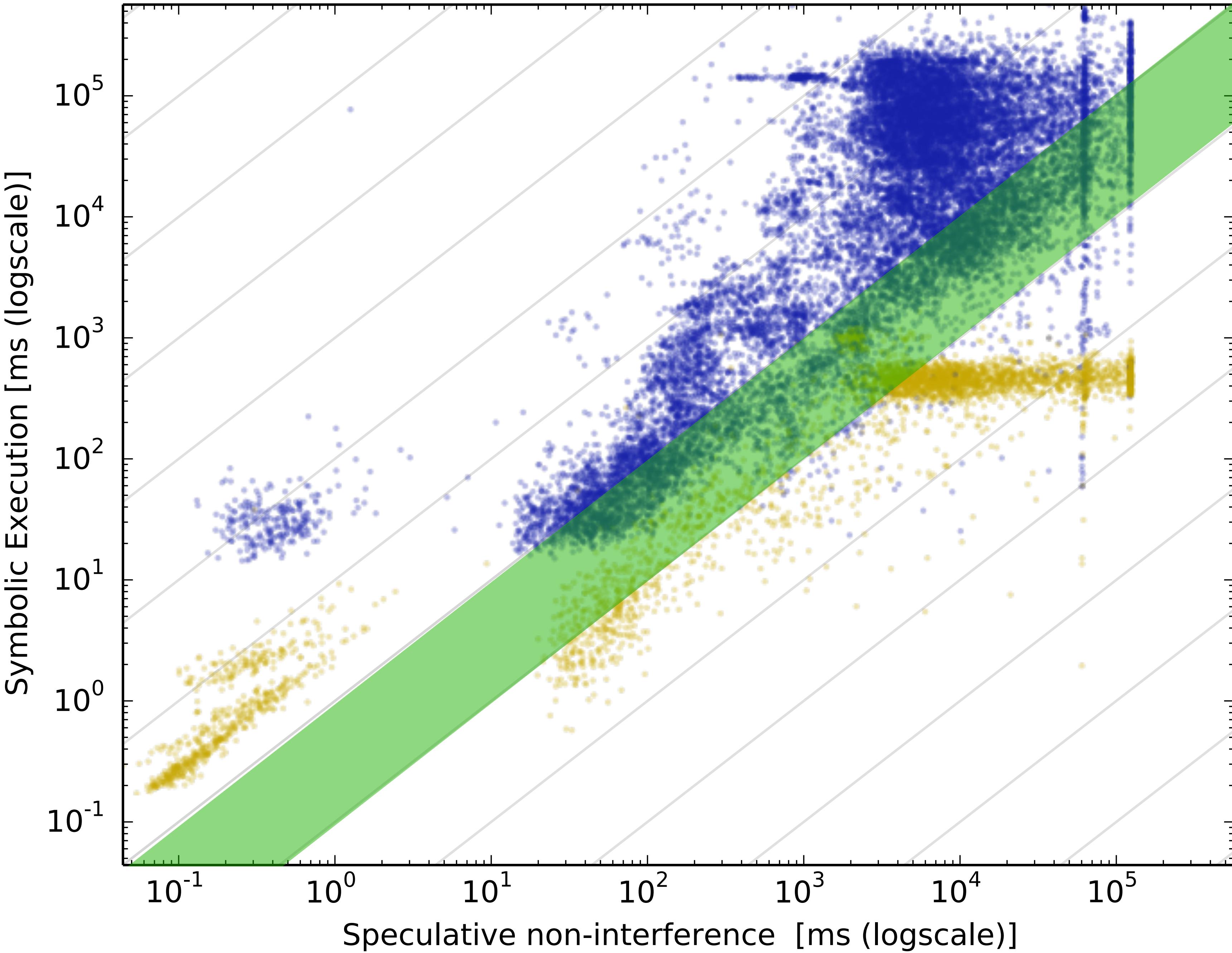
# Results

- SNI 10x-100x faster
  - 20.2% traces
- SNI  $\leq$ 10x faster
  - 41.9% traces



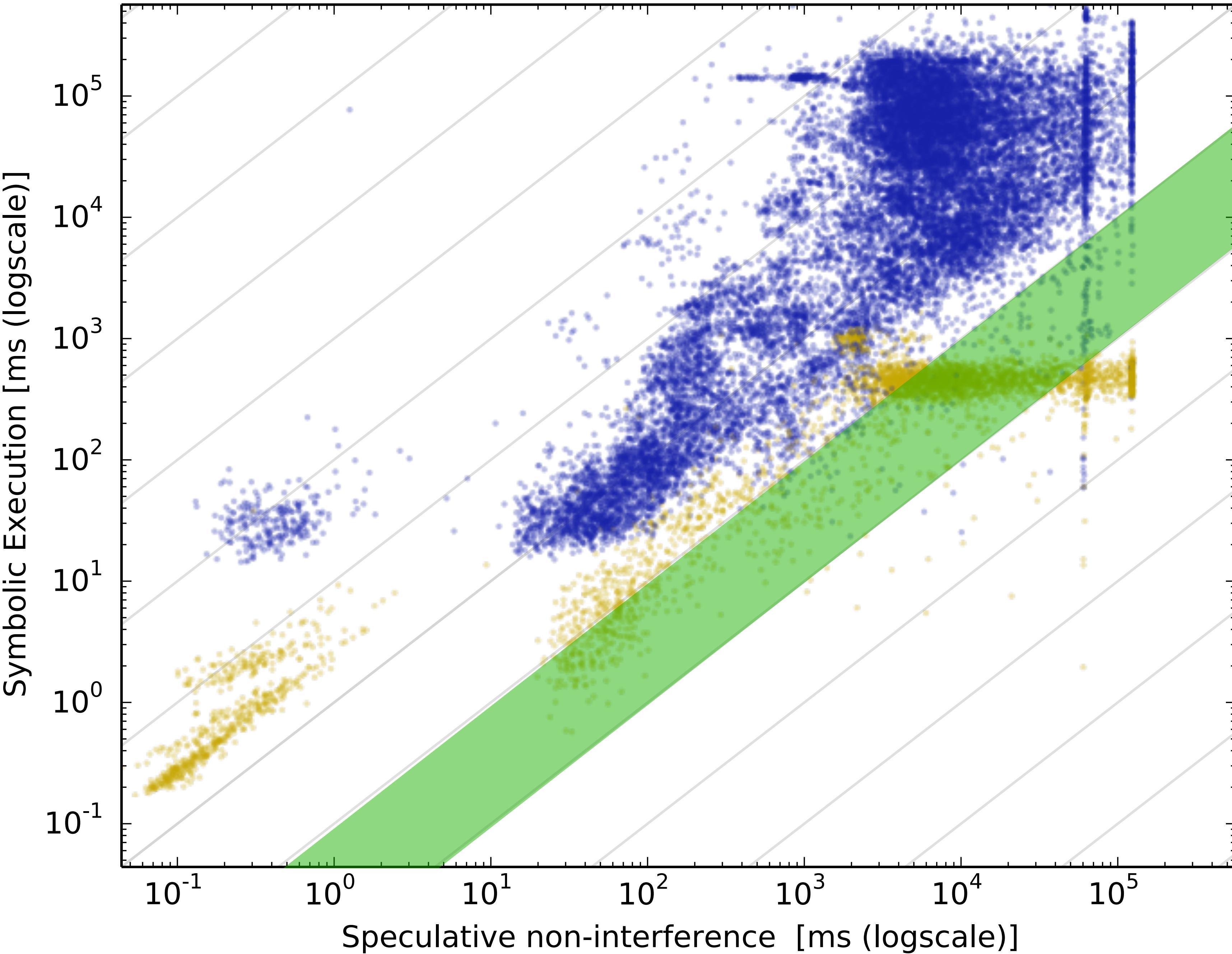
# Results

- SNI 10x-100x faster
  - 20.2% traces
- SNI  $\leq$ 10x faster
  - 41.9% traces
- SNI  $\leq$ 10x slower
  - 26.9% traces



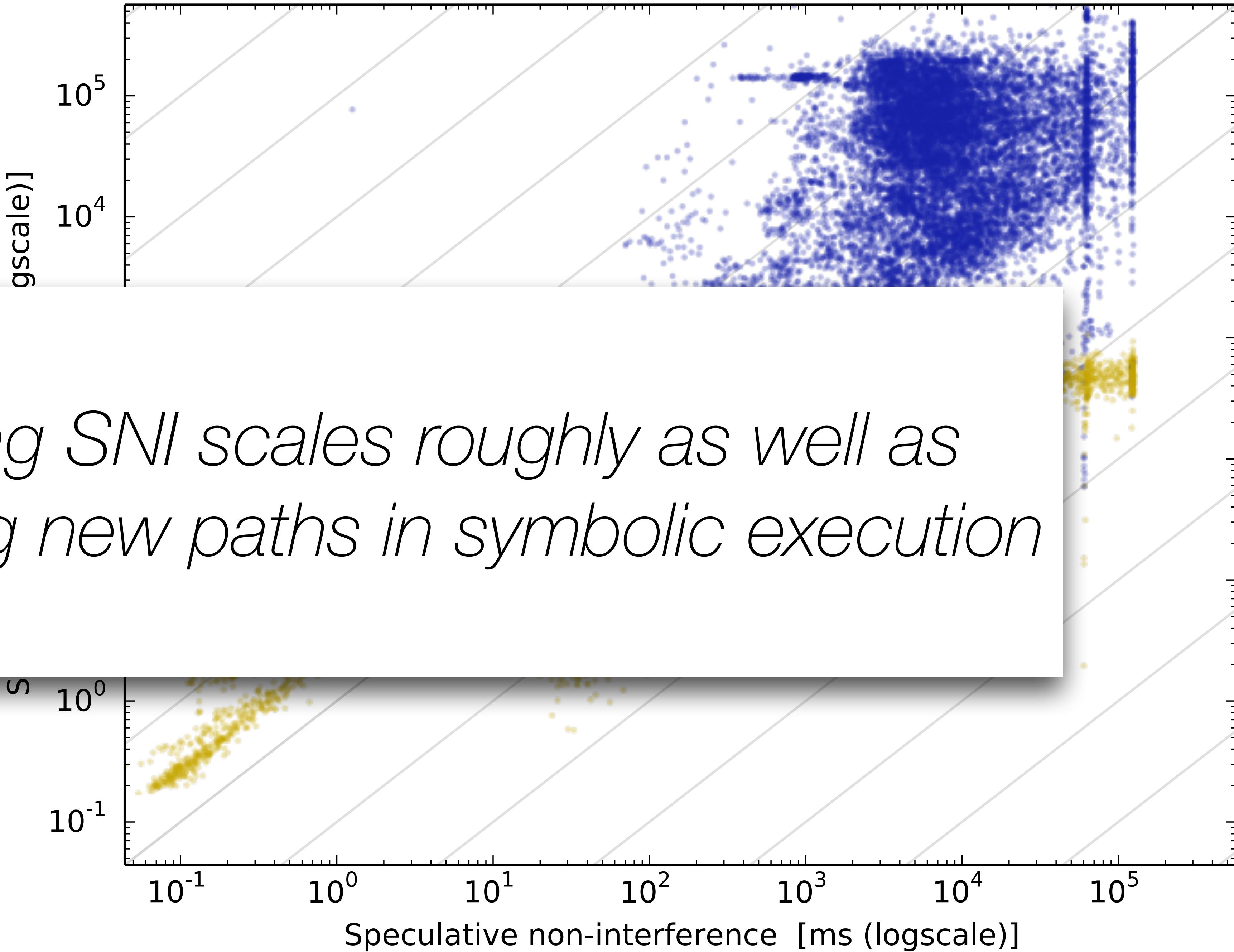
# Results

- SNI 10x-100x faster
  - 20.2% traces
- SNI  $\leq$ 10x faster
  - 41.9% traces
- SNI  $\leq$ 10x slower
  - 26.9% traces
- SNI 10x-100x slower
  - 7.9% traces



# Results

- SNI 10x-100x faster
- 20.2% traces
- SNI scales roughly as well as discovering new paths in symbolic execution
- 41.1% traces
- SNI 10x-100x slower
- 7.9% traces



# Conclusion

# Speculative non-interference

Formally!

Program  $\mathbf{P}$  is **speculatively non-interferent** for prediction oracle  $\mathbf{O}$  if

For all program states  $\mathbf{s}$  and  $\mathbf{s}'$ :

$$\begin{aligned} \mathbf{P}_{\text{non-spec}}(\mathbf{s}) &= \mathbf{P}_{\text{non-spec}}(\mathbf{s}') \\ \Rightarrow \mathbf{P}_{\text{spec}}(\mathbf{s}, \mathbf{O}) &= \mathbf{P}_{\text{spec}}(\mathbf{s}', \mathbf{O}) \end{aligned}$$

## Results

Ex.	VCC				ICC				CLANG							
	UNP		FEN 19.15		FEN 19.20		UNP		FEN		UNP		FEN		SLH	
	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02	-00	-02
01	o	o	•	•	•	•	o	o	•	•	o	o	•	•	•	•
02	o	o	•	•	•	•	o	o	•	•	o	o	•	•	•	•
03	o	o	•	o	•	•	o	o	•	•	o	o	•	•	•	•
04	o	o	o	o	•	•	o	o	•	•	o	o	•	•	•	•
05	o	o	•	o	•	•	o	o	•	•	o	o	•	•	•	•
06	o	o	o	o	o	o	o	o	•	•	o	o	•	•	•	•
07	o	o	o	o	o	o	o	o	•	•	o	o	•	•	•	•
08	o	•	o	•	o	•	o	•	•	•	o	•	•	•	•	•
09	o	o	o	o	o	o	o	o	•	•	o	o	•	•	•	•
10	o	o	o	o	o	o	o	o	•	•	o	o	•	•	•	o
11	o	o	o	o	o	o	o	o	•	•	o	o	•	•	•	•
12	o	o	o	o	•	•	o	o	•	•	o	o	•	•	•	•
13	o	o	o	o	o	o	o	o	•	•	o	o	•	•	•	•
14	o	o	o	o	•	•	o	o	•	•	o	o	•	•	•	•
15	o	o	o	o	o	o	o	o	•	•	o	o	•	•	•	•

# Spectector



```

mov    rax, A_size
mov    rcx, x
cmp    rcx, rax
jae    END
L1:   mov    rax, A[rcx]
      mov    rax, B[rax]

```

x64 to µASM

L1:   END:

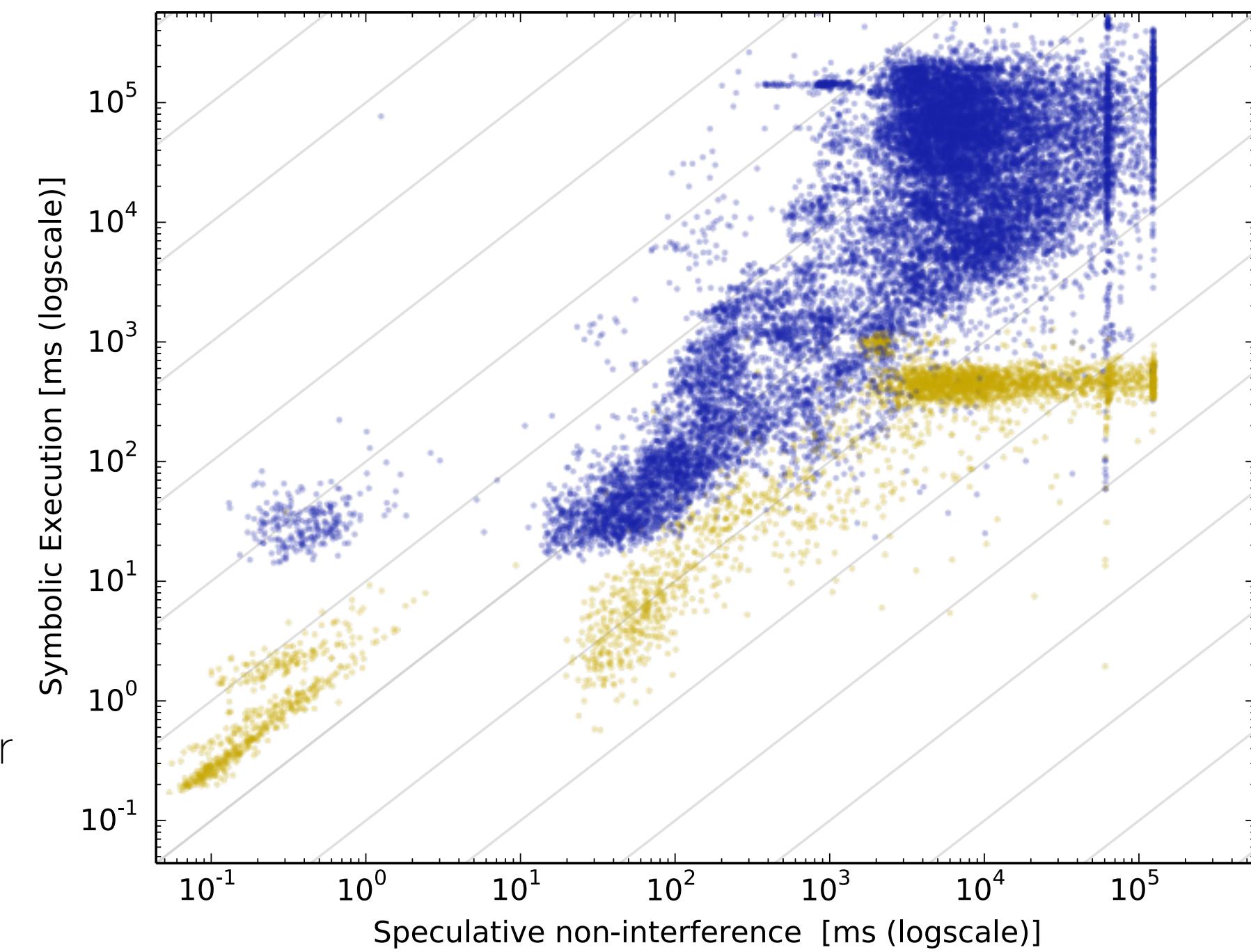


Check for speculative leaks

Symbolic  
execution

## Results

- SNI 10x-100x faster
- 20.2% traces
- SNI  $\leq 10x$  faster
- 41.9% traces
- SNI  $\leq 10x$  slower
- 26.9% traces
- SNI 10x-100x slower
- 7.9% traces



# Speculative non-interference

Formally!

Program **P** is **speculatively non-interferent** for prediction oracle **O** if

For all  
 $P_{\text{non}}$



## Results

Ex.	VCC			
	UNP	-02	-00	FEN 19.15
01	o	o	•	•
02	o	o	•	•
03	o	o	•	o
04	o	o	o	o
05	o	o	•	o
06	o	o	o	o
07	o	o	o	o
08	o	•	o	•
09	o	o	o	o
10	o	o	o	o
11	o	o	o	o
12	o	o	o	•
13	o	o	o	o
14	o	o	o	•
15	o	o	o	o

# Spectector

```
mov    rax, A_size
mov    rcx, x
cmp    rcx, rax
jae    END
L1:   mov    rax, A[rcx]
```

x64 to µASM

L1:  
END:

```
rax <- A_size
rcx <- x
jmp rcx>=rax, END
load rax, A + rcx
load rax, B + rax
```

# Spectector



<https://spectector.github.io>

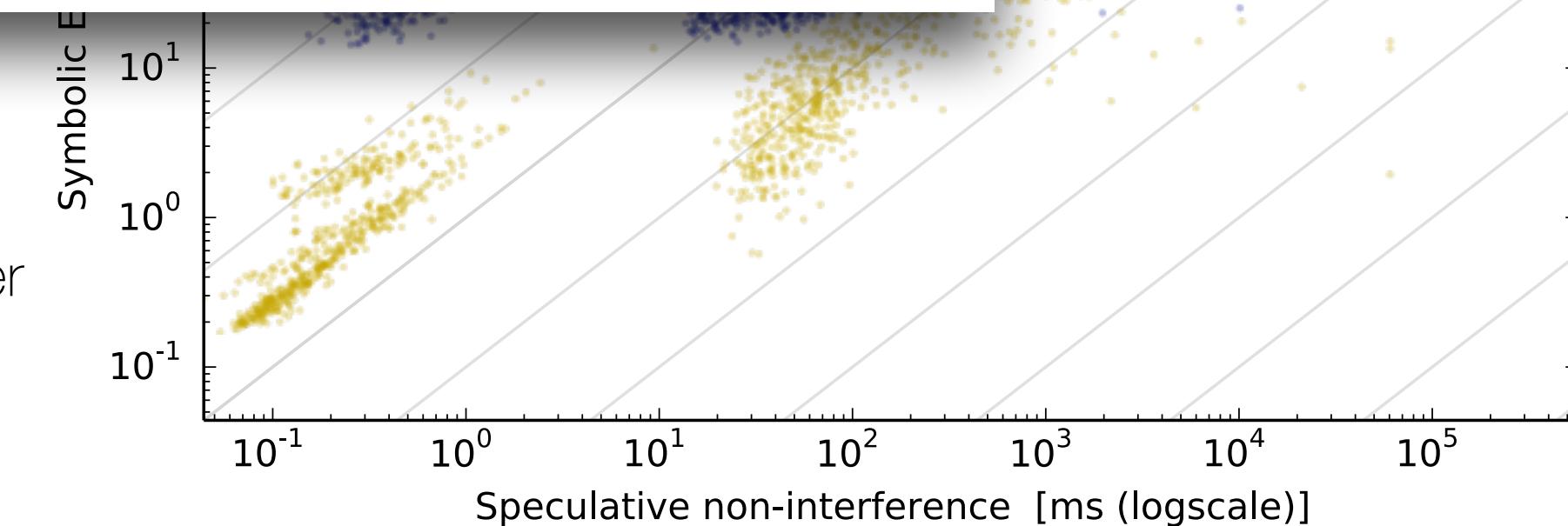


@marco.guarnieri@imdea.org



@MarcoGuarnier1

- SNI  $\leq 10x$  slower
- 26.9% traces
- SNI 10x-100x slower
- 7.9% traces



Symbolic execution

