Abstract Interpretation of LLVM with a Region-Based Memory Model*

Arie Gurfinkel[1] and Jorge A. Navas[2]

[1] University of Waterloo (Canada)
arie.gurfinkel@uwaterloo.ca
[2] SRI International (USA)
jorge.navas@sri.com

Abstract. Static analysis of low-level programs (C or LLVM) requires modeling memory. To strike a good balance between precision and performance, most static analyzers rely on the C memory model in which a pointer is a numerical offset within a memory object. Finite partitioning of the address space is a common abstraction. For instance, the *allocation-site* abstraction creates partitions by merging all objects created at the same allocation site. *Recency* abstraction refines the allocation-site abstraction by distinguishing the most recent allocated memory object from the previous ones. Unfortunately, these abstractions are not often precise enough to infer invariants that are expressed over the contents of dynamically allocated data-structures such as linked lists. In those cases, more expensive abstractions such as *shapes* that consider connectivity patterns between memory locations are often needed.
Instead of resorting to expensive memory abstractions, we propose a new memory model, called *region-based memory model (*RBMM*)*. RBMM is a refinement of the C memory model in which pointers have an extra component called *regions*. Thus, a memory object can spawn multiple regions which can greatly limit aliasing since regions are pairwise disjoint. Since RBMM requires that each memory instruction refers explicitly to a region, we first present a new intermediate representation (IR) based on regions which is the input of our abstract interpreter Crab. Second, we show how abstractions such as allocation-site and recency can be easily adapted to RBMM. Third, we evaluate Crab using our new IR and a simple allocation-site abstraction on widely-used C projects.

1 Introduction

One of the most difficult problems in Abstract Interpretation of a low-level language, such as C or LLVM, is modeling memory. An abstract interpreter must strike a balance between a model that is sufficiently abstract to allow for efficient analysis, yet sufficiently precise to reduce false positives. Abstracting memory

* Jorge A. Navas has been supported in part by NSF grant 1816936. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

consists of partitioning the unbounded number of concrete memory objects at run-time into a finite set of abstract objects, sometimes, further sub-divided into fields. Popular memory abstractions vary from cheaply *smashing* all memory objects originated from the same *allocation site* into a summarized abstract object to more precise but expensive modeling of connectivity between objects allowing to reason about *shapes*. In between, abstractions that partition memory by the *recency* of the objects are also popular because they can allow to distinguish objects allocated in different loop iterations while still being efficient to compute.

In this paper, we focus on Abstract Interpretation of general-purpose, modern (portable) C programs. Our goal is to compute non-trivial invariants that involve both scalar variables and values stored in memory for programs that allocate memory dynamically. We propose a new memory model, called *region-based memory model* (RBMM), which can boost the precision of existing memory abstractions such as smashing[3] and recency while retaining their efficiency.

RBMM partitions memory into a finite set of *regions*. In its simplest form, a *region* is a set of memory objects. More generally, a region is a set of slices (i.e., contiguous sub-fields) of objects. A *reference*[4] is an offset within a memory object refined with a region. In RBMM, all regions are pairwise disjoint and, therefore, a write into one region cannot affect data stored in another region. In this paper, we show that an abstract interpreter based on RBMM can infer interesting properties on realistic programs that manipulate dynamically allocated data structures using very simple memory abstractions such as smashing.

Our approach consists of the following steps: (1) translate the C program into LLVM bitcode and apply a whole-program pointer analysis on the LLVM program so that memory is statically partitioned into regions. The analysis also identifies which parts of the program might not satisfy the assumptions of our memory model; (2) translate the LLVM program into a novel intermediate-representation (IR), called CrabIR, where all LLVM memory instructions are translated to instructions over regions and references; (3) run the Crab abstract interpreter on the CrabIR program.

We argue that having a specialized IR (CrabIR) for our memory model poses several advantages. First, we separate the source language (LLVM in this case) from the language analyzed by the abstract interpreter. This makes the abstract interpreter reusable as long as the memory model is compatible. Second, the use of CrabIR simplifies the design of the abstract interpreter without jeopardising precision. In CrabIR, each memory instruction is explicitly defined over regions. This allows the abstract interpreter to reason about memory contents without necessarily a complex memory abstract domain. Nevertheless, the precision of the analysis in this case would depend on how precise is the pointer analysis from Step 1. In a pessimistic scenario, all the memory instructions can be mapped to a single region. For those cases, more complex memory abstractions (e.g., [20])

[3] In this paper, we use the terms *smashing* and *allocation-site* interchangeably.

[4] Usually, the term *reference* is used as an alias for an address that precludes pointer arithmetic. We do not place such a restriction. We use reference as an alternative to *pointer*, to stress that a reference belongs to a statically known region.

can be used. Third, our multi-step approach allows us to benefit from the active area of pointer analysis of LLVM (e.g., SeaDsa [14, 19], SVF [24]) saving us from dealing with all the intricacies of the LLVM pointer semantics.

Assumptions of RBMM. The soundness of RBMM relies on two key assumptions: (1) memory is *word-addressable*[5], and (2) programs do not exhibit *undefined behaviour* (UB) under C11 standard. The former highly simplifies the semantics of CrabIR and the abstract transfer functions. The latter allows to restrict aliasing by assuming that the strict aliasing rules are always satisfied (the effective type of a *lvalue* must be compatible[6] with the effective type of the object being accessed). Moreover, absence of UB allows our analysis to use abstractions such as smashing by assuming, for instance, that a memory read can only access to initialized data. Although it may seem counter-intuitive, this assumption does not limit our analysis from proving absence of UB. As shown by Conway et al. [8], conditionally sound analyses can prove absence of errors (e.g., memory violations) or otherwise, produce one counterexample although it cannot produce all possible counterexamples. Informally, this means that we can still use our analysis to find the first instruction that causes UB. If the analysis does not find such an instruction then the analysis proves soundly that UB is not possible. In Sec. 7, we show that these assumptions are reasonable and not too restrictive for the analysis of modern C programs.

Contributions. In summary, the contributions of this paper are: (1) A memory model based on regions that refines the C memory model (Sec. 3). (2) Describe the syntax and concrete semantics of a new intermediate representation called CrabIR, based on RBMM (Sec. 4 and Sec. 5). (3) Adapt abstractions such as smashing and recency to our RBMM (Sec. 6). (4) An implementation in the Crab (`https://github.com/seahorn/crab`) abstract interpretation library and empirical evidence of the practicality of our approach on a set of widely-used C projects (Sec. 7).

2 Motivating Example

In this section, we motivate the benefits of our memory model using an example program shown in Fig. 1. Note that while our tool works at the level of LLVM intermediate representation, we present the example in C for readability.

Our analysis understands three special functions: `int_nd` returns an integer chosen non-deterministically; `assume(b)` blocks if the condition `b` is false, and,

[5] Memory is *word-addressable* if all reads and writes access to the same, fixed amount of bytes (the CPU word length). Note that any C program can be ported to word-addressable memory by adding extra instructions that fill a word with empty bytes before a memory write or extract some bytes from a word after a memory read.

[6] Informally, casting any pointer to a `char*` type is allowed but the opposite is not. Moreover, signedness and constness do not affect the strict aliasing rules.

```
1  typedef struct node{
2   int   *data;
3   size_t len; size_t cap;
4   struct node *n;
5  } *List;
6
7  int* init(size_t sz, int val) {
8   int* data =
9    (int*)malloc(sizeof(int)*sz);
10  assume(data > 0);
11  for (int i=0;i<sz;++i)
12    data[i] = val;
13  return data;
14 }
15 List mk_list(size_t list_sz,
       size_t data_sz, int val) {
16  List l = 0;
17  for(size_t i=0;i<list_sz;i++){
18    List tmp = (List)malloc(sizeof
          (struct node));
19    assume(tmp > 0);
20    tmp->data = init(data_sz,val);
21    int used = int_nd();
22    assume(used>=0);
23    assume(used<data_sz);
24    tmp->len = used;
25    tmp->cap = data_sz;
26    tmp->n = l ;
27    l = tmp;
28  }
29  return l;
30 }
31 void main() {
32   // Create a linked-list
33   size_t list_sz = int_nd();
34   size_t data_sz = int_nd();
35   int val = int_nd();
36   List node = mk_list(list_sz,
         data_sz, val);
37
38   // Check properties
39   for(;node;node=node->n) {
40     sassert(node->data > 0);
41     sassert(node->len < data_sz);
42     sassert(node->len <= node->cap);
43     for(int i=0; i<data_sz; ++i)
44       sassert(node->data[i] == val);
45   }
46 }
```

Fig. 1: C program that creates a linked-list and checks some properties.

otherwise, restricts the execution by setting `b` to true; and, `sassert(b)` *statically* reports an error if `b` is false, or it behaves as `assume(b)` otherwise.

In the example, `struct node` defines the type of a node of a singly linked list with a pointer `n` pointing to the next node in the list. The node has also some data (`data`) defined as an array of integers. The field `cap` is the amount of space allocated (capacity) for `data` and the field `len` is the number of actual bytes held in the node which is not necessarily equal to `cap`. First, the program creates an unbounded, non-empty linked list of `list_sz` nodes and for each node it creates some data of capacity `data_sz` (lines 33-36). Both the contents of the data and its length are chosen non-deterministically. Lines 22-23 ensures that `len` is always less than `cap`. Second, the program iterates over the linked list (lines 39-45) and checks for the following four properties:

1. (Line 40) For every node `n` in the list, `n`→`data` is not null.
2. (Line 41) For every node `n` in the list, `n`→`len` is less than `data_sz`.
3. (Line 42) For every node `n` in the list, `n`→`len` is less or equal than `n`→`cap`.
4. (Line 44) For every node `n` in the list, ∀`i` :: `n`→`data[i]` == `val`.

This program is challenging for automatic static analysis tools. In fact, mature abstract interpreters such as Infer [6] and IKOS [5] fail to prove more than one of the four properties. The reason is that the analyzer needs to reason about

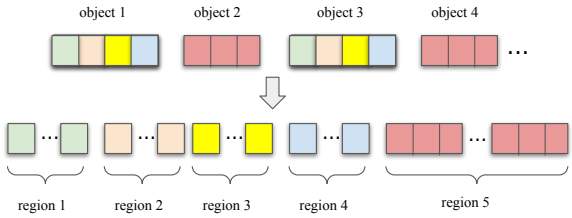


Fig. 2: The C versus the region-based memory model.

two difficult but different aspects. First, it needs to quantify over all nodes of the linked list. Second, it requires inferring numerical relationships between the contents of the linked list. None of the above-mentioned tools can do both.

We argue that these properties should not be hard to prove automatically without resorting to complex memory abstractions. All properties rely on the fact that they uniformly hold on every node in the linked list (Property 1-3) or every array `data` element (Property 4). Smashing or recency do not work because each node of the linked list consists of two different groups of fields: the data (`cap`, `len`, and `data`) and the pointer to the next node (`n`). Smashing both groups make the analysis unable to infer anything about the linked list.

RBMM partitions memory in such a way that smashing alone[7] can prove the four properties. Using an inexpensive flow-insensitive pointer analysis, our memory model partitions memory into four regions: one for every field in the linked list node. This is what makes smashing to be successful in proving all the properties because, properties about the next pointer are not mixed with the other fields, and properties among `cap`, `len`, and `data` are also kept separate.

3 Standard C versus Region-Based Memory Model

A memory model defines the semantics of pointers together with the assumptions that are made about how memory is addressed. We start by defining the most popular memory model for C programs, which we call the standard C memory model, and then we define our new region-based memory model (RBMM).

In the standard C memory model, a pointer is of type $\texttt{ptr} = \mathcal{O} \times \mathbb{Z}$. That is, a pointer is a pair of a memory object and a numerical offset of integer type. $\mathcal{O}$ is the countable infinite set of memory objects. A memory object is a sequence of bytes. Pointer arithmetic or pointer comparison involving pointers from different memory objects is undefined. For instance, consider Line 18 in Fig. 1. Let us assume that each field of `struct node` occupies 8 bytes. Consider the first loop iteration ($i = 0$). The `malloc` instruction allocates a memory object o^0_{l20} of 32 bytes (the superscript indicates the loop iteration and the subscript the line number of the allocation site). Then, `tmp->data`, `tmp->len`, `tmp->cap` and `tmp->n` are the pointers $\langle o^0_{l20}, 0\rangle$, $\langle o^0_{l20}, 8\rangle$, $\langle o^0_{l20}, 16\rangle$, and $\langle o^0_{l20}, 24\rangle$, respectively.

[7] Crab proves the four properties using smashing abstraction and unrolling the loop one iteration. Finite loop unrolling would not help Infer and IKOS.

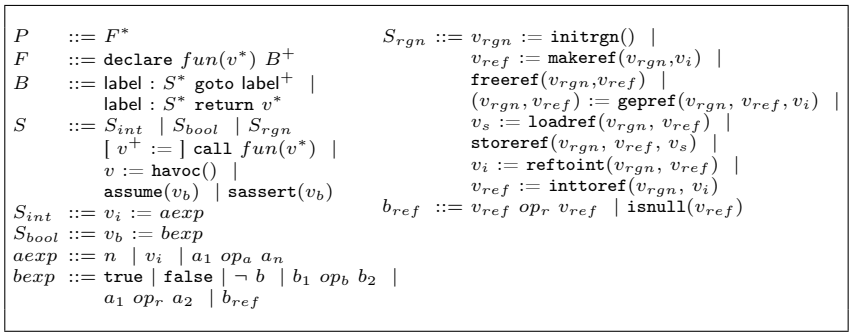
In RBMM, a pointer has an extra component, called a *region*. In the rest, we use the term "reference" instead of "pointer" to make clear that we refer to a pointer in the RBMM. A *reference* is of type $\mathsf{ref} = \mathcal{R} \times \mathcal{O} \times \mathbb{Z}$, where $\mathcal{R}$ is the finite set of regions. In its simplest form, a region is a set (possibly singleton) of memory objects. More generally, a memory object can span multiple regions. That is, different object fields (i.e., subsequence of bytes) can be in different regions. Therefore, in general, a region represents field values of a set of objects.

In RBMM, every memory operation must indicate explicitly which region it is accessing. Since each memory operation indicates a region, accessing memory with a reference from a different region is undefined. To make RBMM compatible with the C memory model, pointer arithmetic and pointer comparison is allowed between references from different regions but they must point to the same memory object. Moreover, RBMM requires that no memory access can span multiple regions. Thus, every memory access must be a word and the partitioning of memory into regions must be done at the word level so that each region contain a multiple number of words. Note that in practice, this is also often required by most hardware platforms, and any potentially unaligned memory access is compiled into multiple aligned accesses. A key property in RBMM is that two references cannot alias if they point to different regions.

Back to Fig. 1, let us assume that in RBMM there are 4 regions $\{r_1, r_2, r_3, r_4\}$. Assume also that each field of `struct node` belongs to one of these regions. The loop at lines 17-28 might allocate multiple memory objects o^i_{l20}, where $0 \leq i < n$ is the loop counter. Then, `tmp->data`, `tmp->len`, `tmp->cap` and `tmp->n` hold the references $\{\langle r_1, o^0_{l20}, 0\rangle, \ldots, \langle r_1, o^{n-1}_{l20}, 0\rangle\}$, $\{\langle r_2, o^0_{l20}, 8\rangle, \ldots, \langle r_2, o^{n-1}_{l20}, 8\rangle\}$, $\{\langle r_3, o^0_{l20}, 16\rangle$,- $\ldots, \langle r_3, o^{n-1}_{l20}, 16\rangle\}$, and $\{\langle r_4, o^0_{l20}, 24\rangle, \ldots, \langle r_4, o^{n-1}_{l20}, 24\rangle\}$, respectively.

The main differences between the two memory models are illustrated in Fig. 2. At the top, we show how memory is represented by the C memory model. Memory can be seen as a collection of memory objects. Each memory object is a sequence of consecutive bytes. At the bottom, we show how memory is organized under RBMM. The same combination of colors represents that the memory object is originated at the same allocation site. In this example, we assume that there are only two allocation sites (as_1 and as_2). Objects 1 and 3 (2 and 4) are allocated at as_1 (as_2). The first key difference is that the number of regions is finite. To achieve this, many (possibly infinite) objects are grouped into a single region (e.g., objects 2, 4, and any other object originated from as_2 into region 5). The second difference is that fields of the same memory object (represented with different colors) can be grouped into different regions. For instance, the green and yellow slices from object 1 are mapped to regions 1 and 3.

In conclusion, RBMM can be seen as a refinement of the standard C model by clustering together an unbounded number of object fields into a finite number of regions. As a result, RBMM can greatly simplify the design of abstract domains since two references cannot alias if they belong to different regions. On the other hand, RBMM does not impede the use of more precise memory abstractions since RBMM is compatible with the C memory model.



$$
\begin{array}{lll}
P & ::= & F^* \\
F & ::= & \texttt{declare}\ fun(v^*)\ B^+ \\
B & ::= & \mathsf{label} : S^*\ \texttt{goto}\ \mathsf{label}^+ \ \mid \\
 & & \mathsf{label} : S^*\ \texttt{return}\ v^* \\
S & ::= & S_{int} \mid S_{bool} \mid S_{rgn} \\
 & & [\ v^+ :=]\ \texttt{call}\ fun(v^*) \ \mid \\
 & & v := \texttt{havoc}() \ \mid \\
 & & \texttt{assume}(v_b) \mid \texttt{sassert}(v_b) \\
S_{int} & ::= & v_i := aexp \\
S_{bool} & ::= & v_b := bexp \\
aexp & ::= & n \mid v_i \mid a_1\ op_a\ a_n \\
bexp & ::= & \texttt{true} \mid \texttt{false} \mid \neg\ b \mid b_1\ op_b\ b_2 \ \mid \\
 & & a_1\ op_r\ a_2 \mid b_{ref}
\end{array}
$$

$$
\begin{array}{lll}
S_{rgn} & ::= & v_{rgn} := \texttt{initrgn}() \ \mid \\
 & & v_{ref} := \texttt{makeref}(v_{rgn}, v_i) \ \mid \\
 & & \texttt{freeref}(v_{rgn}, v_{ref}) \ \mid \\
 & & (v_{rgn}, v_{ref}) := \texttt{gepref}(v_{rgn}, v_{ref}, v_i) \ \mid \\
 & & v_s := \texttt{loadref}(v_{rgn}, v_{ref}) \ \mid \\
 & & \texttt{storeref}(v_{rgn}, v_{ref}, v_s) \ \mid \\
 & & v_i := \texttt{reftoint}(v_{rgn}, v_{ref}) \ \mid \\
 & & v_{ref} := \texttt{inttoref}(v_{rgn}, v_i) \\
b_{ref} & ::= & v_{ref}\ op_r\ v_{ref} \mid \texttt{isnull}(v_{ref})
\end{array}
$$

Fig. 3: CrabIR: Region-based Intermediate Representation.

4 CrabIR: An Intermediate Representation for RBMM

In RBMM, every memory operation must explicitly indicate on which region the memory access is taking place. This motivates us to introduce a new intermediate representation called CrabIR. In this section, we describe the main aspects of CrabIR which is the input of the Crab abstract interpreter. In Sec. 5, we present its concrete semantics.

The simplified syntax[8] of CrabIR is shown in Fig. 3. A program P consists of a set of functions. A function consists of a name, zero or more input arguments, and a non-empty sequence of basic blocks. A basic block is denoted by a unique identifier label, containing zero or more statements S in three-address form. Each block must be terminated by either a `goto` or a `return` statement. The former is accompanied by one or more labels for each successor while the latter by zero or more output parameters of the function. Statements can only be one of these kinds: integers S_{int}, Booleans S_{bool}, and regions S_{rgn}. All statements are strongly typed. Integer, Boolean, reference, and region variables are denoted with symbols v_i, v_b, v_{ref}, v_{rgn}, respectively. Variables of any type are denoted by v. Scalar (non-region) variables are denoted by v_s. Integer variables are sized (i.e., of different bit-width). Control flow is modeled by `goto` and `assume` statements. The statement $v :=$ `havoc`() assigns non-deterministically any value allowed by v's type to v. Properties can only be defined by adding `sassert` statements. The syntax of S_{int} and S_{bool} is self explanatory. Integer and Boolean expressions are described by *aexp* and *bexp*.

We describe now informally the non-standard region statements. Detailed concrete semantics is given in Sec. 5. A key feature in CrabIR is that the language enforces that each reference is always associated uniquely to a region variable. Region variables must be initialized before used by calling `initrgn`. Reference variables can be created and destroyed by calling `makeref` and `freeref`, respectively. `loadref` reads the content of the reference v_{ref} within the region denoted by v_{rgn}, and assigns it to the scalar variable v_s. `storeref` writes the value of v_s in the memory address pointed by v_{ref} within the region v_{rgn}. `gepref` per-

[8] For instance, integer casts and casts between Boolean and integers are omitted.

forms pointer arithmetic. This is similar to the LLVM `getelementptr` instruction but it has been adapted to RBMM. `gepref` generates a new reference from adding an offset v_i (third input parameter) to the base reference v_{ref} (second input parameter). The region associated to the base reference must be provided (first input parameter). `gepref` has two output parameters: the new reference and its region. The key aspect of `gepref` is that it allows "switching" from one region to another as long as the references point to the same memory object. References can be compared to each other and to the null constant (b_{ref}). Finally, CrabIR allows to convert between references and integers (`inttoref` and `reftoint`). We show the translation of our example from Sec. 2 in Appendix A.

5 A Concrete Semantics for CrabIR

The goal of this section is twofold. First, it gives a formal meaning of the region statements in CrabIR by presenting a concrete interpreter. Second, it constitutes the basis for the structural abstractions presented in Sec. 6. By structural, we mean that each sub-component of a concrete state is separately abstracted.

Recall our memory model is *word-addressable* which means that all data location has the same size and all memory addresses are divisible by the *word size*, which is a fixed parameter in our model. Fig. 4 defines the semantic domain to express the concrete semantics of CrabIR. The symbols $\mathcal{V}_{\mathcal{B}}$, $\mathcal{V}_{\mathcal{I}}$, $\mathcal{V}_{\mathcal{R}ef}$, and $\mathcal{V}_{\mathcal{R}gn}$ denote the set of Boolean, integer, reference, and region variables such that $\mathcal{V}_{\mathcal{B}} \cap \mathcal{V}_{\mathcal{I}} \cap \mathcal{V}_{\mathcal{R}ef} \cap \mathcal{V}_{\mathcal{R}gn} = \emptyset$. Memory is partitioned into a finite, disjoint set of regions. Each region is accessible by a handle $v \in \mathcal{V}_{\mathcal{R}gn}$. An *address* is an integer. A *reference* is an address within a region. The data associated with a reference is called a cell. A cell can represent either another reference or an integer value. A *cell* is a pair of the form of $\langle b, o \rangle$. If a cell represents a reference then b is a base address and o is a numerical offset. A cell can be trivially converted to an address (cell2Addr in Appendix B, Fig. 13). The special null reference is encoded as $\langle 0, 0 \rangle$. If a cell represents an integer k then it is encoded as $\langle 0, k \rangle$[9]. Note that since memory is word-addressable, the size of a cell is always the word size, and thus, it is omitted from the concrete semantics. A *valid* program state $\sigma \in$ State is represented by the tuple: $\langle refEnv, rgnEnv, numEnv, nextAddr, rgnAddrs, alloc \rangle$

The environment $refEnv$ maps reference variables to cells. The content of a region is modeled by a map $rgnEnv$ from addresses to cells. $numEnv$ maps Boolean or integer variables to their values. The program state maintains the next available address ($nextAddr$), and for each region, the program state keeps track of all addresses owned by a region ($rgnAddrs$). The state also keeps track of all *allocated* memory objects ($alloc$) by keeping a list of pairs of: the base address and the past-the-end address of the object. $alloc$ is mostly used for error checking and for converting addresses to cells (addr2Cell in Appendix B, Fig. 13). We use the symbol σ_Ω to denote either a valid state $\sigma \in$ State or the error state Ω. The semantics for statements is given by the function $[\![.]\!]_\Omega(.) : S \mapsto \sigma_\Omega \mapsto \sigma_\Omega$:

[9] Note that the cell $\langle 0, 0 \rangle$ can either mean the null reference or the integer 0.

$\{a, nextAddr\}$	$\subseteq$ Address	$=$	$\mathbb{Z}$
c	$\in$ Cell	$=$	Address $\times \mathbb{Z}$
$refEnv$	$\in$ RefEnv	$=$	$\mathcal{V}_{\mathcal{R}ef} \mapsto$ Cell
$rgnEnv$	$\in$ RgnEnv	$=$	$\mathcal{V}_{\mathcal{R}gn} \mapsto$ (Cell $\mapsto$ Cell)
$numEnv$	$\in$ NumEnv	$=$	$(\mathcal{V}_{\mathcal{B}} \cup \mathcal{V}_{\mathcal{I}}) \mapsto \mathbb{Z}$
$rgnAddrs$	$\in$ RgnToAddrs	$=$	$\mathcal{V}_{\mathcal{R}gn} \mapsto 2^{\text{Address}}$
$\{objList, alloc\}$	$\subseteq$ MemObjList	$=$	List(Address $\times$ Address)
σ	$\in$ State	$=$	RefEnv $\times$ RgnEnv $\times$ NumEnv $\times$ Address $\times$ RgnToAddrs $\times$ MemObjList

Fig. 4: Semantic Domains.

$[\![rgn := \texttt{initrgn}()]\!](\sigma) \equiv$

$\texttt{match } \sigma \texttt{ with } (refEnv, rgnEnv, numEnv, nextAddr, rgnAddrs, alloc) \rightarrow$
  $\texttt{if isInitRegion}(\sigma, rgn) \texttt{ then } \sigma$
  $\texttt{else } (refEnv, rgnEnv, numEnv, nextAddr, rgnAddrs[rgn \mapsto \emptyset], alloc)$

$[\![ref := \texttt{makeref}(rgn, n)]\!](\sigma) \equiv$

$\texttt{match } \sigma \texttt{ with } (refEnv, rgnEnv, numEnv, nextAddr, rgnAddrs, alloc) \rightarrow$
  $\texttt{if } \neg \texttt{ isInitRegion}(\sigma, rgn) \texttt{ or } [\![n]\!](\sigma) \leq 0 \texttt{ then } \Omega$
  $\texttt{else}$
    $\texttt{let } \langle base, sz\rangle := \langle nextAddr, [\![n]\!](\sigma)\rangle \texttt{ in}$
    $\texttt{let } refEnv' = refEnv[ref \mapsto \langle base, 0\rangle] \texttt{ in}$
    $\texttt{let } nextAddr' = base + sz \texttt{ in}$
    $\texttt{let } rgnAddrs' = rgnAddrs[rgn \mapsto rgnAddrs(rgn) \cup \{base\}] \texttt{ in}$
    $\texttt{let } alloc' = \langle base, base + sz\rangle :: alloc \texttt{ in}$
    $(refEnv', rgnEnv, numEnv, nextAddr', rgnAddrs', alloc')$

$[\![\texttt{freeref}(rgn, ref)]\!](\sigma) \equiv$

$\texttt{match } \sigma \texttt{ with } (refEnv, rgnEnv, numEnv, nextAddr, rgnAddrs, alloc) \rightarrow$
  $\texttt{if } ref \notin \text{dom}(refEnv) \texttt{ or } \neg \text{ isInList}(refEnv(ref), alloc) \texttt{ then } \Omega$
  $\texttt{else}$
    $\texttt{match removeFromList}(refEnv(ref), alloc) \texttt{ with}$
      $\Omega \rightarrow \Omega$
      $alloc' \rightarrow (refEnv, rgnEnv, numEnv, nextAddr, rgnAddrs, alloc')$

$[\![(rgn_2, ref_2) := \texttt{gepref}(rgn_1, ref_1, n)]\!](\sigma) \equiv$

$\texttt{match } \sigma \texttt{ with } (refEnv, rgnEnv, numEnv, nextAddr, rgnAddrs, alloc) \rightarrow$
  $\texttt{if } \neg \text{ isInitRegion}(\sigma, rgn_1) \texttt{ or } \neg \text{ isInitRegion}(\sigma, rgn_2) \texttt{ or } ref_1 \notin \text{dom}(refEnv) \texttt{ then } \Omega$
  $\texttt{else}$
    $\texttt{let } \langle b, o\rangle = refEnv(ref_1) \texttt{ in}$
    $\texttt{let } c' = \langle b, o + [\![n]\!](\sigma)\rangle \texttt{ in}$
    $\texttt{let } refEnv' = refEnv[ref_2 \mapsto c'] \texttt{ in}$
    $\texttt{let } rgnAddrs' = rgnAddrs[rgn_2 \mapsto rgnAddrs(rgn_2) \cup \{\text{cell2Addr}(c')\}] \texttt{ in}$
    $(refEnv', rgnEnv, numEnv, nextAddr, rgnAddrs', alloc)$

Fig. 5: Concrete semantics for region-based statements (I).

$$[\![stmt]\!]_{\Omega}(\sigma) = \begin{cases} \Omega & \text{if } \sigma = \Omega \\ [\![stmt]\!](\sigma) & \text{otherwse} \end{cases}$$

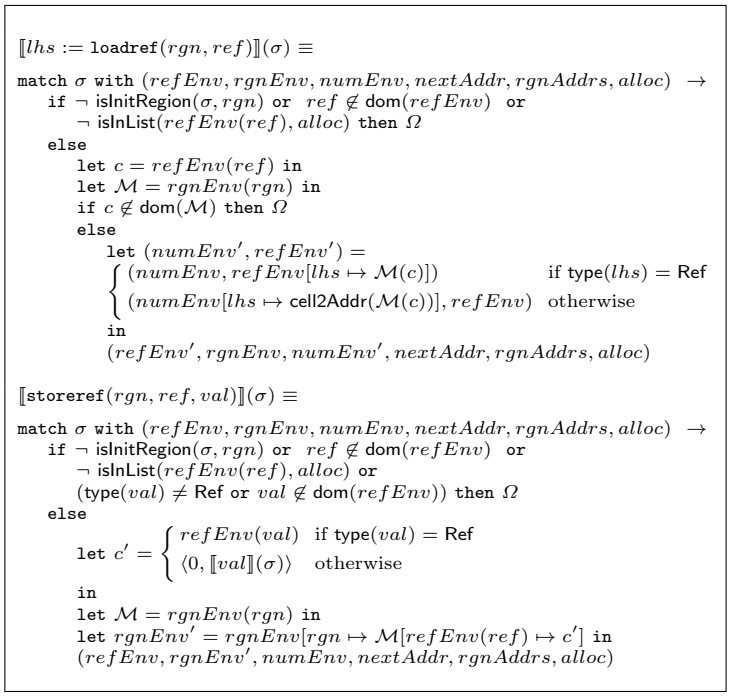


Fig. 6: Concrete semantics for region-based statements (II).

The definition of $[\![.]\!](.)$[10] for region statements is described in Figs. 5,6, and 15. Our semantics is similar to a standard word-level C with two differences: (1) it keeps separate the memory space of each region ($rgnEnv$ is indexed by a region), and (2) it keeps track of all addresses owned by each region ($rgnAddrs$). We omit the semantics of other constructs (e.g., `assume`, `goto`) but they are defined in the usual way. The initial concrete state[11] is $\langle \lambda ref.\bot, \lambda rgn.(\lambda a.\bot), \lambda n.\bot, 1, \lambda rgn.\bot, nil\rangle$.

Region initialization The statement `initrgn`() returns a new region variable. All regions must be initialized before they can be used. The program state is updated by setting the set of owned address by the region to empty. If a region is already initialized then input state is returned.

Allocation and deallocation $ref :=$ `makeref`(rgn, n) returns a reference ref pointing to the base address of a newly allocated memory object of size n. The concrete semantics creates a new cell associated to ref and models the fact a memory object has been allocated by adding it to $alloc$ and increasing $nextAddr$ by n. Moreover, the program state ($rgnAddrs$) records that ref is owned by the region rgn. `freeref`(rgn, ref) identifies the memory object associated to the

[10] We abuse notation and use $[\![.]\!](.)$ to evaluate integer and Boolean expressions defined as usual: $[\![c]\!](\sigma) = c$ for constant c, $[\![v]\!](\langle \_, \_, numEnv, \_, \_, \_\rangle) = numEnv(v)$ for variable v, $[\![e_1 + e_2]\!](\sigma) = [\![e_1]\!](\sigma) + [\![e_2]\!](\sigma)$ for expressions e_1 and e_2, etc.

[11] We use the notation $\lambda x.\bot$ to represent the undefined function.

reference ref and deletes the object from *alloc*. If the memory object was not allocated or already freed then it returns the error state.

Pointer arithmetic The semantics of $(rgn_2, ref_2) := \texttt{gepref}(rgn_1, ref_1, n)$ models pointer arithmetic in CrabIR. Similar to C pointer arithmetic, `gepref` does not access memory and returns a new reference ref_2 obtained by adding the offset n to the input reference ref_1. What differs from the standard C memory model is that both reference variables ref_1 and ref_2 are associated with their corresponding regions denoted by variables rgn_1 and rgn_2, respectively. Therefore, the program state ($rgnAddrs$) must update the set of owned addresses by rgn_2 by adding the address of ref_2.

Accessing memory Both `storeref` and `loadref` (Fig. 6) check first that the reference variable ref points to a valid memory object. If not then the error state is returned. $lhs := \texttt{loadref}(rgn, ref)$ reads the data associated to the ref's cell and updates either $numEnv$ or $rgnEnv$ depending on the type of lhs. $\texttt{storeref}(rgn, ref, val)$ modifies $rgnEnv$ by updating the data associated to ref's cell with val. Note that in both cases, the semantics is greatly simplified thanks to its word-level addressing.

Conversion between references and integers Since CrabIR is strongly typed, all type conversions must be done explicitly. The statements `reftoint` and `inttoref` allow to convert a reference to an integer, and vice-versa. The semantics of these statements is shown in Appendix C.

6 Region Abstract Domains

This section illustrates how our region-based semantics from Sec. 5 can be efficiently computed by applying well-known *address abstractions*. The goal is not to introduce new abstractions but instead, to demonstrate that existing abstractions can be easily adapted to RBMM while they can benefit from the aliasing restrictions provided by our memory model. In this section, we revisit two address abstractions: *smashing*, where all memory objects within a region are grouped into one summarized variable, and *recency* [1], where the abstraction distinguishes between the most recent object and the previous ones. These two abstractions are pictorially shown in Fig. 7.

6.1 Smashing Region Abstract Domain

This domain uses the basic abstract domains: $\mathsf{Bool} = \{\bot_{\mathsf{Bool}}, \mathsf{True}, \mathsf{False}, \mathsf{All}\}$ where $\bot_{\mathsf{Bool}} \sqsubseteq \mathsf{True}$, $\bot_{\mathsf{Bool}} \sqsubseteq \mathsf{False}$, $\mathsf{True} \sqsubseteq \mathsf{All}$, $\mathsf{False} \sqsubseteq \mathsf{All}$; $\mathsf{SmallRange} = \{\bot_{\mathsf{SmallRange}}, 0, 1, 0\text{–}1, 1^+, 0^+\}$ where $\bot_{\mathsf{SmallRange}} \sqsubseteq 0$, $\bot_{\mathsf{SmallRange}} \sqsubseteq 1$, $0 \sqsubseteq 0\text{–}1$, $1 \sqsubseteq 0\text{–}1$, $1 \sqsubseteq 1^+$, $0\text{–}1 \sqsubseteq 0^+$, $1^+ \sqsubseteq 0^+$; and Base, a numerical domain. As usual, we lift Bool and $\mathsf{SmallRange}$ to environment domains, $\mathsf{BoolEnv}$, and $\mathsf{SmallRangeEnv}$, respectively. These environments map variables to abstract values. If a variable is not found in the map

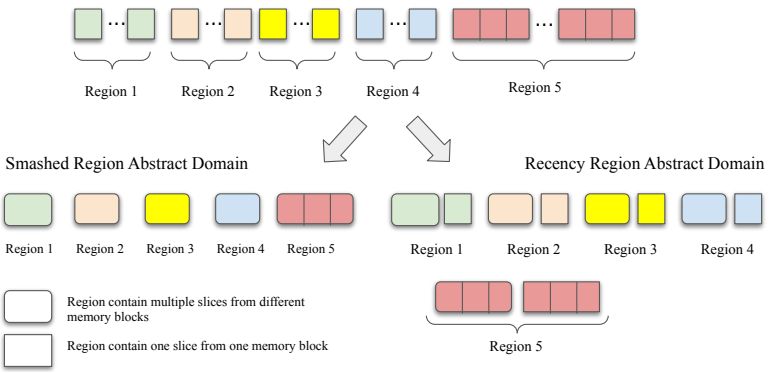


Fig. 7: Examples of Simple Region Abstractions.

then the corresponding top element (All or 0^+) is returned. An abstract state $\sigma^\# \in \mathsf{AState}$ is defined as a triple where $init \in \mathsf{BoolEnv}$ keeps track whether a region has been initialized, $countAddr \in \mathsf{SmallRangeEnv}$ keeps track of the number of owned addresses by a region, and $base \in \mathsf{Base}$, is the base domain.

Abstract operations. The binary abstract operators: inclusion, join, meet, widening, and narrowing are component-wise. Most relevant abstract transformers for the region-based statements are given in Fig. 8. For simplicity, the abstract transformers assume that the input abstract state is not $\bot$ (bottom). Also for simplicity, we omit the abstract transformer for `freeref`. Although conceptually simple, its transformer is a bit more elaborate because the abstract state needs to relate which regions may contain fields of the same memory object. This is because when a reference is being deallocating the whole memory object to which the reference points to must be freed. Our implementation models `freeref`. Most abstract transformers in Fig. 8 are self-explanatory. The idea is to count the number of possible references within each region. If more than one reference is possible then memory writes/reads are modeled as weak updates/reads, otherwise as a strong updates/reads. For that, the abstract states uses $countAddr$. As usual, we assume that the base abstract domain supports the *expand* operation (e.g., [12]) such that $base' = \texttt{expand}(base, v_1, v_2)$ copies all the relationships between v_1 with other variables into a fresh, unconstrained v_2 without relate v_1 and v_2. This operation is used to perform "weak reads". Note that for the abstract domain to perform at least one strong update, it needs the counter of addresses per region to be set to the abstract value 0. This takes place in the abstract transformer of `initrgn`.

6.2 Recency-Based Region Abstract Domain

We show now how recency abstraction [1] can be adapted to our region memory model. For each region variable $v \in \mathcal{V}_{\mathcal{R}gn}$, we introduce two *ghost variables* $v^r \in \mathcal{V}_{\mathcal{R}gn}$ and $v^o \in \mathcal{V}_{\mathcal{R}gn}$ where the former represents the field of the most recent allocated memory object within region v and the latter represents the same field

$[\![rgn := \texttt{initrgn}()]\!]^{\mathsf{Smash}}(\sigma^{\#}) \equiv$

$\texttt{match } \sigma^{\#} \texttt{ with } (init, countAddr, base) \rightarrow$
  $\texttt{if } countAddr(rgn) \sqsubseteq_{\mathsf{SmallRange}} 1^{+} \texttt{ then } \sigma^{\#}$
  $\texttt{else}$
    $\texttt{let } init' = init[rgn \mapsto \mathsf{False}] \texttt{ in}$
    $\texttt{let } countAddr' = countAddr[rgn \mapsto 0] \texttt{ in}$
    $(init', countAddr', base)$

$[\![ref := \texttt{makeref}(rgn, n)]\!]^{\mathsf{Smash}}(\sigma^{\#}) \equiv$

$\texttt{match } \sigma^{\#} \texttt{ with } (init, countAddr, base) \rightarrow$
  $\texttt{let } countAddr' = countAddr[rgn \mapsto countAddr(rgn) +^{\mathsf{SmallRange}} 1] \texttt{ in}$
  $(init, countAddr', base)$

$[\![(rgn_2, ref_2) := \texttt{gepref}(rgn_1, ref_1, n)]\!]^{\mathsf{Smash}}(\sigma^{\#}) \equiv$

$\texttt{match } \sigma^{\#} \texttt{ with } (init, countAddr, base) \rightarrow$
  $\texttt{if } rgn_1 \neq rgn_2 \texttt{ or } [\![ref_2 \neq ref_1 + n]\!]^{\mathsf{Base}}(base) \neq \bot_{\mathsf{Base}} \texttt{ then}$
    $\texttt{let } countAddr' = countAddr[rgn_2 \mapsto countAddr(rgn_2) +^{\mathsf{SmallRange}} 1] \texttt{ in}$
  $\texttt{else}$
    $\texttt{let } countAddr' = countAddr \texttt{ in}$
  $\texttt{let } base' = [\![ref_2 := ref_1 + n]\!]^{\mathsf{Base}}(base) \texttt{ in}$
  $(init, countAddr', base')$

$[\![lhs := \texttt{loadref}(rgn, ref)]\!]^{\mathsf{Smash}}(\sigma^{\#}) \equiv$

$\texttt{match } \sigma^{\#} \texttt{ with } (init, countAddr, base) \rightarrow$
  $\texttt{if } [\![ref \neq 0]\!]^{\mathsf{Base}}(base) = \bot_{\mathsf{Base}} \texttt{ or } init(rgn) = \mathsf{False} \texttt{ then } \bot$
  $\texttt{else}$
    $\texttt{if } countAddr(rgn) \sqsubseteq_{\mathsf{SmallRange}} 0\text{–}1 \texttt{ then}$
      $\texttt{let } base' = [\![lhs := rgn]\!]^{\mathsf{Base}}(base) \texttt{ in}$
    $\texttt{else}$
      $\texttt{let } base' = \mathsf{expand}(base, rgn, rgn_{fresh}) \texttt{ in}$
    $(init, countAddr, [\![lhs := rgn_{fresh}]\!]^{\mathsf{Base}}(base'))$

$[\![\texttt{storeref}(rgn, ref, val)]\!]^{\mathsf{Smash}}(\sigma^{\#}) \equiv$

$\texttt{match } \sigma^{\#} \texttt{ with } (init, countAddr, base) \rightarrow$
  $\texttt{if } [\![ref \neq 0]\!]^{\mathsf{Base}}(base) = \bot_{\mathsf{Base}} \texttt{ then } \bot$
  $\texttt{else}$
    $\texttt{if } init(rng) = \mathsf{False} \text{ or } countAddr(rgn) \sqsubseteq_{\mathsf{SmallRange}} 0\text{–}1 \texttt{ then}$
      $\texttt{let } base' = [\![rgn := val]\!]^{\mathsf{Base}}(base) \texttt{ in}$
    $\texttt{else}$
      $\texttt{let } base' = base \sqcup_{base} [\![rgn := val]\!]^{\mathsf{Base}}(base) \texttt{ in}$
    $(init[rgn \mapsto \mathsf{All}], countAddr, base')$

Fig. 8: Smashing Abstraction for region-based statements.

but from the older memory objects allocated within same region. Moreover, we introduce a third ghost reference variable $v^l \in \mathcal{V}_{\mathcal{R}ef}$ that remembers the address of the last allocated object. The abstract transformers for allocation and accessing memory statements are given in Fig. 9.

The most interesting case is `makeref`. When a new reference is allocated, the old region rgn^o must be joined with the most recent region rgn^r. Then, rgn^r is reset by making the region uninitialized and setting its counter of addresses to either 0 or 1 (0 if we assume that the allocation cannot fail). Moreover, the abstract state remembers the allocated reference in rgn^l. The rest of operations compare the corresponding references with rgn^l to determine whether the operation can be performed on the most recent region rgn^r or the old ones rgn^o.

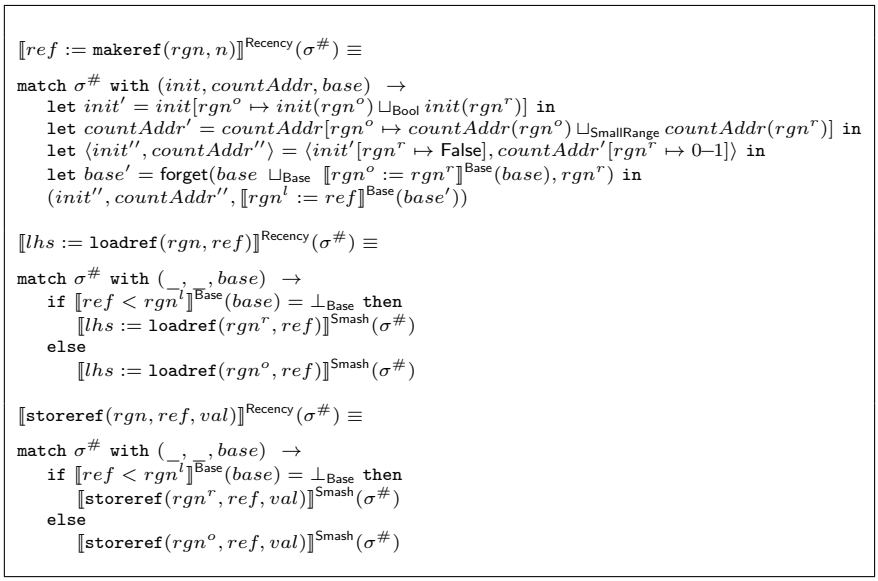


Fig. 9: Recency Abstraction for allocation and accessing memory statements.

7 Experimental Evaluation

To evaluate our RBMM, we aim to answer the following research questions:

RQ1: Are the assumptions of RBMM realistic for modern C projects?
RQ2: Can RBMM and the smashing domain analyze interesting properties?
RQ3: Is RBMM and the smashing domain efficient for realistic projects?

Implementation. We replaced the old IR in Crab with CrabIR (Sec. 4), modified the interface of the abstract domains to support CrabIR, and implemented the smashing region domain (Sec. 6.1). These changes are now part of Crab[12]. The implementation relaxes word-level addressability assumption as long as each memory read accesses the same number of bytes last written.

From LLVM to CrabIR. Crab is a library for performing abstract interpretation of CrabIR programs, and thus, it does not support C or LLVM directly. The translation from LLVM to CrabIR is performed by Clam, a LLVM frontend for Crab. The main architecture of our C analyzer based on LLVM is shown in Fig. 10. [13]

Clam[14] runs a whole-program pointer analysis, called SeaDsa [14, 19], on the LLVM bitcode and builds a *Memory SSA form* representation of the program [7]. In Memory SSA, each memory write is given a unique version of the

[12] Publicly available at `https://github.com/seahorn/crab`
[13] SeaDsa, Clam, and Crab are also integrated into the SeaHorn verification framework [13].
[14] Publicly available at `https://github.com/seahorn/clam`

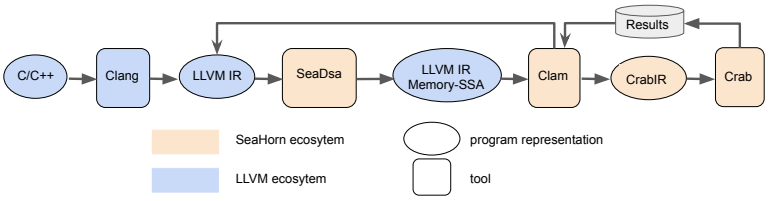


Fig. 10: Abstract Interpreter of LLVM programs.

					All Memory			RBMM-compl.		
Program	KLOC	# I	Time(s)	Trivial #A	#P	#A	%P	#P	#A	%P
`bftpd`	4	11K	4	3 573	134	357	38	31	59	52
`brotli`	31	221K	101	9 470	7 741	14 945	52	535	1 221	43
`brotli-mod`	—	210K	860	8 328	3 937	7 843	51	1 339	2 128	63
`curl`	85	12K	152	840	820	1 601	52	92	125	77
`thttpd`	8	8K	52	1 480	661	1 326	50	149	258	58
`vsftpd`	16	16K	330	3 324	1 021	1 665	62	42	98	43

Table 1: #I is the number of instructions in the LLVM program. Time(s) is the time in seconds of the translation to CrabIR, analysis, and checking phases. Trivial #A is the number of assertions proven by the Clam preprocessor (without using Crab). All Memory considers all dereferenced pointers while RBMM-compliant only checks a pointer if the analysis can ensure that RBMM assumptions hold. #P is the number of proved assertions by the analysis, #A is the number of checked assertions (excluding Trivial #A), and %P is $\lceil \frac{\#P}{\#A} \times 100 \rceil$.

heap and reads can only refer to one version. We use SeaDsa to partition the heap into many disjoint sub-heaps. The translation from Memory SSA to CrabIR maps heap versions to different region variables.

Benchmarks and Setup. We selected popular C projects that focus on portability and, therefore, do not rely on specific architectures. The projects are `bftpd`, `brotli`, `curl`, `thttpd`, and `vsftpd`. We implemented an LLVM instrumentation that adds assertions in CrabIR to check that each pointer is not null before being dereferenced. As the abstract domain, we use the smashing region domain from Sec. 6.1 with a simple reduced product of the sign and constant domain. We also tried the interval domain and differences were small. All artifacts to reproduce Table. 1 are publicly available at `http://doi.org/10.5281/zenodo.5129227`.

Results. We evaluated our analysis on the selected benchmarks checking for null-dereferences. We present the results in Table. 1 to answer each question. **RQ1: Are the assumptions of RBMM realistic for modern C projects?**. To answer this question, we look at the columns #A in All Memory and RBMM-

compliant, respectively. These numbers show that on average only nearly 10% of all checked pointers satisfy statically the assumptions of RBMM. For better understanding, we manually inspected our bigger program, brotli — a compression library. [15] By default, brotli is compiled to the host architecture, however, this can be disabled by a compile-time flag. The portable configuration of brotli complies with the assumptions of RBMM, and we use it for the rest of the experiments. Next, we saw that our pointer analysis, SeaDsa, is not sufficiently precise. Luckily, brotli is designed to be very customizable. For example, it allows different memory managers in addition to the standard `malloc`, provides different layers of caches for small objects, and uses more than 10 different hash function implementations. We simplified the code (brotli-mod) by either specialising some behaviour (e.g., allocating small objects on the heap, using default allocator, choosing fewer hash functions) and by abstracting away some hard-to-analyze functions (they can be analyzed by other less scalable but more precise approaches). With these simplifications, SeaDsa was able to ensure that 30% of memory accesses are compliant with RBMM. Our takeaway is that modern C projects like brotli are designed to be portable and, thus, they are very likely to satisfy our RBMM assumptions. We also believe that it is possible to statically prove compliance with user help and/or using more precise pointer analyses.

RQ2: Can RBMM and the smashing domain analyze interesting properties? The answer to this question is mixed. The results show that such a simple analysis can establish about 50-60% of all the non-trivial memory operations. Refactoring the code to be more analyzable, as we did in `brotli`, can have a significant positive impact. Unfortunately, there is no ground truth – we do not know how many memory accesses can be established safe automatically. We have tried number of existing tools, including IKOS and Infer, but none produce good quality results (either because of crashes or because of automatic hiding of warnings). In our opinion, the results show RBMM is a great building block to combine with more intricate abstract domain (i.e., not simple smashing). This is an avenue for future work.

RQ3: Is RBMM and the smashing domain efficient for realistic projects? The results show that we can analyze a non-trivial project within 15 minutes, which is quite reasonable. Interestingly, the analysis time increases as precision increases (see the difference between `brotli` and `brotli-mod`. This is because more precision implies that the analysis does not generate "top" as often, thus, requiring more time to converge. It is interesting to see whether more aggressive widening strategy or less aggressive inlining can improve performance.

8 Related Work

Our approach can be seen as a form of abstract compilation (e.g., logic programs [27] and array-manipulating programs [9]) but it focuses on the pointer semantics of C. Our RBMM is inspired by similar memory models used by

[15] `https://github.com/google/brotli`

deductive verification tools such as Smack [22, 23], Cascade [26], SeaHorn [13], FramaC/Jessie [10, 21], Cadeceus [11] and [4]. Our work focuses on Abstract Interpretation and thus, it formalizes both its concrete and abstract semantics.

There is a variety of impressive C abstract interpreters such as Astrée [2], EVA [3], IKOS [5], Infer [6], MemCAD [15], Mopsa [18, 17], and Verasco [16]. RBMM restricts further the memory models used by these tools with the goal of scaling on general-purpose C projects while proving interesting properties.

9 Conclusions

We have proposed a new region-based memory model (RBMM) suitable for static analysis based on abstract interpretation. We have formalized the semantics of the memory model and shown how to adapt existing memory abstractions. Our evaluation suggests that the assumptions of our memory model are realistic for modern C projects. However, more work needs to be done to fully prove absence of memory violations in those programs. One advantage of RBMM is its compatibility with standard C memory model and thus, more precise memory abstractions such as [20] and [25] can be used.

References

1. G. Balakrishnan and T. W. Reps. Recency-abstraction for heap-allocated storage. In *SAS*, pages 221–239, 2006.
2. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003.
3. S. Blazy, D. Bühler, and B. Yakobowski. Structuring abstract interpreters through state and value abstractions. In *VMCAI*, pages 112–130, 2017.
4. Q. Bouillaguet, F. Bobot, M. Sighireanu, and B. Yakobowski. Exploiting pointer analysis in memory models for deductive verification. In *VMCAI*, pages 160–182, 2019.
5. G. Brat, J. A. Navas, N. Shi, and A. Venet. IKOS: A framework for static analysis based on abstract interpretation. In *SEFM*, pages 271–277, 2014.
6. C. Calcagno and D. Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NFM*, pages 459–465, 2011.
7. F. C. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *CC*, pages 253–267, 1996.
8. C. L. Conway, D. Dams, K. S. Namjoshi, and C. W. Barrett. Pointer analysis, conditional soundness, and proving the absence of errors. In *SAS*, volume 5079, pages 62–77, 2008.
9. J. R. M. Cornish, G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Analyzing array manipulating programs by program transformation. In *LOPSTR*, pages 3–20, 2014.
10. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. In *SEFM*, pages 233–247, 2012.
11. J. Filliâtre and C. Marché. Multi-prover verification of C programs. In *ICFEM*, volume 3308, pages 15–29, 2004.

12. D. Gopan. *Numeric Program Analysis Techniques with Applications to Array Analysis and Library Summarization*. PhD thesis, University of Wisconsin, 2007.
13. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The seahorn verification framework. In *CAV*, pages 343–361, 2015.
14. A. Gurfinkel and J. A. Navas. A context-sensitive memory model for verification of C/C++ programs. In *SAS*, pages 148–168, 2017.
15. H. Illous, M. Lemerre, and X. Rival. A relational shape abstract domain. In *NFM*, pages 212–229, 2017.
16. J. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified C static analyzer. In *POPL*, pages 247–259, 2015.
17. M. Journault, A. Miné, R. Monat, and A. Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyzer. In *VSTTE*, pages 1–18, 2019.
18. M. Journault, A. Miné, and A. Ouadjaout. Modular static analysis of string manipulations in C programs. In *SAS*, pages 243–262, 2018.
19. J. Kuderski, J. A. Navas, and A. Gurfinkel. Unification-based pointer analysis without oversharing. In *FMCAD*, pages 37–45, 2019.
20. A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *LCTES*, pages 54–63, 2006.
21. Y. Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, 2009.
22. Z. Rakamaric and M. Emmi. SMACK: decoupling source language details from verifier implementations. In A. Biere and R. Bloem, editors, *CAV*, pages 106–113, 2014.
23. Z. Rakamaric and A. J. Hu. A scalable memory model for low-level code. In *VMCAI*, pages 290–304, 2009.
24. Y. Sui and J. Xue. SVF: interprocedural static value-flow analysis in LLVM. In *CC*, pages 265–266, 2016.
25. A. Venet. A scalable nonuniform pointer analysis for embedded programs. In *SAS*, pages 149–164, 2004.
26. W. Wang, C. Barrett, and T. Wies. Cascade 2.0. In *VMCAI*, pages 142–160, 2014.
27. R. Warren, M. V. Hermenegildo, and S. K. Debray. On the practicality of global flow analysis of logic programs. In *ICLP*, pages 684–699, 1988.

```
(ref,region(ref),region(int32),region(int32),region(ref),region(int32))
declare mk_list(n:int32,data_sz:int32,val:int32)
@entry:
  initrgn(R1:region(ref));
  initrgn(R2:region(int32));
  initrgn(R3:region(int32));
  initrgn(R4:region(ref));
  initrgn(R5:region(int32));
  havoc(l:ref)
  assume(l == NULL_REF);
  i = 0;
  goto @bb_1;
@bb_1:
  goto @bb_2,@bb_4;
@bb_2:
  assume(i + 1 <= n);
  goto @bb_3;
@bb_3:
  n = 16;
  tmp:ref = makeref(R1, n);
  assume(tmp > NULL_REF);
  (data:ref,R5) = call init(data_sz,val);
  (R1,tmp#data:ref) = gepref(R1,tmp);
  // update tmp->data
  storeref(R1,tmp#data,data);
  havoc(used:int32);
  assume(used >= 0);
  assume(used <= data_sz -1);
  // update tmp->len
  (R2,tmp#len:ref) = gepref(R1,tmp + 4);
  storeref(R2,tmp#len,used);
  // update tmp->cap
  (R3,tmp#cap:ref) = gepref(R1,tmp + 8);
  storeref(R3,tmp#cap,data_sz);
  // update tmp->n
  (R4,tmp#n:ref) = gepref(R1,tmp + 12);
  storeref(R4,tmp#n,l);
  // l = tmp
  (R1,l) = gepref(R1,tmp);
  i = i+1;
  goto @bb_1;
@bb_4:
  assume(n <= i);
  goto @exit;
@exit:
  return (l,R1,R2,R3,R4,R5);
```

Fig. 11: Simplified version of CrabIR code for `mk_list` function.

A Motivating Example from Sec. 2 in CrabIR

In this section, we show the translation of our motivating example from Sec. 2 to CrabIR. For readability purposes, variable names have been renamed to match those names used in the C program. We have also omitted types when they can be inferred from the context. We show two functions: `mk_list` in Fig. 11 and `main` in Fig. 12. The function `init` is translated in a very similar manner.

```
int32 declare main()
@entry:
  havoc(list_sz:int32)
  assume(list_sz >= 1);
  havoc(data_sz:int32)
  assume(data_sz >= 1);
  havoc(val:int32)
  (node:ref,R1:region(ref),R2:region(int32),
   R3:region(int32),R4:region(ref),R5:region(int32)) =
  call mk_list(list_sz:int32,data_sz:int32,val:int32);
  goto @bb_1;
@bb_1:
  goto @bb_2, @bb_9;
@bb_2:
  assume(node > NULL_REF);
  goto @bb_3;
@bb_3:
  (R1,node#data:ref)  = gepref(R1,node);
  tmp1:ref  = loadref(R1,node#data);
  sassert(tmp1 > NULL_REF);
  (R2,node#len:ref)  = gepref(R1,node + 4);
  tmp2:int32  = loadref(R2,node#len);
  sassert(tmp2 <= data_sz-1);
  (R3,node#cap:ref)  = gepref(R1,node + 8);
  tmp3:int32  = loadref(R3,node#cap);
  sassert(tmp2 <= tmp3);
  i = 0;
  goto @bb_4;
@bb_4:
  goto @bb_5,@bb_7;
@bb_5:
  assume(i <= data_sz-1);
  goto @bb_6;
@bb_6:
  tmp4:ref  = loadref(R1,node#data);
  (R5,tmp5:ref)  = gepref(R5,tmp4 + 4*i);
  tmp6:int32  = loadref(R5,tmp5);
  sassert(tmp6 == val);
  i = i+1;
  goto @bb_4;
@bb_7:
  assume(data_sz <= i);
  goto @bb_8;
@bb_8:
  (R4,node#n:ref)  = gepref(R1,node + 12);
  tmp7:ref  = loadref(R4,node#n);
  (R1,node)  = gepref(R1,tmp7);
  goto @bb_1;
@bb_9:
  assume(node == NULL_REF);
  goto @exit;
@exit:
  return 0;
```

Fig. 12: Simplified version of CrabIR code for `main`.

B Auxiliary Functions for Concrete Semantics

```
cell2Addr(c)
   match c with ⟨base, o⟩ →
      base + o

addr2Cell(a, objList)
   match objList with
      nil → ⟨0, 0⟩
      ⟨start, end⟩ :: tail →
         if (start ≤ a and a < end) then ⟨start, a − start⟩
         else addr2Cell(a, tail)
```

Fig. 13: Conversion between Addresses and Cells.

```
isInitRegion(σ, rgn)
   match σ with (_, _, _, rgnAddrs, _) →
      rgn ∈ dom(rgnAddrs)

isInList(⟨b, o⟩, objList)
   addr2Cell(cell2Addr(⟨b, o⟩), objList) ≠ ⟨0, 0⟩

removeFromList(c, objList)
   let a := cell2Addr(c) in
   match objList with
      nil → Ω
      ⟨start, end⟩ :: tail →
         if (start ≤ a and a < end) then
            tail
         else match removeFromList(c, tail) with
            Ω → Ω
            objList′ → ⟨start, end⟩ :: objList′
```

Fig. 14: Helpers for Concrete Semantics.

C Concrete Semantics of `reftoint` and `inttoref`

The concrete semantics of `reftoint` and `inttoref` is shown in Fig. 15. It is straightforward by delegating to the functions cell2Addr and addr2Cell. The most interesting part occurs in ref := `inttoref`(rgn, x) where the program state ($rgnAddrs$) must add the new reference ref to the set of owned addresses by the region rgn.

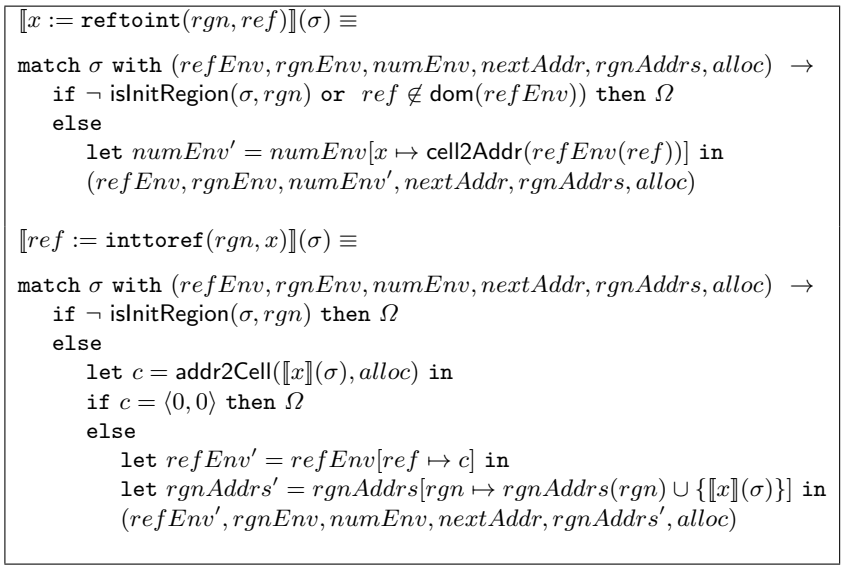


Fig. 15: Concrete Semantics for `reftoint` and `inttoref`.