SemPat: From Hyperproperties to Attack Patterns for Scalable Analysis of Microarchitectural Security

Adwait Godbole (adwait@berkeley.edu), Yatin A. Manerkar, Sanjit A. Seshia

ACM CCS 2024, Salt Lake City, US

Berkeley UNIVERSITY OF CALIFORNIA

UNIVERSITY OF MICHIGAN

Example: Spectre V1 (BCB) Vulnerability

out-of-bound index i

```
void victimA (uint32_t i) {
  if (i < ARR_SIZE)
    temp_ = arr2[arr1[i] << CL_INDEX];
}
```

Example: Spectre V1 (BCB) Vulnerability

out-of-bound index i

```
void victimA (uint32_t i) {
  if (i < ARR_SIZE)
    temp_ = arr2[arr1[i] << CL_INDEX];
}
```

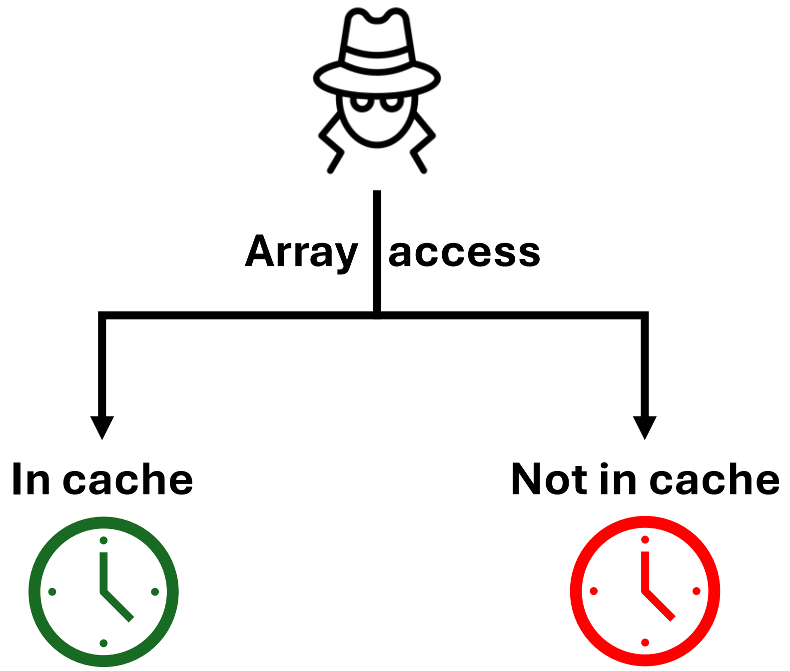
Secret-dependent load

Example: Spectre V1 (BCB) Vulnerability

out-of-bound index i

```
void victimA (uint32_t i) {
  if (i < ARR_SIZE)
    temp_ = arr2[arr1[i] << CL_INDEX];
}
```

Secret-dependent load



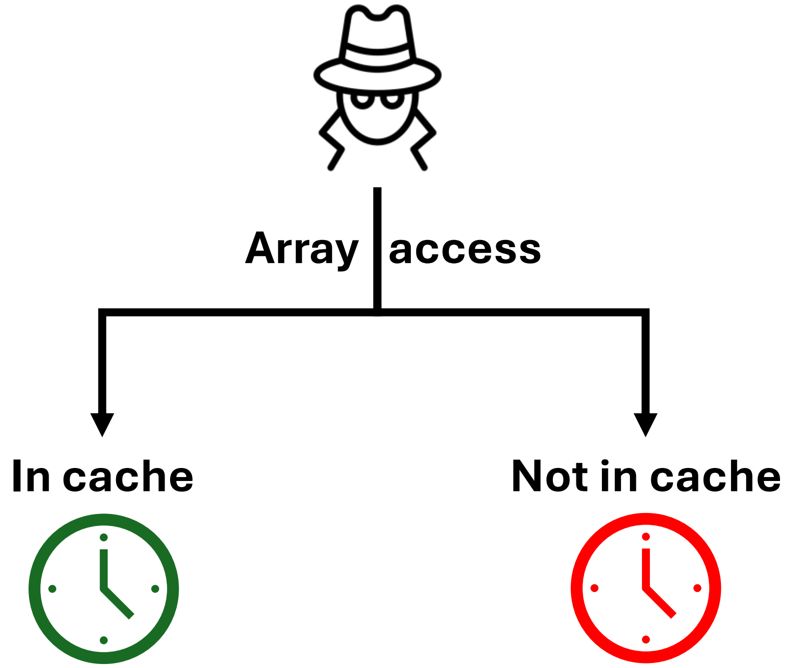
Cache-based timing side-channel

Example: Spectre V1 (BCB) Vulnerability

out-of-bound index i

```
void victimA (uint32_t i) {
  if (i < ARR_SIZE)
    temp_ = arr2[arr1[i] << CL_INDEX];
}
```

Secret-dependent load



Cache-based timing side-channel

SW-verification for microarchitectural security:
Is SW program susceptible to such attacks?

Two approach classes from previous work

Pattern-based

```
void victimA (uint32_t i) {

  if (i < ARR_SIZE) {

    temp1_ = arr1[i];

    temp_ = arr2[temp1_ << CL_INDEX];

  }
}
```

speculation

dependent load address

e.g., Ponce de Leon [S&P 2023],
Mosier et. al. [ISCA 2022]

Noninterference-based

Precondition: Φ_{pre}

```
void victimA (uint32_t i) {
  if (i < ARR_SIZE)
    temp_ = arr2[arr1[i] << CL_INDEX];
}
```

Postcondition: Φ_{post}

e.g., Cheang et. al. [CSF 2019],
Guarneri et. al. [S&P 2020]

This work: convert from NI to patterns

Pattern-based

```
void victimA (uint32_t i) {

  if (i < ARR_SIZE) {

    temp1_ = arr1[i];

    temp_ = arr2[temp1_ << CL_INDEX];

  }
}
```

speculation

dependent load address

This work

Noninterference (NI)-based

Precondition: Φ_{pre}

```
 victimA (uint32_t i) {
  if (i < ARR_SIZE)
    temp_ = arr2[arr1[i] << CL_INDEX];
}
```

Postcondition: Φ_{post}

e.g., Ponce de Leon [S&P 2023],
Mosier et. al. [ISCA 2022]

e.g., Cheang et. al. [CSF 2019],
Guarneri et. al. [S&P 2020]

Pattern-based Analysis

```
void victimA (uint32_t i) {
  if (i < ARR_SIZE)
    temp_ = arr2[arr1[i] << CL_INDEX];
}
```

```
.victimA:
  ...
  bltu a5,a4,66004;  A1:Branch
                        ↓ spec.
  ...
  lw a5,a5,0;         A2:Load
                        ↓ addr. dep.
  ...
  lw a4,a5,0;         A3:Load
  ...
66004:  ←——— architectural
```

Execution embeds the pattern

A1:Branch —speculative→ **A2:Load** —address dependency→ **A3:Load**

Gadget variant

Variant

```
void victimA (uint32_t i) {
  if (i < ARR_SIZE)
    temp_ = arr2[arr1[i] << CL_INDEX];
}
```

```
void victimB (uint32_t i) {
  uint32_t temp1_ = arr1[i];
  if (i < ARR_SIZE)
    temp_ = arr2[temp1_ << CL_INDEX];
}
```

Embeds

Variant execution does not embed!

A1:Branch —speculative→ **A2:Load** —address dependency→ **A3:Load**

Gadget variant needs a new pattern

Patterns do not generalize well

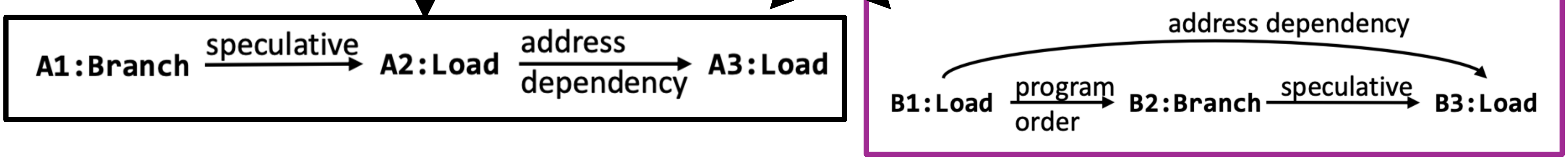
Variant

```
void victimA (uint32_t i) {
  if (i < ARR_SIZE)
    temp_ = arr2[arr1[i] << CL_INDEX];
}
```

```
void victimB (uint32_t i) {
  uint32_t temp1_ = arr1[i];
  if (i < ARR_SIZE)
    temp_ = arr2[temp1_ << CL_INDEX];
}
```

Embeds

Embeds

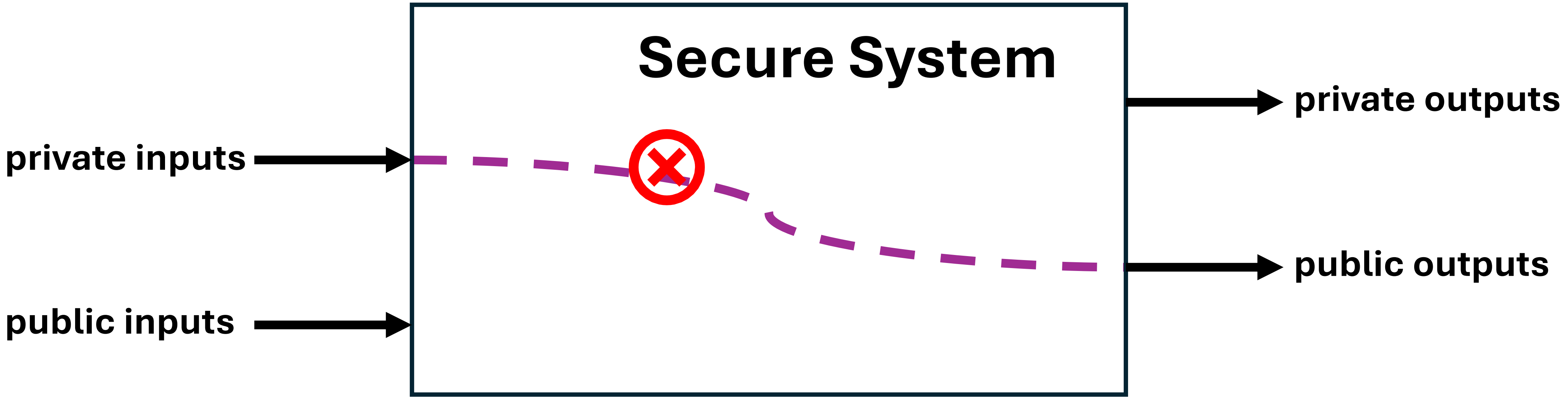


New pattern

Hyperproperty-based Analysis

Hyperproperties formally characterize semantic security

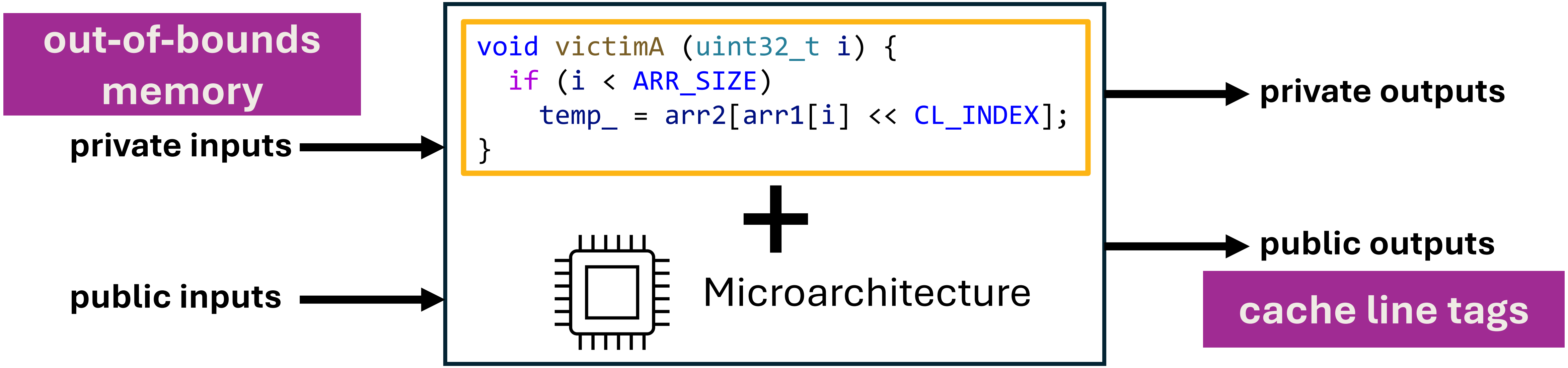
Non-interference (NI)/information-flow-control: *secret inputs do not affect public (observable) outputs*



Hyperproperty-based Analysis

Hyperproperties formally characterize semantic security

Non-interference (NI)/information-flow-control: *secret inputs do not affect public (observable) outputs*

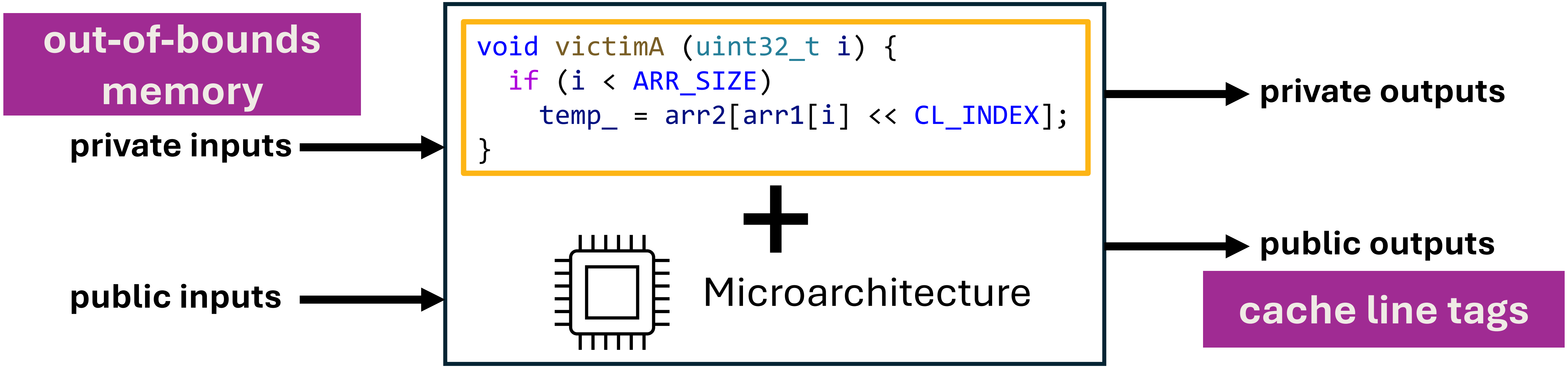


Same non-interference property applies to both variants

```
void victimA (uint32_t i) {
  if (i < ARR_SIZE)
    temp_ = arr2[arr1[i] << CL_INDEX];
}
```

Variant

```
void victimB (uint32_t i) {
  uint32_t temp1_ = arr1[i];
  if (i < ARR_SIZE)
    temp_ = arr2[temp1_ << CL_INDEX];
}
```



Motivation: Orthogonal Advantages

Approach	Pattern-based	Noninterference-based
Pros	Simpler ***verification*** queries, scalable	Uniform ***specification***, Robust
Cons	Sensitive to gadget structure	Scalability

Can we combine specification benefits of hyper-properties and scalable verification of patterns?

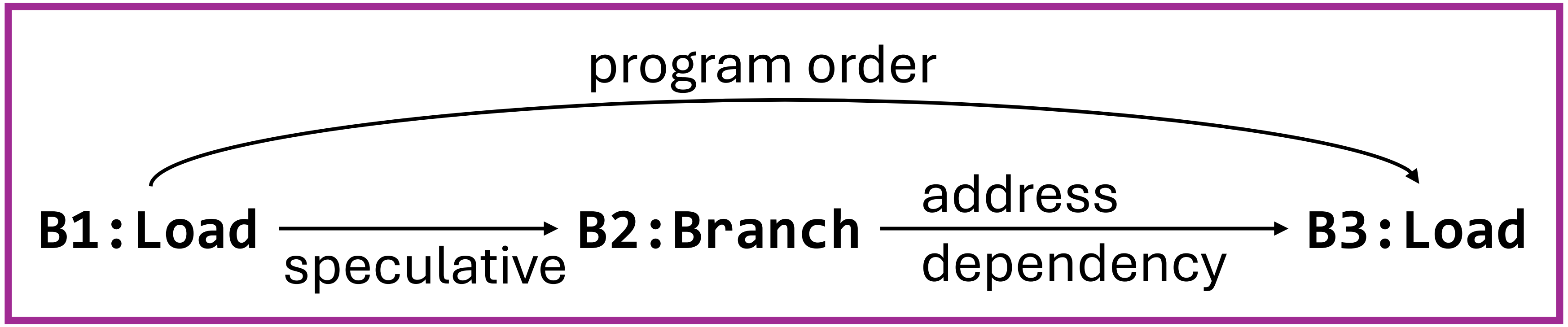
Contributions

- ***k*-completeness condition**: set of patterns covering all non-interference violations up to a size bound *k*
- **Pattern generation algorithm**: grammar-based search to produce a ***k*-complete set** of patterns
- **Evaluation:** (a) scalable pattern generation: **new patterns**, (b) verification: upwards of **100x** improvement over hyperproperties (for models considered)

Outline

- **Problem Formulation**
  - **Pattern Definition**
  - Pattern Generation Problem
- Pattern Generation Approach
- Theoretical Guarantee
- Implementation and Evaluation

A pattern is a pair (*w*, ϕ)



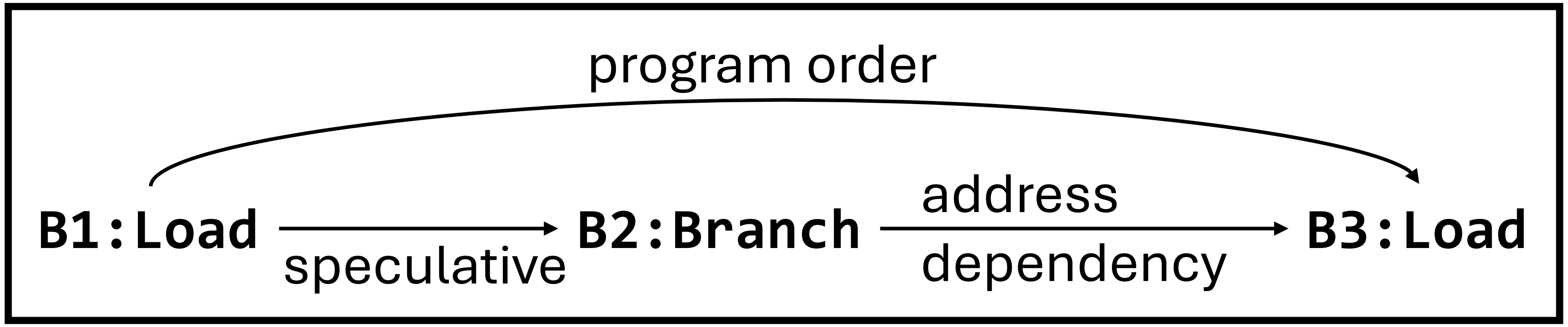
Pattern template (opcode sequence): *w*

(1: Load) -- (2: Branch) -- (3: Load)

A boolean formula constraint: ϕ

addrdep ((1: Load), (3: Load)) && speculative ((2: Branch))

A pattern is a pair (*w*, ϕ)



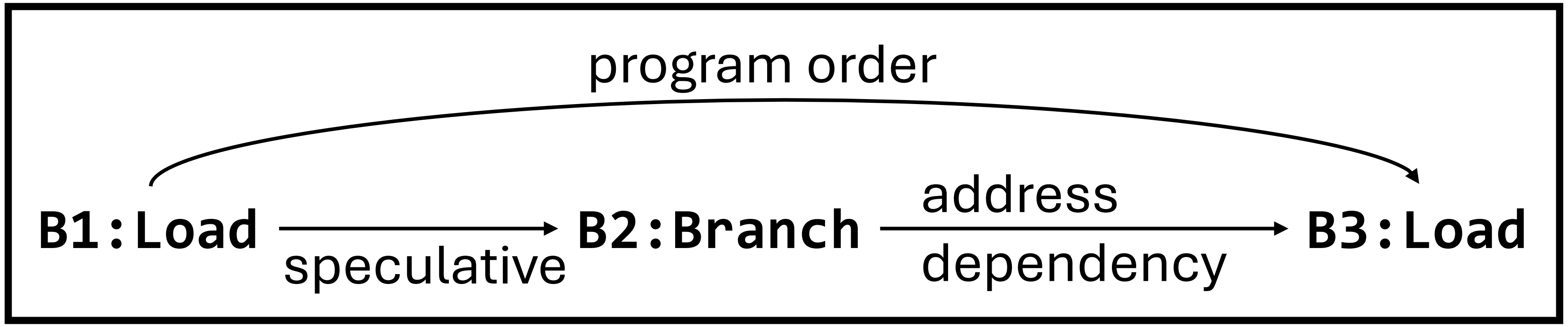
Pattern template (opcode sequence): *w*

`(1: Load) -- (2: Branch) -- (3: Load)`

A boolean formula constraint: ϕ

addrdep ((1: Load), (3: Load)) && speculative ((2: Branch))

A pattern is a pair (*w*, ϕ)



Pattern template (opcode sequence): *w*

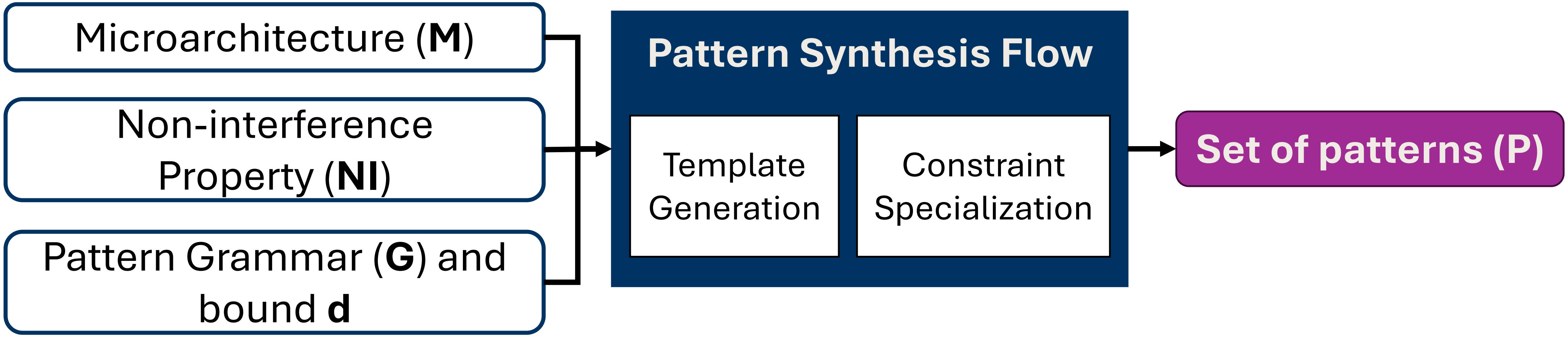
(1: Load) -- (2: Branch) -- (3: Load)

constraint is a conjunction of predicates: p1 && p2 && p3 ...

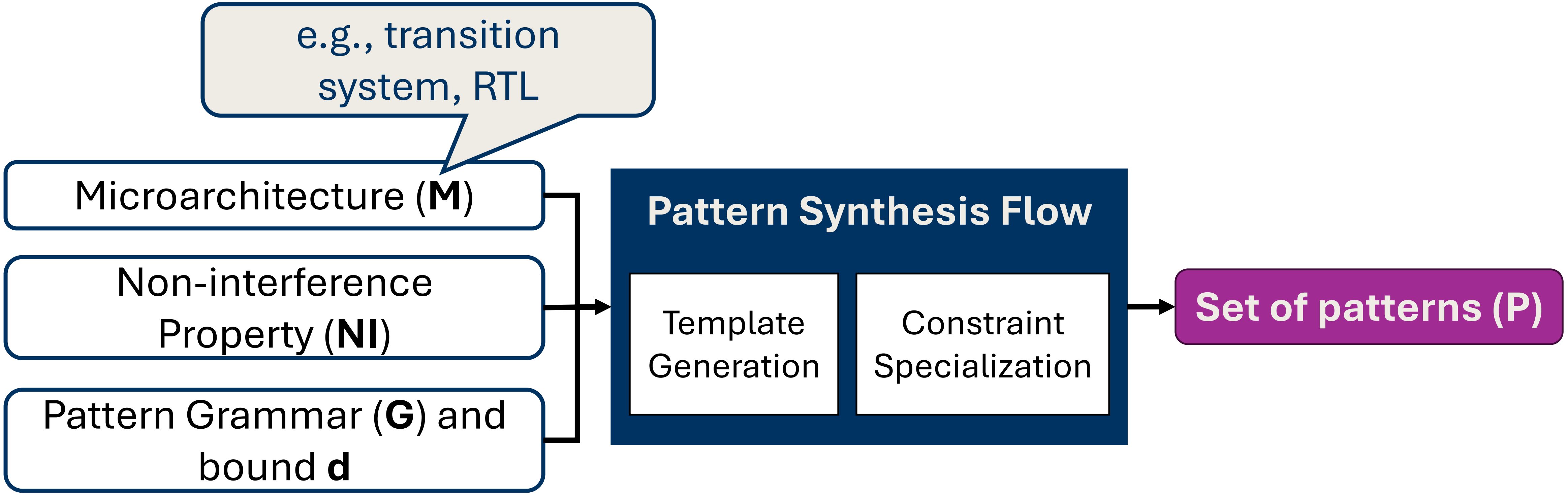
A boolean formula constraint: ϕ

addrdep ((1: Load), (3: Load)) && speculative ((2: Branch))

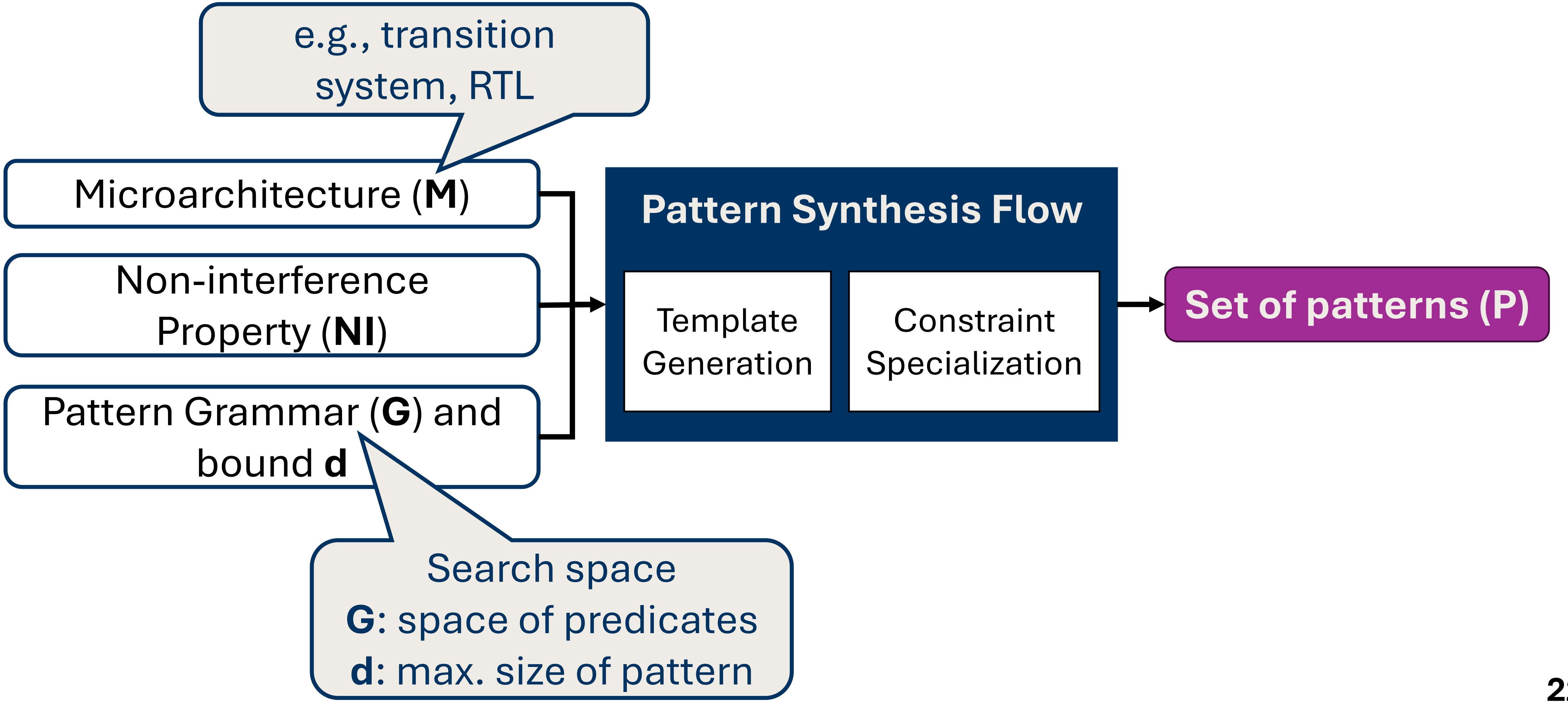
Pattern Generation Problem



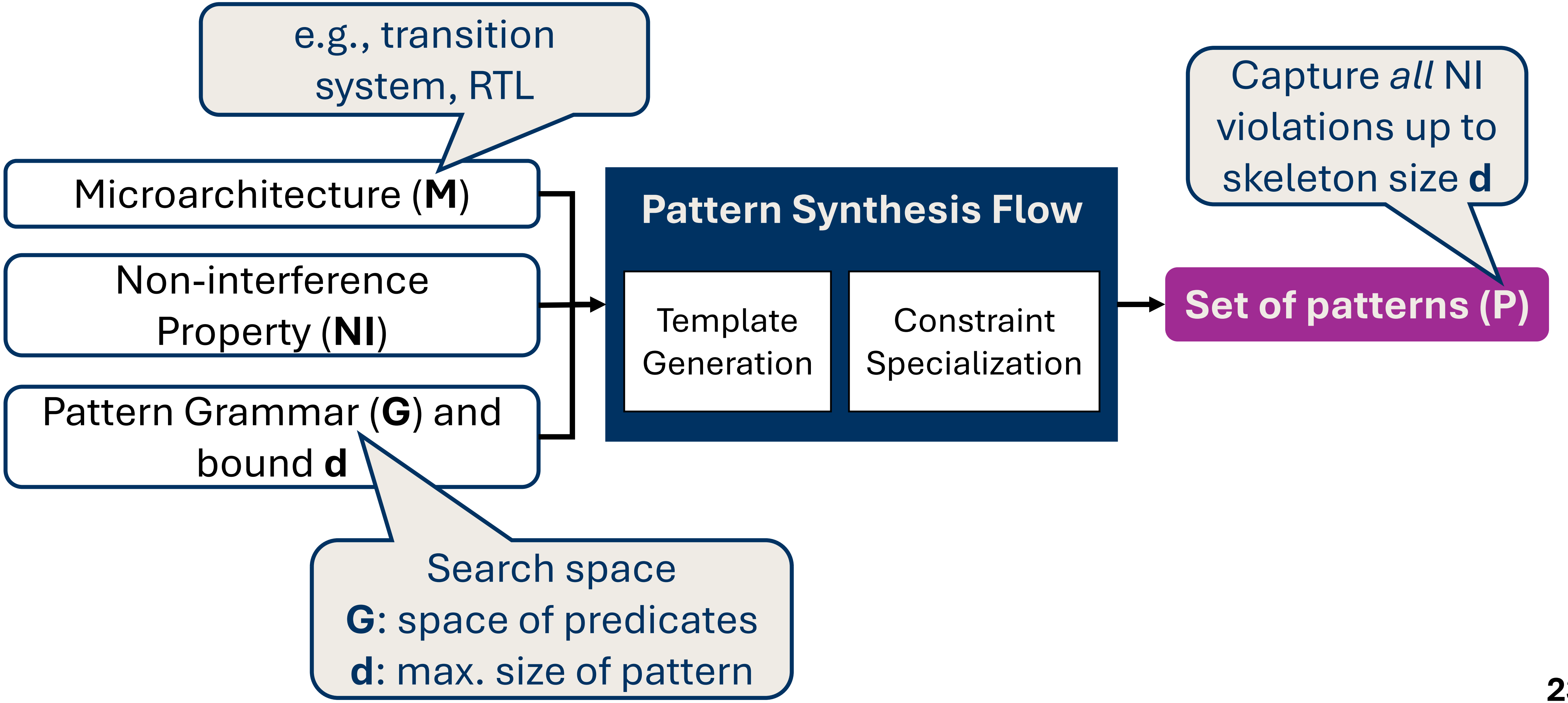
Pattern Generation Problem



Pattern Generation Problem



Pattern Generation Problem

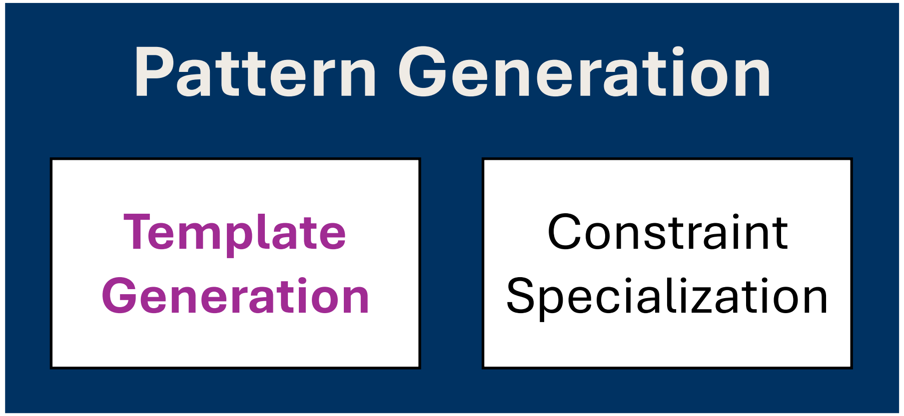


Outline

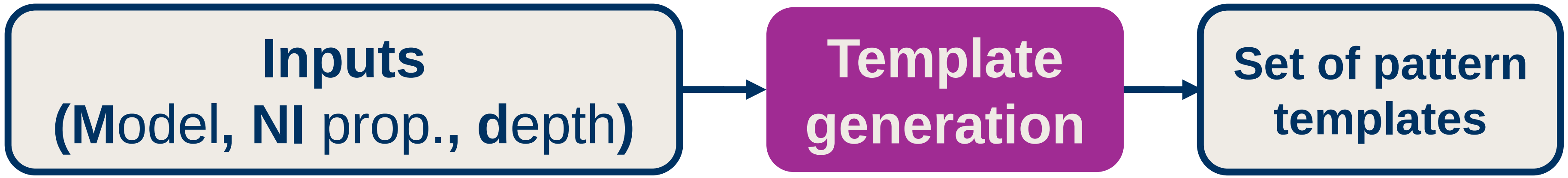
- Problem Formulation
- **Pattern Generation Approach**
- Theoretical Guarantee
- Implementation and Evaluation

Outline

- Problem Formulation
- **Pattern Generation Approach**
  - **Template Generation**
  - Constraint-based Specialization
- Theoretical Guarantee
- Implementation and Evaluation

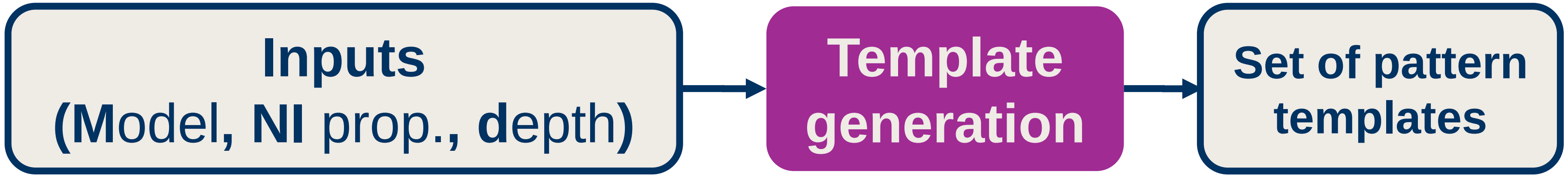


1. Template Generation



Collect all depth **d** templates (opcode seq.) which falsify the **NI** property

1. Template Generation



Collect all depth **d** templates (opcode seq.) which falsify the **NI** property

```
add-add-add          : SAFE
add-add-sub          : SAFE
add-add-load         : SAFE
           ...
branch-load-load     : UNSAFE
```

1. Template Generation

Inputs (Model, NI prop., depth) → **Template generation** → **Set of pattern templates**

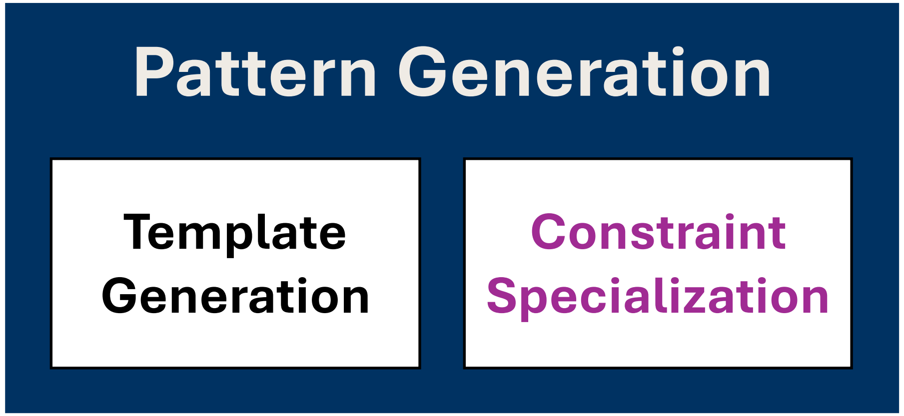
Collect all depth **d** templates (opcode seq.) which falsify the **NI** property

```
add-add-add        : SAFE
add-add-sub        : SAFE
add-add-load       : SAFE
              ...
branch-load-load   : UNSAFE
```

Too overapproximate: add constraints to reduce false positives

Outline

- Problem Formulation
- **Pattern Generation Approach**
  - Template Generation
  - **Constraint-based Specialization**
- Theoretical Guarantee
- Implementation and Evaluation



2. Constraint Specialization

Add constraints to make the template precise (reduce false positives)

2. Constraint Specialization

Add constraints to make the template precise (reduce false positives)

Constraints are sourced from a predicate grammar

Predicate Atom	Meaning
datadep(inst1, inst2)	Data dependency between inst1 and inst2
addrdep(inst1, inst2)	Address dependency
…	…
speculative(inst)	Instruction inst executes speculatively
highoperand(inst)	Instruction operand is secret dependent
…	…

2. Constraint Specialization

Add constraints to make the template precise (reduce false positives)

```
1.br-2.load-3.load :: true
```

2. Constraint Specialization

Add constraints to make the template precise (reduce false positives)

```
1.br-2.load-3.load :: true

1.br-2.load-3.load :: addrdep(2.load, 3.load)
```

2. Constraint Specialization

Add constraints to make the template precise (reduce false positives)

`1.br-2.load-3.load :: true`

`1.br-2.load-3.load :: addrdep(`*`2.load, 3.load`*`)`

`1.br-2.load-3.load ::` **addrdep(*2.load*,*3.load*) && *spec(1.br)***

2. Constraint Specialization

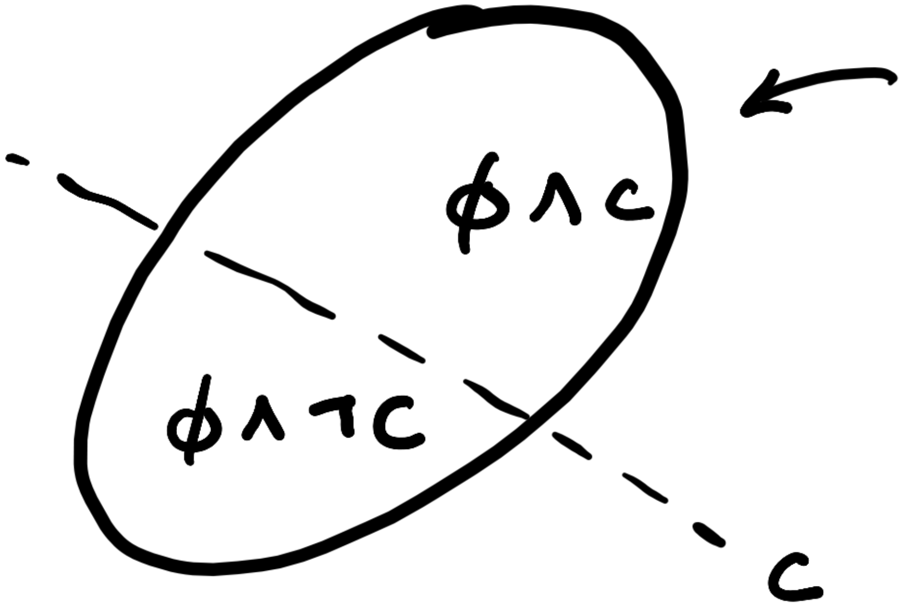
Add constraints to make the template precise (reduce false positives)

How do we add constraints without missing non-interference violations?

Counterfactual atom addition

(Adding constraints without missing non-interference violations)

Should we specialize a pattern (w, ϕ) further by adding constraint *c*?

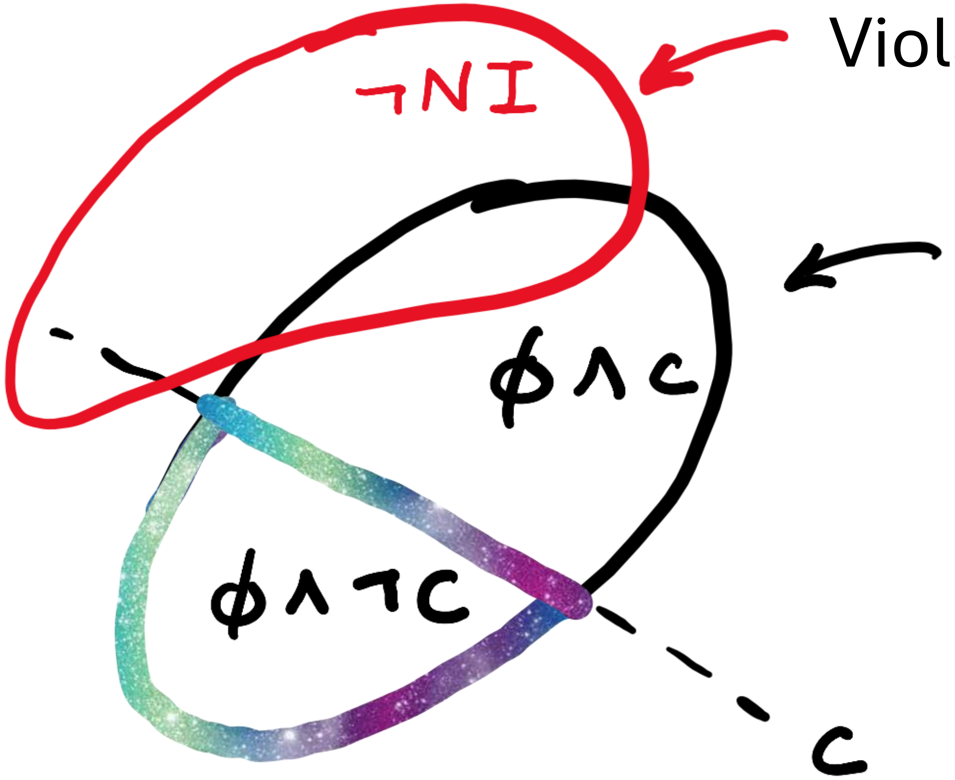


Executions matching pattern (w, ϕ)

Counterfactual atom addition

(Adding constraints without missing non-interference violations)

Should we specialize a pattern (w, ϕ) further by adding constraint *c*?



Violations of non-interference

Executions matching pattern (w, ϕ)

Safe to add *c*, when  and  do not overlap.

Can be cast as a SAT/SMT problem!

2. Constraint Specialization

Constraint-based specialization: high level procedure

For (atom in candidates):

If (adding counterfactual(atom) is SAFE)

Add atom

Outline

- Problem Formulation
- Pattern Generation Approach
- **Theoretical Guarantee**
- Implementation and Evaluation

Theoretical Guarantee

Program C has a violation of **skeleton size *k* if**
C has a dependency-closed sub-sequence of *size* <= *k* that violates NI

$$C \not\models_k \mathrm{NI}(\Sigma_{\mathrm{init}}, \mathrm{V}_{\mathrm{pub}}, \mathrm{V}_{\mathrm{obs}})$$

Theoretical Guarantee

Program C has a violation of **skeleton size *k* if**
C has a dependency-closed sub-sequence of *size* <= *k* that violates NI

$$C \not\models_k \mathrm{NI}(\Sigma_{\mathrm{init}}, \mathrm{V}_{\mathrm{pub}}, \mathrm{V}$$

Generated patterns

$$\forall C.\ C \not\models_k \mathrm{NI}(\Sigma_{\mathrm{init}}, \mathrm{V}_{\mathrm{pub}}, \mathrm{V}_{\mathrm{obs}}) \implies \exists \mathrm{p} \in \mathrm{P}.\ C \models \mathrm{p}$$

"If C has a small skeleton, some pattern in P will catch violation"

Outline

- Problem Formulation
- Pattern Generation Approach
- Theoretical Guarantee
- **Implementation and Evaluation**

Evaluation

- Implementation: prototype tool SECANT (with UCLID5 [1] backend)
  - Scala-embedded model specification DSL
  - Pattern generation and verification engines

[1] Polgreen, et. al. UCLID5: Multi-modal Formal Modeling, Verification, and Synthesis. CAV 2022.

Evaluation

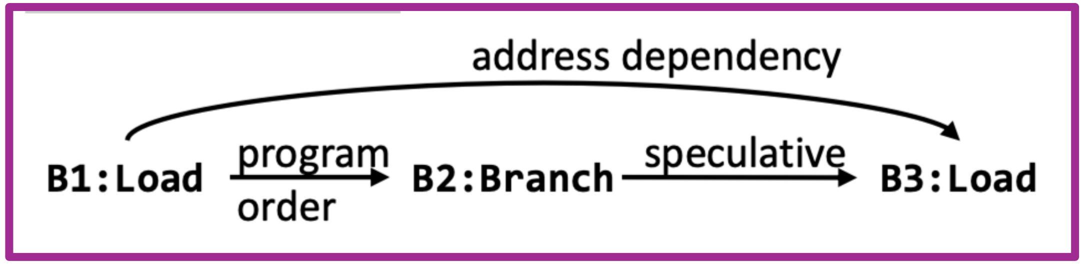
- Implementation: prototype tool SECANT (with UCLID5 [1] backend)
  - Scala-embedded model specification DSL
  - Pattern generation and verification engines

- Evaluation on 3 abstract microarchitecture models:
  - Silent Stores
  - Dynamic Instruction Reuse
  - Branch/STL Speculation

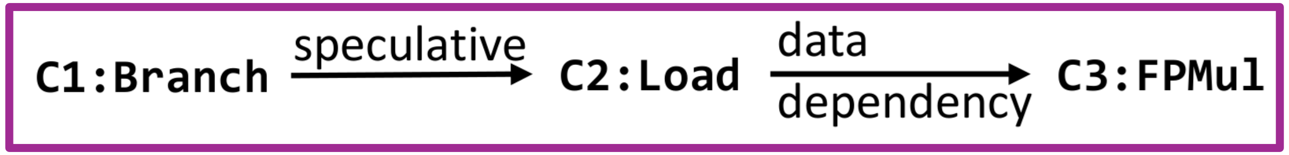
[1] Polgreen, et. al. UCLID5: Multi-modal Formal Modeling, Verification, and Synthesis. CAV 2022.

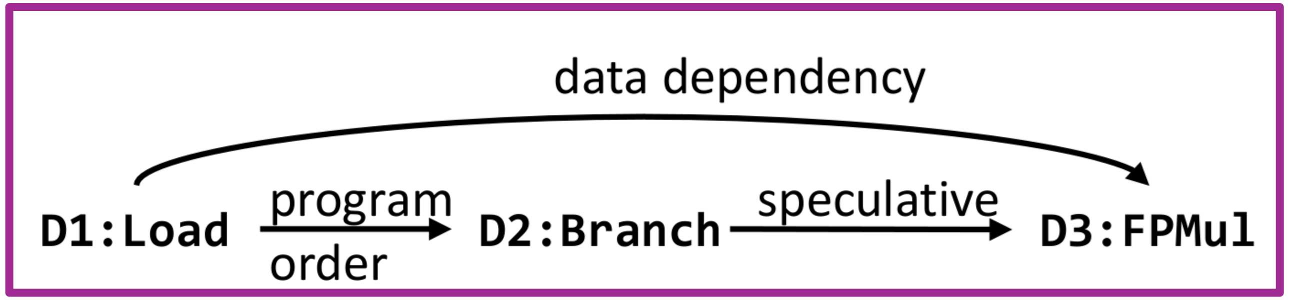
Results: New Patterns

Spectre BCB+Cache:

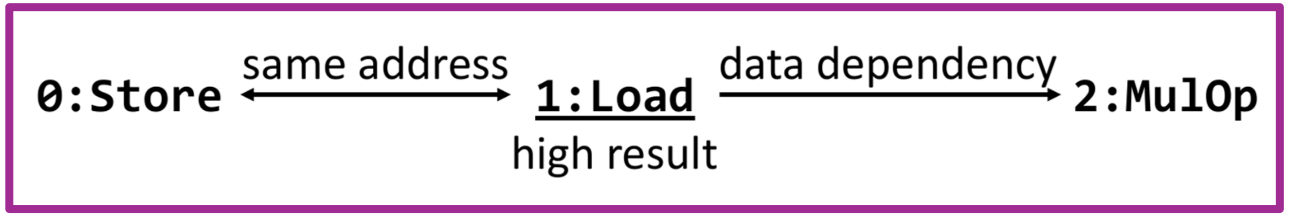


Spectre BCB+CR:





Spectre STL+CR:



Results: Improved Verification Performance

Modified Kocher's BCB/STL tests:

Replaced cache-based side channel with a computation-based side channel.

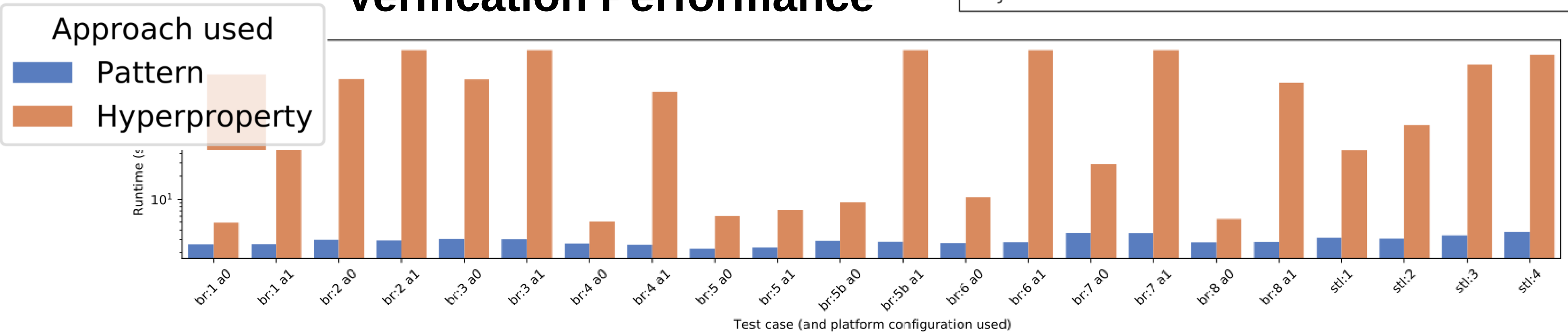
Spectre BCB

```
  void test1 (uint64_t idx) { // INSECURE
    // Bounds-check-bypass
    if (idx < publicarray_size)
-     temp &= publicarray2[publicarray[idx]*512];
+     temp &= publicarray[idx] * SCALAR;
  }
```

Spectre STL

```
  void test2 (uint32_t idx) { // INSECURE
    idx = idx & (publicarray_size - 1);
    /* Access overwritten secret */
-   temp &= publicarray2[publicarray[idx] * 512];
+   temp &= publicarray[idx] * SCALAR;
  }
```

Verification Performance



Results: Improved Verification Performance

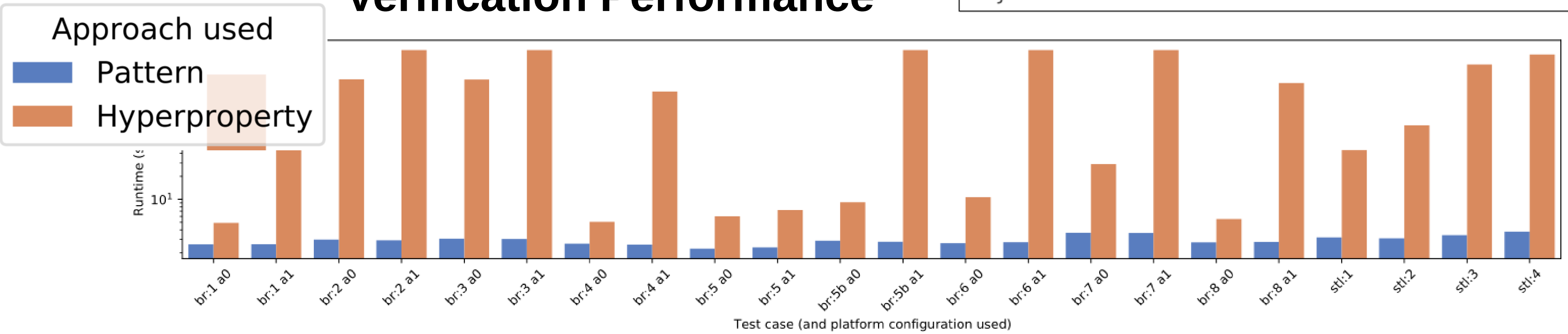
Modified Kocher's BCB/ST

Replaced cache-based sid
with a computation-based

- **~100x improvement, increases with test size**

```
  void test2 (uint32_t idx) { // INSECURE
    idx = idx & (publicarray_size - 1);
    /* Access overwritten secret */
-   temp &= publicarray2[publicarray[idx] * 512];
+   temp &= publicarray[idx] * SCALAR;
  }
```

Verification Performance

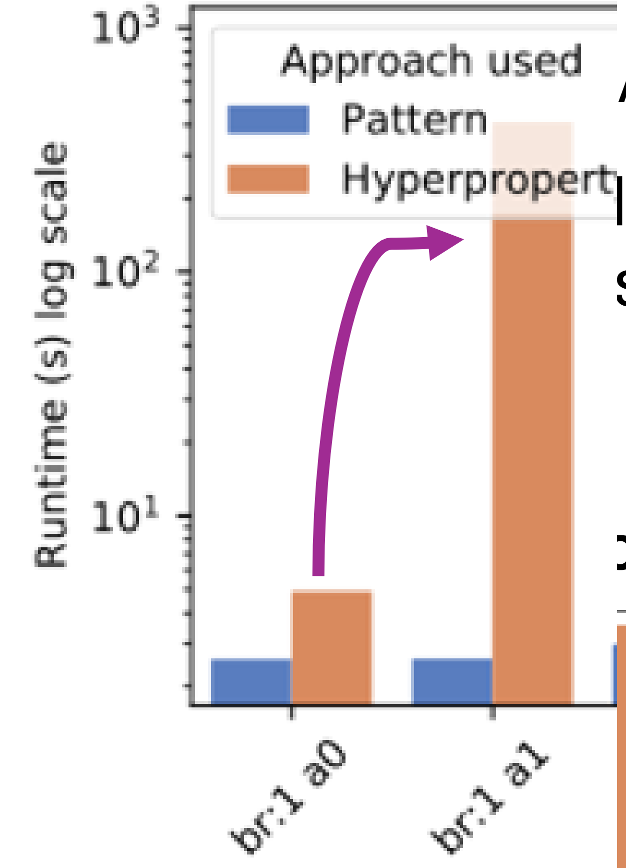


Results: Improved Verification Performance

Moc /ST

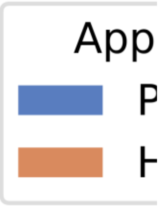
Rep l sic
witl sec

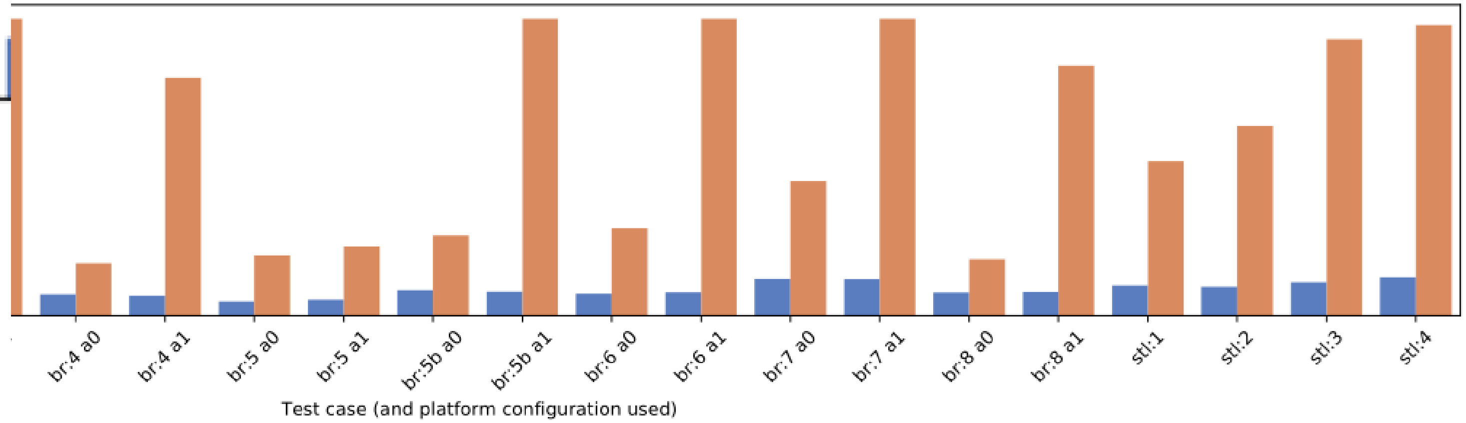
- ~100x improvement, increases with test size
- Microarchitectural complexity affects hyperproperty but not pattern runtime



on Performance

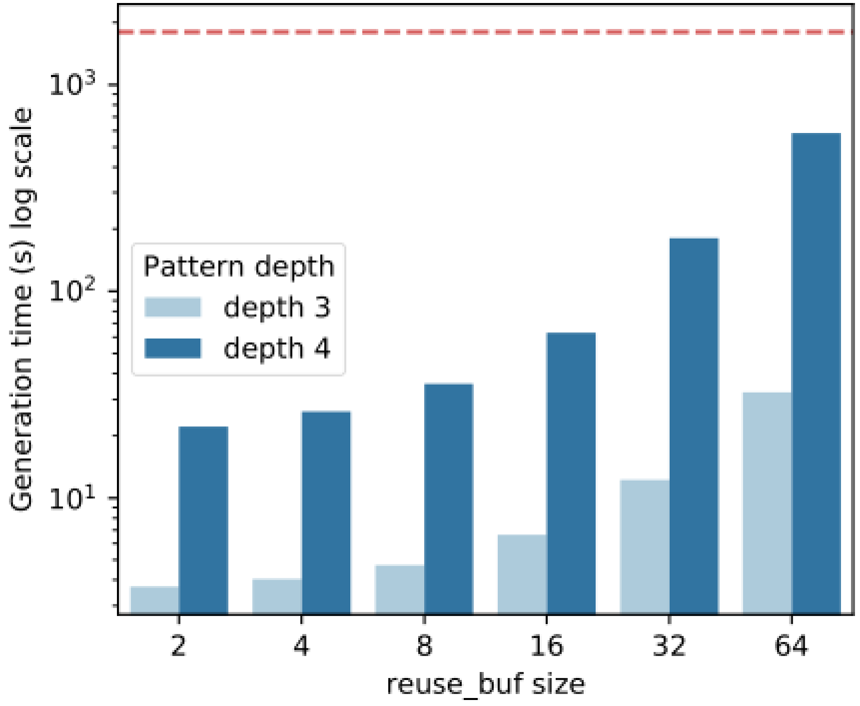
```
+     temp &= publicArray[idx] * SCALAR;
  }
```

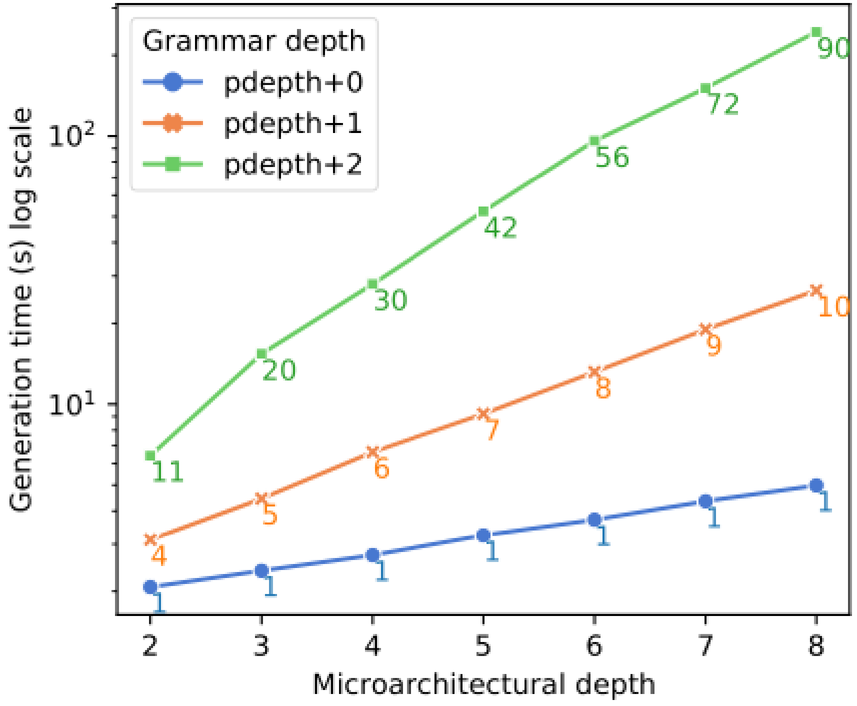




Results: Scalability of Generation

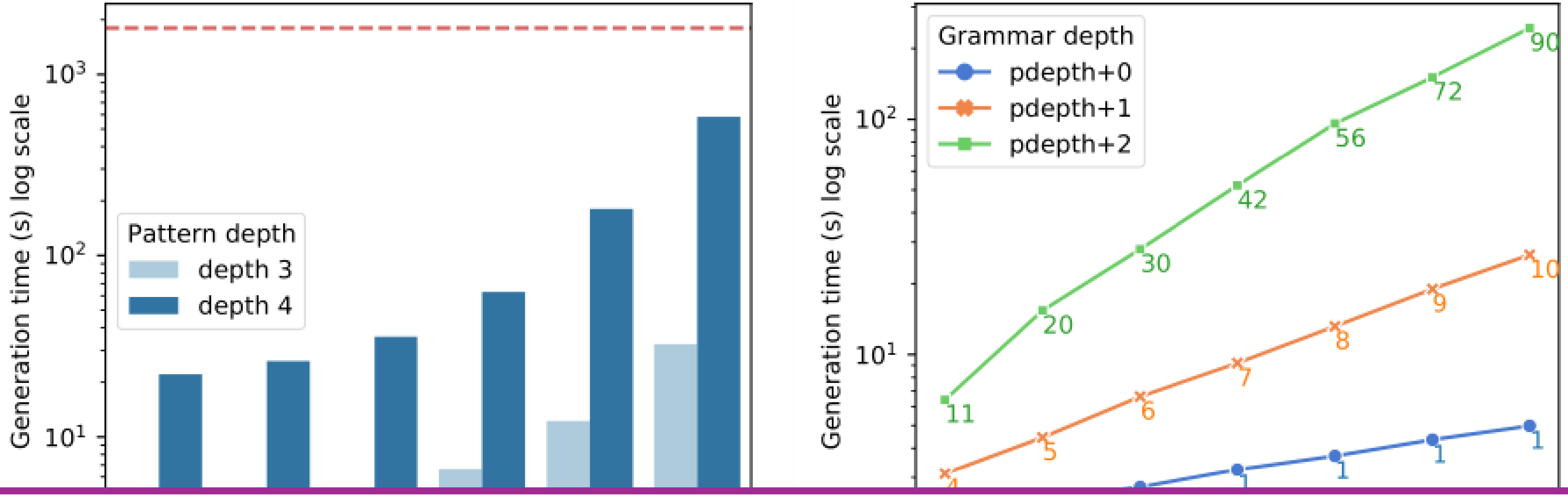
With microarchitectural complexity and grammar depth





Results: Scalability of Generation

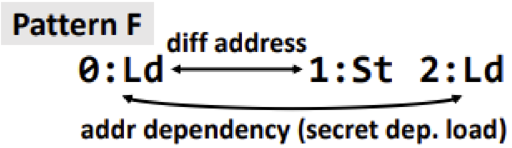
With microarchitectural complexity and grammar depth



- Exponential scaling in microarch. parameters and depth
- Reasonable for abstract models
- Future work: Evaluate performance with RTL designs

Results: False positives

Patterns are prone to false positives

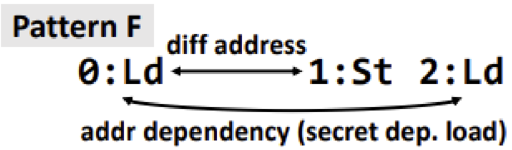


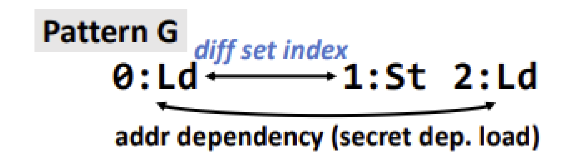

```
void test_K (uint32_t idx) {
  // Address (A) = (arr1+idx)
  _temp = arr1[idx];     // Ld0: LSQC Index = A[SET_W+1:2]
  arr1[idx+(1<<K)] = 0; // St0: LSQC Index = (A+(1<<K))[SET_W+1:2]
  _temp1 = arr2[_temp]; // Ld1
}
```

Check	Result with test_K (Fig. 14) and SET_W set index	
	$K > SET\_W + 2$	$K \leq SET\_W + 2$
Hyperproperty	SAFE	UNSAFE
Pat. F	UNSAFE	UNSAFE

Results: False positives

Patterns are prone to false positives






```
void test_K (uint32_t idx) {
  // Address (A) = (arr1+idx)
  _temp = arr1[idx];    // Ld0: LSQC Index = A[SET_W+1:2]
  arr1[idx+(1<<K)] = 0; // St0: LSQC Index = (A+(1<<K))[SET_W+1:2]
  _temp1 = arr2[_temp]; // Ld1
}
```

Check	Result with test_K (Fig. 14) and SET_W set index	
	$K > SET\_W + 2$	$K \leq SET\_W + 2$
Hyperproperty	SAFE	UNSAFE
Pat. F	UNSAFE	UNSAFE
Pat. G	SAFE	UNSAFE

Grammar exposes a precision-complexity tradeoff

Takeaways

Motivation: extend formal guarantees from hyperproperties to patterns

Generation Approach: template exploration + grammar-based counterfactual constraint addition

Results: new patterns, order of magnitude verification runtime improvement, pattern-grammar tradeoff

SemPat: From Hyperproperties to Attack Patterns for Scalable Analysis of Microarchitectural Security. Adwait Godbole, Yatin A. Manerkar, Sanjit A. Seshia. ACM CCS 2024. Salt Lake City, UT.

Send mail! adwait@berkeley.edu

Questions?