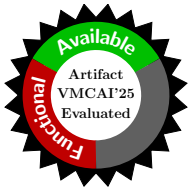


Automatic Inference of Relational Object Invariants

Yusen Su[1][0009−0004−8813−0797], Jorge A. Navas[2][0000−0002−0516−1167], Arie Gurfinkel[1][0000−0002−5964−6792], and Isabel Garcia-Contreras[1,3][0000−0001−6098−3895]

[1] Department of Electrical and Computer Engineering, University of Waterloo
{y256su,arie.gurfinkel,igarciac}@uwaterloo.ca
[2] Certora Inc.
navasjorgea@gmail.com
[3] Black Duck Software, Inc.

Abstract. Relational object invariants (or representation invariants) are relational properties held by the fields of a (memory) object throughout its lifetime. For example, the length of a buffer never exceeds its capacity. Automatic inference of these invariants is particularly challenging because they are often broken temporarily during field updates.
In this paper, we present an Abstract Interpretation-based solution to infer object invariants. Our key insight is a new object abstraction for memory objects, where memory is divided into multiple *memory banks*, each containing several objects. Within each bank, objects are abstracted by separating the *most recently used* (MRU) object, represented precisely with strong updates, while the rest are summarized. For an effective implementation of this approach, we introduce a new composite abstract domain, which forms a reduced product of numerical and equality subdomains. This design efficiently expresses relationships between a small number of variables (e.g., fields of the same abstract object).
We implement the new domain in the Crab abstract interpreter and evaluate it on several benchmarks for memory safety. We show that our approach is significantly more scalable for relational properties than the existing implementation of Crab. To evaluate precision, we have integrated our analysis as a pre-processing step to SeaBMC bounded model checker, and show that it is effective at both discharging assertions during pre-processing, and significantly improving the run-time of SeaBMC.

Keywords: Static Analysis · Abstract Interpretation · Object Invariants · Abstract Domains.

1 Introduction

Program invariants are crucial to capture properties that persist during runtime. Verifying programs with classes or data structures requires determining *representation invariants* [19] that express *consistency* properties (e.g., the length of a

```
1  #define N 100
2  struct byte_buf {
3      int len;
4      int cap;
5      char *buf;
6  };
7  int main() {
8      struct byte_buf *ary[N];
9      for (int i = 0; i < N; ++i) {
10         struct byte_buf *p =
               malloc(sizeof(struct byte_buf));
11         int sz = i + 1;
12         p->len = i; p->cap = sz;
13         p->buf = malloc(sz);
14         ary[i] = p;
15     }
16     char *new_buf = malloc(20);
17     ary[0]->len = 15;
18     ary[0]->cap = 20;
19     ary[0]->buf = new_buf;
20     assert(ary[0]->len <= ary[0]->cap);
21     ary[0]->buf[ary[0]->len] = '\0';
22 }
```

Fig. 1: A simple C program.

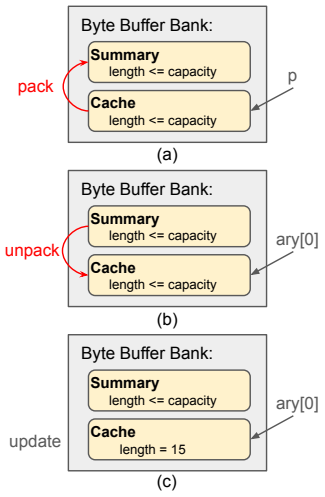


Fig. 2: Abstract memory state on line 17 of Fig. 1.

vector never exceeds its capacity) of those data types. For memory objects, representation invariants as *object invariants* describe relational properties among object fields that hold across all program states where these objects are alive. These invariants are essential for proving memory safety and functional correctness of a program. However, the invariants become imprecise when the static analyzer is uncertain about which memory objects are affected by field updates, typically represented as *weak* updates.

Consider a C program in Fig. 1 that uses a `byte_buf` to represent a resizable byte buffer with length and capacity. The program keeps an array `ary` of byte buffers. Each initialized element of `ary` satisfies an invariant: `len <= cap`. Discovering this invariant is crucial for establishing memory safety (e.g., proving safe access on line 21), yet, notoriously hard for abstract interpreters. Note that *recency* [1] does not help here because all memory stores after the `for` loop are modeled as weak updates. For instance, Mopsa [22] with recency does not prove the assertion on line 20, since the inferred invariant is $len > 0 \wedge cap > 1$.

In this paper, we present a new technique for inferring object invariants. We capture field updates *strongly* in a separate temporary object abstraction and join it with previously established invariants only when necessary. While preserving soundness, our approach produces more precise analysis results by not weakening inferred invariants with intermediate object states between updates.

First, we introduce a new concrete memory model that organizes memory as a collection of *memory banks*, each containing certain memory objects. The partitioning is achieved by a parameterized function that assigns each memory object in the program a corresponding bank. Each bank has two components: *storage*, holding objects, and *cache*, storing the object being read from or written to. For example, all byte buffers in Fig. 1 are placed into the storage of the same

bank. The field updates on line 12 require loading the byte buffer referred by pointer `p` into the cache before updates. The cache singles out the object being modified. For brevity, we specify this usage pattern with a size of one as *most recently used* (MRU) and denote the object in the cache as the MRU object.

Second, we follow a standard summarization-based abstraction with a single summary object with its invariants representing properties common to all the objects stored in each bank. Similar to the concrete model, all memory updates are handled through the MRU object. This avoids temporarily breaking the invariants of the (abstract) summary object, as changes to the MRU object do not impact the summarized invariants until it is merged back. Fig. 2 presents the changes in the abstract memory state at line 17. The memory bank for byte buffers includes one MRU object and one summary object. Before evaluating line 17, as shown in Fig. 2(a), `p` refers to the MRU object, since the last two field updates (line 12) happened on this object. Following the initialization loop, `len <= cap` is kept for both MRU and summary objects.

The cache may *miss* if the cached object is no longer the MRU. For example, the field update, `ary[0]->len = 15`, on line 17 requires access to the byte buffer referenced by `ary[0]`, while the cache still holds the object referred by `p`. In this case, the cached object is *packed* back to the summary (see Fig. 2(a)) and the new MRU object is *unpacked* from the summary (Fig. 2(b)). We track pointer alias information to decide when to pack and unpack. Before each memory access, if the dereferenced pointer does not alias with the pointer accessed to the MRU object, packing and unpacking occur. In this example, after the loop computation, `p` does not alias `ary[0]`.

After the cache is replaced, the field update, `ary[0]->len = 15`, breaks the invariant `len <= cap`, but our solution (Fig. 2(c)) ensures that we update the content of the MRU object properly without affecting the invariants in the summary object. Then, the invariant is restored at line 18, thus proving the assertion on line 20 and memory safety on line 21 through our invariants in the cache.

Third, we introduce a new abstract domain, called *MRUD*, that infers automatically object invariants based on our new memory model. This domain requires combining heap (memory abstraction), must alias (flow-sensitive points-to information) and value (numerical relational invariants) analyses. Using a monolithic numerical domain is highly inefficient because of the large number of dimensions required to model all program variables and their ghost versions that keep track of base addresses, offsets, etc. However, a key insight is that each transfer function typically affects a small subset of variables (e.g., reading a field only updates the corresponding integer/pointer value). Based on this observation, *MRUD* is designed as a composite abstract domain where each memory bank is modeled separately and the propagation of facts between them is carefully limited to a small set of shared variables. This modular design is what makes MRUD both scalable for large code bases and capable of preserving precise object invariants.

We implemented MRUD in the Crab analyzer [13] and evaluated both its scalability and precision. For scalability, we compare it to the summarization-

based abstract domain implemented in Crab. Our approach shows improved scalability, with 75X faster performance than the state-of-the-art. For precision, we compare it to the recency domain implemented on Mopsa using a small set of benchmarks. The results show that our approach successfully proves all assertions in the programs and achieves better precision by preserving object invariants. Additionally, we use MRUD in a case study with the bounded model checker SeaBMC, where it effectively proves and discharges memory safety checks to reduce the verification cost of SeaBMC.

In summary, the contributions of this paper are: (1) We introduce a new memory model designed for object abstraction as an alternative to the C memory model, and describe the concrete semantics of an intermediate representation based on the new model (Section 3); (2) We describe the MRUD and corresponding abstract transfer functions, and introduce a domain reduction for invariant refinement (Section 4); (3) We detail our implementation (Section 5) and evaluate it in the Crab analyzer (Section 6).

2 Preliminaries

Without loss of generality, we assume that the input program is in `CrabIR` [13] intermediate representation. The syntax of `CrabIR` is shown in Fig. 3. We assume that each memory object is a collection of integer and pointer fields. A pointer is a pair of a base address and an offset, where an offset is given by a number num and an optional field name fld. All named fields have fixed offsets. That is, field names are redundant – they are use to simplify the abstraction function in the abstract semantics. In our implementation, the field names are automatically discovered by a whole-program pointer analysis during compilation from the source language to `CrabIR`.

We write $\mathcal{V}$ for the set of all program variables. The set $\mathcal{V}$ is partitioned into: integers $\mathcal{V}_{int}$, pointers $\mathcal{V}_{ptr}$, and fields $\mathcal{V}_{fld}$. The union of $\mathcal{V}_{int}$ and $\mathcal{V}_{ptr}$ is called *scalars*. The statements in `CrabIR` consist of gotos, assumptions, assertions, and arithmetic and memory operations. All statements are strongly typed. Allocation of memory objects is performed by alloc (allocate). Pointer arithmetic is handled by the gep instruction that computes a destination address using the base pointer and an integer offset. Memory reads and writes are done by load and store, respectively. As usual, a program $\mathcal{P}$ is a control flow graph (CFG) whose basic blocks are annotated with statements from Fig. 3. `CrabIR` also supports C-like memory objects and it does not require them to be partitioned into fields. These are handled as in prior work [13]. We omit such objects in the theoretical exposition in the paper, but handle them as in [13] in our implementation.

We assume the reader is familiar with a standard numerical abstract domain that provides the following operations: join ($\sqcup$), meet ($\sqcap$), widen ($\triangledown$), projection ($\mathsf{project}(d, \mathcal{V})$) that projects an abstract value d to the variable set $\mathcal{V}$, forget ($\mathsf{forget}(d, v)$) that removes a variable v from an abstract value d, and constrain ($\mathsf{addCons}(d, c)$) that restricts an abstract value d by a linear constraint c.

$$
\begin{array}{lll}
\mathsf{P} ::= & \mathsf{F}^+ & \mathsf{S}_{\mathsf{ptr}} ::= \ \mathsf{ptr} := \mathsf{alloc}(\mathsf{fld}, \mathsf{num}) \mid \\
\mathsf{F} ::= & \mathtt{declare}\ \mathit{fun}(v^*)\{\ \mathsf{BB}^+\ \} & \quad\quad\quad\quad \mathsf{ptr2}, \mathsf{fld2} := \mathsf{gep}(\mathsf{ptr1}, \mathsf{fld1}, \mathsf{num}) \mid \\
\mathsf{BB} ::= & l : \mathsf{S}^*\ \mathtt{goto}\ l^+ \mid & \quad\quad\quad\quad \mathsf{scl} := \mathsf{load}(\mathsf{ptr}, \mathsf{fld}) \mid \mathsf{store}(\mathsf{ptr}, \mathsf{fld}, \mathsf{scl}) \\
& l : \mathsf{S}^*\ \mathtt{return}\ v^* & \mathsf{E}_{\mathsf{int}} ::= \ \mathsf{Const} \mid \mathsf{num} \mid \mathsf{E}_{\mathsf{int}}\ \mathsf{op}_{int}\ \mathsf{E}_{\mathsf{int}} \\
\mathsf{S} ::= & \mathtt{assert}(\mathsf{E}_{\mathsf{cond}}) \mid \mathtt{assume}(\mathsf{E}_{\mathsf{cond}}) \mid & \mathsf{E}_{\mathsf{cond}} ::= \mathsf{E}_{\mathsf{int}}\ \mathsf{op}_{cmp}\ \mathsf{E}_{\mathsf{int}} \\
& \mathsf{num} := \mathsf{E}_{\mathsf{int}} \mid \mathsf{S}_{\mathsf{ptr}} &
\end{array}
$$

Fig. 3: The syntax of `CrabIR`.

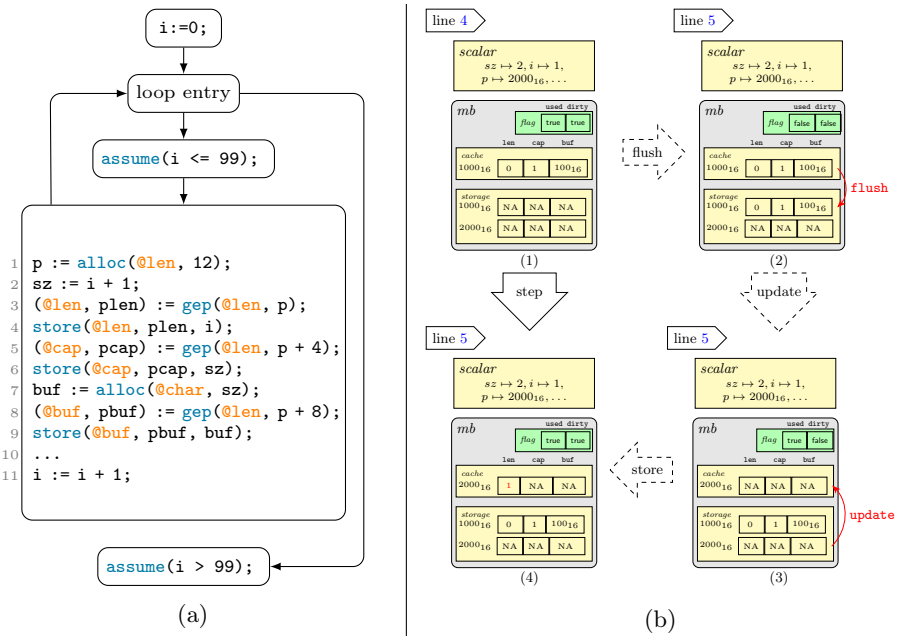


Fig. 4: (a) A program, and (b) an execution of line 4 under RUMM.

We use an equality domain over variable sets $\mathcal{V}$ to express equivalence relations such as $x \approx y$. The equality domain can be implemented using weakly relational numerical domains (e.g., [17,20,21]). We assume the equality domain has the following special operations: `addEqual` for adding an equality, `equals` for testing whether an abstract value entails an equality, and `toCons` for computing closed form of all implied equalities.

3 Recent-Use Memory Model

A memory model defines how memory is structured and accessed in the operational semantics (i.e., execution) of the program. The standard C memory model (CMM) treats each allocation as a blob of bytes. Specifically, each memory

object is a blob of bytes (logically sub-divided into fields). A pointer is a pair (b, o) of an object identifier b (a.k.a., the *base* address) and a numeric offset o within that object. At allocation, an object occupies a blob in memory at an address determined by the memory allocator. Each memory operation is performed through a pointer to access the object's content. In practice, CMM is typically implemented by a flat memory model of the underlying architecture. However, non-flat memory models with multiple address spaces are common, especially in embedded systems [14].

In this paper, we introduce a new memory model, called *recent-use memory model* (RUMM), that differentiates between the most recently used (MRU) object and other memory objects. RUMM partitions memory into multiple *banks*, each with (a) a *storage* – a blob of bytes that permanently stores memory objects, and (b) a *cache* – a blob of bytes that temporarily holds the MRU object of that bank. The notion of objects and pointers in RUMM is exactly as in CMM. Furthermore, RUMM is parameterized by a function findmb that maps allocation sites to specific memory banks of RUMM. This is similar to a pool allocation, where objects are allocated in different pools [18]. Each object is allocated as a blob in the selected bank's storage, with each bank managing its allocations.

What makes RUMM special is its handling of read and write operations. To access an object x from a given bank, x is first loaded into the cache and then accessed from there. If a different object y currently occupies the cache, y is flushed back to its place in its memory bank before x is loaded. Thus, multiple read and write operations that work on the same object only use the cache, until the cache is flushed when a new object, from the same bank, is accessed.

Fig. 4a shows a `CrabIR` for the `for` loop in Fig. 1. Variables prefixed with `@` are the fields of `byte_buf`. The loop starts at the *entry block* and checks whether the counter `i` meets the enter/exit condition. In `CrabIR`, `assume` is used to enforce this condition. The loop initializes a memory object, increments the counter, and loops back to the loop entry. Fig. 4b illustrates the execution of line 4 during the *second* iteration of the loop. Fig. 4b(1) shows the state at line 4, where scalar variables map to their values as *scalar* and a memory bank *mb* is provided to store memory objects allocated at line 1. We assume the first two iterations allocate objects at addresses 1000_{16} and 2000_{16}, respectively. The fields of each object are visually represented as slots, with either concrete values or marked as not available (NA) if uninitialized. The storage keeps two uninitialized objects, while the cache holds the MRU object. The object at address 1000_{16} is the MRU since its last access is at line 9 during the first iteration. The cache status is indicated by two flags: *used*, indicating the cache is active, and *dirty*, meaning the cache value has been updated. When `store` at line 4 accesses the object with address 2000_{16}, the cache flushes the object (1000_{16}) back to the storage (Fig. 4b(2)) and updates with the uninitialized object from the storage (Fig. 4b(3)). The cache is then ready to write `@len` with a value of 1 (Fig. 4b(4)).

We argue that RUMM is compatible with CMM. This follows from: (1) RUMM organizes memory objects into separate, non-overlapping memory banks; (2) The usage of cache is an extra step that does not invalidate the properties of

$[\![\mathsf{ptr} := \mathsf{alloc}(\mathsf{fld}, \mathsf{num})]\!]^{\mathrm{RUMM}}(\sigma) \equiv$
let $\langle scalar, mem\rangle = \sigma$ **in**
let $mb = \mathsf{findmb}(\mathsf{fld}, mem)$ **in**
let $\langle cache, storage, flag\rangle = mb$ **in**
let $\langle \_, sz\rangle = scalar[\mathsf{num}]$ **in**
let $\langle \mathsf{ptr}^{base}, storage'\rangle =$
$\mathtt{allocator}_{mb}(storage, sz)$ **in**
let $scalar' =$
$scalar[\mathsf{ptr} \mapsto \langle \mathsf{ptr}^{base}, 0\rangle]$ **in**
let $mb' = \langle cache, storage', flag\rangle$ **in**
$\langle scalar', mem \setminus \{mb\} \cup \{mb'\}\rangle$

$[\![\mathsf{scl} := \mathsf{load}(\mathsf{ptr}, \mathsf{fld})]\!]^{\mathrm{RUMM}}(\sigma) \equiv$
let $\langle scalar, mem\rangle = \sigma$ **in**
let $mb = \mathsf{findmb}(\mathsf{fld}, mem)$ **in**
let $\langle \mathsf{ptr}^{base}, \_\rangle = scalar[\mathsf{ptr}]$ **in**
let $mb' = \mathtt{cacheSync}(mb, \mathsf{ptr}^{base})$ **in**
let $\langle cache, \_, \_\rangle = mb'$ **in**
let $\langle \_, fields\rangle = cache$ **in**
let $scalar' =$
$scalar[\mathsf{scl} \mapsto fields[\mathsf{fld}]]$ **in**
$\langle scalar', mem \setminus \{mb\} \cup \{mb'\}\rangle$

$[\![\mathsf{ptr2}, \mathsf{fld2} := \mathsf{gep}(\mathsf{ptr1}, \mathsf{fld1}, \mathsf{num})]\!]^{\mathrm{RUMM}}(\sigma) \equiv$
let $\langle scalar, mem\rangle = \sigma$ **in**
let $\langle \mathsf{ptr1}^{base}, offset\rangle = scalar[\mathsf{ptr}]$ **in**
let $\langle \_, val\rangle = scalar[\mathsf{num}]$ **in**
let $offset' = offset + val$ **in**
let $scalar' =$
$scalar[\mathsf{ptr2} \mapsto \langle \mathsf{ptr1}^{base}, offset'\rangle]$ **in**
$\langle scalar', mem\rangle$

$[\![\mathsf{store}(\mathsf{ptr}, \mathsf{fld}, \mathsf{scl})]\!]^{\mathrm{RUMM}}(\sigma) \equiv$
let $\langle scalar, mem\rangle = \sigma$ **in**
let $mb = \mathsf{findmb}(\mathsf{fld}, mem)$ **in**
let $\langle \mathsf{ptr}^{base}, \_\rangle = scalar[\mathsf{ptr}]$ **in**
let $mb' = \mathtt{cacheSync}(mb, \mathsf{ptr}^{base})$ **in**
let $\langle cache, storage, \_\rangle = mb'$ **in**
let $\langle cache^{base}, fields\rangle = cache$ **in**
let $cache' = \langle cache^{base},$
$fields[\mathsf{fld} \mapsto scalar[\mathsf{scl}]]\rangle$ **in**
let $mb'' =$
$\langle cache', storage, \langle \mathsf{true}, \mathsf{true}\rangle\rangle$ **in**
$\langle scalar, mem \setminus \{mb\} \cup \{mb''\}\rangle$

Fig. 5: `CrabIR` statements operating under RUMM.

each object. The semantics of `CrabIR` are the same under both memory models. In the following, we formalize the concrete semantics of `CrabIR` under RUMM.

A `CrabIR` program has scalars (i.e., integers $\mathcal{V}_{int}$ and pointers $\mathcal{V}_{ptr}$) whose values are represented as *cells*. A cell, $cell \in \mathsf{Cell} : \mathbb{N} \times \mathbb{Z}$, represents either a pointer's base address and offset, denoted as $\langle baddr, offset\rangle$, or an integer value: $\langle 0, val\rangle$. Formally, a scalar state is $scalar \in \mathsf{Scalar} : \mathcal{V}_{scl} \mapsto \mathsf{Cell}$. To avoid redundancy, we explicitly associate the base address of a ptr with a ghost variable $\mathsf{ptr}^{base} \in \mathcal{V}_{ptr}^{base}$. For example, if a pointer `p` is $\langle 100_{16}, 8\rangle$, then $\mathsf{p}^{\mathsf{base}}$ is 100_{16}.

The memory is modeled as a set of memory banks, $mem \in \mathsf{Memory} : \{mb \mid mb \in \mathsf{MB}\}$. Each bank, $mb \in \mathsf{MB} : \mathsf{Cache} \times \mathsf{Storage} \times \mathsf{Flag}$, holds memory values for cache, storage, and boolean flags. The cache, $cache \in \mathsf{Cache} : \mathbb{N} \times \mathsf{FldVal}$, includes the cached object's base address (as $cache^{base}$) and field values. The field values (as cells) are kept in an environment $fields \in \mathsf{FldVal} : \mathcal{V}_{fld} \mapsto \mathsf{Cell}$. The storage, $storage \in \mathsf{Storage} : \mathbb{N} \mapsto \mathsf{FldVal}$, maps base addresses of memory objects to the corresponding field environment. The cache boolean flags, *flag*, indicate if it is occupied (*used*) and overwritten (*dirty*). Overall, a concrete program state $\sigma \in \mathsf{State}$ is a tuple: $\langle scalar, mem\rangle$. We assume findmb maps a field variable and memory state to a memory bank, indicating in which bank the field is stored.

Figs. 5 and 6 describe the changes to a program state at each memory and pointer arithmetic statements in `CrabIR`. The function $[\![\cdot]\!]^{\mathrm{RUMM}}(\cdot)$ takes a statement and a program state and returns the computed state under RUMM. The

```
cacheSync(mb, ptr^base) ≡
  let ⟨cache, storage, ⟨used, dirty⟩⟩ = mb in
  let ⟨cache^base, _⟩ = cache in
  let mb′ =
    if ¬used ∧ ptr^base ≠ cache^base then
      let storage′ = if dirty then
        flush(cache, storage) else storage in
      let cache′ = refresh(storage′, ptr^base) in
      let mb′ =
        ⟨cache′, storage′, ⟨true, false⟩⟩ in
      mb′
    else mb
  in mb′
```

```
flush(cache, storage) ≡
  let ⟨cache^base, fields⟩ =
      cache in
  storage[cache^base ↦ fields]

refresh(storage, ptr^base) ≡
  ⟨ptr^base, storage[ptr^base]⟩
```

Fig. 6: Cache operations.

initial state's *scalar* is an empty map. Each bank *mb* contains an empty *cache*, an empty map *storage*, and a ⟨false, false⟩ cache flags.

The alloc statement creates a new memory object of size num, assigns it to a specific bank's storage. The bank is determined by fld through findmb, and its `allocator` constructs the object and returns its base address assigned to ptr.

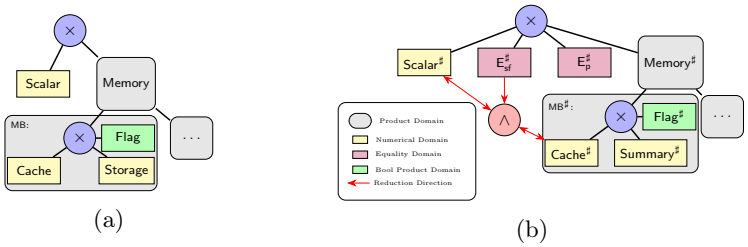
The gep computes a new pointer value for ptr2 by adding an offset num to the pointer value of ptr1. Earlier, we assume all pointer arithmetic stays inbounds, so the ptr2 and ptr1 have the same base address but (presumably) different offsets.

The load operation accesses the object pointed by ptr from the cache associated with the corresponding memory bank. To ensure the object is cached, we use the `cacheSync` function to check if the cache is missed. If so, we flush the cache back to the storage with `flush` if the cache is modified, and then load the new MRU object by calling `refresh`. The `flush` function moves the currently cached object into *storage*, while `refresh` refreshes the cache with the object pointed by ptr. After that, the object at ptr is in the cache, so the flag *used* is set to `true`. The value of scl in *scalar* gets updated by the cached field fld. Similarly, store updates the field for the object, using `cacheSync` to ensure it is in the cache. The flag *dirty* is set to `true`, indicating the object has been modified.

Overall, RUMM offers a different way to organize C memory by partitioning it into multiple banks, with additional space (i.e., the cache) to temporarily hold a memory object for reads and writes. This setup is very convenient for two reasons: first, it allows strong updates on the cache; second, it provides a straightforward memory abstraction by summarizing all objects from the same bank into one and simplifies the design of MRUD, as described in Section 4.

4 An Abstract Domain for Inferring Object Invariants

In this section, we introduce MRUD, a new abstract domain that is a (partially) reduced product of the domains for scalars, pointers, and objects. After setting up the domain, we detail key transfer functions and the reduction procedure.



(a) (b)

Fig. 7: (a) Concrete domain and (b) MRUD hierarchy.

$$
\begin{aligned}
scalar \in \mathsf{Scalar}^\sharp &\stackrel{\text{def}}{=} \mathsf{Num}(\mathcal{V}_{scl}) \\
e_{sf} \in \mathsf{E}^\sharp_{\mathsf{sf}} &\stackrel{\text{def}}{=} \mathsf{Eq}(\mathcal{V}_{scl} \cup \mathcal{V}_{fld}) \\
e_p \in \mathsf{E}^\sharp_{\mathsf{p}} &\stackrel{\text{def}}{=} \mathsf{Eq}(\mathcal{V}^{base}_{ptr}) \\
\langle used, dirty, ispk \rangle \in \mathsf{Flag}^\sharp &\stackrel{\text{def}}{=} \mathsf{Bool}^\sharp \times \mathsf{Bool}^\sharp \times \mathsf{Bool}^\sharp \\
cache \in \mathsf{Cache}^\sharp &\stackrel{\text{def}}{=} \mathsf{Num}(\mathcal{V}_{fld}) \\
sum \in \mathsf{Summary}^\sharp &\stackrel{\text{def}}{=} \mathsf{Num}(\mathcal{V}_{fld}) \\
mb \in \mathsf{MB}^\sharp &\stackrel{\text{def}}{=} \mathsf{Cache}^\sharp \times \mathsf{Summary}^\sharp \times \mathsf{Flag}^\sharp \\
mem \in \mathsf{Memory}^\sharp &\stackrel{\text{def}}{=} \prod_i \{\mathsf{MB}^\sharp_i\} \\
\sigma^\sharp \in \mathsf{State}^\sharp &\stackrel{\text{def}}{=} \mathsf{Scalar}^\sharp \times \mathsf{MB}^\sharp \times \mathsf{E}^\sharp_{\mathsf{p}} \times \mathsf{E}^\sharp_{\mathsf{sf}}
\end{aligned}
$$

Fig. 8: Abstract semantic domains.

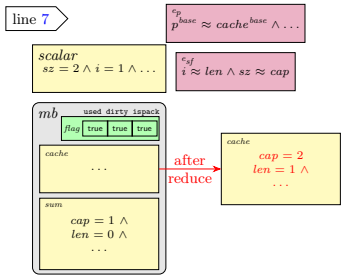


Fig. 9: State at line 7, 2nd iteration.

Similar to the concrete domain in Fig. 7a, the MRUD is shown in Fig. 7b. It is a reduced product of four domains: (a) a numerical domain $\mathsf{Scalar}^\sharp$, (b) an equality domain $\mathsf{E}^\sharp_{\mathsf{p}}$, (c) an equality domain $\mathsf{E}^\sharp_{\mathsf{sf}}$, and (d) a collection of product domains $\mathsf{Memory}^\sharp : \{\mathsf{MB}^\sharp\}$. $\mathsf{MB}^\sharp$ is a product of two numerical domains, and three Boolean domains: $\mathsf{Cache}^\sharp \times \mathsf{Summary}^\sharp \times \mathsf{Flag}^\sharp$. Fig. 8 shows the abstract semantic domains where variables are mapped to unique *dimensions* of each abstract domain. Most domains correspond to those in concrete semantics, except for a few that provide additional information. Specifically, $\mathsf{E}^\sharp_{\mathsf{sf}}$ represents the value equivalence of fields and scalars, which enables information propagation between $\mathsf{Scalar}^\sharp$ and $\mathsf{Cache}^\sharp$ for domain reduction. $\mathsf{E}^\sharp_{\mathsf{p}}$ captures the aliasing properties of pointers, indicating which pointer refers to which object. The added Boolean domain in $\mathsf{Flag}^\sharp$ is a flag for later use. All domains are parameterized by relational abstract domains like Zones [20]. An abstract state $\sigma^\sharp$ is represented by lattice elements within the MRUD.

Fig. 9 shows the abstract state at line 7 during the second iteration of the `CrabIR` example from Fig. 4a. We assume that the Zones domain is used for equality and numerical domains. We only show the invariants for scalars `i` and `sz`, and fields `len` and `cap`. *scalar* shows invariants for the scalars `i` and `sz`. The sole memory bank *mb* represents the objects of type `byte_buf`. The *cache* shows the invariants for the MRU `byte_buf` object referenced by pointer `p`. This follows from the equality $p^{base} \approx cache^{base}$ in e_p. The *cache* does not have any explicit invariants for fields. However, the fields invariants are *implicitly* represented through the invariants in *scalar* and the equalities in e_{sf}, $i \approx len$ and

$[\![\mathsf{ptr} := \mathsf{alloc}(\mathsf{fld}, \mathsf{num})]\!]^{\mathrm{RUMM}}(\sigma^\sharp) \equiv$
let $\langle scalar, e_{sf}, e_p, mem\rangle = \sigma^\sharp$ **in**
let $scalar' = \mathsf{forget}(scalar, \mathsf{ptr})$ **in**
let $scalar'' = \mathsf{addCons}($
$scalar', \mathsf{ptr} \neq 0)$ **in**
let ${e_p}' = \mathsf{forget}(e_p, \mathsf{ptr}^{base})$ **in**
$\langle scalar'', e_{sf}, {e_p}', mem\rangle$

$[\![\mathsf{store}(\mathsf{ptr}, \mathsf{fld}, \mathsf{scl})]\!]^{\mathrm{RUMM}}(\sigma^\sharp) \equiv$
let $\langle scalar, e_{sf}, e_p, mem\rangle = \sigma^\sharp$ **in**
let $mb = \mathsf{findmb}^\sharp(\mathsf{fld}, mem)$ **in**
let $\langle {e_p}', mb'\rangle = \mathtt{cacheSync}^\sharp($
$mb, e_p, \mathsf{ptr})$ **in**
let $\langle cache, sum, \langle \_, \_, ispk\rangle\rangle = mb'$ **in**
let $cache' = \mathsf{forget}(cache, \mathsf{fld})$ **in**
let ${e_{sf}}' = \mathsf{forget}(e_{sf}, \mathsf{fld})$ **in**
let ${e_{sf}}'' = \mathtt{addEqual}({e_{sf}}', \mathsf{scl}, \mathsf{fld})$ **in**
let $flag = \langle \mathsf{true}, \mathsf{true}, ispk\rangle$ **in**
let $mb'' = \langle cache', sum, flag\rangle$ **in**
$\langle scalar, {e_{sf}}'', {e_p}',$
$mem \setminus \{mb\} \cup \{mb''\}\rangle$

$[\![\mathsf{ptr2}, \mathsf{fld2} := \mathsf{gep}(\mathsf{ptr1}, \mathsf{fld1}, \mathsf{num})]\!]^{\mathrm{RUMM}}(\sigma^\sharp) \equiv$
let $\langle scalar, e_{sf}, e_p, mem\rangle = \sigma^\sharp$ **in**
let $scalar' = \mathsf{forget}(scalar, \mathsf{ptr2})$ **in**
let $scalar'' = \mathsf{addCons}($
$scalar, \mathsf{ptr2} = \mathsf{ptr1} + \mathsf{num})$ **in**
let ${e_p}' = \mathsf{forget}(e_p, \mathsf{ptr2}^{base})$ **in**
let ${e_p}'' = \mathtt{addEqual}($
${e_p}', \mathsf{ptr2}^{base}, \mathsf{ptr1}^{base})$ **in**
$\langle scalar'', e_{sf}, {e_p}'', mem\rangle$

$[\![\mathsf{scl} := \mathsf{load}(\mathsf{ptr}, \mathsf{fld})]\!]^{\mathrm{RUMM}}(\sigma^\sharp) \equiv$
let $\langle scalar, e_{sf}, e_p, mem\rangle = \sigma^\sharp$ **in**
let $mb = \mathsf{findmb}^\sharp(\mathsf{fld}, mem)$ **in**
let $\langle {e_p}', mb'\rangle = \mathtt{cacheSync}^\sharp($
$mb, e_p, \mathsf{ptr})$ **in**
let $scalar' = \mathsf{forget}(scalar, \mathsf{scl})$ **in**
let ${e_{sf}}' = \mathsf{forget}(e_{sf}, \mathsf{scl})$ **in**
let ${e_{sf}}'' = \mathtt{addEqual}({e_{sf}}', \mathsf{fld}, \mathsf{scl})$ **in**
$\langle scalar', {e_{sf}}'', {e_p}',$
$mem \setminus \{mb\} \cup \{mb'\}\rangle$

Fig. 10: Abstract transformers for memory operations.

$sz \approx cap$, that connect fields and scalars. These equalities are established during field writes. For instance, $i \approx len$ is there because instruction `store(@len, plen, i)` was used to update the field `len` with scalar `i`. Finally, *sum* shows the object invariants for the objects initialized at the first iteration. Specifically, the fields of that object satisfy `len <= cap`.

The most relevant transfer functions for inferring object invariants are shown in Fig. 10. For the initial state of analysis, we assign all subdomain elements with $\top$, except for *flag* in each memory bank as $\langle \mathsf{false}, \mathsf{false}, \mathsf{false}\rangle$. The third flag, *ispk*, is false to indicate the *sum* does not represent any concrete objects.

For alloc, the transformer assigns a ptr as not NULL in *scalar* indicating the valid address of the allocated object that ptr refers to. For gep, the transformer computes the address for ptr2 by addition in *scalar* and establishes an equivalence between ptr2 and ptr1 in e_p, denoting that the two pointers refer to the same memory object. For load/store, the transformer requires that the object referred by ptr is in the cache before it is accessed. The function $\mathtt{cacheSync}^\sharp$ in Fig. 11 checks for a cache miss and handles operations when a miss happens. It tests whether ptr refers to the cached object by comparing ptr^{base} with $cache^{base}$ in e_p. When the cache is missed, the function performs $\mathtt{pack}^\sharp$ and $\mathtt{unpack}^\sharp$. The $\mathtt{pack}^\sharp$ operation merges *cache* into *sum*. The invariants of the first cached object are copied to *sum* because, initially, *sum* does not represent any concrete objects. We change the flag *ispk* to true since the *sum* now holds the invariants for that object. Any subsequent packs use the join operation. The $\mathtt{unpack}^\sharp$ is achieved by copying the *sum* as the new *cache*. The $\mathtt{pack}^\sharp$ and $\mathtt{unpack}^\sharp$ operations are similar

$$
\begin{array}{l}
\texttt{cacheSync}^\sharp(mb, e_p, \mathsf{ptr}) \equiv \\
\quad \textbf{let } \langle cache, sum, \langle used, dirty, ispk\rangle\rangle = mb \textbf{ in} \\
\quad \textbf{let } \langle {e_p}', mb'\rangle = \\
\qquad \textbf{if } \neg used \wedge \neg \texttt{equals}(e_p, \mathsf{ptr}^{base}, cache^{base}) \textbf{ then} \\
\qquad\quad \textbf{let } sum', ispk' = \textbf{if } dirty \textbf{ then } \texttt{pack}^\sharp(cache, sum, ispk) \textbf{ else } sum, ispk \textbf{ in} \\
\qquad\quad \textbf{let } cache' = \texttt{unpack}^\sharp(sum') \textbf{ in} \\
\qquad\quad \textbf{let } {e_p}' = \mathsf{forget}(e_p, cache^{base}) \textbf{ in} \\
\qquad\quad \textbf{let } {e_p}'' = \texttt{addEqual}({e_p}', \mathsf{ptr}^{base}, cache^{base}) \textbf{ in} \\
\qquad\quad \langle {e_p}'', \langle cache', sum', \langle \mathsf{true}, \mathsf{false}, ispk'\rangle\rangle\rangle \\
\qquad \textbf{else } \langle e_p, mb\rangle \\
\quad \textbf{in } \langle {e_p}', mb'\rangle \\
\texttt{pack}^\sharp(cache, sum, ispk) \equiv \textbf{if } \neg ispk \textbf{ then } \langle \texttt{copy}(cache), \mathsf{true}\rangle \textbf{ else } \langle sum \sqcup cache, ispk\rangle \\
\texttt{unpack}^\sharp(sum) \equiv \texttt{copy}(sum)
\end{array}
$$

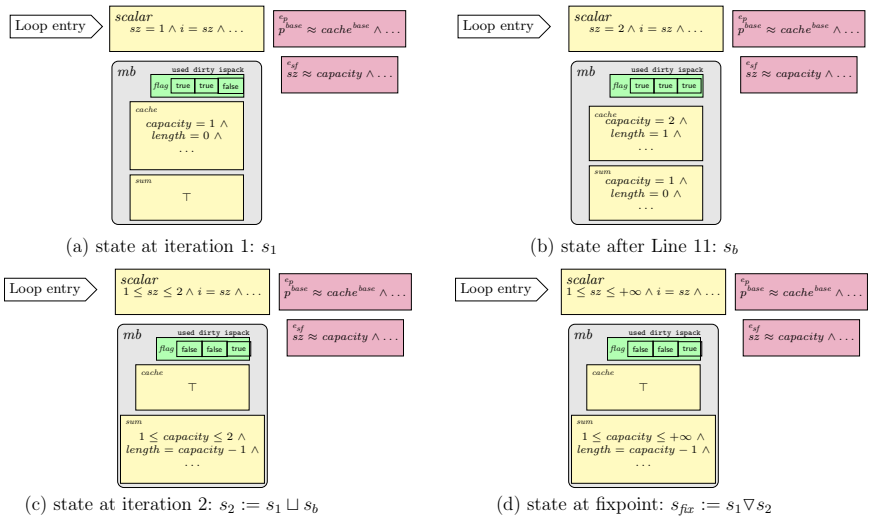
Fig. 11: Abstract cache operations.

to the *fold* and *expand* in [12] but simpler because *cache* and *sum* are two domain values underlying the same field dimensions. After unpacking, $cache^{base}$ equals ptr^{base}, signifying the *cache* is for the new MRU object. The transformer then performs a strong read/update in *cache* without changing any invariant stored in *sum*. The read/update creates an equivalence relation between fld and scl in e_{sf} through `addEqual`. For field read, the transformer discards the information in scl before adding the equality. For field update, the transformer forgets information about fld ahead of setting the equality and sets *dirty* to true afterward.

Other abstract operators, including join, meet, widening, and narrowing, are computed pointwise over subdomains with an additional caching step: packing the dirty cache for each memory bank and resetting it as unused. The full definition for applying domain operators is available in the extended version of the paper [27].

We argue that the abstract semantics is sound as it is systematically derived from the concrete semantics. At each program point, the scalar abstraction over-approximates the set of numeric values or addresses of each scalar variable. For memory objects, the abstraction collapses concrete objects in each memory bank into one summary (abstract) object, also as an over-approximation. The soundness argument follows from our design of abstraction and Galois connections. We omit it here since the abstraction is straightforward.

Fig. 12 illustrates the computation of abstract states at the loop entry of Fig. 4a. In Fig. 12a, state s_1 represents the an abstract state at the loop entry after the first iteration opf the loop. Since during the first iteration only one `byte_buf` object is initialized, the cache in s_1 has the invariants only of that object: $len = 0$ and $cap = 1$, while the summary has no objects (i.e., *ispk* flag is unset). The next abstract state is s_b (Fig. 12b) after line 11. During the second iteration, the cache is flushed for the new `byte_buf` object and the summary only maintains the invariants for the flushed object. Then, s_1 and s_b are joined at the loop entry, resulting in s_2 (Fig. 12c). The join is pairwise across subdomains after



(a) state at iteration 1: s_1

(b) state after Line 11: s_b

(c) state at iteration 2: $s_2 := s_1 \sqcup s_b$

(d) state at fixpoint: $s_{fix} := s_1 \triangledown s_2$

Fig. 12: Fixpoint computation for the entry state of the loop in Fig. 4a.

the caches of both states are flushed. Finally, the widening operator is applied to reach a fixpoint, as shown in Fig. 12d.

As the memory and scalar properties are kept separately, we configure a domain reduction step to exchange information between each bank's *cache* and *scalar* through the equalities that are introduced during load and store. We use a bidirectional reduction (see red arrows on the right of Fig. 7): one direction flows from the $\mathsf{Cache}^\sharp$ of each memory bank to $\mathsf{Scalar}^\sharp$; the other is in the opposite. The domain reduction follows Fig. 13 which reduces an abstract state as $\sigma^\sharp$ in two steps by propagates numerical properties (1) from each *cache* into *scalar*, and (2) from the *scalar* back to each *cache*. The algorithm computes the iterated pairwise reduction through reduce which operates on each bank's *cache* and *scalar*. For example, Fig. 9 shows the *cache* after applying the reduction whose values are refined for `cap` and `len` based on equalities generated for field updates through scalars `sz` and `i` in *scalar*. The *cache* is reduced through the step (2) which involves reduce converting equalities ($len \approx i$ and $cap \approx sz$) into linear constraints and adding them to *scalar*. Then, it performs a meet with *cache* to propagate numerical information from *scalar*. Finally, it projects the result of the meet to the field variables, and obtains the new *cache*.

When reduction is executed once, it refines the abstract values in each bank's *cache* and *scalar* in the state. It adds numerical properties and preserves equalities. This ensures that it is both reductive and sound. We terminate the reduction after one iteration for each of the two directions.

In summary, we introduce MRUD, a composite abstract domain and its corresponding transformer for inferring object invariants. As a reduced product of domains for scalars and objects, MRUD is effective for scalable analysis. The reduction algorithm leverages equalities between variables to avoid precision loss.

$\mathsf{reduce}(base_{src}, base_{dst}, e) \equiv$
  let $e' = \mathsf{project}(e, \mathcal{V}_{src} \cup \mathcal{V}_{dst})$ **and** $cons = \mathsf{toCons}(e')$ **in**
  let $base'_{dst} = \mathsf{project}((base_{dst} \sqcap \mathsf{addCons}(base_{src}, cons)), \mathcal{V}_{dst})$ **in**
  $base'_{dst}$

$\mathsf{reduction}(\sigma^\sharp) \equiv$
  let $\langle scalar, e_{sf}, e_p, mem\rangle = \sigma^\sharp$ **in**
  for all $mb \in mem$ **do** ▷ Step 1: reduce from caches to base
    let $\langle cache, \_, \_\rangle = mb$ **in**
    $scalar' := \mathsf{reduce}(cache, scalar, e_{sf})$
  for all $mb \in mem$ **do** ▷ Step 2: reduce from base to caches
    let $\langle cache, sum, flag\rangle = mb$ **in**
    let $cache' = \mathsf{reduce}(scalar', cache, e_{sf})$ **in**
    $mb := \langle cache', sum, flag\rangle$ ▷ Update mb directly
  $\langle scalar', e_{sf}, e_p, mem\rangle$

Fig. 13: Domain reduction.

5 Implementation

We have implemented the MRUD[4] in Crab [13], a library for building abstract interpretation-based analyses. The $\mathsf{Memory}^\sharp$ is implemented using a Patricia tree [24] for *structural sharing* among multiple abstract elements during analysis. This approach prevents redundant copying of domain values when computing the outputs of domain operators and transfer functions, allowing efficient memory sharing for parts of the abstract state that remain unchanged after an operation. For example, two domain elements of $\mathsf{Memory}^\sharp$ share memory banks if they are unchanged during computation.

We have developed a custom equality domain based on a union-find data structure to represent variable equivalence (e.g., $x \approx y$). The details of this domain are available in the extended version of the paper [27]. Each equivalence class corresponds to a set of variables (e.g., $\{p^{base}, cache^{base}\}$ as $p^{base} \approx cache^{base}$ in Fig. 9). This structure fits the representation of equivalence relations and efficiently supports domain operation. Our implementation also partitions $\mathsf{E}^\sharp_{\mathsf{sf}}$ into reduced product of smaller domains for better alignment with variable packing [3]. Specifically, we use an equality domain $\mathsf{E}^\sharp_{\mathsf{s}}$ for scalars and $\mathsf{E}^\sharp_{\mathsf{f}}$, in each memory bank, for fields. The domain value of $\mathsf{E}^\sharp_{\mathsf{sf}}$ is the union of these smaller domain values. For example, $i \approx len \wedge sz \approx cap$ is maintained as two classes $e_{sf} := \{i, len\}, \{sz, cap\}$ which are equivalent to splitted classes as $e_s := \{i, \tilde{a}\}, \{sz, \tilde{b}\}$ and $e_f := \{len, \tilde{a}\}, \{cap, \tilde{b}\}$ with special representatives $\tilde{a}, \tilde{b}$.

For memory partitioning, we use SeaDsa [10] to divide the memory used by the program into memory banks, with each bank containing objects from the same allocation site. As mentioned earlier, in `CrabIR`, a field variable represents an offset to access an object field. The findmb function of RUMM is defined by

[4] Publicly available at https://github.com/LinerSu/crab/tree/VMCAI-2025.

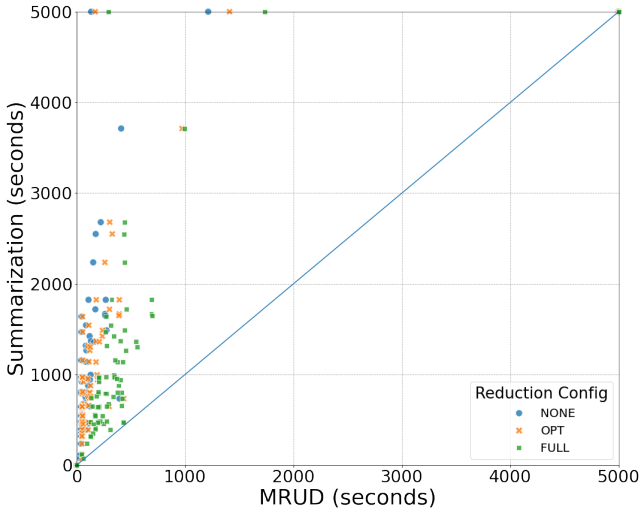


Fig. 14: Scalability results. Summarization refers to $\mathcal{D}_{\mathcal{S}}$ and MRUD to $\mathcal{D}_{\mathcal{O}}$.

mapping fields to their corresponding bank. However, in practice, not all field offsets can be determined statically. We over-approximate the values of such field by $\top$. Improving this is left for future work.

For effective and efficient domain reduction, we use heuristics to balance precision and performance. MRUD tracks which direction needs reduction. For example, if equalities between fields and scalars only affect memory reads, there is no need to apply a reduction to refine the corresponding cache. We also allow reduction to be performed on demand. For instance, reduction is applied when an assertion is present in the program.

6 Evaluation

We performed three kinds of experiments: **scale**, **precision**, and **case study**. All experiments were conducted on a desktop computer with an Intel Xeon E5-2680 @2.50GHz, with 256 GB RAM, and are available at `https://doi.org/10.5281/zenodo.13849174`.

First, the **scale** experiment compares the performance of MRUD ($\mathcal{D}_{\mathcal{O}}$) with the summarization-based [13] domain ($\mathcal{D}_{\mathcal{S}}$) from Crab by timing analysis of 114 programs: 5 from [13], and 109 from GNU Coreutils [11]. We used the Zones[5] [10] abstract domain for its simplicity and sufficiency in expressing (relational) memory safety invariants. The primary goal is to show that $\mathcal{D}_{\mathcal{O}}$ scales better than $\mathcal{D}_{\mathcal{S}}$ due to the effect of variable packing [3] in $\mathcal{D}_{\mathcal{O}}$ that follows from representing each partition with a different DBM, while $\mathcal{D}_{\mathcal{S}}$ relies on a single DBM for expressing

[5] The Zones domain represents all the binary relationships between two-variable difference (including zero), stored in a Difference-Bound Matrix (DBM).

```
1  void foo(){
2    char ary1[1], ary2[2];
3    struct byte_buf o1 = {.len = 0,
           .cap = 1, .buf=ary1};
4    struct byte_buf o2 = {.len = 1,
           .cap = 2, .buf=ary2};
5    struct byte_buf *p;
6    if (/*some conditions*/) {
7      p = &o1;
8    } else {
9      p = &o2;
10   }
11   p->len = 15; p->cap = 20;
12   ...
13 }
```

Fig. 15: Another C program.

Program	#A	$\mathcal{D}_{\mathcal{O}}$	$\mathcal{D}_{\mathcal{S}}$		$\mathcal{D}_{\mathcal{R}}$	
		safe	safe	warn	safe	warn
bytebuf	3	3	0	3	0	3
bytebuf_memcpy	3	3	0	3	0	3
bytebuf_path	3	3	1	2	1	2
ipc_handler	3	3	2	1	2	1
mult_bytebuf	3	3	0	3	0	3
object	1	1	0	1	0	1
range	2	2	1	1	0	2

Table 1: Precision results.

all scalars (included ghost ones) and summary variables. Another goal is to measure the overhead introduced by domain reduction, which incurs extra costs. To evaluate this, we provide two additional strategies: FULL, which applies reduction at each transfer function, and NONE, where no reduction is applied, and compare them with the heuristic strategy, OPT. These three strategies highlight the different costs of reduction.

Fig. 14 shows the timing results, with a timeout of 5 000 seconds per program. Both domains time out on 6 cases, while $\mathcal{D}_{\mathcal{S}}$ times out on 2 more cases. Excluding timeout cases, $\mathcal{D}_{\mathcal{O}}$ outperforms $\mathcal{D}_{\mathcal{S}}$ on nearly every benchmark. On average, $\mathcal{D}_{\mathcal{O}}$ with NONE, OPT, and FULL configurations is 81x, 76x, and 57x faster than $\mathcal{D}_{\mathcal{S}}$, respectively. This demonstrates the advantage of composite abstract domains for inferring object invariants in large and complex programs, regardless of the domain reduction strategy used.

We analyze `ginstall` from GNU Coreutils to understand why $\mathcal{D}_{\mathcal{O}}$ is faster. The running time for $\mathcal{D}_{\mathcal{S}}$ is 1 846s, while for $\mathcal{D}_{\mathcal{O}}$, it takes 273s. Most of the time in both domains is spent on join operations, where $\mathcal{D}_{\mathcal{S}}$ spends 600s, while $\mathcal{D}_{\mathcal{O}}$ takes 95s. Joining in $\mathcal{D}_{\mathcal{O}}$ is also efficient because it allows to share DBMs across memory banks from other states (structural sharing for $\mathsf{Memory}^\sharp$ domain). Another reason is that most DBMs in $\mathcal{D}_{\mathcal{O}}$ are small, making their joins less costly compared to $\mathcal{D}_{\mathcal{S}}$, where large DBMs are involved. This efficiency is also reflected in the time to copy DBMs: $\mathcal{D}_{\mathcal{S}}$ takes 260s, while $\mathcal{D}_{\mathcal{O}}$ takes 20s.

As for domain reduction, applying it at each transfer function is inefficient, as FULL takes 144 (177) seconds longer than OPT (NONE) on average. The heuristics strategy (OPT) effectively handles complex programs without significant performance loss.

Second, the **precision** experiment compares $\mathcal{D}_{\mathcal{O}}$ against existing heap abstract domains: $\mathcal{D}_{\mathcal{S}}$ and Mopsa with recency abstraction ($\mathcal{D}_{\mathcal{R}}$). Since all three domains follow allocation-site abstraction, which summarizes multiple objects into one and treats them indistinguishable, it becomes challenging to precisely track field updates on individual concrete objects. Specifically, $\mathcal{D}_{\mathcal{S}}$ cannot overcome this limitation. $\mathcal{D}_{\mathcal{R}}$ improves precision by differentiating the most recently

allocated object at the same site. $\mathcal{D}_{\mathcal{O}}$ provides a more general strategy by distinguishing the most recently used object. As a result, $\mathcal{D}_{\mathcal{O}}$ still precisely models field updates after object initialization, such as field updates on lines 17 and 19 in Fig. 1, which either $\mathcal{D}_{\mathcal{R}}$ or $\mathcal{D}_{\mathcal{S}}$ cannot handle.

Another challenge is path sensitivity since unclear pointer aliasing leads to imprecise modeling of field updates. For example, in Fig. 15, two `byte_buf` objects, `o1` and `o2`, are allocated separately, and a pointer `p` is referred to either `o1` or `o2`. Modeling strong field updates in line 11 requires knowing which object is being updated, but it is unknown which object the pointer `p` refers to. Both $\mathcal{D}_{\mathcal{R}}$ and $\mathcal{D}_{\mathcal{S}}$ can track field updates precisely, but they need more accurate points-to information. $\mathcal{D}_{\mathcal{O}}$, however, allows strong updates by placing `o1` and `o2` in the same memory bank. When updating a field on either object, we load it into the cache and perform strong updates without precise pointer aliasing.

We provide a set of 7 benchmarks[6] with similar code pattern like examples in Figs. 1 and 15 for evaluation and configure all three domains using the octagon domain. Table 1 shows that $\mathcal{D}_{\mathcal{O}}$ successfully proves all assertions, showing the effectiveness of our methodology in providing a more precise memory abstraction. Conversely, $\mathcal{D}_{\mathcal{S}}$ and $\mathcal{D}_{\mathcal{R}}$ largely fail due to weak updates, as discussed above.

Third, we present a **case study** which integrates an Abstract Interpreter (AbsInt) into a Bounded Model Checker (BMC) pipeline for memory safety verification. This new pipeline, AI4BMC, uses AbsInt to verify and remove a number of assertions before passing the problem to the SMT solver.

The AI4BMC pipeline, shown in Fig. 16, starts by compiling and instrumenting the input program with buffer overflow checks. Next, AbsInt is applied to remove as many of these checks as possible. Now, the program still keeps the original loops. Then, the loops are unrolled using a user-supplied bound for BMC. Later, we run another AbsInt round to eliminate buffer overflow checks in the simplified program with unrolled loops. Last, we continue with the BMC pipeline, as in SeaBMC [25], that generates a Verification Condition (VC) in SMT-LIB and uses an SMT-solver to check the VC's satisfiability such that the original program is safe if and only if SMT-LIB formula is unsatisfiable.

The motivation for AI4BMC is that many memory safety arguments are simple and are established independently of loop bounds. We expect AbsInt to verify those, leaving less work for BMC. Thus, we consider AI4BMC pipeline successful if (a) AbsInt discharges some buffer overflow checks before loop unwinding, and (b) AI4BMC requires less overall runtime than the BMC pipeline.

We developed two benchmark suites from industrial code. The first is based on `aws-c-commons` verification tasks, where we reduce assertions only to memory safety. The second is based on a more complex code from AWS C SDK in C99 implementation. Together, there are 109 verification tasks. The benchmarks[7] have been adapted to simplify control flow since proving all memory safety checks requires path-sensitivity.

[6] Available at: https://github.com/LinerSu/MRU-Domain-Benchmarks.
[7] Available at https://github.com/LinerSu/verify-c-common/tree/VMCAI-2025.

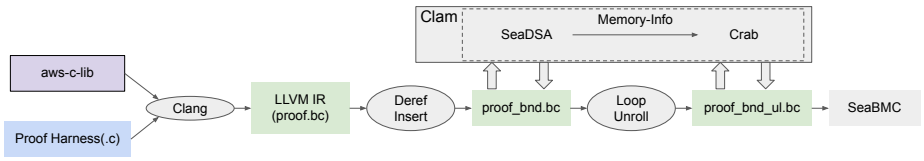


Fig. 16: The AI4BMC pipeline.

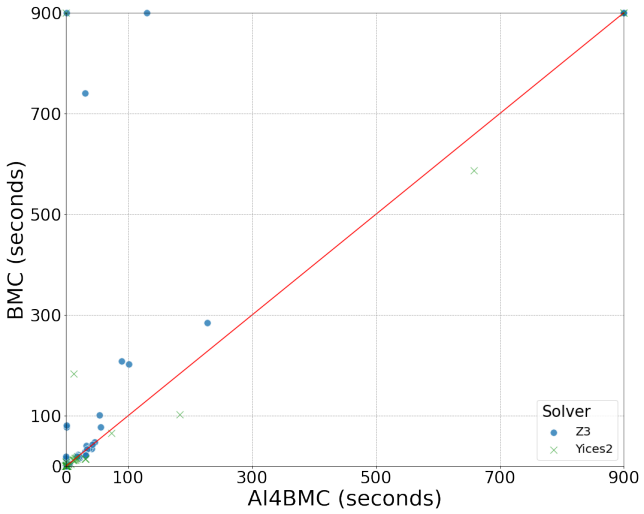


Fig. 17: AI4BMC vs. BMC.

We evaluate the effectiveness AI4BMC by comparing it with SeaBMC which was previously compared against other state-of-the-art tools in [25]. Our performance evaluation focuses on these metrics: (1) *Faster* indicates AI4BMC outperforms BMC; (2) *Slower* means AI4BMC is slower than BMC; (3) *AbsInt Time* expresses the run-time of AbsInt in the AI4BMC pipeline. For precision, we provide the *AbsInt Solving Rate*, showing how many checks are solved before or after loop unrolling (LU). We used MRUD for Crab (AbsInt) and chose two SMT-solvers for SeaBMC: Z3[8] [23], and Yices2 [9]. Experiments were conducted under 900 seconds timeout and all results are summarized in Fig. 17 and Table 2.

First, comparing performance between AI4BMC and BMC. With Z3, AI4BMC timed out in 5 cases, while BMC timed out in 7 cases; AbsInt helped solving 2 more cases. Excluding timeouts, AI4BMC is at least 5s *faster* than BMC in 16 cases. The speed-up comes from AbsInt proving and discharging assertions checks. In 10 of these 16 cases, the speed-up exceeds over 95%, with AbsInt

[8] We fixed the performance issue on Z3. The one we used is available at: https://github.com/LinerSu/z3/tree/fix-performance.

Category	Metric	% Metric	Number of Cases	
			AI4BMC (Z3)	AI4BMC (Y2)
Performance Comparison	*Faster* (Time Difference > 5s)	> 95%	10	2
		others	6	1
	Slower (Time Difference > 5s)	≤ 50%	4	2
		others	0	4
AbsInt Performance	*AbsInt Time* in AI4BMC time	> 40%	65	74
		≤ 40%	39	29
Precision	*AbsInt Solving Rate* before LU	100%	37	37
		> 50%	52	52
	AbsInt Solving Rate after LU	100%	6	6
		> 50%	1	1

Table 2: AI4BMC vs. BMC details.

completely solving the checks in 9 cases. The other 6 cases show at least a 20% speed-up. AbsInt takes under one second on average in all 16 cases. There are 4 cases in which AI4BMC is at least 5s *slower* than BMC. In two of these, the slowdowns are due to Z3 taking 6s extra solving time on average, which is not surprising since the SMT performance is not always deterministic. In the other two, although Z3 solving time is decreased, AbsInt slows down by taking around 11s, roughly a third of the total run-time.

The results with Yices2 are similar, but Yices2 is faster and exhibits better stability. Both AI4BMC and BMC timed out in 4 cases and 5 cases individually, with 1 case where AbsInt improves performance. AI4BMC outperforms BMC in 3 cases with at least a 93% improvement. However, AI4BMC is slower in 6 cases, 4 of which are affected by the slowdown of AbsInt. The other 2 cases are due to the slows down of SeaBMC and Yices2. The SeaBMC experiences a slowdown due to lambda-encoding, where the beta-reduction simplification time is not deterministic. While switching to array-encoding shows the effectiveness of AbsInt, this slows overall performance for both AI4BMC and BMC.

Overall, the performance results show that AbsInt improves the overall performance of using BMC regardless of the solver used.

Second, in evaluating the performance of AbsInt, runtime ratios depend on the total running time of AI4BMC and the solver selected. With Z3, AbsInt takes over 40% of the time on 65 cases, but these cases terminate within 50s, with AbsInt averaging only 0.1s and maxing at 1.2s. For the rest of the 39 cases, AbsInt takes 40% or less, with 5 cases exceeding 50s and 34 cases under 50s. For these 5 longer cases, AbsInt accounts for under 2%, averaging 1s with a maximum of 1.5s. For the 34 shorter cases, AbsInt contribution was below 36%. With Yices2, the runtime percentage of AbsInt increases because Yices2 is efficient, with more cases where AbsInt accounts for a significant portion of the runtime. In summary, using AbsInt has no big cost, compared with the solving time of SMT solver.

Last, for assertion rate, AbsInt solved more than 50% of assertions in 89 cases before LU, completely solving 37 cases, and in 7 cases after LU, fully solving 6 cases. We only have 8 cases where AbsInt solves less than half of the checks. The reasons are: (1) the widening operation produces too imprecise

invariants that cannot be recovered by narrowing. AbsInt needs more precise widening techniques to prove more checks; (2) Some memory safety invariants cannot be expressed by Zones or Octagons, and instead require more complex abstract domains such as Polyhedra; (3) Memory safety checks for C string require tracking the length of strings that our implementation does not support. We believe using [15] to determine the null character of each string will improve overall precision.

In this case study, we demonstrate the effectiveness of using AbsInt in the BMC pipeline. By using the Zones, it proves most memory safety checks in this industry project and reduces the number of checks BMC handles. This speeds up both BMC encoding and SMT solver performance.

7 Related Works

To deal with a potentially unbounded number of memory objects, most abstract analysis frameworks group memory objects together into *summary objects* (e.g., [12]). A summary object represents properties that are common to all objects it stands for. The most common summarization is *Allocation Site Abstraction* (ASA) [5] that groups objects by their allocation site. In ASA, all concrete objects allocated at a certain line of a program are represented by one abstract summary object. Since each summary object represents a set of objects, it supports only *weak* updates – an assignment to the field of an object does not override previous value, but rather adds to it, to capture that the field update may modify only one object out of the summary. This significantly degrades analysis precision.

The loss of precision is specifically important during object creation, when an object is first allocated and then initialized field-by-field. In ASA, because of weak updates, this results in all properties of the summary being lost since the newly allocated object has no properties in common with already summarized objects. A common solution, e.g., used by Mopsa, is *recency abstraction* [1] that refines ASA into: (a) the most recently allocated object, and (b) the rest. Since most recent object is a singleton, it can be updated *strongly*, i.e., field updates overwrite previous values. Our approach is a further refinement that separates objects not by recency of *creation*, but by recency of *use*. In principle, other extensions of recency, such as [2] can be combined with our technique for further precision improvement.

The temporary isolation of recenctly-used objects avoids invariant violations in summarized objects during individual field updates. Our pack and unpack methods communicate changes between these two types of objects. This is similar to corresponding methods in [4], where the annotated *pack/unpack* statements manage transitions of mutable objects during class method calls, allowing temporary updates while maintaining class invariants (i.e., invariants for all instances of a given class). Similarly, JayHorn [16] uses *push/pull* statements for encoding each memory access. Each *pull* statement reads fields of an object to make invariants available, while a following *push* statement updates fields to

ensure modifications preserve invariants. The concept of *pack/unpack* has been used in refinement types [26], where the inference algorithm obtains predicates with *fold/unfold* operations to prevent temporary invariant violations of objects from the same allocation site. Unlike our work, all prior work uses heuristics to manage placement of *fold/unfold* operations. In contrast, our analysis automatically processes these during analysis.

The domain hierarchy in our MRUD uses two strategies. First, variable packing [3] is used to pack program variables for fields of memory objects in each memory bank. With two numerical domains per pack, our approach allows for the independent updating of invariants for each bank. The packing is rarely used in computing memory properties, but Toubhans et al. [28] introduced a product of memory domains that pack variables used for lists, trees, and other fixed-size structures. Second, domain reduction [6] helps exchange equivalences between scalars and object fields. This is commonly used when abstract domains are organized modularly. Astrée [7] combines various abstract domains in a sequence, using reduction steps for forward and backward propagation of information between them. [8] interprets the Nelson-Oppen procedure as a domain reduction, propagating (dis)equalities across domains.

8 Conclusion

In this work, we present a new methodology for inferring object invariants that avoids temporarily breaking invariants following the concept of caching. Our new abstract domain, parameterized by numerical and equality domains, organizes a structured hierarchy, enabling scalable analysis of complex programs. We design a reduction algorithm following equalities introduced across numerical domains to avoid significant precision loss. Our results demonstrate that MRUD enhances both precision and scalability and can be effectively integrated with other verification techniques for memory safety.

References

1. Balakrishnan, G., Reps, T.W.: Recency-abstraction for heap-allocated storage. In: Yi, K. (ed.) Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4134, pp. 221–239. Springer (2006). https://doi.org/10.1007/11823230_15, https://doi.org/10.1007/11823230_15
2. Balatsouras, G., Smaragdakis, Y.: Structure-sensitive points-to analysis for C and C++. In: Rival, X. (ed.) Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9837, pp. 84–104. Springer (2016). https://doi.org/10.1007/978-3-662-53413-7_5, https://doi.org/10.1007/978-3-662-53413-7_5
3. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Cytron, R., Gupta, R. (eds.) Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June

9-11, 2003. pp. 196–207. ACM (2003). https://doi.org/10.1145/781131.781153, https://doi.org/10.1145/781131.781153
4. Chang, B.E., Leino, K.R.M.: Inferring object invariants: Extended abstract. In: Cortesi, A., Logozzo, F. (eds.) Proceedings of the First International Workshop on Abstract Interpretation of Object-oriented Languages, AIOOL@VMCAI 2005, Paris, France, January 21, 2005. Electronic Notes in Theoretical Computer Science, vol. 131, pp. 63–74. Elsevier (2005). https://doi.org/10.1016/J.ENTCS.2005.01.023, https://doi.org/10.1016/j.entcs.2005.01.023
5. Chase, D.R., Wegman, M.N., Zadeck, F.K.: Analysis of pointers and structures. In: Fischer, B.N. (ed.) Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990. pp. 296–310. ACM (1990). https://doi.org/10.1145/93542.93585, https://doi.org/10.1145/93542.93585
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Aho, A.V., Zilles, S.N., Rosen, B.K. (eds.) Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979. pp. 269–282. ACM Press (1979). https://doi.org/10.1145/567752.567778, https://doi.org/10.1145/567752.567778
7. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of abstractions in the astrée static analyzer. In: Okada, M., Satoh, I. (eds.) Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers. Lecture Notes in Computer Science, vol. 4435, pp. 272–300. Springer (2006). https://doi.org/10.1007/978-3-540-77505-8_23, https://doi.org/10.1007/978-3-540-77505-8_23
8. Cousot, P., Cousot, R., Mauborgne, L.: The reduced product of abstract domains and the combination of decision procedures. In: Hofmann, M. (ed.) Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6604, pp. 456–472. Springer (2011). https://doi.org/10.1007/978-3-642-19805-2_31, https://doi.org/10.1007/978-3-642-19805-2_31
9. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 737–744. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49, https://doi.org/10.1007/978-3-319-08867-9_49
10. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Exploiting sparsity in difference-bound matrices. In: Rival, X. (ed.) Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9837, pp. 189–211. Springer (2016). https://doi.org/10.1007/978-3-662-53413-7_10, https://doi.org/10.1007/978-3-662-53413-7_10
11. GNU Project: Gnu core utilities official page, https://www.gnu.org/software/coreutils/
12. Gopan, D., DiMaio, F., Dor, N., Reps, T.W., Sagiv, S.: Numeric domains with summarized dimensions. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on The-

ory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2988, pp. 512–529. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_38, https://doi.org/10.1007/978-3-540-24730-2_38

13. Gurfinkel, A., Navas, J.A.: Abstract interpretation of LLVM with a region-based memory model. In: Bloem, R., Dimitrova, R., Fan, C., Sharygina, N. (eds.) Software Verification - 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18-19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18-19, 2021, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13124, pp. 122–144. Springer (2021). https://doi.org/10.1007/978-3-030-95561-8_8, https://doi.org/10.1007/978-3-030-95561-8_8
14. Huston, B.: Single-chip microcomputers can be easy to program. In: American Federation of Information Processing Societies: 1982 National Computer Conference, 7-10 June, 1982, Houston, Texas, USA. AFIPS Conference Proceedings, vol. 51, pp. 85–93. AFIPS Press (1982). https://doi.org/10.1145/1500774.1500786, https://doi.org/10.1145/1500774.1500786
15. Journault, M., Miné, A., Ouadjaout, A.: Modular static analysis of string manipulations in C programs. In: Podelski, A. (ed.) Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11002, pp. 243–262. Springer (2018). https://doi.org/10.1007/978-3-319-99725-4_16, https://doi.org/10.1007/978-3-319-99725-4_16
16. Kahsai, T., Kersten, R., Rümmer, P., Schäf, M.: Quantified heap invariants for object-oriented programs. In: Eiter, T., Sands, D. (eds.) LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017. EPiC Series in Computing, vol. 46, pp. 368–384. EasyChair (2017). https://doi.org/10.29007/ZRCT, https://doi.org/10.29007/zrct
17. Karr, M.: Affine relationships among variables of a program. Acta Informatica **6**, 133–151 (1976). https://doi.org/10.1007/BF00268497, https://doi.org/10.1007/BF00268497
18. Lattner, C., Adve, V.S.: Automatic pool allocation: improving performance by controlling data structure layout in the heap. In: Sarkar, V., Hall, M.W. (eds.) Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005. pp. 129–142. ACM (2005). https://doi.org/10.1145/1065010.1065027, https://doi.org/10.1145/1065010.1065027
19. Meyer, B.: Object-oriented software construction (2nd ed.). Prentice-Hall, Inc., USA (1997)
20. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) Programs as Data Objects, Second Symposium, PADO 2001, Aarhus, Denmark, May 21-23, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2053, pp. 155–172. Springer (2001). https://doi.org/10.1007/3-540-44978-7_10, https://doi.org/10.1007/3-540-44978-7_10
21. Miné, A.: The octagon abstract domain. In: Burd, E., Aiken, P., Koschke, R. (eds.) Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001. p. 310. IEEE Computer Society (2001). https://doi.org/10.1109/WCRE.2001.957836, https://doi.org/10.1109/WCRE.2001.957836

22. Monat, R., Ouadjaout, A., Miné, A.: Mopsa-c: Modular domains and relational abstract interpretation for C programs (competition contribution). In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13994, pp. 565–570. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_37, https://doi.org/10.1007/978-3-031-30820-8_37
23. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24, https://doi.org/10.1007/978-3-540-78800-3_24
24. Okasaki, C., Gill, A.: Fast mergeable integer maps. In: Notes of the ACM SIGPLAN Workshop on ML. pp. 77–86 (1998)
25. Priya, S., Su, Y., Bao, Y., Zhou, X., Vizel, Y., Gurfinkel, A.: Bounded model checking for LLVM. In: Griggio, A., Rungta, N. (eds.) 22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022. pp. 214–224. IEEE (2022). https://doi.org/10.34727/2022/ISBN.978-3-85448-053-2_28, https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_28
26. Rondon, P.M., Kawaguchi, M., Jhala, R.: Low-level liquid types. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010. pp. 131–144. ACM (2010). https://doi.org/10.1145/1706299.1706316, https://doi.org/10.1145/1706299.1706316
27. Su, Y., Navas, J.A., Gurfinkel, A., Garcia-Contreras, I.: Automatic inference of relational object invariants (2024), https://arxiv.org/abs/2411.14735
28. Toubhans, A., Chang, B.E., Rival, X.: An abstract domain combinator for separately conjoining memory abstractions. In: Müller-Olm, M., Seidl, H. (eds.) Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8723, pp. 285–301. Springer (2014). https://doi.org/10.1007/978-3-319-10936-7_18, https://doi.org/10.1007/978-3-319-10936-7_18