Scalable Program Analysis: Abstract Interpretation Techniques and Practical Applications

YUSEN SU

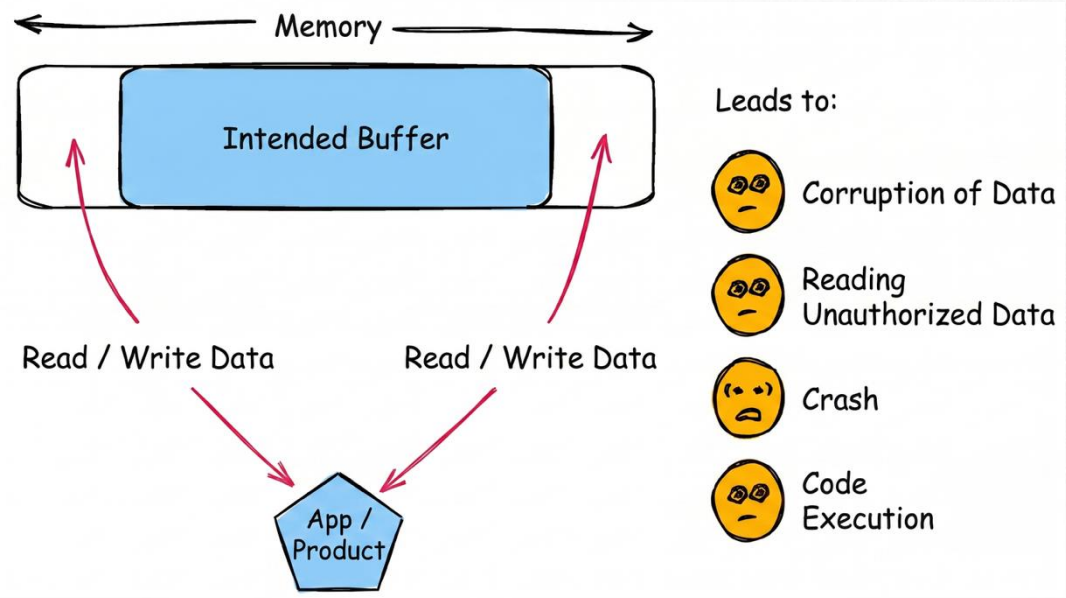
SUPERVISOR: PROF. ARIE GURFINKEL

PHD DEFENSE

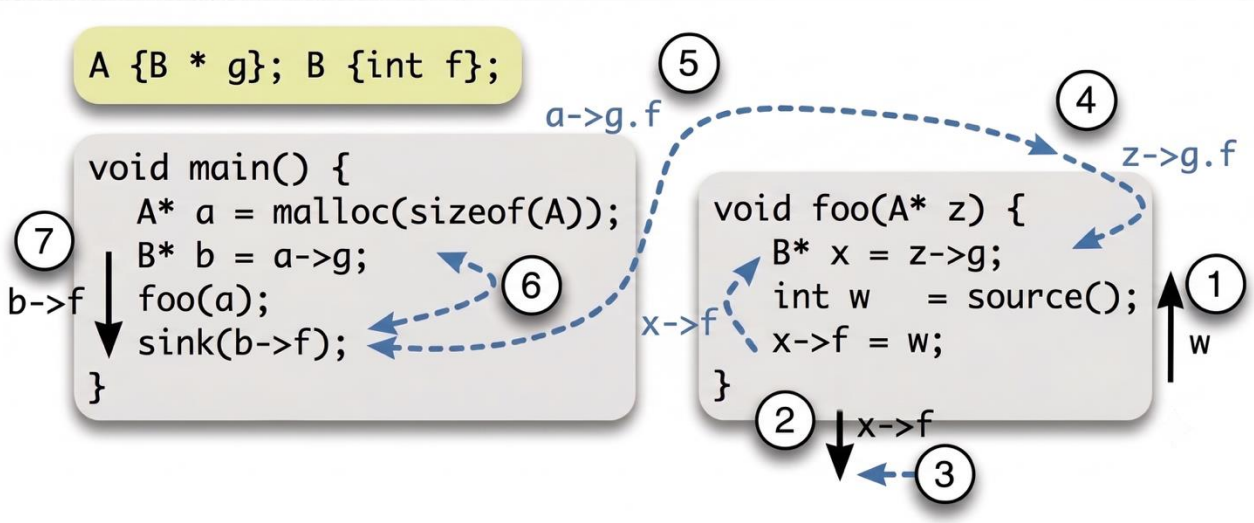
UNIVERSITY OF WATERLOO

Interests - Two C Security Properties

Spatial Memory Safety



Data-flow Security



Note: Code adapted from FlowDroid (Java) to C. Source: Arzt et al., " FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps" ACM SIGPLAN PLDI, 2014.

Two related memory corruption weaknesses:
out-of-bounds read (CWE-125) write (CWE-787)

Code illustrating exposure of sensitive information:
Information leak (CWE-200)

Approach - Formal Verification

Formal Verification

- Mathematically proves properties hold:

Buffer Bounds Safety Property	Data-flow Security Property
• Memory is byte-addressable $\text{is\_accessible}(p, by)$ • A valid pointer p has base $b \in \mathbb{N}$, *offset* $o \in \mathbb{N}$ • Access touches bytes $by \in \mathbb{N}_{>0}$ starting at $b + o$ • Range must stay within allocated region $[b, b+n]$, $n \in \mathbb{N}_{>0}$ $\underbrace{0 \leq o}_{\text{No underflow}} \wedge \underbrace{o + by \leq n}_{\text{No overflow}}$ • Assume $p.\texttt{offset} = o$ and $p.\texttt{size} = n$	• Focuses on taint via explicit data dependencies • Variables $V = T \cup C$ • Two sets *tainted* T and *clean* C, $T \cap C = \emptyset$ • Every argument x passed to a sink function must not be tainted: $x \notin T$

Specifically, Abstract Interpretation (AbsInt)

A theory of sound approximation of the semantics of computer programs

We can build a static analysis that:

- Sound --- over-approximates all program executions
  - For checked properties: no false negatives
- Incomplete --- not all warnings are true errors, false positives (false alarms)
- Automatic --- infers invariants from a predefined abstract domain

Static Analysis by Abstract Interpretation

Input Program

Code annotations
- nd for external input
- assert for property check

Collecting Semantics
2^{States}

Semantics
- All reachable states at every program point

Precision
- Precise, but generally **uncomputable**

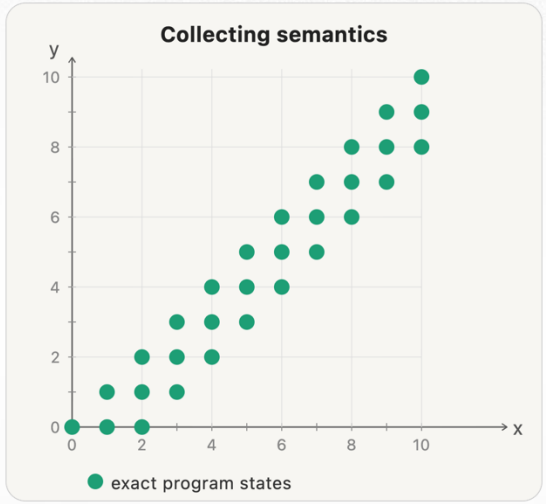
Abstract Semantics
$D^{\#}$

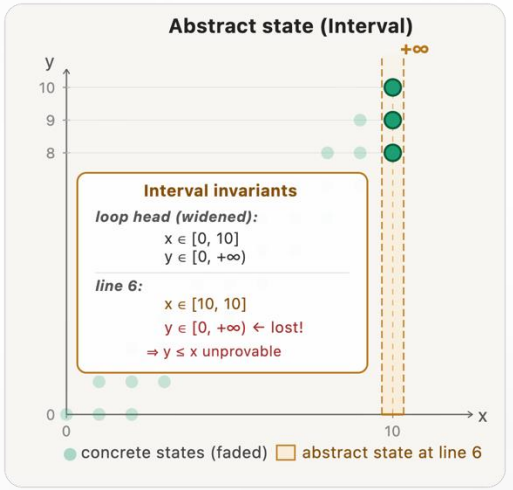
Approximation
- Abstract states over-approximate all concrete states

Domain Operations
- join, meet, widening, etc.

Precision
- Imprecise but **computable**

```
1  x = nd(0, 2); y = 0;
2  while (x < 10) {
3    x = x + 1;
4    y = y + 1;
5  }
6  assert(y <= x);
```





The Challenge — Precision vs. Performance

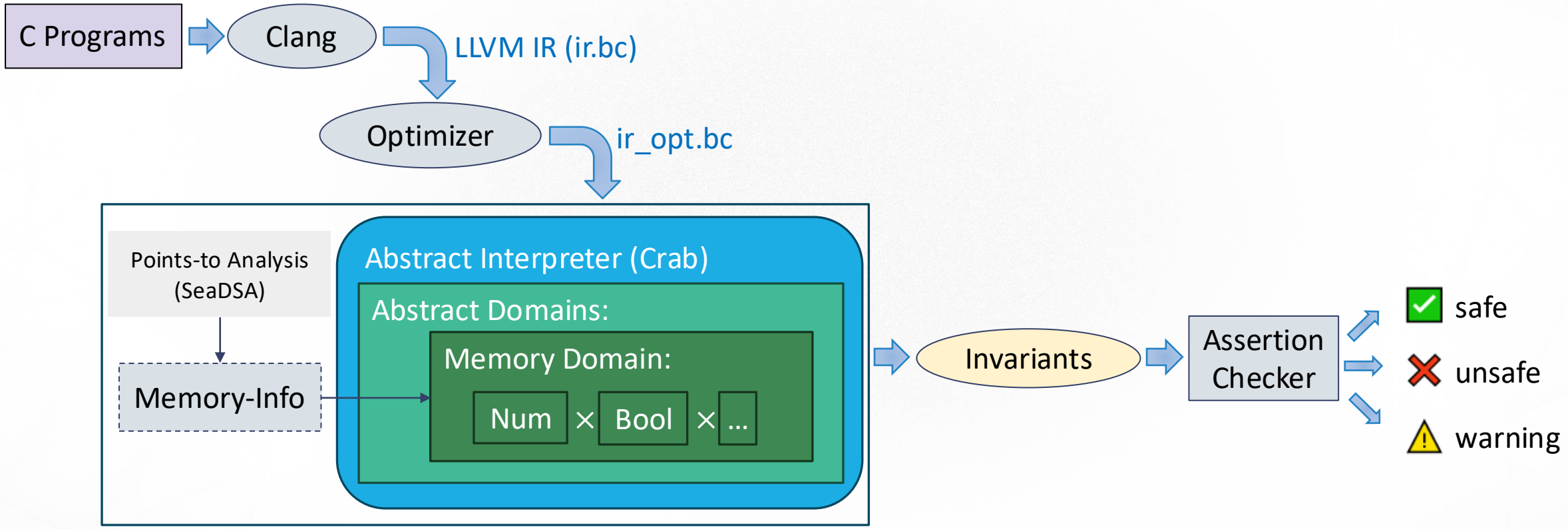
Domain name	Representable Constraints	Domain Operation Cost	Memory
Intervals	$a \leq x \leq b$	$O(n)$	$O(n)$
Zones	$x - y \leq b$	$O(n^3)$	$O(n^2)$
Octagons	$\pm x \pm y \leq c$	$O(n^3)$	$O(n^2)$
Convex Polyhedra	$\Sigma_i a_i x_i \leq c$	exponential	exponential

Example 

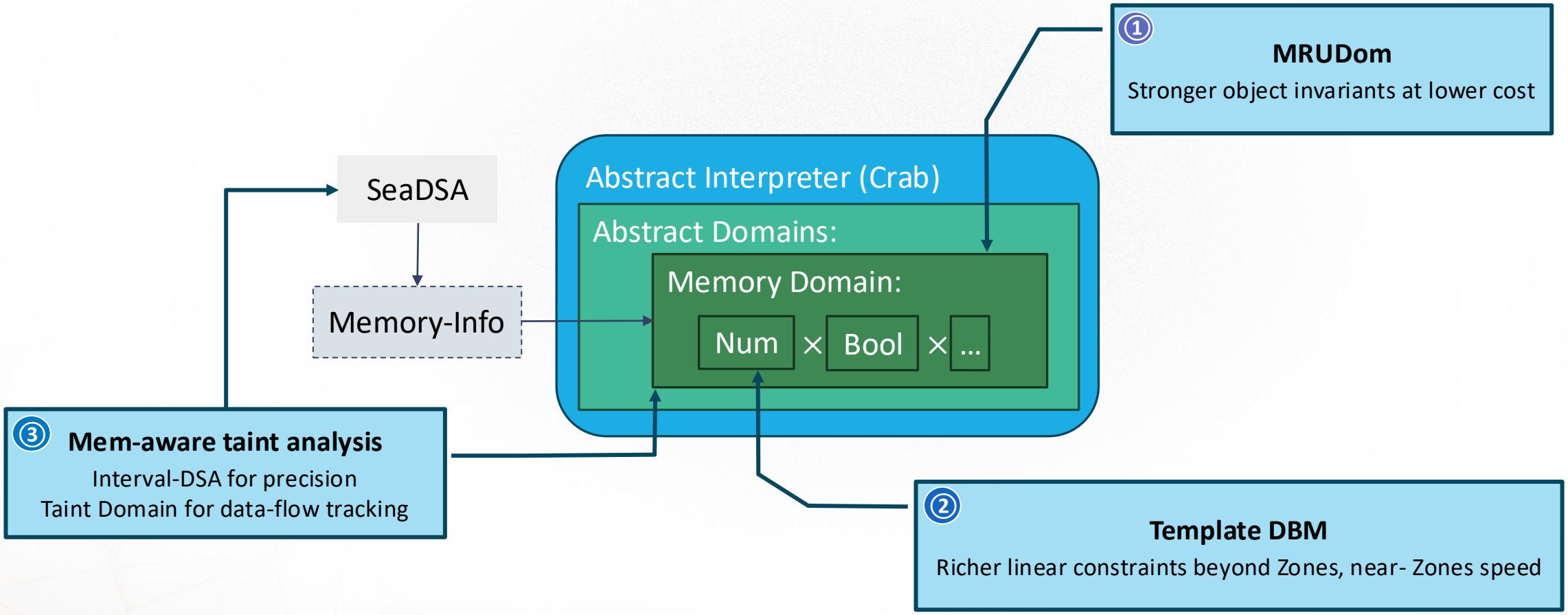
```
1  x = nd(0, 2); y = 0;
2  while (x < 10) {
3    x = x + 1;
4    y = y + 1;
5  }
6  assert(y <= x);
```

How can we make verification precise without extra cost?

AbsInt Analysis for LLVM IR



My Contributions



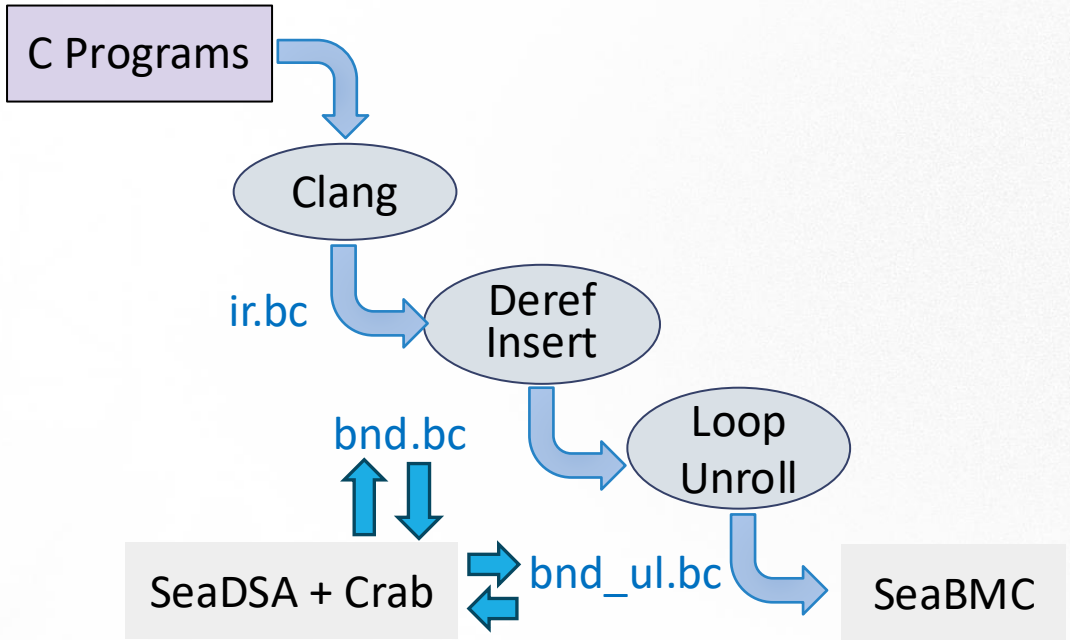
Spatial Memory Safety

MRUDOM: A MEMORY DOMAIN[1]

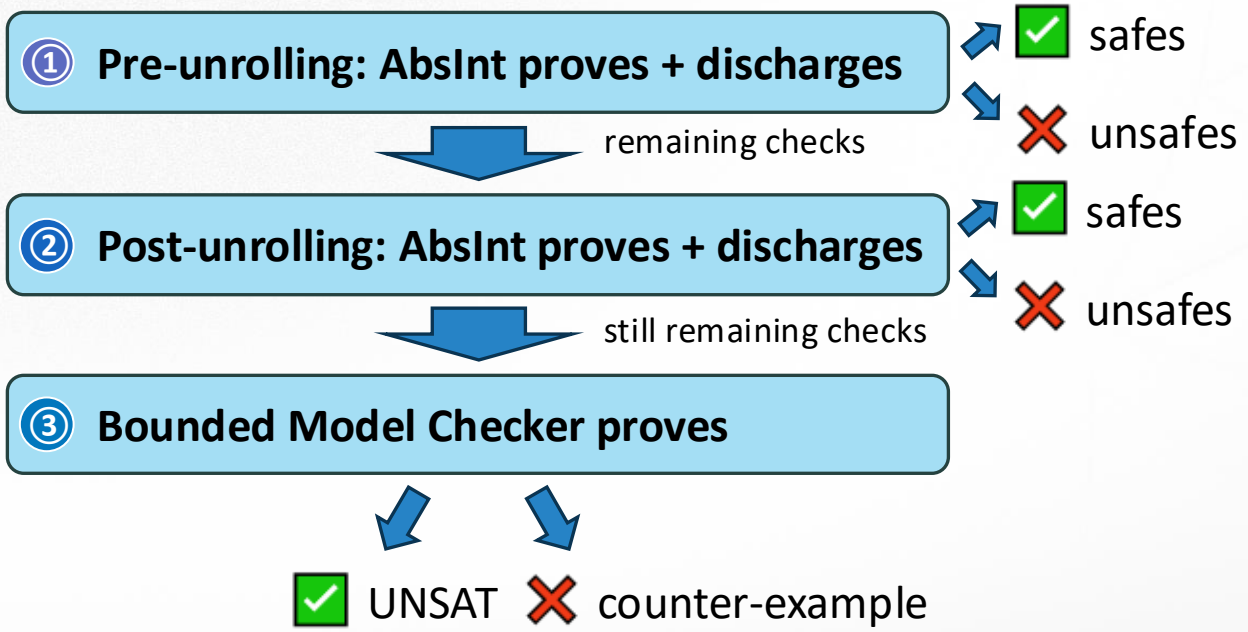
TEMPLATE DBM: A NUMERICAL DOMAIN

1. Su, Y., et al.: Automatic inference of relational object invariants. In: VMCAI 2025.

Motivation: AbsInt + BMC



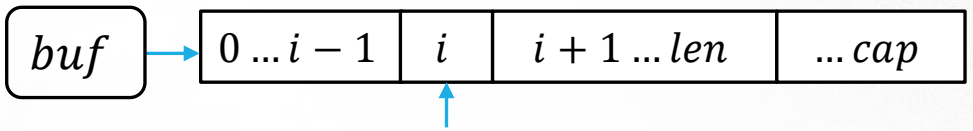
- Bounded model checker (BMC)
- AI4BMC: AbsInt + BMC pipeline
- AbsInt discharges **buffer checks**, reducing BMC overhead



Relational Object Invariants (1)

An **object invariant (OI)** is a property that holds for an object whenever the object is valid (after initialization).

```
struct byte_buf {
 int len; int cap;
 char *buf;
};
```



The invariants of a non-empty byte_buf are:

$$0 < cap \ \wedge 0 \leq len \ \wedge len \leq cap \ \wedge buf.\texttt{size} = cap$$

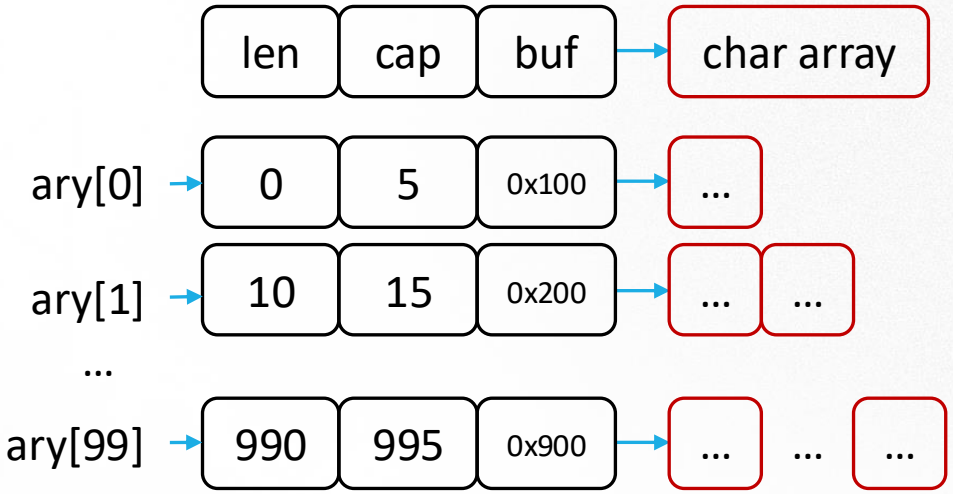
index i is guaranteed by OI: $i < len \leq cap$ *ensures no out-of-bounds access*

Memory Safety Property: Any read/write `buf[i]` must satisfy $0 \leq i < cap$

But OI may be *temporarily violated* during execution

Motivation: Precision Loss

OI shared across objects at the same site

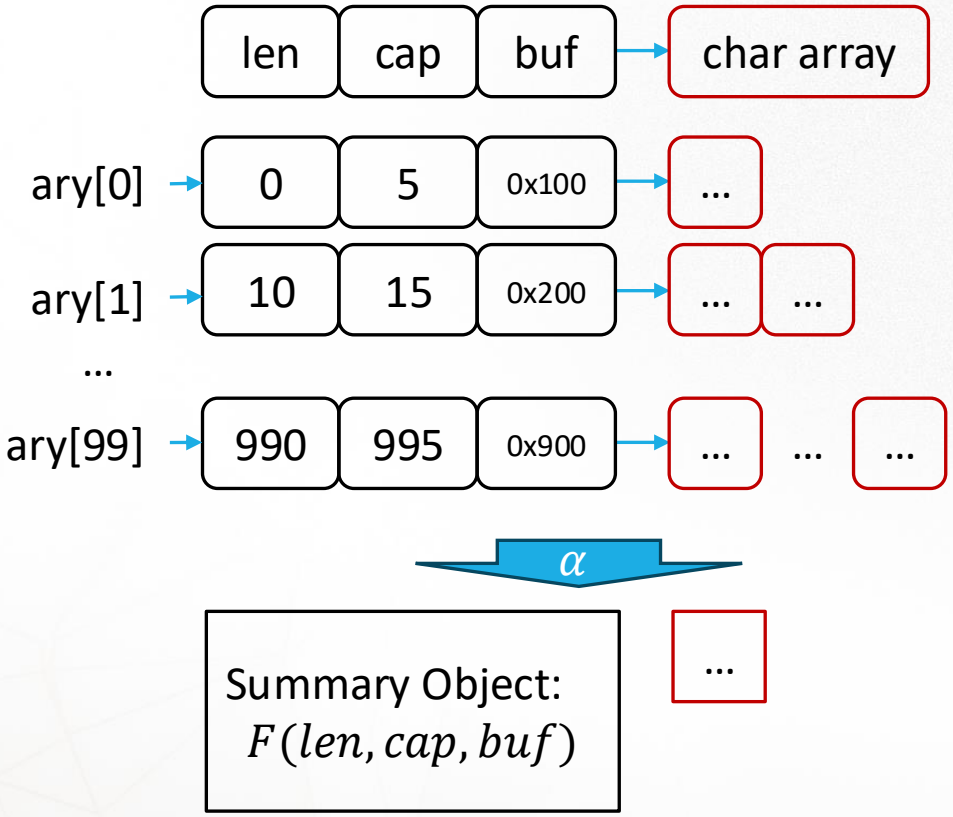


$$cap = len + 5 \ \wedge buf.size = cap$$

```
1   struct byte_buf { int len; int cap; char *buf; };

2   void main() {
3     struct byte_buf *ary[100];
4     for (int i = 0; i < 100; i++) {
5       ary[i] = malloc(sizeof(struct byte_buf));
6       int chunk = 10 * i;
7       ary[i]->len = chunk;
8       ary[i]->cap = chunk + 5;
9       ary[i]->buf = malloc(sizeof(char) * chunk);
10    }
11    ...
12  }
```

Motivation: Precision Loss

OI shared across objects at the same site

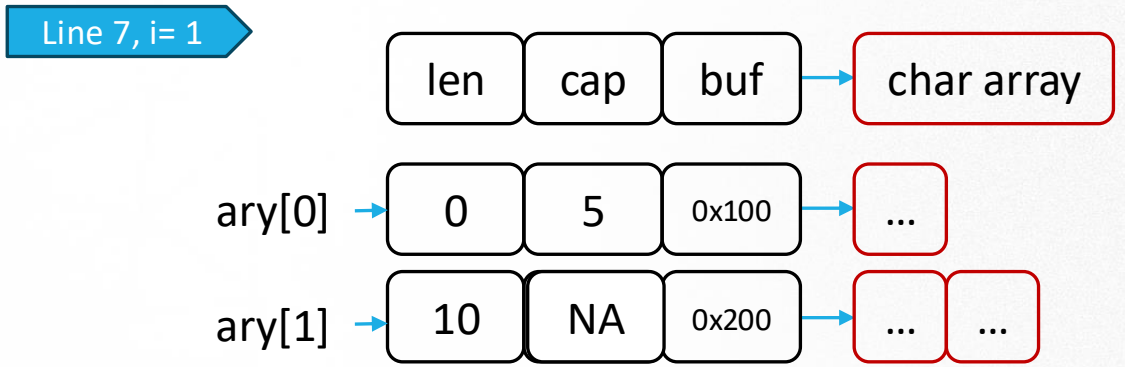



```
1   struct byte_buf { int len; int cap; char *buf; };

2   void main() {
3     struct byte_buf *ary[100];
4     for (int i = 0; i < 100; i++) {
5       ary[i] = malloc(sizeof(struct byte_buf));
6       int chunk = 10 * i;
7       ary[i]->len = chunk;
8       ary[i]->cap = chunk + 5;
9       ary[i]->buf = malloc(sizeof(char) * chunk);
10    }
11    ...
12  }
```

Allocation Site Abstraction:
one abstract object to summarize OI for all objects

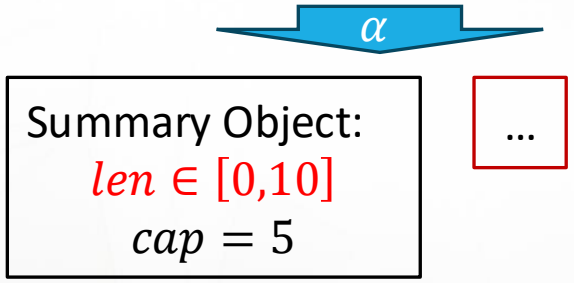
Motivation: Precision Loss

OI shared across objects at the same site




```
1  struct byte_buf { int len; int cap; char *buf; };

2  void main() {
3    struct byte_buf *ary[100];
4    for (int i = 0; i < 100; i++) {
5      ary[i] = malloc(sizeof(struct byte_buf));
6      int chunk = 10 * i;
7      ary[i]->len = chunk;
8      ary[i]->cap = chunk + 5;
9      ary[i]->buf = malloc(sizeof(char) * chunk);
10   }
11   ...
12 }
```

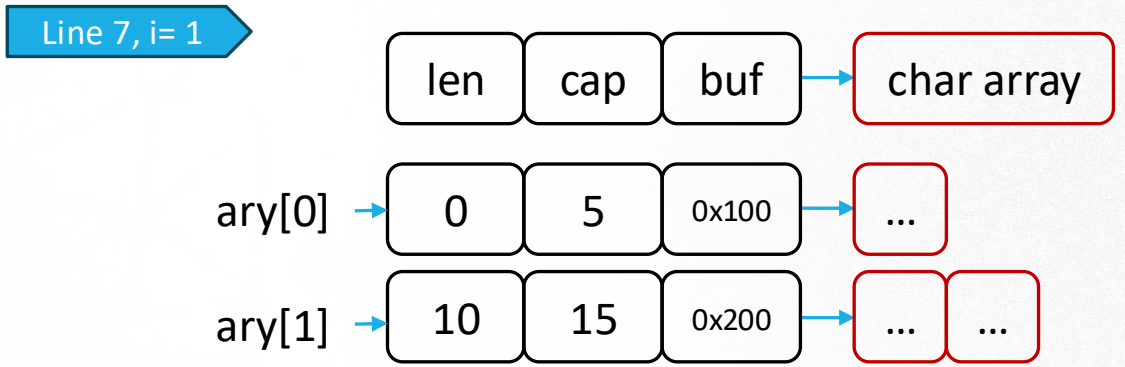
Weak update: one write corrupts all invariants



Allocation Site Abstraction:
one abstract object to summarize OI for all objects

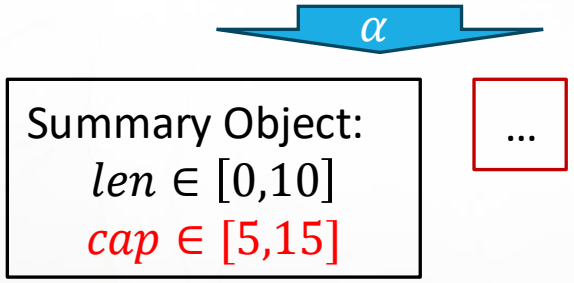
Motivation: Precision Loss

OI shared across objects at the same site




```
1   struct byte_buf { int len; int cap; char *buf; };

2   void main() {
3     struct byte_buf *ary[100];
4     for (int i = 0; i < 100; i++) {
5       ary[i] = malloc(sizeof(struct byte_buf));
6       int chunk = 10 * i;
7       ary[i]->len = chunk;
8       ary[i]->cap = chunk + 5;
9       ary[i]->buf = malloc(sizeof(char) * chunk);
10    }
11    ...
12  }
```

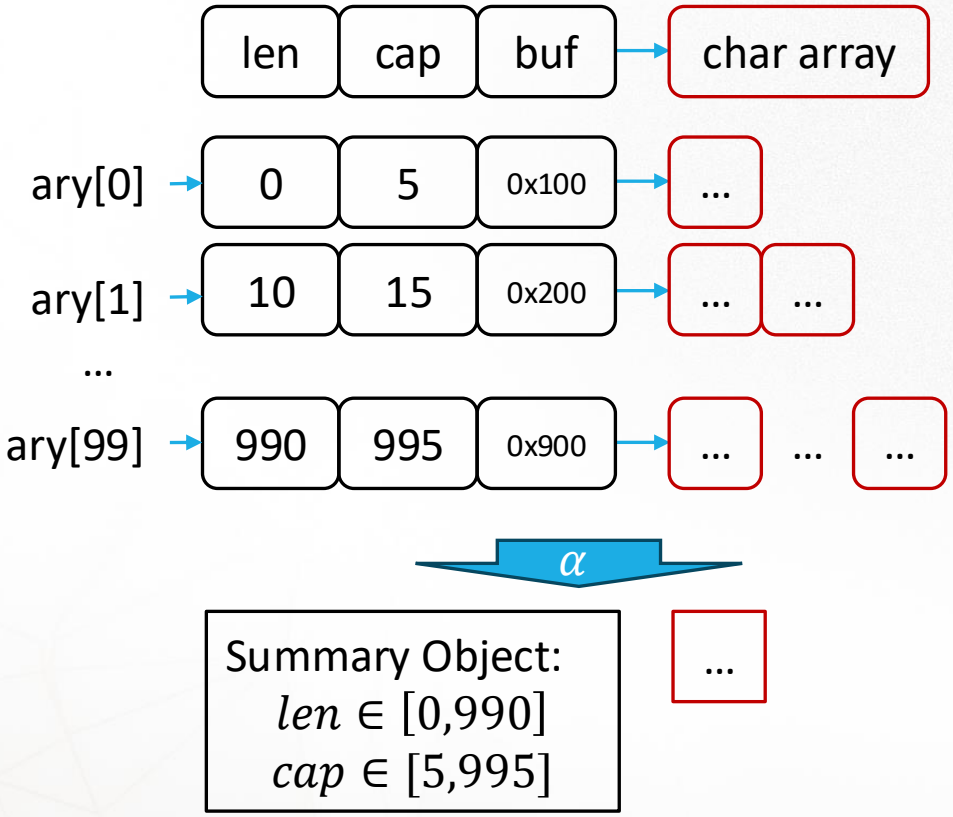
Weak update: one write corrupts all invariants



Allocation Site Abstraction:
one abstract object to summarize OI for all objects

Motivation: Precision Loss

OI shared across objects at the same site

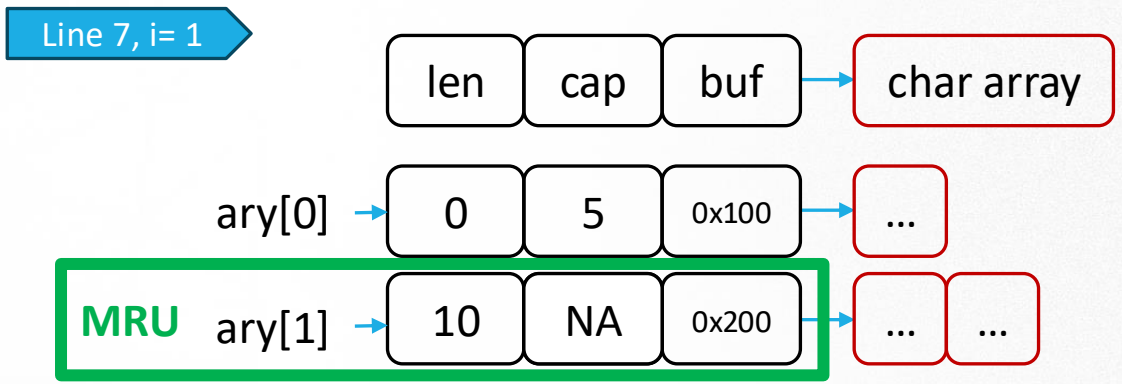



```
1   struct byte_buf { int len; int cap; char *buf; };

2   void main() {
3     struct byte_buf *ary[100];
4     for (int i = 0; i < 100; i++) {
5       ary[i] = malloc(sizeof(struct byte_buf));
6       int chunk = 10 * i;
7       ary[i]->len = chunk;
8       ary[i]->cap = chunk + 5;
9       ary[i]->buf = malloc(sizeof(char) * chunk);
10    }
11    ...
12  }
```

Weak update: one write corrupts all invariants

Allocation Site Abstraction:
one abstract object to summarize OI for all objects

Our Solution: Recency-Use




```
1   struct byte_buf { int len; int cap; char *buf; };

2   void main() {
3     struct byte_buf *ary[100];
4     for (int i = 0; i < 100; i++) {
5       ary[i] = malloc(sizeof(struct byte_buf));
6       int chunk = 10 * i;
7       ary[i]->len = chunk;
8       ary[i]->cap = chunk + 5;
9       ary[i]->buf = malloc(sizeof(char) * chunk);
10    }
11    ...
12  }
```

Strong update: write only for MRU object

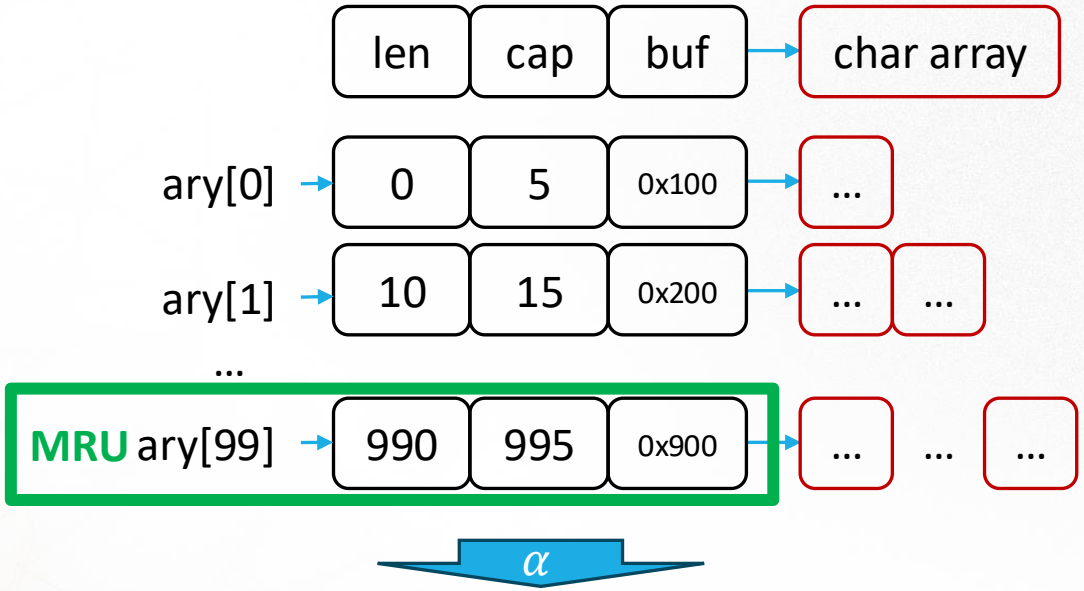
α

MRU:
$len = 10$

Summary Object:
$len = 0, cap = 5$

Recency-Use abstraction:
the accessed object stored OI for itself
\+ one abstract object to summarize OI for other objects

Our Solution: Recency-Use



MRU:
$len = 990, cap = 995$

Summary Object:
$len \in [0,980]$,
$cap = len + 5$

```
1   struct byte_buf { int len; int cap; char *buf; };

2   void main() {
3     struct byte_buf *ary[100];
4     for (int i = 0; i < 100; i++) {
5       ary[i] = malloc(sizeof(struct byte_buf));
6       int chunk = 10 * i;
7       ary[i]->len = chunk;
8       ary[i]->cap = chunk + 5;
9       ary[i]->buf = malloc(sizeof(char) * chunk);
10    }
11    ...
12  }
```

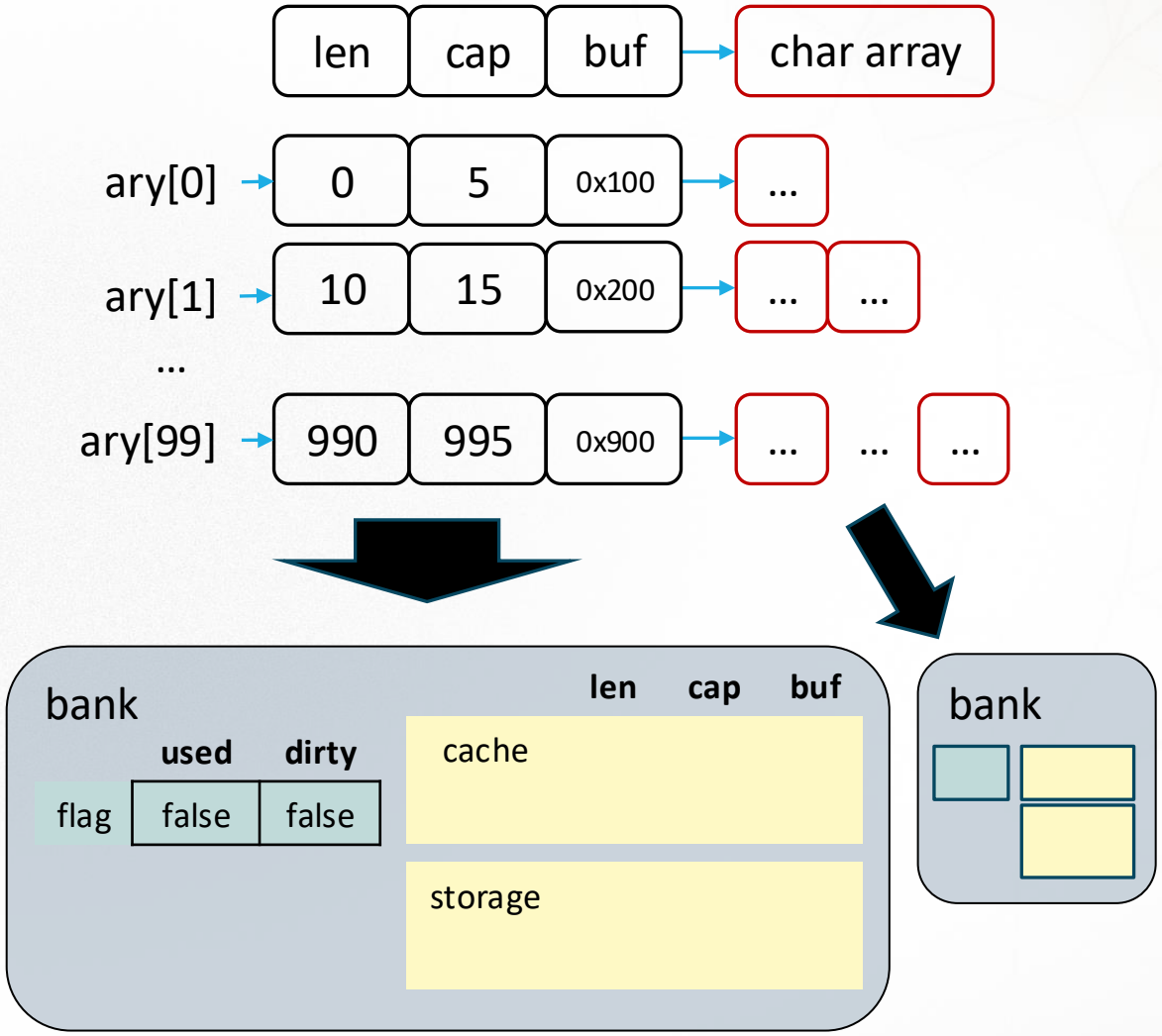
Strong update: write only for MRU object

Recency-Use abstraction:
the accessed object stored OI for itself
\+ one abstract object to summarize OI for other objects

Methodology

① Memory Model for Recency-Use

Specify how memory is represented, and establish rules for access



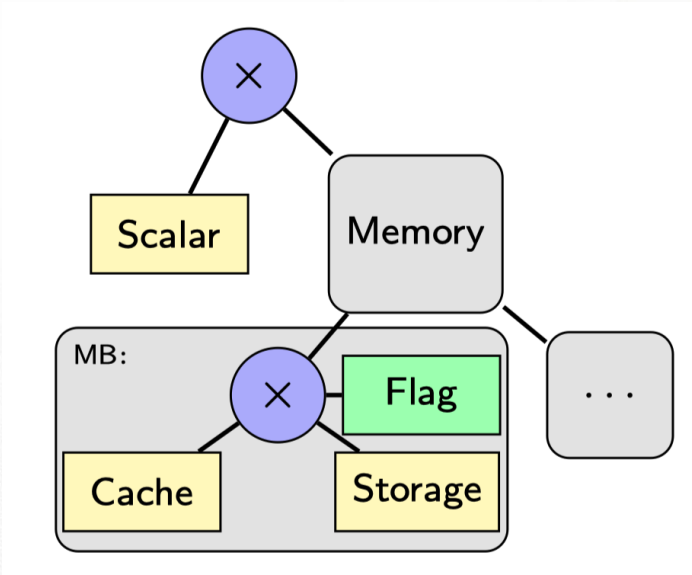
Methodology

① **Memory Model for Recency-Use**

Specify how memory is represented, and establish rules for access

② **Concrete Domain**

Define concrete semantics



Methodology

① Memory Model for Recency-Use

Specify how memory is represented, and establish rules for access

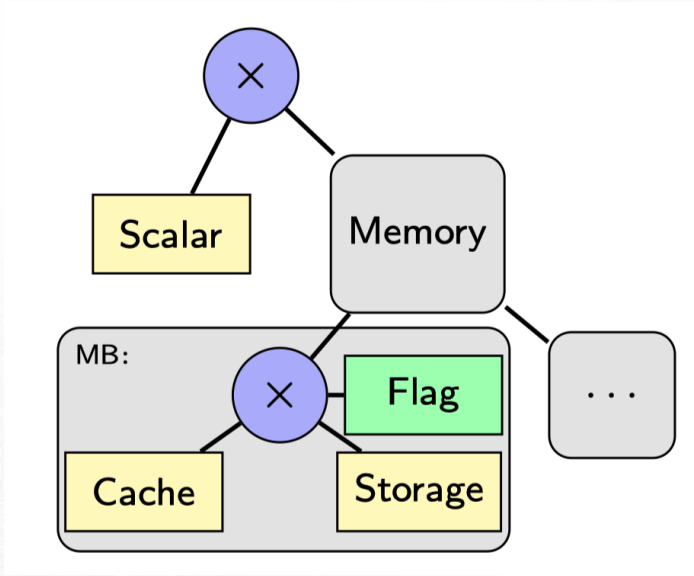
②

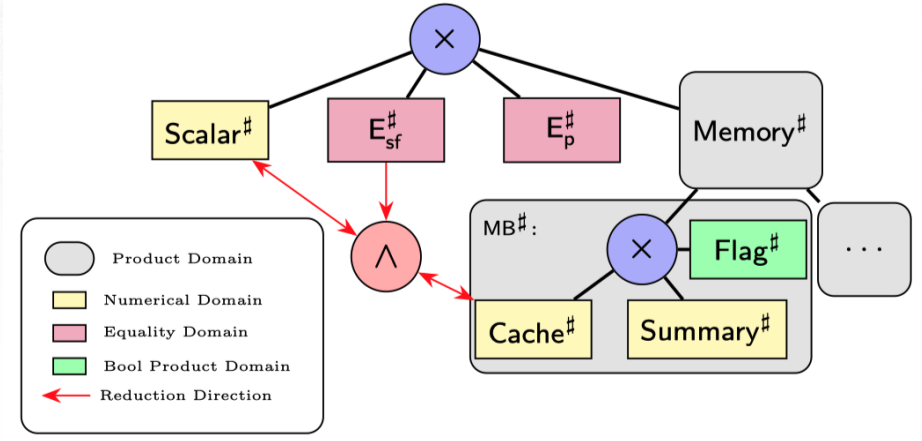
Concrete Domain

Define concrete semantics

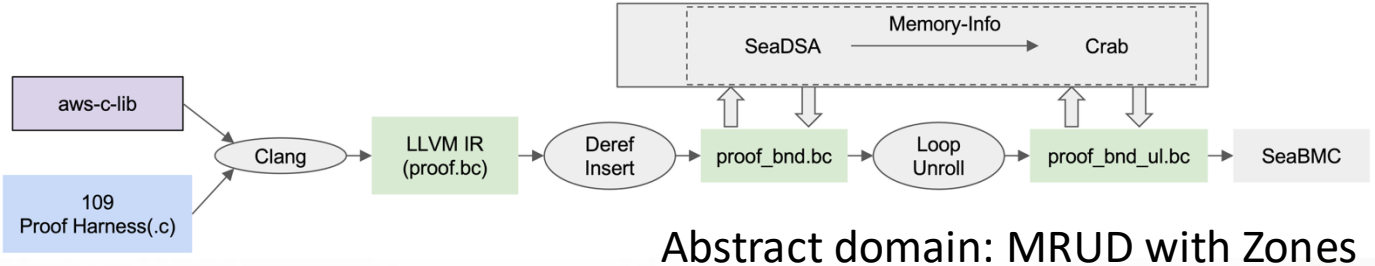
③ Abstract Domain (MRUD)

Introduce a new memory domain following Recency-Use abstraction





AI4BMC vs. BMC



Abstract domain: MRUD with Zones
SMT-solvers: Z3 and Yices2

Performance (Z3)

Fewer Timeouts

Solver	Z3	
Pipeline	AI4BMC	BMC
Timeouts (900s)	5	7

AI4BMC performs better on 16 cases because:

- AbsInt proved assertions
  - (9 programs, 100%; 6 programs, > 50%)

AbsInt adds minimal cost:

- most < 2s

Precision of Absint

Among 104 cases,

AbsInt Solving Rate before LU	100%	37
	> 50%	52
AbsInt Solving Rate after LU	100%	6
	> 50%	1

- 8 cases solved $\leq 50\%$ of assertions

Precision loss from:

- Widening
- Not handling C strings
- Using Zones instead of Polyhedra

Spatial Memory Safety

MRUDOM: A MEMORY DOMAIN

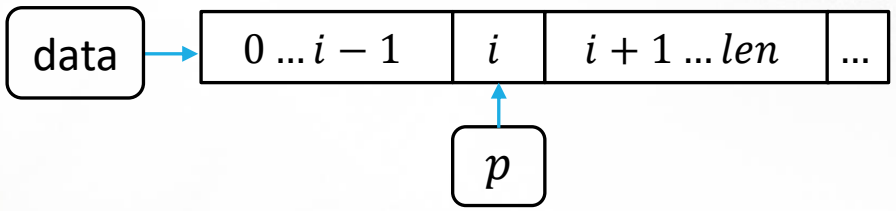
TEMPLATE DBM: A NUMERICAL DOMAIN[1]

1. Su, Y., et al.: Template DBM: A New Weakly Relational Domain for Efficient Memory-Access Validation. In: VSTTE 2025.

Relational Object Invariants (2)

Not all OIs can be represented by Zones, for example:

```
struct array_list {
  int size;
  int len;
  int isz; // item size
  void *data;
};
```



The invariants of a non-empty integer array_list are:

$$isz = 4 \wedge 0 \leq len \wedge 1 \leq size \wedge 4 * len \leq size \wedge data.\texttt{size} = size$$

Pointer p access to `data[i]` is granted by OI:

$$4 * i = p.\texttt{offset}$$
$$4 * len \leq p.\texttt{size}$$
$$0 \leq i \leq len - 1$$

Safe access must satisfy:

$$p.\texttt{offset} + 4 \leq p.\texttt{size}$$

Which **numerical domain** represents this set?

How is this **implicit constraint** discovered?

Existing Numerical Domains

$$4 * i = p.\texttt{offset}$$
$$4 * len \leq p.\texttt{size}$$
$$0 \leq i \leq len - 1$$

Unit Two Variable Per Inequality (UTVPI):

$$-i \leq 0$$
$$i - len \leq -1$$

Two Variable Per Inequality (TVPI):

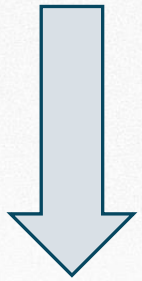
$$p.\texttt{offset} - 4 * i \leq 0$$
$$4 * i - p.\texttt{offset} \leq 0$$
$$4 * len - p.\texttt{size} \leq 0$$

Domain name	Representable Constraints	Expressible	Cost and Expressiveness
Intervals	$c \leq x \leq d$	✗	
UTVPI (Zones)	$x - y \leq c$	✗	
UTVPI (Octagons)	$\pm x \pm y \leq c$	✗	
TVPI[1]	$ax + by \leq c$	✓	
Convex Polyhedra	$\Sigma_i a_i x_i \leq c$	✓	

1. Simon, Axel, et al.: Two Variables per Linear Inequality as an Abstract Domain. In: LOPSTR 2002.

Overview

GOAL

Domain name	Representable Constraints	Cost and Expressiveness
UTVPI (Zones)	$x - y \leq c$	
Template DBM	$ax - by \leq c, a, b \in T$ where T is the **template set**.	
TVPI	$ax + by \leq c$	

METHODOLOGY

① **Express Subset of TVPI Constraints**
More expressive than Zones

② **Derive Implicit Constraint**
Algorithm for implicit derivation

③ **New Domain (Template DBM)**
New numerical domain based on DBM

Idea: TVPI as UTVPI

Express TVPI constraint as UTVPI constraint

- UTVPI constraint $\Longrightarrow$ Difference Bound Matrix (DBM)
- Non-unit coefficient (e.g., $4 * i$) $\Longrightarrow$ one DBM dimension ($\texttt{4i}$)
- Dimensions fixed by coefficient template (e.g., $\{1,4\}$)

UTVPI:

$$-i \leq 0$$
$$i - len \leq -1$$

TVPI:

$$p.\texttt{offset} - \texttt{4i} \leq 0$$
$$\texttt{4i} - p.\texttt{offset} \leq 0$$
$$\texttt{4len} - p.\texttt{size} \leq 0$$

UTVPI:

$$\begin{array}{c} \\ v^0 \\ i \\ len \end{array} \begin{array}{c} \begin{array}{ccc} v^0 & i & len \end{array} \\ \begin{pmatrix} +\infty & +\infty & +\infty \\ 0 & +\infty & +\infty \\ +\infty & -1 & +\infty \end{pmatrix} \end{array}$$

TVPI:

$$\begin{array}{c} \\ p.\texttt{offset} \\ p.\texttt{size} \\ \texttt{4i} \\ \texttt{4len} \end{array} \begin{array}{c} \begin{array}{cccc} p.\texttt{offset} & p.\texttt{size} & \texttt{4i} & \texttt{4len} \end{array} \\ \begin{pmatrix} +\infty & +\infty & 0 & +\infty \\ +\infty & +\infty & +\infty & 0 \\ 0 & +\infty & +\infty & +\infty \\ +\infty & 0 & +\infty & +\infty \end{pmatrix} \end{array}$$

DBM

Deriving the implicit constraint

Fourier-Motzkin variable elimination:

i. $(b = d)$: $ax - by \leq c \ \wedge dy - ez \leq f \overset{①}{\Rightarrow} ax - ez \leq c + f$

ii. $(b \neq \mathrm{d})$: $ax - by \leq c \ \wedge dy - ez \leq f \overset{②}{\Rightarrow} a'x - b'z \leq c'$

DBM Saturation (Closure) to **implicit → explicit**

Handles cases DBM closure cannot

UTVPI:

$$-i \leq 0$$
$$i - len \leq -1$$

TVPI:

$$p.\texttt{offset} - \texttt{4i} \leq 0$$
$$\texttt{4i} - p.\texttt{offset} \leq 0$$
$$\texttt{4len} - p.\texttt{size} \leq 0$$

$$p.\texttt{offset} + 4 \leq p.\texttt{size}$$

Deriving the implicit constraint

Fourier-Motzkin variable elimination:

i. $(b = d)$: $ax - by \leq c \ \wedge dy - ez \leq f \Rightarrow^{①} ax - ez \leq c + f$

ii. $(b \neq \mathrm{d})$: $ax - by \leq c \ \wedge dy - ez \leq f \Rightarrow^{②} a'x - b'z \leq c'$

DBM Saturation (Closure) to **implicit → explicit**

Handles cases DBM closure cannot

UTVPI:

$-i \leq 0$

$i - len \leq -1$

TVPI:

$p.\texttt{offset} - 4\texttt{i} \leq 0$

$4\texttt{i} - p.\texttt{offset} \leq 0$

$4\texttt{len} - p.\texttt{size} \leq 0$

$\Rightarrow^{②} p.\texttt{offset} - 4\texttt{len} \leq -4$

$p.\texttt{offset} + 4 \leq p.\texttt{size}$

Deriving the implicit constraint

Fourier-Motzkin variable elimination:

i. $(b = d)$: $ax - by \leq c \ \wedge dy - ez \leq f \Rightarrow$ ① $ax - ez \leq c + f$

ii. $(b \neq \mathrm{d})$: $ax - by \leq c \ \wedge dy - ez \leq f \Rightarrow$ ② $a'x - b'z \leq c'$

DBM Saturation (Closure) to **implicit → explicit**

Handles cases DBM closure cannot

UTVPI:

$-i \leq 0$

$i - len \leq -1$

TVPI:

$p.\mathtt{offset} - 4\mathtt{i} \leq 0$

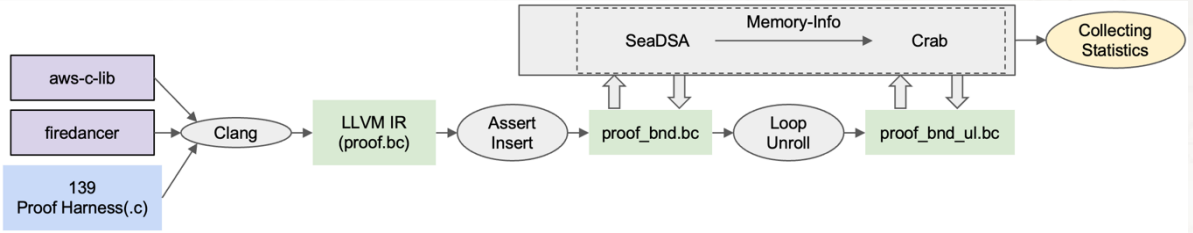
$4\mathtt{i} - p.\mathtt{offset} \leq 0$

$4\mathtt{len} - p.\mathtt{size} \leq 0$

② $\Rightarrow p.\mathtt{offset} - 4\mathtt{len} \leq -4$

① $\Rightarrow p.\mathtt{offset} - p.\mathtt{size} \leq -4$

$p.\mathtt{offset} + 4 \leq p.\mathtt{size}$



Zones vs. Ours vs. Polyhedra

Performance

Close to Zones, faster than Polyhedra

Domain	Before Unrolling		After Unrolling	
	Mean(s)	Std	Mean	Std
Zones	0.2	0.6	0.1	0.5
Template DBM	0.4	1.9	0.2	0.7
Polyhedra	2.5	15.0	2.2	7.4

Scatter plot

Precision

Above Zones, near PK

Bench. (before unroll)	Zones	Template DBM	Polyhedra
aws/array_list	75%	83%	83%
firedancer/tango	36%	85%	100%

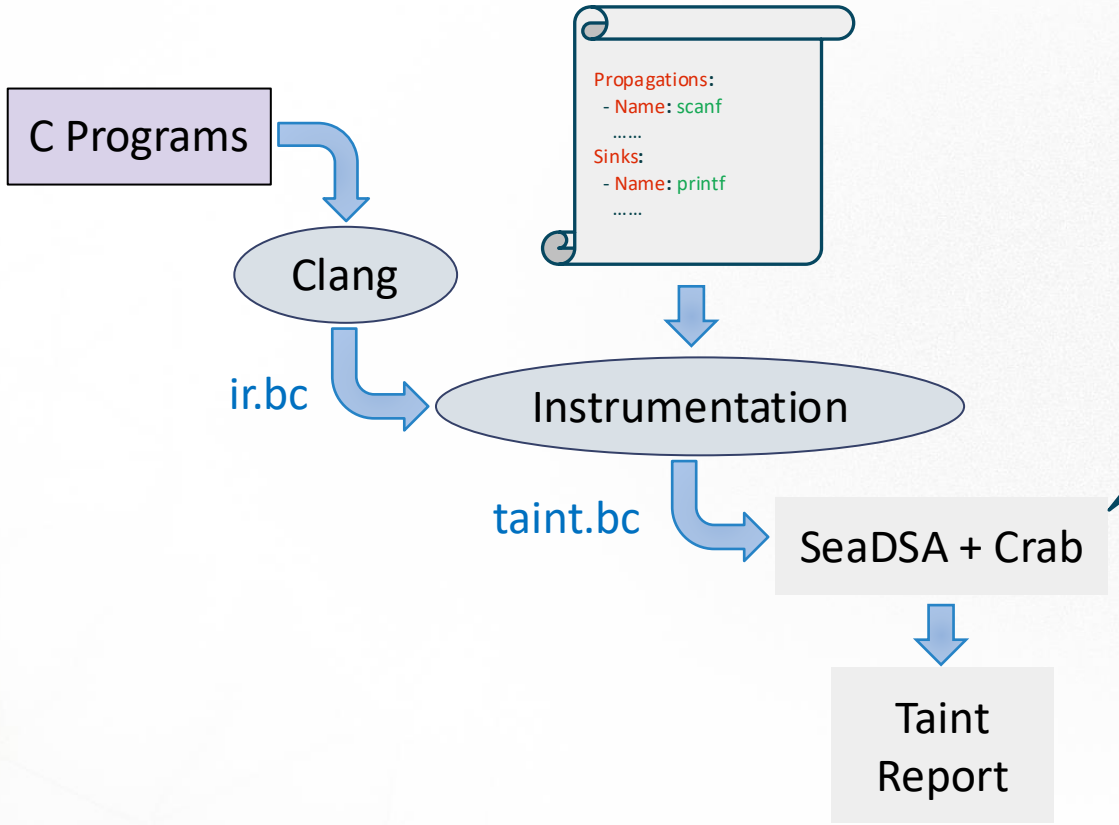
Table

Taint Analysis

A NEW FIELD-SENSITIVE LLVM IR TAINT ANALYSIS[1]

1. Su, Y., et al.: Taint Analysis via Heap-Aware Propagation. Submitted to: SAS 2026.

Motivation: AbsInt + Taint Analysis



Field-sensitivity + Explicit Data-flow

Source $\overset{\text{data}}{\rightsquigarrow}$ Object.Field $\overset{\text{data}}{\rightsquigarrow}$ Sink

Data Structure Analysis (DSA): memory graph

nodes = { $n \mid n = \alpha(objects)$ } *edges* = *points-to*

Data Flow Analysis (DFA): taint flow

tracks tainted fields/registers

Motivation: Precision Loss

Field-insensitive ➡ Spurious taint flows

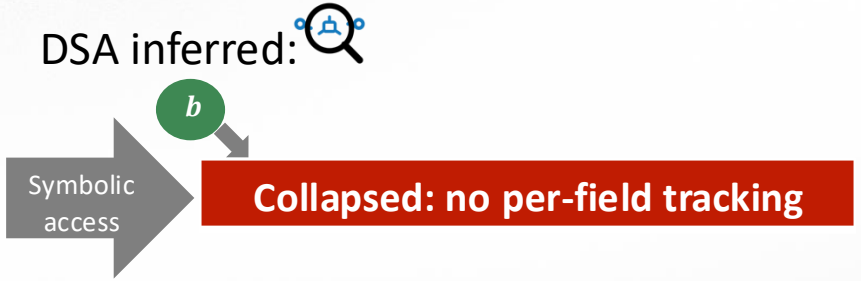
```
typedef struct repl_block {
  int id;
  size_t size;
  size_t used;
  char buf[]; // flexible array member
} repl_block_t;
```

```
1  static int block_id = 0;
2  void feedReplicationBuffer(
3        const char *s, size_t len // sources
4  ){
5    repl_block_t *b = malloc(sizeof(repl_block_t) + len);
6    b->id   = block_id++;  // NOT from user input
7    b->size = len; b->used = len;
8    for (size_t i = 0; i < len; i++)
9      b->buf[i] = s[i];
10   log(b->id, b->used, b->size);  // sink
11 }
```

The dataflow of a repl_block are:
id ← internal; size, used ← len; $\forall i \in [0, len)$. buf[i] ← s[i]

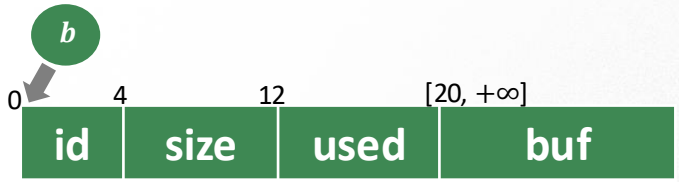
Expected:

DSA inferred:



Methodology

I-DSA: cell (field) offset is an interval



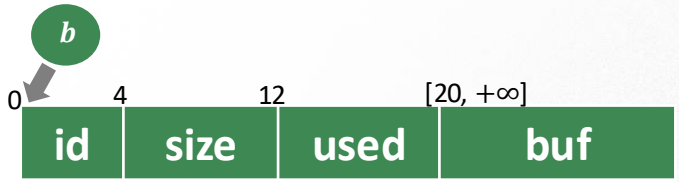

```
1  static int block_id = 0;
2  void feedReplicationBuffer(
3        const char *s, size_t len // sources
4  ){
5    repl_block_t *b = malloc(sizeof(repl_block_t) + len);
6    b->id   = block_id++;  // NOT from user input
7    b->size = len; b->used = len;
8    for (size_t i = 0; i < len; i++)
9      b->buf[i] = s[i];
10   log(b->id, b->used, b->size);  // sink
11 }
```

① **Interval-offset based DSA**

Specify cell (field) offset to be an interval

Methodology

I-DSA: cell (field) offset is an interval




```
1  static int block_id = 0;
2  void feedReplicationBuffer(
3        const char *s, size_t len // sources
4  ){
5    repl_block_t *b = malloc(sizeof(repl_block_t) + len);
6    b->id   = block_id++; // NOT from user input
7    b->size = len; b->used = len;
8    for (size_t i = 0; i < len; i++)
9      b->buf[i] = s[i];
10   log(b->id, b->used, b->size); // sink
11 }
```

① **Interval-offset based DSA**

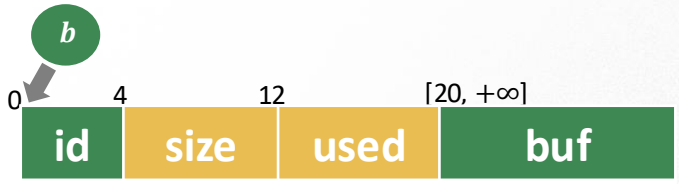
Specify cell (field) offset to be an interval

② **Taint Domain**

Track taint over a set of abstract locations (variables and DSA cells)

Methodology

I-DSA: cell (field) offset is an interval




```
1   static int block_id = 0;
2   void feedReplicationBuffer(
3         const char *s, size_t len // sources
4   ){
5     repl_block_t *b = malloc(sizeof(repl_block_t) + len);
6     b->id   = block_id++;  // NOT from user input
7     b->size = len; b->used = len;
8     for (size_t i = 0; i < len; i++)
9       b->buf[i] = s[i];
10    log(b->id, b->used, b->size);  // sink
11  }
```

① **Interval-offset based DSA**

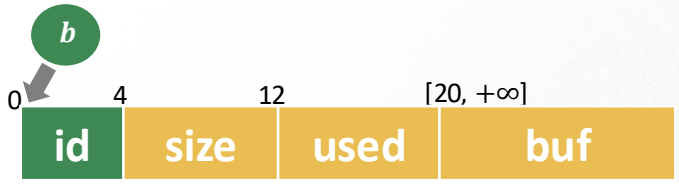
Specify cell (field) offset to be an interval

② **Taint Domain**

Track taint over a set of abstract locations (variables and DSA cells)

Methodology

I-DSA: cell (field) offset is an interval




```
1   static int block_id = 0;
2   void feedReplicationBuffer(
3         const char *s, size_t len // sources
4   ){
5     repl_block_t *b = malloc(sizeof(repl_block_t) + len);
6     b->id   = block_id++; // NOT from user input
7     b->size = len; b->used = len;
8     for (size_t i = 0; i < len; i++)
9       b->buf[i] = s[i];
10    log(b->id, b->used, b->size); // sink
11  }
```

① **Interval-offset based DSA**

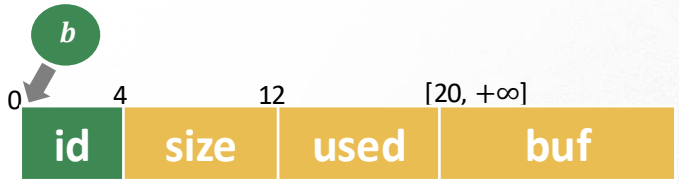
Specify cell (field) offset to be an interval

② **Taint Domain**

Track taint over a set of abstract locations (variables and DSA cells)

Methodology

I-DSA: cell (field) offset is an interval




```
1  static int block_id = 0;
2  void feedReplicationBuffer(
3        const char *s, size_t len // sources
4  ){
5    repl_block_t *b = malloc(sizeof(repl_block_t) + len);
6    b->id   = block_id++; // NOT from user input
7    b->size = len; b->used = len;
8    for (size_t i = 0; i < len; i++)
9      b->buf[i] = s[i];
10   log(b->id, b->used, b->size); // sink
11 }
```

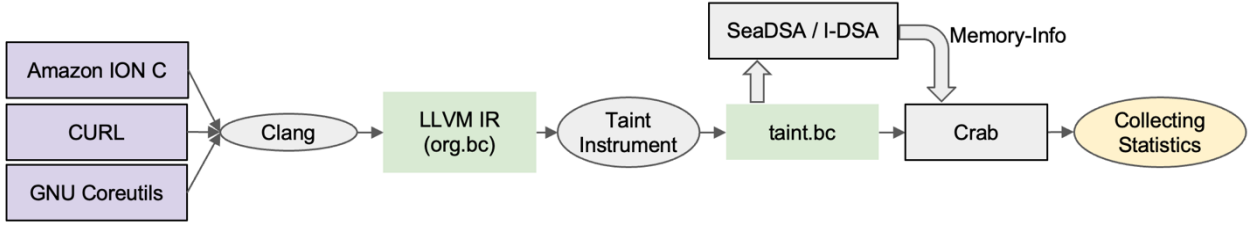
① **Interval-offset based DSA**

Specify cell (field) offset to be an interval

② **Taint Domain**

Track taint over a set of abstract locations (variables and DSA cells)

Avoid report id is tainted.



SEADSA vs. NEWDSA

Performance (No Inline)

DSA only

DSA	Avg.	Std.	Max.
SEADSA	0.06	0.136	0.860
I-DSA	0.06	0.134	0.870

Overall

Pipeline	Time (s)
SeaDSA + DFA	2402.1
I-DSA + DFA	2473.6

Scatter plot

Precision (No Inline)

Benchmark Suite	Assertions	Warnings		Reduced (%)
		SeaDSA + DFA	I-DSA + DFA	
IONC	107	43	41	4.7%
CURL	138	53	46	13.2%
COREUTILS	1753	181	169	6.6%

May-alias improvement example

Table

Conclusions

Contributions

Memory Safety:

- MRUD: new Recency-Use Abstraction for object invariants
- Template DBM: new numerical domain beyond UTVPI constraint, and near UTVPI domain const

Taint Analysis:

- A LLVM-IR based field-sensitive analysis
- A new DSA avoids field-sensitivity loss

Implementation:

- All domains and analyses are implemented and experimented

Future Work

MRUD:

- Extend Recency-Use abstraction with last k objects (k > 1)

Template DBM:

- Better join and inclusion test design and implementation

Taint Analysis:

- Sanitization rules
- Control-dependency

Data Structure Analysis:

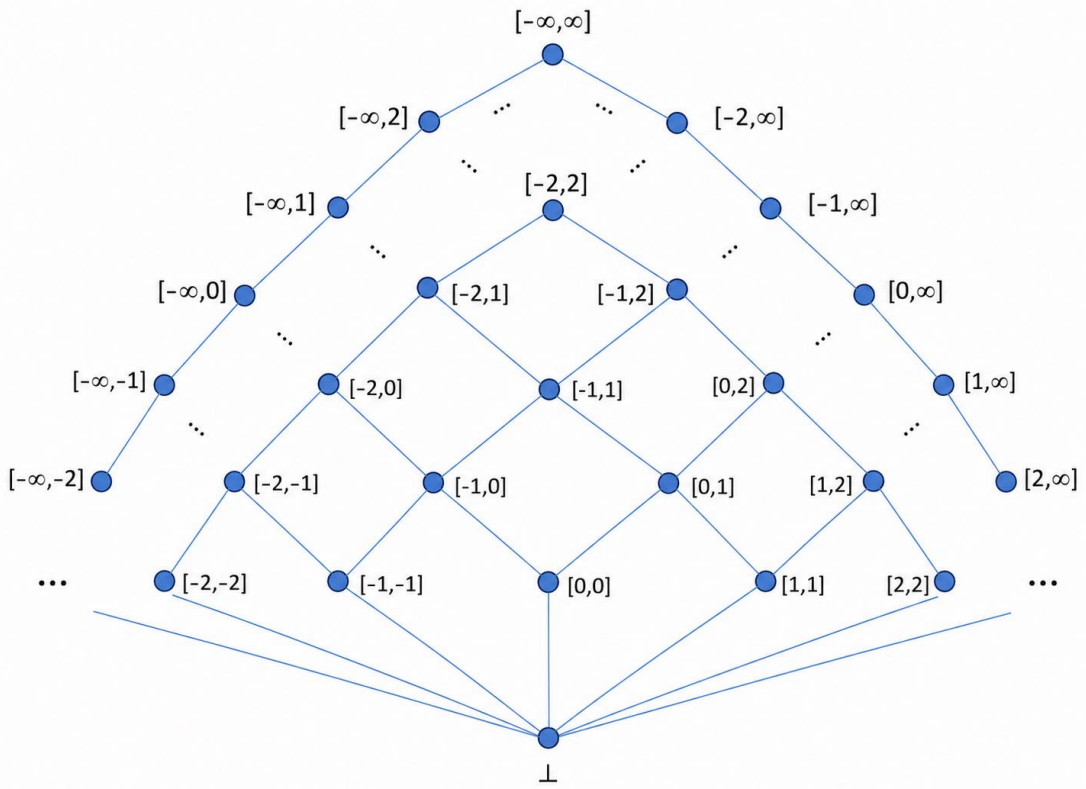
- Parameterized offset abstraction of DSA

Questions

The Interval Lattice

join (⊔):
smallest interval
containing both

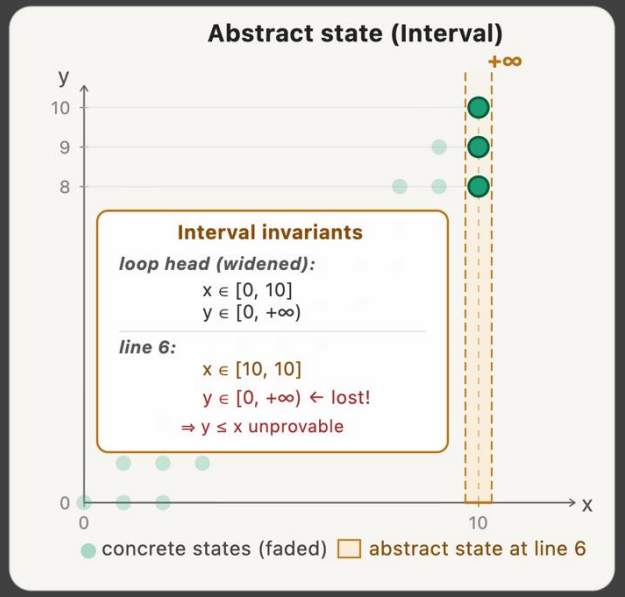
meet (⊓):
largest interval
contained in both

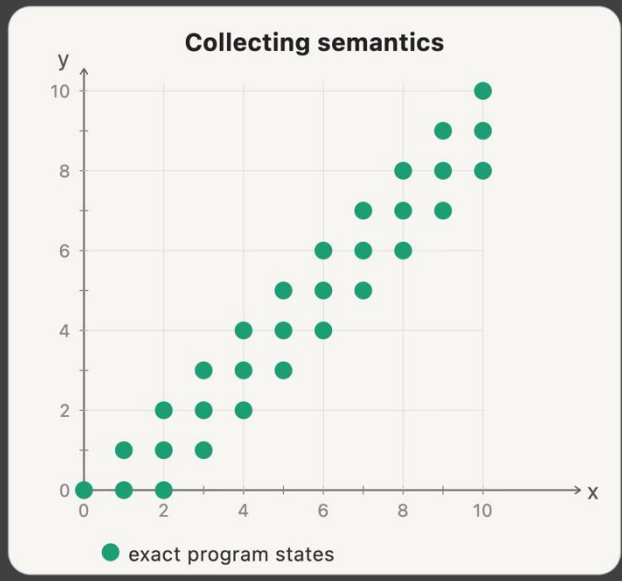


⊤ = [−∞, +∞]
(any value)

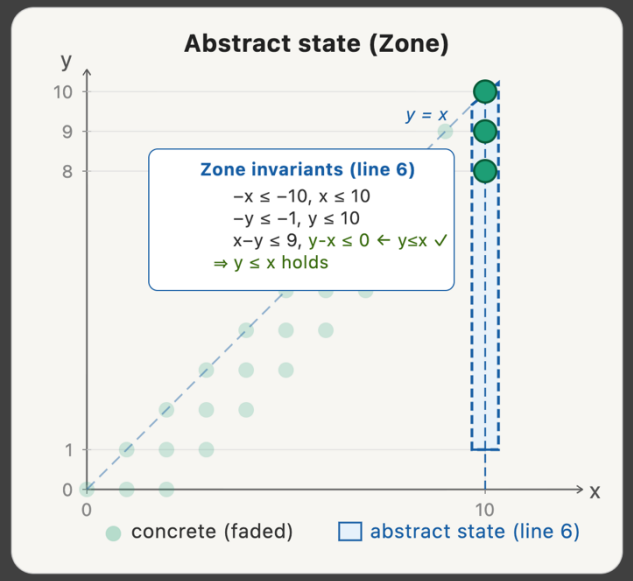
⊥ = ∅
(unreachable)

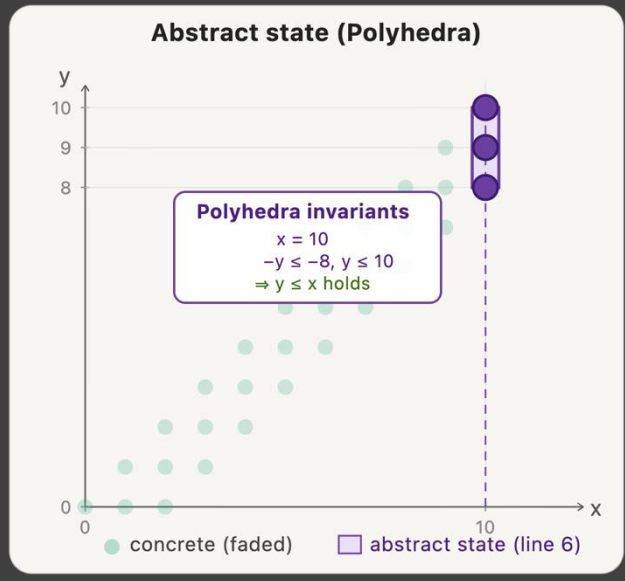
[a,b] ⊑ [c,d] iff c ≤ a and b ≤ d



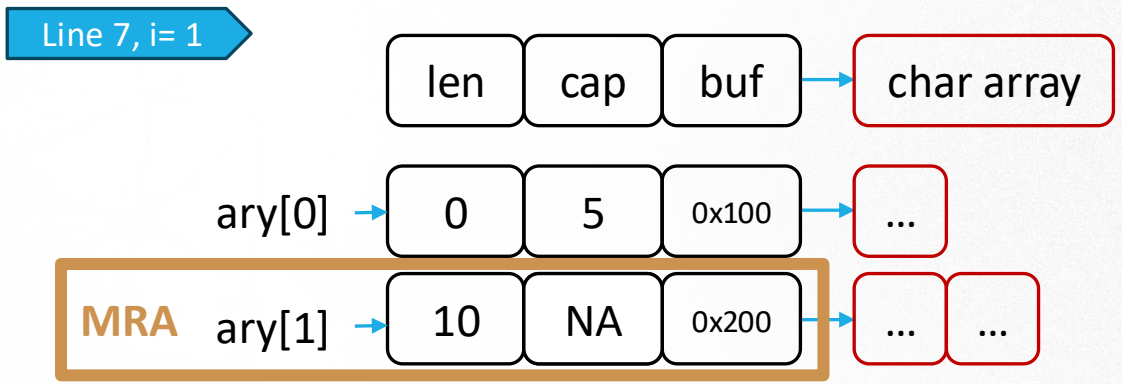



```
1    x = nd(0, 2); y = 0;
2    while (x < 10) {
3      x = x + 1;
4      y = y + 1;
5    }
6    assert(y <= x);
```





Precision restore




```
1   struct byte_buf { int len; int cap; char *buf; };

2   void main() {
3     struct byte_buf *ary[100];
4     for (int i = 0; i < 100; i++) {
5       ary[i] = malloc(sizeof(struct byte_buf));
6       int chunk = 10 * i;
7       ary[i]->len = chunk;
8       ary[i]->cap = chunk + 5;
9       ary[i]->buf = malloc(sizeof(char) * chunk);
10    }
11    ...
12  }
```

Strong update: write only for MRA object

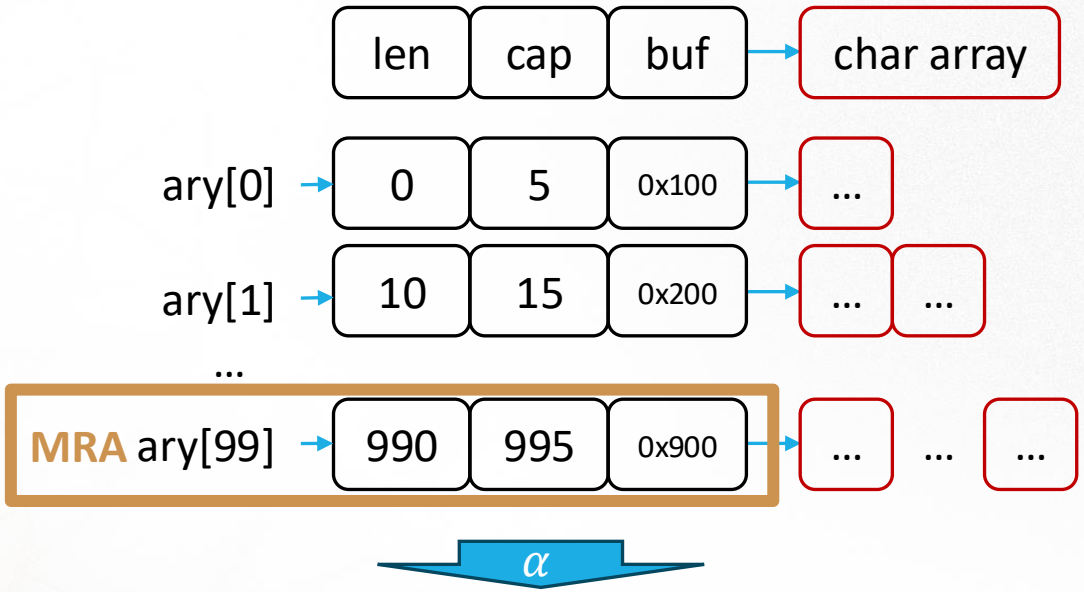
α

MRA:
$len = 10$

Summary Object:
$len = 0, cap = 5$

Recency Abstraction:
one abstract object to summarize OI for older objects
\+ MRA object stored OI for itself

Precision restore



MRA:
$len = 990, cap = 995$

Summary Object:
$len \in [0,980]$,
$cap = len + 5$

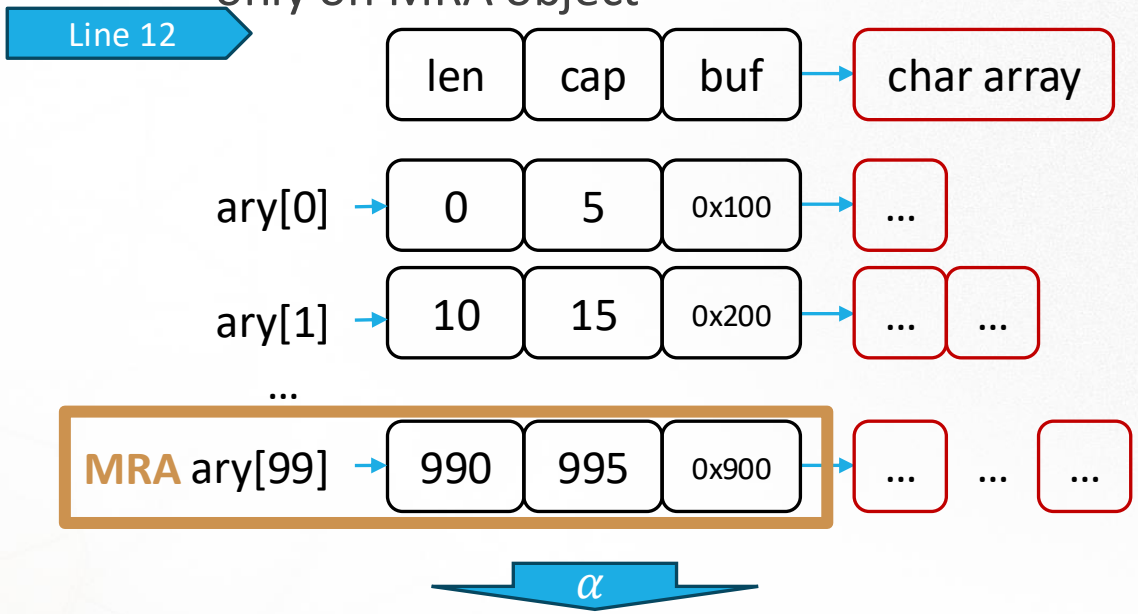
```
1   struct byte_buf { int len; int cap; char *buf; };

2   void main() {
3     struct byte_buf *ary[100];
4     for (int i = 0; i < 100; i++) {
5       ary[i] = malloc(sizeof(struct byte_buf));
6       int chunk = 10 * i;
7       ary[i]->len = chunk;
8       ary[i]->cap = chunk + 5;
9       ary[i]->buf = malloc(sizeof(char) * chunk);
10    }
11    ...
12  }
```

Strong update: write only for MRA object

Recency Abstraction:
one abstract object to summarize OI for older objects
\+ MRA object stored OI for itself

Motivation: precision still lost

Recency abstraction: weak updates avoided only on MRA object



MRA Object:
$len = 990, cap = 995$

Summary Object:
$len \in [0,980]$,
$cap \in [5,985]$

```
1   struct byte_buf { int len; int cap; char *buf; };

2   void main() {
3     struct byte_buf *ary[100];
4     for (int i = 0; i < 100; i++) {
5       ary[i] = malloc(sizeof(struct byte_buf));
6       int chunk = 10 * i;
7       ary[i]->len = chunk;
8       ary[i]->cap = chunk + 5;
9       ary[i]->buf = malloc(sizeof(char) * chunk);
10    }
11    char *new_buf = malloc(20);
12    ary[0]->len = 15;
13    ary[0]->cap = 20;
14    ary[0]->buf = new_buf;
15    assert(valid_access(p, l->isz));
16    ary[0]->buf[ary[0]->len] = '\0';
17  }
```

Weak update: still occurs

After Initialization

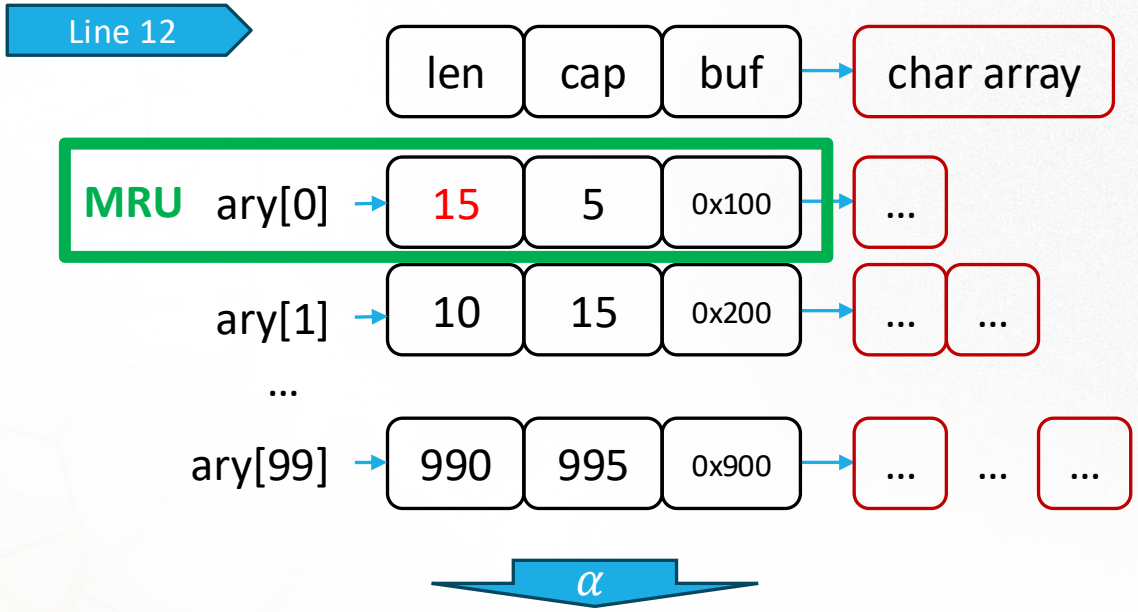
Recency-Use abstraction: weak updates avoided on every accessed object (MRU)

Observations:

- During loop initialization, the lasted allocated object is MRU
- At Line 12, MRU is the first allocated object

```
1   struct byte_buf { int len; int cap; char *buf; };

2   void main() {
3     struct byte_buf *ary[100];
4     for (int i = 0; i < 100; i++) {
5       ary[i] = malloc(sizeof(struct byte_buf));
6       int chunk = 10 * i;
7       ary[i]->len = chunk;
8       ary[i]->cap = chunk + 5;
9       ary[i]->buf = malloc(sizeof(char) * chunk);
10    }
11    char *new_buf = malloc(20);
12    ary[0]->len = 15;
13    ary[0]->cap = 20;
14    ary[0]->buf = new_buf;
15    ary[0]->buf[ary[0]->len] = '\0';
16  }
```

After Initialization



MRU Object:
$len = 15, cap \in [5,995]$

Summary Object:
$len \in [0,990], cap = len + 5$

```
1   struct byte_buf { int len; int cap; char *buf; };

2   void main() {
3     struct byte_buf *ary[100];
4     for (int i = 0; i < 100; i++) {
5       ary[i] = malloc(sizeof(struct byte_buf));
6       int chunk = 10 * i;
7       ary[i]->len = chunk;
8       ary[i]->cap = chunk + 5;
9       ary[i]->buf = malloc(sizeof(char) * chunk);
10    }
11    char *new_buf = malloc(20);
12    ary[0]->len = 15;
13    ary[0]->cap = 20;
14    ary[0]->buf = new_buf;
15    ary[0]->buf[ary[0]->len] = '\0';
16  }
```

Strong update: write only for the most-recently-used (MRU) object

[VSTTE 21]: Arie Gurfinkel, Jorge A. Navas: Abstract Interpretation of LLVM with a Region-Based Memory Model.

D_S = VSTTE'21

Experiment I - Scalability

MRUD (D_O) vs. allocation site abstraction domain (D_S) (i.e., VSTTE'21)

Goal: show that MRUD (as a product domain) is more lightweight

Benchmarks: 5 used for D_S, and 109 from GNU Coreutils

Numerical Domain: Zones

Timeout threshold: 5,000 seconds

Results: D_O outperforms D_S on nearly every benchmark

Metrics	D_O (NONE)	D_O (OPT)	D_O (FULL)	D_S
Timeout cases	6	6	6	8
Speedup over D_S	81x	76x	57x	-

Key Observations

- In D_S, storing all properties in one domain value makes the join operation inefficient, while in D_O, most memory banks only have simple properties, reducing the cost of joins
- D_O uses structural sharing, allowing abstract states to share unchanged memory banks

D_O scales better than D_S.

Relational Object Invariants (2)

Not all OIs can be represented by Zones, for example:

```
struct array_list {
  int size; int len;
  int isz; // item size
  void *data;
};
```

The invariants of a non-empty integer array_list are:

$$isz = 4 \wedge 0 \leq len \ \wedge 4 * len \leq size \ \wedge data.size = size$$

Memory Safety Property: Access `data[i]` must satisfy $0 \leq 4*i < 4*size$

Access Index i is guaranteed by OI: $4*i < 4*len \leq size$ ensures no OOB access

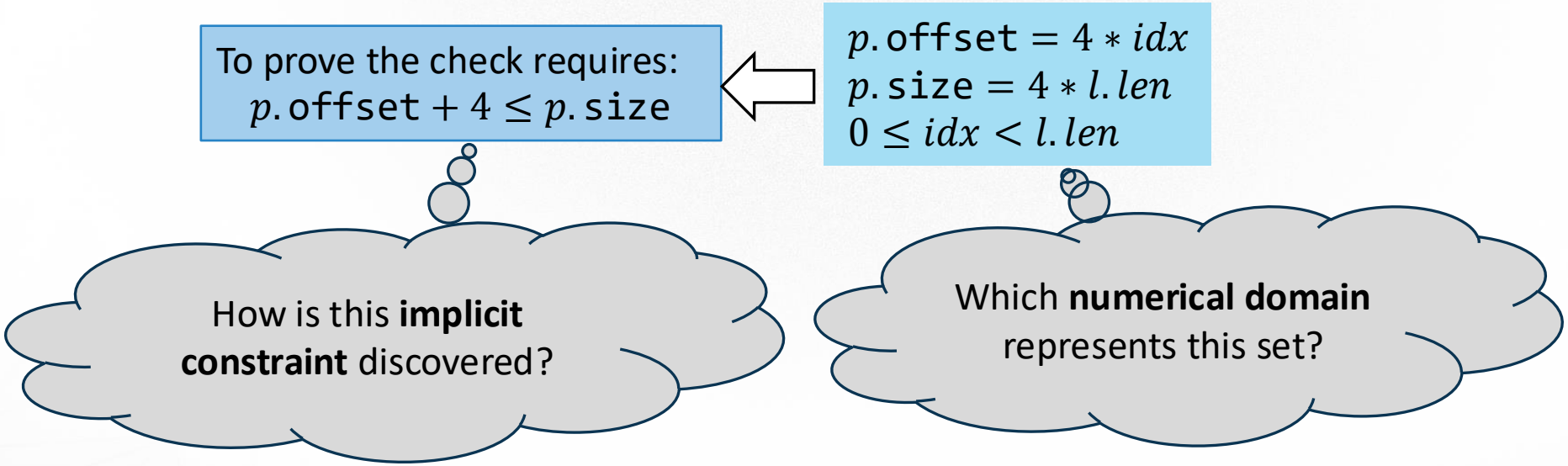
```
1  void main(int len, int idx) {
2   if (0 <= idx && idx < len) {
3    struct array_list l;
4    l.len = len;
5    l.isz = 4; // 4 bytes
6    l.size = l.isz * l.len;
7    l.data = malloc(l.size);
8    int ofs = l.isz * idx;
9    void *p = (uint8_t *)l.data + ofs;
10   assert(valid_access(p, l.isz));
11  }
12 }
```

To prove the check requires:
$p.\texttt{offset} + 4 \leq p.\texttt{size}$

$p.\texttt{offset} = 4 * idx$
$p.\texttt{size} = 4 * l.len$
$0 \leq idx < l.len$

Relational Object Invariants (2)

Not all OIs can be represented by Zones, for example:



Template DBM Abstract Domain

Given:

- A set of variables $\mathcal{V} = \{x, y, z, \ldots\} \cup \{v^0\}$, and some constant $c \in \mathbb{Z} \cup \{+\infty\}$
- A coefficient template $T: \{a, b, \ldots\}$, where $a, b \in \mathbb{Z}^{\geq 1}$

Template DBM expresses TVPI constraints:

$$\pm ax \leq c \mid a \cdot x - b \cdot y \leq c$$

Saturation:

- Apply previous steps back and forth until an upper bound iterations: $\lceil \lg(n) \rceil - 1$

Other domain operations:

$$\textbf{Join: } \sqcup^{tDBM} \overset{\text{def}}{=} \bar{m}_1 \sqcup^{DBM} \bar{m}_2$$

$$\textbf{Meet: } \sqcap^{tDBM} \overset{\text{def}}{=} \bar{m}_1 \sqcap^{DBM} \bar{m}_2$$

$$\textbf{Order: } \sqsubseteq^{tDBM} \overset{\text{def}}{=} \bar{m}_1 \sqsubseteq^{DBM} \bar{m}_2$$

$$\textbf{Widening: } \triangledown^{tDBM} \overset{\text{def}}{=} \bar{m}_1 \triangledown^{DBM} \bar{m}_2$$

DBM Limitations

Saturation may lose precision

E.g., $T = \{1,2,3,4\}$

$$f_1 : \ 2x - 3y \leq 1, \qquad f_2 : \ y - 4z \leq 1$$

$$\text{RESULTANT (scale } f_2 \text{ by 3)} : \quad 2x - 12z \leq 4 \implies x - 6z \leq 2.$$

DBM coefficient limit reached

Join: follow DBM join (elementwise join)

E.g., $T = \{1,10\}$

$s_1 : x \in [0,9], y \in [-10,-1] \quad \sqcup \quad s_2 : x = 10, y = 0$

$m^{s_1} =$

	v_0	x	y
v_0	0	0	10
x	9	0	$+\infty$
y	-1	$+\infty$	0

$\sqcup$

$m^{s_2} =$

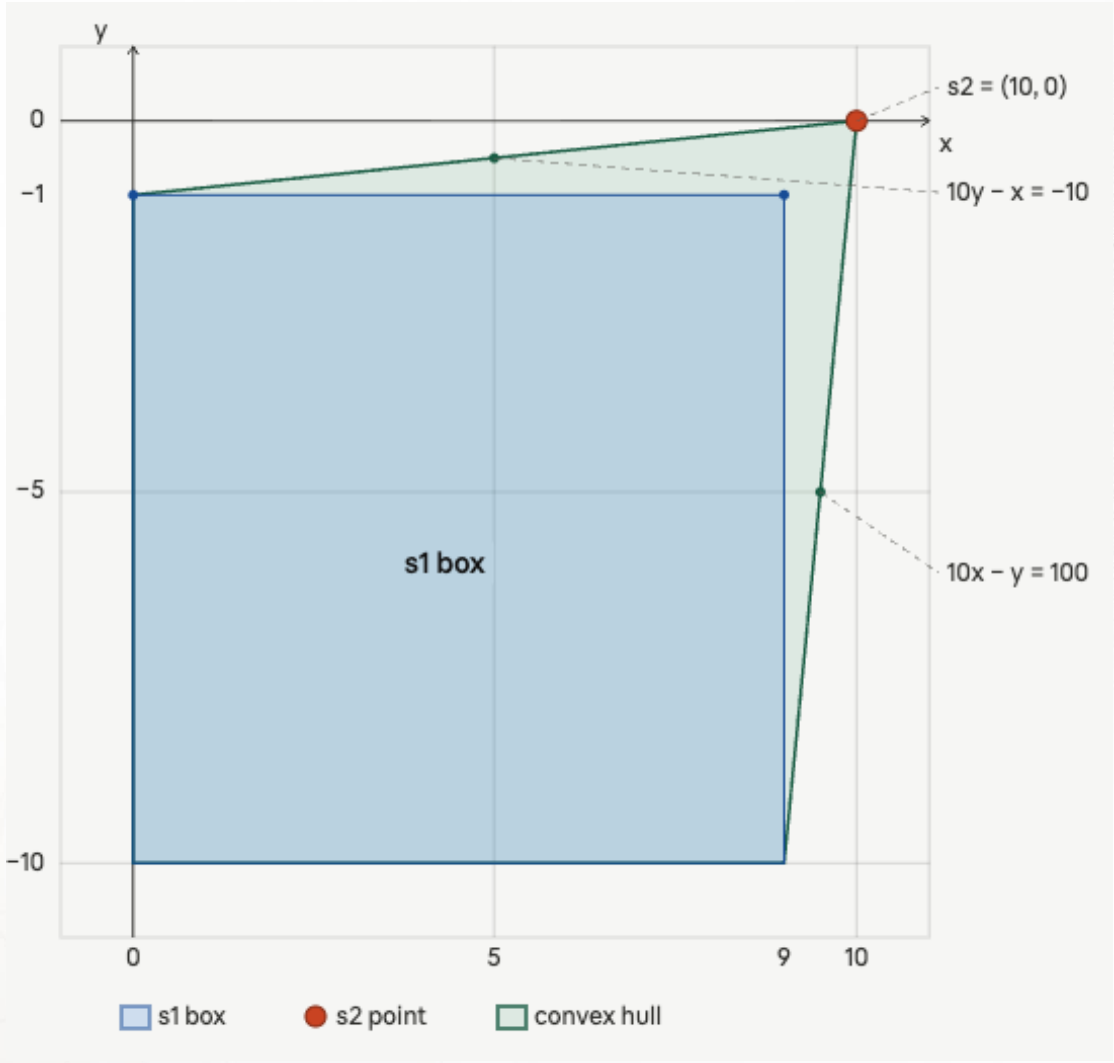
	v_0	x	y
v_0	0	-10	0
x	10	0	$+\infty$
y	0	$+\infty$	0

$s_{res} : x \in [0,10], y \in [-10,0]$

$=$

	v_0	x	y
v_0	0	0	10
x	10	0	$+\infty$
y	0	$+\infty$	0

DBM Limitations



Join: follow DBM join (elementwise join)

E.g., $T = \{1,10\}$

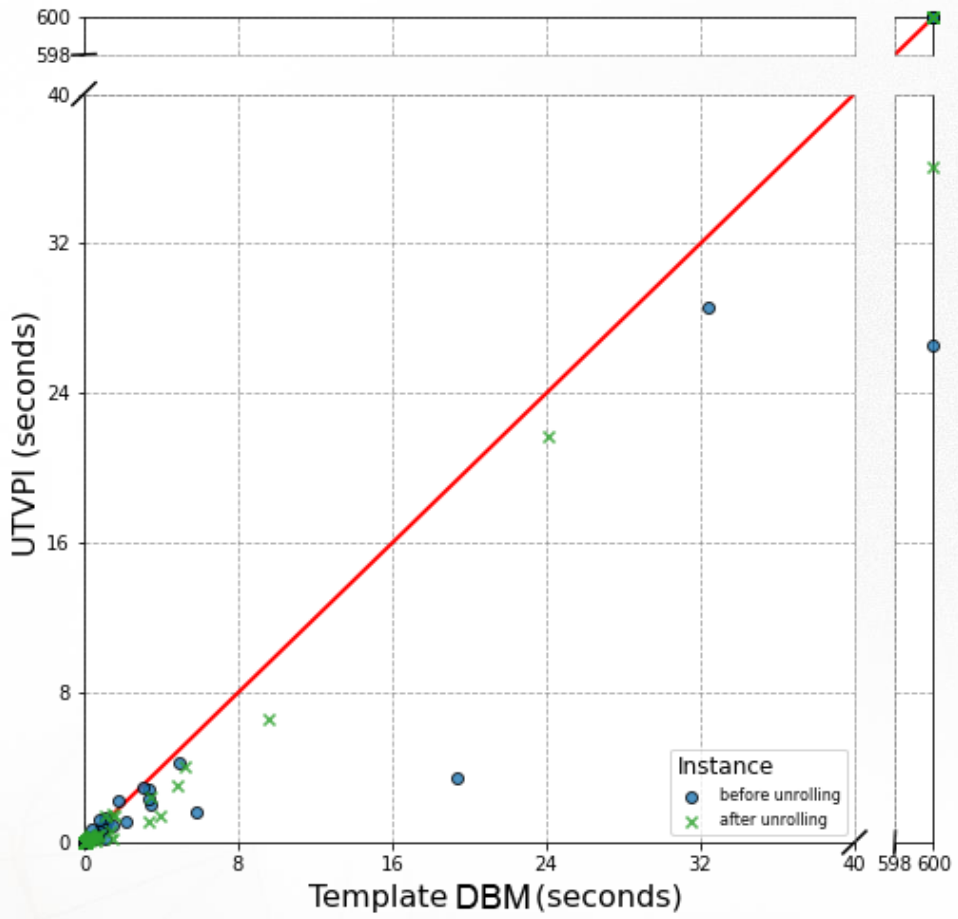
$s_1: x \in [0,9], y \in [-10,-1]$ ⊔ $s_2: x = 10, y = 0$

$$m^{s_1} = \begin{array}{c|ccc} & v_0 & x & y \\ \hline v_0 & 0 & 0 & 10 \\ x & 9 & 0 & +\infty \\ y & -1 & +\infty & 0 \end{array} \quad \sqcup \quad m^{s_2} = \begin{array}{c|ccc} & v_0 & x & y \\ \hline v_0 & 0 & -10 & 0 \\ x & 10 & 0 & +\infty \\ y & 0 & +\infty & 0 \end{array}$$

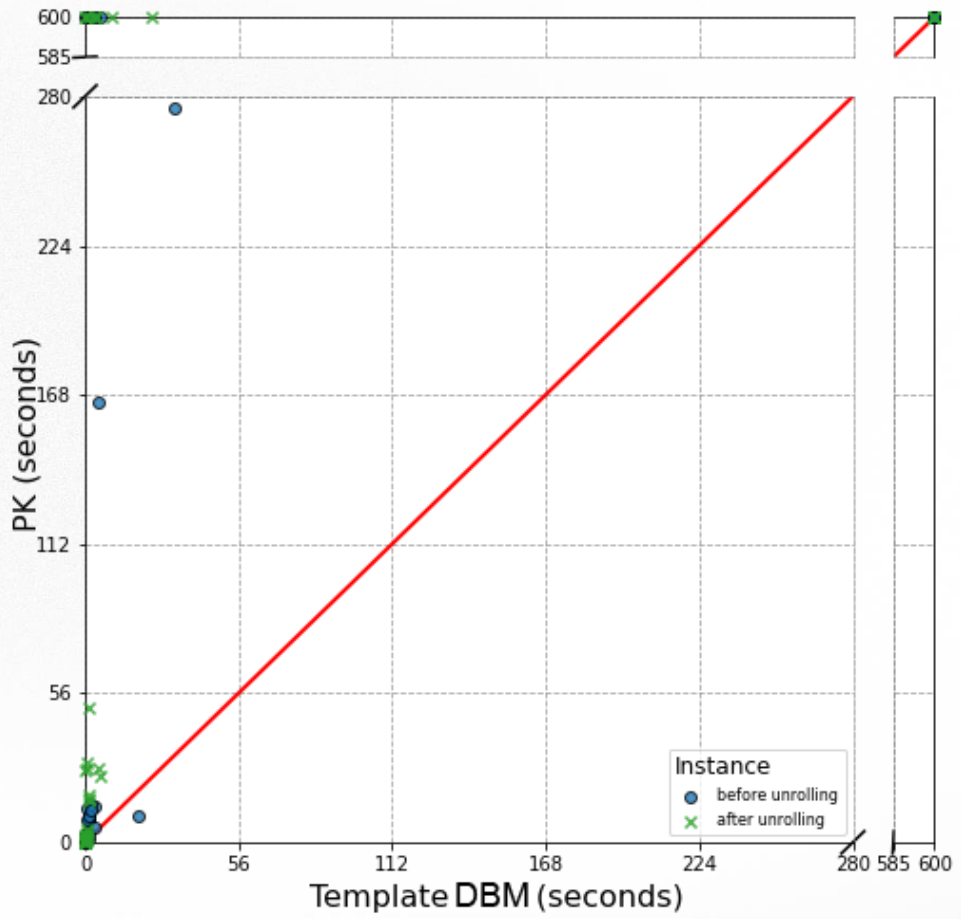
$s_{res}: x \in [0,10], y \in [-10,0]$

$$= \begin{array}{c|ccc} & v_0 & x & y \\ \hline v_0 & 0 & 0 & 10 \\ x & 10 & 0 & +\infty \\ y & 0 & +\infty & 0 \end{array}$$

Performance (Template DBM)



Running time per task close to Zones (UTVPI) on average.



Template DBM has less timeout than Polyhedra (PK)

Precision (Template DBM)

Template DBM proves more checks than Zones.

The motivating example from here.

suite	category	Before loop unroll				After loop unroll			
		Total	Zones	tDBM	PK	Total	Zones	tDBM	PK
aws-c-common	array_list	24	75%	83%	83%	62	61%	65%	65%
	hash_table	498	83%	85%	85%	2171	54%	58%	58%
	others	1689	67%	67%	67%	2853	56%	59%	59%
firedancer	tango	33	36%	85%	100%	151	17%	85%	100%
	util	106	62%	62%	62%	195	75%	75%	87%
	others	270	24%	24%	11%	305	95%	95%	86%
	total	2620	65%	66%	65%	5737	57%	62%	62%

Table 1: Precision across Zones, Template DBM (tDBM), and Polyhedra (PK).

PK join **overflows on coefficients**.

Data Structure Analysis (DSA)

Memory objects $\xrightarrow{\alpha}$ Nodes $\Leftrightarrow \{n \mid \alpha(objs)\}$

Object fields $\xrightarrow{\alpha}$ Cells $\Leftrightarrow \{c \mid c \triangleq \langle n, o \rangle, o \in \mathbb{N}\}$

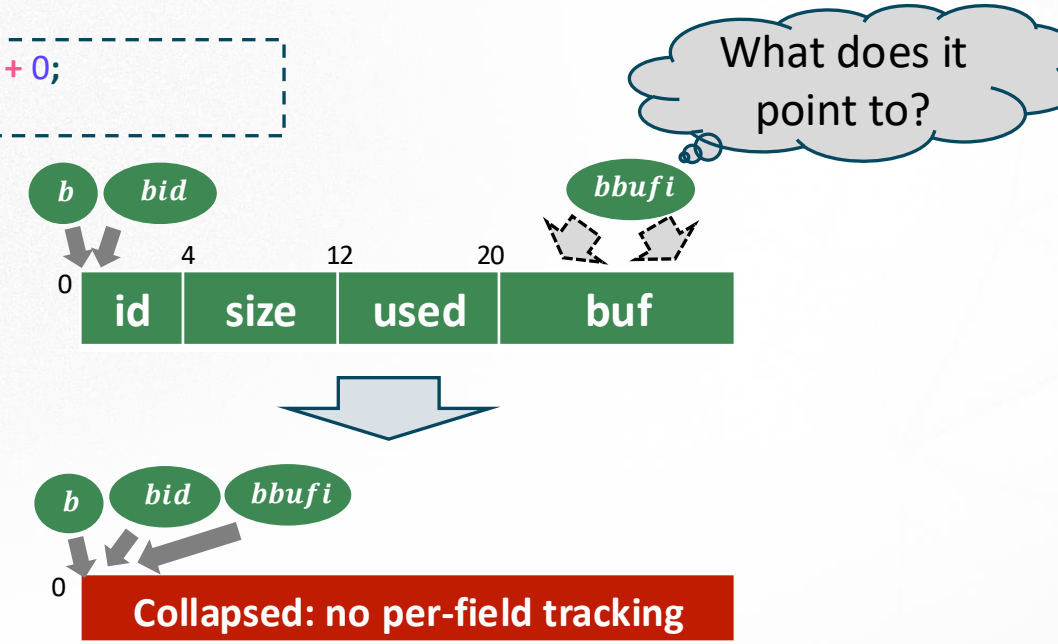
Pointer references $\xrightarrow{\alpha}$ Edges $\Leftrightarrow \{e \mid e \triangleq p/c \rightarrow c\}$

```
typedef struct repl_block {
  int id;
  size_t size;
  size_t used;
  char buf[]; // flexible array member
} repl_block_t;
```

```
1  static int block_id = 0;
2  void feedReplicationBuffer(
3        const char *s, size_t len // sources
4  ){
5    repl_block_t *b = malloc(sizeof(repl_block_t) + len);
6    b->id   = block_id++;  // NOT from user input
7    b->size = len; b->used = len;
8    for (size_t i = 0; i < len; i++)
9      b->buf[i] = s[i];
10   log(b->id, b->used, b->size);  // sink
11 }
```

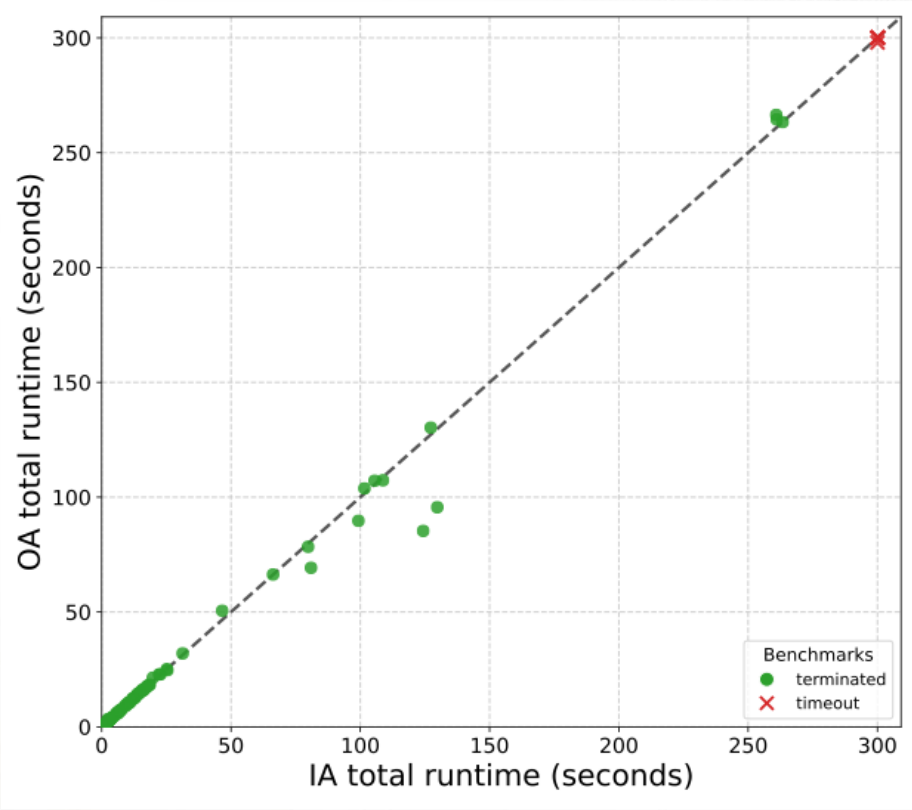
```
char *bid = (char *)b + 0;
...
```

```
char *bbufi = (char *)b + 20 + i;
const char *si = (const char *) s + i;
*bbufi = *si;
```

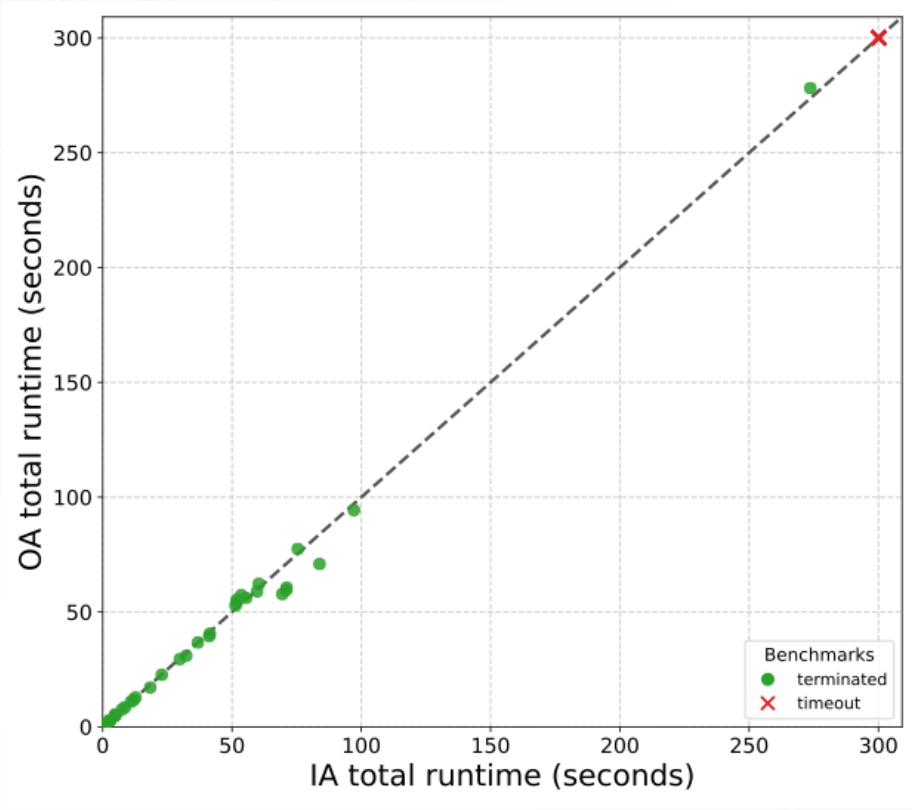


Performance (OA vs. IA)

NO-INLINING



INLINING



Precision (OA vs. IA)

Suite	#Bench.	T/O		#Succ.	#Assert.	Warnings		Red.	Time (s)	
		OA	IA	Bench.		OA	IA		OA	IA
Experiment 1: Not inlined										
SMALL	32	0	0	32	83	39	39	0	2.5	2.4
IONC	25	2	2	23	107	43	41	4.7%	401.1	486.9
CURL	17	0	0	17	138	53	46	13.2%	4.2	4.2
COREUTILS	109	2	3	106	1753	181	169	6.6%	1994.3	1980.1
Subtotal	183	4	5	178	2081	316	295	6.6%	2402.1	2473.6
Experiment 2: Inlined										
SMALL	32	0	0	32	87	41	41	0	2.1	2.0
IONC	25	7	7	18	1970	66	65	1.5%	248.7	237.4
CURL	17	0	0	17	254	84	77	8.3%	3.0	3.3
COREUTILS	109	15	15	94	6689	138	120	13.0%	1178.2	1224.5
Subtotal	183	22	22	161	9000	329	303	7.9%	1432.1	1467.2

Fig. 11: Comparison of SeaDSA + DFA (OA) and I-DSA + DFA (IA).

All these suites have the flexible array we showed.

I-DSA avoids precision loss

Another I-DSA precision improved

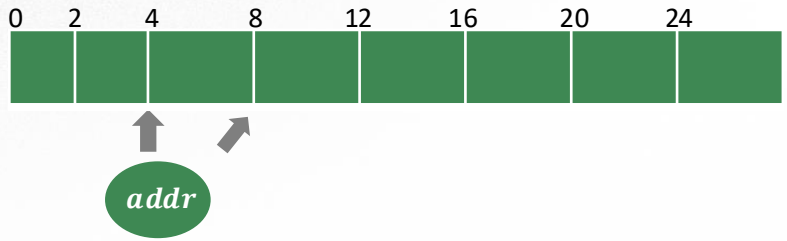
```
1  struct in_addr { uint32_t s_addr; };
2  struct in6_addr { uint32_t s6_addr[4]; };
3  struct ifaddrs{struct sockaddr *ifa_addr;};
4
5  struct sockaddr_in {
6    uint16_t sin_family, sin_port;
7    struct in_addr sin_addr;
8    uint8_t sin_zero[8];
9  };
10 struct sockaddr_in6 {
11   uint16_t sin6_family, sin6_port;
12   uint32_t sin6_flowinfo;
13   struct in6_addr sin6_addr;
14   uint32_t sin6_scope_id;
15 };
```

```
16 int if2ip(int af, char *interf, char *buf) {
17   struct ifaddrs *iface;
18   ...
19   if (getifaddrs(&iface) >= 0) {
20     void *addr;
21     if (af == AF_INET6)
22       addr = &((struct sockaddr_in6 *)
23                iface->ifa_addr)->sin6_addr;
24     else
25       addr = &((struct sockaddr_in *)
26                iface->ifa_addr)->sin_addr;
27     ...
28   }
29 }
```

As DSA (path-insensitive), `addr` may have different points-to offsets

DSA only allows one pointer to refer to one cell (field)

- Unification on I-DSA avoids fully collapsed node



Another I-DSA precision improved

```
1  struct in_addr { uint32_t s_addr; };
2  struct in6_addr { uint32_t s6_addr[4]; };
3  struct ifaddrs{struct sockaddr *ifa_addr;};
4
5  struct sockaddr_in {
6    uint16_t sin_family, sin_port;
7    struct in_addr sin_addr;
8    uint8_t sin_zero[8];
9  };
10 struct sockaddr_in6 {
11   uint16_t sin6_family, sin6_port;
12   uint32_t sin6_flowinfo;
13   struct in6_addr sin6_addr;
14   uint32_t sin6_scope_id;
15 };
```

```
16 int if2ip(int af, char *interf, char *buf) {
17   struct ifaddrs *iface;
18   ...
19   if (getifaddrs(&iface) >= 0) {
20     void *addr;
21     if (af == AF_INET6)
22       addr = &((struct sockaddr_in6 *)
23                iface->ifa_addr)->sin6_addr;
24     else
25       addr = &((struct sockaddr_in *)
26                iface->ifa_addr)->sin_addr;
27     ...
28   }
29 }
```

As DSA (path-insensitive), `addr` may have different points-to offsets

DSA only allows one pointer to refer to one cell (field)

- Unification on I-DSA avoids fully collapsed node