Language Based Security (LBT)

Leaky Abstractions: When Hardware Undermines Language Security

Chiara Bodei, Gian-Luigi Ferrari

Lecture May 2 2025

Outline

Speculative Execution

Security issues

Cache analysis under speculation

Speculation

- Caches are **fast** when data is already there, **otherwise** we need to wait for the DRAM 😢
- **Idea:**
  - Upon branch on a value in memory (**uncached**), the CPU speculates on what's **probably right** to do and does that
  - When the value arrives*, if it was the wrong thing: **rollback**
- For that, the CPU is equipped with a **branch predictor**
  - **Roughly:** should the CPU take the next branch?
  - The CPU **trains** its branch predictor by **observing** what happened before
  - Modern branch predictors are complex, but for our purposes **just one bit**!

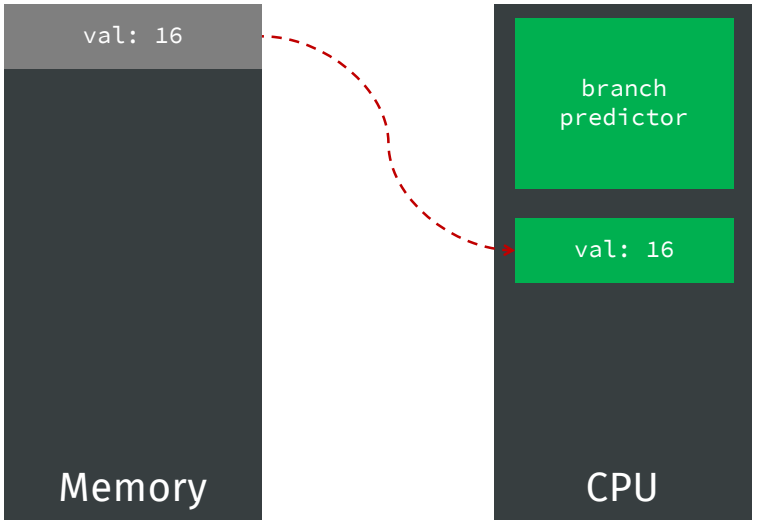
If you used cache during speculation, cache is dirty

* We change but not rollback value in the cache. This changes our analysis.

Speculation: when the CPU is right

Your branch predictor says true, so you move. If right, commit

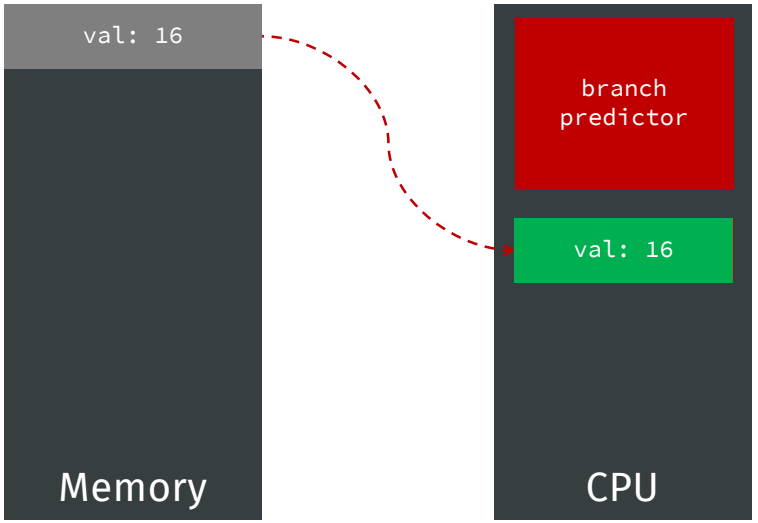
```
for (…)
  if val >= 0
    foo ();
  else
    bar ();
```



The CPU was right: commit the changes!

Speculation: when the CPU is wrong

```
for (…)
  if val >= 0
    foo ();
  else
    bar ();  rollback
```



When the CPU is wrong: rollback

Imagine that this is your CPU pipeline. You predicted that you would run lines 1 to 20, but on line 10 you jump to line 30:

1 2 3 4 5 6 7
2 3 4 5 6 7 8
3 4 5 6 7 8 9
4 5 6 7 8 9 10
5 6 7 8 9 10 11
6 7 8 9 10 11 12
7 8 9 10 11 12 13 ← This is where you realise you mispredicted
8 9 10 X X X 30
9 10 X X X 30 31 etc.

Side channels in this setting…

Designers are trying to optimize the average case!

- In the **worst** case the execution will be **slow**
- In the **best** case the execution will be **very fast**

Consider an attacker that just **observes** the *execution* of a program

- If it can learn some information from its observation, then there is a side channel!
- **Informally:** programs that resist against attackers able to measure time are said **constant-time programs**
  - **Actually:** wait till the end for a better (i.e., implementation-independent) definition

More precisely:

"A **side channel** is any observable side effect of computation that an attacker could measure and possibly influence." [Lawson, 2009]*

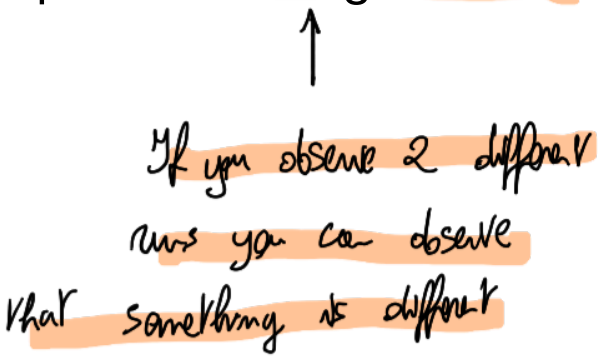
*[Lawson, 2009] Lawson, Nate. "Side-channel attacks on cryptographic software." *IEEE Security & Privacy* 7, no. 6 (2009): 65-68.

Speculative Execution

Speculative execution can introduce side effects on the cache state

- A cache hit only takes 1-3 CPU cycles while a cache miss may take up to a hundred CPU cycles
- Speculative execution can then lead to different execution times than traditional execution
- By observing the difference in the execution time of a victim program, the attacker may be able to deduce a certain amount of information about the secret: consequently, speculative execution leads to possible side channel attacks

- We need techniques for checking whether there are secret-dependent divergent cache behaviors



Divergent cache behaviour: example

Suppose we use a 32-KB fully-associative cache: 512 lines of 64 bytes each, initially empty ph, l1, l2, and p, which are mapped to different cache lines:

- l1, l2, and p occupy of 64 bytes each (one block)
- ph is an array that occupies $64 * 510$ bytes

Depending on the branching condition, either l1 or l2 may be loaded to the cache

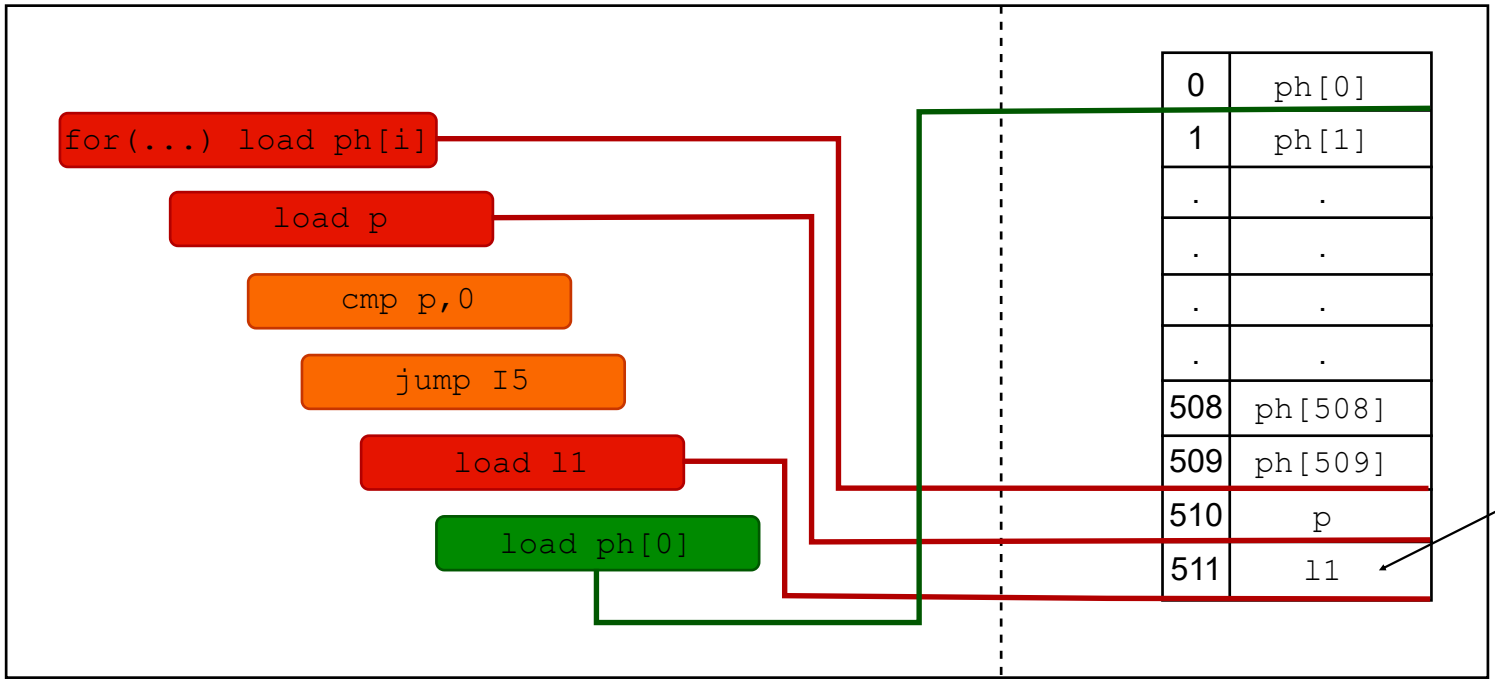
```
1  char ph[64*510], l1[64], l2[64], p;
2  reg char k;
3  for(reg int i=0;i<64*510; i+=64) load ph[i];
4  if(p==0)
5   load l1[0];
6  else
7   load l2[0];
8  load ph[k];
```

Divergent cache behaviour: example

Non speculative execution: 512 cache miss + 1 cache hit

Cache miss

Cache hit



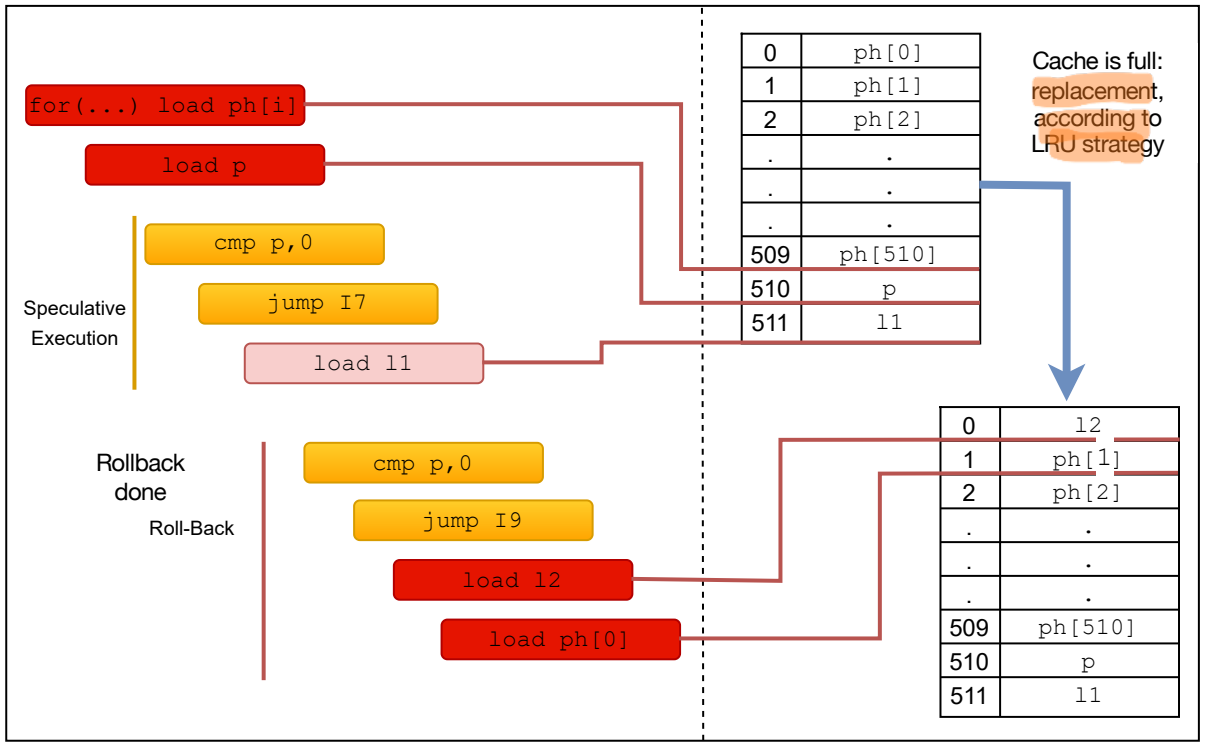
Divergent cache behaviour: example

Speculative execution: 514 cache miss ≠ 512 cache miss + 1 cache hit

Cache miss

Cache hit

You load l1 and discover you should have loaded l2.



Side Channel Detection: example

Suppose that:

- ph represents an S-box
- k stores the secret data, e.g., a cryptographic key, and
- the value of k is used as an index to access the S-Box ph

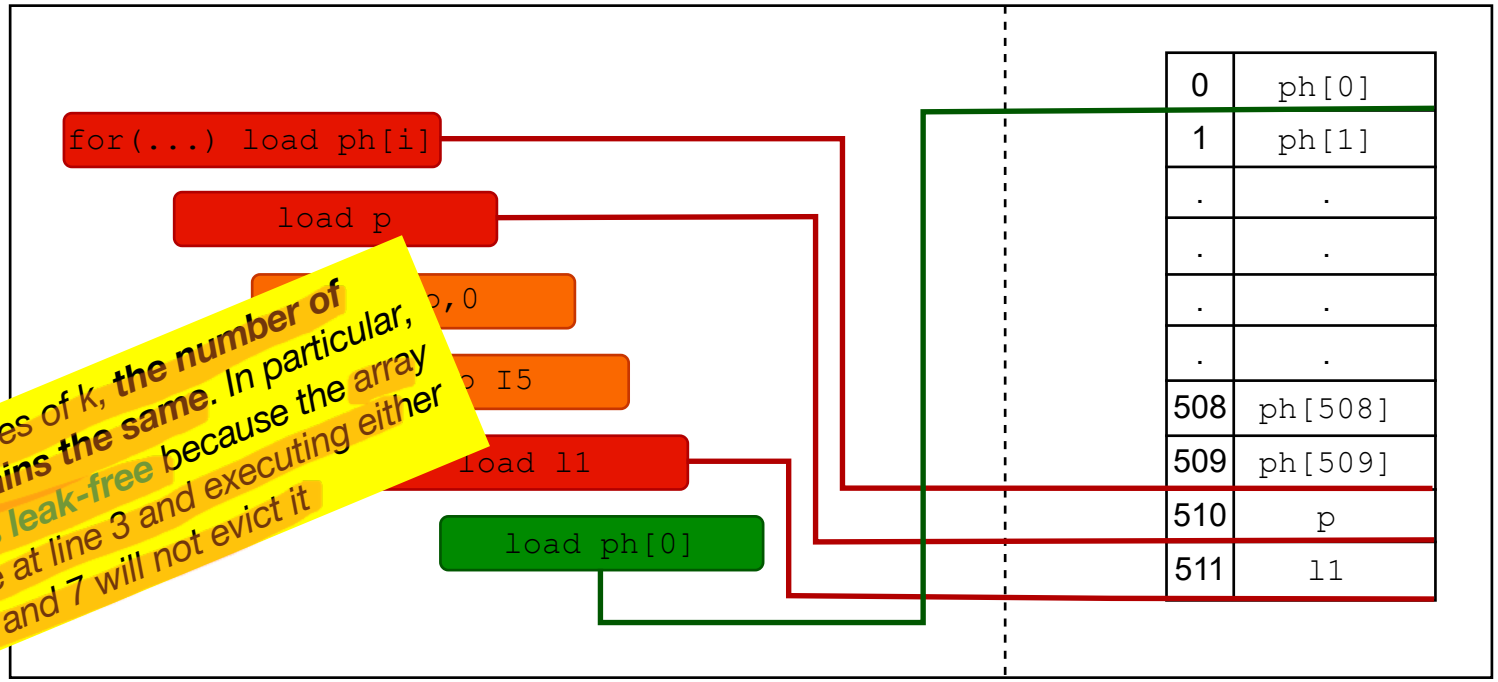
```
1  char ph[64*510], l1[64], l2[64], p;
2  reg char k;
3  for(reg int i=0;i<64*510; i+=64) load ph[i];
4  if(p==0)
5   load l1[0];
6  else
7   load l2[0];
8  load ph[k];
```

Divergent cache behaviour: example

Non speculative execution: no leak

Cache miss

Cache hit



Must hit

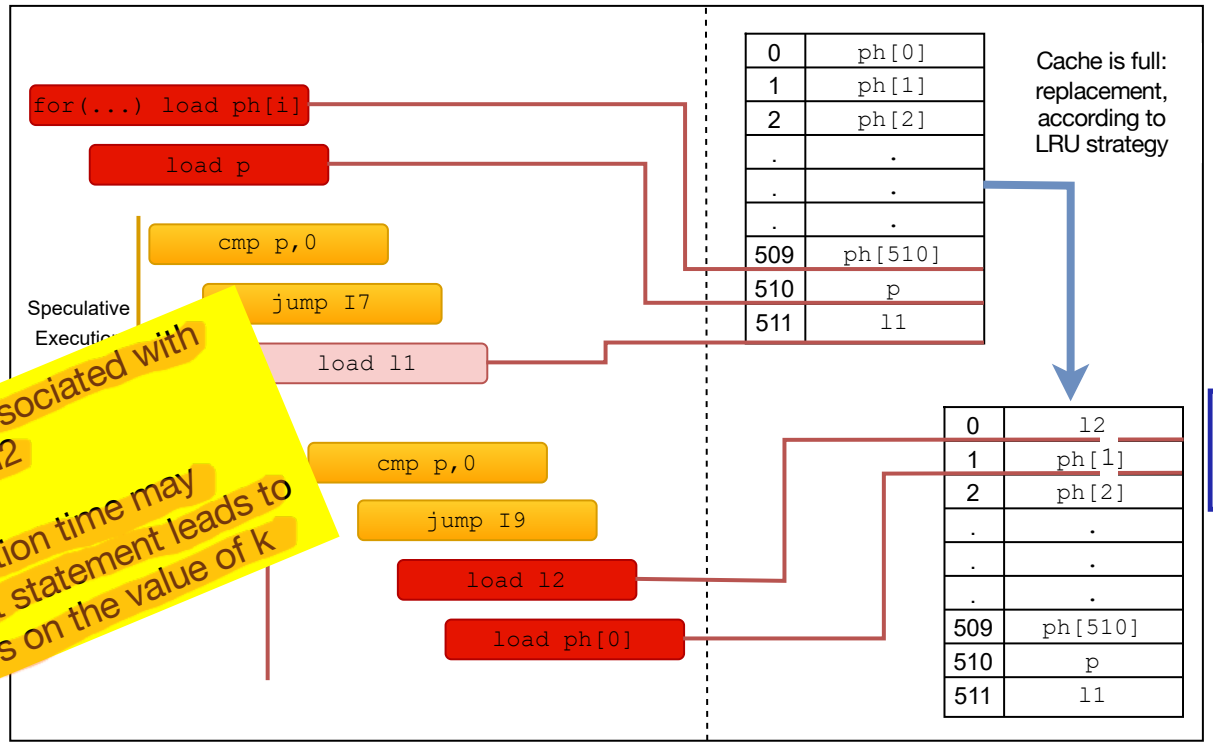
For all paths and values of k, **the number of cache misses remains the same**. In particular, accessing ph[k] is leak-free because the array is loaded to cache at line 3 and executing either branch at lines 5 and 7 will not evict it

Divergent cache behaviour: example

Speculative execution: leak

Cache miss

Cache hit



Some of the cache lines associated with `ph` will be evicted to load l2

The difference in execution time may be observed: if the last statement leads to a cache miss depends on the value of k

k = 0 ⇒ cache miss

k ≠ 0 ⇒ cache hit

`load ph[]`

Difference in execution time can be observed

Speculative Execution

- The previuously introduced static cache analysis is no longer sound under speculative execution

- We need to it so that it becomes sound again under speculative execution

Speculative Execution and cache analysis

- We present an extension of the previous static cache technique that predicts the cache behaviour in the **presence of speculation**
- The new analysis is able to capture possible cache divergent behaviour, due to memory-related instruction that causes a cache miss for some input value but a cache hit for some other input value
- Whenever there is a dependency between the secret and some divergent cache behaviors, i.e., the input value causing divergence is a secret (e.g., a password, security token, or cryptographic key), there is a **side-channel leak**

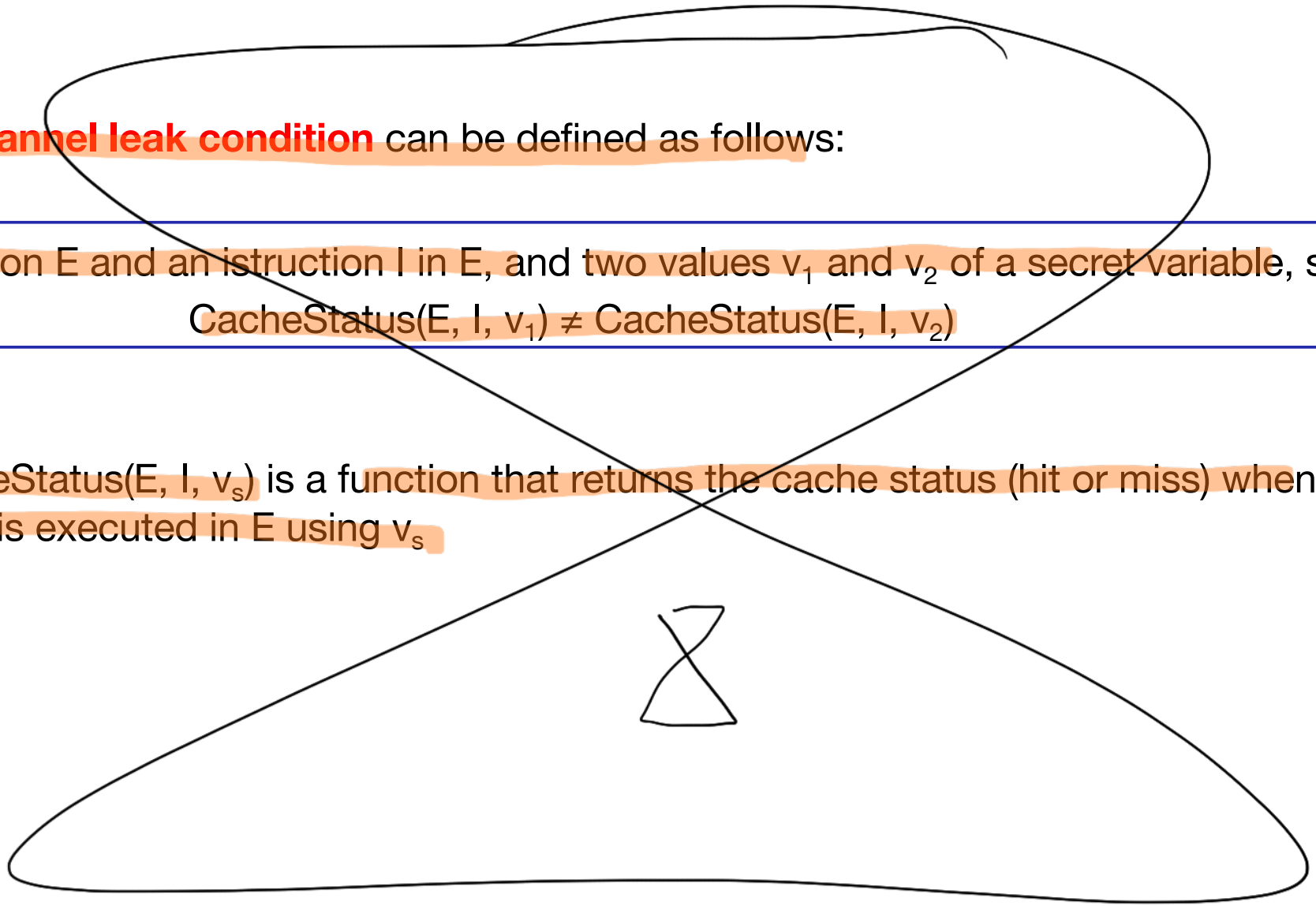
The Side-channel Leak Condition

The **side-channel leak condition** can be defined as follows:

$\exists$ an execution E and an istruction I in E, and two values v_1 and v_2 of a secret variable, s.t.

$$\text{CacheStatus}(E, I, v_1) \neq \text{CacheStatus}(E, I, v_2)$$

where $\text{CacheStatus}(E, I, v_s)$ is a function that returns the cache status (hit or miss) when instruction I is executed in E using v_s



Modeling the Speculative Execution

- We focus on **must analysis**
- To model all possible control flows produced by speculative executions, CFG is augmented with **Virtual Control Flow** and in a sinkable way merged in the framework together with what we have

A **virtual control flow** occurs at every if-else statement where the branching condition depends on some variables stored in memory, and it will be explored (speculatively) even if c is unsatisfiable

In case of mis-prediction, the rollback will re-direct the control to the other branch

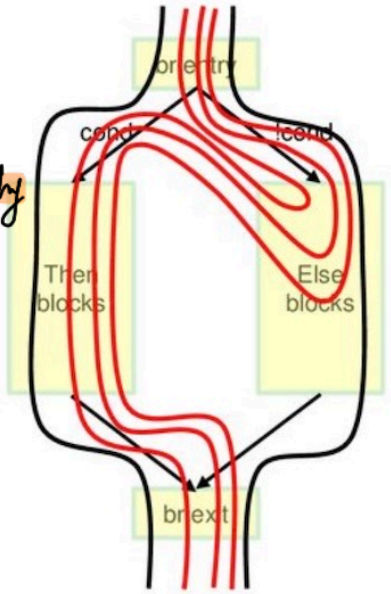
To model all these behaviors, we add the following **special nodes and edges** to the CFG

Modeling the Speculative Execution

VIRTUAL

CONTROL FLOW GRAPH

We added a virtual part in CFG corresponding to the possible speculative flows



Modeling the Speculative Execution

We need to extend our CFG with special nodes that relate to speculative exec.

For every branch that may be explored speculatively:

- vn_{start}: special CFG node that denotes the start of a virtual control flow
- vn_{stop}: special CFG node that denotes the end of a virtual control flow

Categories of edges (with n normal CFG node):

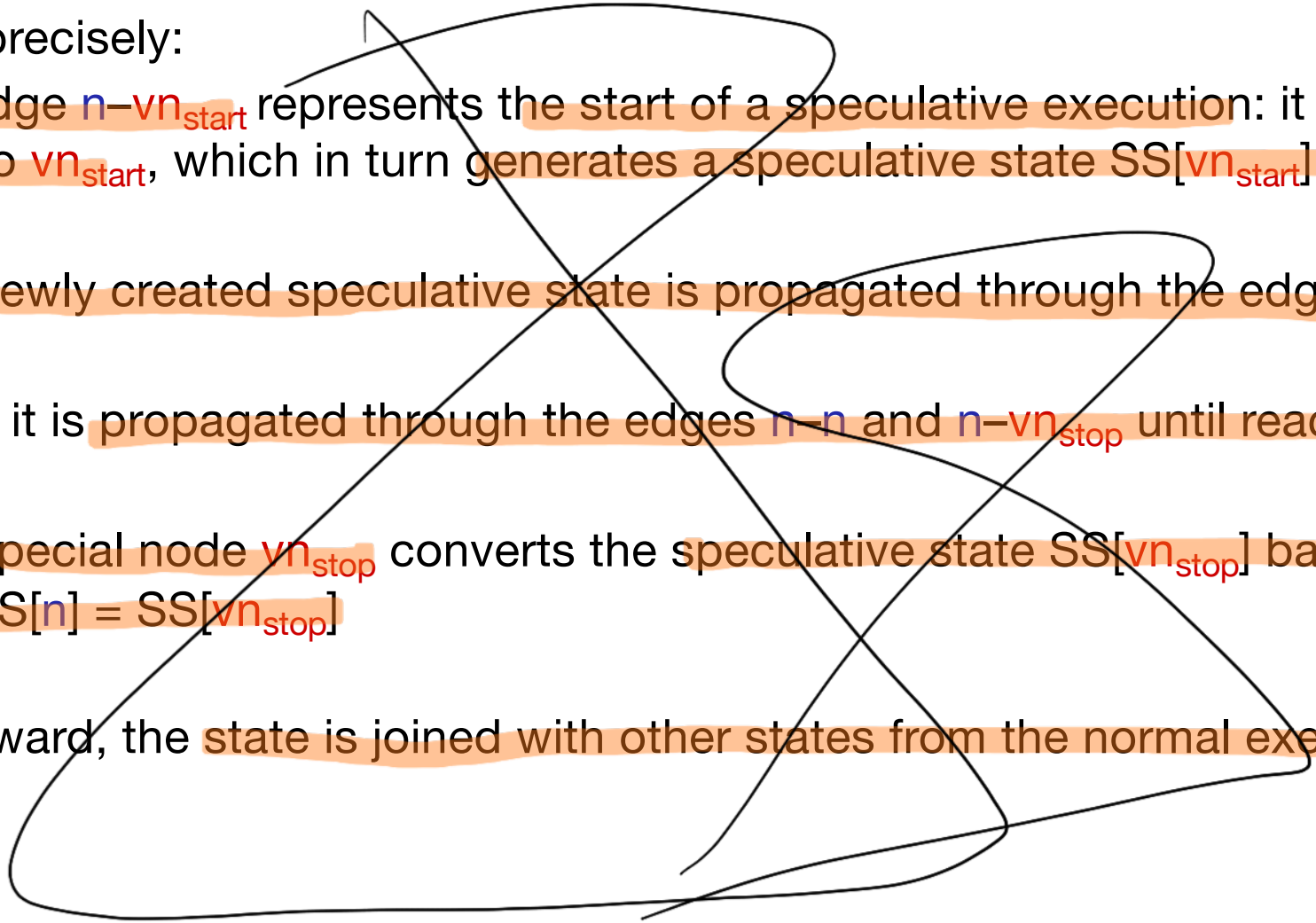
1. $n–vn_{start}$: start of a speculative execution
2. $vn_{start}–n$: connection point
3. n–n: standard edges
4. $n–vn_{stop}$: connection point
5. $vn_{stop}–n$: end of speculative execution

We also need to extend edges for speculative flow we can traverse (virtual)

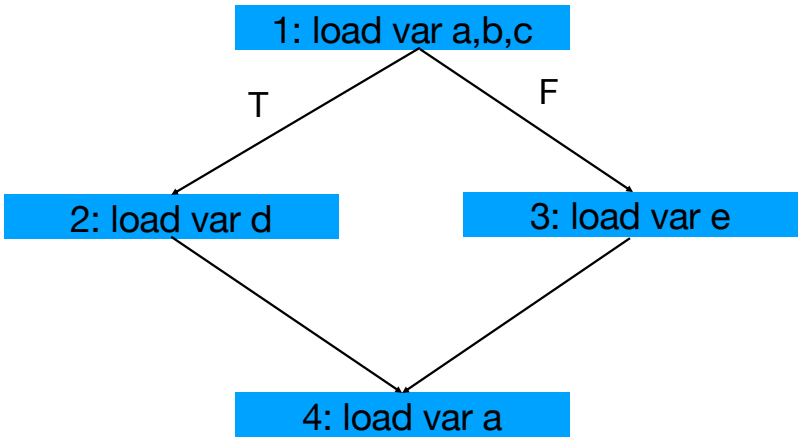
Modeling the Speculative Execution

More precisely:

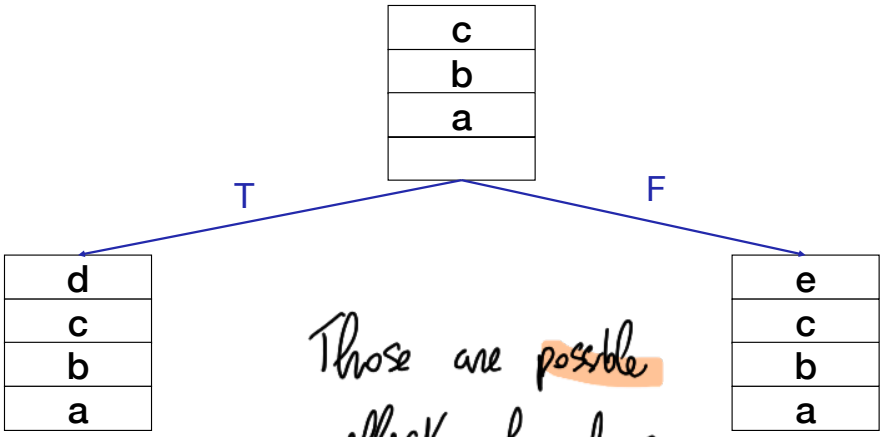
- the edge n–vn_{start} represents the start of a speculative execution: it feeds the state S[n] to vn_{start}, which in turn generates a speculative state SS[vn_{start}] = S[n]
- the newly created speculative state is propagated through the edge vn_{start}–n
- next, it is propagated through the edges n–n and n–vn_{stop} until reaching vn_{stop}–n
- the special node vn_{stop} converts the speculative state SS[vn_{stop}] back to the normal state S[n] = SS[vn_{stop}]
- afterward, the state is joined with other states from the normal execution



An example: normal control flows



Must analysis



Those are possible effects of choosing branches

Strategies for merging speculative control flows

In the speculative analysis, we also need to decide **when to join** the normal and the speculative control flows

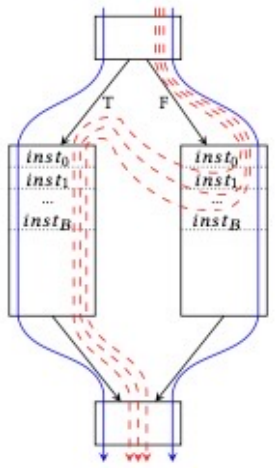
There are more than one possible strategies

We will see four of them:

a) No merging
b) Merging after branch
c) Merging before branch
d) Merging into normal flow

- Idea is provably analysis also for speculative execution
- So you predict one branch and if you have a misprediction you need to move to another branch. And you should put things together in terms of cache info. You can do this in a lot of ways.

No merging



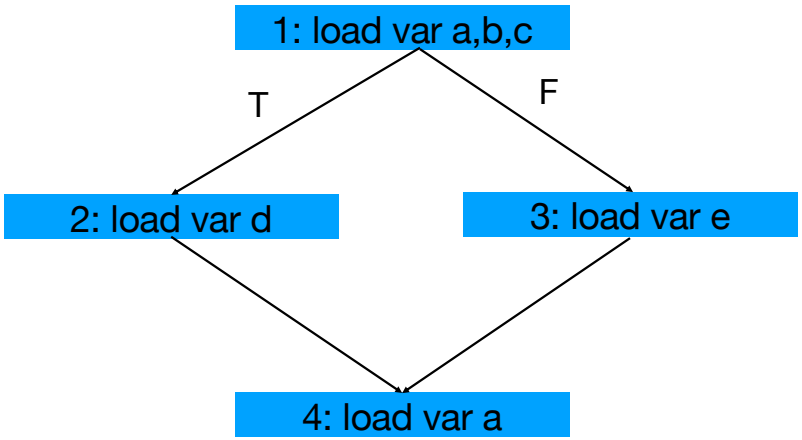
(a) flows without merging

For each **if-else statement**,
we add virtual control flow edges from instructions in one of the branch to the entry node of the other branch under the same branching condition

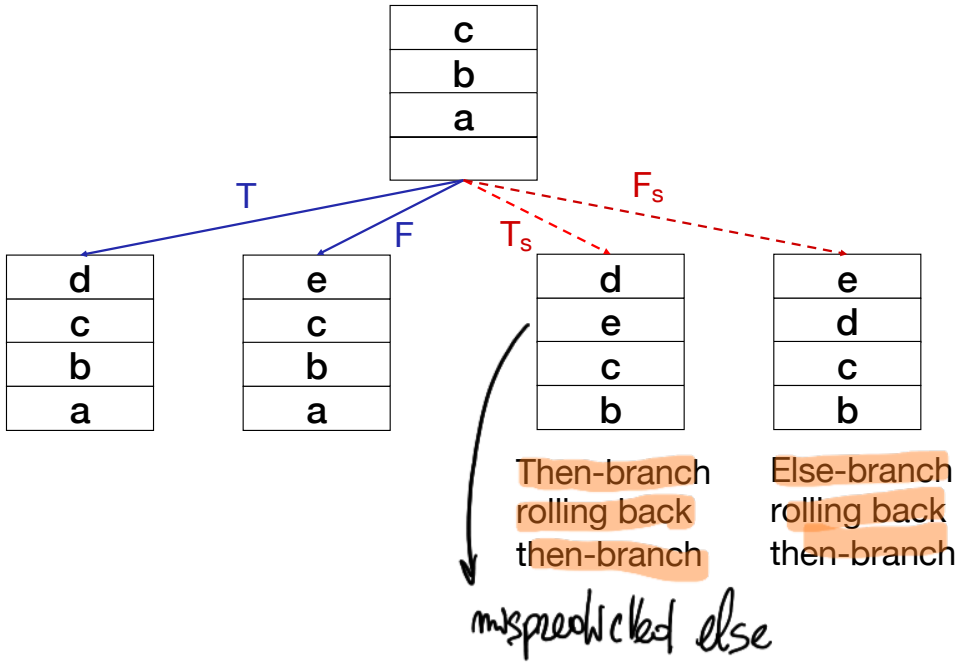
In the figure, the edges of the speculative else branch are added to the entry node of then branch (after the rollback)

- Blue solid lines represent normal executions
- **Red dashed lines** represent virtual control flows associated with speculative executions of the else-branch
- There are many red lines because the roll-back point may occur at any moment within the maximum speculation depth
- Virtual control flows associated with the then-branch are omitted in the figure for clarity

An example: speculative control flows



Must analysis

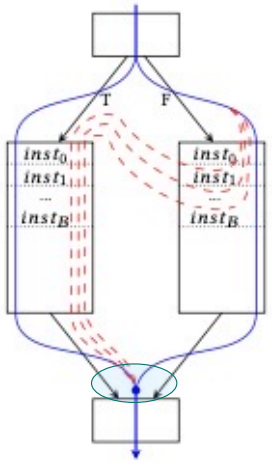


No merge strategy

Merging after branch

- Merge everything after the branch

All strategies can be proved as sound approximations



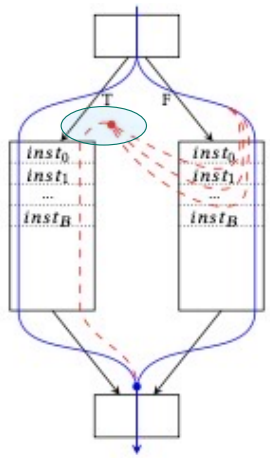
(b) merged after branch

Since the executions before the branch entry node are identical, they are merged without losing accuracy

- Blue solid lines represent normal executions
- Red dashed lines represent virtual control flows associated with speculative executions of the else-branch
- There are many red lines because the roll-back point may occur at any moment within the maximum speculation depth
- Virtual control flows associated with the then-branch are omitted in the figure for clarity

Merging before branch

• Merge before the branch



(c) merged before branch

More approximate solution

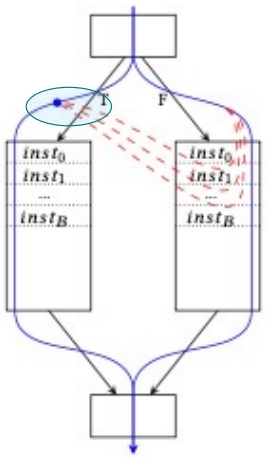
- It merges all speculative states of the else-branch before reaching the then-branch, at the end of speculation
- The merged speculative state is propagated through the then-branch before it is merged with the normal state

- Blue solid lines represent normal executions
- Red dashed lines represent virtual control flows associated with speculative executions of the else-branch
- There are many red lines because the roll-back point may occur at any moment within the maximum speculation depth
- Virtual control flows associated with the then-branch are omitted in the figure for clarity

Merging into normal flow

- Merge together with the normal flow at entry node of correct branch node

More aggressive over-approximation
It merges the speculative
states with the non-speculative state at
the entry node of the then-branch

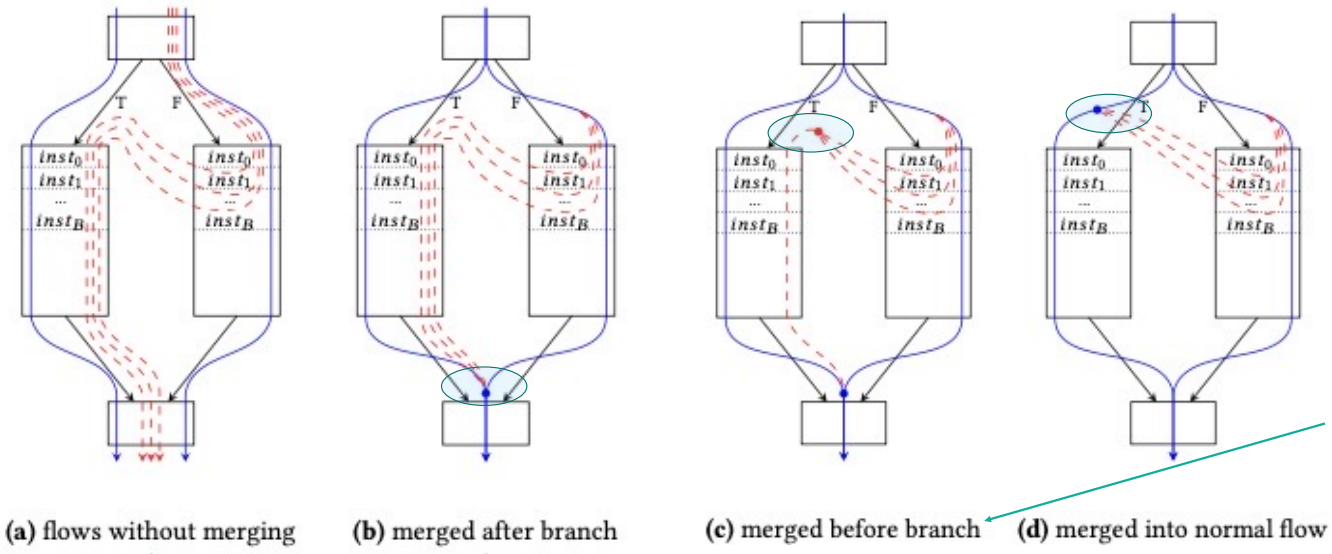


(d) merged into normal flow

- Blue solid lines represent normal executions
- Red dashed lines represent virtual control flows associated with speculative executions of the else-branch
- There are many red lines because the roll-back point may occur at any moment within the maximum speculation depth
- Virtual control flows associated with the then-branch are omitted in the figure for clarity

Strategies for merging speculative control flows

In every strategy, result is a sound approximation; in general the later the merging occurs, the more accurate the result is: you keep things as separate as possible. Of course late merging is more expensive.



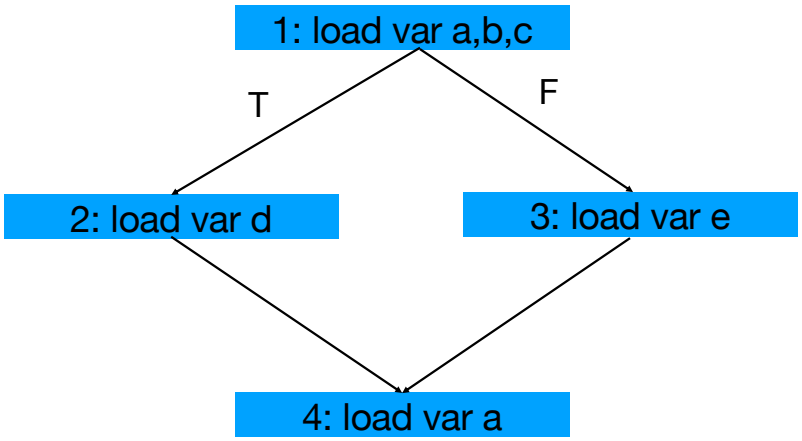
(a) flows without merging (b) merged after branch (c) merged before branch (d) merged into normal flow

Just-in-Time merging
The best merging strategy: more accurate results, w.r.t. (d) and equally fast

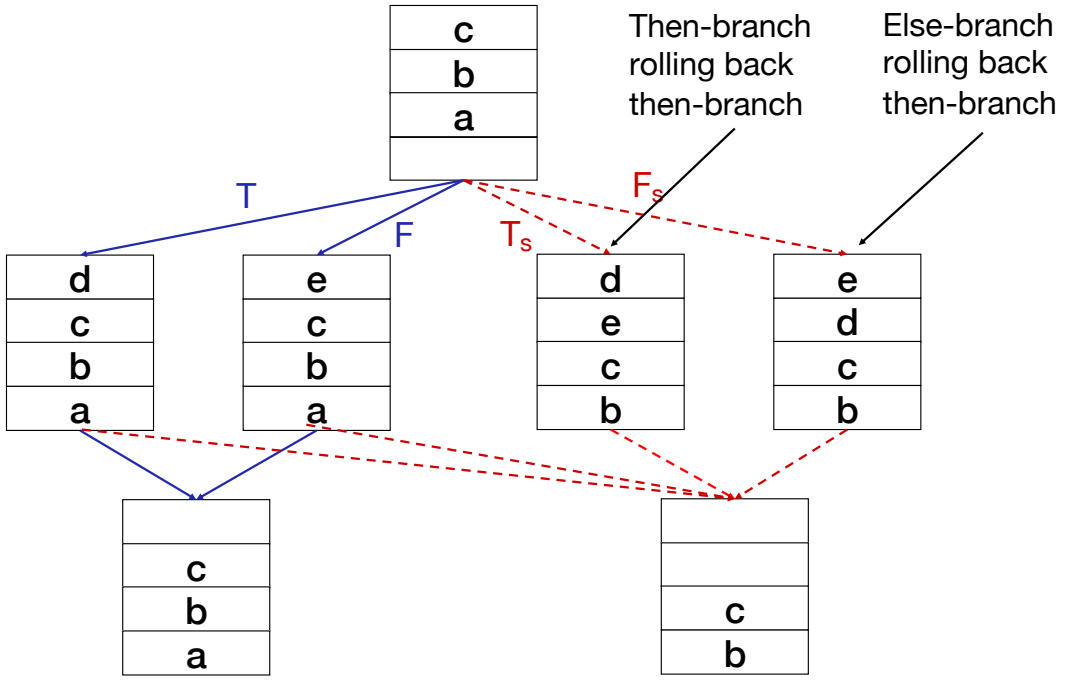
- Regardless of the merging strategy, however, the result is a sound overapproximation: as the join operator (⊔) used for merging is over-approximated, strategies b), c) and d) are all a sound over-approximation of a)
- Since every time state merging occurs, it may lose information, in general, the later that merging occurs, the more accurate the result is, but there is no guarantee
- Furthermore, late merging may lead to a more expensive analysis

- Authors of paper, after experimentation noted that merged before branch approach is a good compromise between performance and accuracy. They called it JIT merging.

Just-in-Time merging: an example



Must analysis



Only c and b are guaranteed to result in **cache hits** thus capturing the side effect of speculative execution

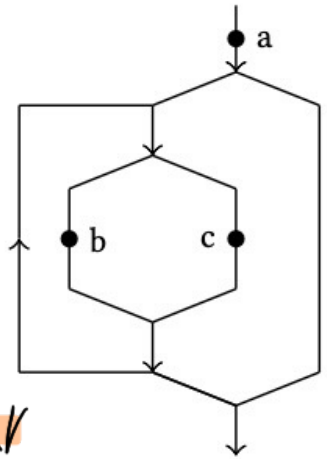
- like pulling together things coming together from speculative and non after the branch.

This is more precise than non speculative strategy.

A loop-related problem

- First, variable **a** is loaded into the cache
- Then, inside the loop, every time the branch is executed, Age(**a**) is increases by 1
- After the join, however, neither **b** nor **c** will be in the cache
- Thus, eventually, **a** is evicted from the cache as well

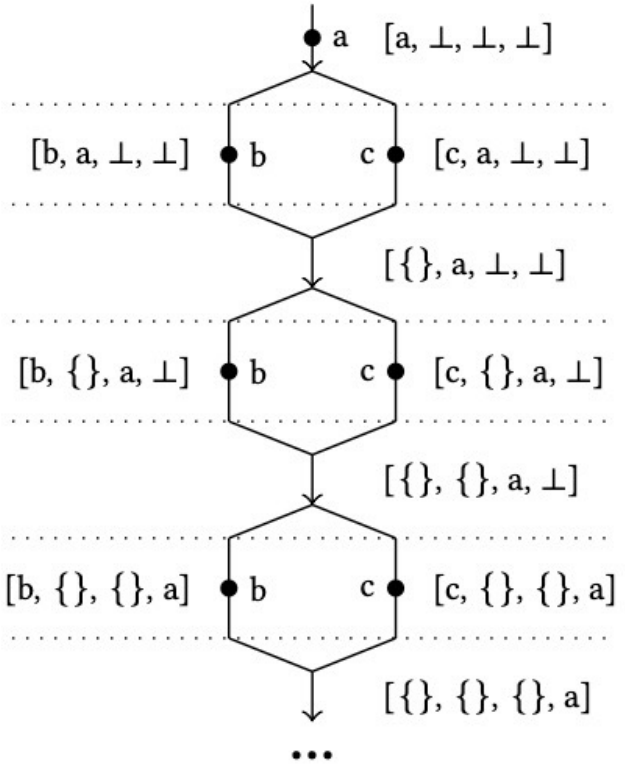
This is not accurate because, during the actual execution, a will never be evicted



- There is a loop problem: if you use classical cache analysis you lose info in the end, so to solve problem use existing analysis with shadow variable.

The sequence of memory accesses in three iterations

The abstract cache states are listed on both sides at each memory access and merge point



Possible optimizations: EH analysis

A possible optimization is the **Exists Hit (EH) Analysis**

That allows to refine he join operator (⊔) by adding extra information into the cache state

We introduce

- **Shadow variable ∃v**: represents the youngest cache line that may hold v along some program path

Kinda inserting may analysis with must analysis

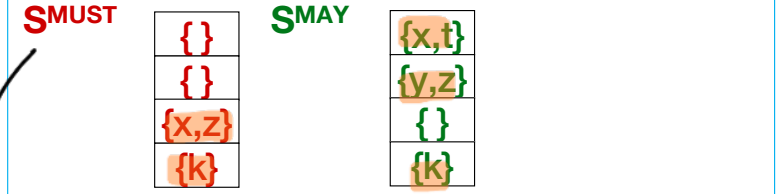
- States are therefore $S = \langle Age(v_1),\ldots, Age(v_n), Age(\exists v_1),\ldots, Age(\exists v_n)\rangle$

Shadow variables: merging

Whenever two states are merged and v appears in only one of the two states, ∃v will remain in the merged state, while the variable v will not

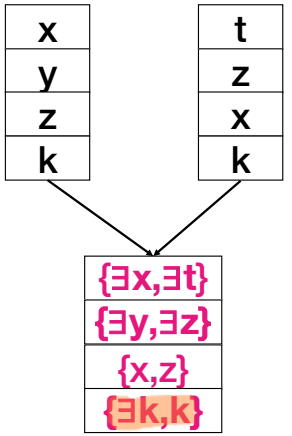
Here both ∃y and ∃t appear in the merged state, i.e., there exists some path in which y (or t) is cached

The modified state can be viewed as two separate states:



- S^{MUST} represents the oldest cache lines along all paths, and
- S^{MAY} represents the youngest cache lines along some paths

Every state split in a must + may pair

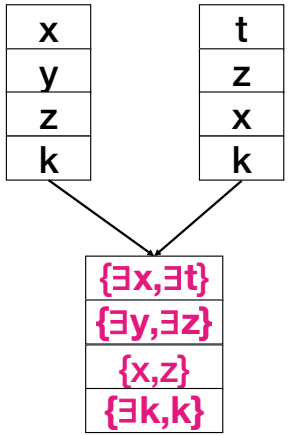


- for runtime use for analysis look at formal merging rule

Shadow variables: merging

Whenever two states are merged and v appears in only one of the two states, **∃v** will remain in the merged state, while the variable v will not

Here both ∃y and ∃t appear in the merged state, i.e., there exists some path in which y (or t) is cached



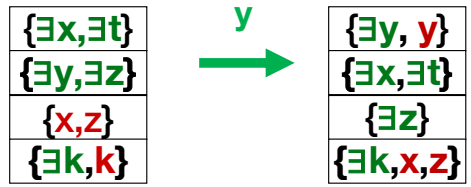
Given two states

- $S = \langle Age(v_1),\ldots, Age(v_n), Age(\exists v_1),\ldots, Age(\exists v_n)\rangle$ and
- $S' = \langle Age'(v_1),\ldots, Age'(v_n), Age'(\exists v_1),\ldots, Age'(\exists v_n)\rangle$, as follows:
- $S'' = S \sqcup S' =$

$\langle \mathbf{max}(Age(v_1),Age'(v_1)), \ldots, \mathbf{max}(Age(v_n),Age'(v_n)),$
$\mathbf{min}(Age(\exists\, v_1),Age'(\exists\, v'_1)),\ldots,\mathbf{min}(Age(\exists\, v_n),Age'(\exists\, v_n))\rangle$

Shadow variables: tranfer function

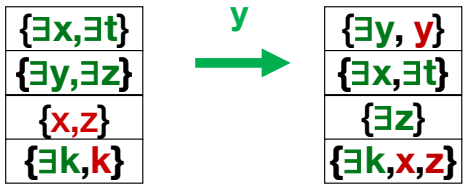
The transfer function are revised as well, by using the number of shadow variables younger than or equal to **u** to refine the rule for aging the regular variable **u**



They need to be redesigned to keep information together.

Shadow variables: tranfer function

The transfer function are revised as well, by using the number of shadow variables younger than or equal to **u** to refine the rule for aging the regular variable **u**



Not necessary to know details:

Transfer function

Transfer(S, inst(v))

- Age′(v) = 1 (in all possible executions)
- Age′(∃v) = 1 (1 is also a safe upper bound on its minimal age)
- Age(∃u) ≤ Age(∃v) ⇒ Age′(∃u) = Age(∃u) + 1
- Age(∃u) > Age(∃v) ⇒ Age′(∃u) = Age(∃u)
- $N_{Young}(u) \geq Age(u) \Rightarrow Age'(u) = Age(u) + 1$
- $N_{Young}(u) < Age(u) \Rightarrow Age'(u) = Age(u)$
- Age′(w) = Age(w) for any other variable

$N_{Young}(u) = |\{\exists w \mid Age(\exists w) \leq Age(u) \wedge w \neq u\}|$

$N_{Young}(u)$ represents the number of variables younger than or equal to u

Shadow [illegible]s: tr[illegible]tion

The transfer functi[illegible] shadow variables younger than or eq[illegible] variable **u**

{∃x,∃t}
{∃y,∃z}
{x,z}
{∃k,k}

- Age′(y) = 1
- Age′(∃y) = 1
- Age(∃x) = 1 ≤ 2 = Age(∃y) ⇒ Ag[illegible]
- Age(∃z) = 2 = Age(∃y) ⇒ Age′(∃z) [illegible] + 1
- Age(∃k) = 4 > 2 = Age(∃y) ⇒ Age′(∃k) = Age(∃k)
- $N_{Young}(x) = 1 \geq 1 = Age(x) \Rightarrow Age'(x) = Age(x) + 1$
- $N_{Young}(z) = 1 \geq 1 = Age(z) \Rightarrow Age'(z) = Age(z) + 1$
- Age′(k) = Age(k)

[illegible]ble executions)

[illegible]a safe upper bound on

[illegible] ⇒ Age′(∃u) = Age(∃u) + 1

[illegible]Age(∃v) ⇒ Age′(∃u) = Age(∃u)

[illegible]u) ≥ Age(u) ⇒ Age′(u) = Age(u) + 1

[illegible]ng(u) < Age(u) ⇒ Age′(u) = Age(u)

[illegible]e′(w) = Age(w) for any other variable

[illegible]oung(u) = |{∃w | Age(∃w) ≤ Age(u) ∧ w ≠ u}|

$N_{Young}(u)$ represents the number of variables younger than or equal to u

Shadow variables: tranfer function

In general, S^{MUST} is either the same as, or more accurate than the result of the original analysis

This is what we get

This is due to the following

modified rule for aging the variable **u**:

$(Age(u) < Age(v)) \wedge (N_{Young}(u) < Age(u)) \Rightarrow Age'(u) = Age(u)$

must st

This optimization is safe because, if the number of shadow variables that are younger than or equal to **u** ($N_{Young}(u)$), is less than the current age of u, there must be younger cache lines to hold all of them

In such a case, we should not (unnecessarily) increase the age of **u**

Transfer function

Transfer(S, inst(v))

- $Age'(v) = 1$ (in all possible executions)
- $Age'(\exists v) = 1$ (1 is also a safe upper bound on its minimal age)
- $Age(\exists u) \leq Age(\exists v) \Rightarrow Age'(\exists u) = Age(\exists u) + 1$
- $Age(\exists u) > Age(\exists v) \Rightarrow Age'(\exists u) = Age(\exists u)$
- $N_{Young}(u) \geq Age(u) \Rightarrow Age'(u) = Age(u) + 1$
- $N_{Young}(u) < Age(u) \Rightarrow Age'(u) = Age(u)$
- $Age'(w) = Age(w)$ for any other variable

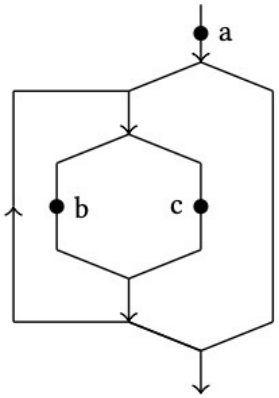
$N_{Young}(u) = |\{\exists w \mid Age(\exists w) \leq Age(u) \wedge w \neq u\}|$
$N_{Young}(u)$ represents the number of variables younger than or equal to u

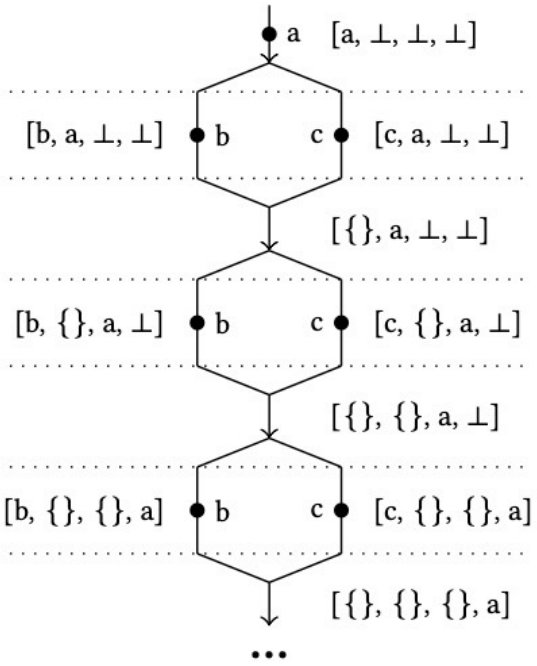
- In general idea is: if you do both analyses without optimization and with, the must analysis is either same as or more accurate than before.

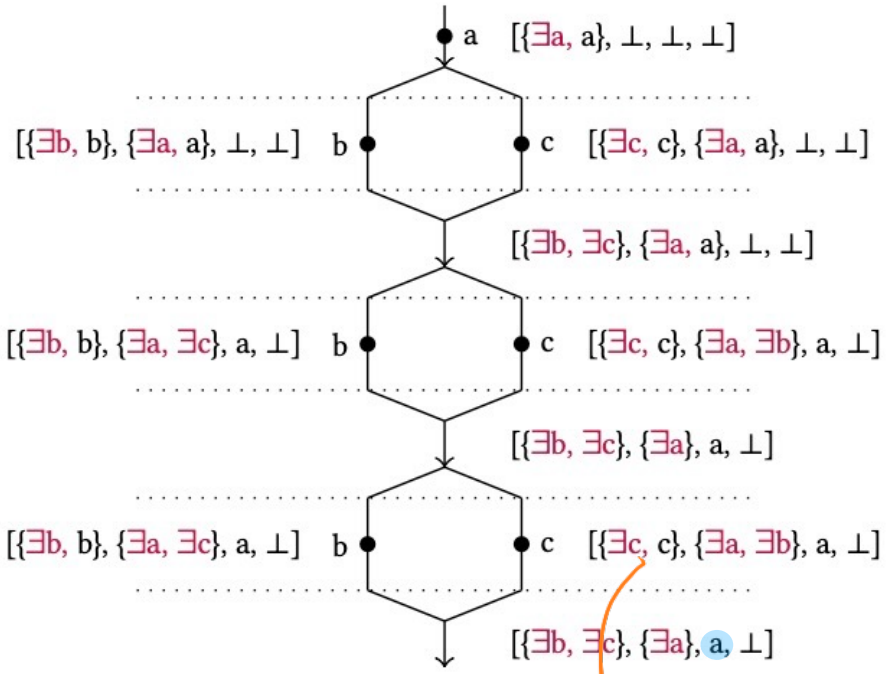
Shadow variables: example

With the shadow variables, the cache states are able to reach the fixed-points after only three iterations and **avoid evicting a**

The sequences of memory accesses



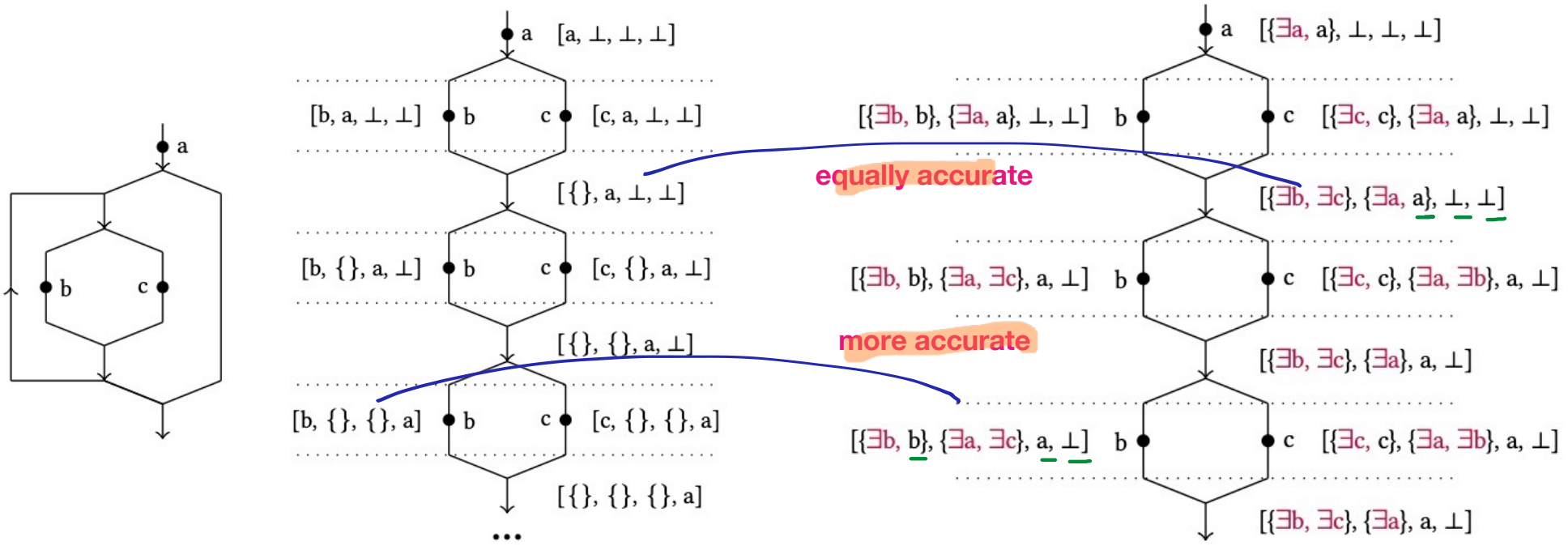




Sostituisci a ∃x => {∃x, x} quando accedo

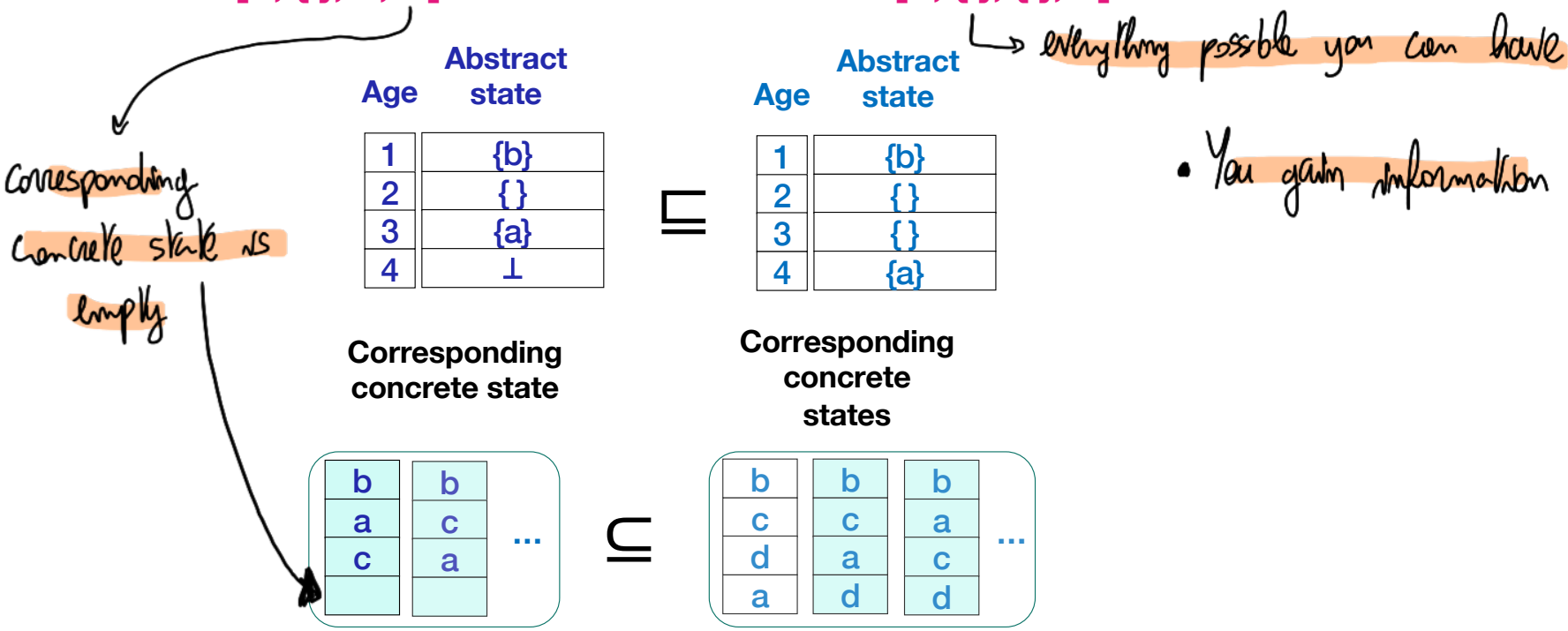
Shadow variables: example

With the shadow variables, the cache states are able to reach the fixed-points after only three iterations and **avoid evicting a**



Shadow variables: example

[b, { }, a, ⊥] is more accurate than [b, { }, { }, a]



- {2,3} = Concrete Age(a) ≤ Abstract Age(a) = 3 < 4
- {2,3,4} = Concrete Age(a) ≤ Abstract Age(x) = 4

Summary

We presented an abstract interpretation technique that can soundly analyze a program under speculative execution

upgrades detection of side channel leaks

Experimental results show that the method can detect many cache misses and side-channel leaks overlooked by a stateof- the-art non-speculative analysis technique

Beyond cache static analysis

The discussed analyses are able to classify memory accesses as:

- "always hit", or
- "always miss", or
- "definitely unknown"

- From here on we can proceed with symbolic execution and model checking that goes over the analysis and gives you hint on points that might also give problems. (1)

Due to the incomplete nature of the analysis, the exact status of some blocks remains unknown

To classify these remaining blocks using **model checking** and relying on **symbolic execution** (assuming all paths to be feasible)

(1) Not only is the model-checking phase sound, i.e. its classifications are correct, it is also complete relative the our control-flow-graph model, i.e., there remain no unclassifed accesses

Bibliography

- Meng Wu, Chao Wang. *Abstract interpretation under speculative execution*. PLDI 2019
- Meng Wu, Chao Wang. *Abstract Interpretation under Speculative Execution*. CoRR abs/1904.11170 (2019)

- Valentin Touzeau, Claire Maiza, David Monniaux, and Jan Reineke. 2017. *Ascertaining uncertainty for efficient exact cache analysis*. In International Conference on Computer Aided Verification. 22–40.

End