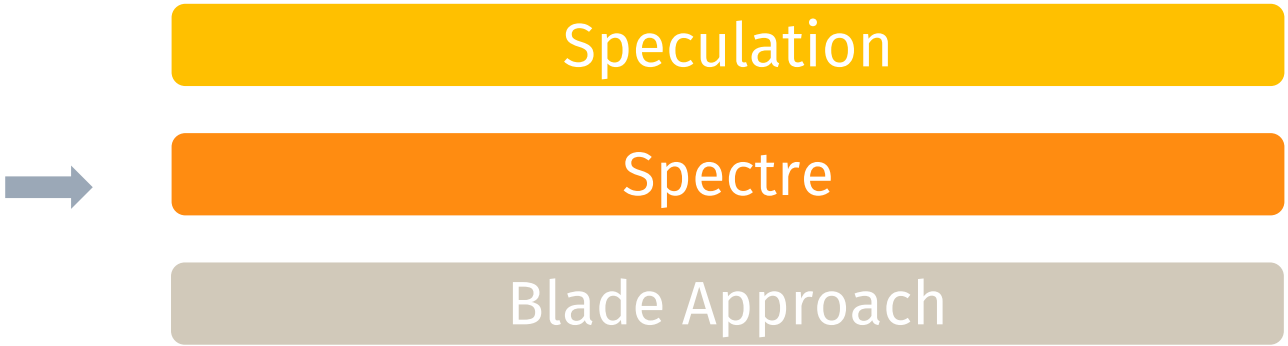
Outline

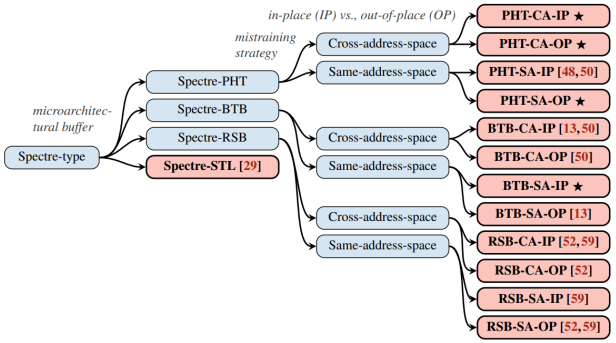


Spectre exploits spec ex to leak secrets.

We are now ready for

Spectre is an attack exploiting **speculative execution** to **leak secrets**

- It is part of a recent wave of **micro-architectural attacks**, i.e., attacks using side-channels induced by the micro-architecture of a processor (including **cache**, timers, virtual memory, …)
  design choices
- "**Spectre** […] transiently bypasses software-defined security policies (e.g., bounds checking, function call/return abstractions, memory stores) to leak secrets out of the program's intended code/data paths." [Canella et al., 2019] (← bit outdated, but a good survey on Spectre!)
- Lot of variants:



There are a lot of different variants.

[Canella et al., 2019] Canella, Claudio, et al. "A systematic evaluation of transient execution attacks and defenses." *28th USENIX Security Symposium (USENIX Security 19)*. 2019. URL: https://arxiv.org/abs/1811.05441v3

What is a gadget?

A **gadget** is a piece of existing code in an (unmodified) existing program binary

- A malicious actor can influence program control flow to cause gadget code to execute
- Gadget code can perform some action of interest to the attacker

That a malicious actor can influence to implement an attack

Spectre v1/v1.1 (special-cases of PHT)

Spectre, also called **Spectre-PHT**: the attacker **mistrains** the branch predictor, by poisoning the **P**attern **H**istory **T**able

↳ To force guess to one branch or the other.

How? **For example, as follows**

- The attacker**:** ↱ we focus on SW that handles secrets
  1. Looks for a piece of "vulnerable code" (**code gadget**) including a condition (next slide)
  2. Runs that code multiple times with an input such that the condition always hold (so training the branch predictor to take the "true" branch)
  3. Then, runs the code with a specially-crafted input making the condition false: now the CPU mis-speculates and executes the "true" branch with wrong data (!!!)
  4. Finally, it looks for information left behind after the CPU discovered it mis-speculated and rolled-back the computation (e.g., part of the contents of caches)

We focus on **Spectre v1** and **v1.1**: due to the fact that modern microprocessors may speculate beyond a **bounds check** condition

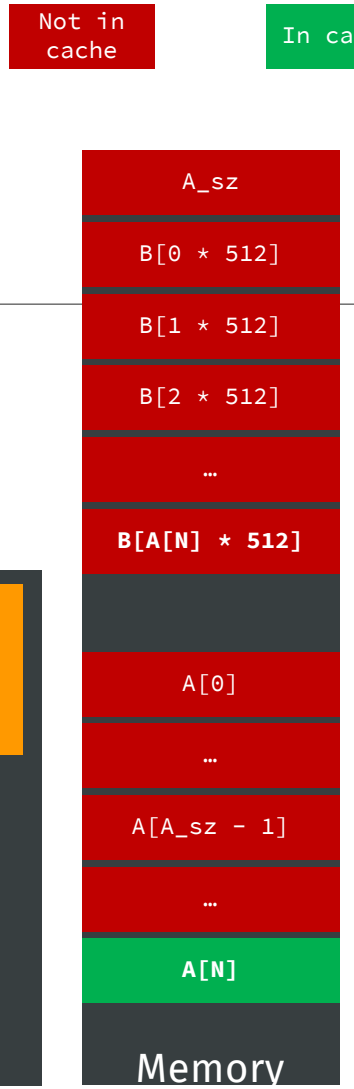
Spectre v1 (Bounds Check Bypass)

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor
3. Suppose **B** and **A_sz** not in cache
4. Call the gadget with x = N with N > A_sz (**A[N]** cached, **value secret!**)
5. CPU mis-speculates on the bound check and accesses **A[N]** (fast!)
6. At that point, **B[A[N] * 512]** is loaded in the cache
7. CPU discovers that the condition was false: **rollback!**
8. When control returns to caller (attacker) it just goes over **B[i * 512]** for each **i** and measures access time:
   - if **i** ≠ **A[N]** the access will be slow
   - if **i** = **A[N]** the access will be fast, the secret is leaked!

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

→ You have a bound check①

The attacker can successfully read past the end of the array **A**



① And this is exploited at lower level. We suppose the predictor that bound check holds. Attacker forces program to access A[n] and B[A[n]]. We want to discover A[n]:

You can try to access B[A * 512] until you have a fast access (already cached). Attacker can access B[N].
Attacker exploits this to read beyond A array.

Spectre v1.1 (Bounds Check Bypass on stores)

1. Find a **code gadget** in victim
2. Mis-train the branch predictor
3. Let:
   - A, A_sz, B as above
   - **A[N]** contains a **secret**
   - y (supplied by the attacker) s.t. A[y] points on the return address on the stack
   - z (supplied by the attacker) to point to an instruction accessing **B[A[N] * 512]**
4. If function returns during mis-speculation, the CPU loads **B[A[N] * 512]** in cache
5. Despite rollback, when control returns to caller (attacker) it just goes over B[i * 512] and measures access time:
   - if i ≠ **A[N]** the access will be slow 
   - if i = **A[N]** the access will be fast, the secret is leaked! 

```
if (y < A_sz)
  A[y] = z;
```

Spectre: recap

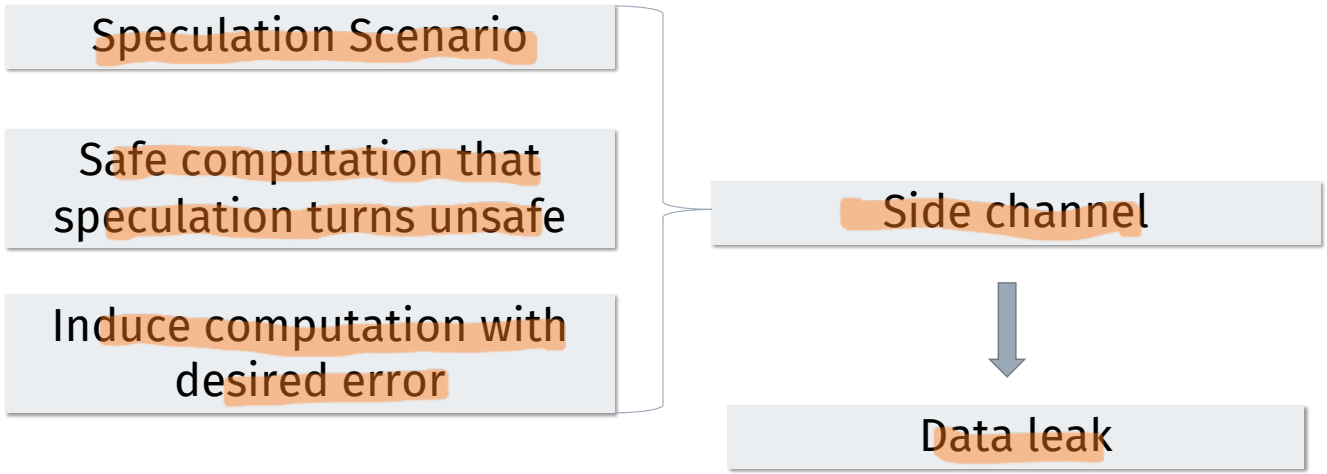
↳ Speculation is used here!

- Branch condition evaluation might require (slow) loads from memory
- Rather than wait for branch resolution, predict outcome of the branch

Then **Spectre**

- uses side channels to obtain the information from the accessed memory location
- allows an attacker to trick error-free programs into leaking their secrets
- breaks the isolation between different application

Spectre pattern



Not a bug: everything complies w/ Archi Vec. spec.

Is Spectre a bug? No, it is a symptom

Architecture is ambiguous under this regard.
Priority given to performance instead of security

- No, everything complies with the architecture specifications
- It is a symptom of excessive architectural ambiguity: typical architectures' guarantees are not enough for security
- Priority given to performance
- In summary, Spectre reveals an architecture-software security gap (SW and HW developers make different assumptions)

Mitigating Spectre

1. Give up speculation, but its costly

- Mitigations for Spectre are hard
- Also, the number of different variants does not help
- One simple idea is to **stop speculation** from accessing to data it should not:

Fences
insertion of a serializing instruction
↳ insert points in which you stop speculation, you force CPU to serialize instructions

Speculative Load Hardening (SLH)
conditional masking of the untrusted_offset
creates artificial data dependecies to stop speculation.

Mitigating Spectre: fences

A fence or speculation barrier is a processor instruction that inhibits load speculation

- On x86_64, we use `lfence` [load fence] on all branches to inhibit speculation;
- Despite the name, `lfence` serializes execution completely, and not just loads
- In pseudo-C:

lfence works against load speculation and direct branch prediction

```
if (x < A_sz) {
  __asm__("lfence");
  y = B[A[x] * 512];
}
else {
  __asm__("lfence");
  /* … */
}
```

Load speculation is when the CPU tries to predict the outcome of a memory load instruction *before* knowing whether it's actually safe or legal to do so. It does this to run faster, but this can accidentally expose data if the speculation goes wrong.

Mitigating Spectre: fences

- The **Good:**
  - It works (mostly :)
  - Probably nothing else!
- The **Bad:**
  - **Highly** inefficient (it stops speculation **completely**)
  - Requires re-compilation

```
if (x < A_sz) {
  __asm__("lfence");
  y = B[A[x] * 512];
}
else {
  __asm__("lfence");
  /* … */
}
```

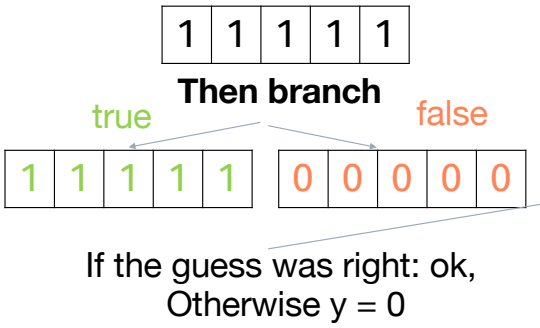
You have to decide where to put them, you can use heuristics but it is not able to cover everything

- Heuristics placement is risky: no guarantees
- To make this a bit better one could use static analysis to decide where fences should go
  - E.g., **Microsoft MSVC** does that by detecting known problematic patterns

Mitigating Spectre: SLH

introduce a mask before branches

- Speculative Load Hardening (SLH) is a bit more sophisticated than fences
- **Idea:**
  - It may be more efficient to introduce an **artificial data dependency** between the condition of a jump and the pointer:
  - Simplifying, for Spectre v1 (assuming "_ ? _ : _" to be implemented without branching, e.g., using a cmov in x86)




```
uint mask = ALL_ONES; /* all bits set to 1 */
if (cond) {
  mask = !cond ? ALL_ZEROES : mask;
  // …
  y = B[(A[x] * 512) & mask];
}
else {
  mask = cond ? ALL_ZEROES : mask;
  /* … */
}
```

You set mask all to 1.

Conditional update **cannot** be speculated

before doing anything you set mask that changes bits to 0 if cond is not satisfied.

So we won't execute speculatively

Mitigating Spectre: SLH

```
if (cond) {
  mask = !cond ? ALL_ZEROES : mask;
  // …
  y = B[(A[x] * 512) & mask];
}
else {
  mask = cond ? ALL_ZEROES : mask;
  /* … */
}
```

- The **Good:**
  - It works (with some changes also for other Spectre variants)
  - More efficient than fences
  - No need of static analysis code (but still could help)
- The **Bad:**
  - Still slow (mask must be known when accessing to B, also when speculating!)
  - Must be carefully implemented
  - Still requires re-compilation

Fixing Spectre: other ideas

Other possible fixes:

- Place secrets in different processes, to make them isolated and non reachable:
  - Just like **Chrome** and **Firefox**
- Reduce the bandwidth of side-channels:
  - e.g., via shadow micro-arch state that can be discarded, less accurate timers, adding noise, …

Is Spectre is fixed?

Only if you are willing to re-compile/re-design your system and make it slow!

Outline

Speculation

Spectre

Blade Approach

Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade: a short guide

[Vassena et al., 2021] M. Vassena, C. Disselkoen, K. von Gleissenthall, S. Cauligi, R. Gökhan Kıcı, R. Jhala, D. Tullsen, and D. Stefan. "Automatically eliminating speculative leaks from cryptographic code with blade." POPL 2021. URL: https://arxiv.org/abs/2005.00294

We use formal methods

Blade approch

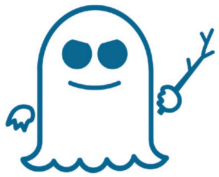
stops spec. in precise and minimum # of prs

where introduces spec.

Insight:

- Blade stops speculation only along specific paths
- It **cuts** the dataflow from expressions that speculatively introduce secrets (**sources**) to those that leak them through the cache (**sinks**), rather than prohibit speculation altogether

Blade



They provide us w/ structure

Transient-Flow Type System

Static *detection of Spectre vulnerabilities*

technique through a type sys

With minimal number of fences

Automatic *repair of vulnerable programs*

↳ Automatic repair alg. to transform programs in safe ones.

How? With a **static type system** that types each expression as

- **transient** (possibly containing speculative secrets) or
- **stable**, and

prohibits speculative leaks by requiring that all **sink** expressions are **stable**

- Blade relies on an abstract primitive, **protect**, to halt speculation at fine granularity, which is implemented using existing architectural mechanisms
- Blade's type system can **automatically synthesize** a **minimal number of protects** to provably eliminate speculative leaks

in your words

The new primitive: protect

x: = protect(e)

primitive as a sort of fine-grained fence

It is a sort of fine-grained fence based on SLH

Semantics: evaluate **e** and assign it to **x**, only after its value is non speculative, i.e., it is **stable**

Insertion of protect

Blade inserts protect to stop speculation in specific points

```
if (i < A_sz)
  x = A[i];
  y = B[x];
…
```

Insertion of protect

Blade inserts protect to stop speculation in specific points

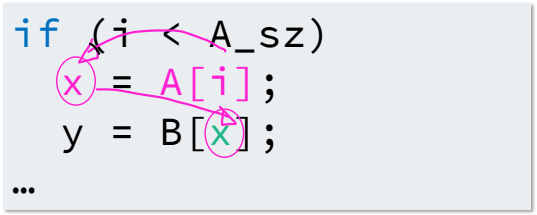
1. Identify sources and sinks

```
if (i < A_sz)
  x = A[i];
  y = B[x];
…
```

Insertion of protect

Blade inserts protect to stop speculation in specific points

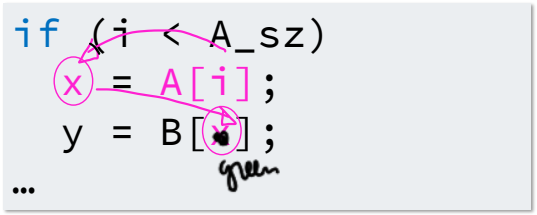
1. Identify sources and sinks
2. Detect data flow from sources to sinks



Insertion of protect

Blade inserts protect to stop speculation in specific points

1. Identify sources and sinks
2. Detect data flow from sources to sinks



Protect to the rescue

Blade inserts protect to stop speculation in specific points

1. Identify sources and sinks
2. Detect data flow from sources to sinks
3. Insert protect to **cut** these data flows

Sources are x and A[i], the sink is x.

```
if (i < A_sz)
  x = A[i];
  y = B[x];
…
```

There is a formal fw to do this

only problematic instr are stopped, not everything

```
if (i < A_sz)
  x = protect(A[i]);
  y = B[x];
…
```

Only instructions depending on x are stopped!

- Type sys statically approximates the runtime propag. of spec. secrets.

A type system for speculation

Static approximation of the runtime propagation of **speculative secrets**

Expressions are

↳ Think of transient like taint

- **transient** possibly containg speculative secrets
- **stable** not containing speculative secrets

Transient = tainted

Propagate taints in expresions

Restricts using transient data In commands

Static Taint Tracking

Judgment of exp. ↱ Judgment

$\Gamma \vdash e : \tau$ ① $\quad \Gamma \vdash c$ ② for command.

↳ That has a type

Well-typed expressions are free of speculation leaks

① Exp e has type τ. ② Command is well typed

Typing rules

Crucial in speculation: even though both are stable results is transient: every time you access memory through array you can leak something.

Array and index must be stable

Array rule

$$\frac{\Gamma \vdash e_1 : S \quad \Gamma \vdash e_2 : S}{\Gamma \vdash e_1\,[e_2] : T}$$

Array reads can introduce transient data

No branching on transient data

Condition needs to be stable. You want to avoid misprediction

$$\frac{\Gamma \vdash e : S \quad \Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash \text{if } e \text{ then } c_1 \text{ else } c_2}$$

Automatic repair algorithm

Well-typed programs are free of speculation leaks

And ill-typed programs?

Ill-typed programs can be repaired by inserting **protect** statements, which **cut** the data-flow between **sources** and **sinks**

Transient-Flow Type System by Example

1. Build the def-use graph that caputers the data dependencies

↳ definition-use variable: show where you define and where you use variables.

EXAMPLE

$x := a[i_1]$

$y := a[i_2]$

$z := x + y$

$w := b[z]$

Source of transient values

Sink

z is transient instead of being stable

Transient types

Sink

No transient expressions in sinks!

Repair by Example

1. Build the **def-use graph** that caputers the data dependencies

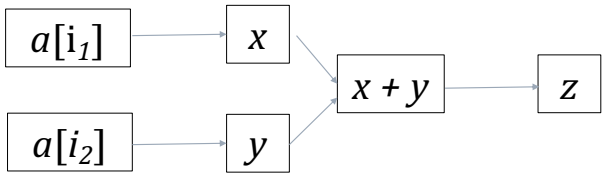
EXAMPLE

$x := a[i_1]$

$y := a[i_2]$

$z := x + y$

$w := b[z]$



Repair by Example

1. Build the **def-use graph** that caputers the data dependencies
2. Add source and sink nodes and connect

EXAMPLE

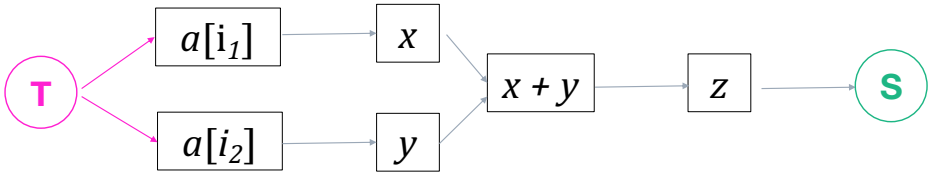
$x := a[i_1]$

$y := a[i_2]$

$z := x + y$

$w := b[z]$

- the **source node** to expressions that read memory
- expressions that must be stable to the **sink node**



Sink for stable nodes

Not a problem, but we have to repair it, We need to put protects.

Repair by Example

1. Build the **def-use graph** that caputers the data dependencies
2. Add source and sink nodes and connect

EXAMPLE

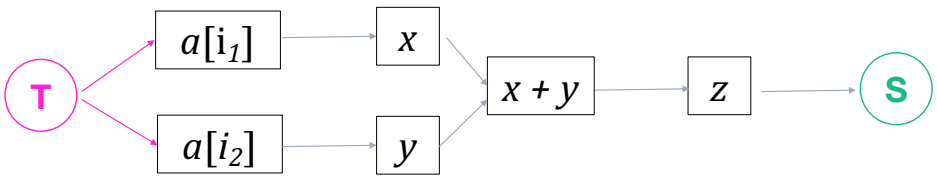
$x := a[i_1]$

$y := a[i_2]$

$z := x + y$

$w := b[z]$

- the **source node** to expressions that read memory
- expressions that must be stable to the **sink node**



Sink for stable nodes

A path from source to sink indicates that the program may be **leaky**

Repair by Example

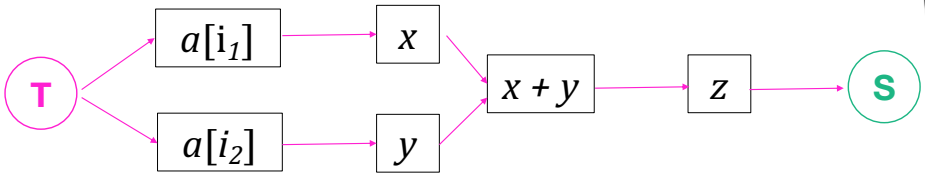
1. Cut the path from sink to source, i.e., find a set of variables that disconnect source and sink

Equivalent to the problem of finding a **cut set**
A **cut set** is set of variables, the removal of which from the graph eliminates all paths from T to S

↳ find variables whose removal eliminates the path you don't want.

EXAMPLE

$x := a[i_1]$
$y := a[i_2]$
$z := x + y$
$w := b[z]$



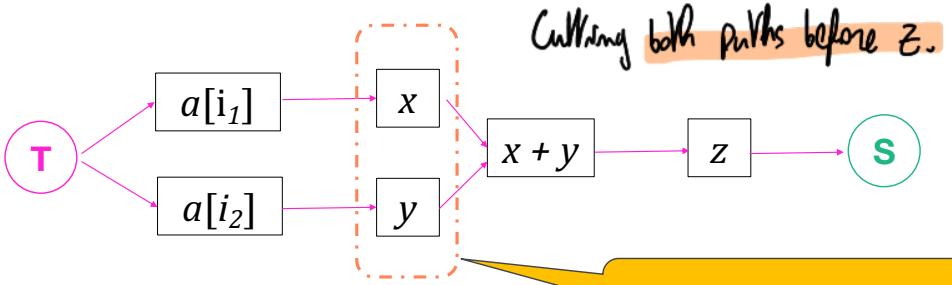
Repair by Example

1. Cut the path from sink to source, i.e., find a set of variables that disconnect source and sink

EXAMPLE

$x := a[i_1]$
$y := a[i_2]$
$z := x + y$
$w := b[z]$

Equivalent to the problem of finding a **cut set**
A **cut set** is set of variables, the removal of which from the graph eliminates all paths from **T** to **S**



Insert a protect
$x :=$ **protect**$(a[i_1])$
$y :=$ **protect**$(a[i_2])$

Sub-optimal cut

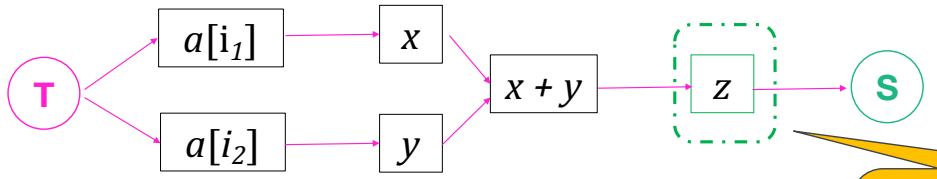
Repair by Example

1. Cut the path from sink to source, i.e., find a set of variables that disconnect source and sink

Equivalent to the problem of finding a **cut set**
A **cut set** is set of variables, the removal of which from the graph eliminates all paths from **T** to **S**

EXAMPLE

$x := a[i_1]$
$y := a[i_2]$
$z := x + y$
$w := b[z]$



$z :=$ **protect**$(x+y)$

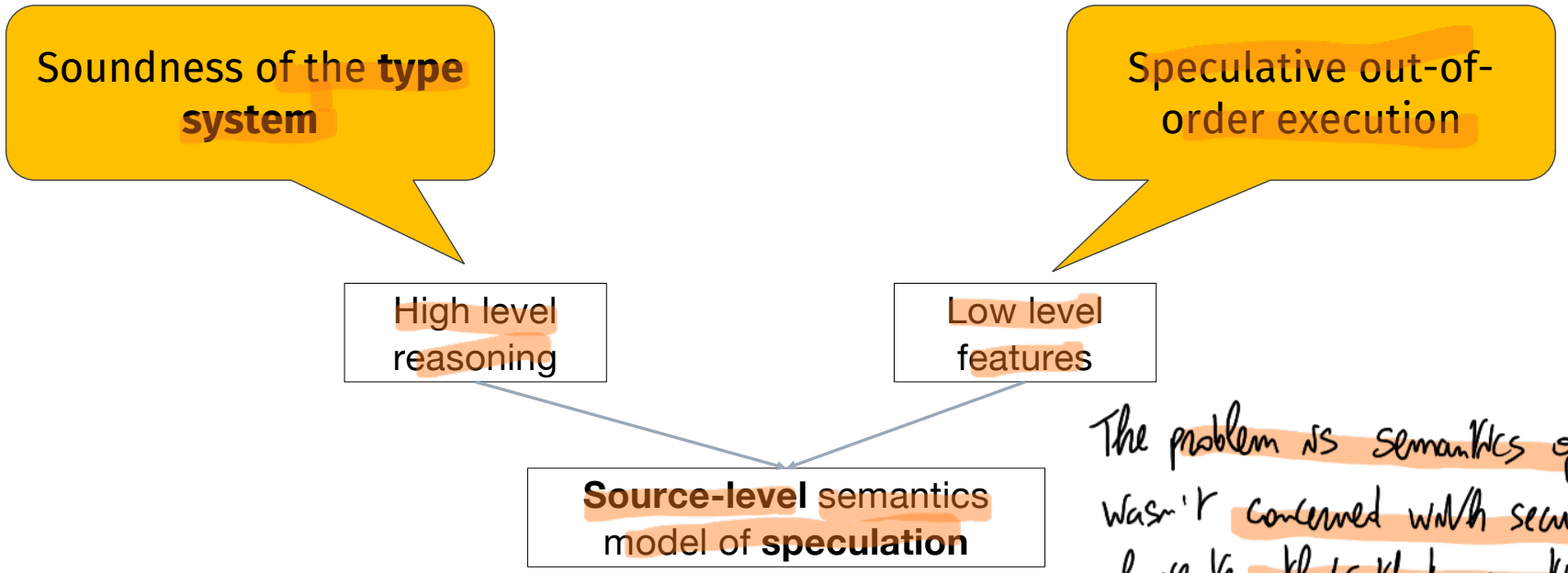
Minimal cut
Classic max-flo min cut problem

You can explore a lot of known solutions; you can use

max-flow min cut polynomial algorithm! Efficient.

Challenges

You can use Operat. Semantics to model low level behavior, but you will need to introduce abstractions for it.



The problem is semantics of speculation wasn't concerned with security. You have to think what semantics of commands mentions w/ cache

Modelling Speculation

A relatively simple language is chosen. Why? It

1. Simplifies the reasoning
2. Allows to reason directly at **source level**

We will use a simple language

Not too many details about the architecture must be known

- A **Just In Time (JIT) semantics** taking high-level source programs, translating them on-the-fly to low-level processor instructions and executing them
- Two structures:
  - Instruction reorder buffer
  - Command Stack

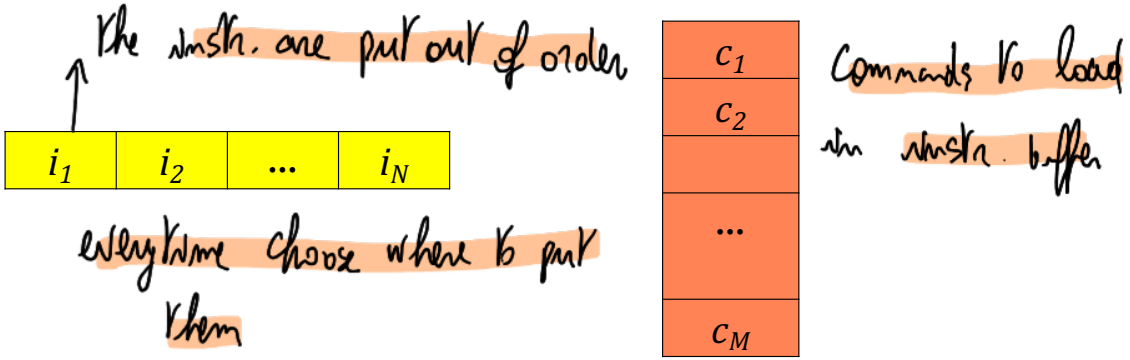
It takes high level source progs, and

This is the semantics we will use (one is high level, one low).

Modeling speculation

Used structures:

- Instruction reorder buffer (is)
- Command Stack (cs)
- memory (μ)
- map from variables to values (ρ)



Three stages (**fetch, exec, retire**)

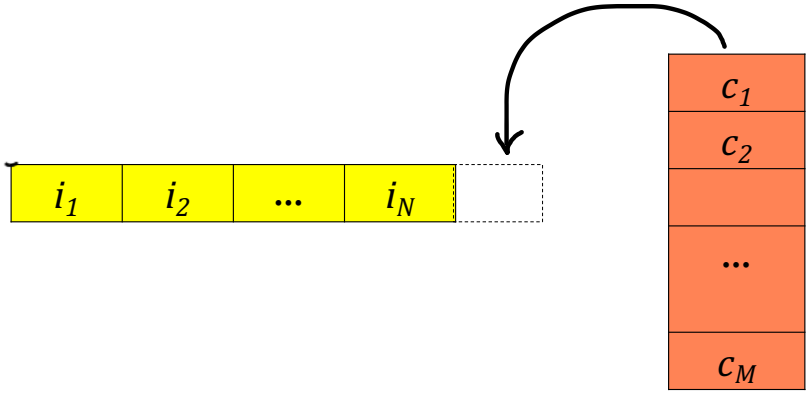
The branch prediction is not modeled:

- the attacker can control the processor with a set of **directives**

Fetch stage

- Take a command from the top of cs,
- flatten it into a sequence of instructions, and
- store them in is

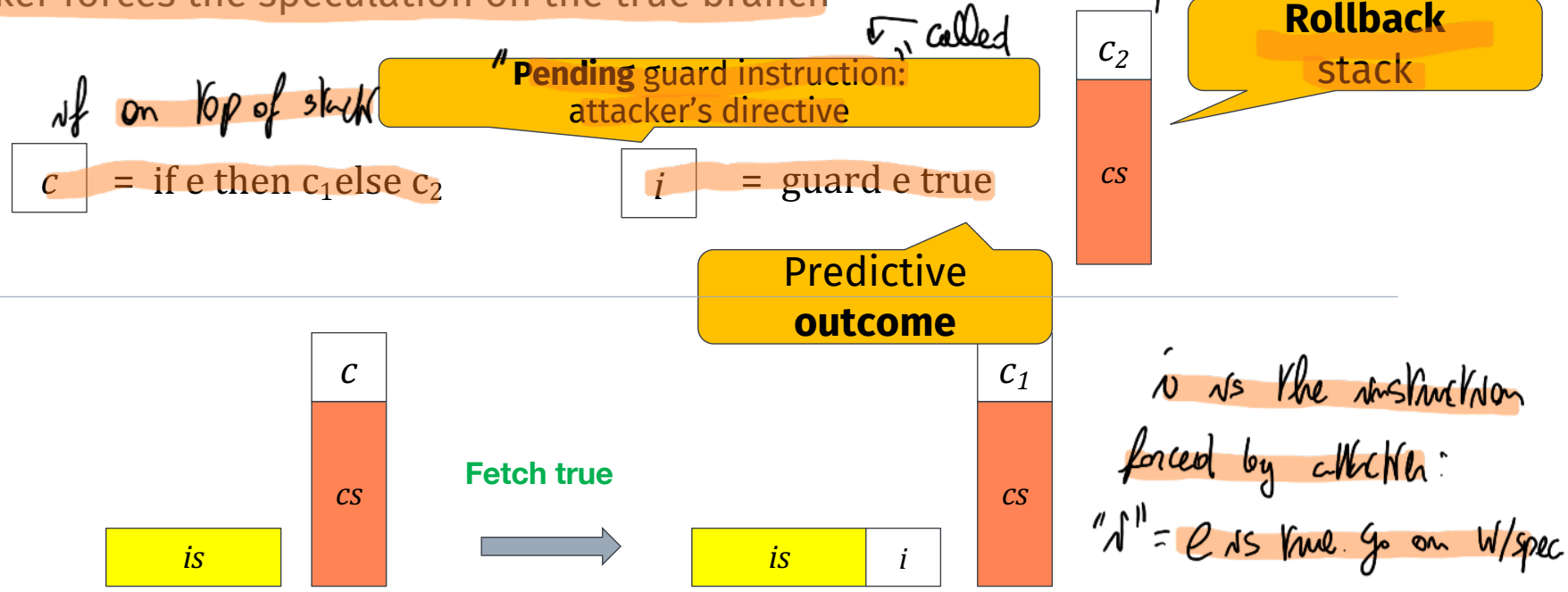
Both is and cs are updated in is' and cs'



You start from one config and update it into (cs', is')

Fetch-if-true

The attacker forces the speculation on the true branch



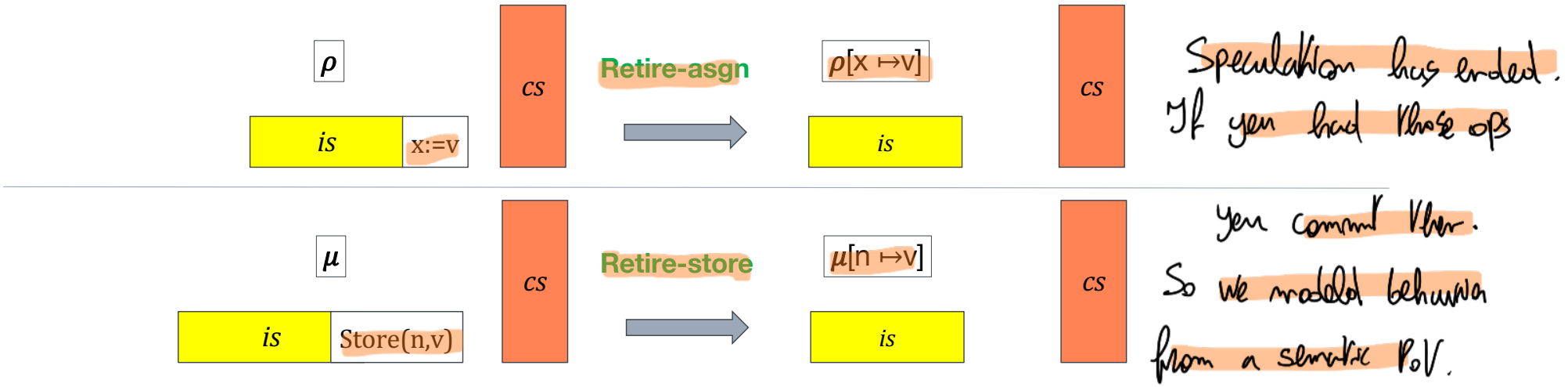
Execute stage

- take the n-th instruction from is (provided by the attacker) and
- execute it **speculatively**:
  - memory μ and map from variables to values ρ **do not change**

Intuitively **operands of instruction** are evaluated just using non-transient information

Retire stage

- Remove complete instructions from is and
- **commit changes to memory** μ and **variable map** ρ that are modified in μ' and ρ' (done in-order, so giving the illusion of no instruction-level parallelism!)



Using formal method gives guarantees.

Formal guarantees

→ IR models behavior in the same way but it includes speculation and out of order exec.

- **Consistency**: the JIT semantics **agree** with the standard sequential (in-order) semantics → agrees with a standard semantics of command, you can commit or rollback too.

  We will see it

- **Correctness**: the repair algorithm synthetizes **well-typed** programs (provably secure repair)

- **Soundness**: well-typed constant time programs are also **speculative constant-time** (1)

Constant time programs do not allow attackers to infer info on exec. time. under this approach this is true for spec too.

↳ With Spectre you need to reason over SCT (1)

The “source” language

$$
\begin{aligned}
\text{Values } v &::= n \mid b \mid a \\
\text{Expr. } e &::= v \mid x \mid e + e \mid e < e \\
&\mid e \otimes e \mid e\,?\,e : e \\
&\mid \mathit{length}(e) \mid \mathit{base}(e) \\
\text{Rhs. } r &::= e \mid *e \mid a[e] \\
\text{Cmd. } c &::= \mathbf{skip} \mid x := r \mid *e = e \\
&\mid a[e] := e \mid \mathbf{fail} \mid c; c \\
&\mid \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c \\
&\mid \mathbf{while}\ e\ \mathbf{do}\ c \\
&\mid x := \mathbf{protect}(r)
\end{aligned}
$$

- This is **easy**, just the WHILE-language!
- Sure?
  - **Question:** how do you think `protect` works?
    - **Hint:** SLH prevent “wrong” accesses. Can you “abstract” SLH in a single command?
  - **Answer:** completes the assignment only when the evaluation of `r` is **stable**, i.e., no more rollbacks are possible, and `r` is the final, non speculative value

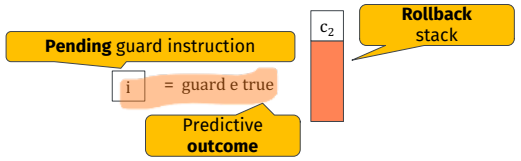
The “target” language

move

Instructions $i ::= \mathbf{nop} \mid x := e \mid x := \mathbf{load}(e)$
$\mid \mathbf{store}(e, e) \mid x := \mathbf{protect}(e)$
$\mid \mathbf{guard}(e^b, cs, p) \mid \mathbf{fail}(p)$

↓ Plus models a Wrong choosing the branch, slide 54.

- Again, we have `protect`!
- **guard** (e^b, cs, p):
  - expresses a pending jump
  - e is the condition, b is its predicted outcome, cs is what to do if a rollback is needed (ignore p, an identifier useful for the security analysis)



The source language can be translated on-the-fly to a target one, using a JIT semantics that models speculation

The «Compiler»: A JIT semantics for speculation

The semantics formalizes the execution of source progams on a pipelined processor

$$C \rightarrow_o^d C'$$

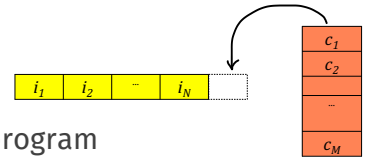
Read as: "under the directive d, the configuration C moves to C' with observation o" (what the attacker can see), where

force guard, execute a group of instr.

- d ::= **fetch** | **fetch** b | **exec** n | **retire**, attacker-supplied directives (also each "class" of directives is a **stage** of the processor's pipeline)

in this model attacker is choosing what we can do

- o ::= ϵ | **read** (n, ps)|**write** (n, ps)| **fail** |**rollback** (p) , attacker's observations
- C, C' are of the form $< is, cs, \mu, \rho >$, where: Configurations
  - is is a reorder buffer, i.e., a queue of in-flight (target) instructions (directives give the order)
  - cs represents the stack of (source) commands (the current execution path), i.e., the continuation of the program
  - μ is a memory and ρ is a map from variables to values



JIT semantics for speculation: the fetch stage

either fetch next or force a guard

$$< is, cs, \mu, \rho > \rightarrow_o^{\textbf{fetch}} < is', cs', \mu, \rho > \qquad < is, cs, \mu, \rho > \rightarrow_o^{\textbf{fetch}\ b} < is', cs', \mu, \rho >$$

- **Idea:** take a command from the top of cs, flatten it into a sequence of instructions and store them in is

FETCH-SEQ

$$\langle is, (c_1; c_2) : cs, \mu, \rho \rangle \xrightarrow{\textbf{fetch}}_\epsilon \langle is, c_1 : c_2 : cs, \mu, \rho \rangle$$

FETCH-ASGN

$$\langle is, x := e : cs, \mu, \rho \rangle \xrightarrow{\textbf{fetch}}_\epsilon \langle is +\!\!+ [x := e], cs, \mu, \rho \rangle$$

FETCH-PTR-LOAD

$$\langle is, x := *e : cs, \mu, \rho \rangle \xrightarrow{\textbf{fetch}}_\epsilon \langle is +\!\!+ [x := \textbf{load}(e)], cs, \mu, \rho \rangle$$

FETCH-ARRAY-LOAD

$$\frac{c = x := a[e] \qquad e_1 = e < length(a) \qquad e_2 = base(a) + e \qquad c' = \textbf{if } e_1 \textbf{ then } x := *e_2 \textbf{ else fail}}{\langle is, c : cs, \mu, \rho \rangle \xrightarrow{\textbf{fetch}}_\epsilon \langle is, c' : cs, \mu, \rho \rangle}$$

FETCH-IF-TRUE

$$\frac{c = \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \qquad \text{fresh}(p) \qquad i = \textbf{guard}(e^{\textbf{true}}, c_2 : cs, p)}{\langle is, c : cs, \mu, \rho \rangle \xrightarrow{\textbf{fetch true}}_\epsilon \langle is +\!\!+ [i], c_1 : cs, \mu, \rho \rangle}$$

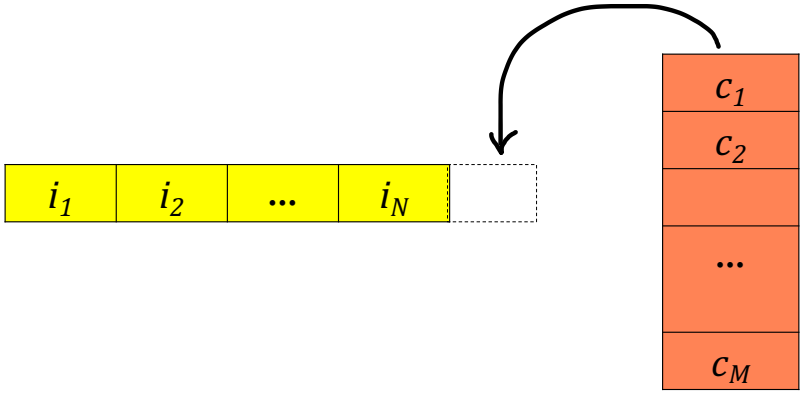
The attacker decides which branch

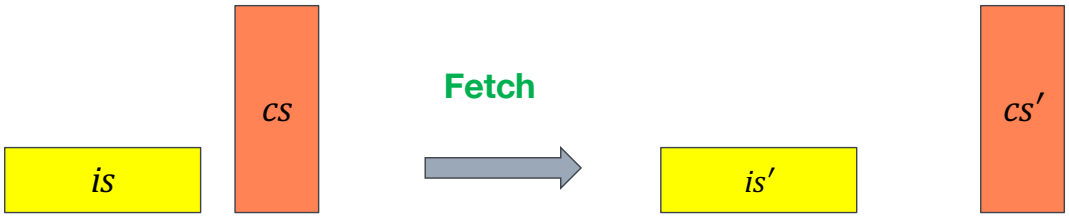
Fetch stage

$$< is, cs, \mu, \rho > \rightarrow_o^{\text{fetch}} < is', cs', \mu, \rho >$$

- Take a command from the top of cs,
- flatten it into a sequence of instructions, and
- store them in is

Both is and cs are updated in is' and cs'

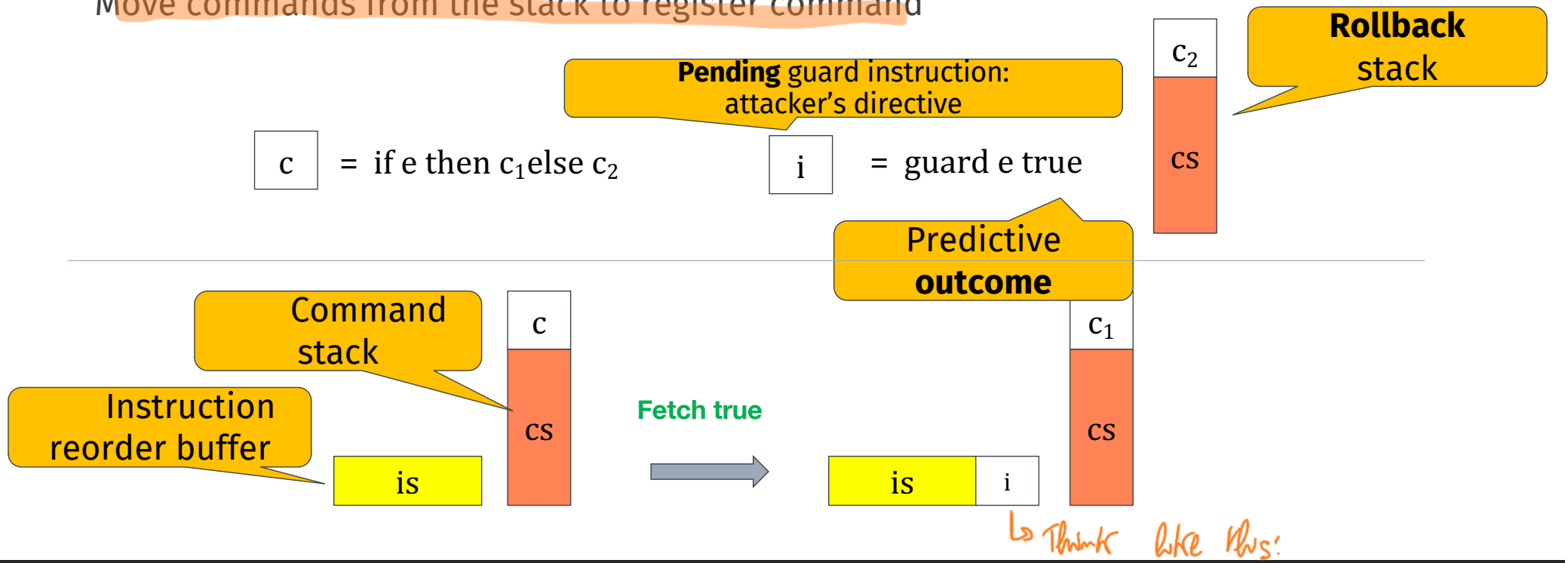




Fetch-if-true

$$\textsc{Fetch-If-True}\quad \frac{c = \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \qquad \text{fresh}(p) \qquad i = \textbf{guard}(e^{\text{true}}, c_2 : cs, p)}{\langle is, c : cs, \mu, \rho\rangle \xrightarrow{\text{fetch true}}_{\epsilon} \langle is +\!\!+ [i], c_1 : cs, \mu, \rho\rangle}$$

Move commands from the stack to register command



JIT semantics for speculation: the execute stage

$$< is, cs, \mu, \rho > \rightarrow_o^{\mathbf{exec}\ n} < is', cs', \mu, \rho >$$

You change NS, CS but no branch of memory or env.

- **Idea:** take the n-th instruction from is (provided by the attacker) and execute it **speculatively**
  - Lots of details here, but intuitively **operands of instruction** are evaluated just using non-transient information (computed by an ad hoc function ϕ that marks variables from pending assignments)

EXECUTE

$$\frac{|is_1| = n - 1 \qquad \rho' = \phi(is_1, \rho) \qquad \langle is_1, i, is_2, cs\rangle \xrightsquigarrow{(\mu,\rho',o)} \langle is', cs'\rangle}{\langle is_1 +\!\!+ [\,i\,] +\!\!+ is_2, cs, \mu, \rho\rangle \xrightarrow{\mathbf{exec}\ n}_o \langle is', cs', \mu, \rho\rangle}$$

EXEC-LOAD

$$\frac{i = (x := \mathbf{load}(e)) \qquad \mathbf{store}(\_,\_) \notin is_1 \qquad n = [\![e]\!]^\rho \qquad ps = (\!|is_1|\!) \qquad i' = (x := \mu(n))}{\langle is_1, i, is_2, cs\rangle \xrightsquigarrow{(\mu,\rho,\mathbf{read}(n,ps))} \langle is_1 +\!\!+ [\,i'\,] +\!\!+ is_2, cs\rangle}$$

EXEC-BRANCH-OK

$$\frac{i = \mathbf{guard}(e^b, cs', p) \qquad [\![e]\!]^\rho = b}{\langle is_1, i, is_2, cs\rangle \xrightsquigarrow{(\mu,\rho,\epsilon)} \langle is_1 +\!\!+ [\,\mathbf{nop}\,] +\!\!+ is_2, cs\rangle}$$

EXEC-BRANCH-MISPREDICT

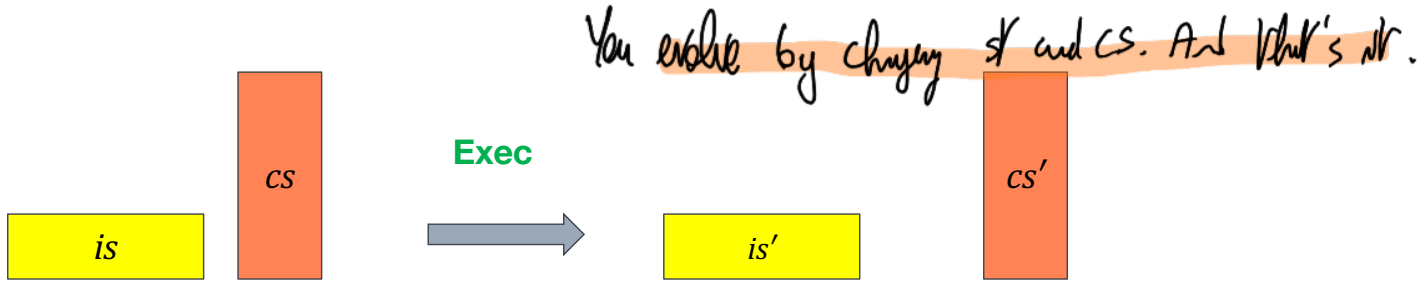
$$\frac{i = \mathbf{guard}(e^b, cs', p) \qquad b' = [\![e]\!]^\rho \qquad b' \neq b}{\langle is_1, i, is_2, cs\rangle \xrightsquigarrow{(\mu,\rho,\mathbf{rollback}(p))} \langle is_1 +\!\!+ [\,\mathbf{nop}\,], cs'\rangle}$$

Execute stage

$$< is, cs, \mu, \rho > \rightarrow_{o}^{\mathbf{fetch}} < is', cs', \mu, \rho >$$

- take the n-th instruction from is (provided by the attacker) and
- execute it **speculatively**:
  - memory μ and map from variables to values ρ **do not change**

Intuitively **operands of instruction** are evaluated just using non-transient information



JIT semantics for speculation: the retire stage

$$< is, cs, \mu, \rho > \rightarrow_o^{\textbf{retire}} < is', cs', \mu', \rho' >$$

- **Idea:** remove complete instructions from is and commit changes to memory and variable map
  - Do that in-order, so giving the illusion of no instruction-level parallelism!

RETIRE-NOP

$$\langle \textbf{nop} : is, cs, \mu, \rho \rangle \xrightarrow{\textbf{retire}}_{\epsilon} \langle is, cs, \mu, \rho \rangle$$

RETIRE-ASGN

$$\langle x := v : is, cs, \mu, \rho \rangle \xrightarrow{\textbf{retire}}_{\epsilon} \langle is, cs, \mu, \rho[x \mapsto v] \rangle$$

RETIRE-STORE

$$\langle \textbf{store}(n, v) : is, cs, \mu, \rho \rangle \xrightarrow{\textbf{retire}}_{\epsilon} \langle is, cs, \mu[n \mapsto v], \rho \rangle$$

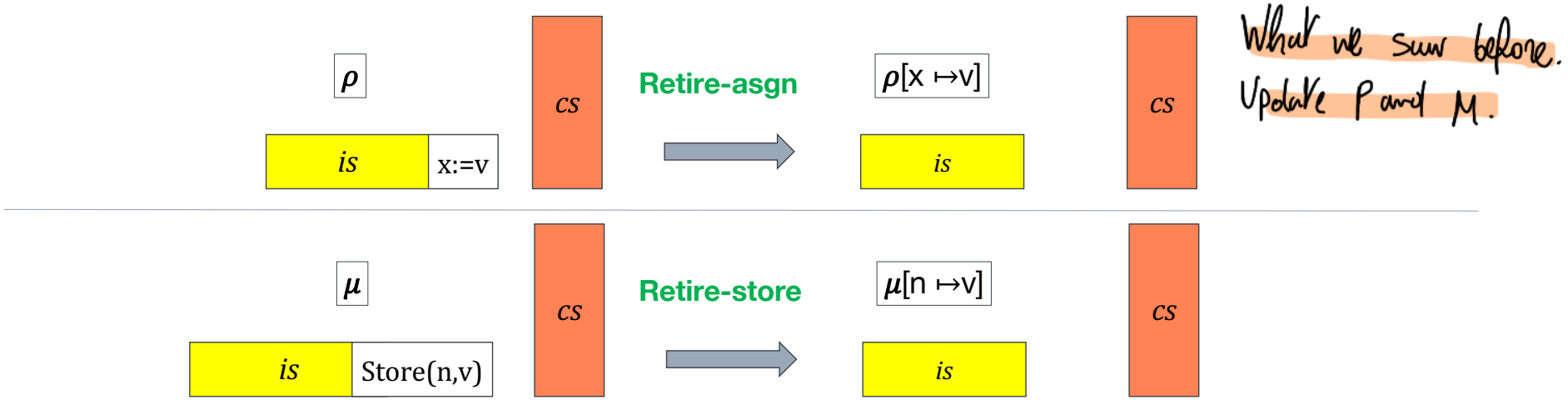
RETIRE-FAIL

$$\langle \textbf{fail}(p) : is, cs, \mu, \rho \rangle \xrightarrow{\textbf{retire}}_{\textbf{fail}(p)} \langle [\,], [\,], \mu, \rho \rangle$$

Retire stage

$$< is, cs, \mu, \rho > \rightarrow_o^{\textbf{retire}} < is', cs', \mu', \rho' >$$

- Remove complete instructions from is and
- **commit changes to memory** μ and **variable map** ρ that are modified in μ' and ρ' (done in-order, so giving the illusion of no instruction-level parallelism!)



You can implement protect in diff ways

A few words about `protect`

- `protect` can be implemented both in software and in hardware
- **Software implementation:** no support from hardware needed
  - exactly as SLH for array access (i.e., computes a mask depending on the array's bound check, and then executes the access depending using the mask)
    - ↳ MaskS
- **Hardware implementation:** Can you guess? What needs to be added?
  - **Fetch stage:** translates `protect` into `protect`, easy
  - **Exec aux. relation:** reduces the parameter to a value, then if no `guard` is pending removes the `protect`
  - Instruction x:=protect(r) assigns the value of r , only after all previous guard instructions have been executed, i.e., when the value has become stable and no more rollbacks are possible

`Protect`: software implementation

FETCH-PROTECT-SLH

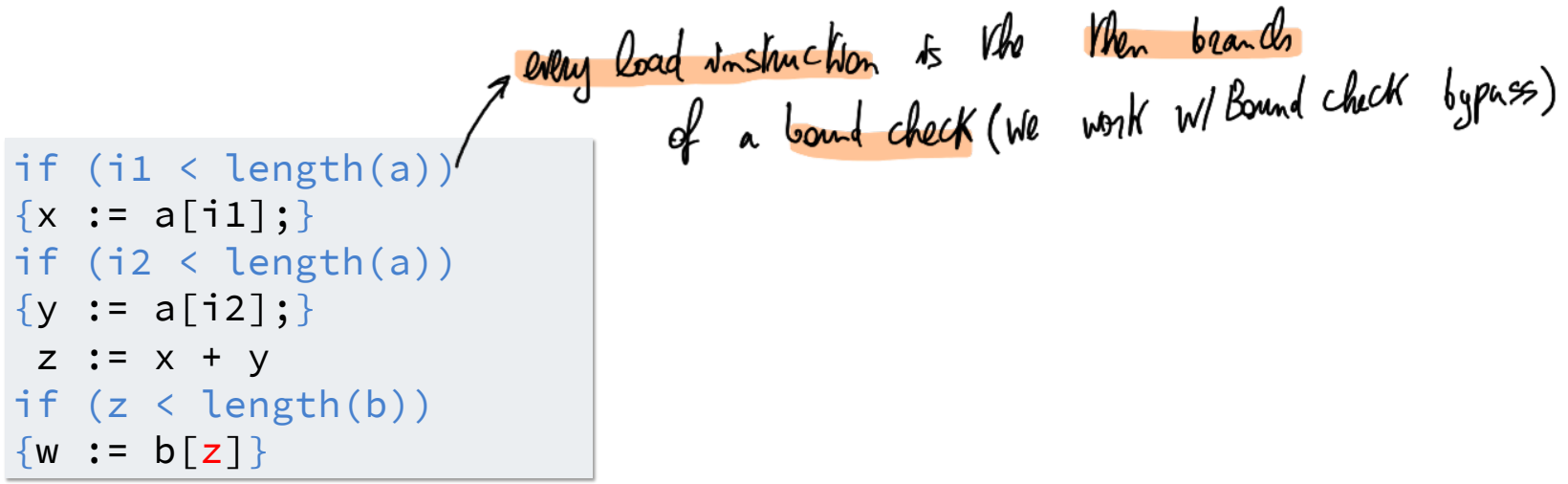
$$\frac{c = x := \mathbf{protect}(a[e]) \qquad e_1 = e < length(a) \qquad e_2 = base(a) + e}{c_1 = m := e_1 \qquad c_2 = m := m\ ?\ \mathbf{1} : \mathbf{0} \qquad c_3 = x := *(e_2 \otimes m) \qquad c' = c_1; \mathbf{if}\ m\ \mathbf{then}\ c_2; c_3\ \mathbf{else}\ \mathbf{fail}}$$

$$\langle is, c : cs, \mu, \rho \rangle \xrightarrow{\mathbf{fetch}}_{\epsilon} \langle is, c' : cs, \mu, \rho \rangle$$

(b) Software implementation of **protect**$(a[e])$.

```
uint mask = ALL_ONES; /* all bits set to 1 */
if (cond) {
  mask = !cond ? ALL_ZEROES : mask;
  // …
  y = B[(A[x] * 512) & mask];
}
else {
  mask = cond ? ALL_ZEROES : mask;
  /* … */
}
```

Example: the source



Example: the target

Reformulate a program in a way that complies with semantics.

```
if (i1 < length(a))
{x := a[i1];}
if (i2 < length(a))
{y := a[i2];}
 z := x + y
if (z < length(b))
{w := b[z]}
```

x := load(e)
guard(eb, cs, p)

Attacker's directives

1	$\mathbf{guard}((i_1 < length(a))^{\mathbf{true}}, [\mathbf{fail}], 1)$
2	$x := \mathbf{load}(base(a) + i_1)$
3	$\mathbf{guard}((i_2 < length(a))^{\mathbf{true}}, [\mathbf{fail}], 2)$
4	$y := \mathbf{load}(base(a) + i_2)$
5	$z := x + y$
6	$\mathbf{guard}((z < length(b))^{\mathbf{true}}, [\mathbf{fail}], 3)$
7	$w := \mathbf{load}(base(b) + z)$

the

base addr. + offset

every time you have an if you have a guard

The attacker instructs the processor to speculatively execute the load instructions (2,4,5,7) and the assignment, but not the guards

Example of leaking execution

Evolution of the buffer is after each attacker directive (execute 2,4,5,7); the memory μ and variable map ρ are shown above

$C \rightarrow_o^d C'$
C, C' are of the form
$< is, cs, \mu, \rho >$

b[0]
a[0]
a[1]
SECRET s[0]

Memory Layout

$\mu(0) = 0$	$b[0]$
$\mu(1) = 0$	$a[0]$
$\mu(2) = 0$	$a[1]$
$\mu(3) = 42$	$s[0]$
...	...

Variable Map

$\rho(i_1) = 1$
$\rho(i_2) = 2$
...

	Reorder Buffer is
1	$\textbf{guard}((i_1 < length(a))^{\textbf{true}}, [\textbf{fail}], 1)$
2	$x := \textbf{load}(base(a) + i_1)$ this is an ok move: $i_1 = 1$, a[1] is okay
3	$\textbf{guard}((i_2 < length(a))^{\textbf{true}}, [\textbf{fail}], 2)$
4	$y := \textbf{load}(base(a) + i_2)$
5	$z := x + y$
6	$\textbf{guard}((z < length(b))^{\textbf{true}}, [\textbf{fail}], 3)$
7	$w := \textbf{load}(base(b) + z)$
	Observations: o

Example: exec 2

Directive **exec 2** executes the first load by computing the memory address 2 and replacing the instruction with the assignment x := $\mu(2)$ containing the loaded value 0

$C \rightarrow_o^d C'$
C, C' are of the form
$< is, cs, \mu, \rho >$

Memory Layout

$\mu(0) = 0$	$b[0]$
$\mu(1) = 0$	$a[0]$
$\mu(2) = 0$	$a[1]$
$\mu(3) = 42$	$s[0]$
...	...

Variable Map

$\rho(i_1) = 1$
$\rho(i_2) = 2$
...

	Reorder Buffer is	**exec** 2
1	**guard**$((i_1 < length(a))^{\textbf{true}}, [\textbf{fail}], 1)$	
2	$x :=$ **load**$(base(a) + i_1)$	$x := \mu(2)$
3	**guard**$((i_2 < length(a))^{\textbf{true}}, [\textbf{fail}], 2)$	
4	$y :=$ **load**$(base(a) + i_2)$	
5	$z := x + y$	
6	**guard**$((z < length(b))^{\textbf{true}}, [\textbf{fail}], 3)$	
7	$w :=$ **load**$(base(b) + z)$	
	Observations: o	**read**$(2, [1])$

Example: exec 4

Directive **exec 4** transiently reads public array a past its bound, at index 2, reading into the memory ($\mu(3) = 42$) of secret array s [0] and generates the correspondingobservation

Memory Layout

$\mu(0) = 0$	$b[0]$
$\mu(1) = 0$	$a[0]$
$\mu(2) = 0$	$a[1]$
$\mu(3) = 42$	$s[0]$
...	...

Variable Map

$\rho(i_1) = 1$
$\rho(i_2) = 2$
...

	Reorder Buffer is	exec 2	exec 4
1	$\mathbf{guard}((i_1 < length(a))^{\mathbf{true}}, [\mathbf{fail}], 1)$		
2	$x := \mathbf{load}(base(a) + i_1)$	$x := \mu(2)$	
3	$\mathbf{guard}((i_2 < length(a))^{\mathbf{true}}, [\mathbf{fail}], 2)$		
4	$y := \mathbf{load}(base(a) + i_2)$		$y := \mu(3)$
5	$z := x + y$		
6	$\mathbf{guard}((z < length(b))^{\mathbf{true}}, [\mathbf{fail}], 3)$		
7	$w := \mathbf{load}(base(b) + z)$		
	Observations: o	$\mathbf{read}(2, [1])$	$\mathbf{read}(3, [1, 2])$

$i_2 = 2$, so base + 2 : 2nd pos. of the memory

Example: exec 5

The processor forwards the values of x and y through the transient variable map ρ [x $\mapsto \mu(2)$, y $\mapsto \mu(3)$] to compute their sum in the fifth instruction, (z :=42)

Memory Layout

$\mu(0) = 0$	$b[0]$
$\mu(1) = 0$	$a[0]$
$\mu(2) = 0$	$a[1]$
$\mu(3) = 42$	$s[0]$
...	...

Variable Map

$\rho(i_1) = 1$
$\rho(i_2) = 2$
...

	Reorder Buffer is	exec 2	exec 4	exec 5
1	$\textbf{guard}((i_1 < \mathit{length}(a))^{\textbf{true}}, [\textbf{fail}], 1)$			
2	$x := \textbf{load}(\mathit{base}(a) + i_1)$	$x := \mu(2)$		
3	$\textbf{guard}((i_2 < \mathit{length}(a))^{\textbf{true}}, [\textbf{fail}], 2)$			
4	$y := \textbf{load}(\mathit{base}(a) + i_2)$		$y := \mu(3)$	
5	$z := x + y$			$z := 42$
6	$\textbf{guard}((z < \mathit{length}(b))^{\textbf{true}}, [\textbf{fail}], 3)$			
7	$w := \textbf{load}(\mathit{base}(b) + z)$			
	Observations: o	$\textbf{read}(2, [1])$	$\textbf{read}(3, [1, 2])$	ϵ

↳ active guards

Example: exec 7

The value of z is then used as an index in the last instruction and leaked to the attacker via observation read(42, [1, 2, 3])

Memory Layout

$\mu(0) = 0$	$b[0]$
$\mu(1) = 0$	$a[0]$
$\mu(2) = 0$	$a[1]$
$\mu(3) = 42$	$s[0]$
...	...

Variable Map

$\rho(i_1) = 1$
$\rho(i_2) = 2$
...

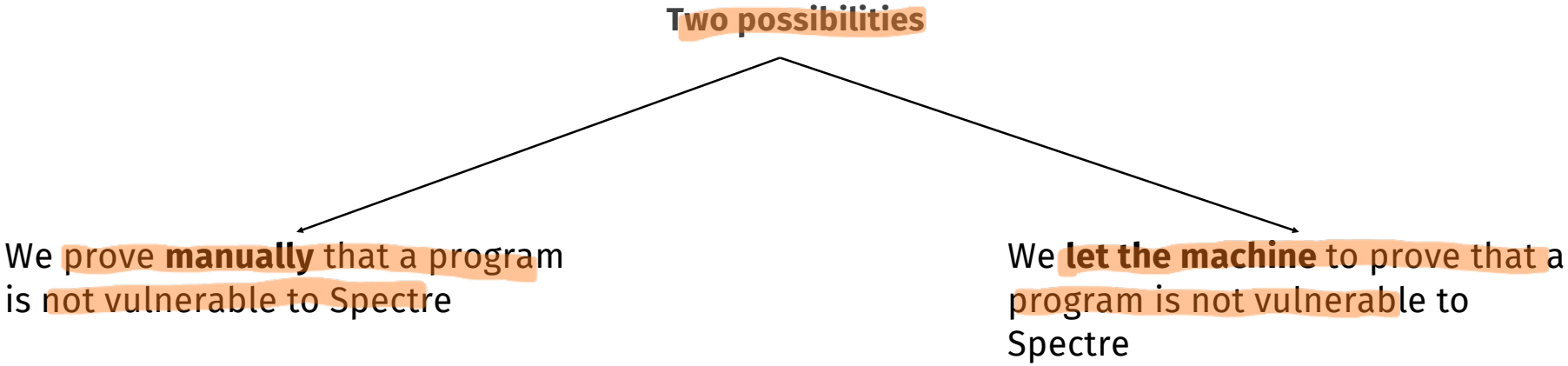
	Reorder Buffer is	exec 2	exec 4	exec 5	exec 7
1	$\mathbf{guard}((i_1 < length(a))^{\mathbf{true}}, [\mathbf{fail}], 1)$				
2	$x := \mathbf{load}(base(a) + i_1)$	$x := \mu(2)$			
3	$\mathbf{guard}((i_2 < length(a))^{\mathbf{true}}, [\mathbf{fail}], 2)$				
4	$y := \mathbf{load}(base(a) + i_2)$		$y := \mu(3)$		
5	$z := x + y$			$z := 42$	
6	$\mathbf{guard}((z < length(b))^{\mathbf{true}}, [\mathbf{fail}], 3)$				
7	$w := \mathbf{load}(base(b) + z)$				$w := \mu(42)$
	Observations: o	$\mathbf{read}(2, [1])$	$\mathbf{read}(3, [1, 2])$	ϵ	$\mathbf{read}(42, [1, 2, 3])$

Secret 42 is leaked through a read!

Almost there...

The language(s) we just saw are nice, but

- Can we be sure that the programs are *speculatively secure*?



Classify vars as transient and stable.

Let the machine prove things...

... via **type checking**!

Transient-flow type system that statically rejects programs that can potentially leak through transient execution attacks: i.e. that exhibit any source-to-sink data flows

Expressions

Types to vars

$\Gamma \vdash e : \tau$

Reads as: "expression e has type τ under Γ"

- Γ maps variables to types
- τ is a type, can be either
  - Stable **S**, i.e., contains no transient values during executions
  - Transient **T**, otherwise
  - **S** $\sqsubseteq$ **T** (can-flow-to relation) not viceversa

• We use a lattice

Lattice

S $\sqsubseteq$ **T**

This order should be respected

if not, type sys rejects program

Commands

$\Gamma, \mathrm{Prot} \vdash c$

Reads as: "command c is well-typed under Γ and Prot"

- Γ as before, Prot a set of implicitly protected vars (1)
- **Idea:** if all assignments to variables in Prot are protected, then c does not leak

(1) You have to prove if sys is well typed when fixed system.

Vars over which you use protect or they can be vars assumed to be protected.

Let the machine prove things… (cont)

… via a type **inference** (type constraints restrict the types)!

You have a further part in Judgment that allows you to also infer the type constraints as you are analyzing programs.

Expressions:

$$\Gamma \vdash e : \tau \Rightarrow k$$

Commands:

We record which variables are protected

$$\Gamma, \mathrm{Prot} \vdash c \Rightarrow k$$

In both cases, the algorithm

i. generates a set of **type constraints** k, involving concrete types and atoms (i.e., program variables and type variables)
ii. builds a **def-use** graph: nodes are types and atoms, edges their relations; If there is a path from T to S the program is leaky (*Thm:* k is satisfiable *iff* no path from T to S exists) and reconstructs type information
iii. **cuts** all the T to S paths so to make k satisfiable, and protects each program variable in the cut

Let the machine prove things... (cont)

This rule disallows assignment from **T** to **S**, since **T** ⊑ **S** is not true

$$\textsc{Asgn}\quad \frac{\Gamma \vdash r : \tau \Rightarrow k \qquad \tau \sqsubseteq \Gamma(x)}{\Gamma, \mathrm{Prot} \vdash x := r \Rightarrow k \cup (r \sqsubseteq x)}$$

The rule generates the constraint r ⊑ x disallowing transient to stable assignments

Let the machine prove things... (cont)

In case of operation with an expression of type **T** and one of type **S**, since **T** $\sqsubseteq$ **S**, the overall type will be **T**

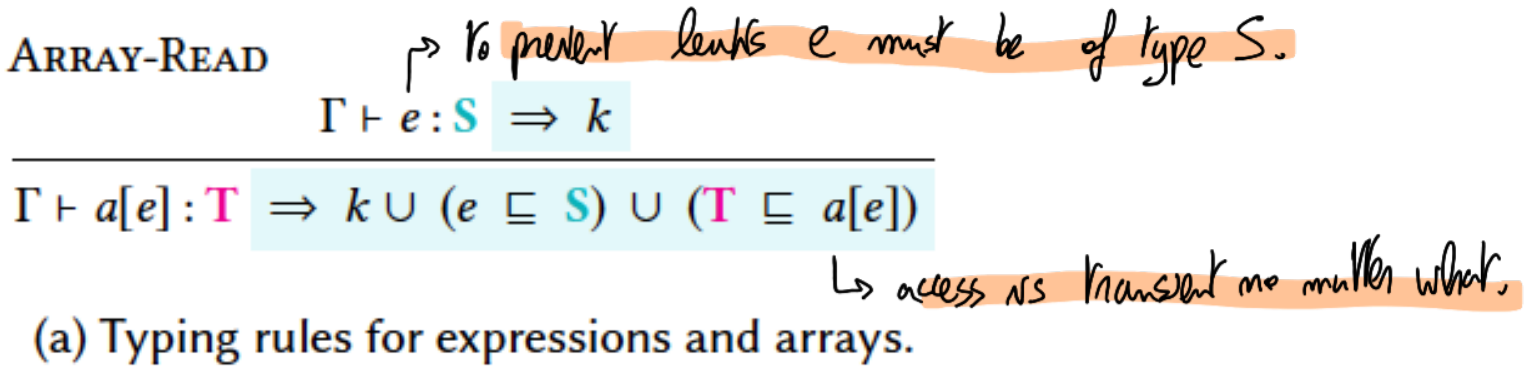
$$\textsc{Bop}\quad \frac{\Gamma \vdash e_1 : \tau_1 \Rightarrow k_1 \qquad \Gamma \vdash e_2 : \tau_2 \Rightarrow k_2 \qquad \tau_1 \sqsubseteq \tau \qquad \tau_2 \sqsubseteq \tau}{\Gamma \vdash e_1 \oplus e_2 : \tau \Rightarrow k_1 \cup k_2 \cup (e_1 \sqsubseteq e_1 \oplus e_2) \cup (e_2 \sqsubseteq e_1 \oplus e_2)}$$

the unknown type of e1 $\oplus$ e2 should be at least as transient as the (unknown) type of e1 and e2

Why to propagate types of your exp.

Let the machine prove things... (cont)

Array can be indexed out of bounds during speculation, hence the rule assigns the transient type T to array reads. E must be of type to avoid leaks



(a) Typing rules for expressions and arrays.

The rule generates constraint e ⊑ S for the unknown type of the array index, thus forcing it to typed be S and forces the type of a[e] to be T

Let the machine prove things... (cont)

The indexed expression used in array writes must be S, while the stored value can have any type

$$\textsc{Array-Write}\quad \frac{\Gamma \vdash e_1 : \mathbf{S} \Rightarrow k_1 \qquad \Gamma \vdash e_2 : \tau \Rightarrow k_2}{\Gamma, \mathrm{Prot} \vdash a[e_1] := e_2 \Rightarrow k_1 \cup k_2 \cup (e_1 \sqsubseteq \mathbf{S})}$$

Let the machine prove things... (cont)

To prevent leaks, branch condition must be S

To prevent implicit flow leaks: expression has to be evaluated

IF-THEN-ELSE

$$\frac{\Gamma \vdash e : \mathbf{S} \Rightarrow k \qquad \Gamma, \mathrm{Prot} \vdash c_1 \Rightarrow k_1 \qquad \Gamma, \mathrm{Prot} \vdash c_2 \Rightarrow k_2}{\Gamma, \mathrm{Prot} \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \Rightarrow k \cup k_1 \cup k_2 \cup (e \sqsubseteq \mathbf{S})}$$

Example: constraint generation

EXAMPLE

$$x := a[i_1]$$
$$y := a[i_2]$$
$$z := x + y$$
$$w := b[z]$$

Example: constraint generation

EXAMPLE

$x := a[i_1]$

$y := a[i_2]$

$z := x + y$

$w := b[z]$

ARRAY-READ

$$\frac{\Gamma \vdash e : \mathbf{S} \Rightarrow k}{\Gamma \vdash a[e] : \mathbf{T} \Rightarrow k \cup (e \sqsubseteq \mathbf{S}) \cup (\mathbf{T} \sqsubseteq a[e])}$$

ASGN

$$\frac{\Gamma \vdash r : \tau \Rightarrow k \qquad \tau \sqsubseteq \Gamma(x)}{\Gamma, \mathrm{Prot} \vdash x := r \Rightarrow k \cup (r \sqsubseteq x)}$$

Example: constraint generation

EXAMPLE

$x := a[i_1]$
$y := a[i_2]$
$z := x + y$
$w := b[z]$

CONSTRAINTS
T ⊑ a[i1]
a[i1] ⊑ x

ARRAY-READ

$$\frac{\Gamma \vdash e : \mathbf{S} \Rightarrow k}{\Gamma \vdash a[e] : \mathbf{T} \Rightarrow k \cup (e \sqsubseteq \mathbf{S}) \cup (\mathbf{T} \sqsubseteq a[e])}$$

ASGN

$$\frac{\Gamma \vdash r : \tau \Rightarrow k \qquad \tau \sqsubseteq \Gamma(x)}{\Gamma, \mathrm{Prot} \vdash x := r \Rightarrow k \cup (r \sqsubseteq x)}$$

- Reading memory creates **T** value
- Types are propagated with assignment: **T** ⊑ *x*, i.e. x can contain transient values at runtime
- Constant values like *a1* do not produce constraints

Example: constraint generation

EXAMPLE

$x := a[i_1]$

$y := a[i_2]$

$z := x + y$

$w := b[z]$

CONSTRAINTS
T ⊑ a[i1]
a[i1] ⊑ x
T ⊑ a[i2]
a[i2] ⊑ y

ARRAY-READ

$$\frac{\Gamma \vdash e : \mathbf{S} \Rightarrow k}{\Gamma \vdash a[e] : \mathbf{T} \Rightarrow k \cup (e \sqsubseteq \mathbf{S}) \cup (\mathbf{T} \sqsubseteq a[e])}$$

ASGN

$$\frac{\Gamma \vdash r : \tau \Rightarrow k \qquad \tau \sqsubseteq \Gamma(x)}{\Gamma, \mathrm{Prot} \vdash x := r \Rightarrow k \cup (r \sqsubseteq x)}$$

Example: constraint generation

EXAMPLE

$x := a[i_1]$

$y := a[i_2]$

$z := x + y$

$w := b[z]$

CONSTRAINTS
T ⊑ a[i1]
a[i1] ⊑ x
T ⊑ a[i2]
a[i2] ⊑ y
x ⊑ x+y
y ⊑ x+y
x+y ⊑ z

BOP

$$\frac{\Gamma \vdash e_1 : \tau_1 \Rightarrow k_1 \qquad \Gamma \vdash e_2 : \tau_2 \Rightarrow k_2 \qquad \tau_1 \sqsubseteq \tau \qquad \tau_2 \sqsubseteq \tau}{\Gamma \vdash e_1 \oplus e_2 : \tau \Rightarrow k_1 \cup k_2 \cup (e_1 \sqsubseteq e_1 \oplus e_2) \cup (e_2 \sqsubseteq e_1 \oplus e_2)}$$

ASGN

$$\frac{\Gamma \vdash r : \tau \Rightarrow k \qquad \tau \sqsubseteq \Gamma(x)}{\Gamma, \text{Prot} \vdash x := r \Rightarrow k \cup (r \sqsubseteq x)}$$

x and y used to compute x+y

Example: constraint generation

EXAMPLE

$x := a[i_1]$
$y := a[i_2]$
$z := x + y$
$w := b[z]$

CONSTRAINTS
T ⊑ a[i1]
a[i1] ⊑ x
T ⊑ a[i2]
a[i2] ⊑ y
x ⊑ x+y
y ⊑ x+y
x+y ⊑ z
z ⊑ S
T ⊑ b[z]
b[z] ⊑ w

ARRAY-READ

$$\frac{\Gamma \vdash e : \mathbf{S} \Rightarrow k}{\Gamma \vdash a[e] : \mathbf{T} \Rightarrow k \cup (e \sqsubseteq \mathbf{S}) \cup (\mathbf{T} \sqsubseteq a[e])}$$

ASGN

$$\frac{\Gamma \vdash r : \tau \Rightarrow k \qquad \tau \sqsubseteq \Gamma(x)}{\Gamma, \mathrm{Prot} \vdash x := r \Rightarrow k \cup (r \sqsubseteq x)}$$

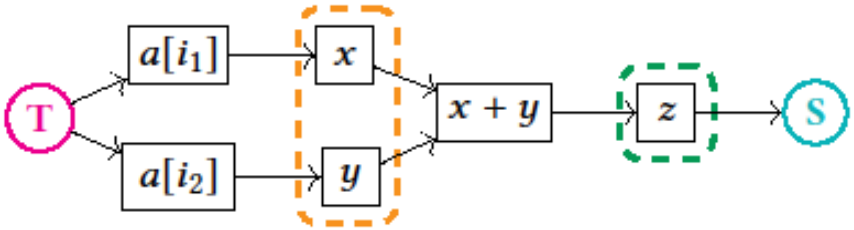
The dataflow

EXAMPLE

$x := a[i_1]$
$y := a[i_2]$
$z := x + y$
$w := b[z]$

CONSTRAINTS
T ⊑ a[i1]
a[i1] ⊑ x
T ⊑ a[i2]
a[i2] ⊑ y
x ⊑ x+y
y ⊑ x+y
x+y ⊑ z
z ⊑ S
T ⊑ b[z]
b[z] ⊑ w

Subset of the **Def-Use Graph**



- T is thesource of transient values of the program
- S is the sink of stable values of the program, i.e., values of a program that must be stable to avoid transient execution attacks

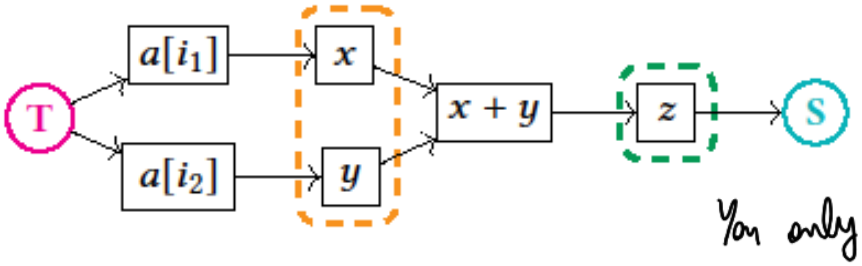
The dataflow

EXAMPLE

$x := a[i_1]$
$y := a[i_2]$
$z := x + y$
$w := b[z]$

CONSTRAINTS
T ⊑ a[i1]
a[i1] ⊑ x
T ⊑ a[i2]
a[i2] ⊑ y
x ⊑ x+y
y ⊑ x+y
x+y ⊑ z
z ⊑ S
T ⊑ b[z]
b[z] ⊑ w

Subset of the **Def-Use Graph**

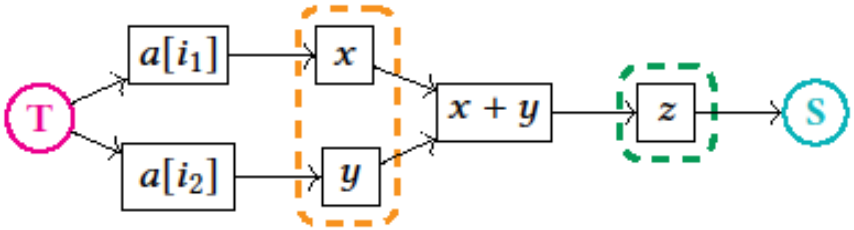


- A set of constraints k is **satisfiable** if and only if there is no T-S path in k
- Here we have two T-S paths: transient data flows through data dependencies into (what should be) a stable index expression and thus the program may be leaky

Cut the dataflow

EXAMPLE

$x := a[i_1]$
$y := a[i_2]$
$z := x + y$
$w := b[z]$



- If a set of constraints is unsatisfiable, as here, we can make it satisfiable by removing some of the nodes or equivalently protecting some of the variables
- A set of variables A is a **cut-set** for a set of constraints k, if A cuts all T-S paths in k

CUT-SETS

{x,y}

{z}

{z} is a minimal cut

Program repair

Program Repair

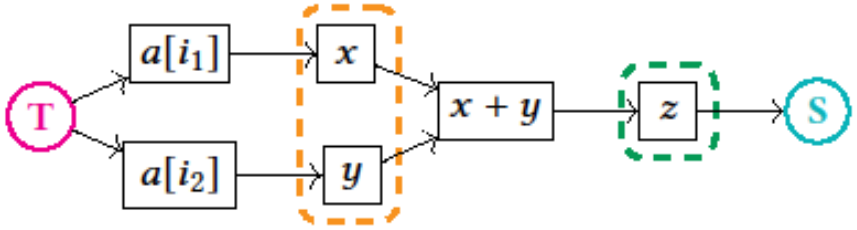
As a final step, our repair algorithm repair traverses program c and inserts a **protect** for each variable in the **cut-set**

EXAMPLE

$x := a[i_1]$
$y := a[i_2]$
$z := x + y$
$w := b[z]$

EXAMPLE PATCHED

$x := \text{protect}(a[i_1])$
$y := \text{protect}(a[i_2])$
$z := \text{protect}(x + y)$
$w := b[z]$



The orange patch is sub-optimal because it requires more protect statements than the optimal green patch

Putting everything together

- **Theorem 1 (Consistency):** Programs executed speculatively produce the same result as if they were executed sequentially (no out of order: result is the same: protect when needed)
- **Constant-time (CT) program** if and only if its branches and memory accesses do not depend on secrets (e.g., crypto keys) – I know, it's strange You can define CT programs under speculation, otherwise CT progs can be leaky.
- **Speculative constant-time (SCT) program** if and only if its observables and final configuration do not depend on secrets (assume the sequence of attacker-provided directives as fixed before execution)
- **Theorem 2 (Soundness):** If a program c is CT and well-typed $\Gamma, \emptyset \vdash c$ then it is also SCT
  - If you want: when compiling c from the source to the target using the JIT "compiler", a secure source program (CT & well-typed) is mapped into a secure target program (SCT)

Summing up

- We worked with the “bare-metal”: caches, speculation, side-channels
- We saw how that can be abstracted elegantly
  - A source language translated on-the-fly to a target one, using a JIT semantics
  - The JIT semantics has nice properties: programs deemed secure at the source are such also at the target
  - Also: it is possible to make programs secure by synthesizing protections
  - Nice theoretical framework 😄

Bibliography

- M. Vassena, C. Disselkoen, K. von Gleissenthall, S. Cauligi, R. Gökhan Kıcı, R. Jhala, D. Tullsen, and D. Stefan. *Automatically eliminating speculative leaks from cryptographic code with blade*. POPL 2021. URL: https://arxiv.org/abs/2005.00294 https://www.youtube.com/watch?v=mCzbjGRIVeQ

- Canella, Claudio, et al. *A systematic evaluation of transient execution attacks and defenses*. 28th USENIX Security Symposium (USENIX Security 19). 2019. URL: https://arxiv.org/abs/1811.05441v3

Take away message

Focus on:

- security vs. performance is required
- hardare and software

- Changes to how we design hardware are required
  - all hardware developers should know which software abstractions are adopted

- Changes to the way we design software are required
  - all software engineers should have some notion of how processors behave and deal with microarchitectural complexity and, in general with all layers

- Also: no more HW and SW people!

Take away message (cont.)

- Software is eating the world
- Insecurity costs scale faster and faster
- Security is going to eat the software

End