

Seguridad Informatica - Fall 2024

Module IV: Physical security

Lecture 5

Reasoning about speculative execution attacks

Marco Guarnieri
IMDEA Software Institute

Module IV

- Lecture 1 – Introduction
- Lecture 2 – Cache-based side channel attacks
- Lecture 3 – Speculative execution attacks
- Lecture 4 – Non-interference
- Lecture 5 – Automated detection of speculative leaks

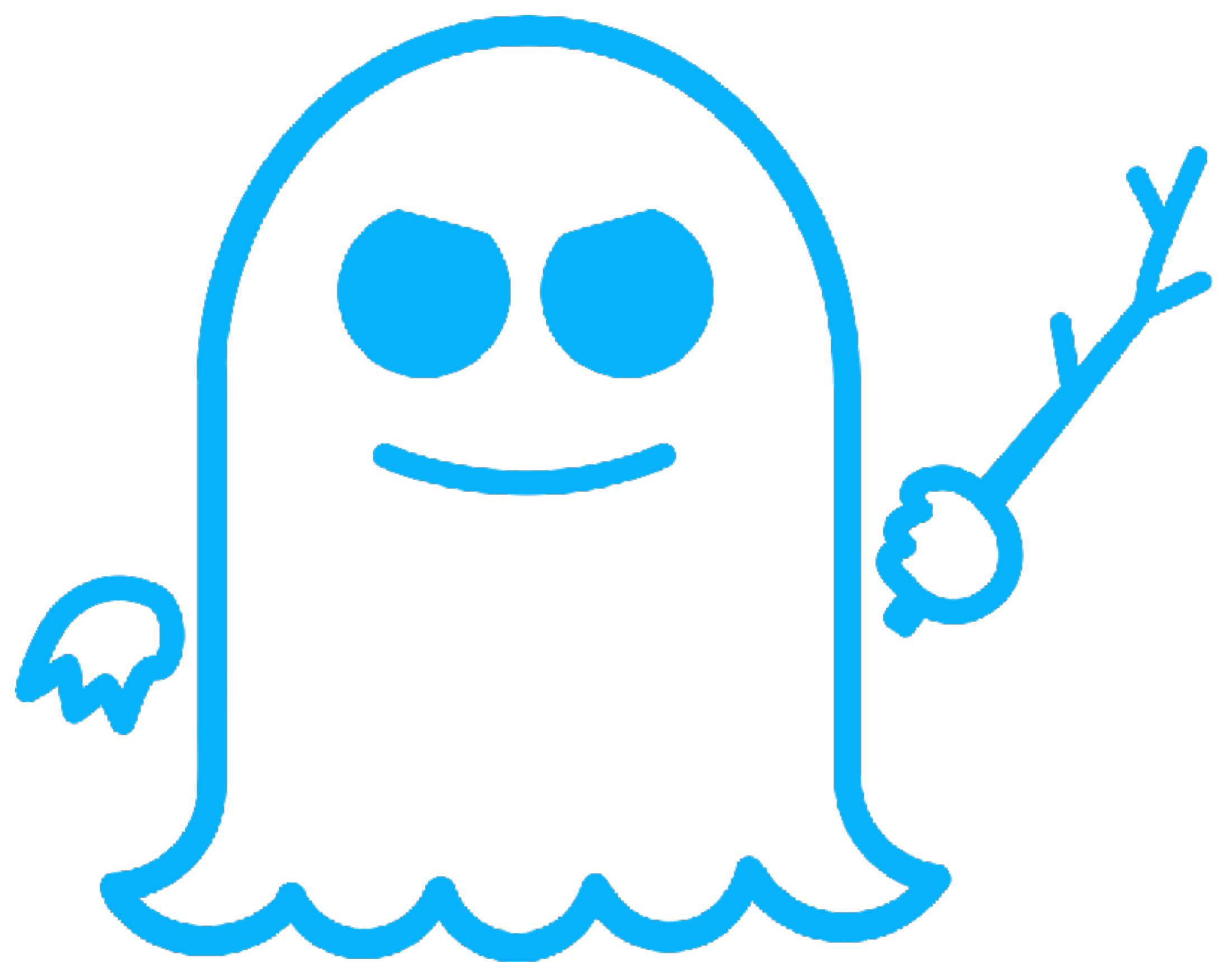
Module IV

- Lecture 1 – Introduction
- Lecture 2 – Cache-based side channel attacks
- Lecture 3 – Speculative execution attacks
- Lecture 4 – Non-interference
- Lecture 5 – Automated detection of speculative leaks

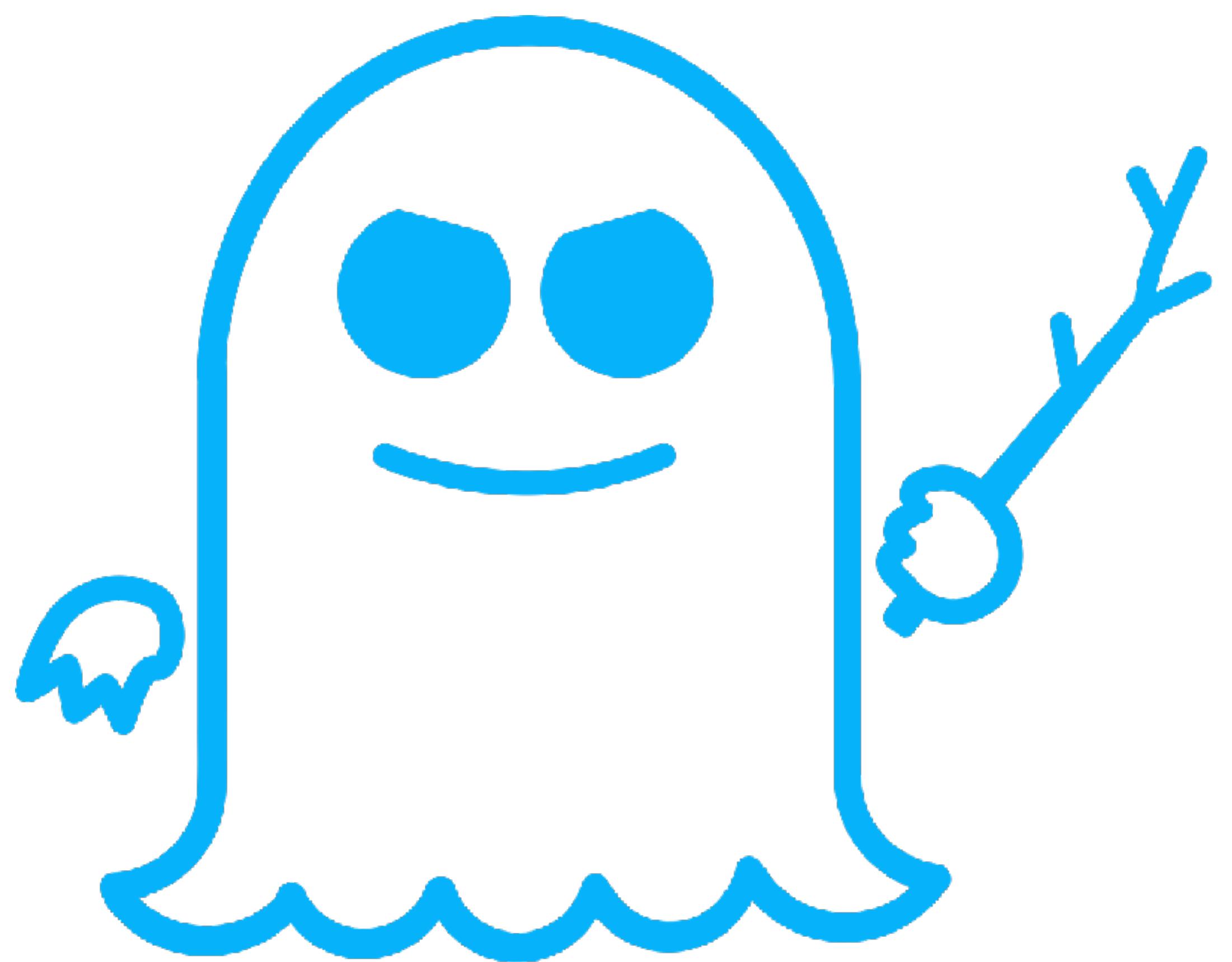
Recommended reading

Spectector: Principled detection of speculative information flows by Marco Guarnieri, José F. Morales, Andrés Sánchez, Boris Köpf, Jan Reineke

Available at <https://spectector.github.io/papers/spectector.pdf>

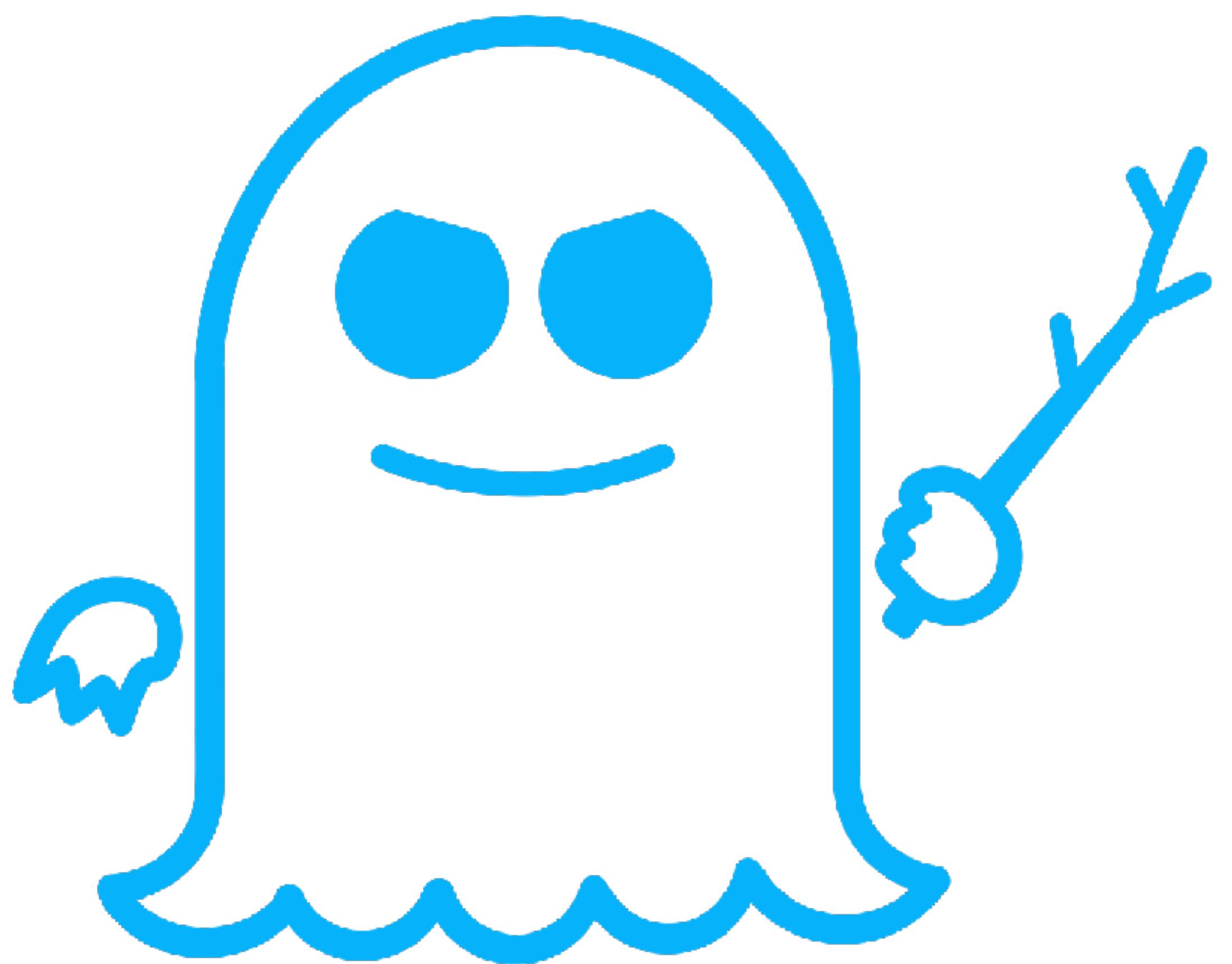


SPECTRE



SPECTRE

Exploits ***speculative execution***



SPECTRE

Exploits ***speculative execution***

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

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!

Contributions

Contributions

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

Contributions

1. *Semantic notion* of **security**
against **speculative execution attacks**
2. Analysis to **detect vulnerability** or **prove security**

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)
```

```
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

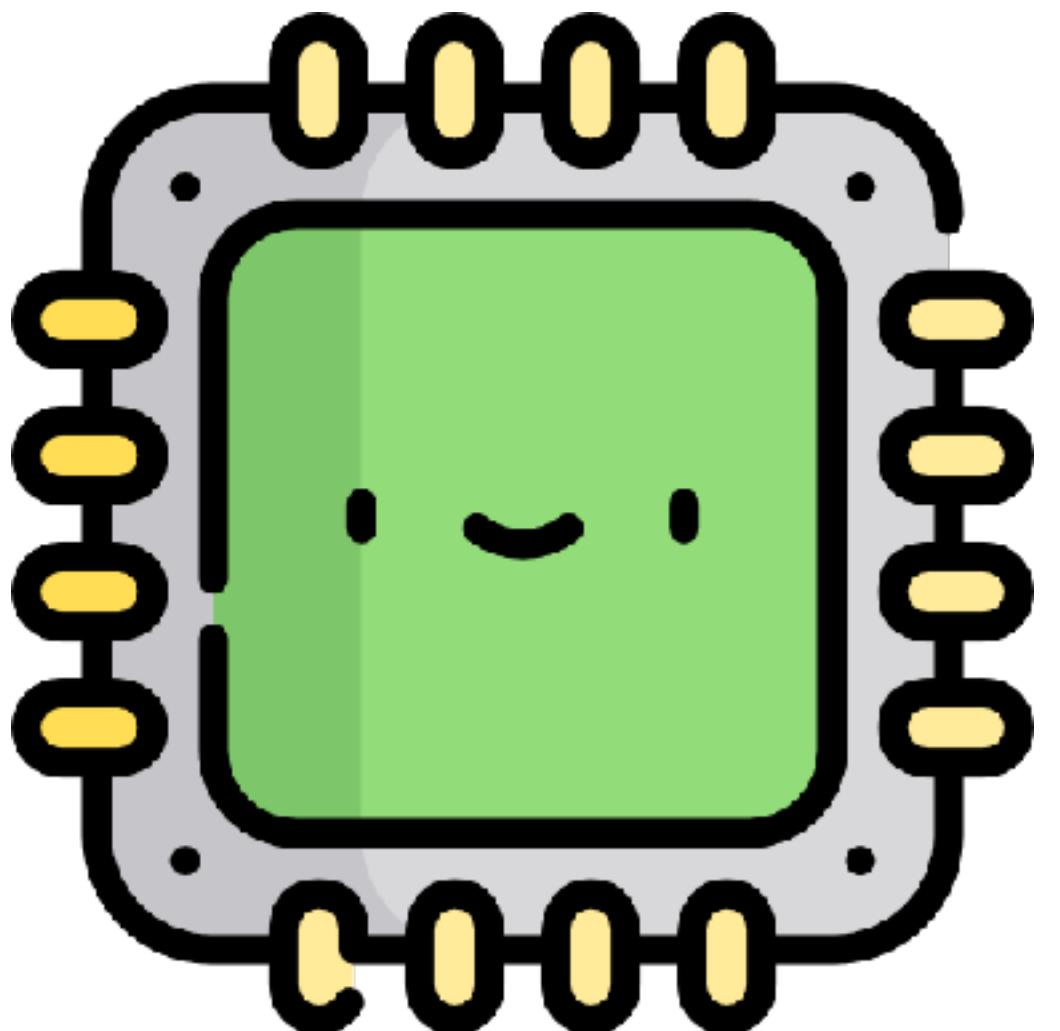
Branch predictor

Spectre V1

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

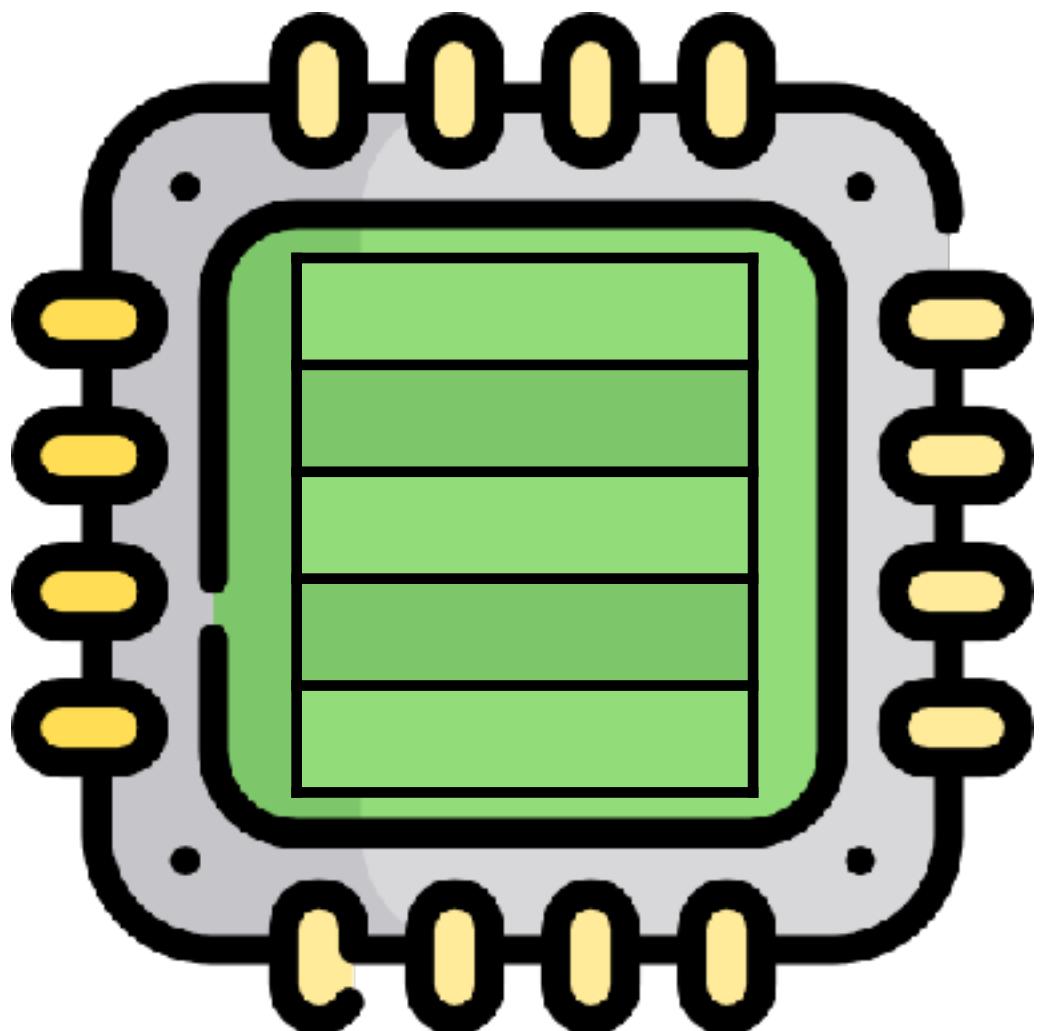
Spectre V1

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



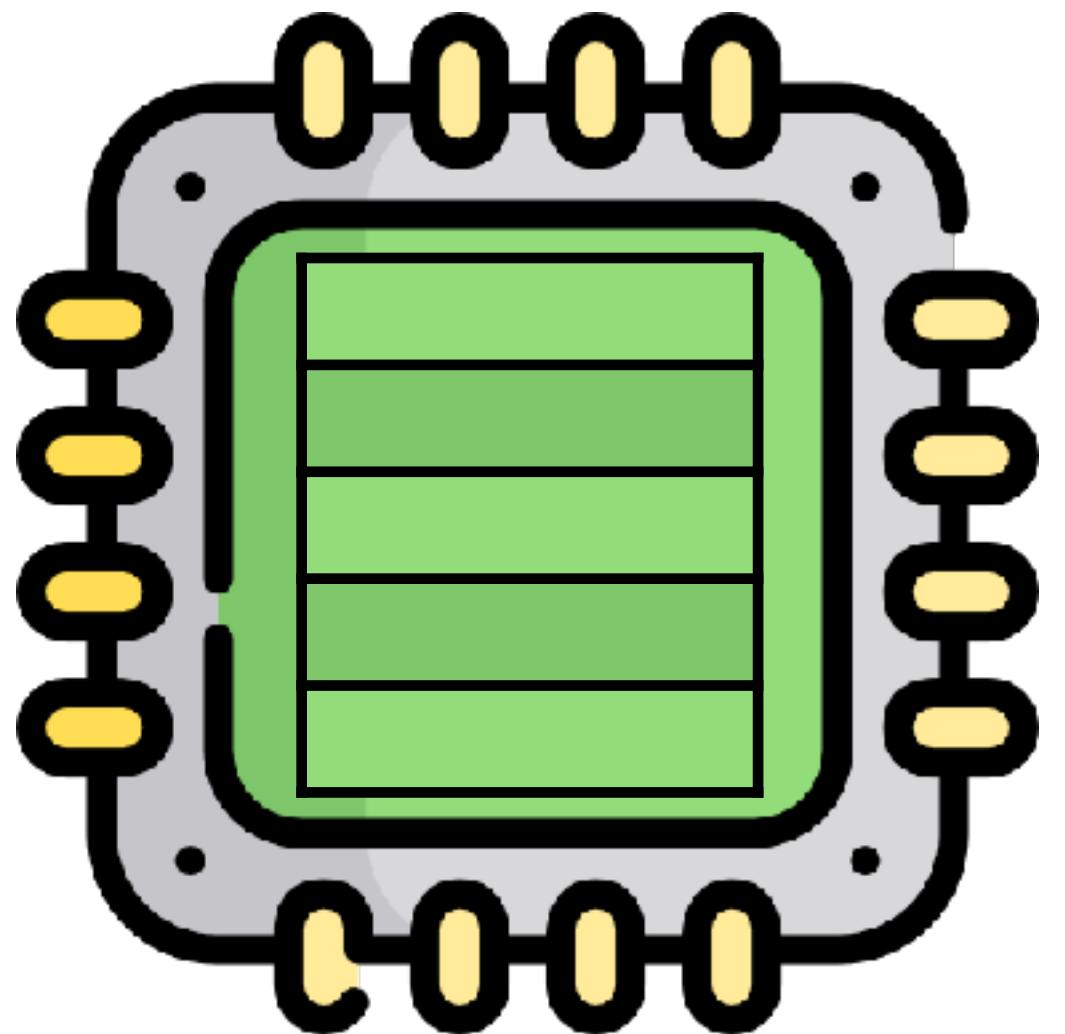
Spectre V1

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

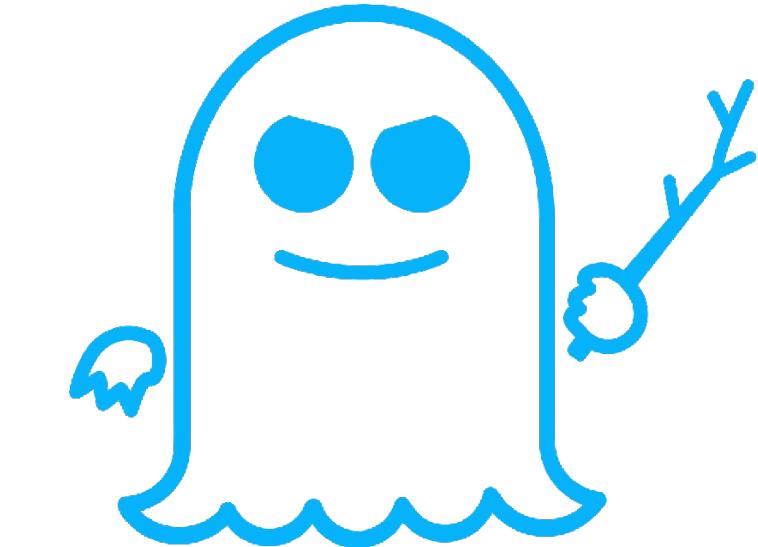


Spectre V1

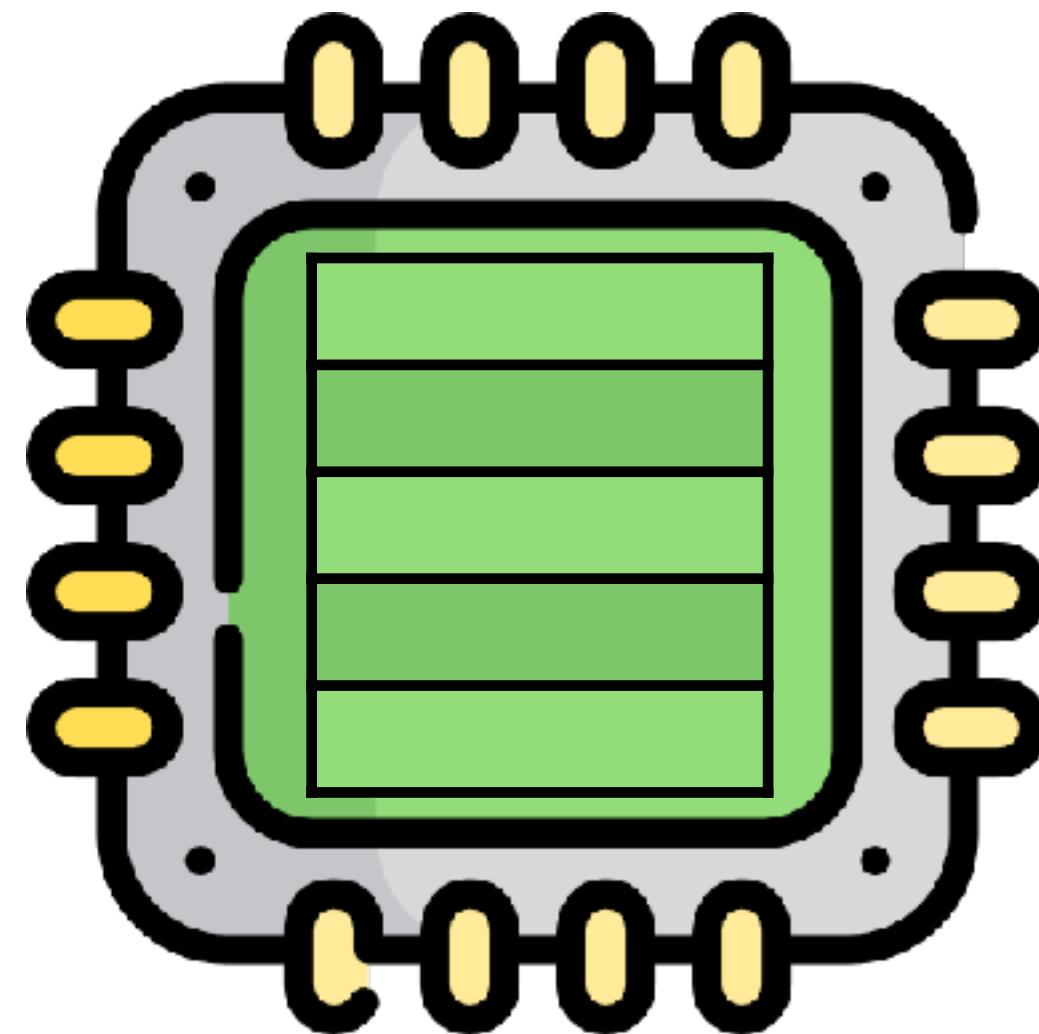
```
void f(int x)
  if (x < A_size)
    y = B[A[x]]
```



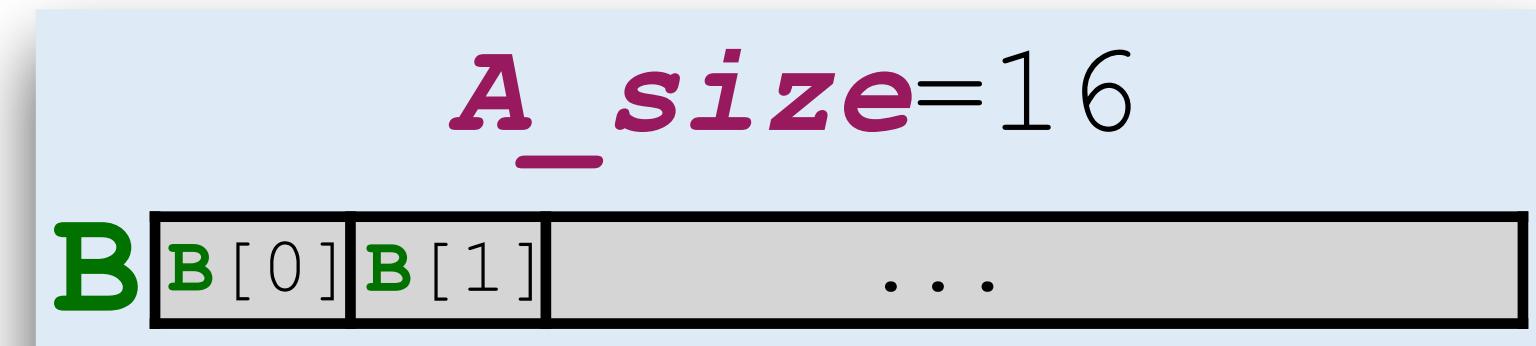
Spectre V1



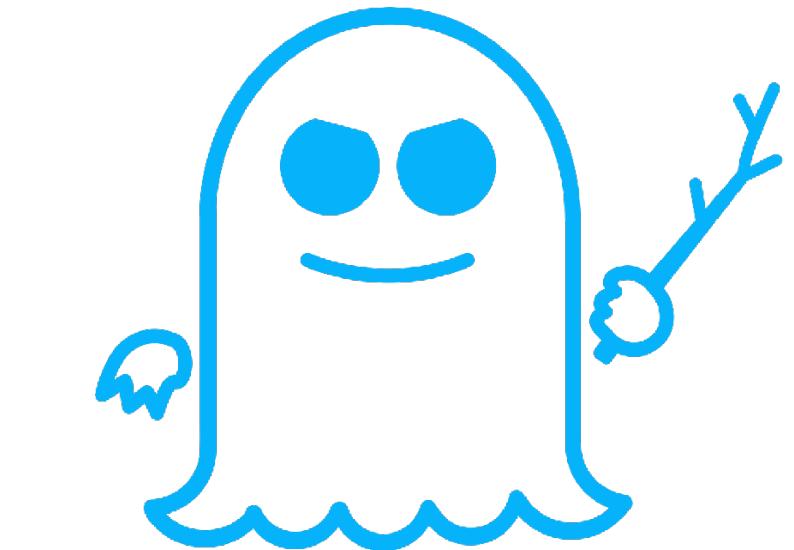
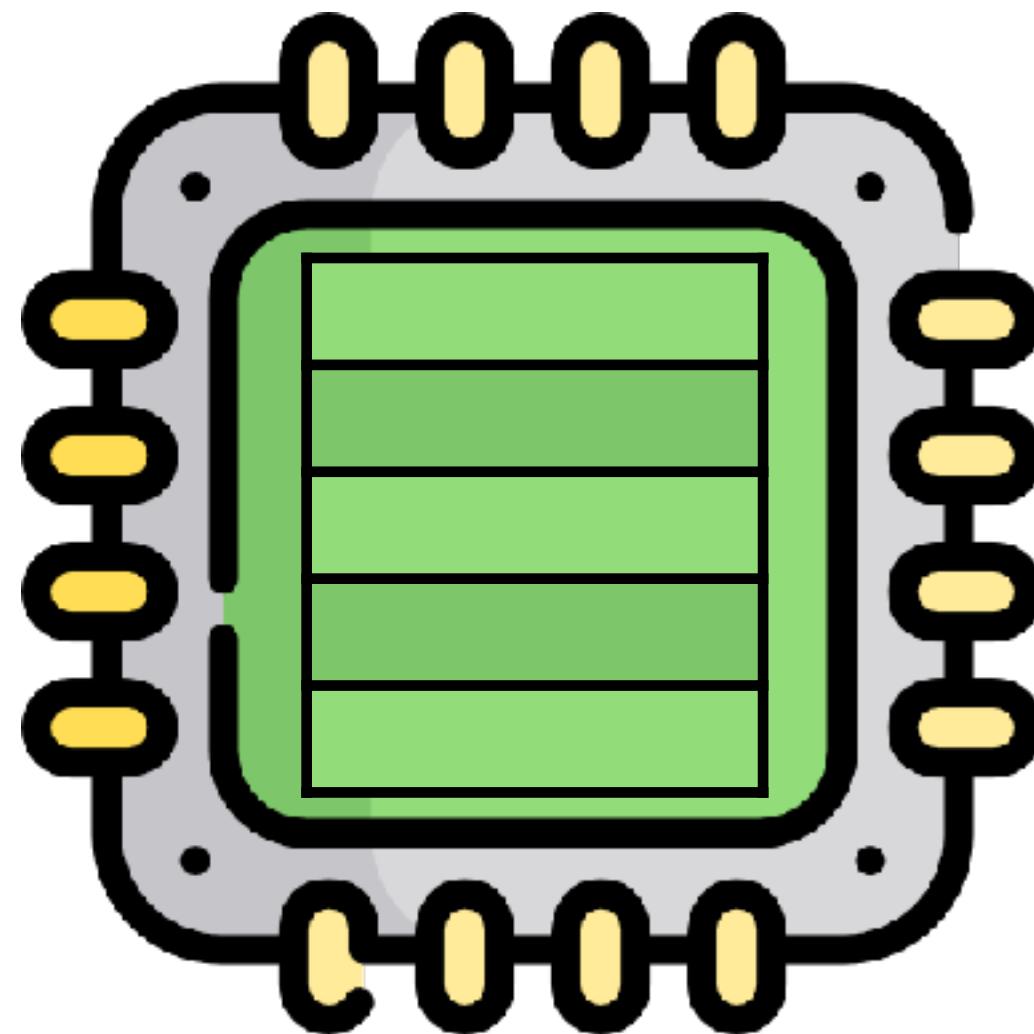
```
void f(int x)
  if (x < A_size)
    y = B[A[x]]
```



Spectre V1

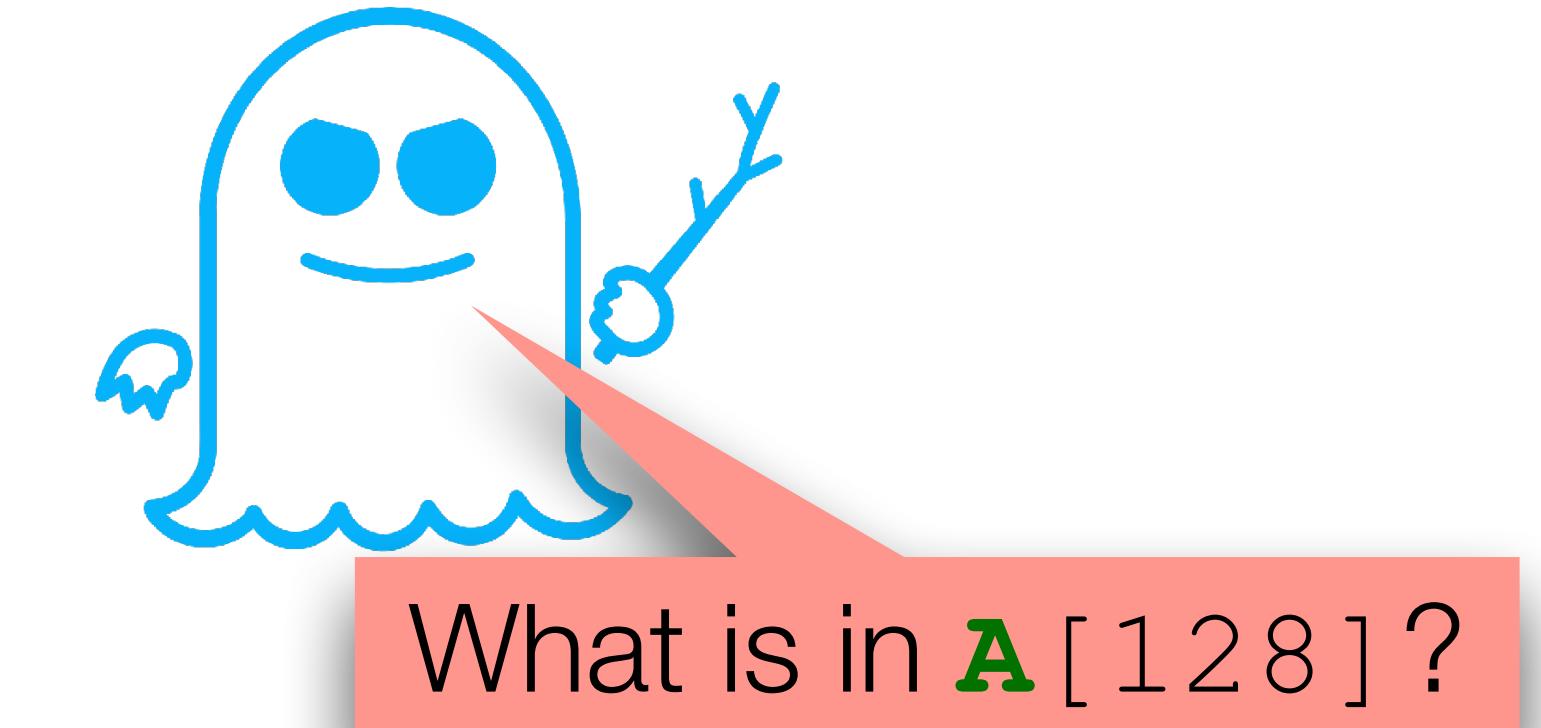
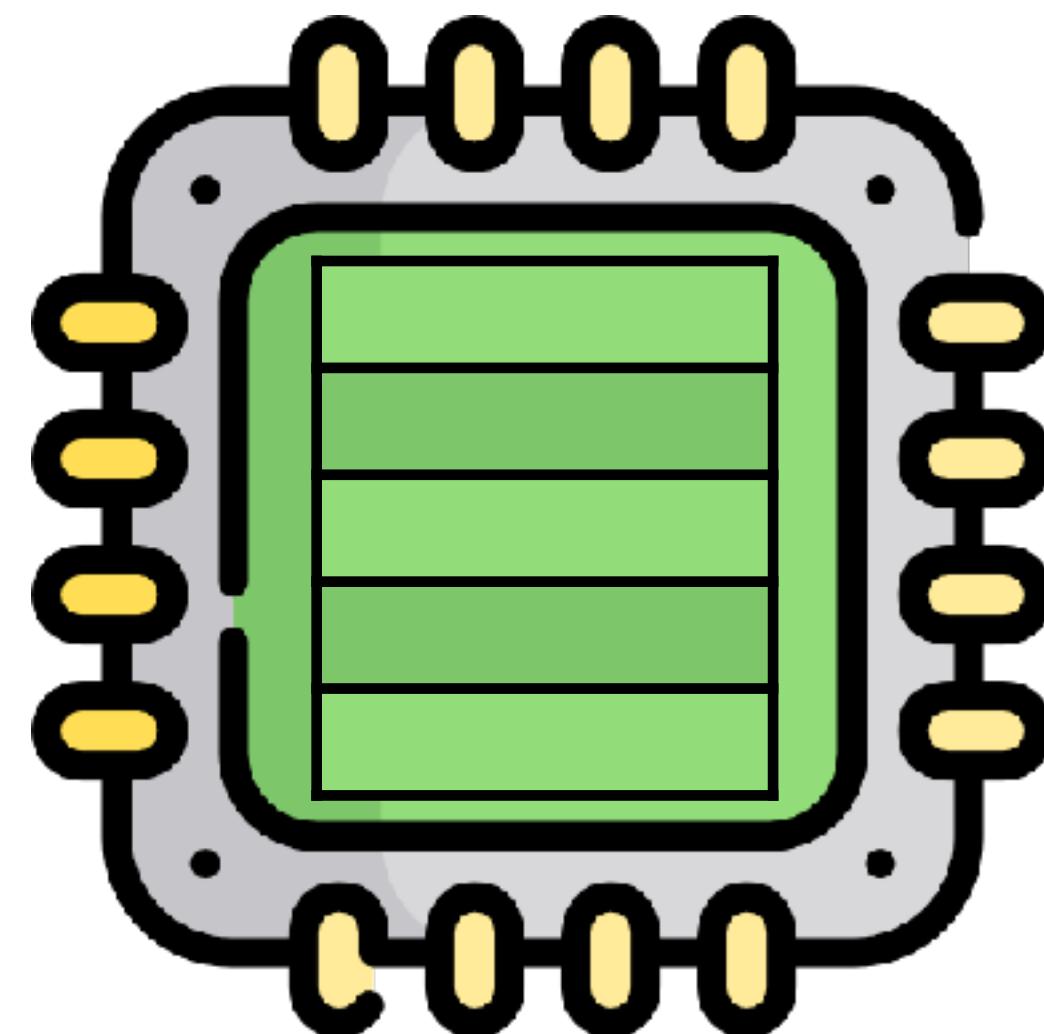


```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



Spectre V1

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



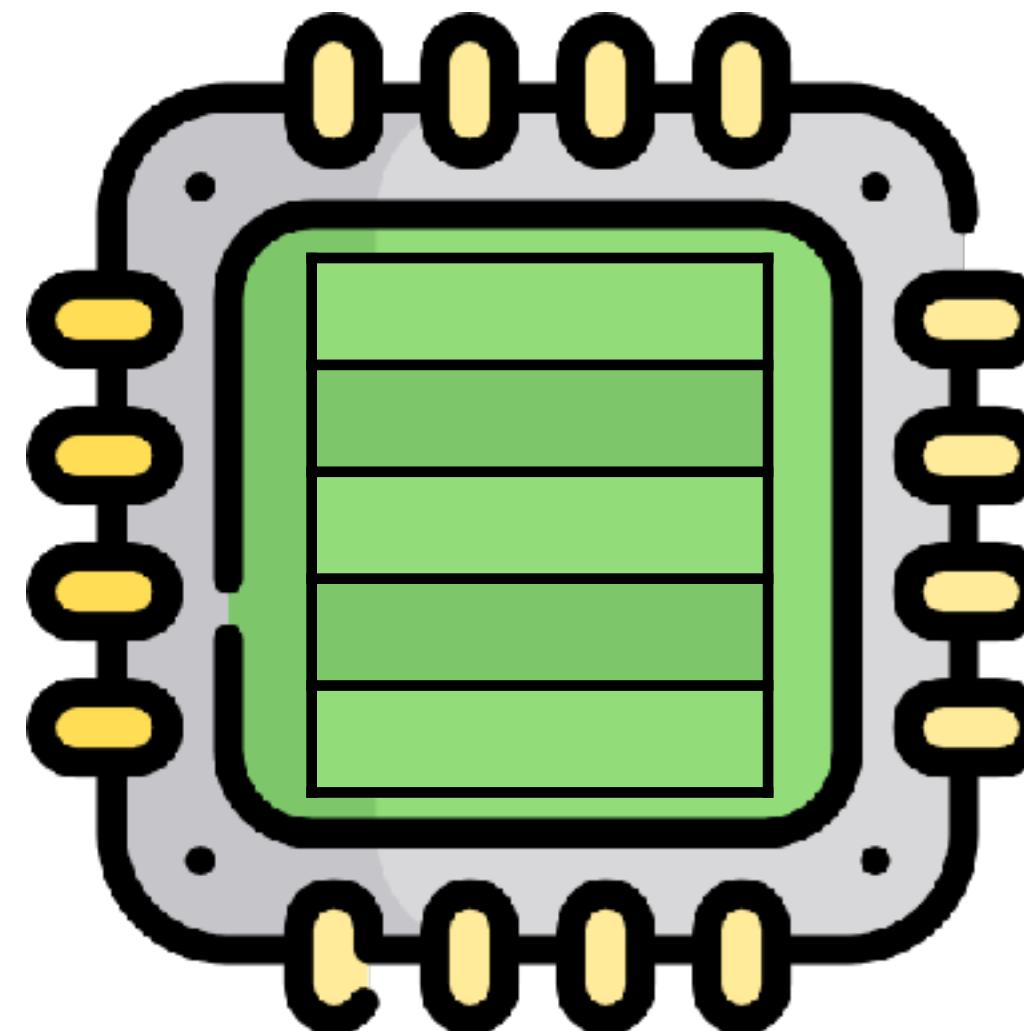
Spectre V1

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



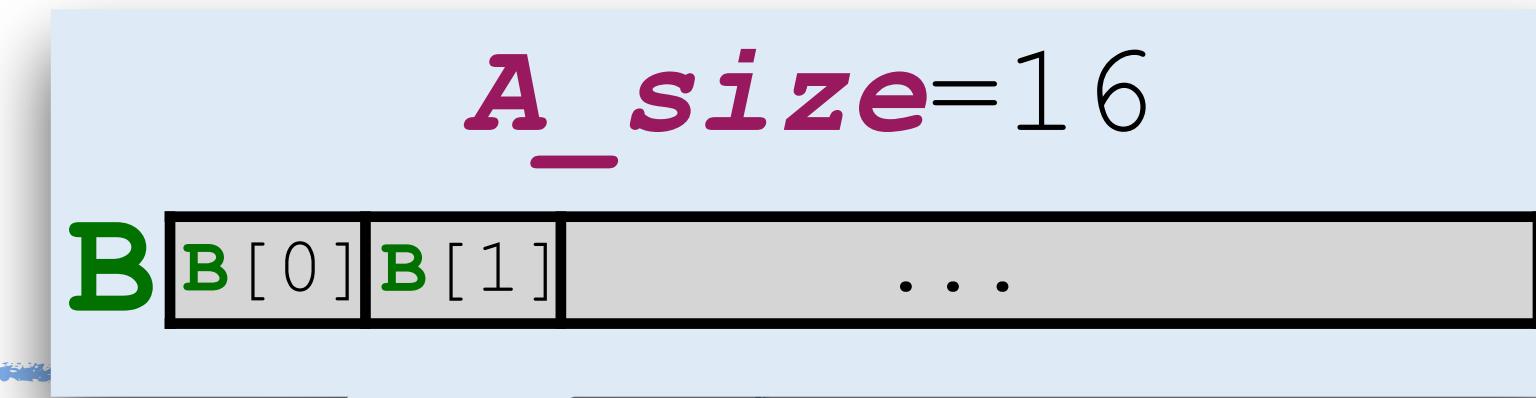
What is in A[128]?

1) Training



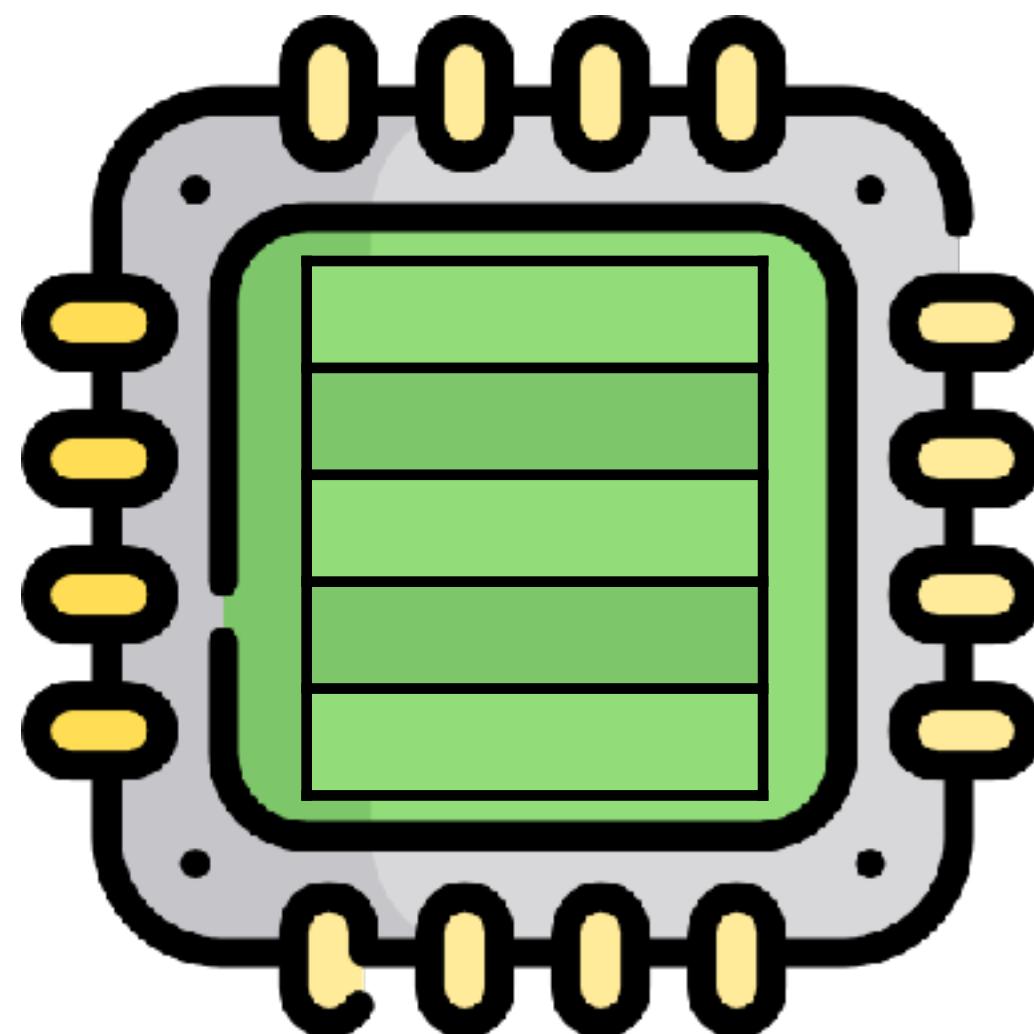
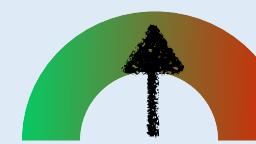
Spectre V1

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



What is in $A[128]$?

1) Training

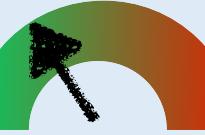


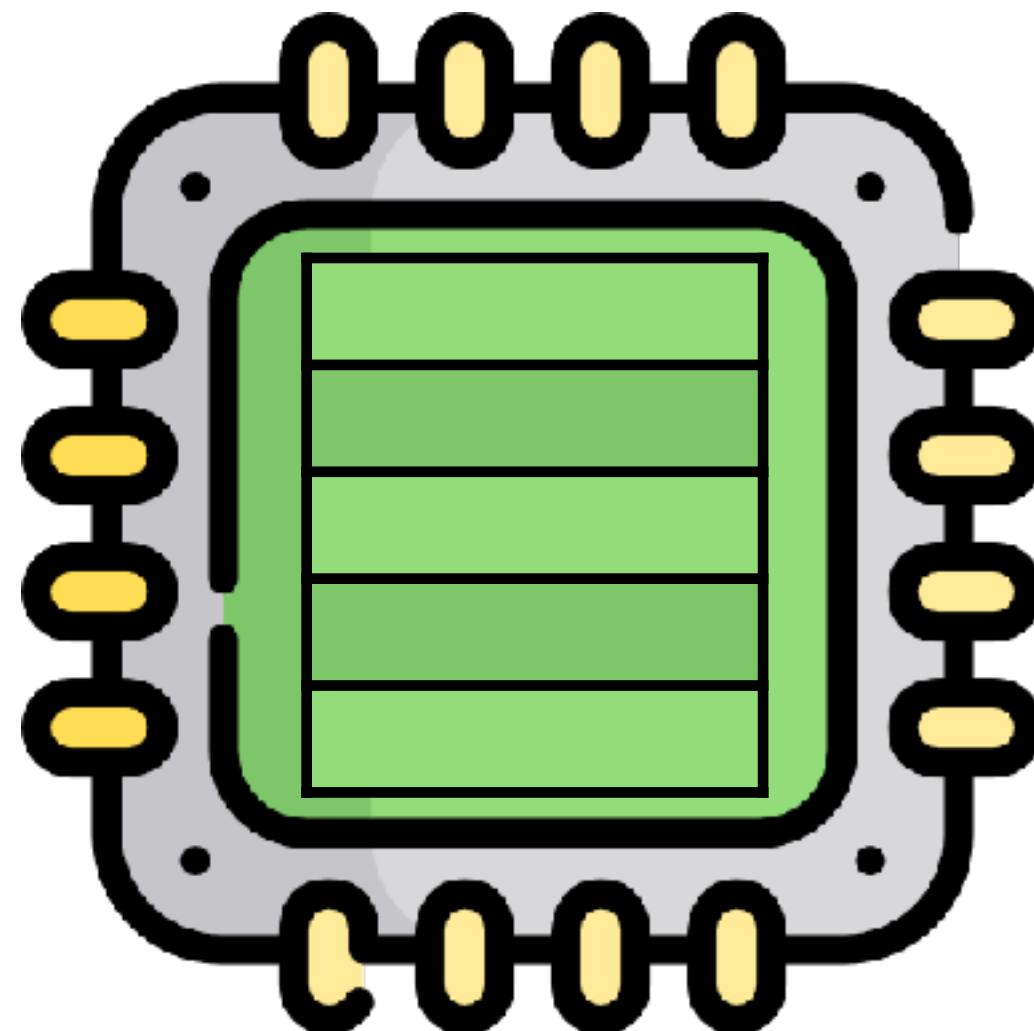
Spectre V1

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



What is in $A[128]$?

1) Training  $f(0);$

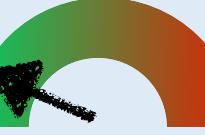


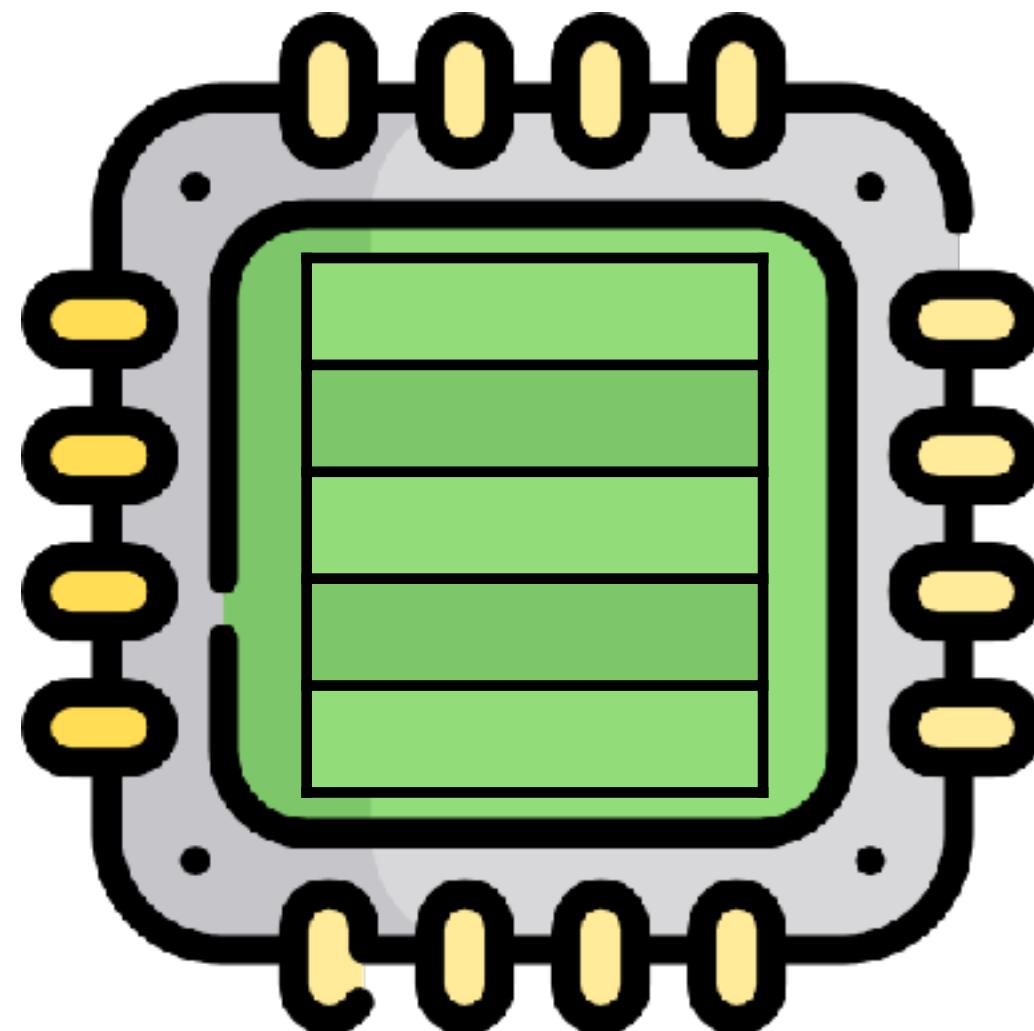
Spectre V1

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



What is in A[128]?

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



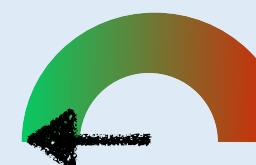
Spectre V1

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

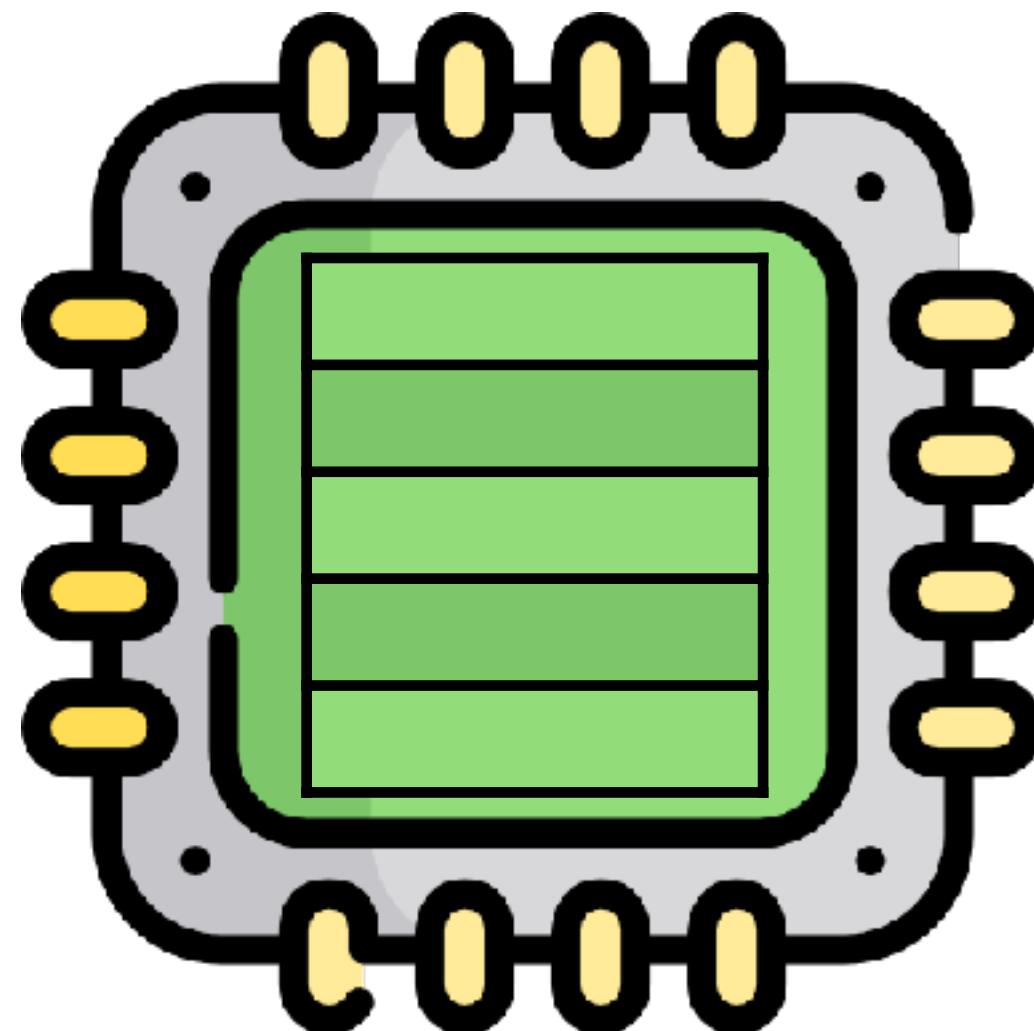


What is in $A[128]$?

1) Training



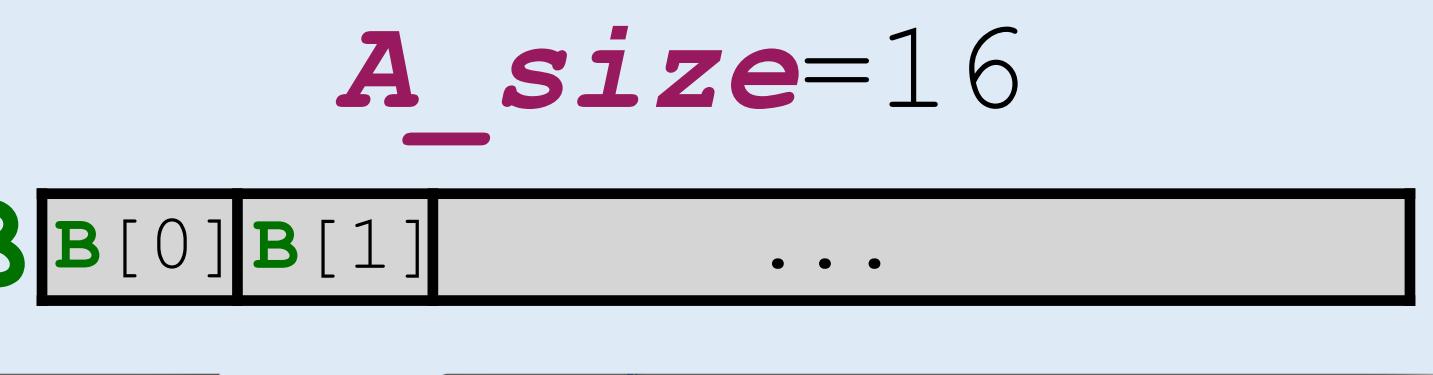
f(0); f(1); f(2); ...



Spectre V1

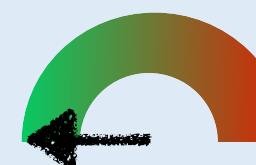


```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



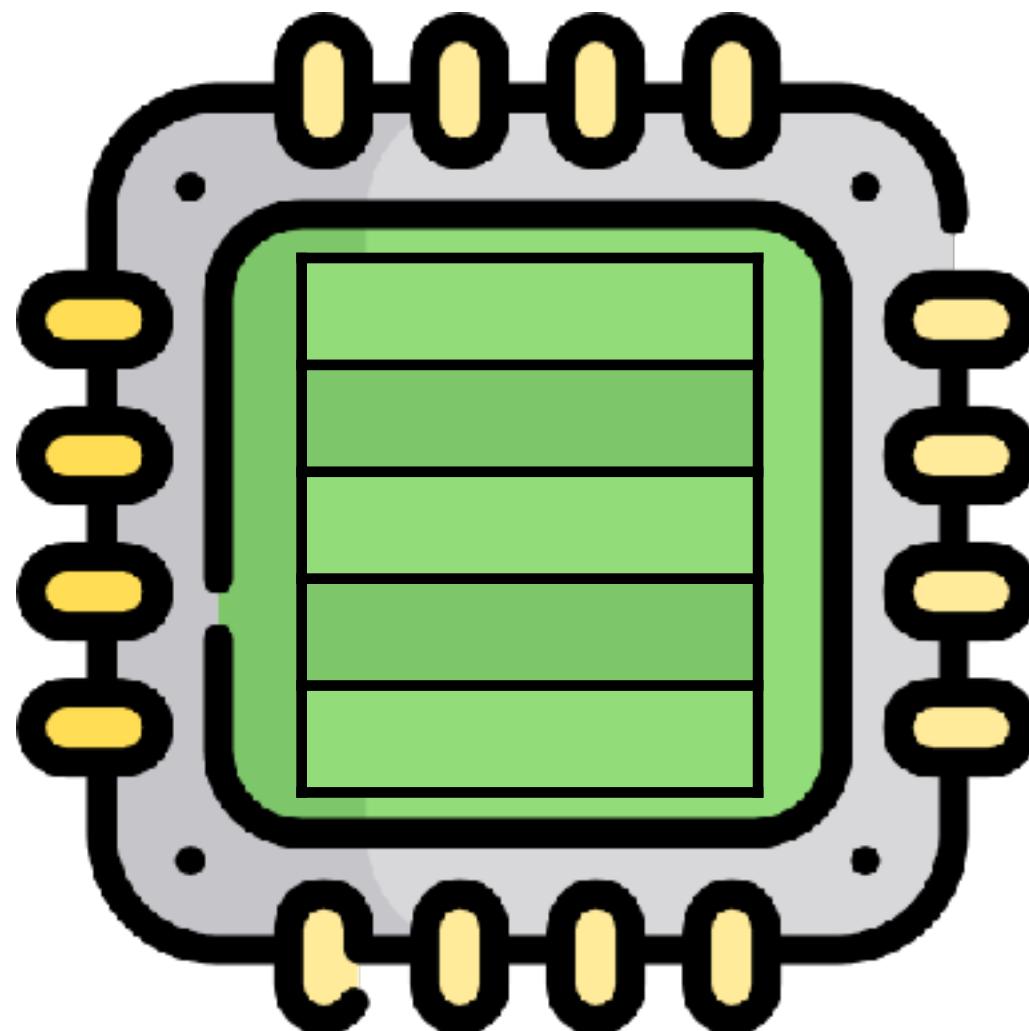
What is in $A[128]$?

1) Training



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

2) Prepare cache



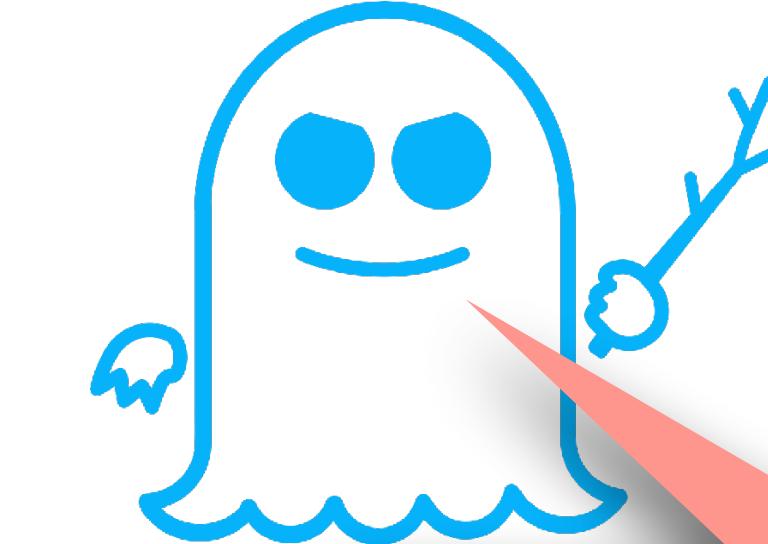
Spectre V1



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

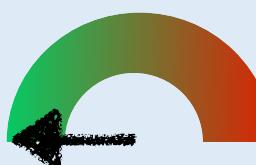
A_size=16

B[B[0] B[1] ...]



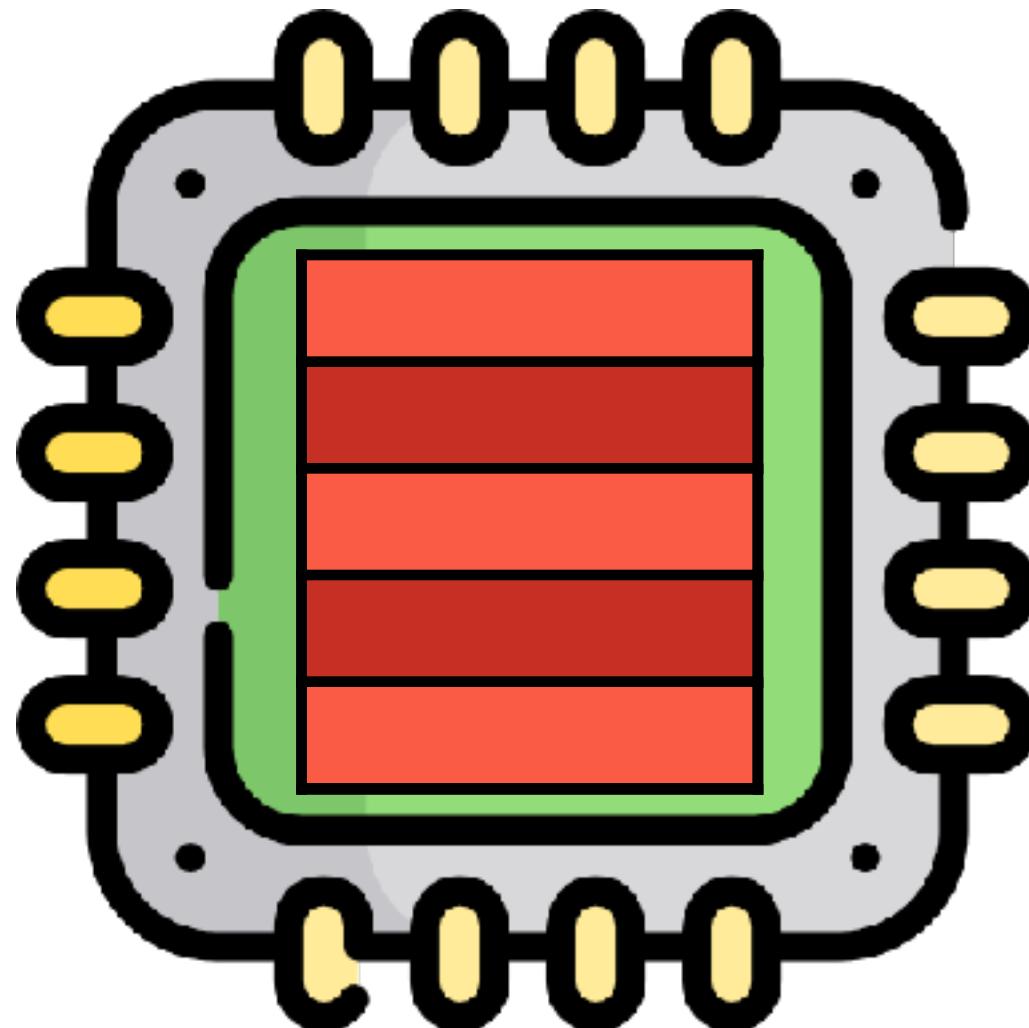
What is in **A**[128]?

1) Training



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

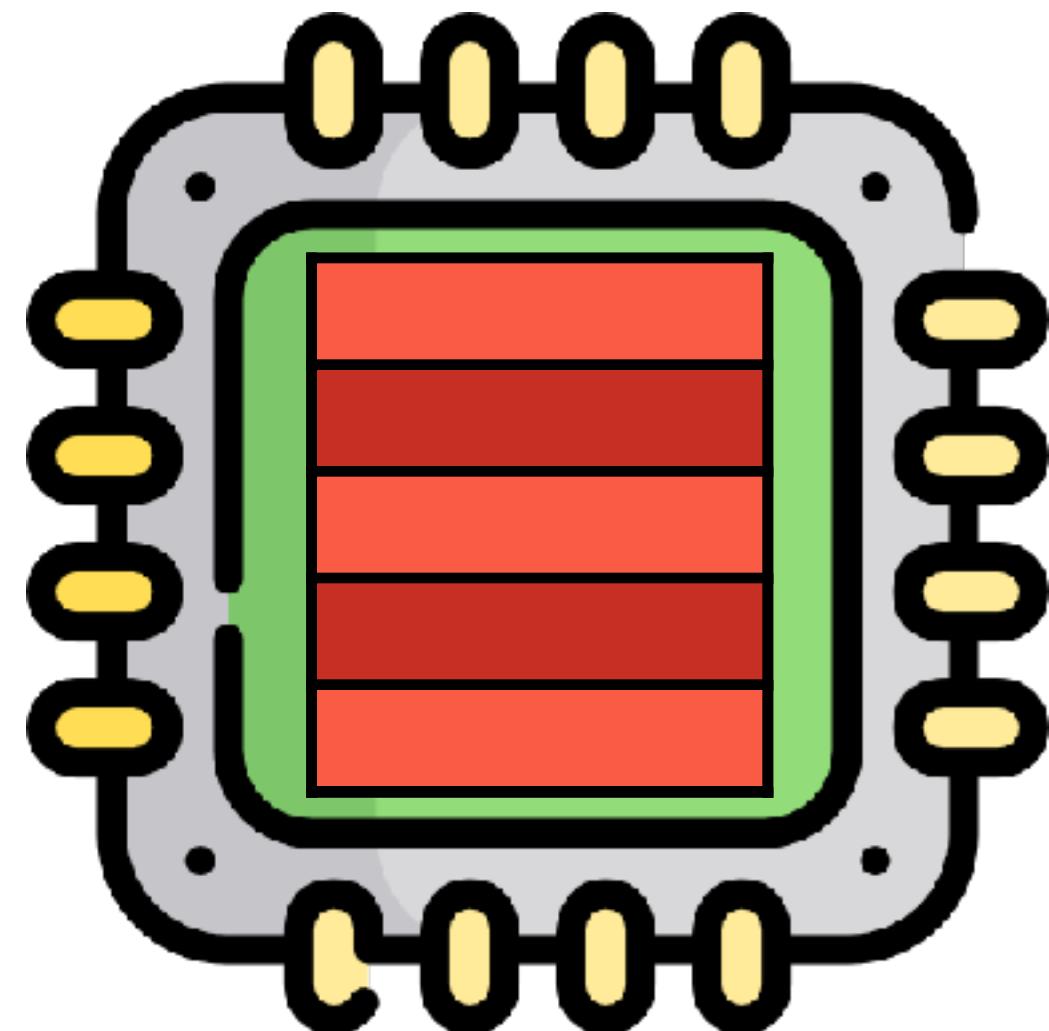
2) Prepare cache



Spectre V1

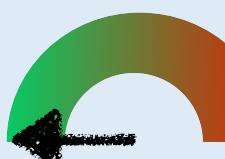


```
void f(int x)
if (x < A_size)
    y = B[A[x]]
```



What is in $A[128]$?

1) Training

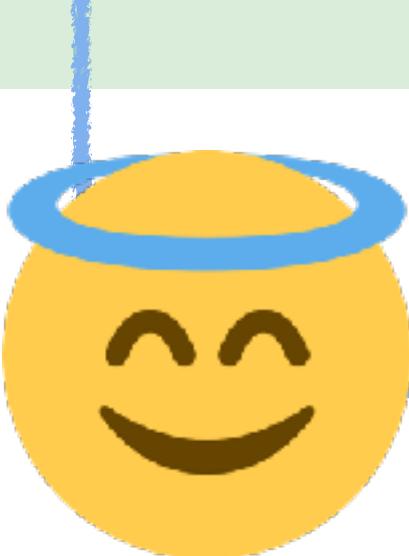


$f(0); f(1); f(2); \dots$

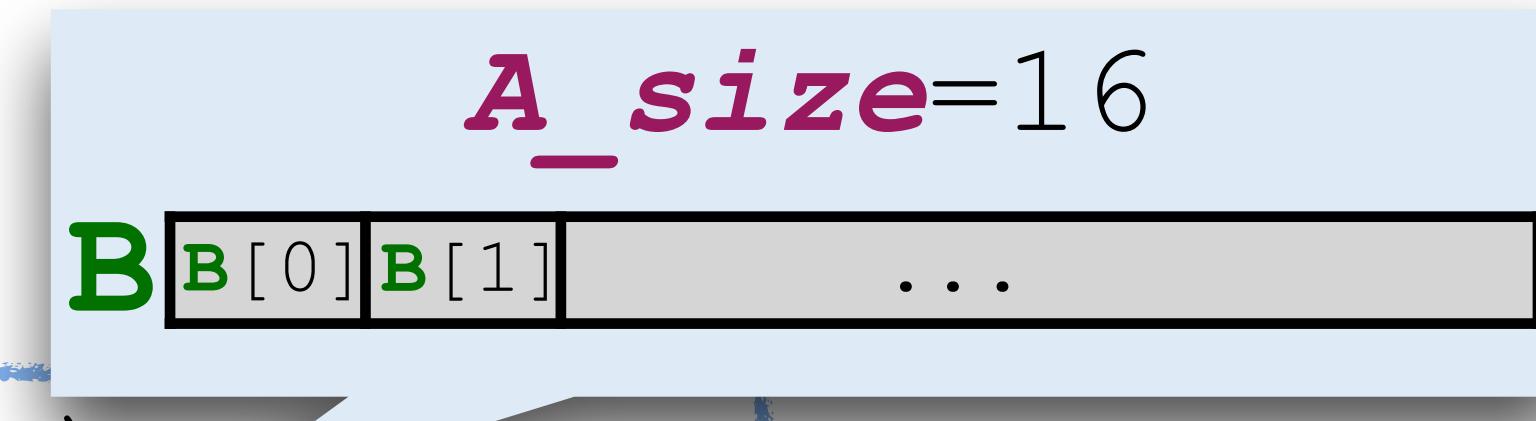
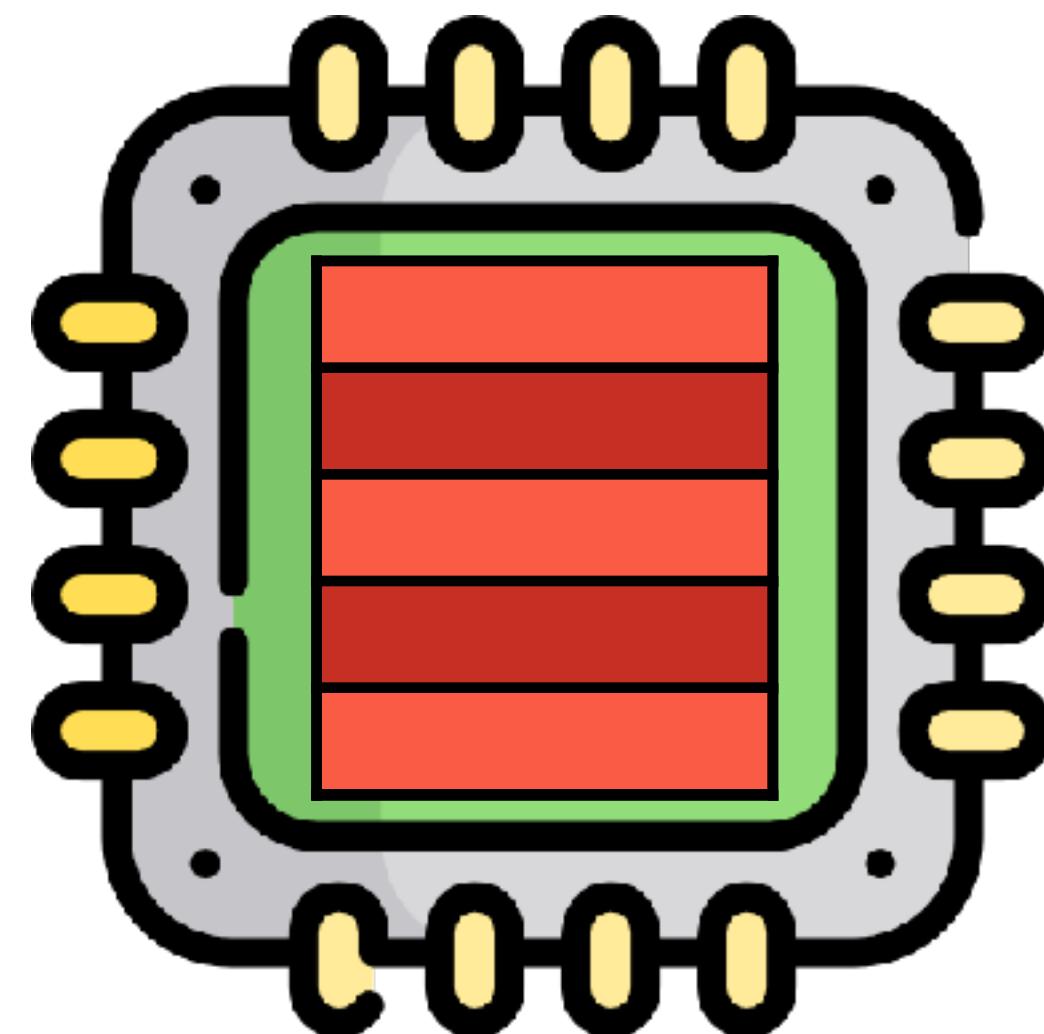
2) Prepare cache

3) Run with $x = 128$

Spectre V1



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



What is in *A[128]*?

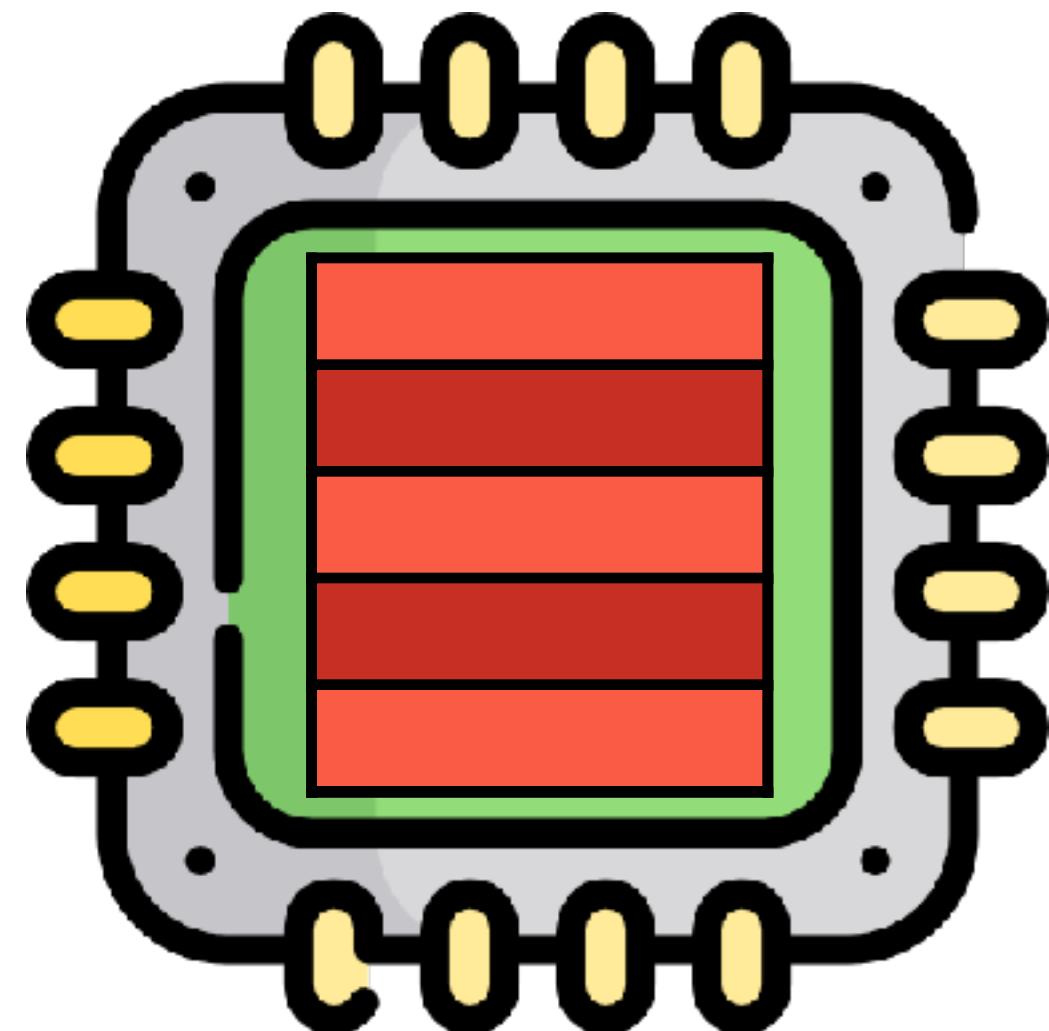
1) Training *f(0); f(1); f(2); ...*

2) Prepare cache

3) Run with *x* = 128

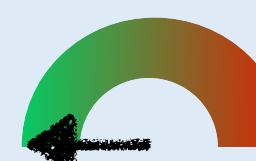
Spectre V1

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



What is in $A[128]$?

1) Training



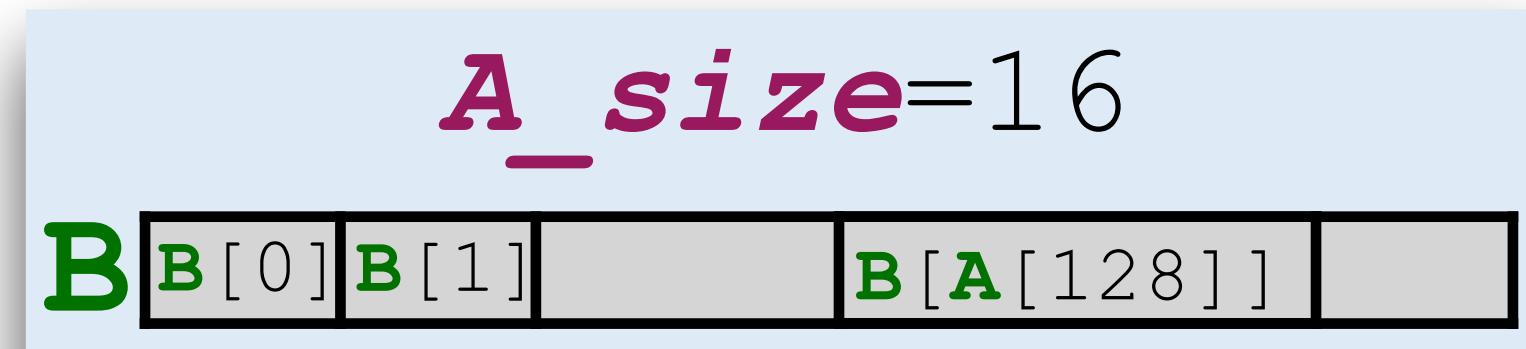
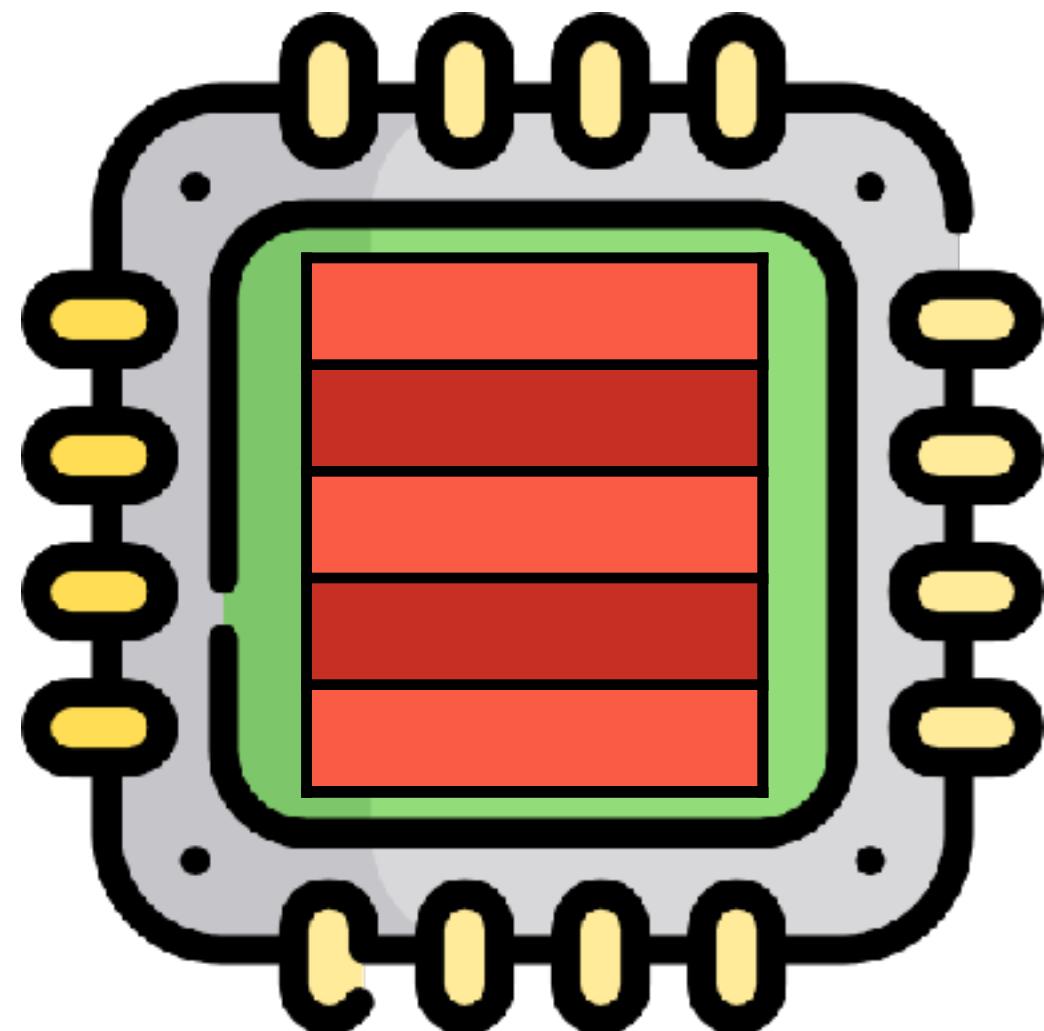
$f(0); f(1); f(2); \dots$

2) Prepare cache

3) Run with $x = 128$

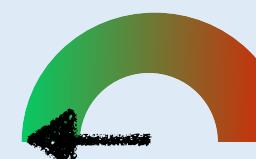
Spectre V1

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



What is in $A[128]$?

1) Training



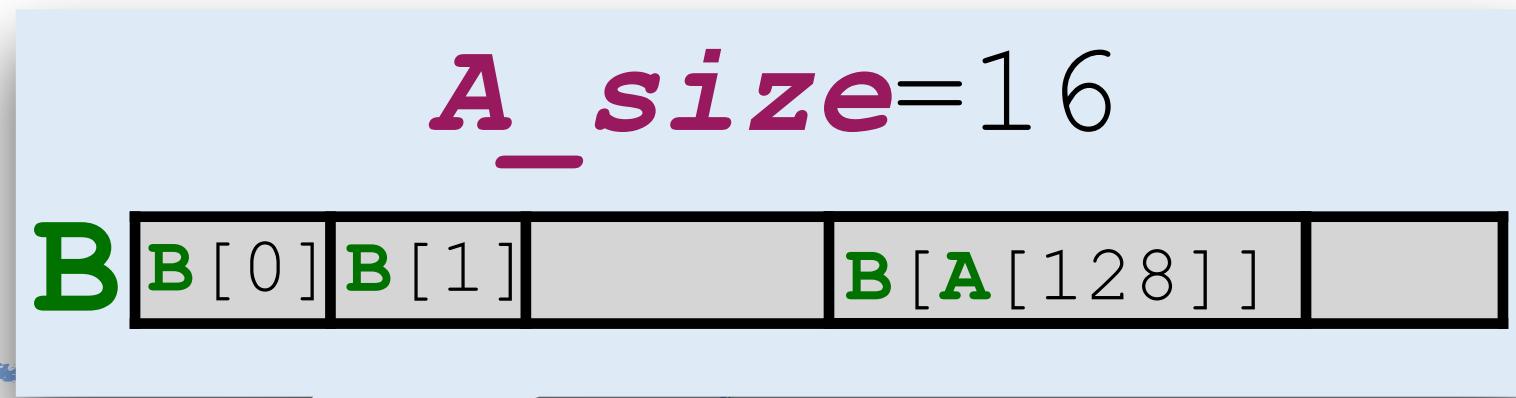
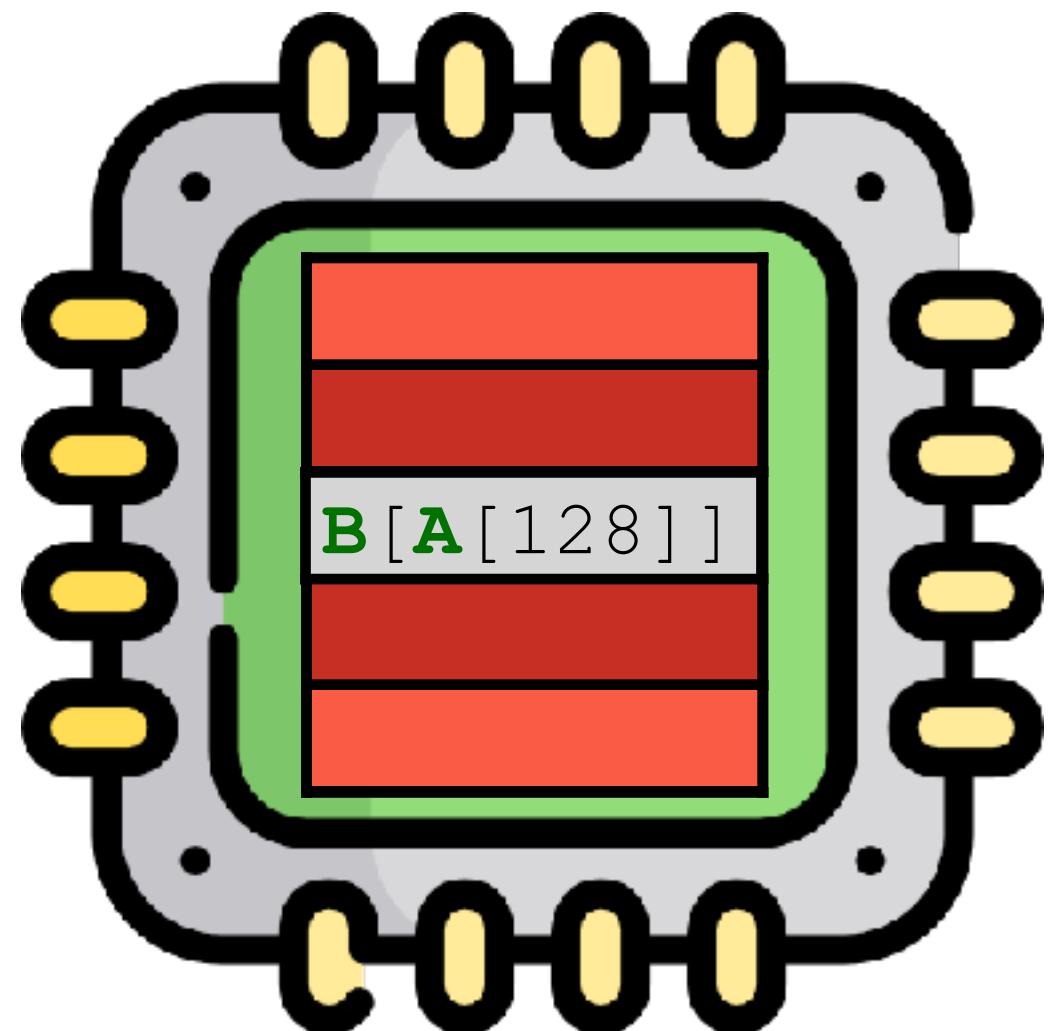
$f(0); f(1); f(2); \dots$

2) Prepare cache

3) Run with $x = 128$

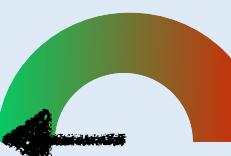
Spectre V1

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



What is in $A[128]$?

1) Training



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

2) Prepare cache

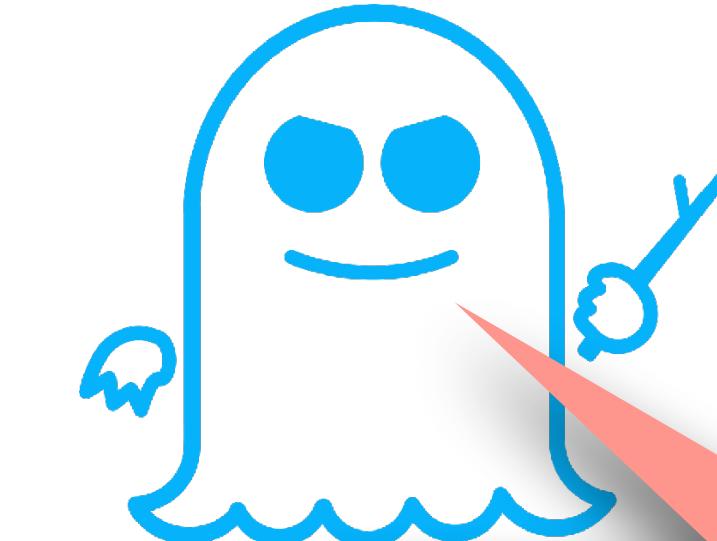
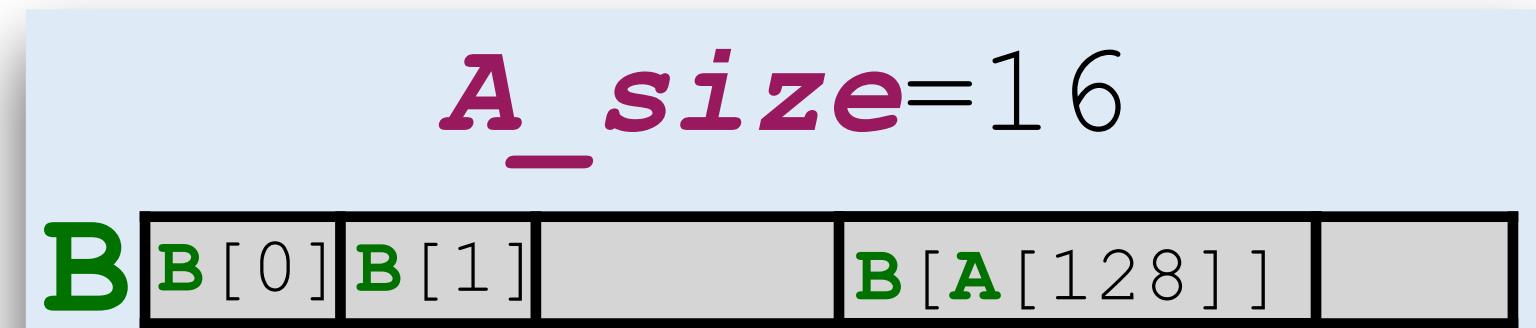
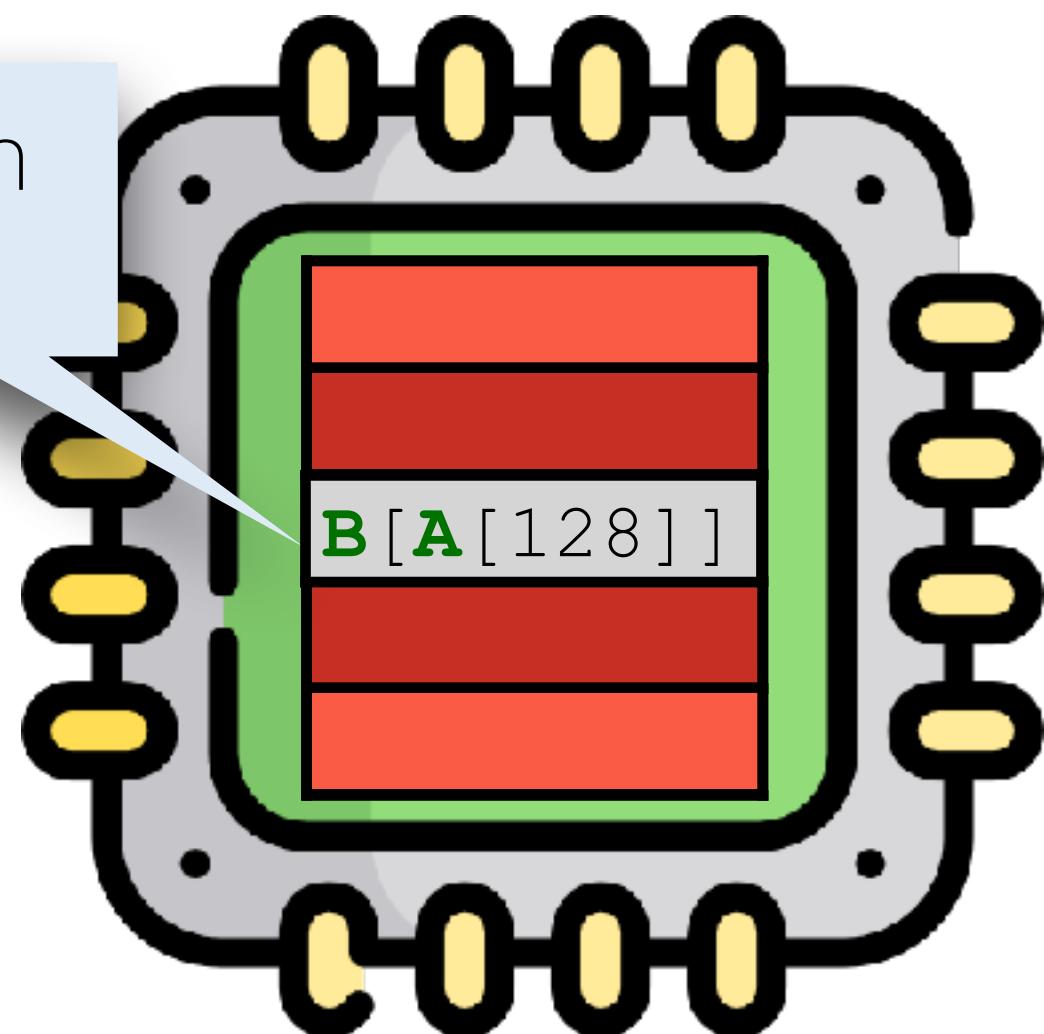
3) Run with $x = 128$

Spectre V1

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

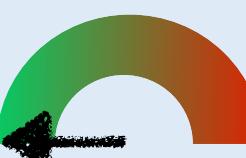


Depends on
A[128]



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

1) Training



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

2) Prepare cache

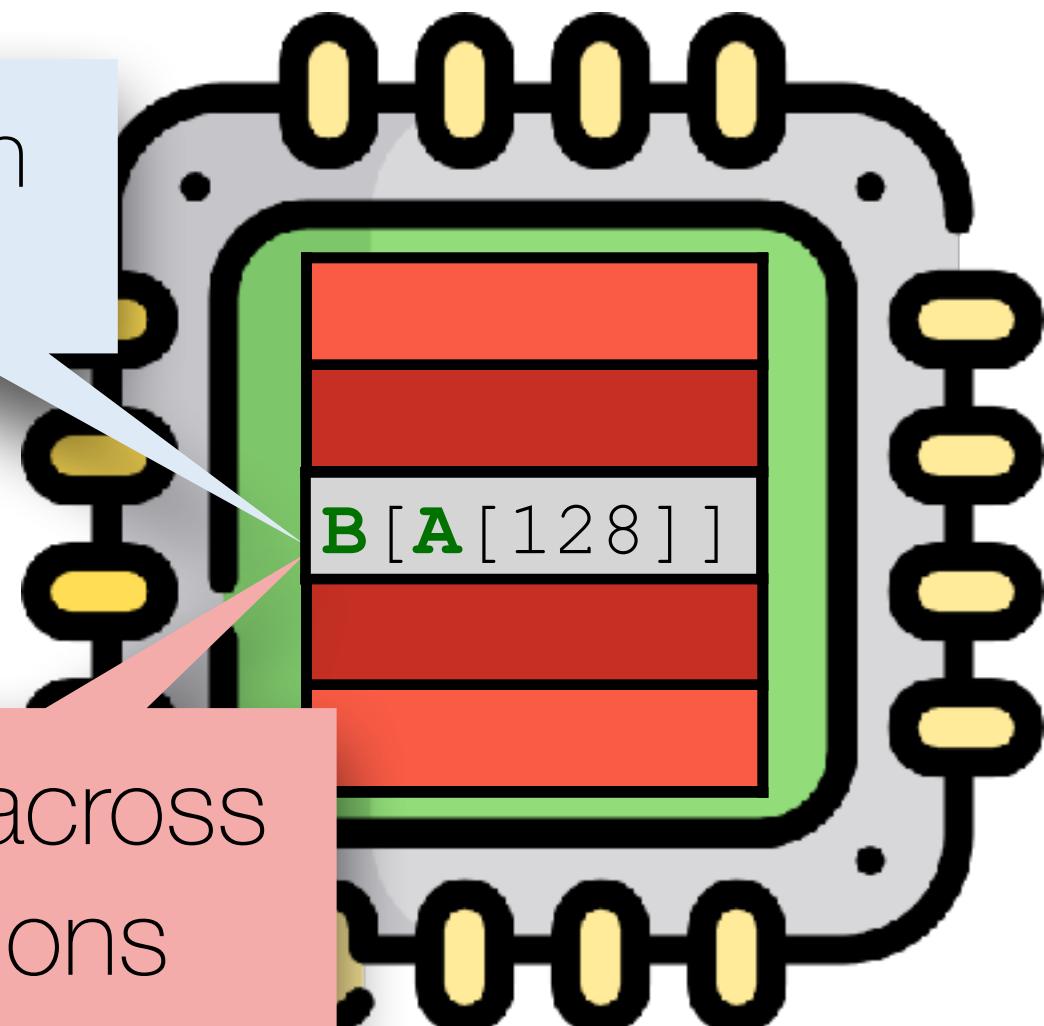
3) Run with **x = 128**

Spectre V1

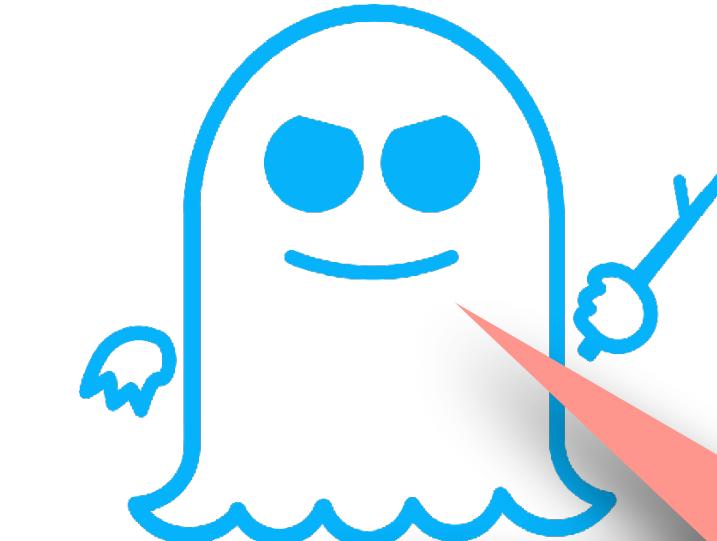
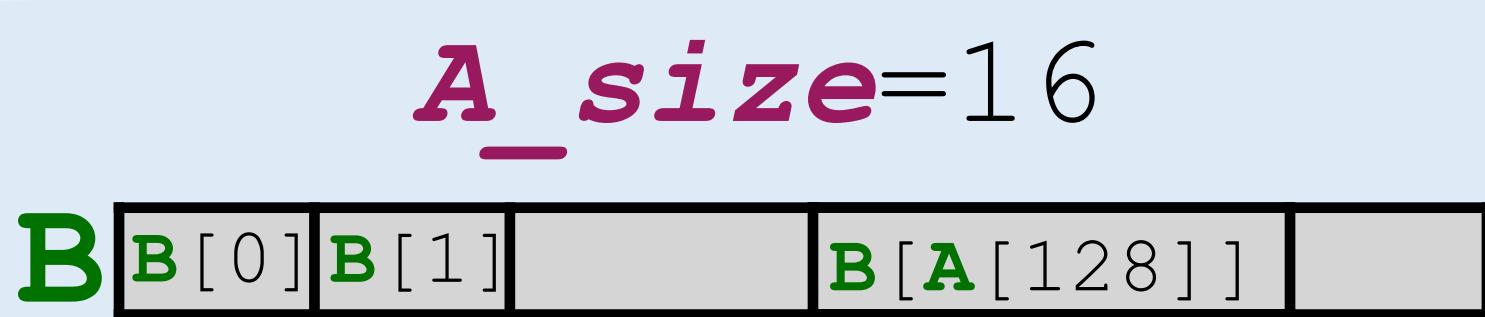
```
void f(int x)
if (x < A_size)
y = B[A[x]]
```



Depends on
A[128]

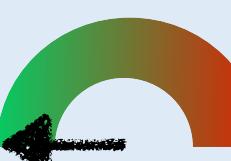


Persistent across
speculations



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

1) Training



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

2) Prepare cache

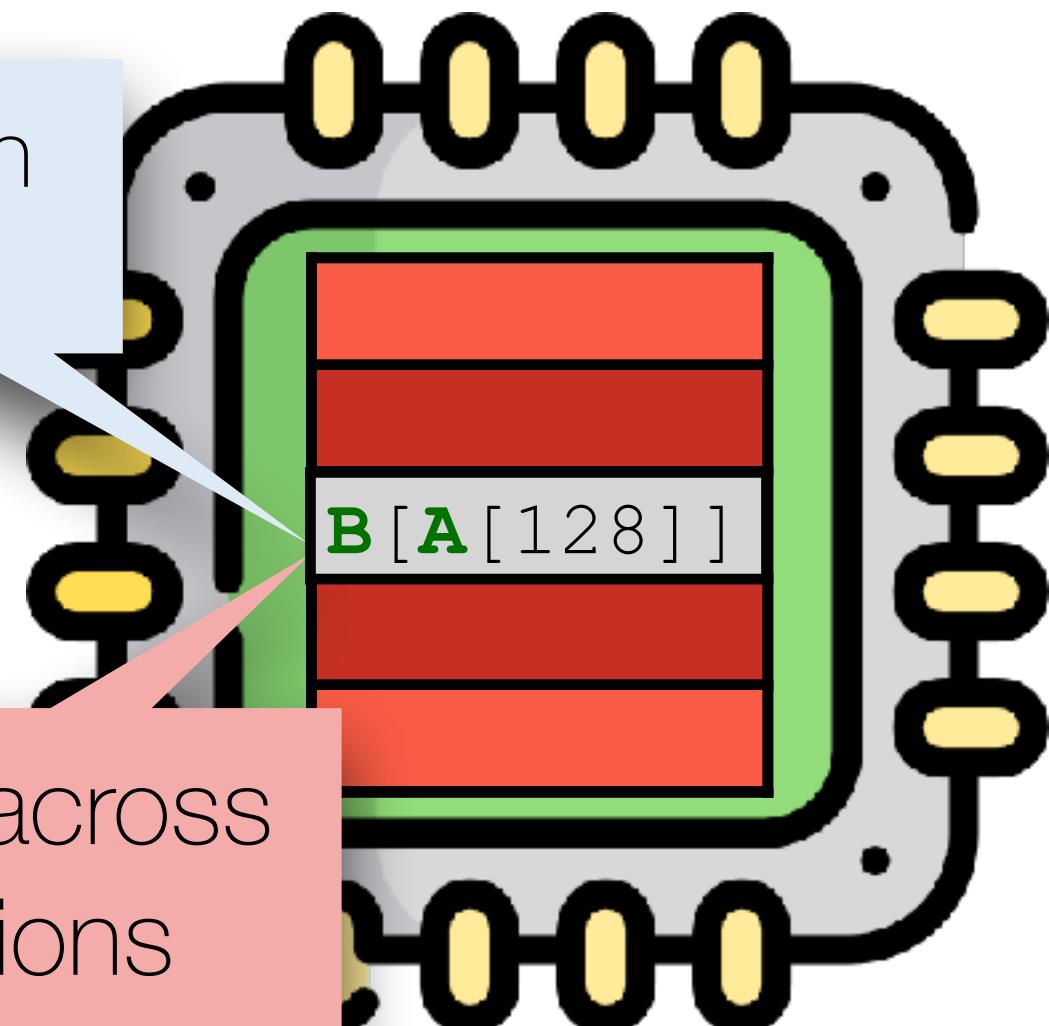
3) Run with **x = 128**

Spectre V1

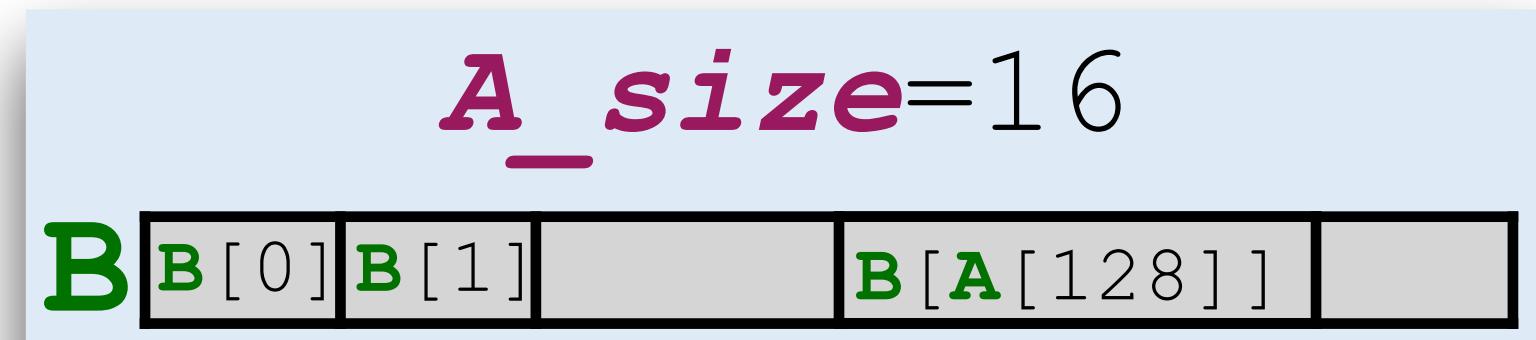


```
void f(int x)
if (x < A_size)
y = B[A[x]]
```

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

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:

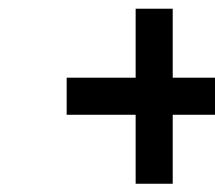
Leakage of P in
non-speculative
execution

=

Leakage of P in
speculative
execution

How to capture leakage?

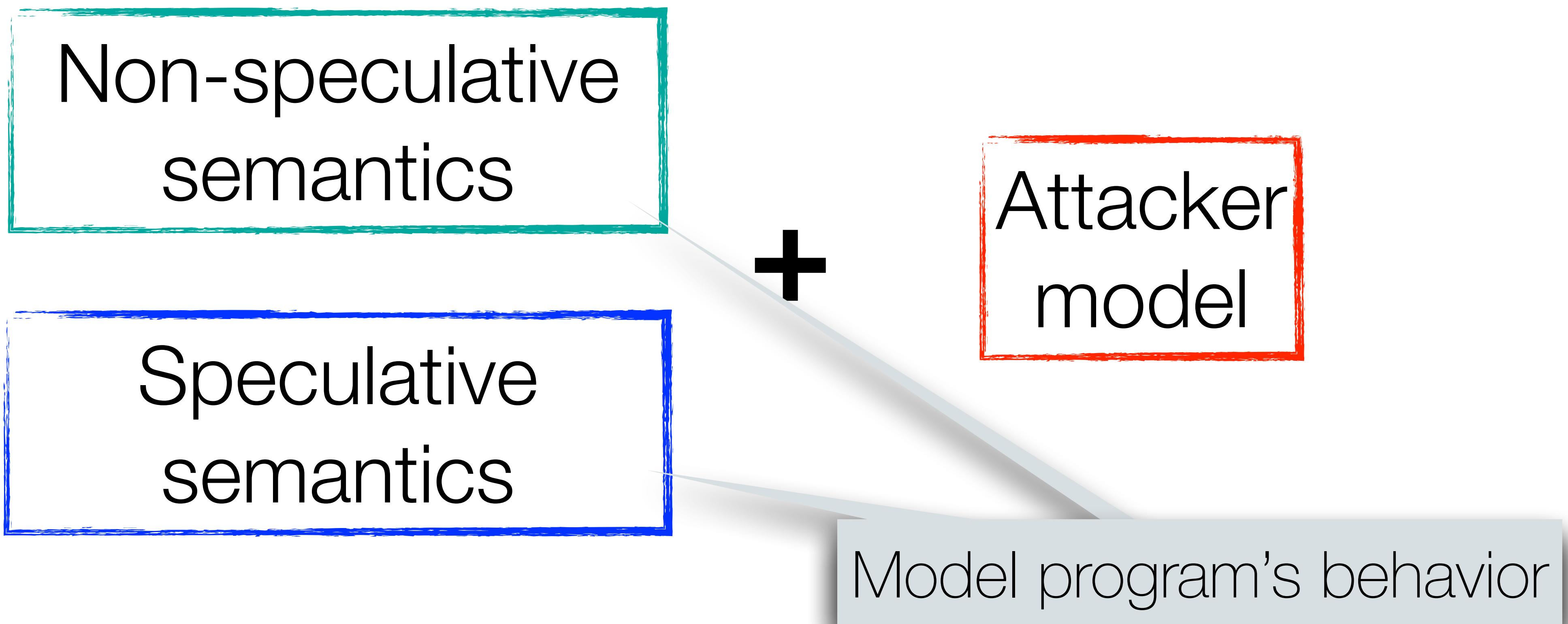
Non-speculative
semantics



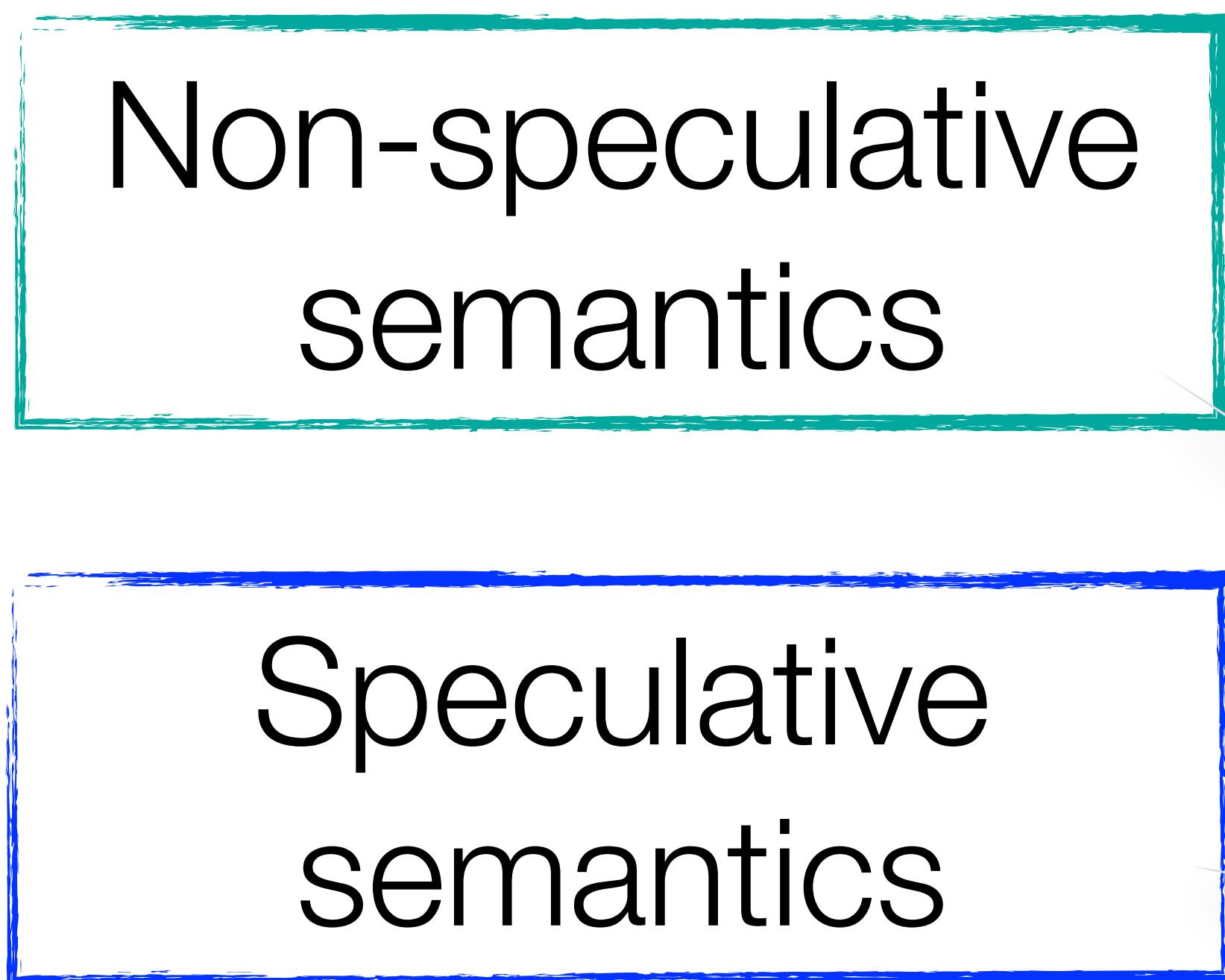
Speculative
semantics

Attacker
model

How to capture leakage?



How to capture leakage?



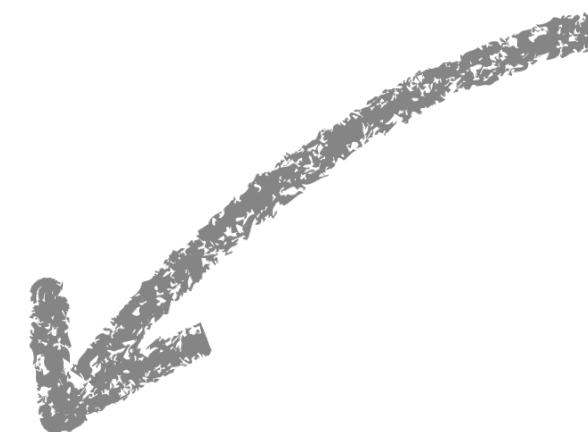
Attacker
model

Capture attacker's
observational power

Model program's behavior

μ Assembly + non-speculative semantics

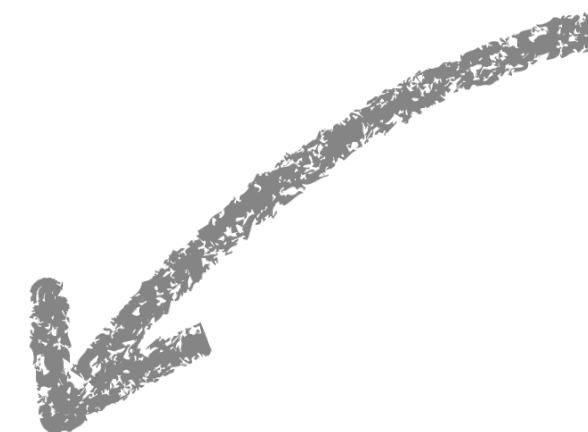
if ($x < A_size$)
y = B[A[x]]



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

μ Assembly + non-speculative semantics

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



rax <- A_size

rcx <- x

jmp rcx≥rax, END

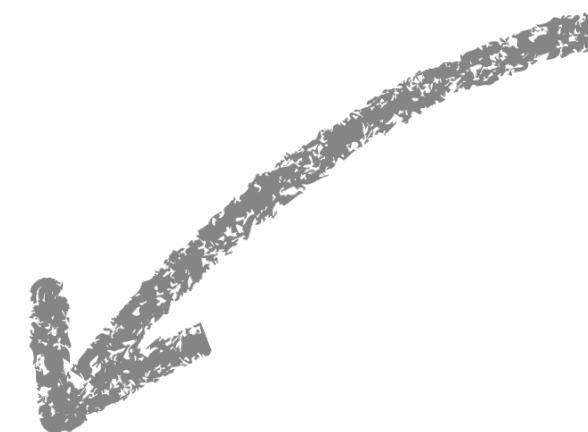
L1: load rax, A + rcx
load rax, B + rax

END:

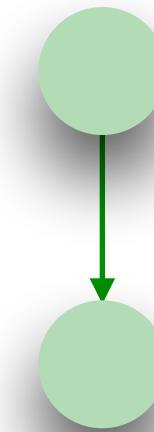


μ Assembly + non-speculative semantics

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

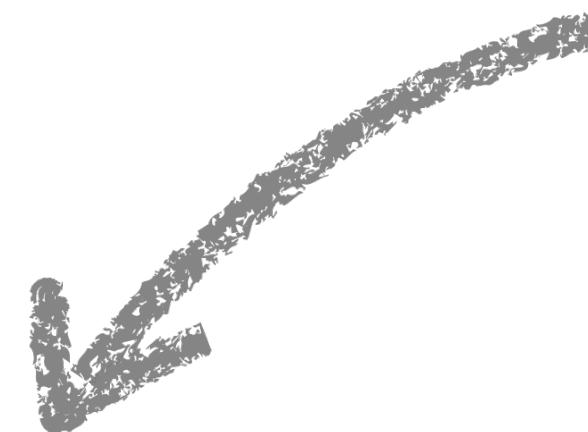


```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



μ Assembly + non-speculative semantics

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



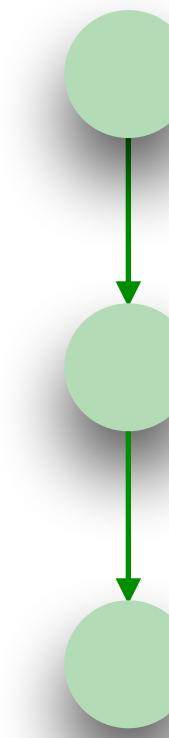
rax <- A_size

rcx <- x

jmp rcx≥rax, END

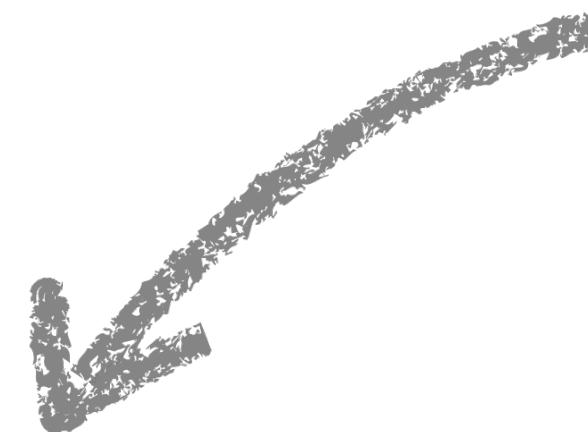
L1: load rax, A + rcx
load rax, B + rax

END:



μ Assembly + non-speculative semantics

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



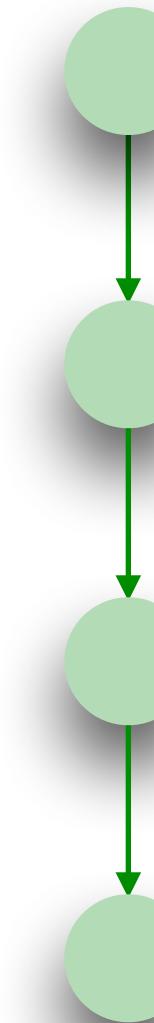
rax <- A_size

rcx <- x

jmp rcx≥rax, END

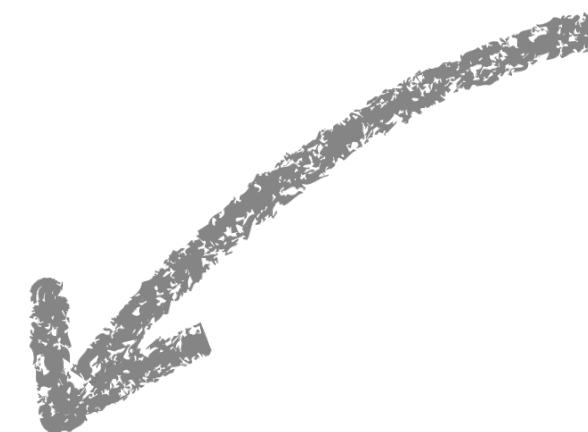
L1: load rax, A + rcx
load rax, B + rax

END:



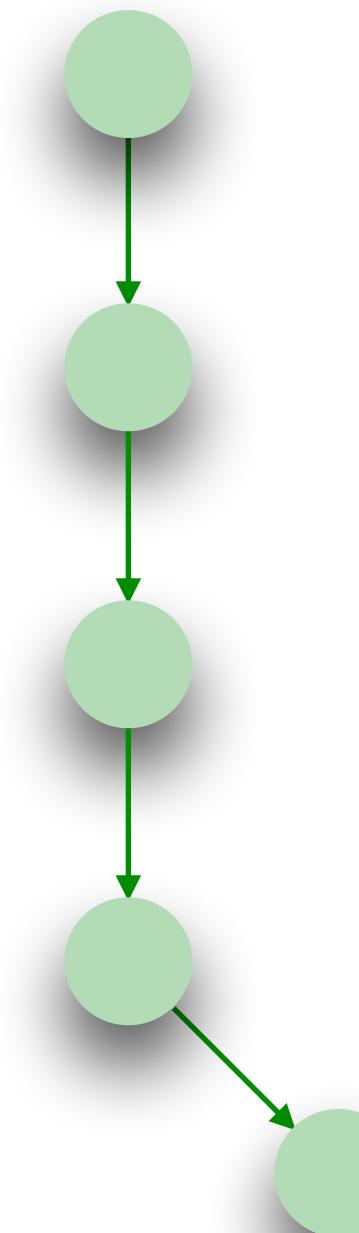
μ Assembly + non-speculative semantics

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

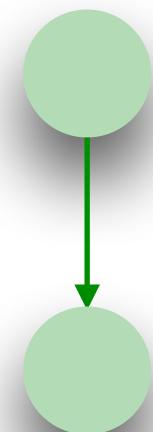
Rolled back upon misspeculation

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

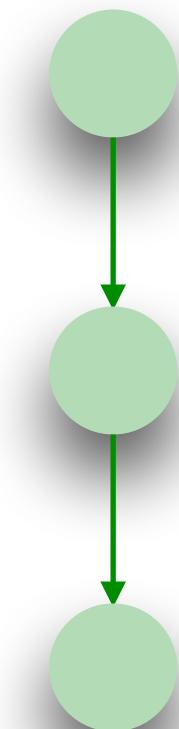
```
rax <- A_size
```

```
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

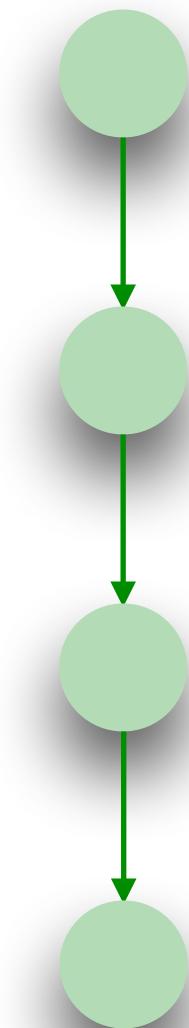
Speculative semantics

```
rax <- A_size
rcx <- x
```

```
jmp rcx≥rax, END
```

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

```
END:
```



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

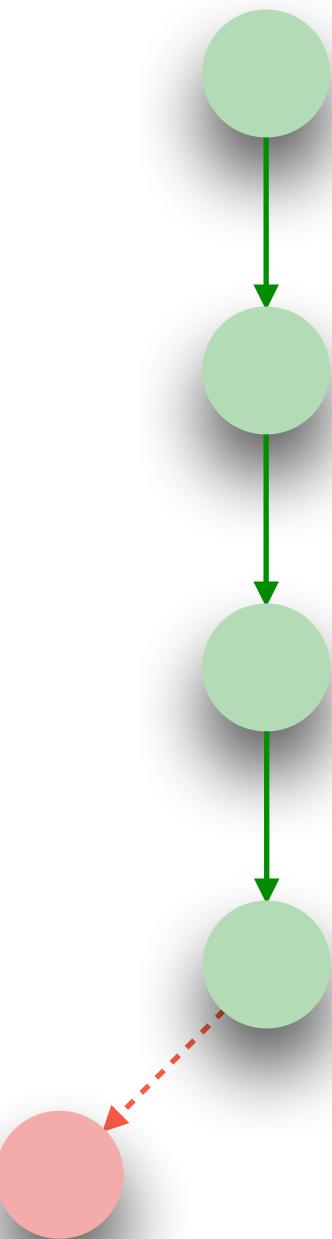
Committed upon
correct speculation

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

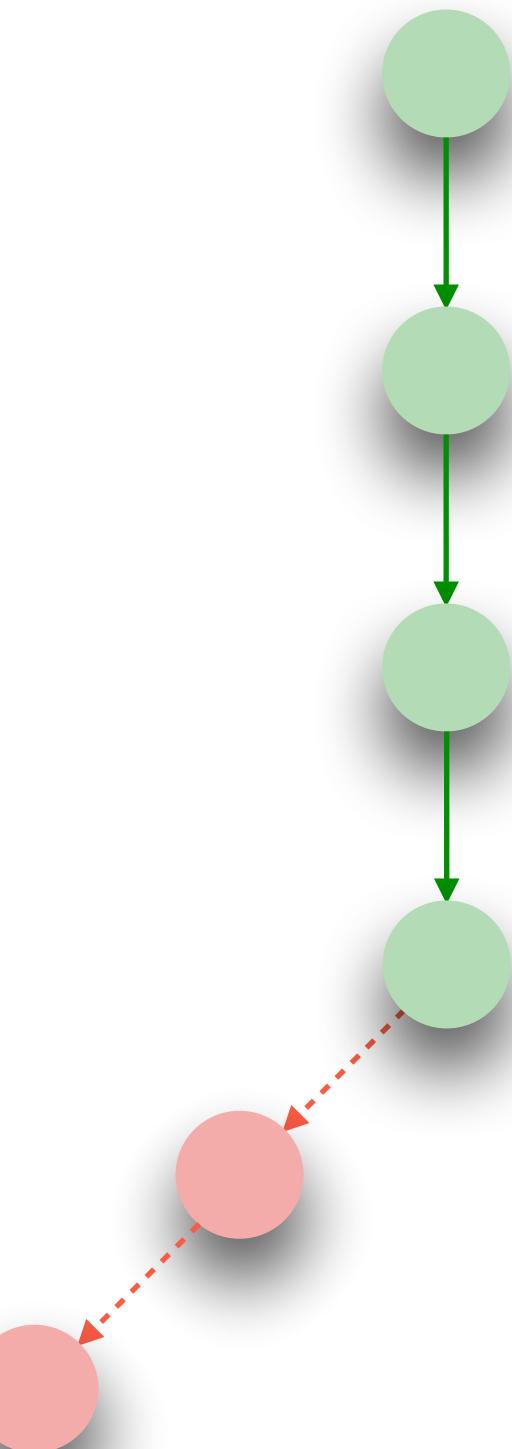
Committed upon
correct speculation

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

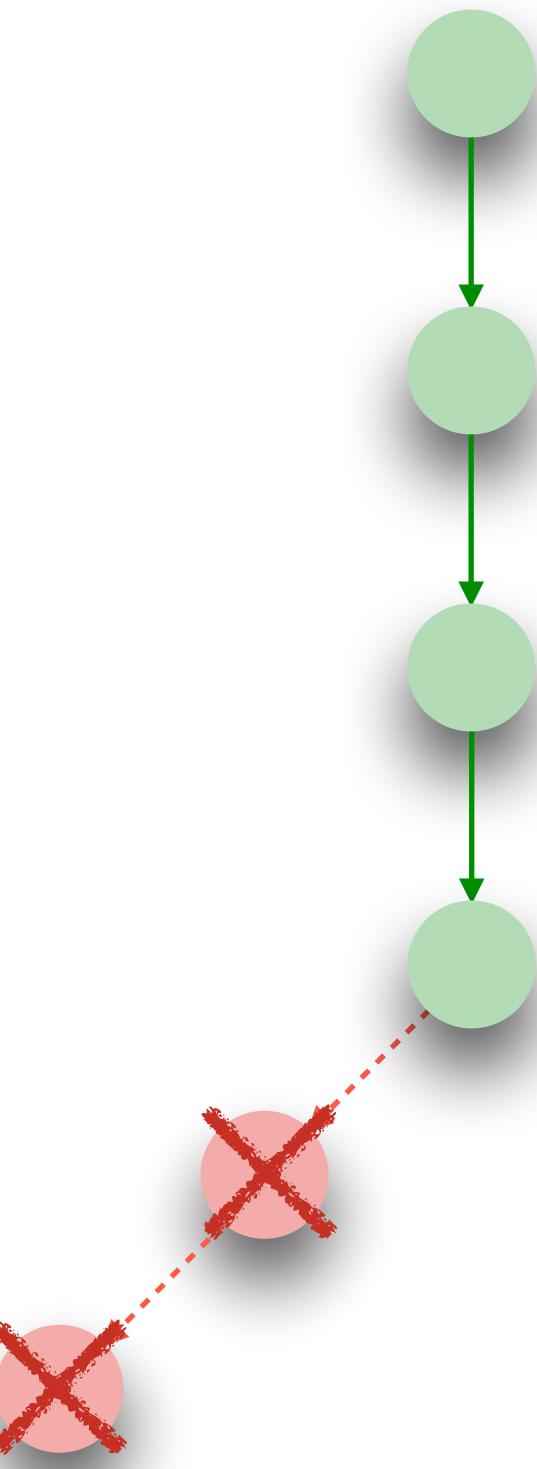
Committed upon
correct speculation

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

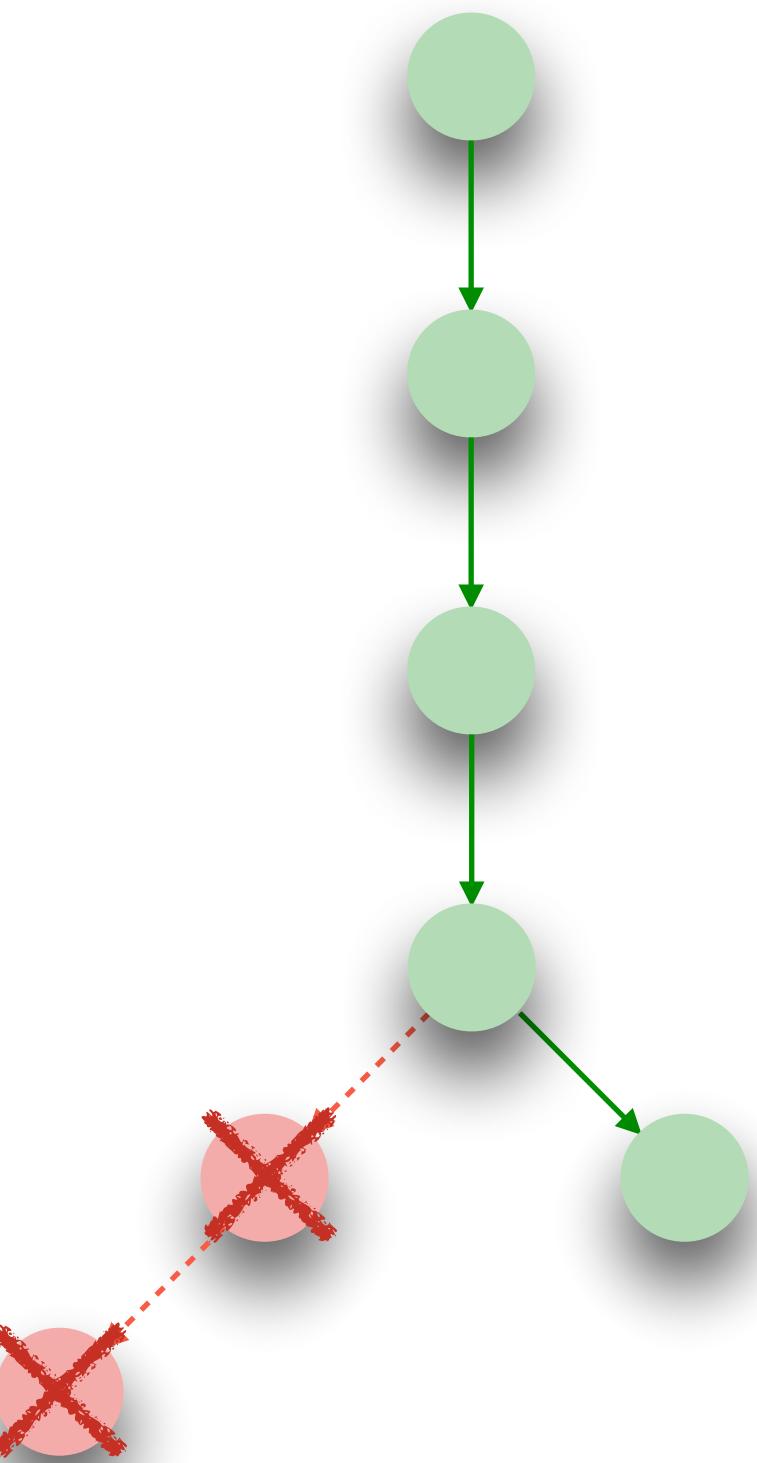
Committed upon
correct speculation

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

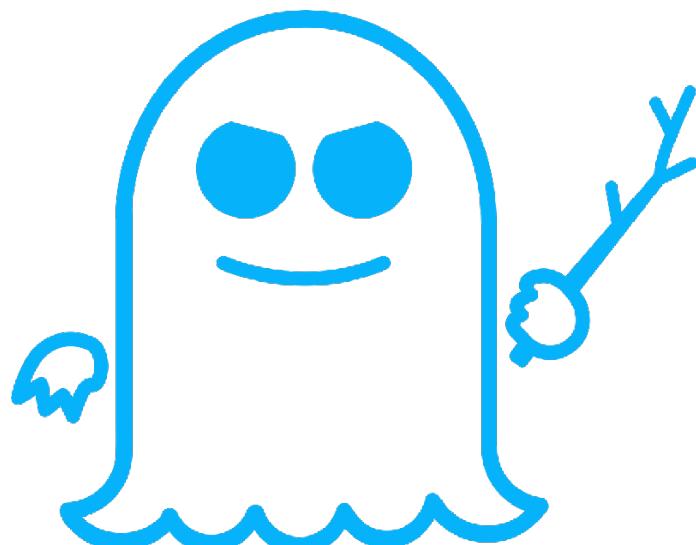
Committed upon
correct speculation

Rolled back upon misspeculation

Prediction Oracle O : branch prediction + length of speculative window

Leakage into μarchitecture

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

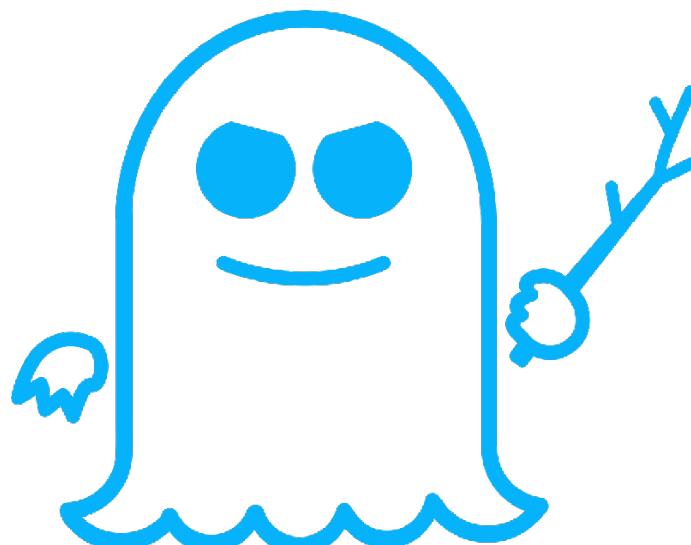


Leakage into μarchitecture

```
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



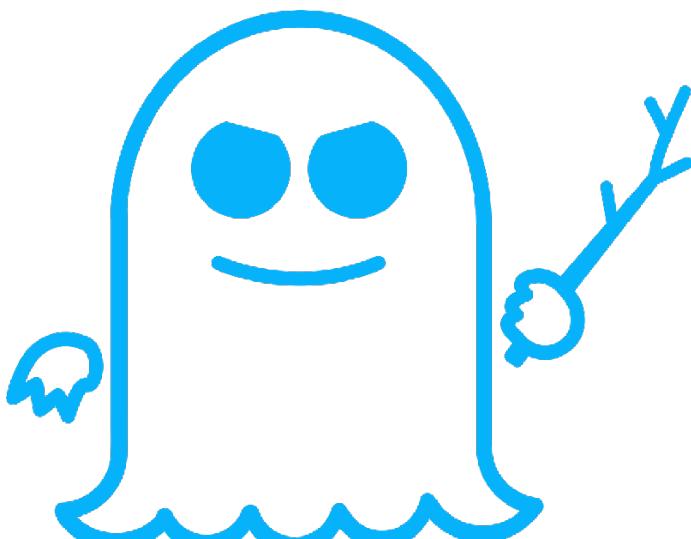
Leakage into μarchitecture

```
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



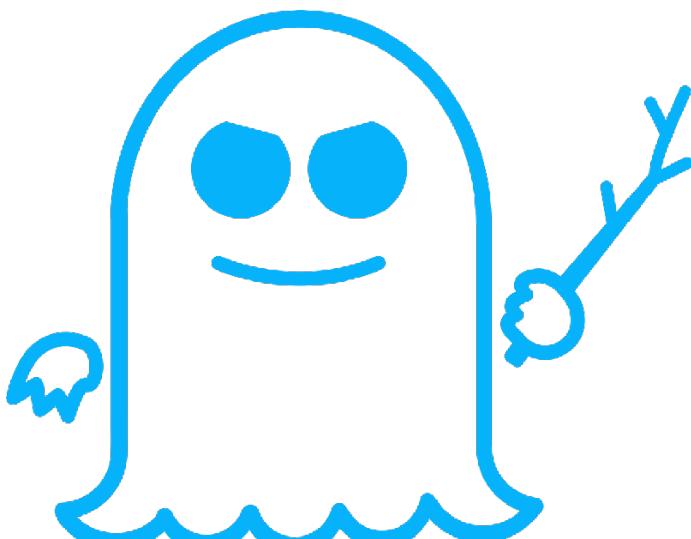
Leakage into μarchitecture

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

Attacker can observe:

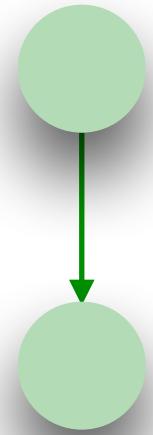
- locations of **memory accesses**
- **branch/jump** targets
- **start/end** speculative execution

Inspired by “constant-time” rqmts



Leakage into μarchitecture

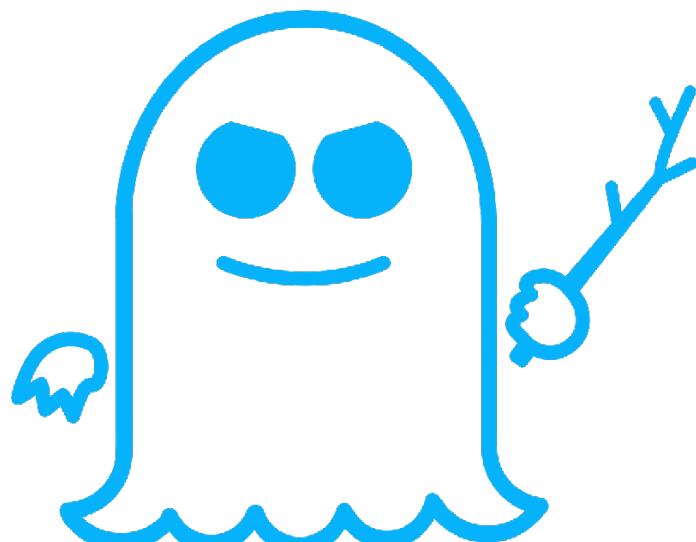
```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
    load rax, B + rax
END:
```



Attacker can observe:

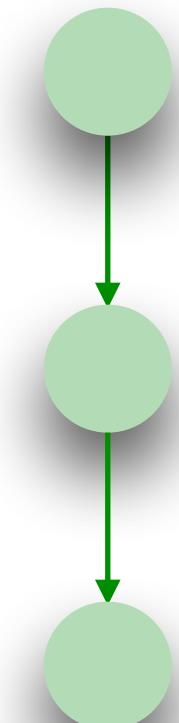
- locations of **memory accesses**
- **branch/jump** targets
- **start/end** speculative execution

Inspired by “constant-time” rqmts



Leakage into μarchitecture

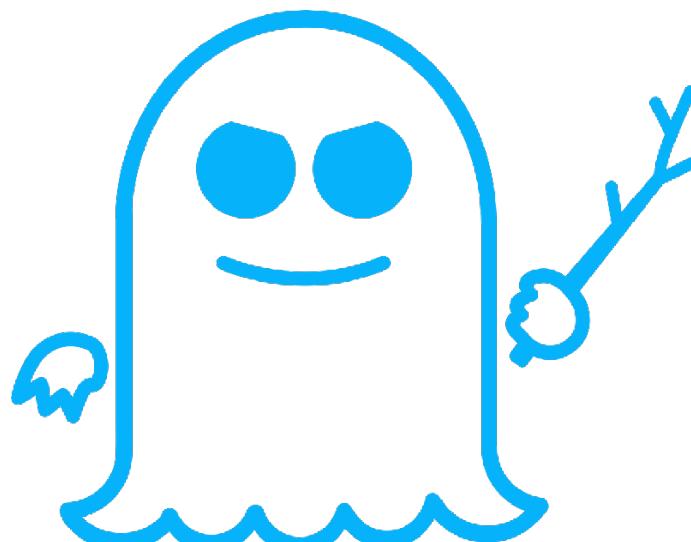
```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
    load rax, B + rax
END:
```



Attacker can observe:

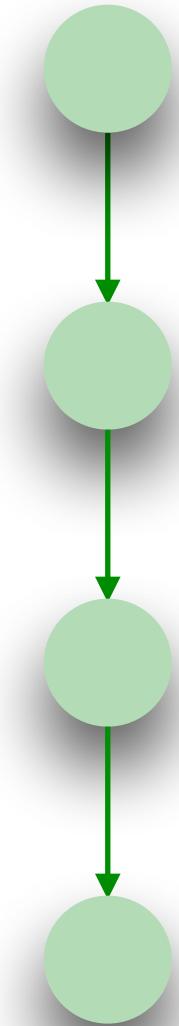
- locations of **memory accesses**
- **branch/jump** targets
- **start/end** speculative execution

Inspired by “constant-time” rqmts



Leakage into μarchitecture

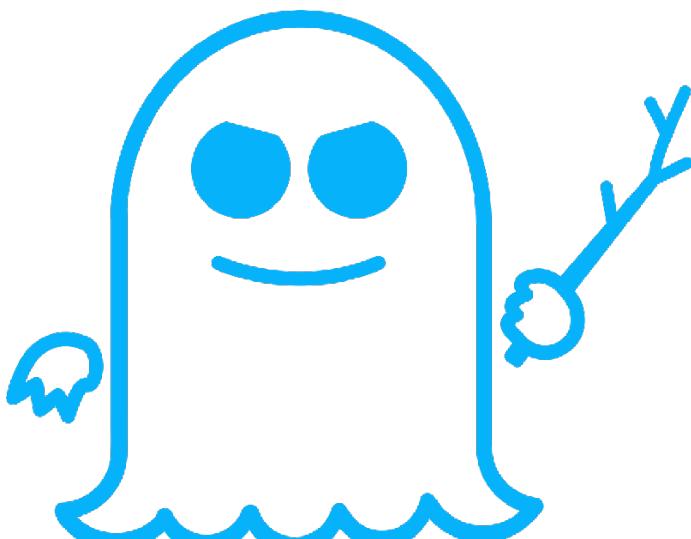
```
rax <- A_size  
rcx <- x  
jmp rcx≥rax, END  
L1: load rax, A + rcx  
     load rax, B + rax  
END:
```



Attacker can observe:

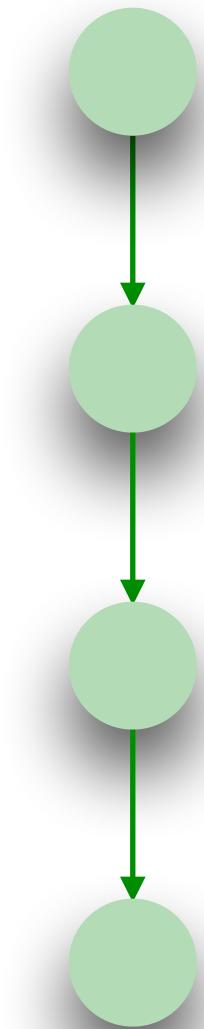
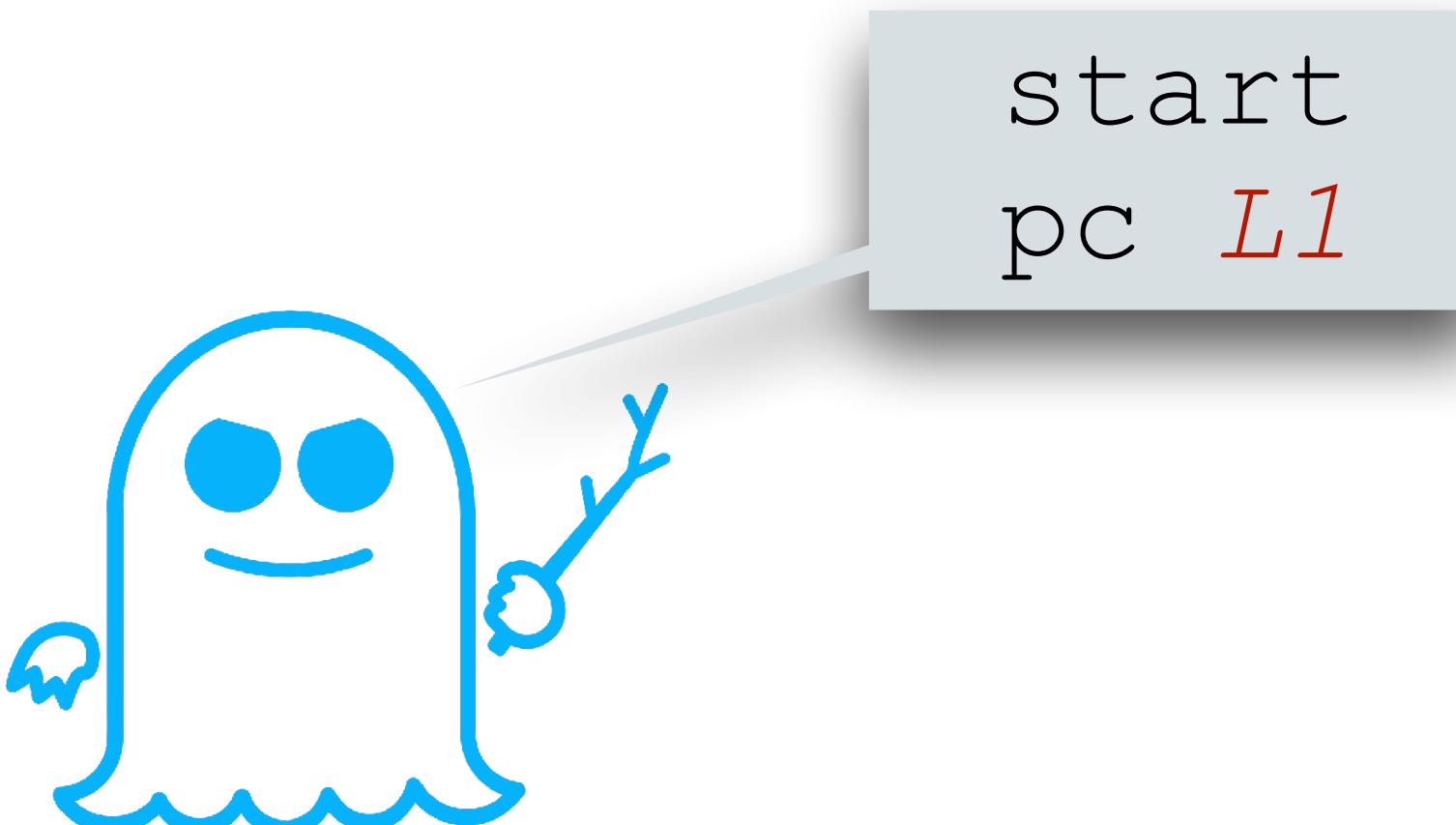
- locations of **memory accesses**
- **branch/jump** targets
- **start/end** speculative execution

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:
```



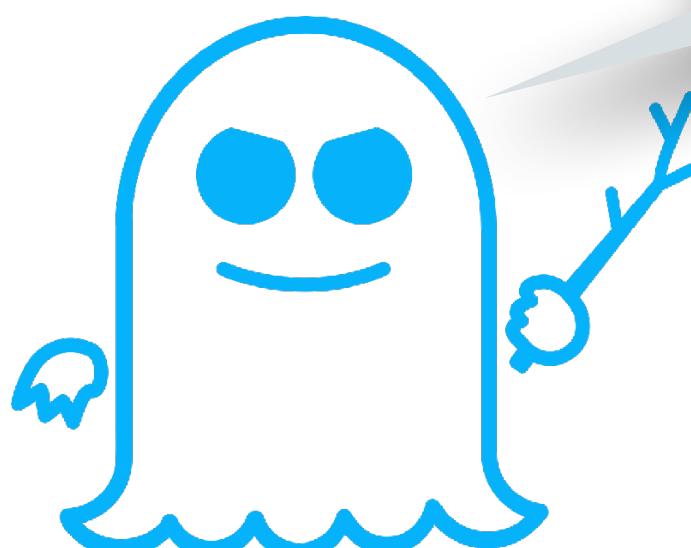
Attacker can observe:

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

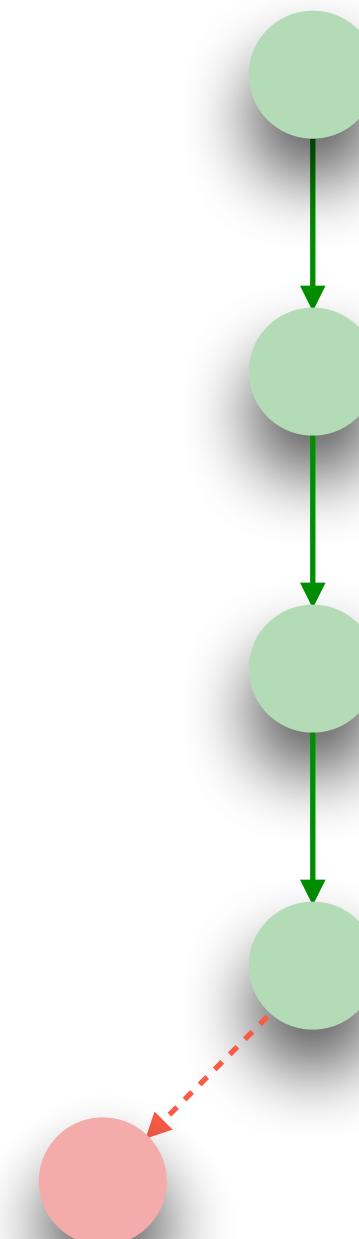
Inspired by “constant-time” rqmts

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

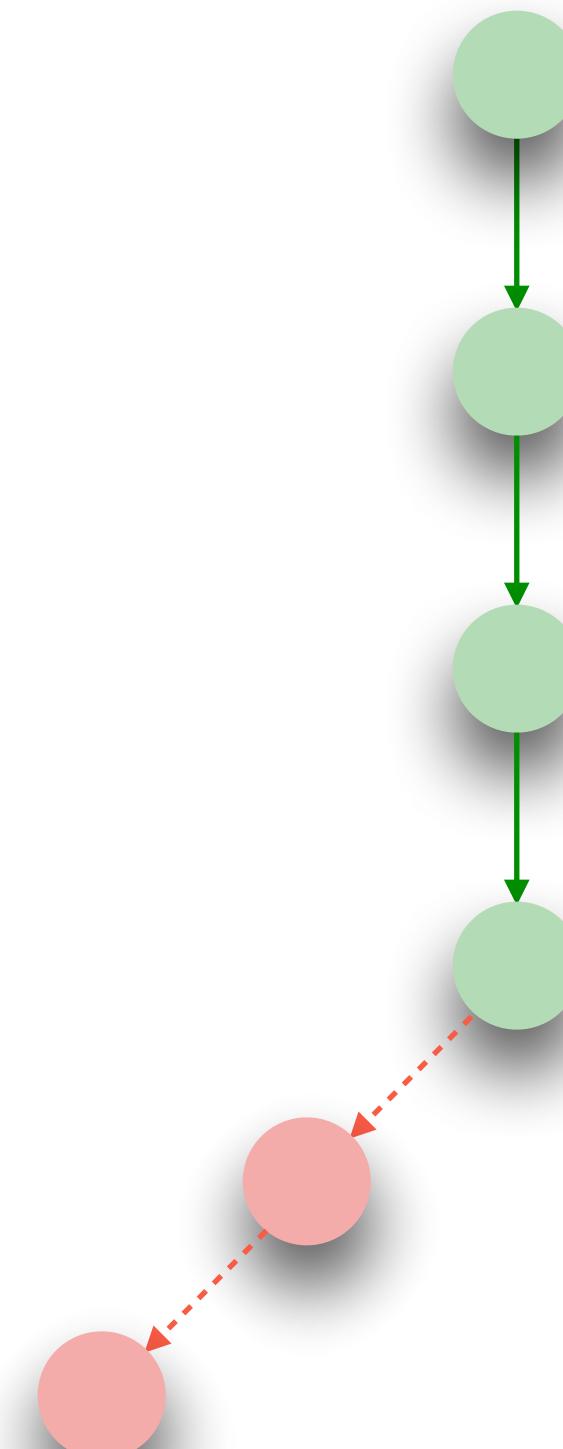
Inspired by “constant-time” rqmts

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**]



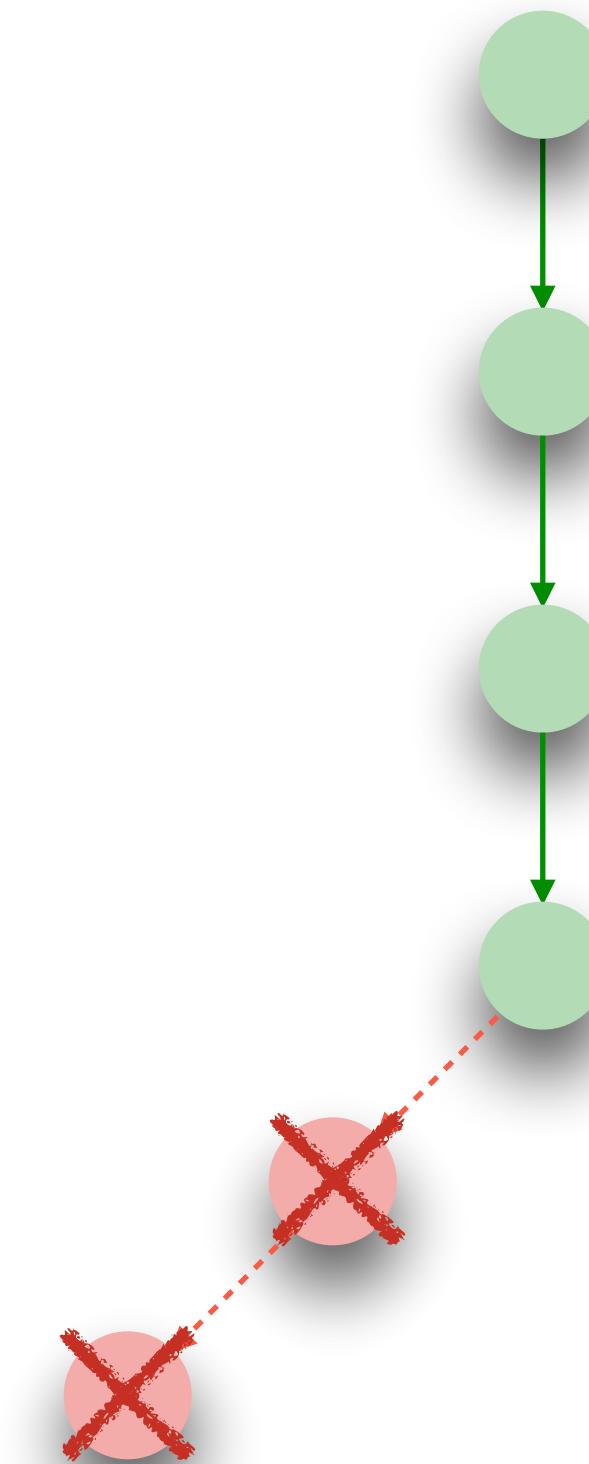
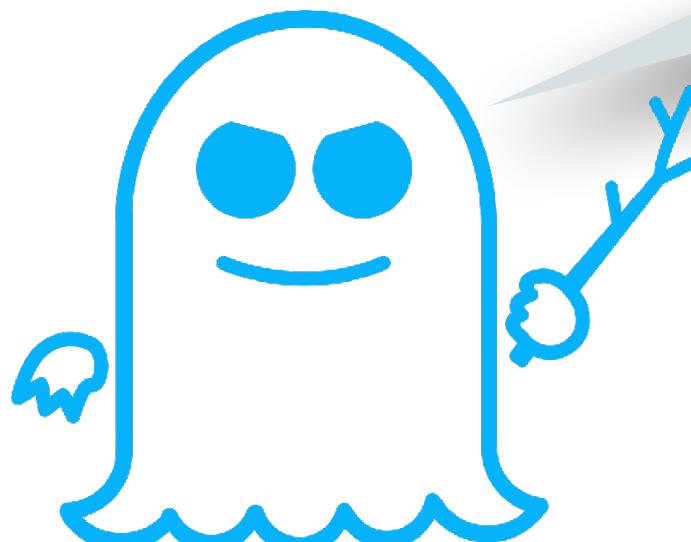
Attacker can observe:

- 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*

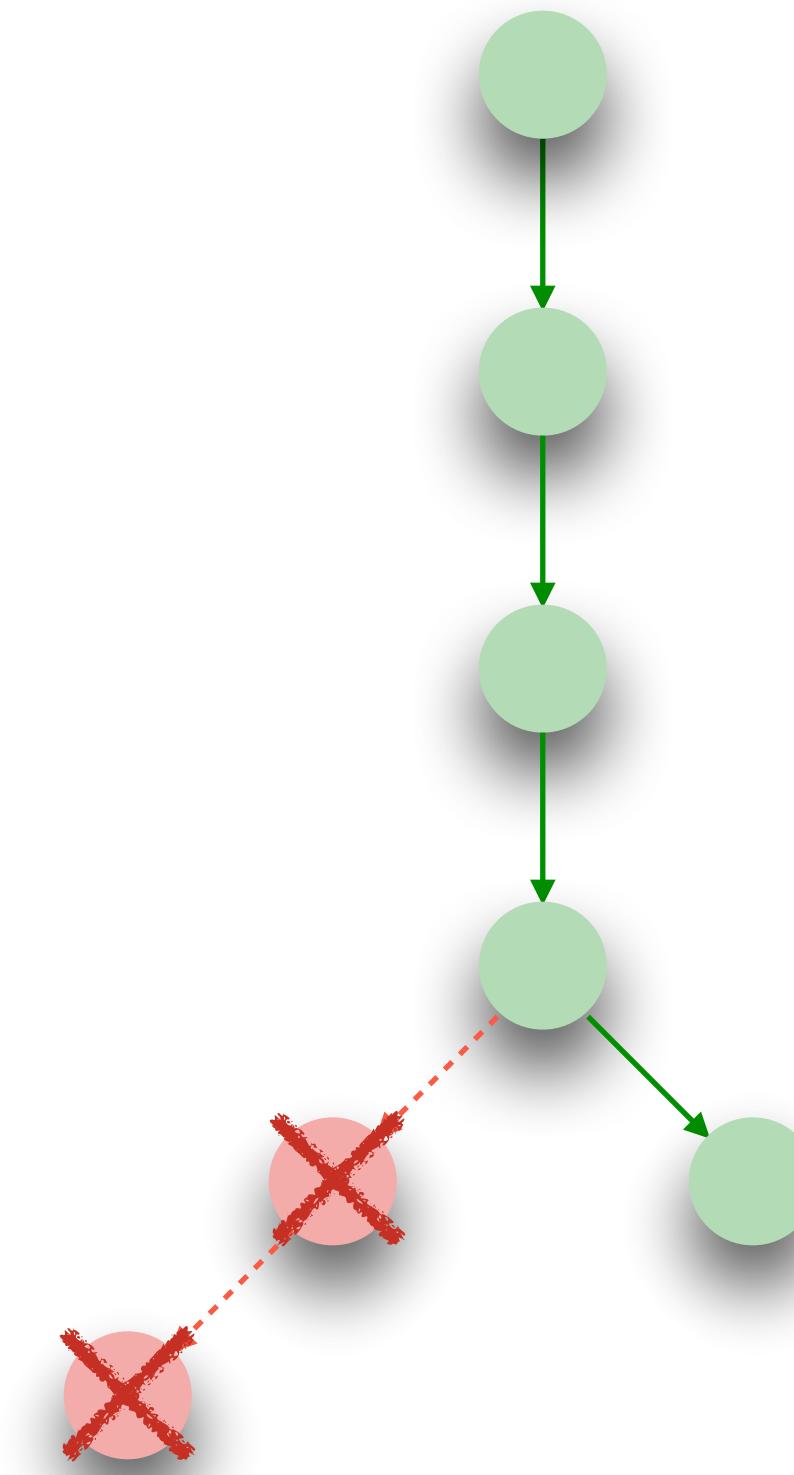
Attacker can observe:

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

Inspired by “constant-time” rqmts

Leakage into μarchitecture

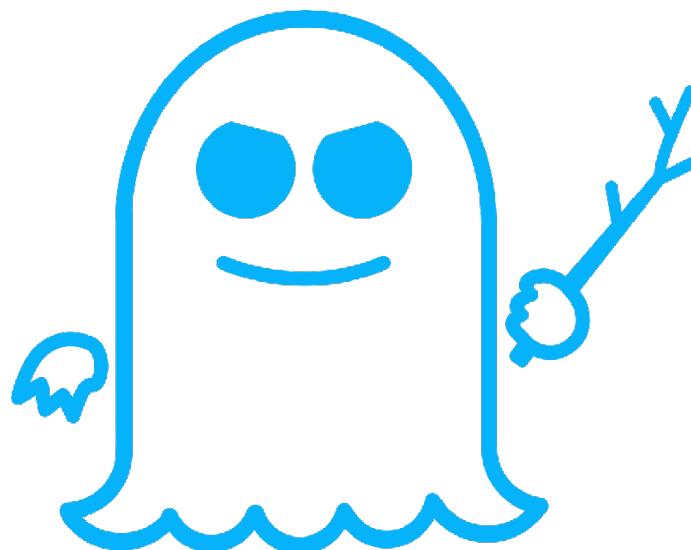
```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



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}' :

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

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**

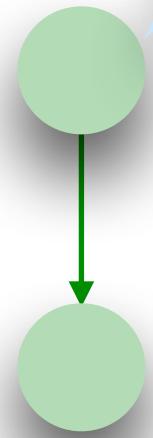
x=128
A_size=16
A[128]=**1**

Speculative non-interference

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

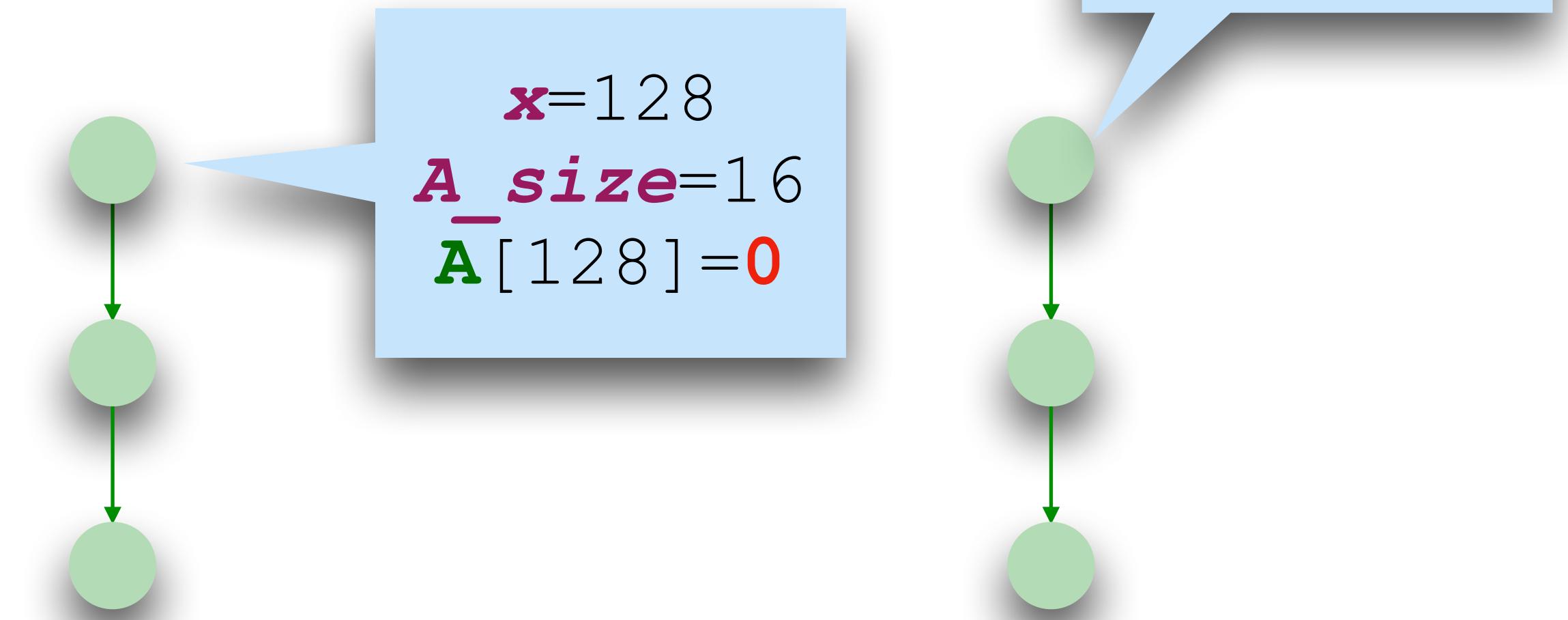
```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
     load rax, B + rax
END:
```

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



Speculative non-interference

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



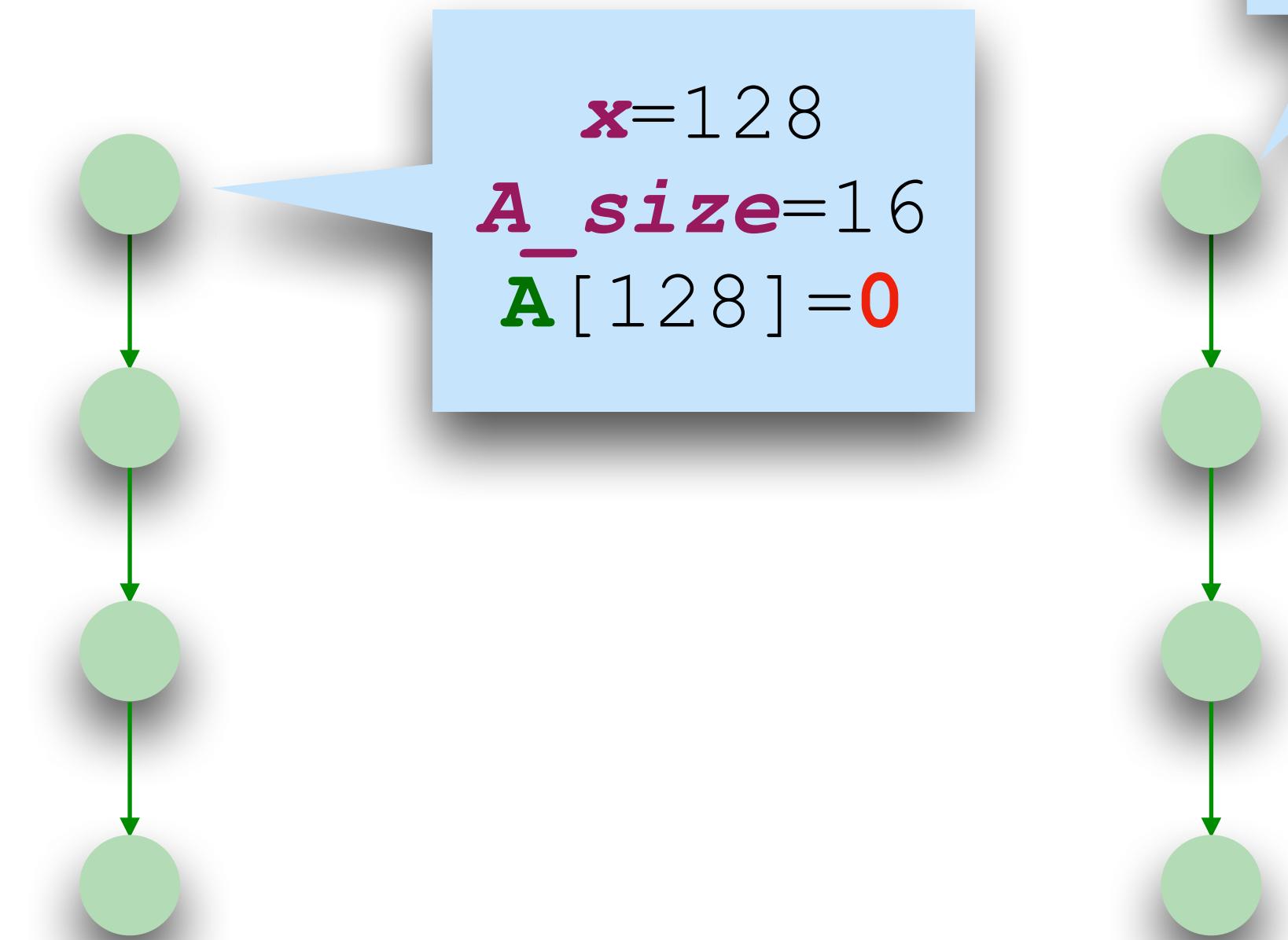
Speculative non-interference

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

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

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

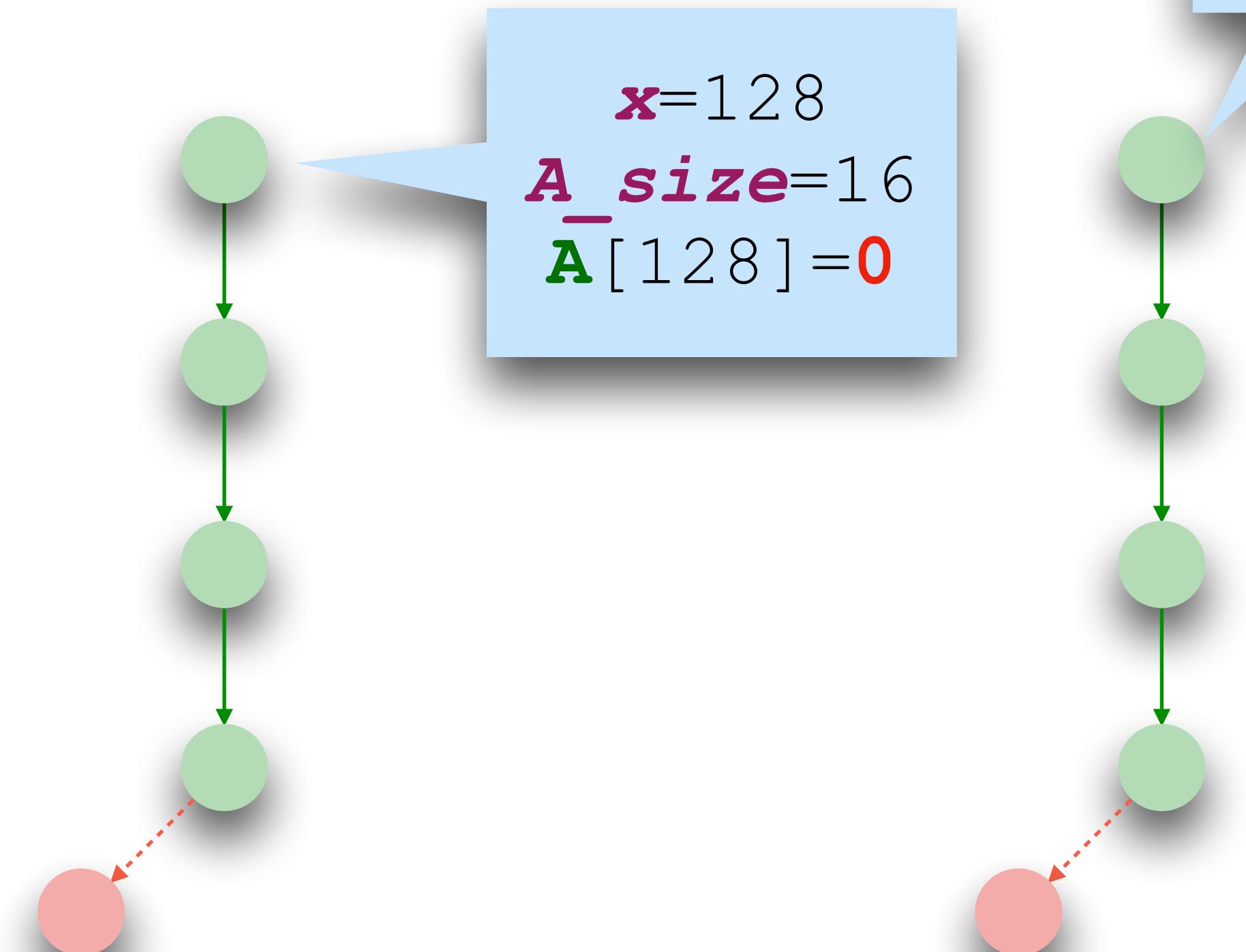
END:



Speculative non-interference

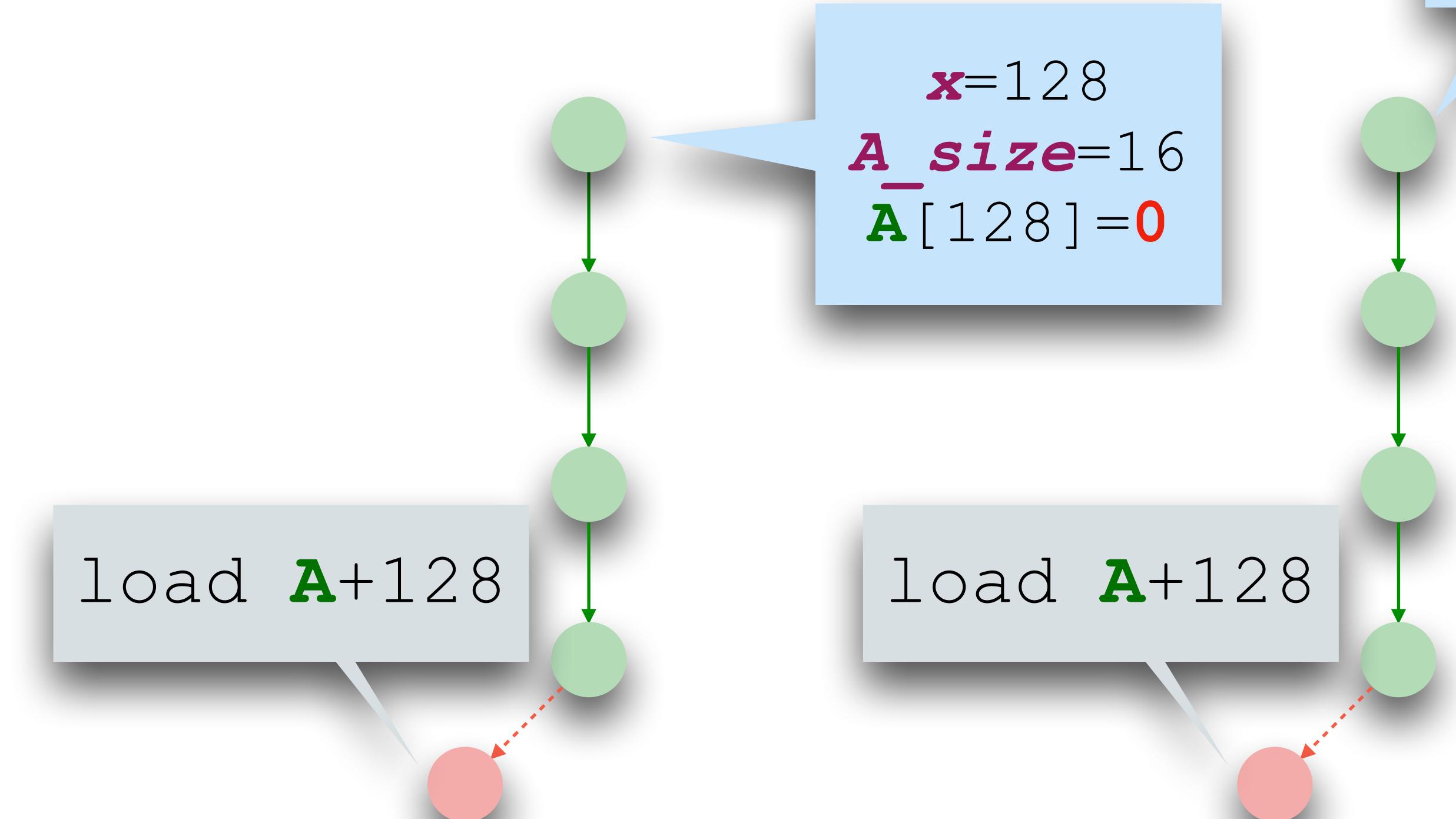
$x=128$
 $A_size=16$
 $\mathbf{A}[128]=1$

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



Speculative non-interference

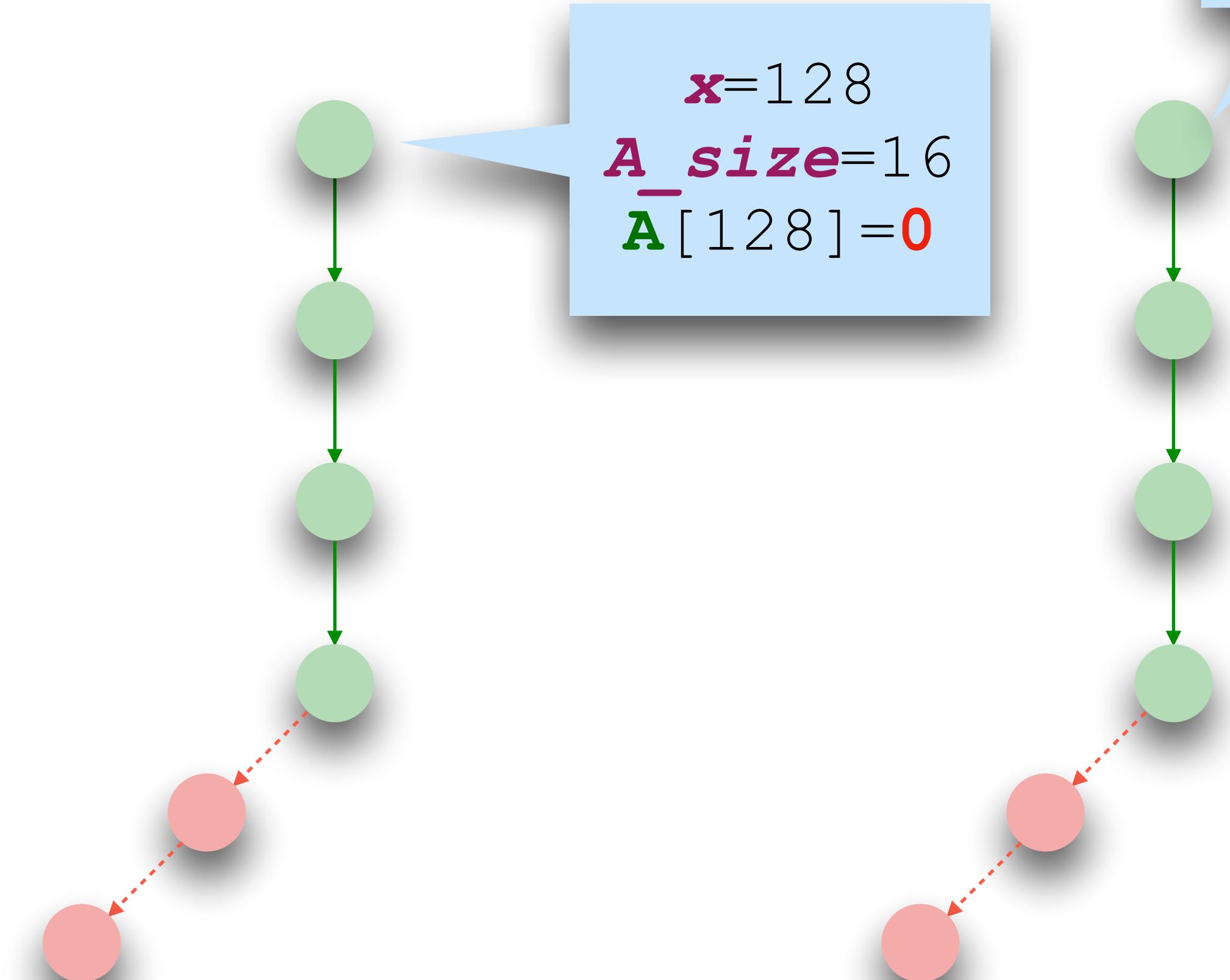
```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



Speculative non-interference

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

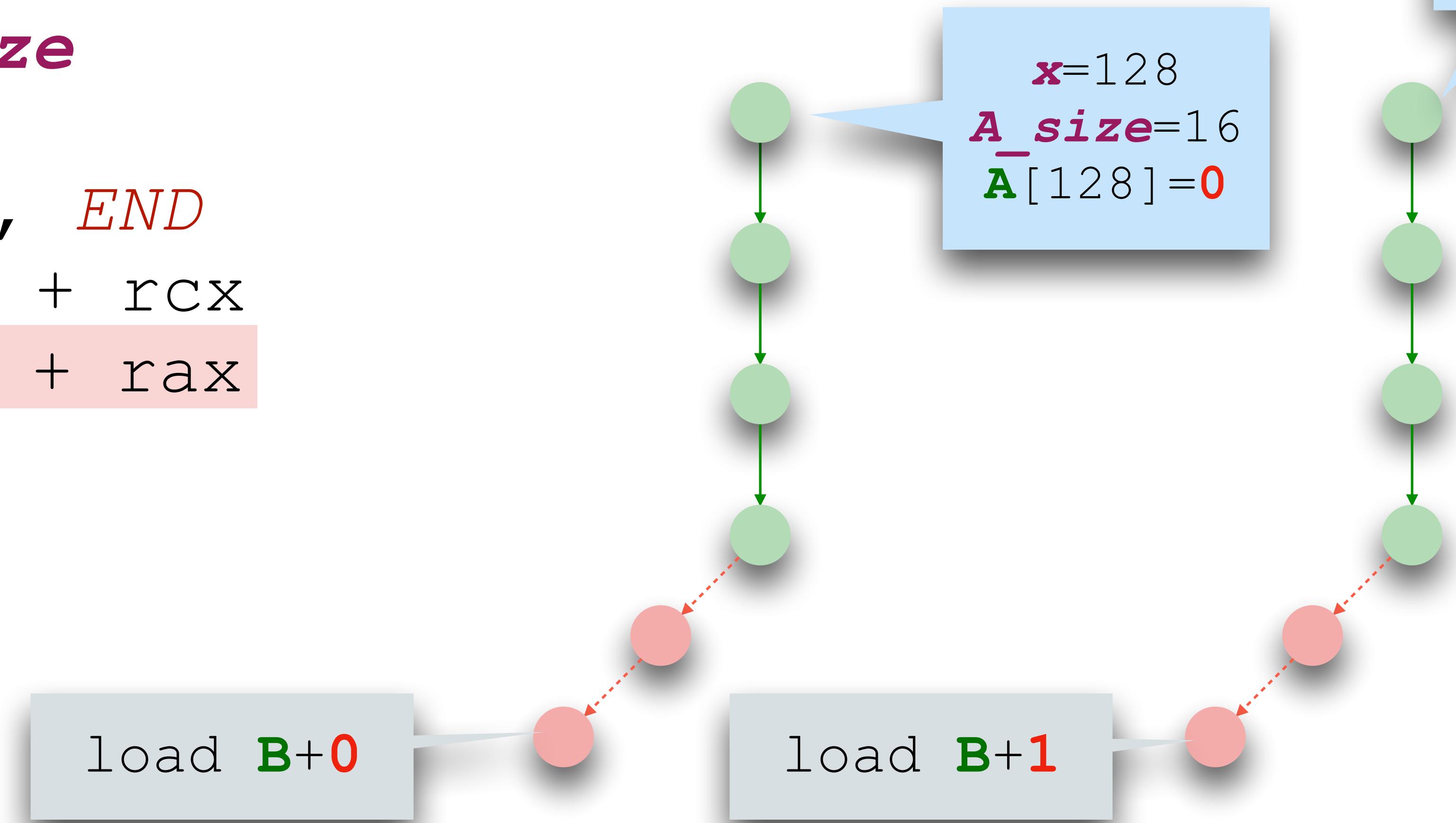
```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



Speculative non-interference

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

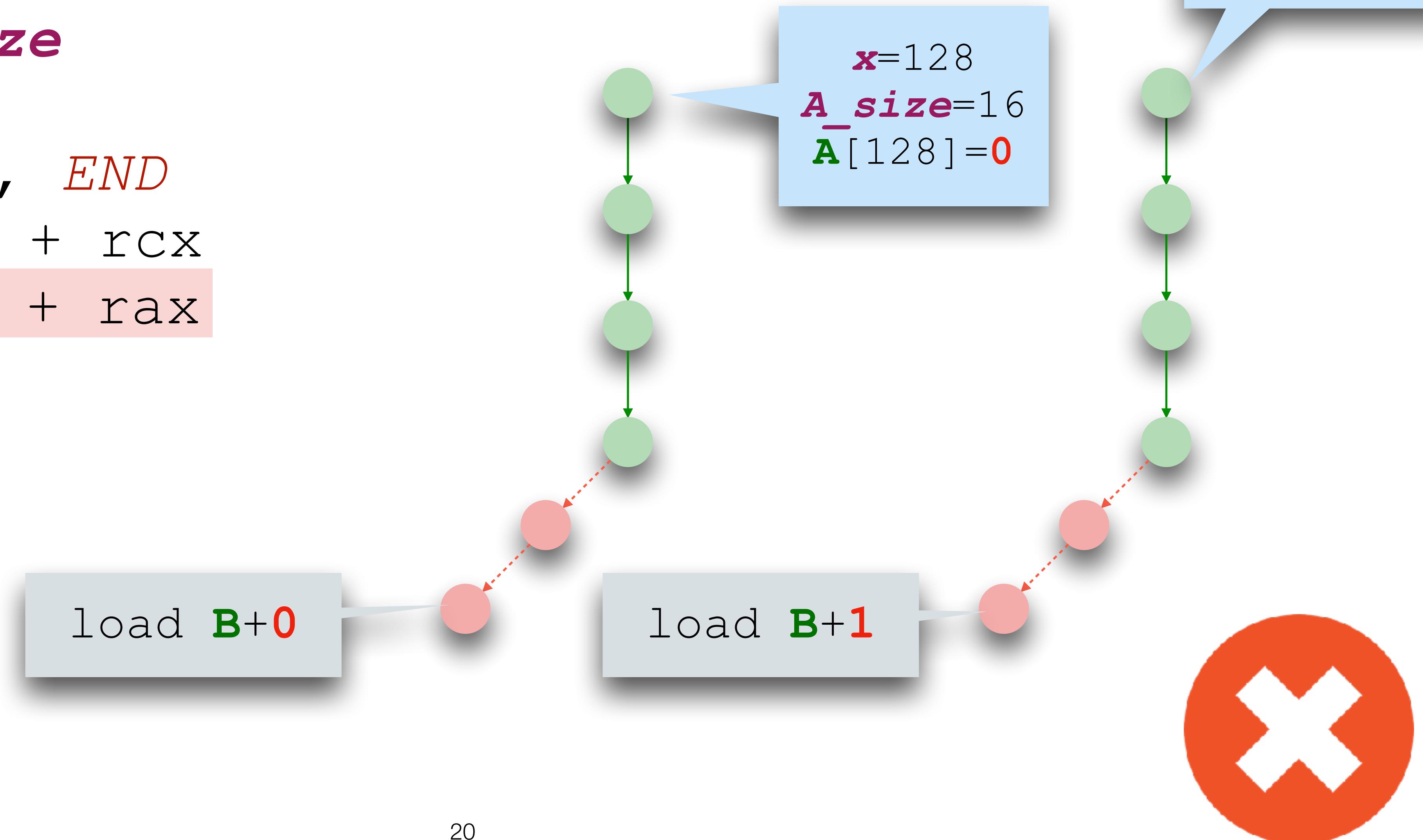
```
rax <- A_size
rcx <- x
jmp rcx≥rax, END
L1: load rax, A + rcx
      load rax, B + rax
END:
```



Speculative non-interference

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

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



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

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

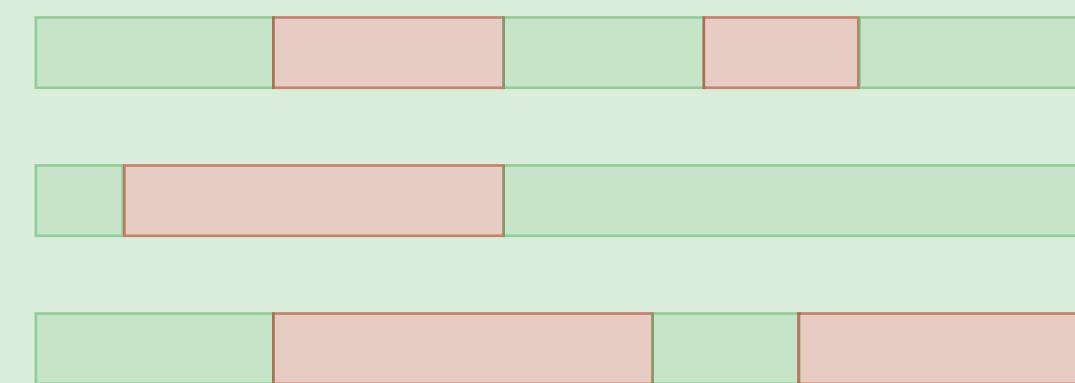
then **P** satisfies **SNI** w.r.t. all **O**

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



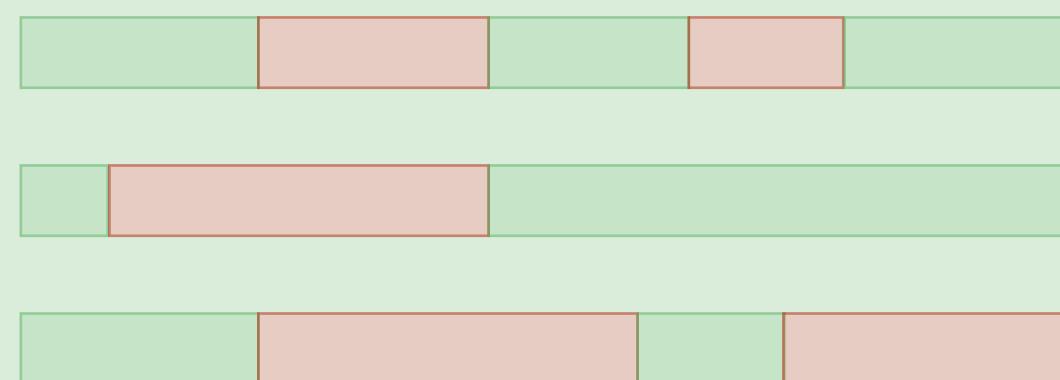
Detect leaks



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

Symbolic execution

- Program analysis technique

Experiments with a symbolic evaluation system

SIAM E. HOWDEN
IBM Thomas J. Watson Research Center

Programming Languages
Symbolic Execution and Program Testing
B. Webbrait
Editor

James C. King
IBM Thomas J. Watson Research Center

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 to prove things about symbolic expressions, a simple program testing manager, and a program verifier. A brief discussion of the relationship between symbolic execution and program verification is also included. It execution and program verification, symbolic expression and key words and phrases: symbolic execution, program testing, program debugging, program proving, program verification, symbolic interpretation.

CR Categories: 4.13, 5.21, 5.24

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 in the immediate future.

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 formally proves that the program meets its specification for all executions without being required to execute the program at all. To do this he gives a precise specification for the correct program to show that the program and the specification are consistent. The confidence in this formal proof procedure is based on the fact that the specification is correct and the accuracy employed in both the creation of the specification and the construction 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 an 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 the dependence of 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 variables it as consisting of the interpolation process, routes it as consisting of the interpolation process, each 48 are supposed to compute a set of 48 are given in terms of y_1 and A_1 .

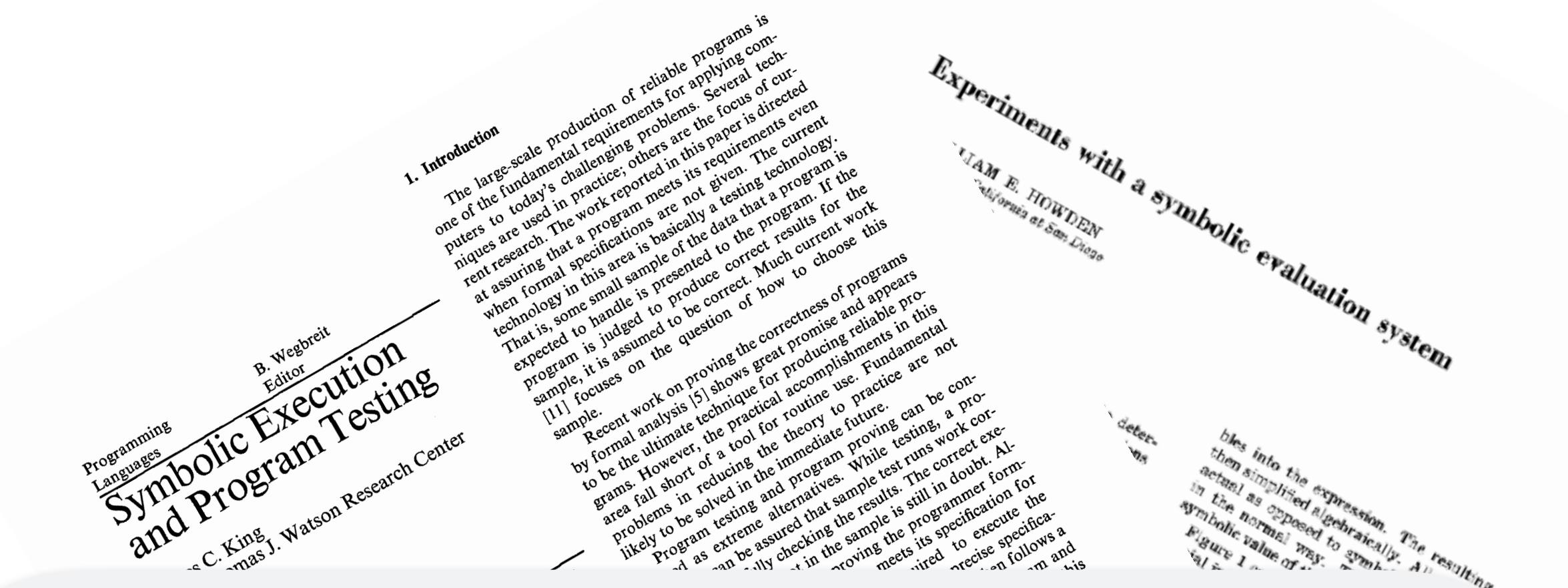
July 1976
Volume 19
Number 7

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 to the author that reprinting privileges were granted by permission of the Association for Computing Machinery. P.O. Box 218, Yorktown Heights, N.Y. 10598.

385

Symbolic execution

- Program analysis technique
- Execute programs over symbolic values



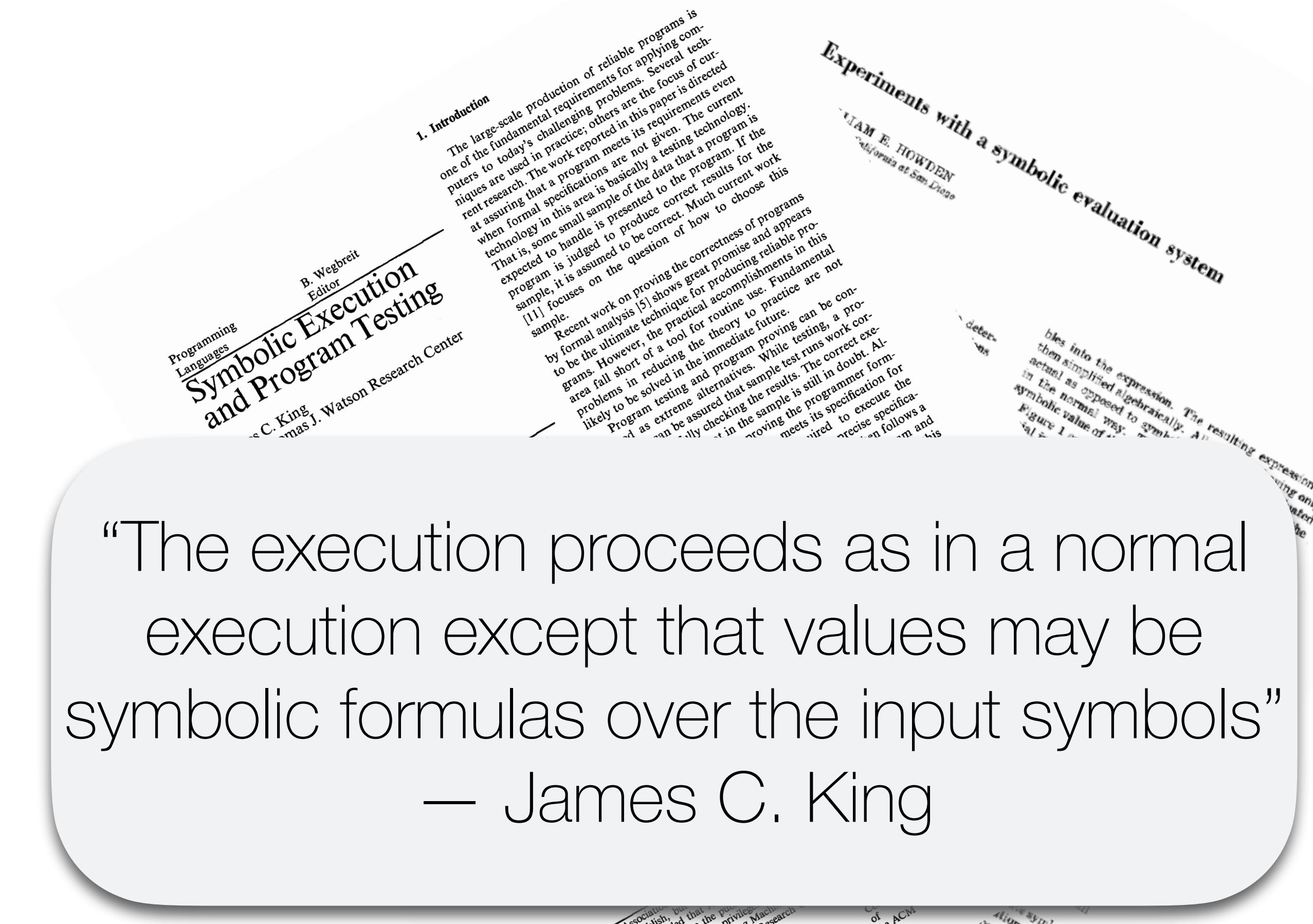
“The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols”
— James C. King

Copyright © 1976 Association
of Computing Machinery, Inc.
General permission to republish
is given and to the extent that
the original source is given,
by permission of the author or
by permission of the Association for
Computing Machinery.
P.O. Box 218, Yorktown Heights, N.Y. 10598.

385

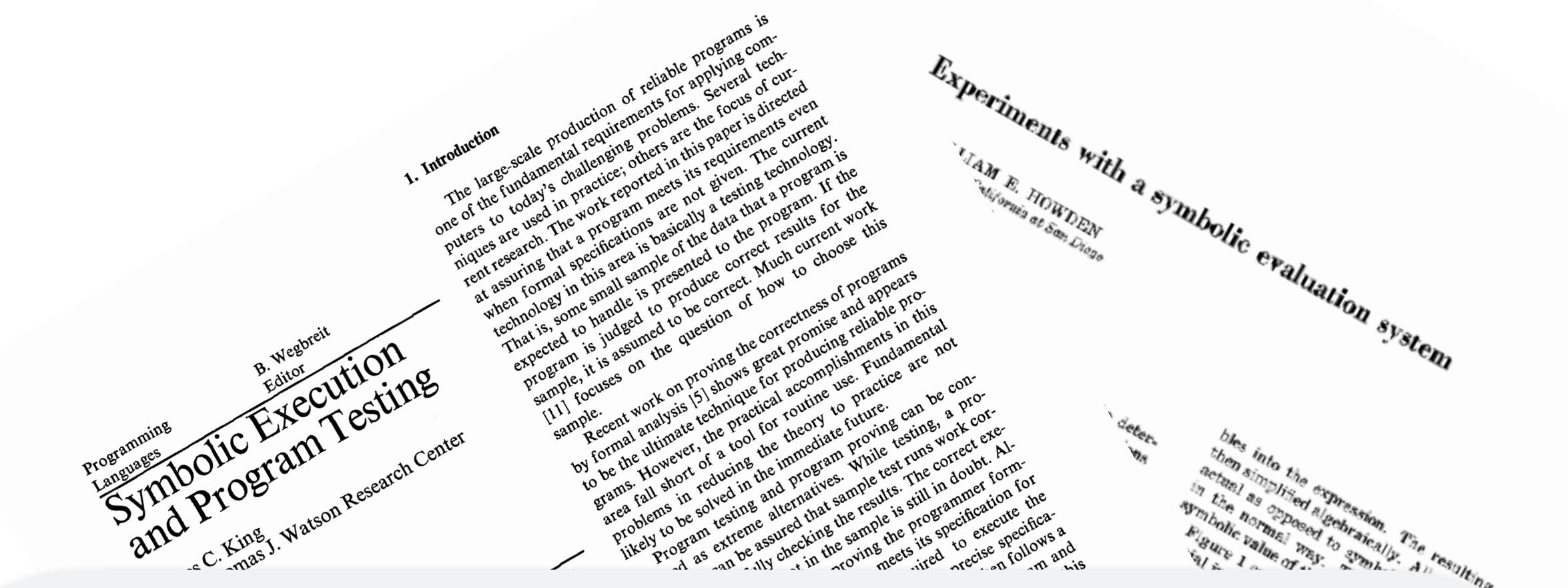
Symbolic execution

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



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

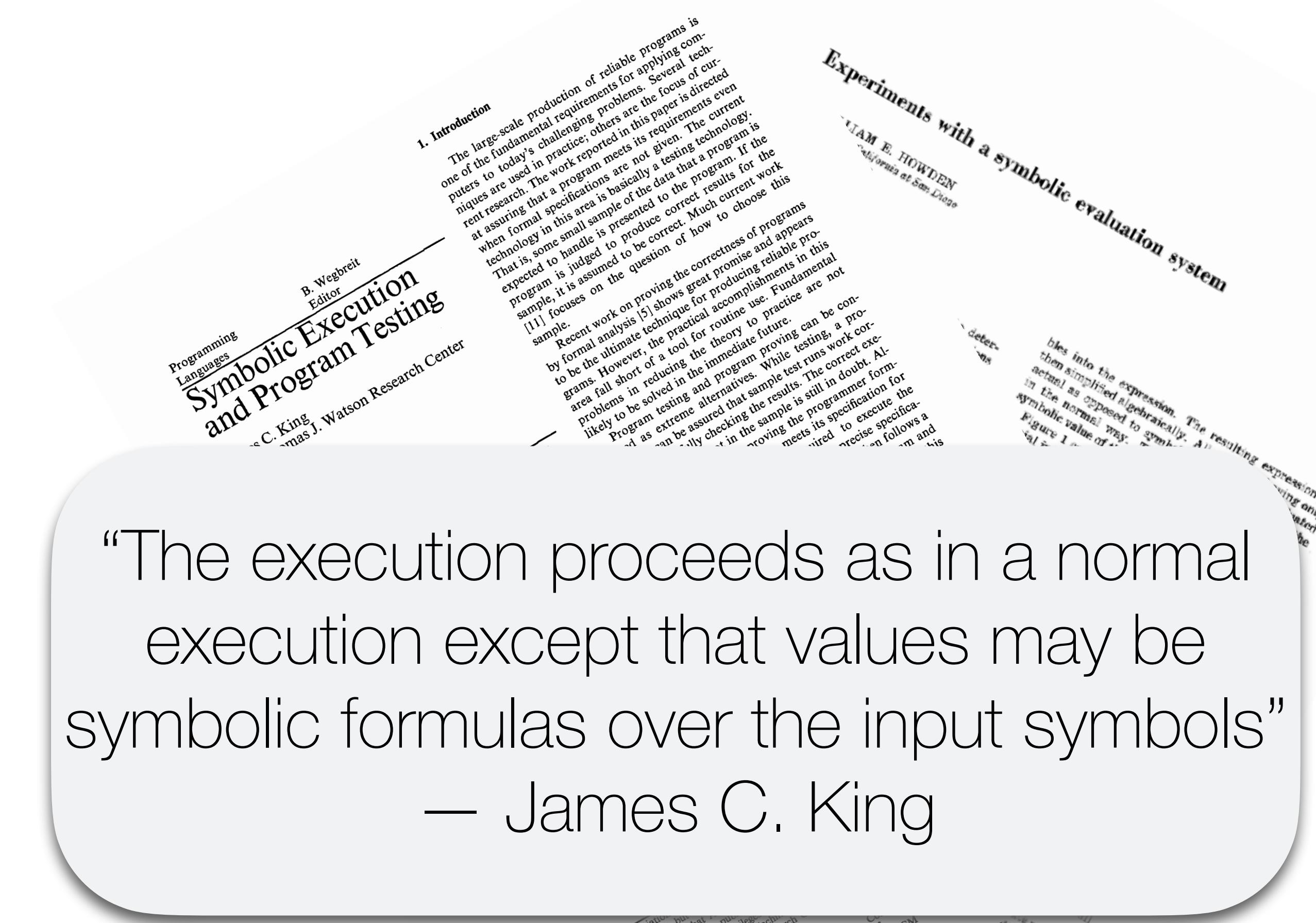


“The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols”
— James C. King

Copyright © 1976 Association
of Computing Machinery, Inc.
General permission to republish
is given and to the extent that
the original source is given,
by permission of the author or
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
 - Each path represents all concrete executions satisfying the constraint
 - Branch and jump instructions: fork paths and update path constraint



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:
```

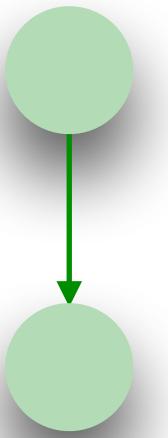


Always mispredict
branch instructions

Symbolic execution

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

true

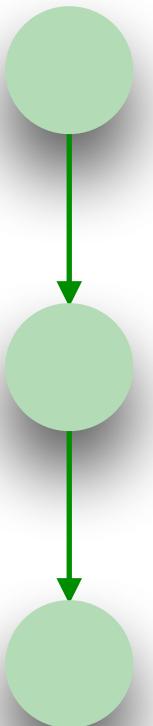


Always mispredict
branch instructions

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
```

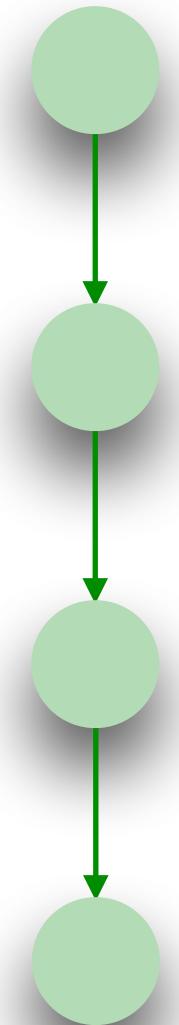
L1: load rax, **A** + rcx
load rax, **B** + rax

END:

Always mispredict
branch instructions



true



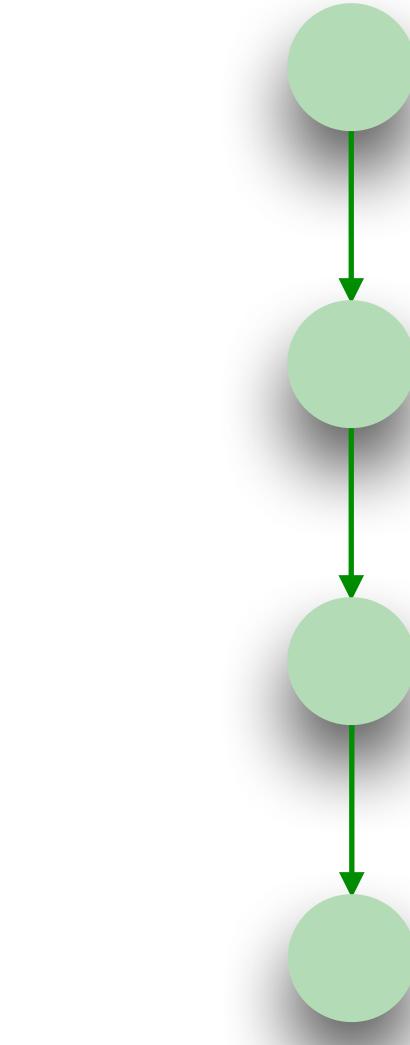
Symbolic execution

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

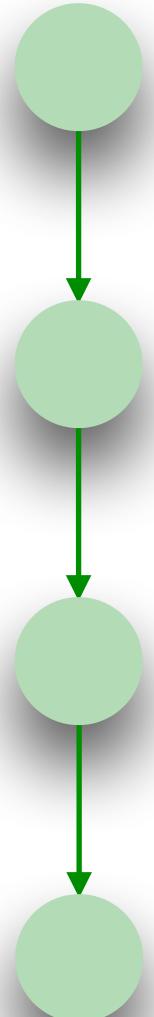


Always mispredict
branch instructions

$x \geq A_size$



$x < A_size$

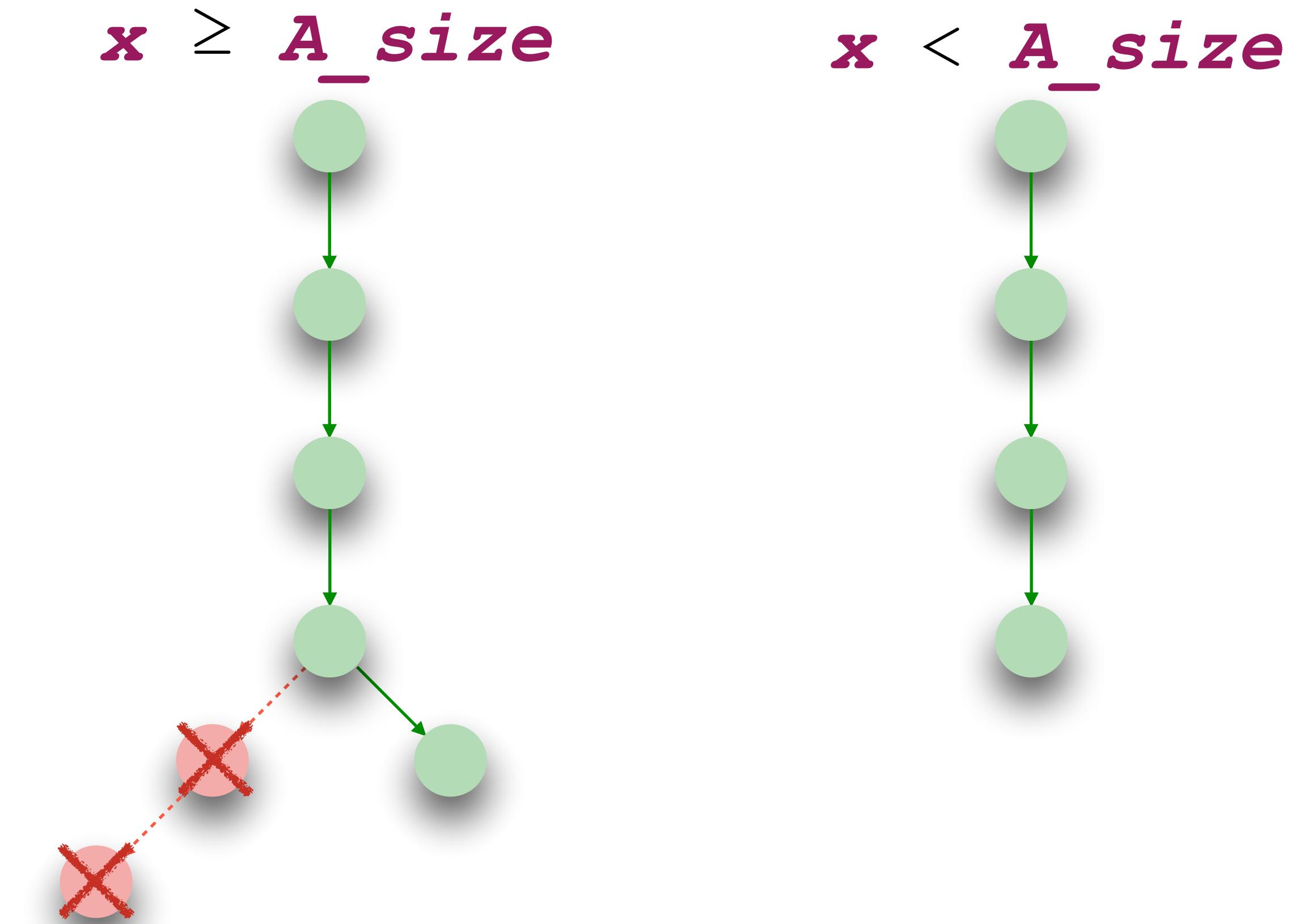


Symbolic execution

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



Always mispredict
branch instructions

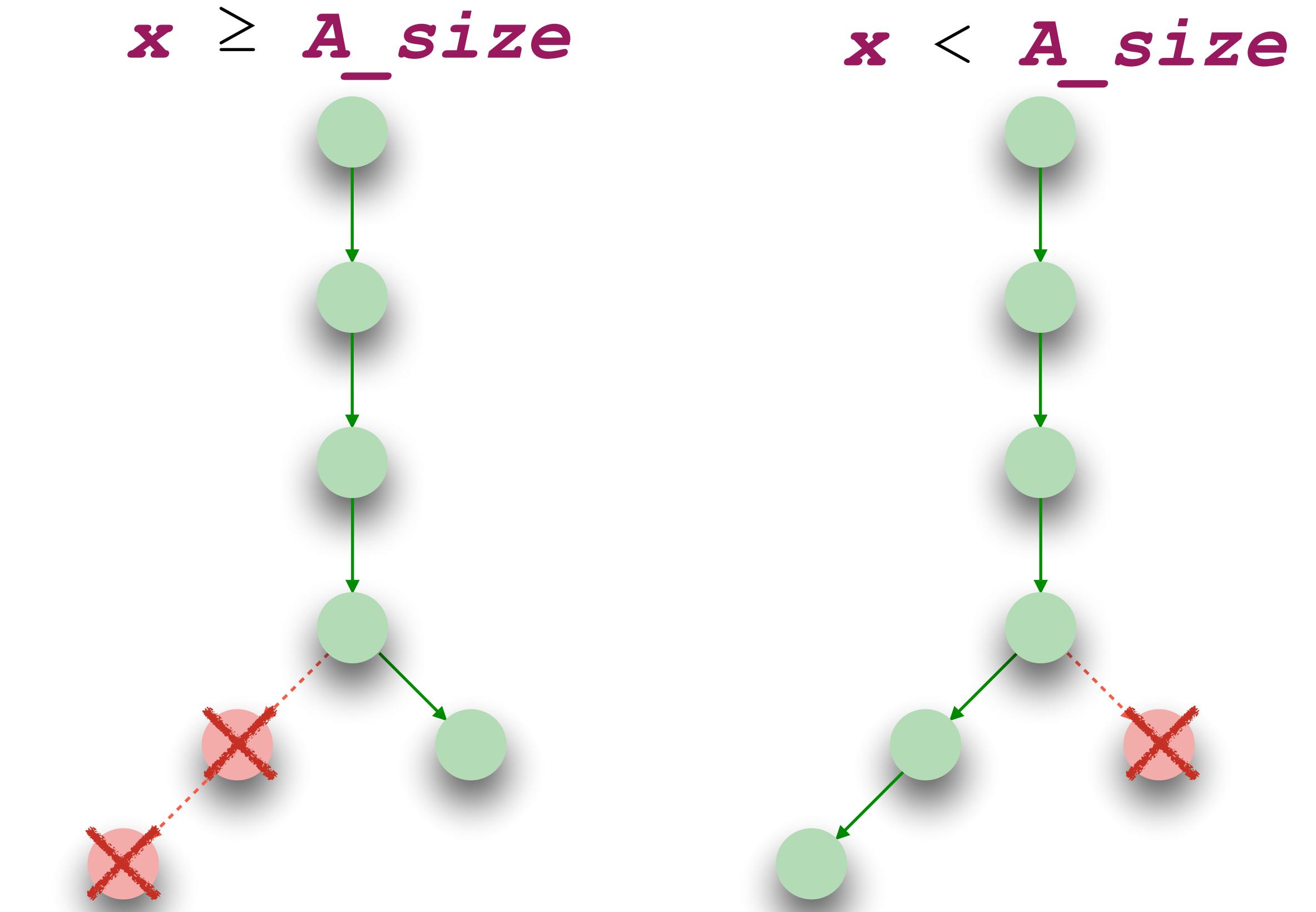


Symbolic execution

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



Always mispredict
branch instructions

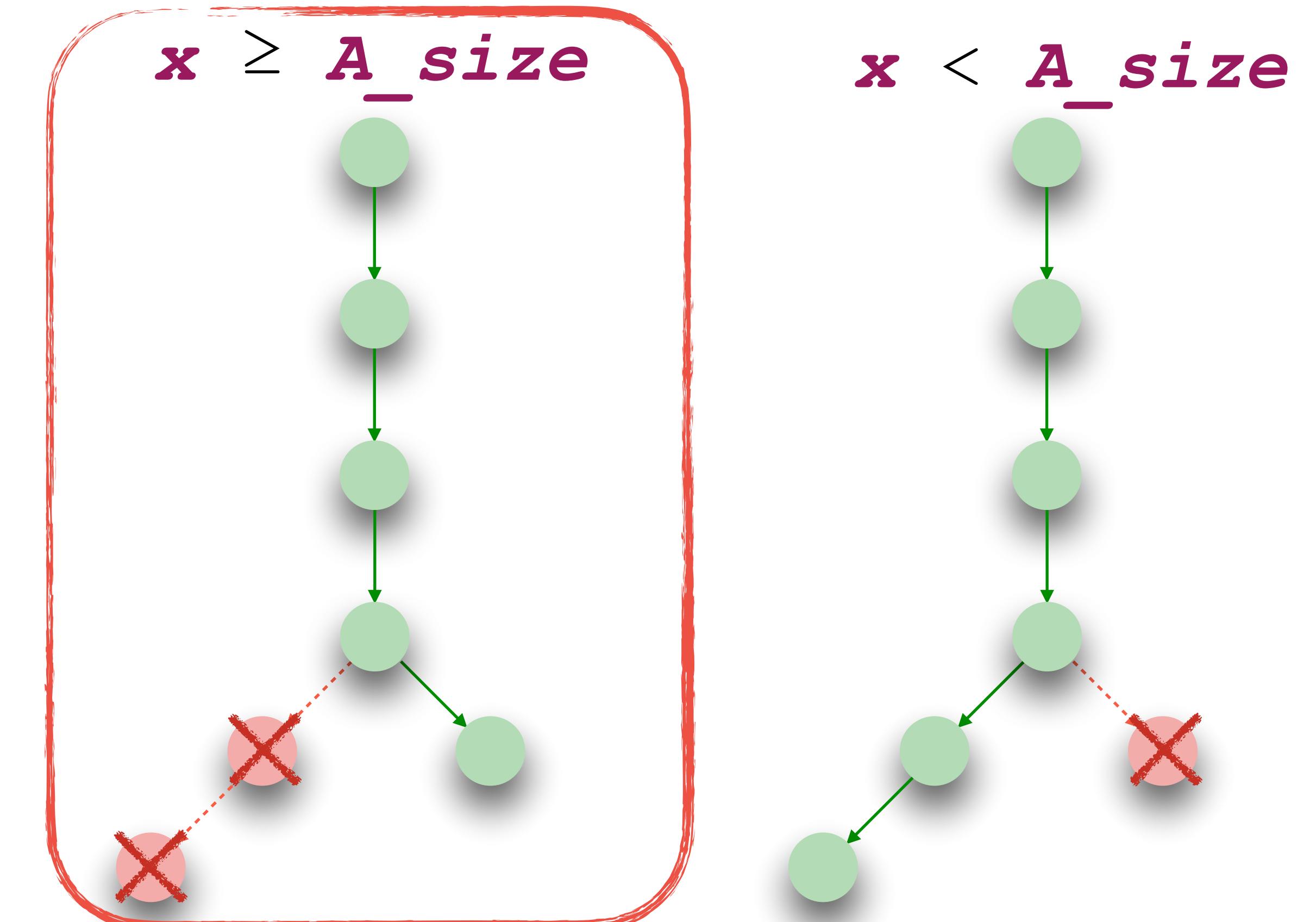


Symbolic execution

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



Always mispredict
branch instructions

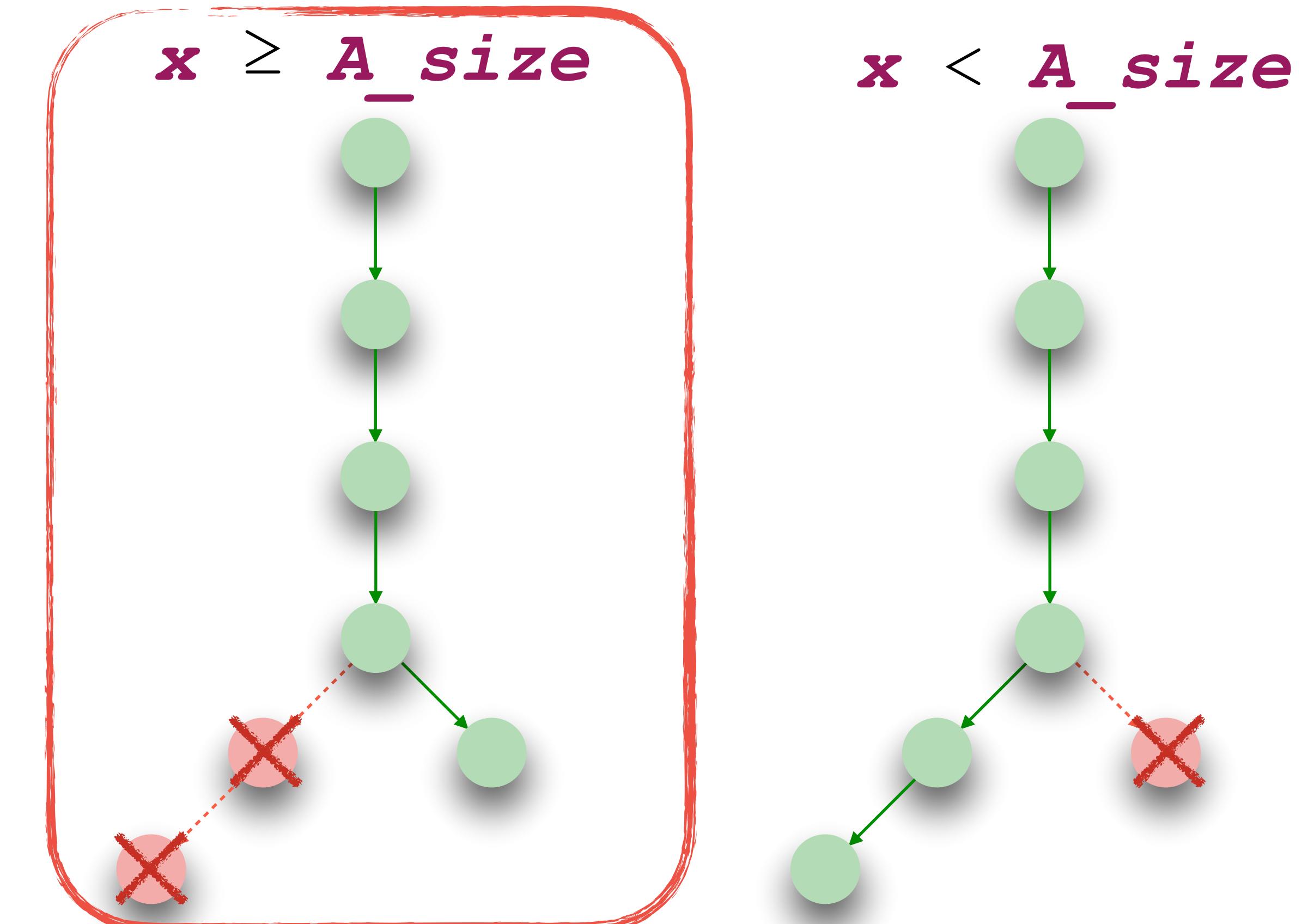


Symbolic execution

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



Always mispredict
branch instructions



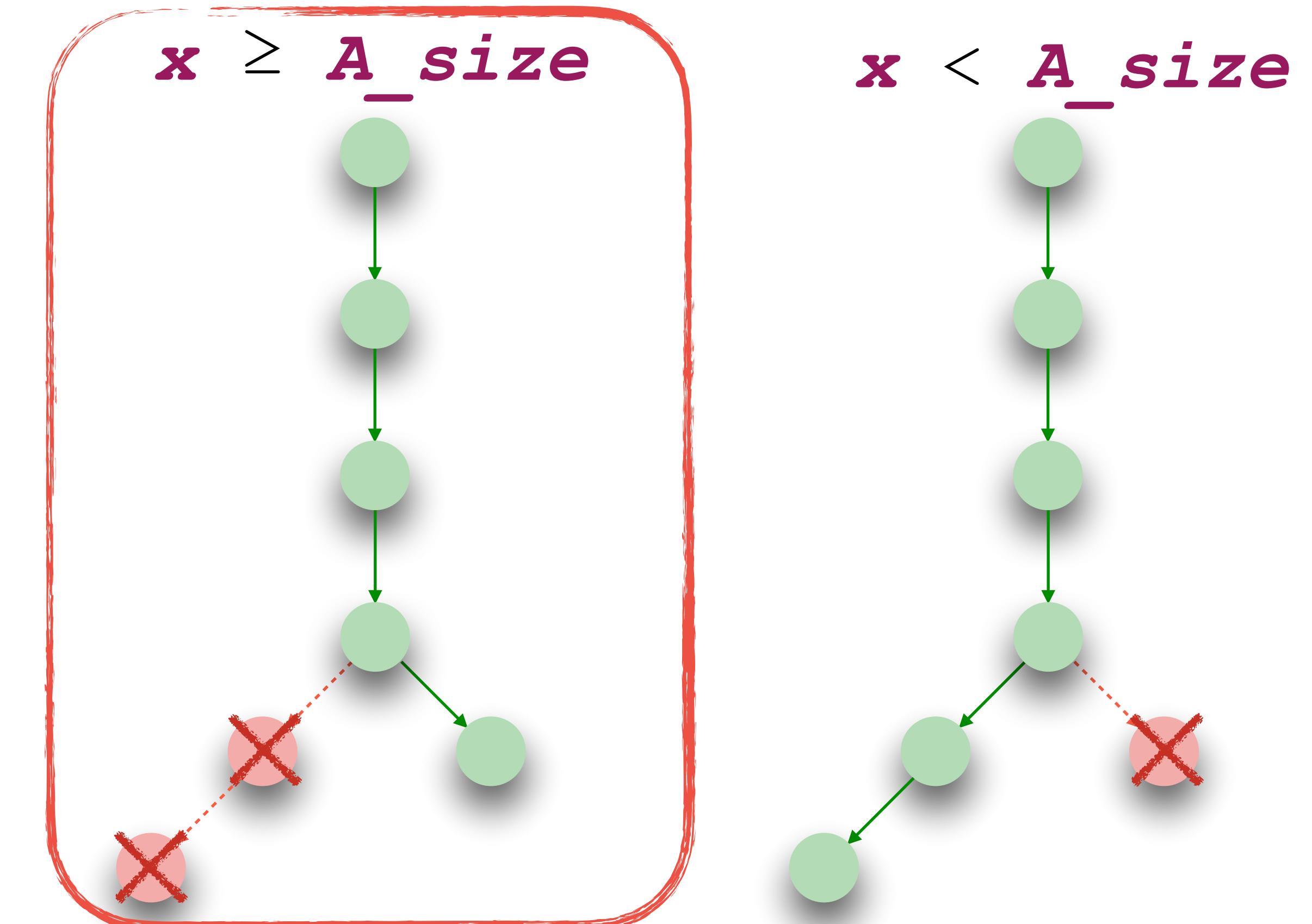
start pc L1 load A+x load B+A[x] rollback pc END

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



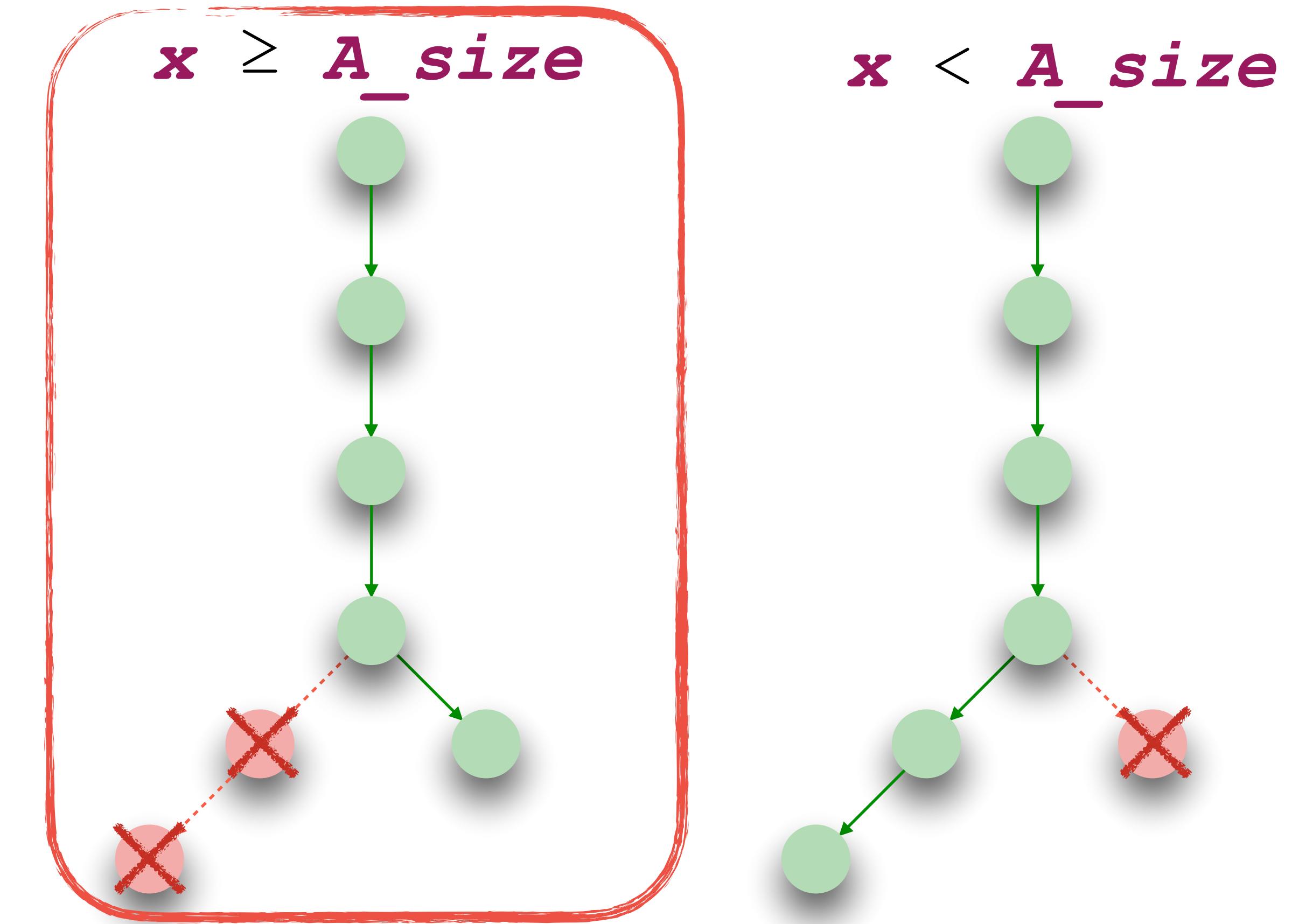
start pc L1 load A+x load B+A[x] rollback pc END

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

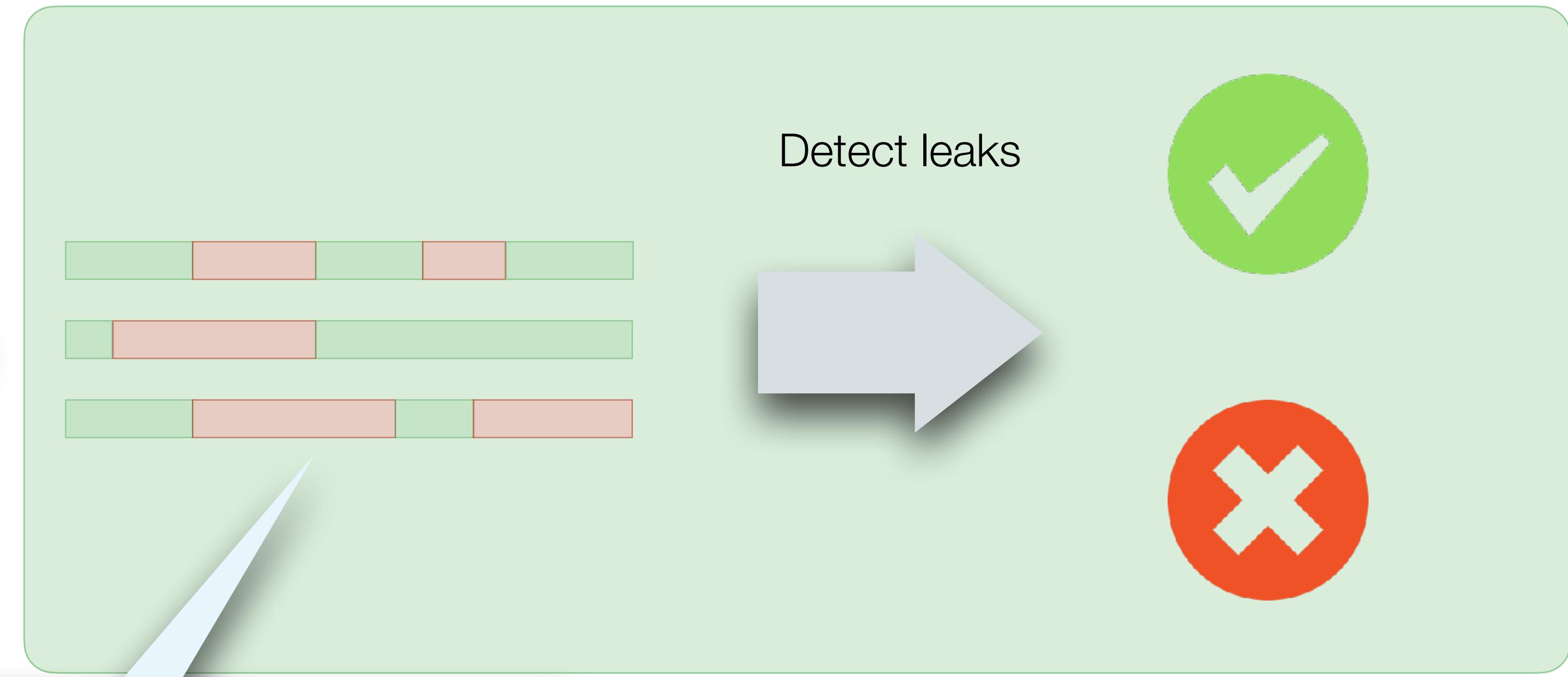


start	pc	L1	load A+x	load B+A[x]	rollback	pc	END
-------	----	----	----------	-------------	----------	----	-----

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

τ

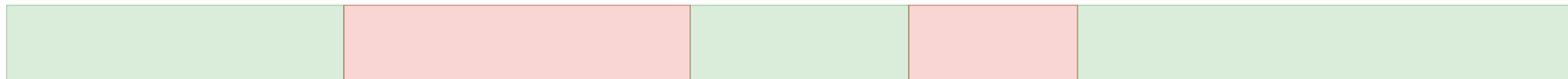


Memory leaks

Speculative memory accesses **must** depend only on

- Non-sensitive information
- Non-speculative observations

τ



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

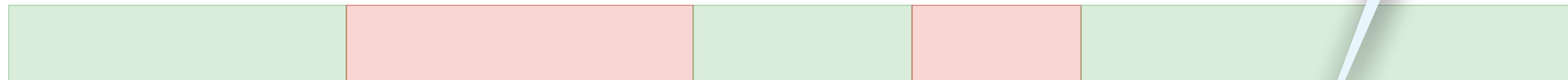
Memory leaks

Speculative memory accesses **must** depend only on

- Non-sensitive information
- Non-speculative observations

Check with self-composition

τ



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

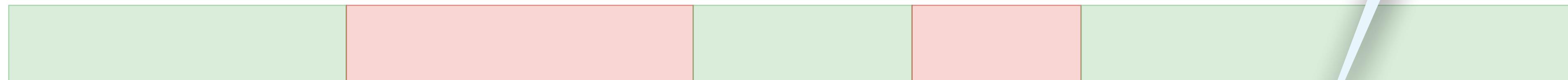
Memory leaks

Speculative memory accesses **must** depend only on

- Non-sensitive information
- Non-speculative observations

Check with self-composition

τ



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

s_1

s_2

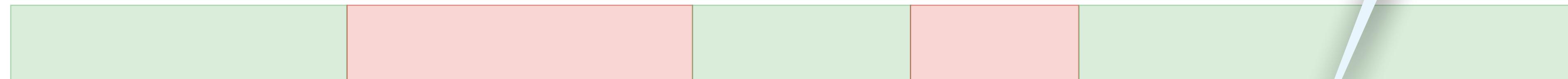
Memory leaks

Speculative memory accesses **must** depend only on

- Non-sensitive information
- Non-speculative observations

Check with self-composition

τ



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

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

τ



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

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

τ



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

Equivalent
wrt **policy**

$$s_1 \models \varphi$$



$$s_2 \models \varphi$$

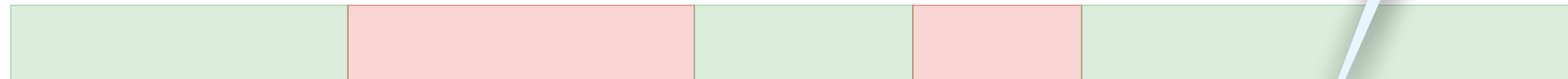
Memory leaks

Speculative memory accesses **must** depend only on

- Non-sensitive information
- Non-speculative observations

Check with self-composition

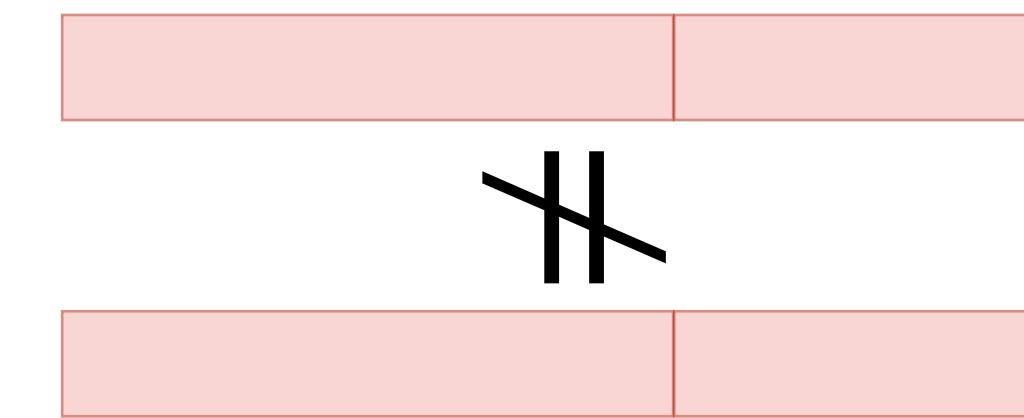
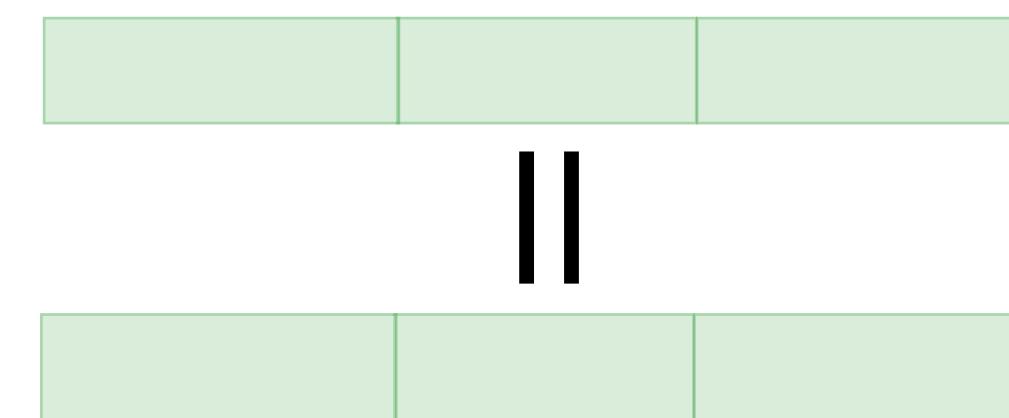
τ



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

Equivalent
wrt **policy**

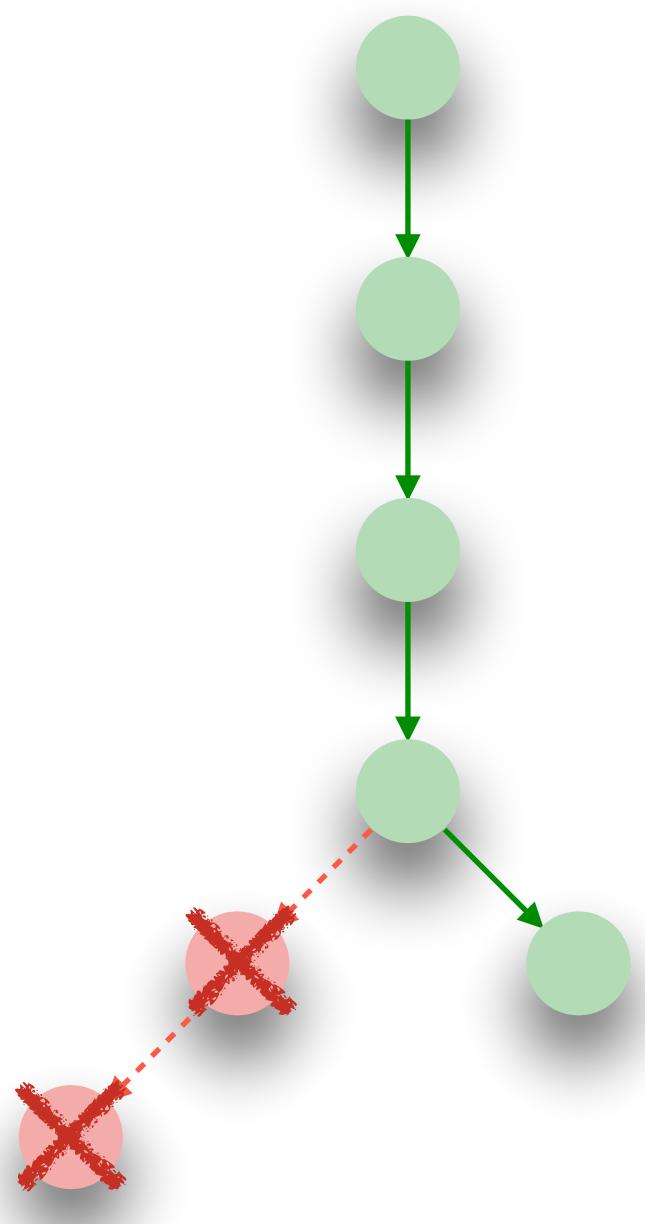
$$\begin{array}{c} s_1 \models \varphi \\ s_2 \models \varphi \end{array}$$



Memory leaks

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

END:



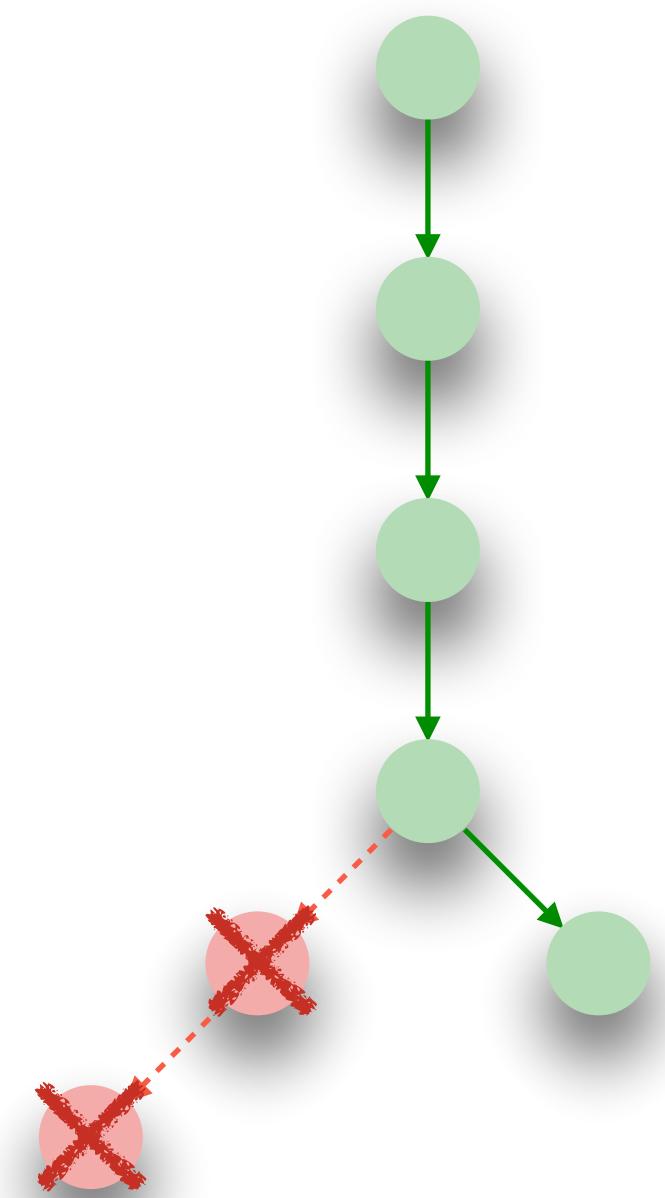
Policy
x, A_size, A, B
are public

start pc <i>L1</i>	load A+x	load B+A[x]	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

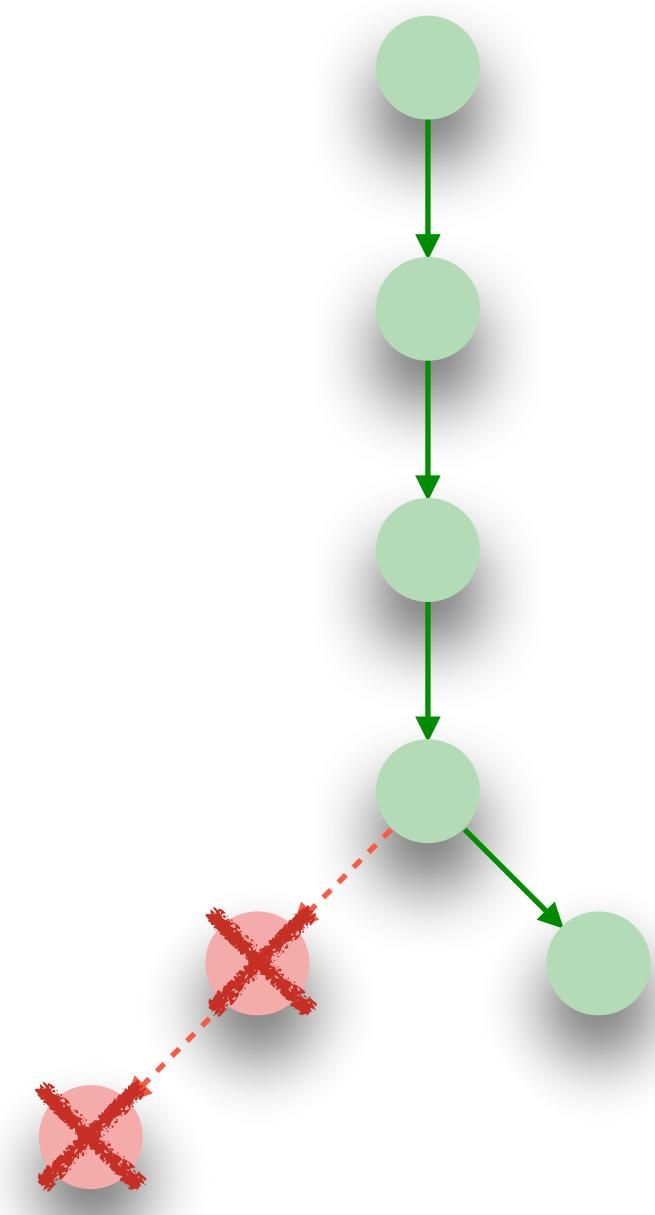
start pc <i>L1</i> load A+x load B+A[x] rollback	pc <i>END</i>
--	---------------

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

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

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

s₁

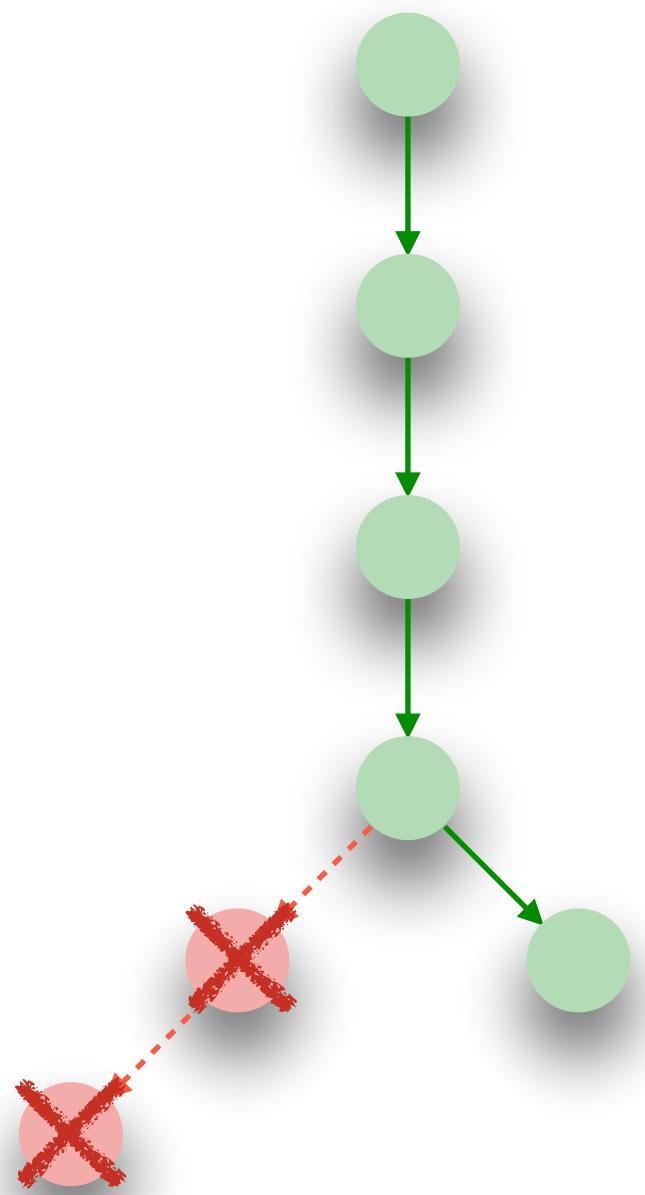
s₂

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

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

S₁

S₂

$$\mathbf{x_1=x_2} \wedge \mathbf{A_size}_1=\mathbf{A_size}_2 \wedge \mathbf{A}_1=\mathbf{A}_2 \wedge \mathbf{B}_1=\mathbf{B}_2$$

Memory leaks

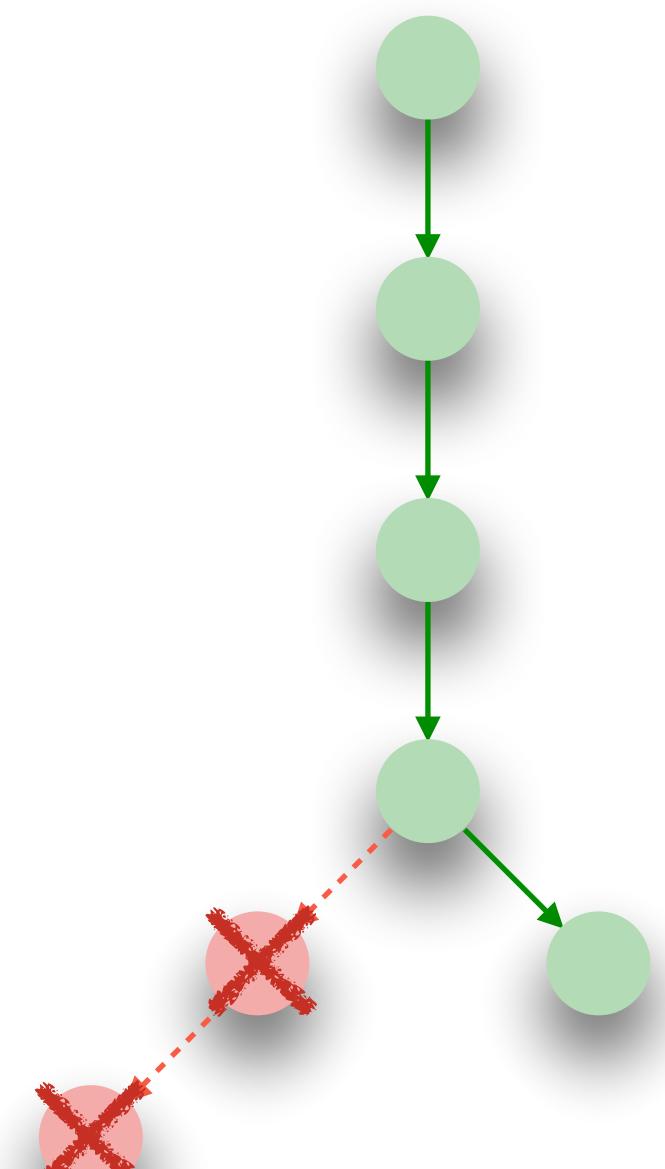


```

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

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

$$S_1 \models \mathbf{x}_1 \geq \mathbf{A_size}_1$$

$$S_2 \models \mathbf{x}_2 \geq \mathbf{A_size}_2$$

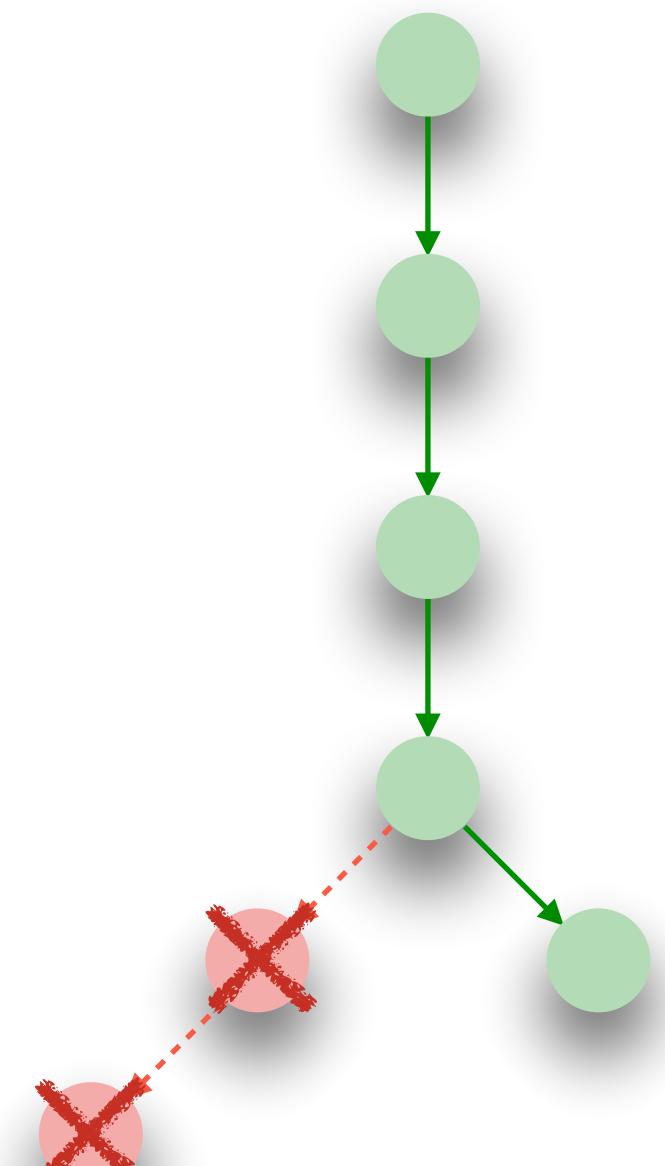
$$\mathbf{x}_1 = \mathbf{x}_2 \wedge \mathbf{A_size}_1 = \mathbf{A_size}_2 \wedge \mathbf{A}_1 = \mathbf{A}_2 \wedge \mathbf{B}_1 = \mathbf{B}_2$$

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

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

$$S_1 \models \mathbf{x}_1 \geq \mathbf{A_size}_1$$

$$S_2 \models \mathbf{x}_2 \geq \mathbf{A_size}_2$$

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

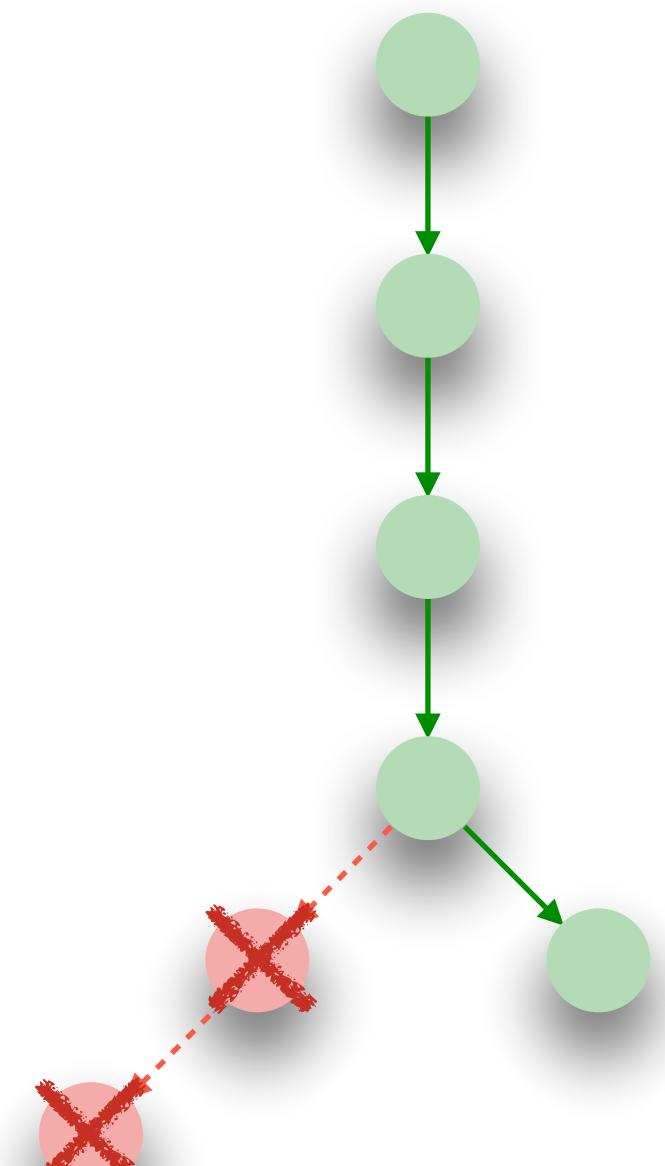
Always true!

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

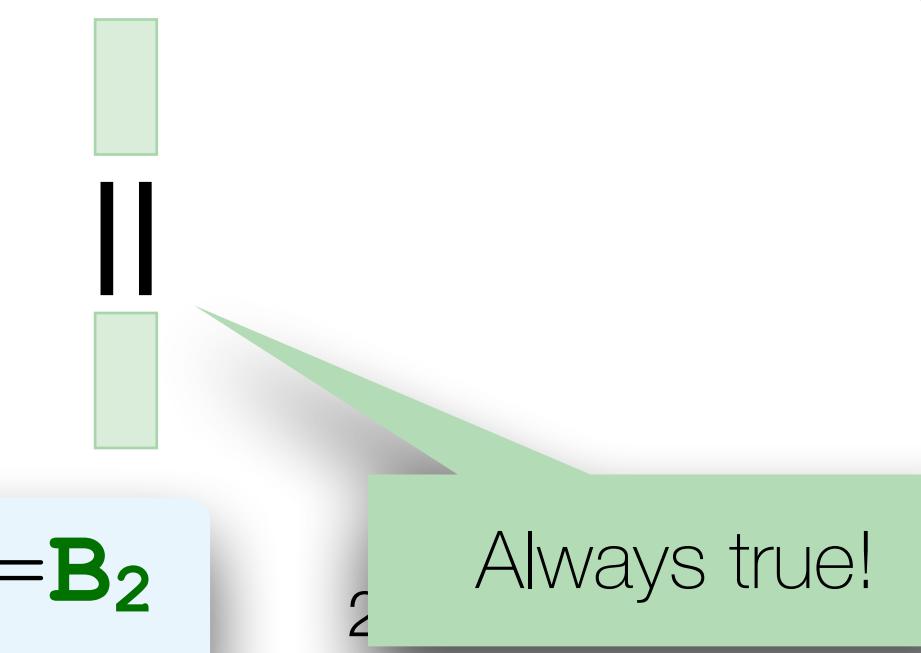
start pc <i>L1</i>	load A+x	load B+A[x]	rollback	pc <i>END</i>
--------------------	-----------------	--------------------	----------	---------------

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

$$S_1 \models \mathbf{x}_1 \geq \mathbf{A_size}_1$$

$$S_2 \models \mathbf{x}_2 \geq \mathbf{A_size}_2$$

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

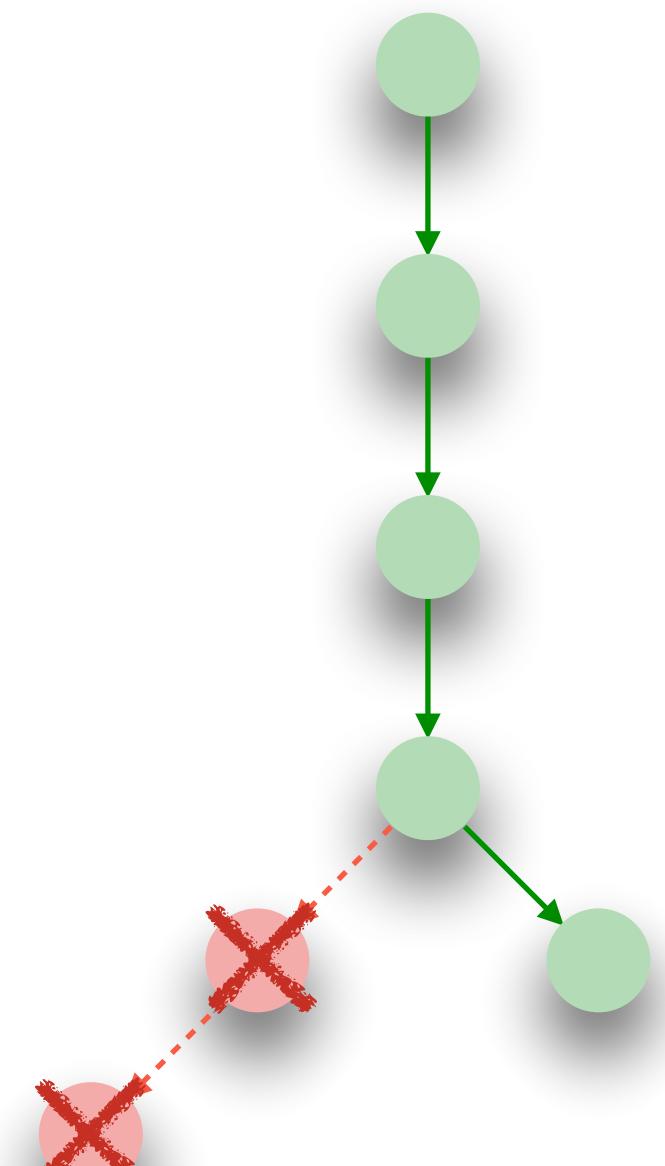


Memory leaks



```

    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

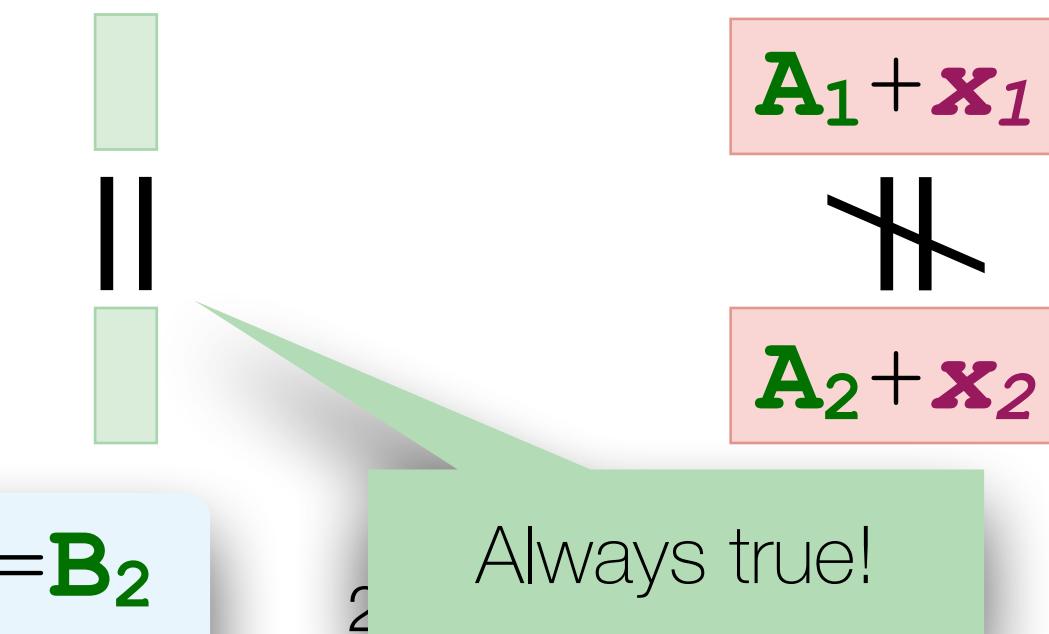
start pc <i>L1</i>	load A+x	load B+A[x]	rollback	pc <i>END</i>
--------------------	-----------------	--------------------	----------	---------------

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

$S_1 \models \mathbf{x}_1 \geq \mathbf{A_size}_1$

$S_2 \models \mathbf{x}_2 \geq \mathbf{A_size}_2$

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

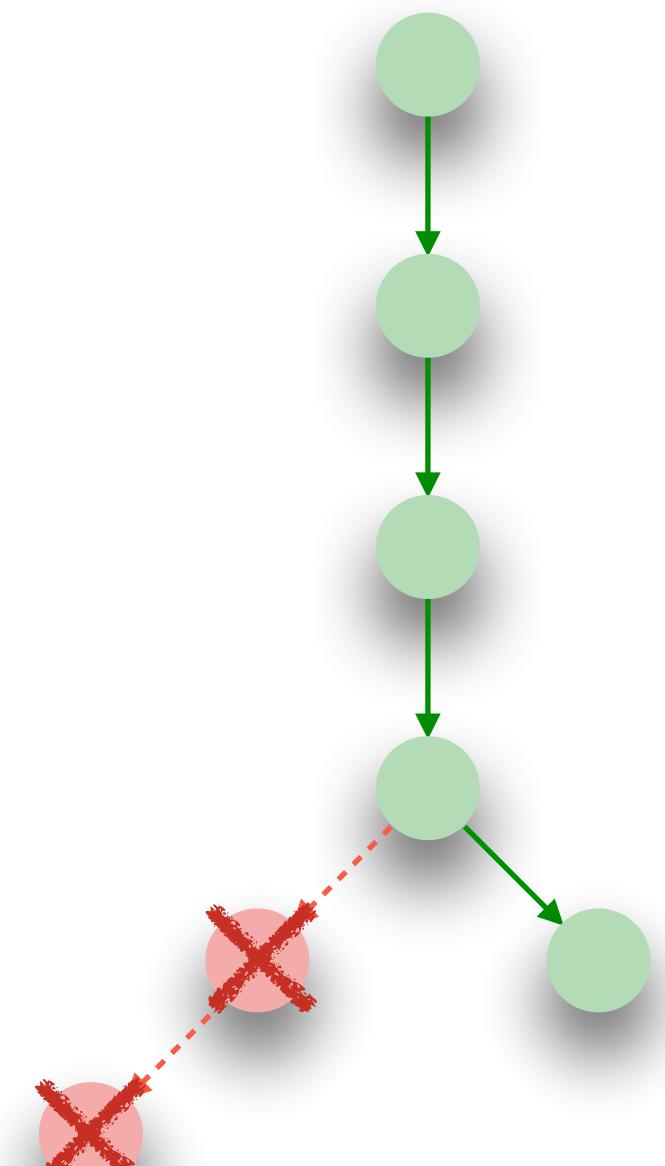


Memory leaks



```

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

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

$$S_1 \models \mathbf{x}_1 \geq \mathbf{A_size}_1$$

$$S_2 \models \mathbf{x}_2 \geq \mathbf{A_size}_2$$

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

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

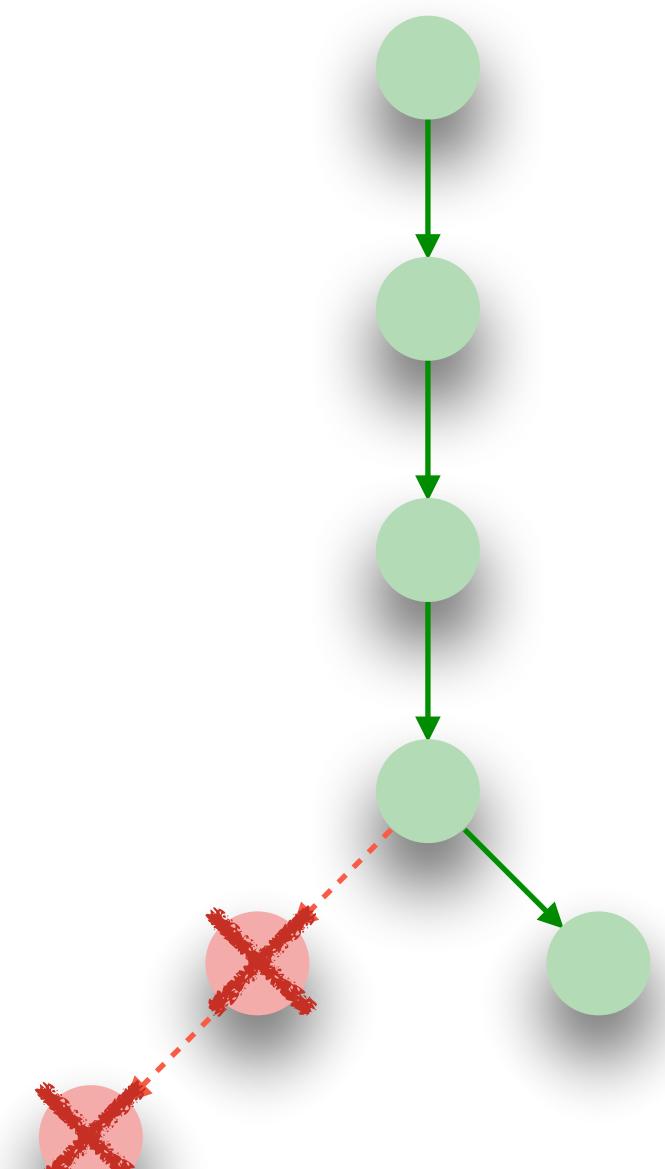
Always true!

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

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

$$S_1 \models \mathbf{x}_1 \geq \mathbf{A_size}_1$$

$$S_2 \models \mathbf{x}_2 \geq \mathbf{A_size}_2$$

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

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

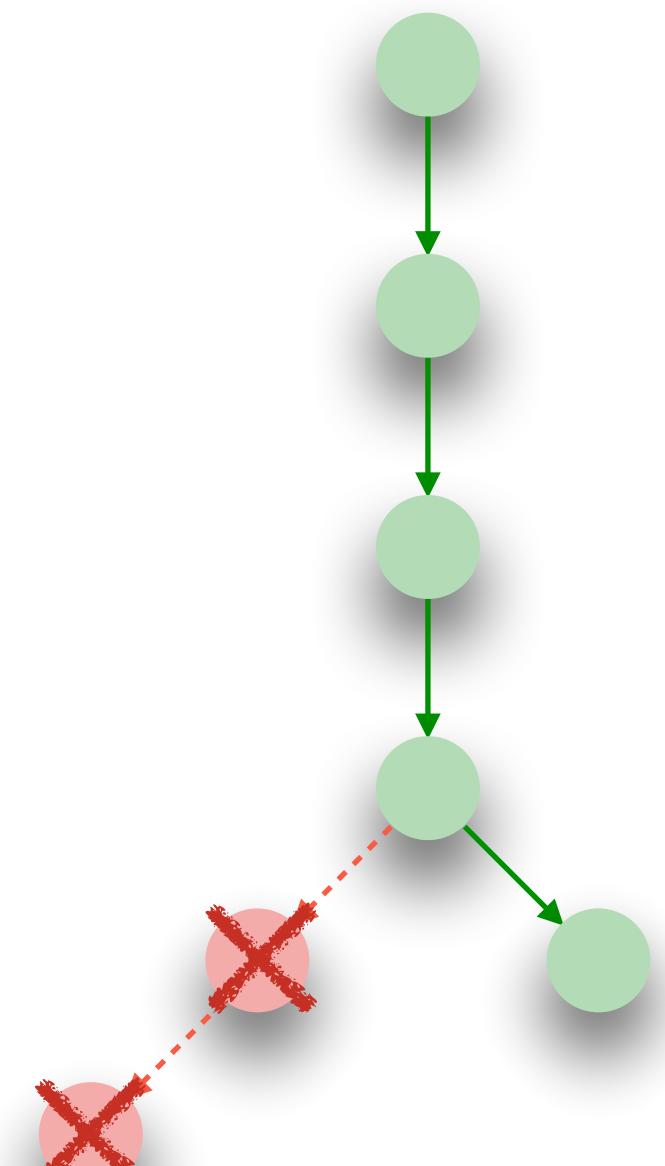
Always true!

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

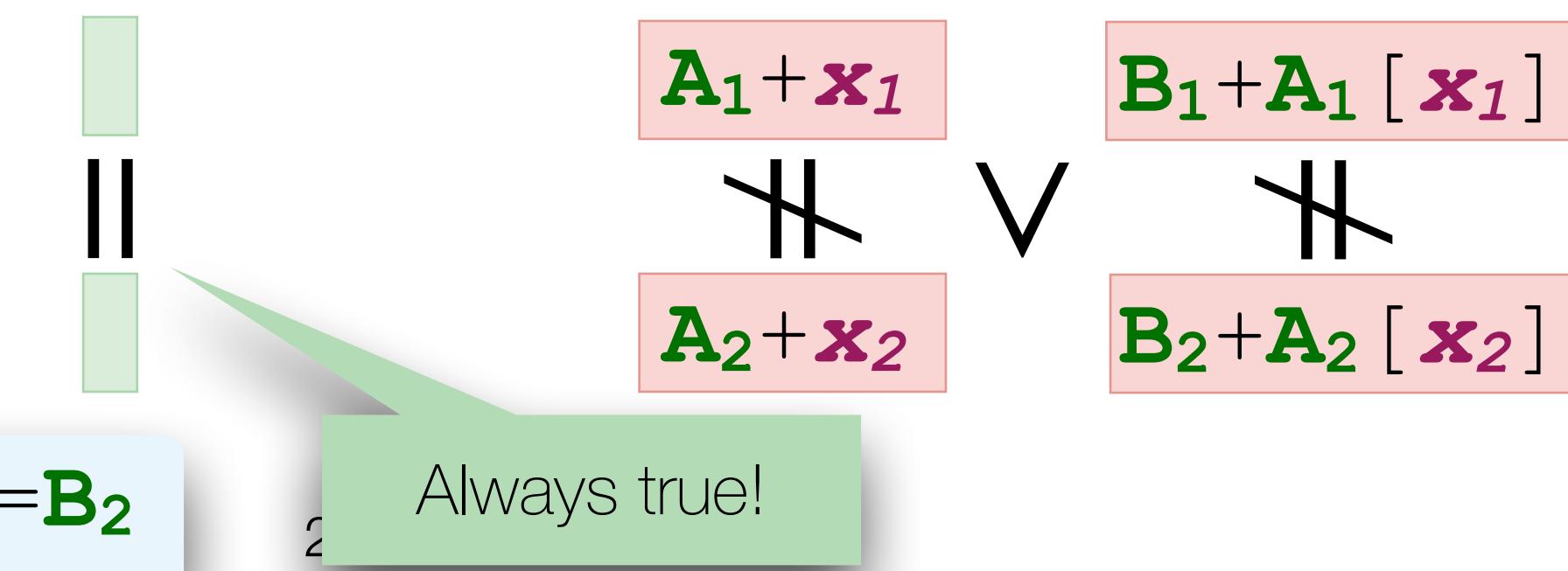
start	pc <i>L1</i>	load A+x	load B+A[x]	rollback	pc <i>END</i>
-------	--------------	-----------------	--------------------	----------	---------------

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

$$S_1 \models \mathbf{x}_1 \geq \mathbf{A_size}_1$$

$$S_2 \models \mathbf{x}_2 \geq \mathbf{A_size}_2$$

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



Spectector + Case studies

Spectector



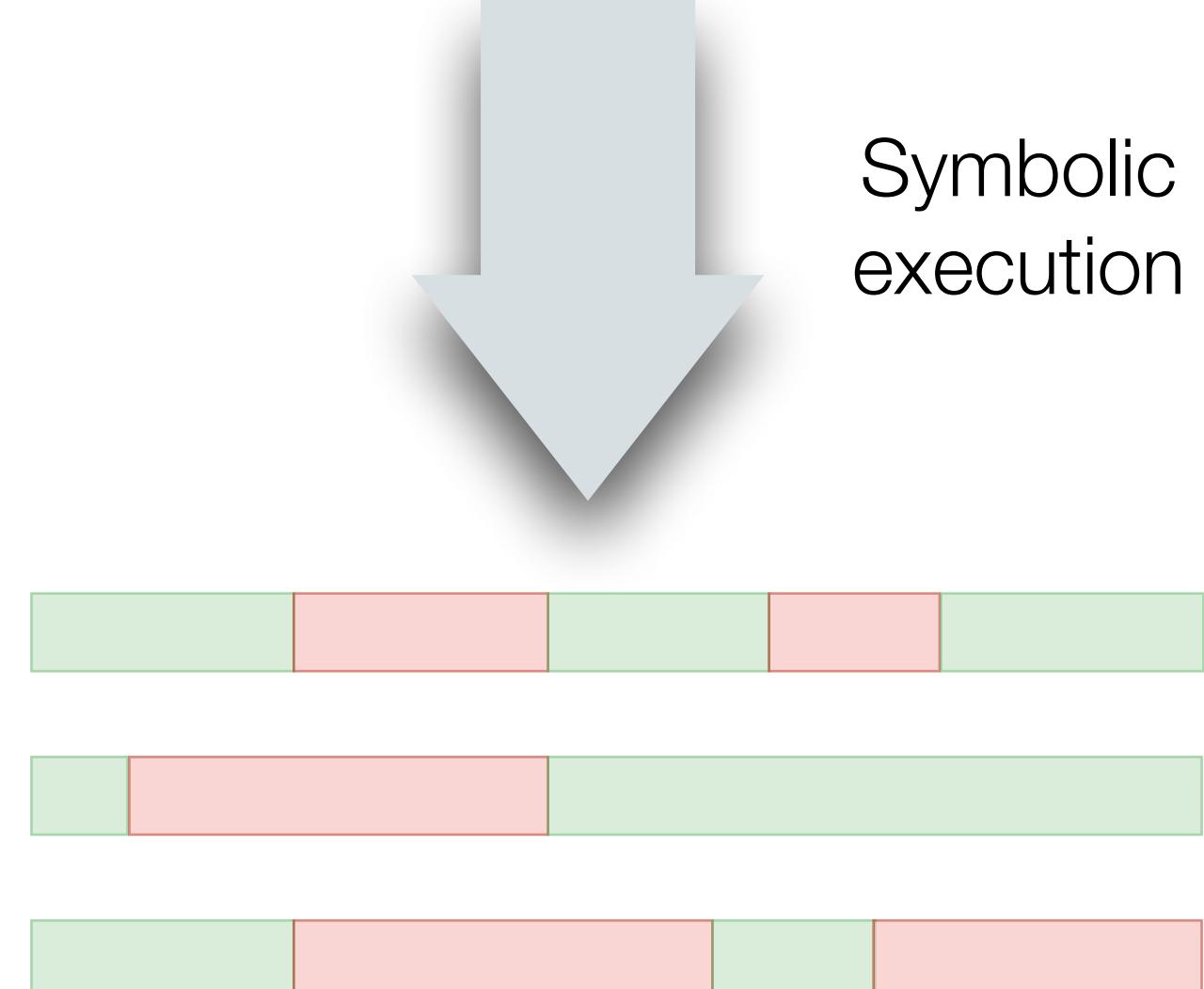
```
mov    rax, A_size
mov    rcx, x
cmp    rcx, rax
END
L1:   mov    rax, A[rcx]
      mov    rax, B[rax]
```

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



Symbolic
execution

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

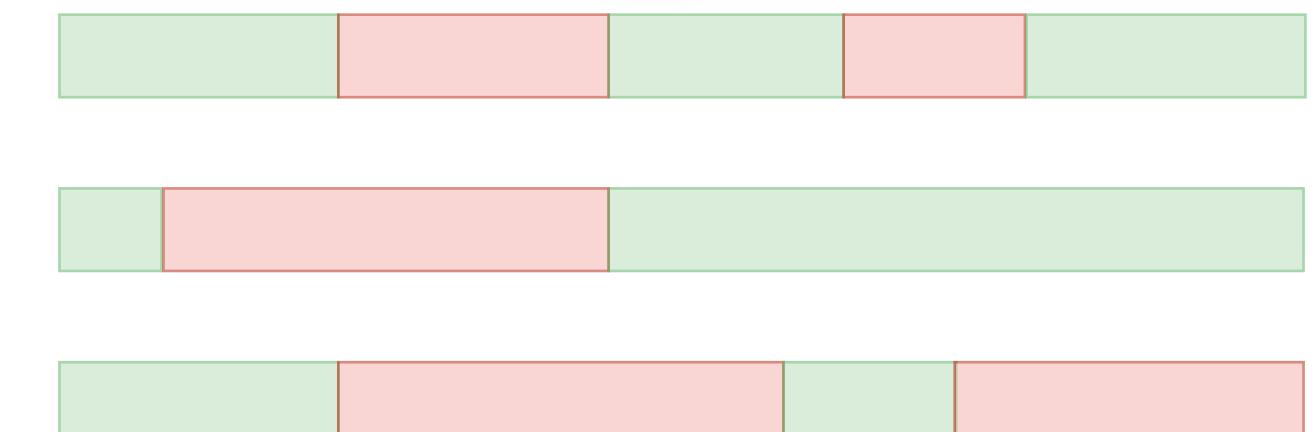


Check for speculative leaks

```
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	-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	UNP	FEN	SLH						
	-00	-02	-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	●	●	●	●	●	●	●
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	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	●	●	●	●	o	●

Results

Speculative load
hardening

Ex.	VCC						ICC						CLANG					
	UNP	FEN	19.15	FEN	19.20	UNP	FEN	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		
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

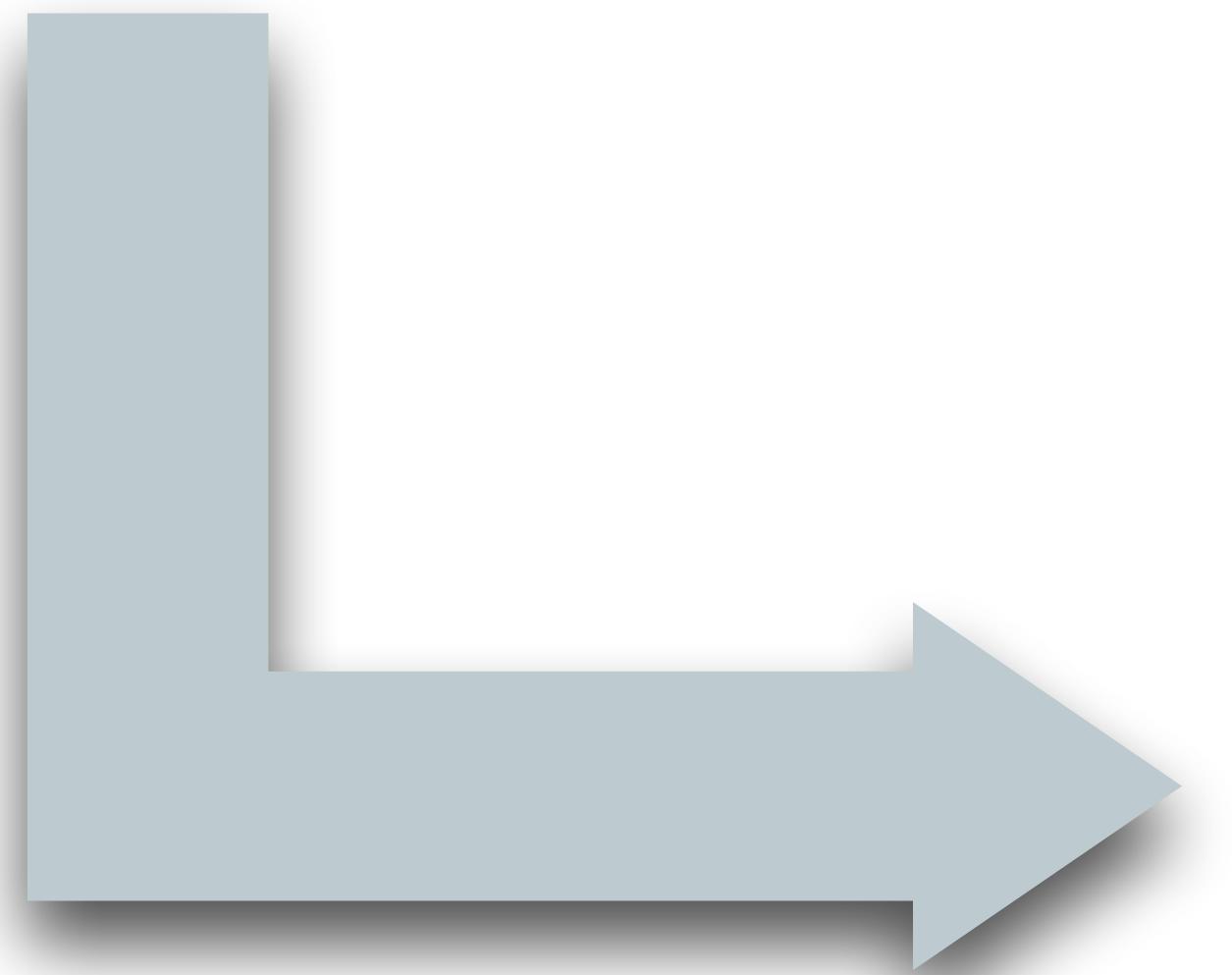
- Leaks in all unprotected programs (except example #08 with optimizations)
- Confirm all vulnerabilities in VCC pointed out by Paul Kocher
- Programs with fences (ICC and Clang) are secure
- Unnecessary fences
- Programs with SLH are secure except #10 and #15

Example #01 - SLH

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

Example #01 - SLH

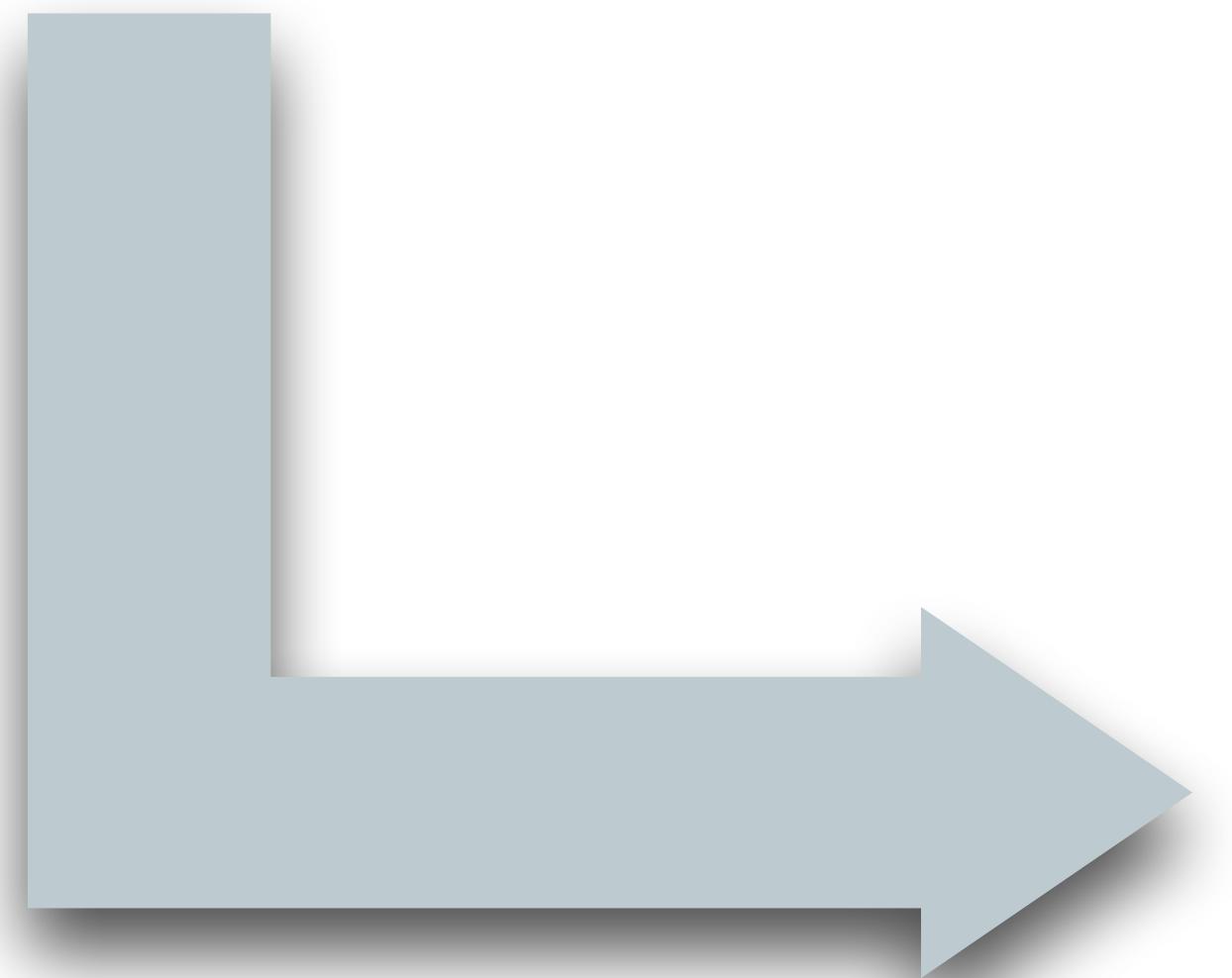
```
if (x < A_size)
    y = B[A[x] * 512]
```



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

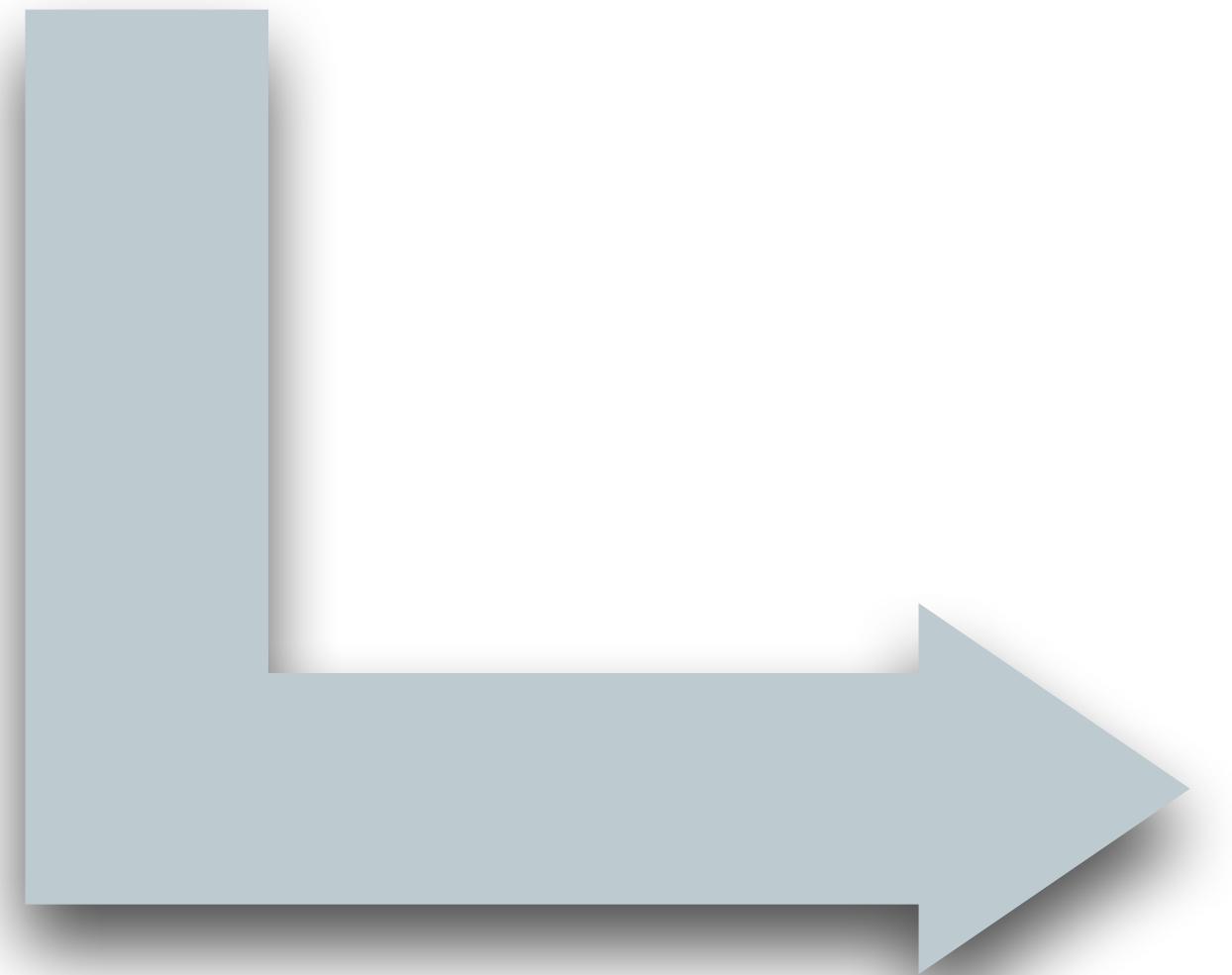
mov	rax, A_size
mov	rcx, x
mov	rdx, 0
cmp	rcx, rax
jae	<i>END</i>
cmove	-1, rdx
mov	rax, A[rcx]
shl	rax, 9
or	rax, rdx
mov	rax, B[rax]

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!

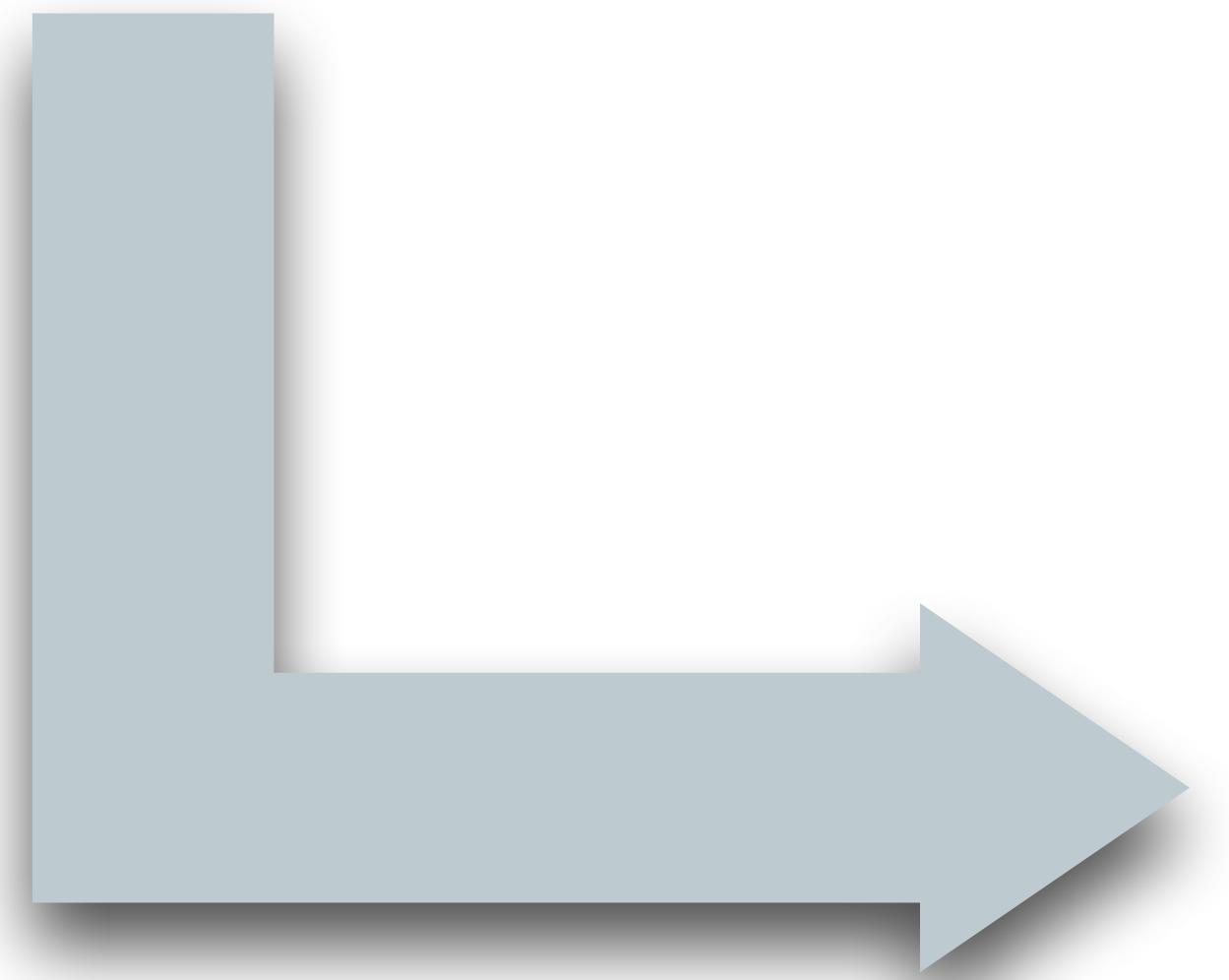
mov	rax, A_size
mov	rcx, x
mov	rdx, 0
cmp	rcx, rax
jae	END
cmovae	-1, rdx
mov	rax, A[rcx]
jne	rax, END
cmovne	-1, rdx
mov	rax, [B]

Example #08 - FEN

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

Example #08 - FEN

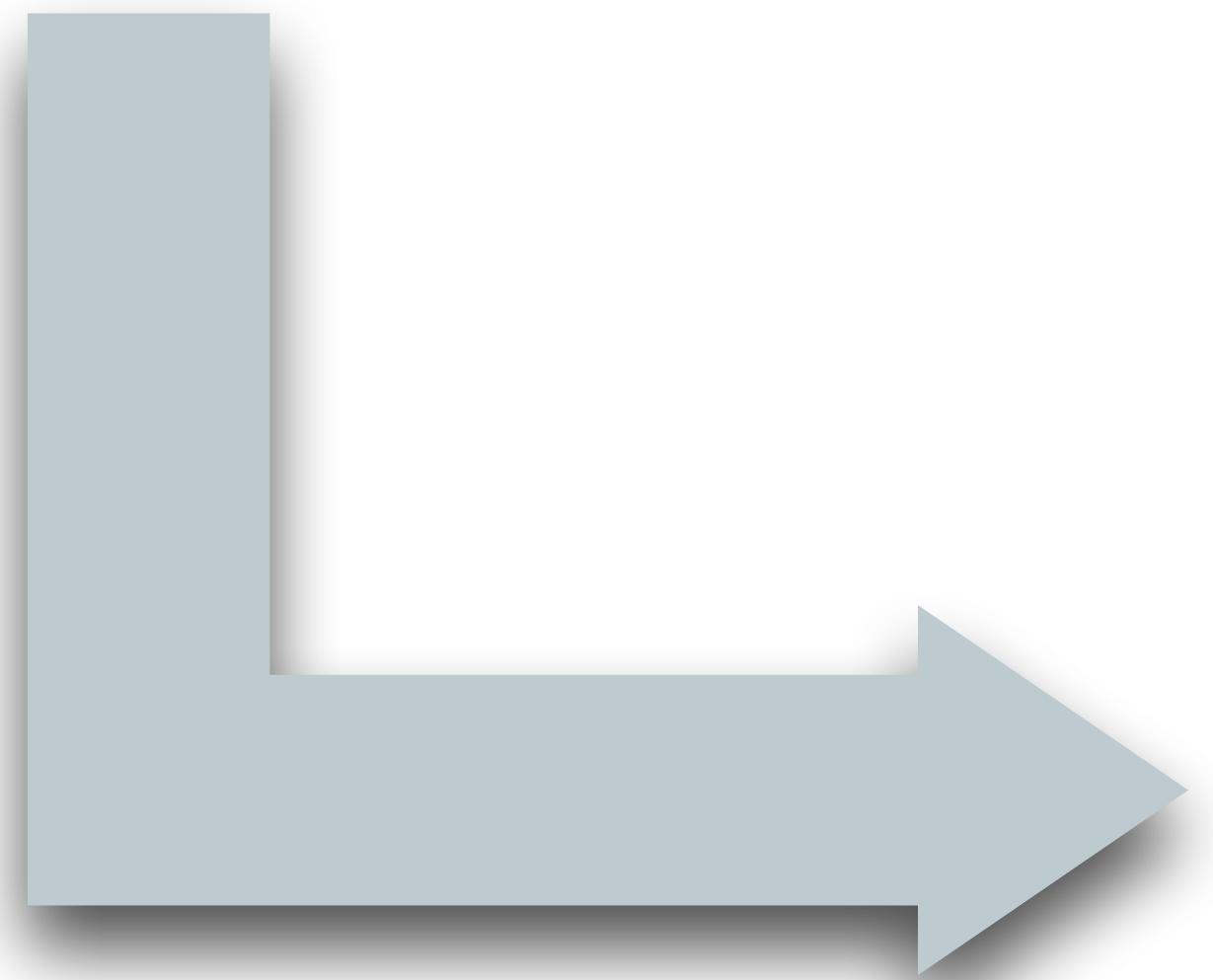
```
y = B[A[x < A_size ? (x+1) : 0] * 512]
```



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

Example #08 - FEN

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



lfence is unnecessary

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

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]

```

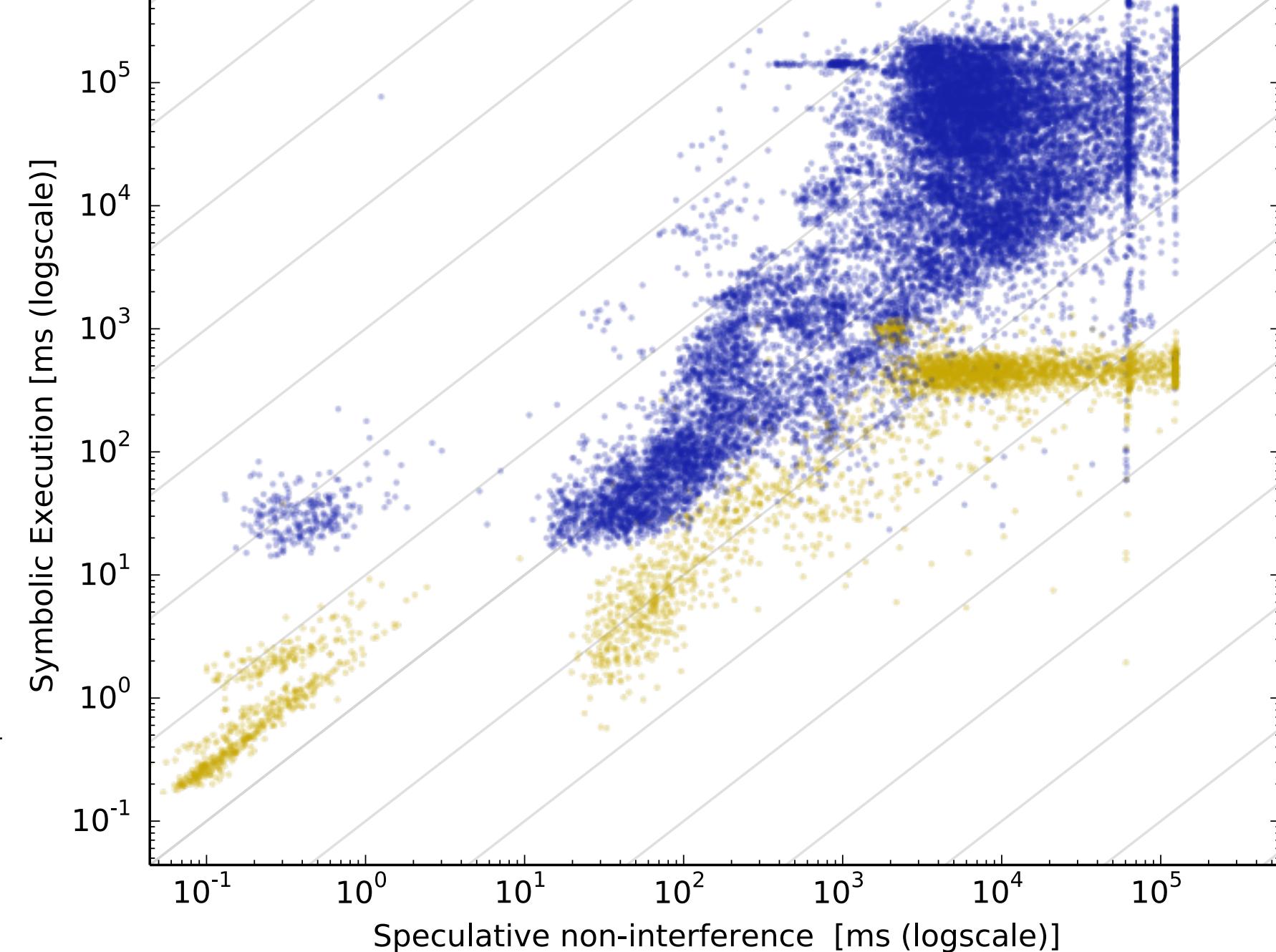
x64 to µASM

L1:
END:

Symbolic
execution



Check for speculative leaks

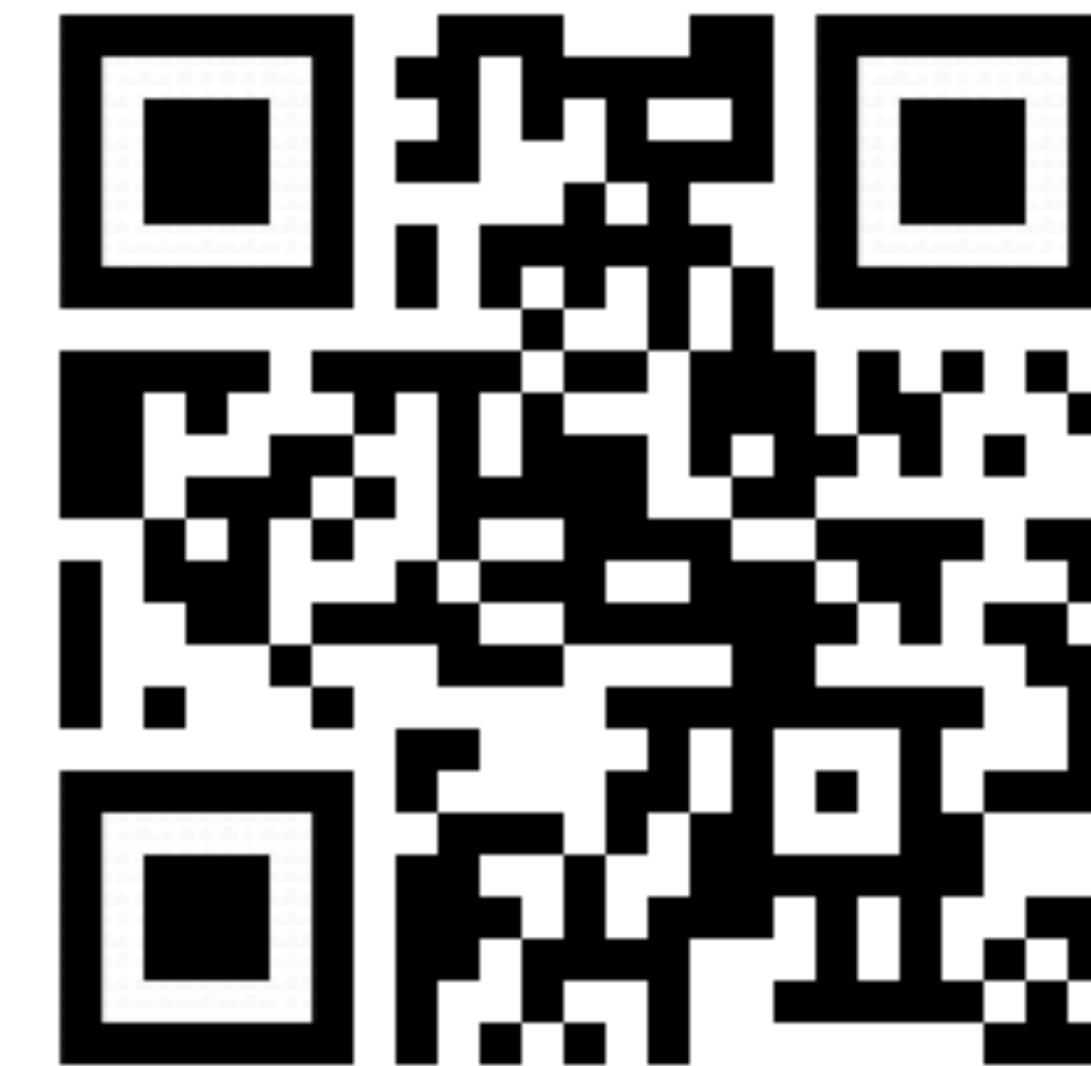


Speculative non-interference

Formally!

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

For all
 P_{non}



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	•
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

