Language Based Security (LBT)

Leaky Abstractions: When Hardware Undermines Language Security

Chiara Bodei, Gian-Luigi Ferrari

Lecture May 16 2025

Outline

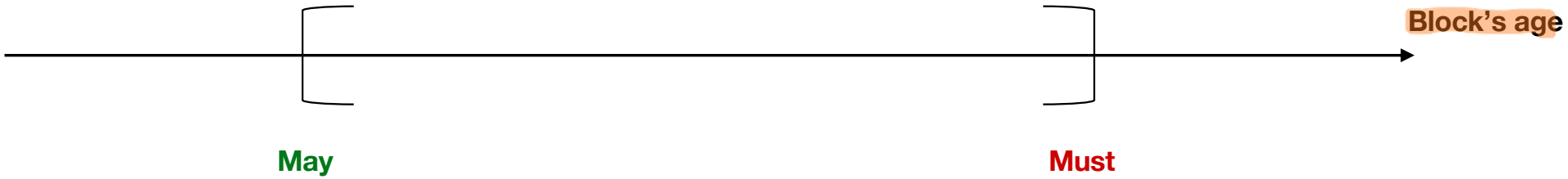
Cache analysis and model checking

Must and May analyses

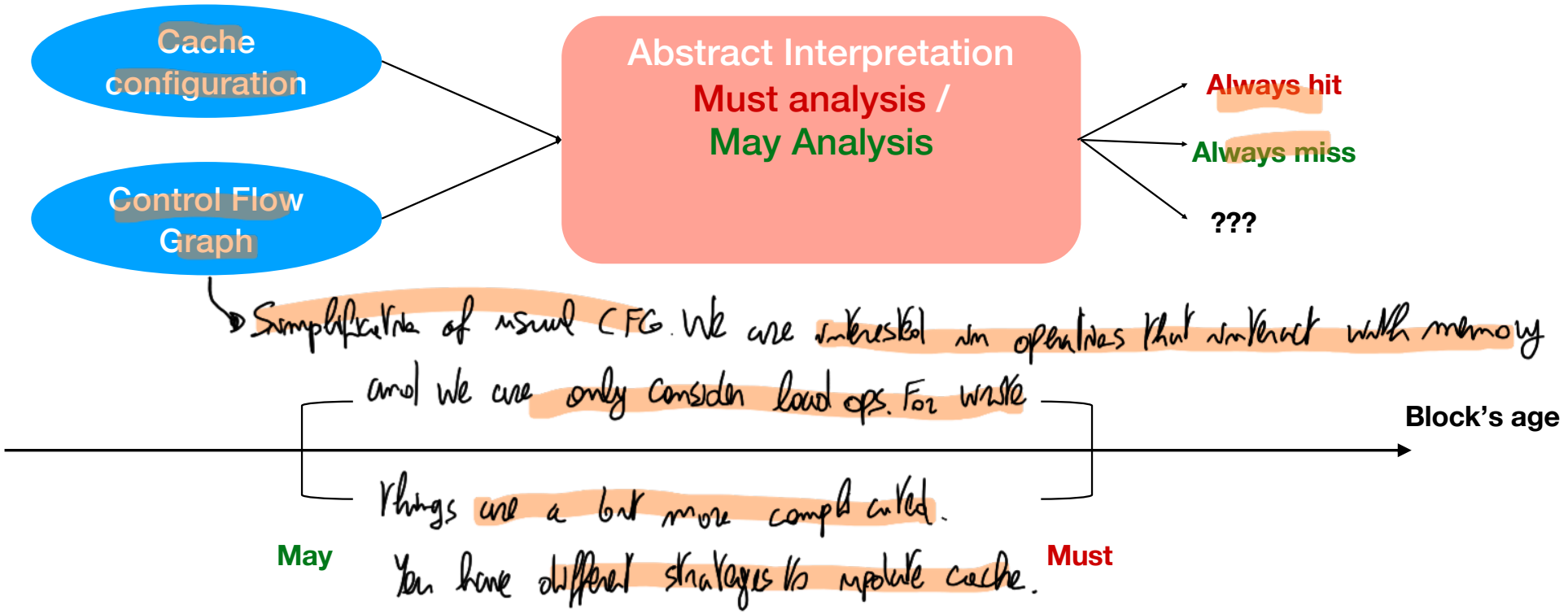
- A **May analysis** needs to compute, at each program location, a safe lower bound of ages

- **Must analysis** computes, at each program location, a safe upper bound of ages

MayAbstract age(v) ≤ Concrete age ≤ MustAbstract age(v)



Cache Analysis



Cache analysis

May analysis

[a, {}, {}, {}]

L = [b, a, {}, {}]

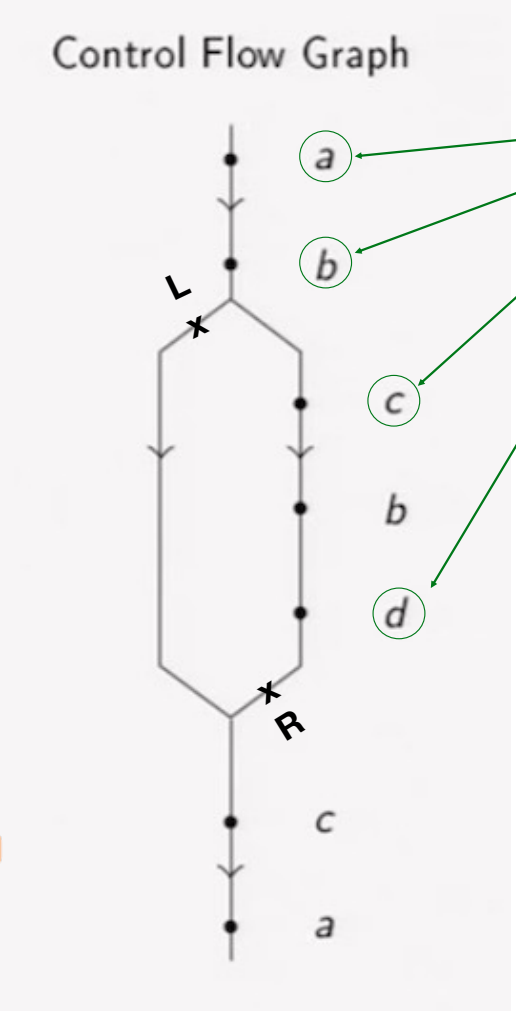
R= [d, b, c, a]

L ⊔ R=

[{b,d}, a, c, {}]

$S \sqcup S' = \langle \min(Age(v_1), Age'(v_1)), \ldots, \min(Age(v_n), Age'(v_n)) \rangle$

a has age at least 1



Misses : they are not there at the beginning of the execution

Cache analysis

May analysis

[a, {}, {}, {}]

L = [b, a, {}, {}]

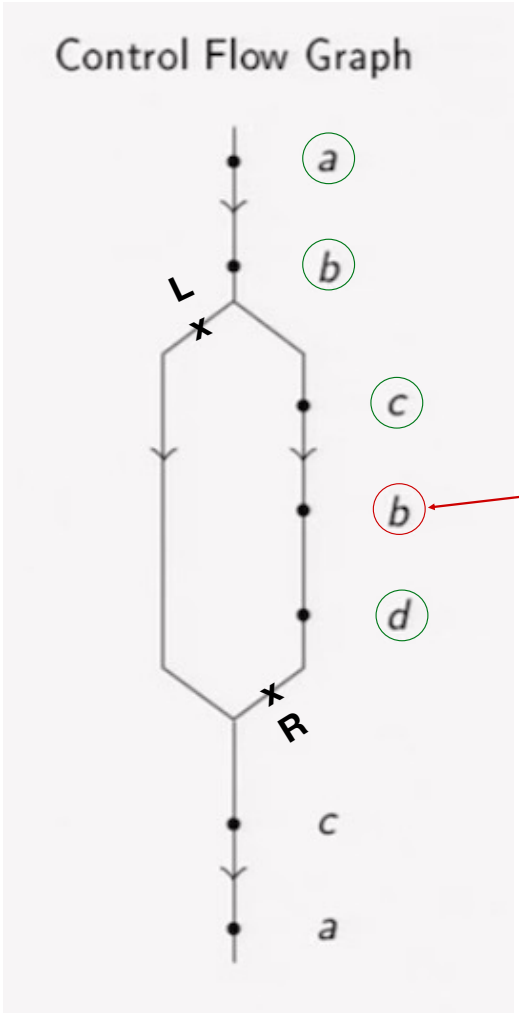
R= [d, b, c, a]

L ⊔ R=

[{b,d}, a, c, {}]

$S \sqcup S' = \langle \mathbf{min}(Age(v_1), Age'(v_1)), \ldots, \mathbf{min}(Age(v_n), Age'(v_n)) \rangle$

a has age at least 1



Must analysis

[a, {}, {}, {}]

L = [b, a, {}, {}]

Hit This analysis says that b is always hit! But we still have accesses that are unclassified

R= [d, b, c, a]

L ⊔ R=

[{}, b, {}, a]

$S \sqcup S' = \langle \mathbf{max}(Age(v_1), Age'(v_1)), \ldots, \mathbf{max}(Age(v_n), Age'(v_n)) \rangle$

unclassified

Union and minimal age → here interception with maximal age: something that holds in all the executions

EH and EM analyses

- **(EH) analysis** computes safe upper bounds on minimal ages
- **(EM) analysis** computes safe lower bounds on maximal ages

To gain new info, we don't do Always, but Exist Analysis.
EH: finds when we have a hit by computing existence of 1 path in which you have a hit. EM does the opposite.

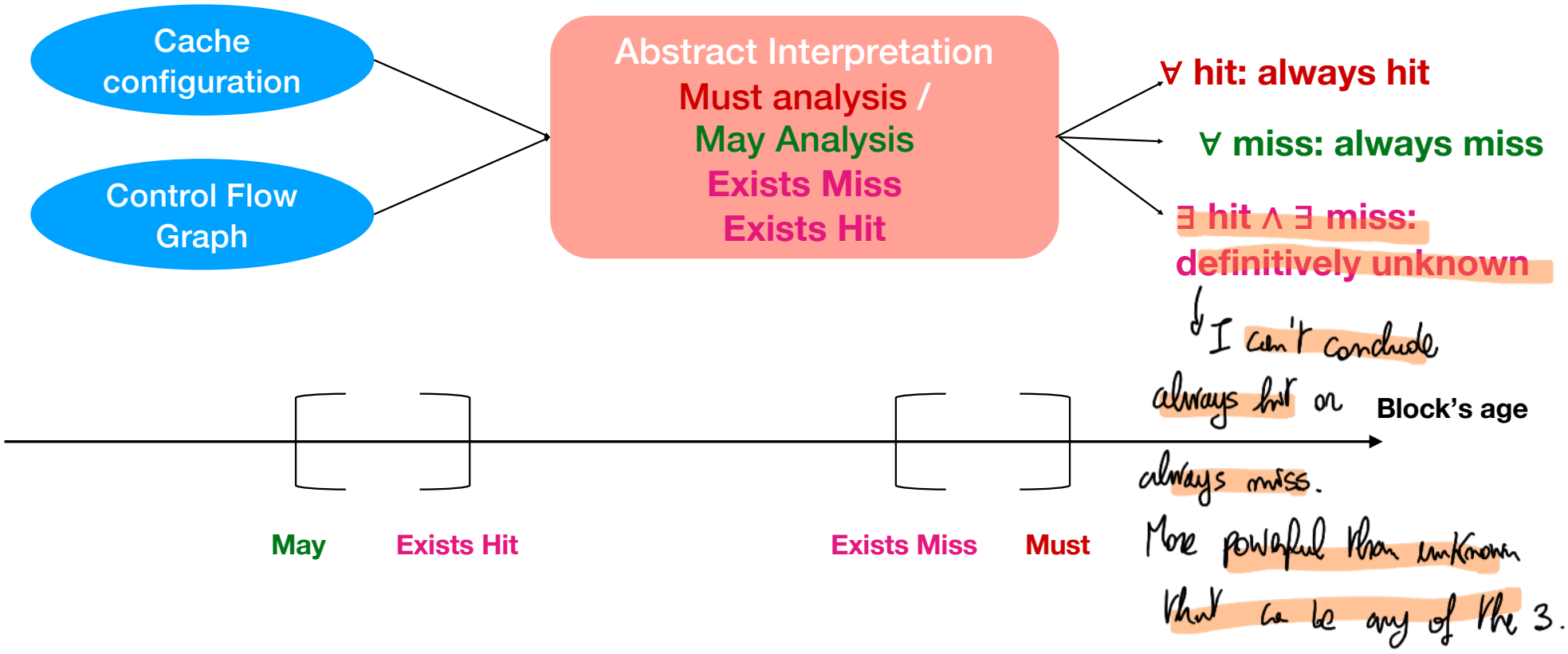
MayAbstract age(v) ≤ Concrete age ≤ MustAbstract age(v)

MayAbstract age(v) ≤ Concrete age ≤ ExistHit age(v) ①

ExistMiss age(v) ≤ Concrete age ≤ MustAbstract age(v)

① Upper Bound on minimal age.

Cache Analysis



Cache analysis

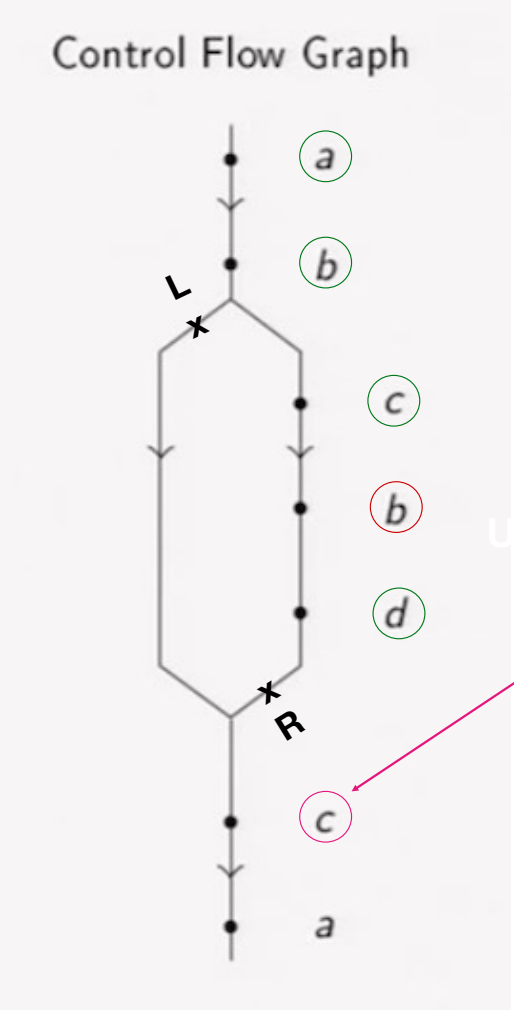
Must analysis

[a, {}, {}, {}]

L = [b, a, {}, {}]

R= [d, b, c, a]

L ⊔ R=
[{}, b, {}, a]



Exist miss analysis

Lower bound on max ages

[a, {}, {}, {}]

L = [b, a, {}, {}]

Miss in one path

R= [d, b, c, a]

L ⊔ R=
[b, {}, a, {}]
[c, {}, b, a]

Using exist analysis we conclude there is a hit on our state as a max age, so can't conclude

max instead of anything for min bounds

For C, opposite for V

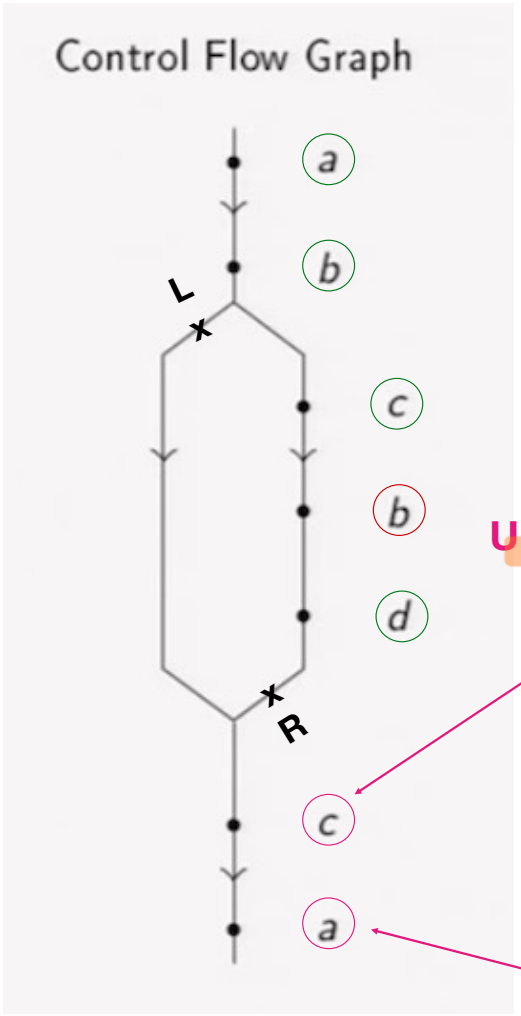
[{}, b, {}, a] "can't say there exist a definite miss for a"

Cache analysis

May analysis

[a, {}, {}, {}]

L = [b, a, {}, {}]



R= [d, b, c, a]

L ⊔ R=
[{b,d}, a, c, {}]

a has age at least 1

Exist hit analysis

[a, {}, {}, {}]

Upper bound on max ages

L = [b, a, {}, {}]

Miss in one path
Hit in another one
Undefinitely unknown

R= [d, b, c, a]

[{b, d}, a, c, {}]

L ⊔ R=
[b, {}, a, {}]

min instead of max bounds

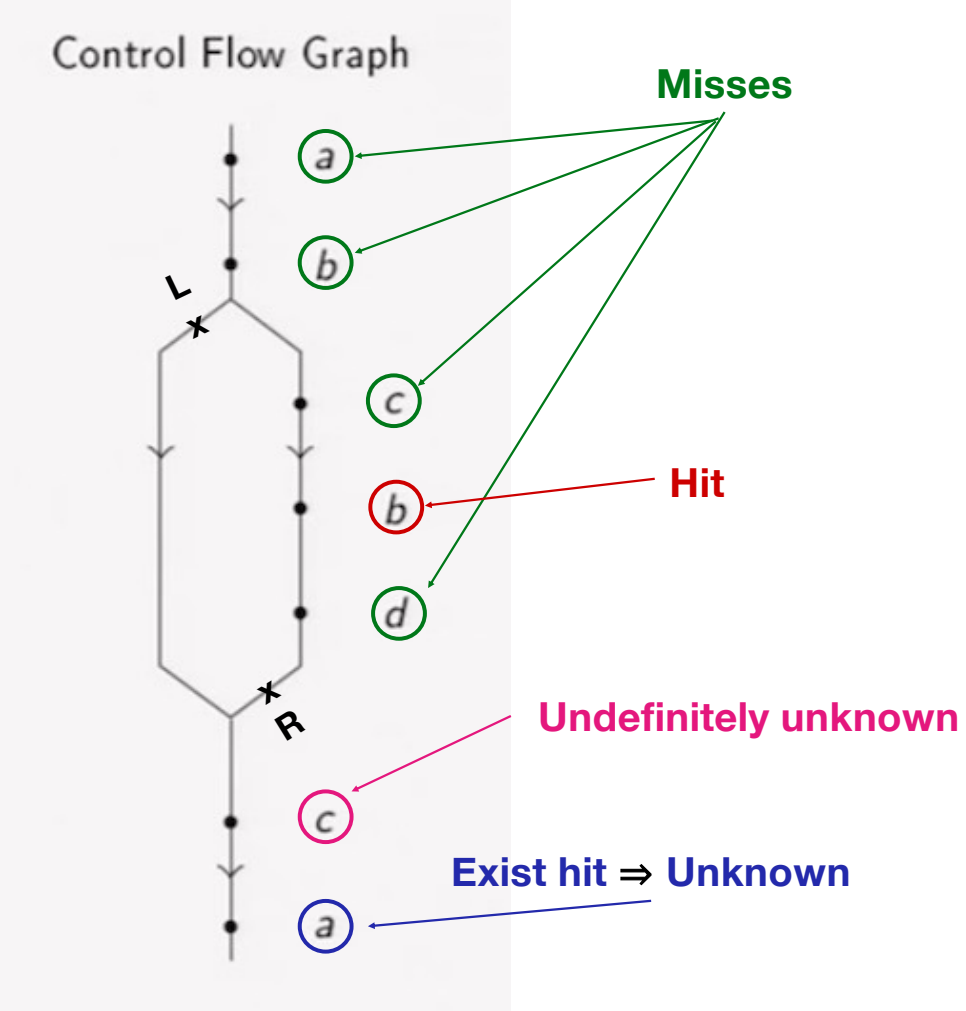
[{}, b, {}, a]

Hit in one path

[b, a, {}, {}]

We have both a and c in our states so we conclude there is a hit

Cache analysis



Beyond cache static analysis (reloaded)

The discussed analyses are able to classify memory accesses as:

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

We can try to fill the gap to have a complete analysis. Something both sound and complete.

Due to the incomplete nature (over-approximation) of the analysis, the exact status of some blocks remains **unknown**

How can we refine their classification and to gain precision?

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

Both sound and complete

What is model checking?

Model Checking is an an automated technique used to ensure that some system satisfies a given property: it checks correctness of a finite model of the program under analysis

Just exploring all the possible executions. Instead of running the program, we explore all possible runs

It checks the correctness of a finite model of the program by exhaustively exploring possible executions (paths or states) to see if a property is satisfied.

Instead of running a program, model checking explores all possible runs to see if a property always holds

Model checking is a very effective technique to expose potential design errors

① Checks correctness: checks we follow a property

Model checking

In particular, here the analysis uses the **NuXmv** symbolic model checker, able to verify synchronous finite-state and infinite-state systems

NuXmv is based on based on the **NuSMV**, that in turn builds on **SMV** (Symbolic Model Verifier), introduced in 1993

What Does a Symbolic Model Checker Do?

It verifies that a property holds **on *all* possible executions** of a system

How?
By constructing a state graph of the model, where states are encoded into **symbolic** representations

state = configuration of program variables

In **SMV**, the number of states = 2^n, with n = number of state variables
- only reachable states from the initial state need to be considered

My constructs a state graph of the model where states are represented in a symbolic way. So it is a symbolic execution, not an actual execution. Problem is states can explode. You can have a very huge transition system.

A symbolic model checker verifies that a property holds on all possible executions, but how? By constructing a state graph of the model where states are encoded in symbolic ways, so it is not just execution of a program you would have run a run, but works on symbolic representation of the run.
The state is just a configuration of the program vars.
Problem: # of states can explode.
The model checker should explore everything!

Idea: # of reachable states can grow and have an impact on computational side. Symbolic representation mitigates this explosion if you find a way to factorize states with Binary decision diagrams.

So idea is, you have this FSM that represents in a symbolic way the possible evolutions of your program. The FSM models the sys you are analyzing, and then you need a property.
A property is a temporal logic formula. It has temporal references that you can make.

The model checker will ensure that all the possible evolutions of the FSM comply with the specified property.

State explosion issue

The number of reachable states grows exponentially with the number of variables

- This makes exhaustive checking computationally expensive

Symbolic representation mitigates this explosion

If you find a way to represent states efficiently you work better. Ex. you can use:

SMV uses Binary Decision Diagrams (BDDs) to:

- concisely represent the state graph implicitly (avoiding listing all states explicitly)
- operate symbolically on sets of states and transitions

You have a FSM modelling the system you are analysing to check for a property.

Model checking, in practice

Model checking,

- given a system modeled as a finite-state machine and
- a formal property expressed as a temporal logic formula over its variables,

systematically verifies whether the finite state machine evolves safely according to the formula

property is a temporal logic proposition

The model checker ensures that all possible evolutions of the finite-state machine comply with the specified property, guaranteeing correct behavior across all states

Model checking: example

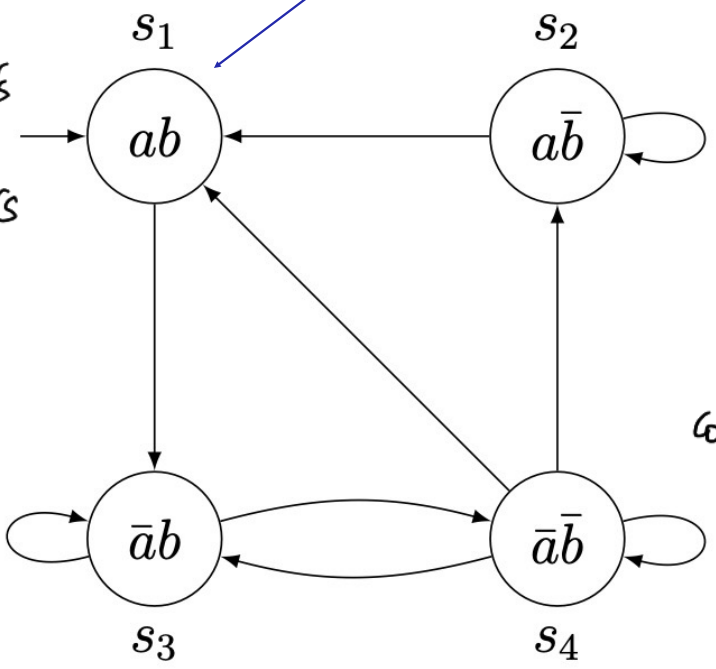
Module including
- state variable declarations
- assignments defining the valid initial states
- assignments defining the transition relation

In state s1, where both a and b are true, the only reachable state is s3. where a and b will take the values false (first case of the next(a)) and true (first case of the next(b))

```
init(a) := true
init(b) := true
next(a) := case
             b : false
             a : true
             true : {false, true}
next(b) := case
             a & b : true
             true : {false, true}
```

Each assignment to variables corresponds to a state

(a) Example of boolean program



(b) Example of Finite State Machine

Here assignments define the transition relations of these machines. They say how a and b evolve in time.

next(a) depends on the case. It can be false or true or be non deterministic.

Temporal logic

Temporal logics are extensions of propositional logic with temporal operators

of FSM modeling time considerations

They can thus express relations between states of a program model and distant by several time steps

Idea of "what comes next"

Temporal formulas are useful to express complex properties of a program, involving several states at different time

They capture a lot of interesting properties

Back to our example

Property: there is **no** way to go from a state where both a and b are true (s1) to a state where they are both false (s4) in a single step

Formally:

states from which s4 is **immediately** reachable

$\Phi := (a \wedge b) \Rightarrow \neg X (\neg a \wedge \neg b)$ ✓

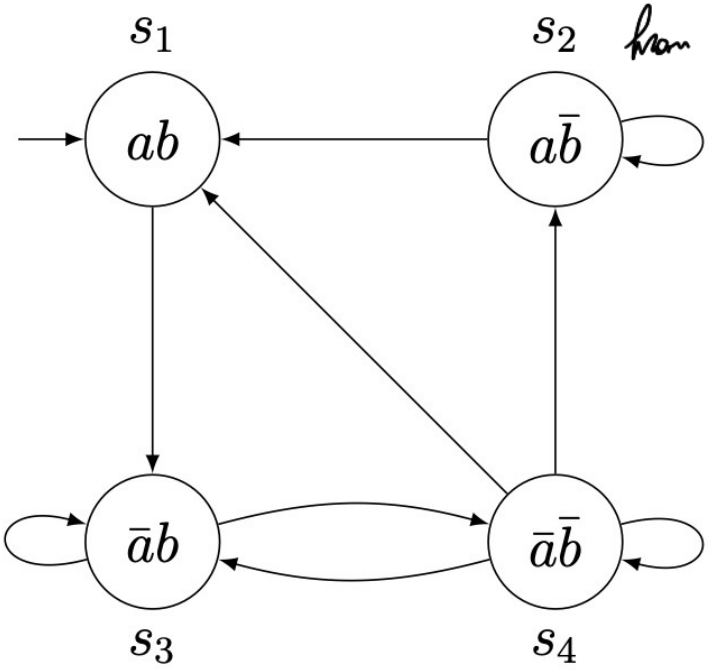
neXt: temporal operator that express property of the next state

$\Phi' := (a \wedge b) \Rightarrow X (\neg a \wedge \neg b)$ ✗

S_4 should not be immediately reachable.

Φ is expressed as follow: not X of not a and b.

We can also say: something will be satisfied, or something will be true from one point to the followings.



(b) Example of Finite State Machine

Model Checking here

In order to classify the remaining unclassified accesses, the model checker is feeded with the model of the program's memory access behavior + the cache»:

- a finite-state machine modeling the cache behavior of the program, composed of
  - a model of the program, yielding the possible sequences of memory accesses
  - a model of the cache
- a logic formula encoding the possibility of a hit or a miss, to check the properties:

  "*Is this access always a hit? Is this access always a miss?*"

the property of interest

To classify remaining accesses, you provide model checker a FSM that models the cache behaviour of the program.

To apply this FM, we need to classify remaining unclassified accesses. You have to provide the model checker with a FSM that models the cache behavior, so we need both a model for the program and the model of the cache to simulate the evolution of each access. Then you need a logic formula encoding property of interest, in our case "is this access always a hit/miss?"

After the AI based analysis phase

In our setting the model checking is quite efficient

We could use it even w/our static analysis, but performance is worse.

- It is used **only for accesses left unclassified**, after the AI based analysis phase

↳ Part of program is already solved

- It only looks at a **small slice of the program**: the focus is only on the blocks and instructions involved

[more than only loads in consideration]

CFG is simplified even more, because we only focus on each block at a time, in particular for the ones we need to classify.

abstract interpretation

Exploting AI results

Always-Hit problem answers a question like:
does any path in the given graph lead to hit at the given access?

Exist-Miss problem: Counter example of the always hit
is there any path in the given graph that lead to a miss at the given access?

The two are strictly related: if there exists a path that leads to a miss at the given access, then it is not true that any path lead to a hit

Therefore, the Always-Hit problem can be reformulated in terms of Exist-Miss problem

- Similarly, for Always-Miss

Model checking starting from AI results

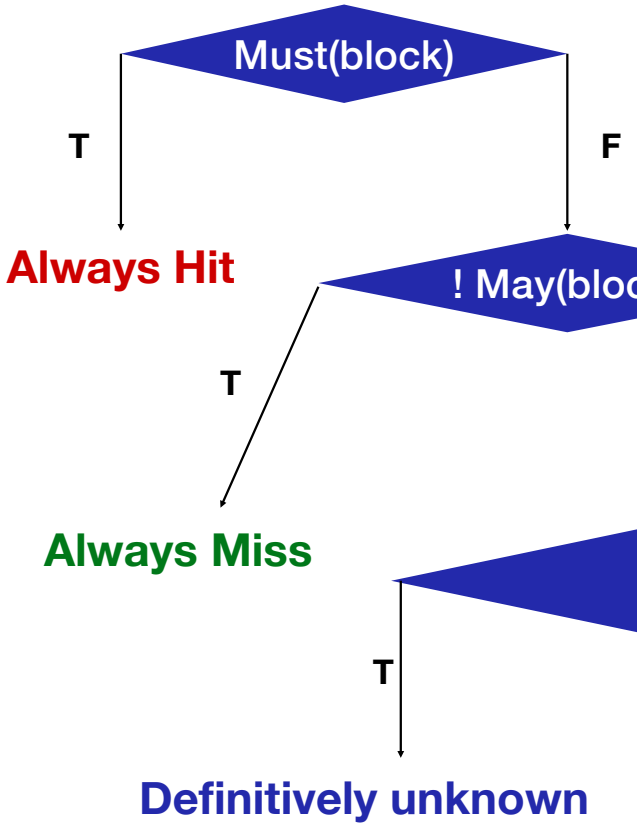
- Therefore a block that is not fully classified as "definitely unknown" can still benefit from the Exists Hit and Exists Miss analysis during the model-checking phase

- If the AI phase shows that there exists a path on which the block is a hit (respectively a miss), then the model checker does not have to check the "always miss" (respectively "always hit") property

- Exist-miss and exist-hit properties amount to checking that certain states are reachable in the finite-state machine

Even though a block is not definitely unknown as classification, you can explain Exists Hit and Exists Miss analysis during model checking phase.

AI phase



If a block is always hit, good. If always miss good.
If both DU otherwise model checker.

Otherwise you move w/ MC.

Focused cache model: abstraction

- The model checker is run one time for every block that is left unclassified
- Abstraction is based on the following LRU simple property:

> a memory block is cached if and only if its age is less than N,
> or, in other words, if there are less than N younger blocks

- Abstraction is relatively to a single variable **a** and tracks the set of blocks younger than a
- Very efficient abstraction: we only keep information useful for classifying **a** as a hit or a miss

Focused cache model: abstraction

- The analysis tracks the set of blocks younger than a

younger

older

Age	State
1	b
2	c
3	a
4	d

{b,c}

b and **c** younger than **a**
Age(**b**) > Age(**a**) and Age(**c**) > Age(**a**)

Age	State
1	a
2	b
3	c
4	d

{ }

a: most recently used element

Age	State
1	b
2	c
3	d
4	e

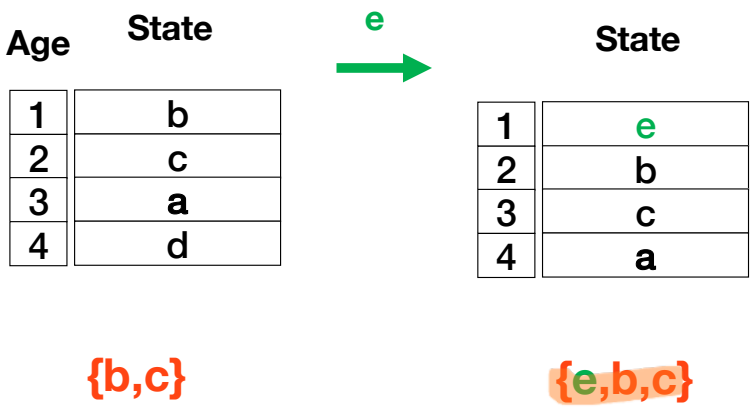
ε

Age(**a**)>N
a is not in the cache

Special symbol, a is not in the cache

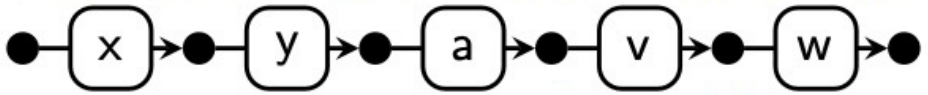
New cache model: update function

- A new access impacts on the set of blocks younger than **a**. For instance:



Small example

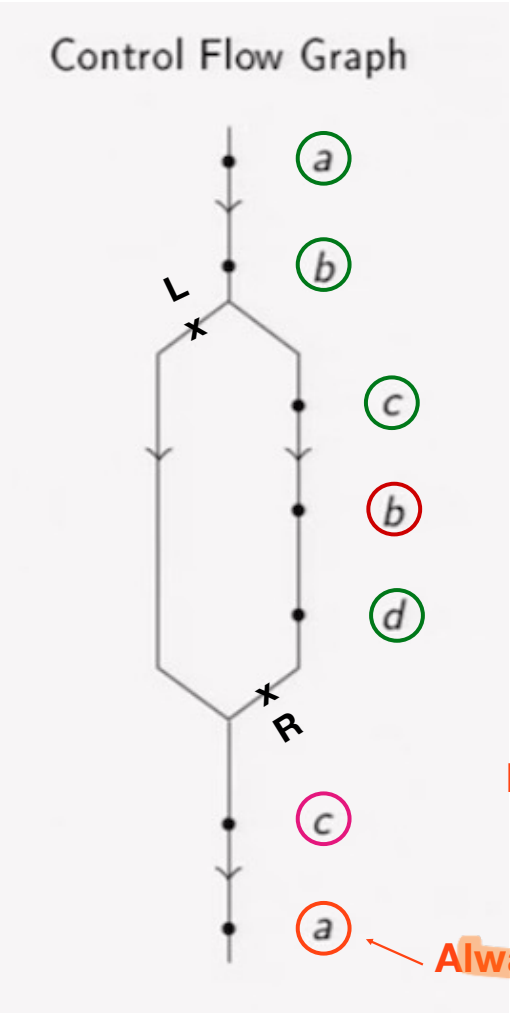
Concrete cache model: $[-,-]$ $[x,-]$ $[y,x]$ $[a,y]$ $[v,a]$ $[w,v]$



Focused cache model: ϵ ϵ ϵ $\{\}$ $\{v\}$ ϵ

Empty cache. a is not in the cache

Back to our example



ε

{ { } }

L = {{b}}

{{b,c}}

{{b,c}}

R = {{b,c,d}}

L ⊔ R = {{b},{b,c,d}}

{{b,c},{b,c,d}}

Always hit (age(a) in [2,3])

Simplification of the CFG

For the program model, we simplify the CFG without affecting the correctness nor the precision of the analysis:

We can filter out irrelevant instructions: if a is not in the cache at point n, that instr. can't affect a's eviction.

Many irrelevant instructions are filtered out:

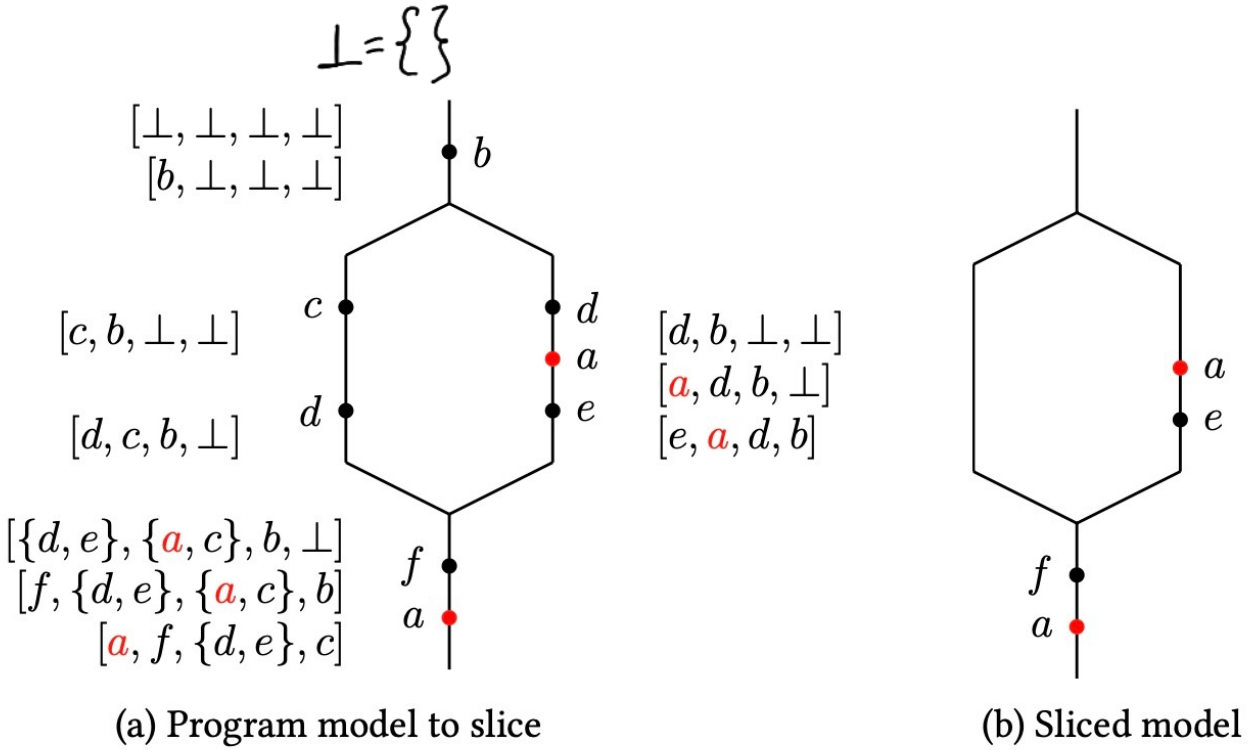
- If from may analysis, we know that in a given program instruction **a** is never in the cache, then this instruction cannot affect **a**'s eviction: thus we simplify the program model by not including this instruction

- When we encode the set of blocks younger than **a** as a bit vector, we do not include blocks that the may analysis proved not to be in the cache at that location: these bits would anyway always be 0

If you use a bit vector to represent their presence, since they would

Simplification of the CFG

If from may analysis, we know that in a given program instruction **a** is never in the cache, then this instruction cannot affect **a**'s eviction: thus we simplify the program model by not including this instruction



(a) Program model to slice

(b) Sliced model

In summary

The presented model-checking phase is **sound**, i.e. its classifications are correct, but it is also **complete** relative the our CFG model, i.e.,

- there remain **no unclassifed accesses**

The novelty of this approach: **block focusing** technique, able to abstract cache state relatively to a given memory block

- The cache state representation is smaller because it does not keep information about the blocks accessed before the block we are interested in
- We only need the set of blocks that are younger than the block we focus on
- However, this abstraction is still able to provide the exact age of the block we focus on, and enables us to exactly classify it among the three categories Always-Hit, Always-Miss and Definitely Unknown

You are able to compute exact ages of blocks and so you can complete classification.

Representation is correct, but also complete. There remain no unclass. accesses.

In summary, model checking class is sound and complete. After cache analysis and MC, there remain no unclassified accesses.
Novelty of this approach is on this block focusing technique that allows us to simplify our transition system. It allows us to focus only on one block at a time. This abstraction has been proven to be able to provide exact age of the blocks we focus on, so we can complete characterization.

Bibliography

- 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.

Model checking is a technique you can use w/ Monotone Framework

- Model checking is also used as a technique when you can't find an easy monotone FW to work in. Out of order execution is another approach to speed up execution. In these cases AI techs won't work but you can use model checking in that case.