Partitioned Memory Models for Program Analysis

Wei Wang[1], Clark Barrett[2], and Thomas Wies[1]

[1] New York University
[2] Stanford University

Abstract. Scalability is a key challenge in static analysis. For imperative languages like C, the approach taken for modeling memory can play a significant role in scalability. In this paper, we explore a family of memory models called *partitioned memory models* which divide memory up based on the results of a points-to analysis. We review Steensgaard's original and field-sensitive points-to analyses as well as Data Structure Analysis (DSA), and introduce a new *cell-based* points-to analysis which more precisely handles heap data structures and type-unsafe operations like pointer arithmetic and pointer casting. We give experimental results on benchmarks from the software verification competition using the program verification framework in Cascade. We show that a partitioned memory model using our cell-based points-to analysis outperforms models using other analyses.

1 Introduction

Solvers for Satisfiability Modulo Theories (SMT) are widely used as back ends for program analysis and verification tools. In a typical application, portions of a program's source code together with one or more desired properties are translated into formulas which are then checked for satisfiability by an SMT solver. A key challenge in many of these applications is scalability: for larger programs, the solver often fails to report an answer within a reasonable amount of time because the generated formula is too complex. Thus, one key objective in program analysis and verification research is finding ways to reduce the complexity of SMT formulas arising from program analysis.

For imperative languages like C, the modeling of memory can play a significant role in formula complexity. For example, a *flat model* of memory represents all of memory as a single array of bytes. This model is simple and precise and can soundly and accurately represent type-unsafe constructs and operations like unions, pointer arithmetic, and casts. On the other hand, the flat model implicitly assumes that any two symbolic memory locations could alias or overlap, even when it is known statically that such aliasing is impossible. Introducing *disjointness* constraints for every pair of non-overlapping locations leads to a quadratic blow-up in formula size, quickly becoming a bottleneck for scalability.

The *Burstall model* [6] is a well-known alternative that addresses this by having a separate array for each distinct type in the language. It is based on the assumption that pointers with different types never alias, eliminating the need for disjointness constraints between such pointers. However, it cannot model many type-unsafe operations, making it unsuitable for languages like C.

This paper discusses a family of memory models, which we call *partitioned memory models*, whose goal is to provide much of the efficiency of the Burstall model without losing the accuracy of the flat model. These models rely on a points-to analysis to determine a conservative approximation of which areas of memory may alias or overlap. The memory is then partitioned into distinct arrays for each of these areas.

When deciding which points-to analysis performs best in this context, there are two key attributes to consider: (1) the precision of the analysis (i.e. how many alias groups are generated); and (2) the ability to track the access size of objects in the alias group. The first is important because more alias groups means more partitions and fewer disjointness constraints. The second is important because if we know that all objects in the group are of a certain access size, then we can model the corresponding partition as an array whose elements are that size. After a review of existing field-sensitive points-to analyses (including Steensgaard's original analysis [27], Steensgaard's field-sensitive analysis [26], and Data Structure Analysis (DSA) [18]), we introduce a new *cell-based* points-to analysis and show that it improves on both of these attributes when compared to existing analyses. This is supported by experiments using the Cascade verification platform [29].

2 Memory Models for C Program Analysis

Consider the C code in Fig. 1. We will look at how to model the code using the flat memory model, the Burstall memory model, and the partitioned memory models.

```
int a;

void foo() {
  int *b = &a;
  *b = 0xFFF;
  char *c = (char *) b;
  *c = 0x0;
  assert(a != 0xFFF);
}
```

Fig. 1. Code with unsafe pointer cast.

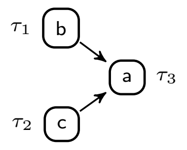

Fig. 2. The points-to graph of foo. Each τ_i represents a distinct alias group.

Flat model. In the flat model, a single array of bytes is used to track all memory operations, and each program variable is modeled as the content of some address in memory. Suppose M is the memory array, a is the *location* in M which stores the value of the variable a, and b is the *location* in M which stores the value of the variable b. We could then model the first two lines of foo in Fig. 1 (following SMT-LIB syntax [3]) as follows:

```
(assert (= M1 (store M b a))                ; M[b] := a
(assert (= M2 (store M1 (select M1 b) #xfff)) ; M[M[b]] := 0xfff
```

This is typical of the flat model: each program statement layers another store on top of the current model of memory. As a result, the depth of nested stores can get very large. Also, note that C guarantees that the memory regions of any two variables are non-overlapping. But the flat model must explicitly model this using an assumption on each pair of variables. This can be done with the following disjointness predicate, where p and q denote locations of variables, and $size(p)$ is the size of the memory region starting at location p:

$$disjoint(p, q) \equiv p + size(p) \leq q \vee q + size(q) \leq p.$$

For the code in Fig. 1, the required assumption is: $disjoint(a, b) \wedge disjoint(a, c) \wedge disjoint(c, b)$. Deeply nested stores and the need for such disjointness assertions severely limit the scalability of the flat model.

Burstall model. In the Burstall model, memory is split into several arrays based on the *type* of the data being stored. In Fig. 1, there are four different types of data, so the model would use four arrays: M_{int}, M_{char}, M_{int*} and M_{char*}. In this model, a is a location in M_{int}, b is a location in M_{int*}, and c is a location in M_{char*}. Disjointness is guaranteed implicitly by the distinctness of the arrays. The depth of nested stores is also limited to the number of stores to locations having the same type rather than to the number of total stores to memory. Both of these features greatly improve performance. However, the model fails to prove the assertion in Fig. 1. The reason is that the assumption that pointers to different types never alias is incorrect for type-unsafe languages like C.

Partitioned models. In the partitioned memory model, memory is divided into regions based on information acquired by running a points-to analysis.[3] The result of a standard points-to analysis is a points-to graph whose vertices are sets of program locations called alias groups. An edge from an alias group τ_1 to an alias group τ_2 indicates that dereferencing a location in τ_1 gives a location in τ_2. As an example, the points-to graph for the code in Fig. 1 is shown in Fig. 2. There are three alias groups identified: one for each of the variables `a`, `b`, and `c`. We can thus store the values for these variables in three different memory arrays (making their disjointness trivial). Note that according to the points-to graph, a dereference of either `b` or `c` must be modeled using the array containing the location of `a`, meaning that the model is sufficiently precise to prove the assertion. Like the Burstall model, the partitioned model divides memory into multiple arrays, reducing the burden on the solver. However, it does this in a way that is sound with respect to type-unsafe behavior.

The points-to analysis employed by the partitioned memory model must derive points-to relations that are also functions. That is, each node must have only a single successor in the points-to graph. This is because the successor determines the array to use when modeling a pointer dereference. The unification-based analyses proposed by Steensgaard [27] naturally guarantee this property.

[3] A formal presentation of the semantics of the partitioned memory model is presented in [28].

However, inclusion-based analyses proposed by Andersen [1] and others violate it. For this reason, we focus on Steensgaard's method and its variants.

Partitioned memory models that rely on unification-based points-to analyses are already used in a number of verification tools. For example, SMACK [24] leverages data structure analysis (DSA) [18], a field-sensitive and context-sensitive points-to analysis, to divide the memory. SeaHorn [13] uses a simpler context-insensitive variant of DSA. CBMC [7] and ESBMC [20] also use may-points-to information to refine their memory models. These tools illustrate the practical value of partitioned memory models. Note that in each case, the efficiency of the memory model depends heavily on the precision of the underlying points-to analysis. The more precise the analysis, the more partitions there are, which means fewer disjointness constraints and shallower array terms, both of which reduce the size and difficulty of the resulting SMT formulas.

3 Field-sensitive Points-to Analyses

Field-sensitivity is one dimension that measures the precision of points-to analyses. We say a pointer analysis is field-sensitive if it tracks individual record fields with a set of unique locations. That is, a store to one field does not affect other fields. In this sense, a field-sensitive analysis is much more precise than a field-insensitive one. Unfortunately, field-sensitivity is complicated by the weak type system of the C language. With the presence of union types, pointer casts, and pointer arithmetic, fields may not be guaranteed to be disjoint. The original Steensgaard's points-to analysis treats fields as a single alias group, while his field-sensitive analysis [26] improves the precision by separating fields into distinct alias groups only if the fields do not participate in any type-unsafe operations. Yong *et al.* [30] also propose to collapse fields upon detecting a field access through a pointer whose type does not match the declared type of the field. However, none of these analyses are able to distinguish fields on heap data structures (dynamic memory allocations).

Lattner *et al.* [18] present a data structure based points-to analysis (DSA), a unification-based, field-sensitive, and context-sensitive analysis. DSA explicitly tracks the type information and data layout of heap data structures and performs a conservative but safe field-sensitive analysis. The other key feature of DSA is that two objects allocated at the same location in a function called from different call sites are distinguished. It is in this sense that the analysis is context-sensitive. This feature, however, is orthogonal to our focus in this paper.

DSA computes a Data Structure Graph (DS graph) to explicitly model the heap. A DS graph consists of a set of nodes (DS nodes) representing memory objects and a set of points-to edges. Each DS node tracks the type information and data layout of the represented objects. If two DS nodes or their inner fields may be pointed to by the same pointer, they are merged together. When two DS nodes are merged, their type information and data layouts are also merged if *compatible*; otherwise, both nodes are *collapsed*, meaning that all fields are combined into a single alias group. To illustrate the DS node merging and collapsing process, we use the code in Fig. 3 as a running example (for simplicity, we assume pointers are 32 bits in this example).

```
typedef struct list {
  struct list *prev, *next;
  int32 data;
} list;

void bar(int32 undef, list **p) {
  list *k = malloc(sizeof(list));
  k->data = 1;
  p = undef < 0 ?
        &k->prev : &k->next;
}
```

```
void foo(int32 undef, list **p) {
  list *k1 = malloc(sizeof(list));
  list *k2 = malloc(sizeof(list));
  k1->data = 1; k2->data = 2;
  p = undef < 0 ?
        &k1->next : &k2->prev;
}
```

Fig. 3. Code for running example.

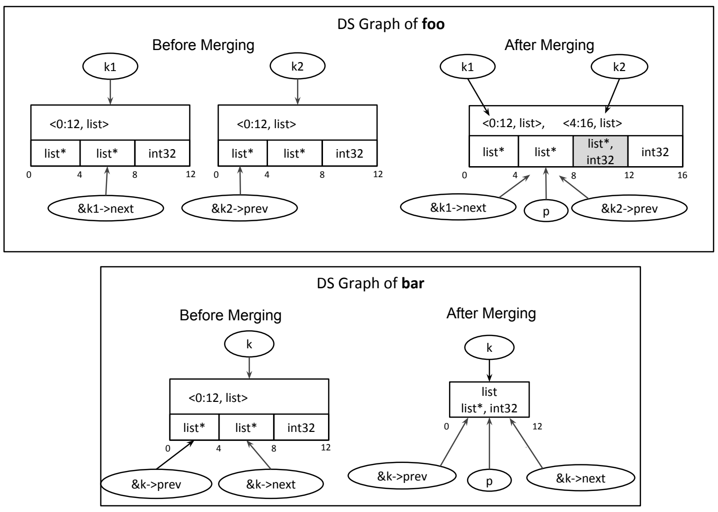

Fig. 4. DS Graph for function foo and bar (see [18]). Note that the numbers under each DS node are the offsets of its inner fields in bytes.

In function foo, an alias relationship between k1−>next and k2−>prev is introduced via a conditional assignment. The aliasing of the two fields, as mentioned above, causes the merging of their containing DS nodes. The merging process is shown in Fig. 4. Before being merged, the DS nodes pointed to by k1 and k2 are disjoint, and each holds the type information of structure list , including the field layout. During the process of merging, the DS node pointed by k2 is shifted by 4 bytes to align the aliased fields and is then merged with the node pointed to by k1. In the resulting graph, we can see that the aliased fields k1−>next and k2−>prev are placed together, but the subsequent fields k1−>data and k2−>next are also placed in the same cell (colored gray), even though they are not aliased.

In function bar, the conditional assignment introduces an alias relationship between two fields in the same DS node pointed to by k. In this case, as shown in Fig. 4, the whole DS node is collapsed and all the fields are merged into a single alias group. The reason for this is that in DSA, if a DS node is merged with itself with shifting, this is considered an *incompatible* merge. In other words, DSA does not support field sensitivity for records with inner field aliasing.

As shown above, while DSA does support field-sensitive points-to analysis on heap data structures, the computed alias information is still rather conservative. By performing the merging process at the object level (DS nodes) rather than at the field level, invalid alias relationships are introduced. Partly to address these issues, we developed a novel *cell-based* field-sensitive (CFS) points-to analysis.

4 Cell-based Points-to Analysis

A *cell*[4] is a generalization of an alias group. Initially, each program expression that corresponds to a memory location at runtime (i.e. an l-value) is associated with a unique cell whose *size* is a positive integer denoting the size (in bytes) of the values that expression can have. In addition, each cell has a type, which is *scalar* unless its associated program expression is of structure or union type (in which case the cell type is *record*). Under certain conditions, the analysis may merge cells. If two cells of two different sizes are merged, then the result is a cell whose size is $\top$. The analysis maintains an invariant that the locations associated with any two isolated scalar cells are *always disjoint*, which makes the memory partitioning using the analysis possible.

Our analysis creates a points-to graph whose vertices are cells. The graph has two kinds of edges. A *points-to* edge plays the same role here as in the points-to graphs mentioned in the previous section: $\alpha \rightharpoonup \beta$ denotes that dereferencing some expression associated with cell α yields an address that must be in one of the locations associated with cell β. A key contribution of our approach which improves precision is that unlike other field-sensitive analyses (e.g. [18, 26]), inner cells (fields) may be nested in *more than one* outer cell (record). Thus, we use additional graph edges to represent containment relations. A *contains edge* $\alpha \hookrightarrow_{i,j} \beta$ denotes that cell α is of record type and that β is associated with a field of the record whose location is at an offset of i from the record start and whose size in bytes is $j - i$. Each cell may have multiple outgoing contains edges to its inner cells and multiple incoming contains edges from its outer cells.

Fig. 5 shows a simple example. On the left is the memory layout of a singly-linked list with one element. The element is a record with two fields, a data value and a *next* pointer (which points back to the element in this case). The graph, shown on the right, contains three cells. The square cell is associated with the entire record element and the round cells with the inner fields (here and in the other points-to graphs below, we follow the convention that square cells are of

[4] We borrow this term from Miné [19], but use it in a different context. Miné aimed to build a cell-based abstract domain for value analysis, while we target a cell-based points-to analysis.

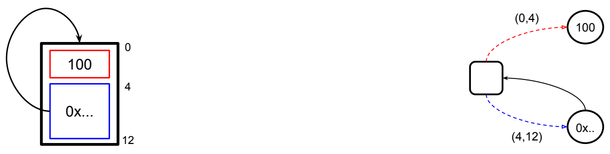

Fig. 5. Concrete memory state and its graph representation.

record type and round cells are of scalar type). The solid edge is a points-to edge from the next field to the record cell, and the dashed edges are contains edges from the record cell to the field cells. These contains edges are labeled with their corresponding starting and ending offsets within the record. The following properties hold for contains edges:

– reflexivity: $\alpha \hookrightarrow_{0,s} \alpha$, if α is a cell with a numeric size s;
– transitivity: if $\alpha_1 \hookrightarrow_{i_1,j_1} \alpha_2$ and $\alpha_2 \hookrightarrow_{i_2,j_2} \alpha_3$, then $\alpha_1 \hookrightarrow_{i_1+i_2,i_1+j_2} \alpha_3$;
– linearity: if $\alpha_1 \hookrightarrow_{i_1,j_1} \alpha_2$ and $\alpha_1 \hookrightarrow_{i_2,j_2} \alpha_3$, then $\alpha_2 \hookrightarrow_{i_2-i_1,j_2-i_1} \alpha_3$ if $i_1 \leq i_2 < j_2 \leq j_1$.

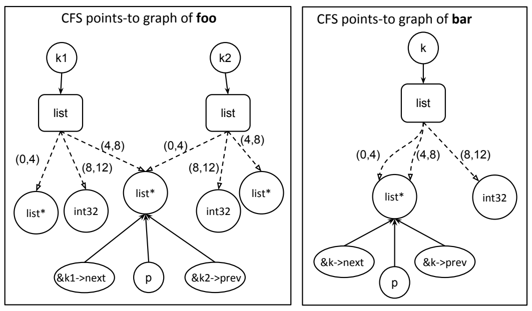

Fig. 6. CFS points-to graph for function foo and bar.

Let us revisit the running example from Fig. 3. The points-to graphs computed by our cell-based points-to analysis are shown in Fig. 6. In these graphs, record fields are separated out into individual cells. When field aliasing is detected, the individual field cells are merged (rather than their containing record cells), and any associated contains edges are kept unchanged. As shown in the graph on the left, in function foo, fields k1−>next and k2−>prev are merged into a single cell with contains edges from the record cells pointed to by k1 and k2, while other unaliased fields are kept separate. In function bar, as shown in the graph on the right, fields k−>prev and k−>next share the same cell with two incoming contains edges from the record cell pointed to by k, each labeled with different offsets. The record cell itself is not collapsed. Note that for these examples (and unlike DSA), our analysis introduces no extraneous alias relationship. Below, we explain our analysis in more detail by illustrating how it behaves in the presence of various type-unsafe operations.

Union types declare fields that share the same memory region. In a union, all fields with scalar types (e.g. float or int) are aliased. If there are nested records in a union, then two nested fields are aliased only if their offset ranges overlap. Both cases can be captured naturally in our analysis using contains edges.

Pointer arithmetic is particularly problematic for points-to analyses as it can (in principle) be used to access any location in the memory. We follow the standard approach of assuming that pointer arithmetic will not move a pointer outside of the memory object it is pointing to [30]. This assumption, coupled with the appropriate checks in the verification tool, is still sound in the sense that if it is possible to access invalid memory, the tool will detect it [10]. In our algorithm, any cell pointed to by operands of a pointer arithmetic expression is collapsed, meaning all of its outer record and inner field cells are merged into a single scalar cell. Consider function buz in Fig. 7. The call to foo(undef, p) induces the same graph for foo as in Fig. 6. However, the expression ∗(p + undef) results in this graph being collapsed into a single cell (colored in gray), as shown on the left of Fig. 9.

```
list** buz(int32 undef) {
  list **p;
  foo(undef, p);
  *(p + undef) = 0;
  return p;
}
```

Fig. 7. Code with pointer arithmetic.

```
typedef struct dlist {
  struct dlist *prev, *next;
} dlist;

dlist qux(int32 undef) {
  list **p;
  foo(undef, p);
  dist *q = (dist *) p;
  return *q;
}
```

Fig. 8. Code with pointer casting.

Pointer casting creates an alternative view of a memory region. To model this, a fresh cell is added to the points-to graph representing the new view. To illustrate, consider function qux in Fig. 8. Again, the function call foo(undef, p) induces the same graph for foo as in Fig. 6. After the call, pointer p is cast to the type dlist ∗. A new record cell (colored in gray) is added to the graph as shown in the middle of Fig. 9. The newly added cell is essentially a copy of the cell pointed to by p except that the offset intervals are enlarged in order to match the size of dlist . The casting introduces an alias between k1−>data and k2−>next, as both may alias with q−>next. By applying the properties of contains edges, the graph can be simplified into the one on the right of Fig. 9, which precisely captures the alias introduced by casting.

Another contribution of our analysis is that we track the *size* of each alias group (either a numeric value or $\top$). The size enables further improvements in the memory model: the memory array for an alias group whose size is $\top$ is modeled as an array of bytes, while the memory array for a group whose size is some numeric value n can be modeled as an array of n-byte elements. For these latter arrays, it then becomes possible to read or write n bytes with a single array operation (whereas with an array of bytes, n operations are needed). Not having to decompose array accesses into byte-level operations reduces the size and complexity of the resulting SMT formulas.

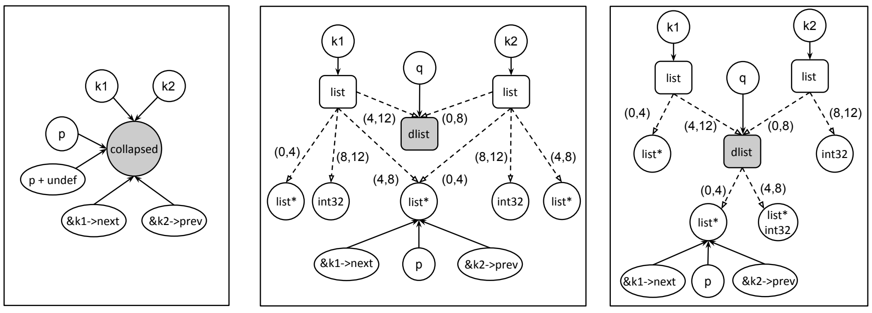

Fig. 9. CFS point-to graphs of function buz and qux.

5 Constraint-based Analysis

In this section, we formalize the cell-based field-sensitive points-to analysis described above using a constraint framework. Our constraint-based program analysis is divided into two phases: constraint generation and constraint resolution. The constraint generation phase produces constraints from the program source code in a syntax-directed manner. The constraint resolution phase then computes a solution of the constraints in the form of a cell-based field-sensitive points-to graph. The resulting graph describes a safe partitioning of the memory for all reachable states of the program.

5.1 Language and Constraints

For the formal treatment of our analysis, we consider the idealized C-like language shown in Fig. 10. To simplify the presentation, complex assignments are broken down to simpler assignments between expressions of scalar types, static arrays are represented as pointers to dynamically allocated regions, and a single type ptr is used to represent all pointer types. Function definitions, function calls, and function pointers are omitted.[5]

$t ::=$	$uint8 \mid int8 \mid \ldots \mid int64$	integer types	$e ::=$	$n \mid x$	constants, variables
$\mid$	ptr	pointer types	$\mid$	$*\, e \mid \&e$	pointer operations
$\mid$	$struct\{\, t_1 f_1; \ldots t_n f_n;\, \}\, S$	structure types	$\mid$	$(t*)\, e \mid e.f$	cast, field selection
$\mid$	$union\{\, t_1 f_1; \ldots t_n f_n;\, \}\, U$	union types	$\mid$	$(t*)\, malloc(e)$	heap allocation
			$\mid$	$e_1 \odot e_2$	binary operation
$\odot ::=$	$+ \mid - \mid * \mid /$	operators	$\mid$	$e_1 = e_2$	assignment
			$\mid$	e_1, e_2	sequencing

Fig. 10. Language syntax

Let $\mathbb{C}$ be an infinite set of *cell variables* (denoted τ or τ_i). We will use cell variables to assign program expressions to cells in the resulting points-to graph.

[5] They can be handled using a straightforward adaptation of Steensgaard's approach.

To do so, we assume that each subexpression e' of an expression e is labeled with a *unique* cell variable τ, with the exception that program variables x are always assigned the same cell variable, τ_x. Cell variables associated with program variables and heap allocations are called *source* variables. To avoid notational clutter, we do not make cell variables explicit in our grammar. Instead, we write $e : \tau$ to indicate that the expression e is labeled by τ.

Constraints. The syntax of our constraint language is defined as follows:

$$\begin{array}{lll} \eta & ::= & i \mid \top \mid \mathsf{size}(\tau) \qquad\qquad i \in \mathbb{N} \\ \phi & ::= & i < \eta \mid \eta_1 = \eta_2 \mid \tau_1 = \tau_2 \mid \tau_1 \rightharpoonup \tau_2 \mid \tau_1 \hookrightarrow_{i,j} \tau_2 \mid \tau_1 \trianglelefteq \tau_2 \\ & \mid & \mathsf{source}(\tau) \mid \mathsf{scalar}(\tau) \mid \mathsf{cast}(i, \tau_1, \tau_2) \mid \mathsf{collapsed}(\tau) \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \end{array}$$

Here, a term η denotes a *cell size*. The constant $\top$ indicates an unknown cell size. A constraint ϕ is a positive Boolean combination of cell size constraints, equalities on cell variables, points-to edges $\tau_1 \rightharpoonup \tau_2$, contains edges $\tau_1 \hookrightarrow_{i,j} \tau_2$ and special predicates whose semantics we describe in detail below. We additionally introduce syntactic shorthands for certain constraints. Namely, we write $i \sqsubseteq \eta$ to stand for the constraint $i = \eta \vee \eta = \top$, $i \leq \eta$ to stand for $i < \eta \vee i = \eta$, and $i \preceq \eta$ to stand for $i \leq \eta \vee \eta = \top$.

Constraints are interpreted in cell-based field-sensitive points-to graphs (CFPGs). A CFPG is a tuple $G = (C, \mathit{cell}, \mathit{size}, \mathit{source}, \mathit{scalar}, \mathit{contains}, \mathit{ptsto})$ where

- C is a finite set of cells,
- $\mathit{cell} : \mathbb{C} \to C$ is an assignment from cell variables to cells,
- $\mathit{size} : C \to \mathbb{N} \cup \{\top\}$ is an assignment from cells to cell sizes,
- $\mathit{source} \subseteq C$ is a set of *source cells*,
- $\mathit{scalar} \subseteq C$ is a set of *scalar cells*,
- $\mathit{contains} \subseteq C \times \mathbb{N} \times \mathbb{N} \times C$ is a *containment relation* on cells, and
- $\mathit{ptsto} : C \to C$ is a *points-to map* on cells.

For $c_1, c_2 \in C$, and $i, j \in \mathbb{N}$, we write $c_1 \overset{G}{\hookrightarrow}_{i,j} c_2$ as notational sugar for $(c_1, i, j, c_2) \in \mathit{contains}$, and similarly $c_1 \overset{G}{\rightharpoonup} c_2$ for $\mathit{ptsto}(c_1) = c_2$. Let $\mathit{contains}'$ be the projection of $\mathit{contains}$ onto $C \times C$: $\mathit{contains}'(c_1, c_2) \equiv \exists\, i, j. \mathit{contains}(c_1, i, j, c_2)$.

The functions and relations of G must satisfy the following consistency properties. These properties formalize the intuition of the containment relation and the roles played by source and scalar cells:

- the $\mathit{contains}'$ relation is reflexive (for cells of known size), transitive, and anti-symmetric. More specifically,

$$\forall\, c \in C.\ \mathit{size}(c) \neq \top \implies c \overset{G}{\hookrightarrow}_{0,\mathit{size}(c)} c \tag{1}$$

$$\forall \begin{pmatrix} c_1, c_2, c_3 \in C, \\ i_1, i_2, j_1, j_2 \in \mathbb{N} \end{pmatrix}.\ c_1 \overset{G}{\hookrightarrow}_{i_1,j_1} c_2 \wedge c_2 \overset{G}{\hookrightarrow}_{i_2,j_2} c_3 \implies c_1 \overset{G}{\hookrightarrow}_{i_1+i_2,i_1+j_2} c_3 \tag{2}$$

$$\forall \begin{pmatrix} c_1, c_2 \in C, \\ i_1, i_2, j_1, j_2 \in \mathbb{N} \end{pmatrix}.\ c_1 \overset{G}{\hookrightarrow}_{i_1,j_1} c_2 \wedge c_2 \overset{G}{\hookrightarrow}_{i_2,j_2} c_1 \implies c_1 = c_2 \tag{3}$$

– cells that are of unknown size or that point to other cells must be scalar:

$$\forall\, c \in C.\, \mathit{size}(c) = \top \implies c \in \mathit{scalar} \tag{4}$$

$$\forall\, c, c' \in C.\, c \overset{G}{\rightharpoonup} c' \implies c \in \mathit{scalar} \tag{5}$$

– the *contains* relation must satisfy the following linearity property:

$$\forall \begin{pmatrix} c_1, c_2, c_3 \in C, \\ i_1, i_2, j_1, j_2 \in \mathbb{N} \end{pmatrix} . \begin{pmatrix} i_1 \le i_2 < j_2 \le j_1 \,\land \\ c_1 \overset{G}{\hookrightarrow}_{i_1,j_1} c_2 \,\land \\ c_1 \overset{G}{\hookrightarrow}_{i_2,j_2} c_3 \end{pmatrix} \implies c_2 \overset{G}{\hookrightarrow}_{i_2-i_1, j_2-i_1} c_3 \tag{6}$$

– scalar cells do not contain other cells:

$$\forall\, c, c' \in C, i, j \in \mathbb{N}.\, c \in \mathit{scalar} \land c \overset{G}{\hookrightarrow}_{i,j} c' \implies c = c' \tag{7}$$

– overlapping scalar cells are equivalent. The notion of overlap is formally expressed as

$$\mathit{overlap}^G(c_1, c_2) \equiv \exists \begin{pmatrix} c \in C, \\ i_1, i_2, j_1, j_2 \in \mathbb{N} \end{pmatrix} . \begin{pmatrix} c \overset{G}{\hookrightarrow}_{i_1,j_1} c_1 \,\land \\ c \overset{G}{\hookrightarrow}_{i_2,j_2} c_2 \end{pmatrix} \land \begin{pmatrix} i_1 \le i_2 < j_1 \,\lor \\ i_2 \le i_1 < j_2 \end{pmatrix}$$

$$\forall\, c, c' \in C.\, c \in \mathit{scalar} \land c' \in \mathit{scalar} \land \mathit{overlap}^G(c, c') \implies c = c' \tag{8}$$

– cell sizes must be consistent with the *contains* relation:

$$\forall \begin{pmatrix} c_1, c_2 \in C, \\ i, j \in \mathbb{N} \end{pmatrix} . \, c_1 \overset{G}{\hookrightarrow}_{i,j} c_2 \implies \begin{pmatrix} 0 \le i < j \,\land \\ j \preceq \mathit{size}(c_1) \land j - i \sqsubseteq \mathit{size}(c_2) \end{pmatrix} \tag{9}$$

Semantics of Constraints. Let G be a CFPG with components as above. For a cell variable $\tau \in \mathbb{C}$, we define $\tau^G = \mathit{cell}(\tau)$ and for a size term η we define $\eta^G = \mathit{size}(\tau^G)$ if $\eta = \mathsf{size}(\tau)$ and $\eta^G = \eta$ otherwise. The semantics of a constraint ϕ is given by a satisfaction relation $G \models \phi$, which is defined recursively on the structure of ϕ in the expected way. Size and equality constraints are interpreted in the obvious way using the term interpretation defined above. Though, note that we define $G \not\models i < \top$ and $G \not\models i = \top$. Points-to constraints $\tau_1 \rightharpoonup \tau_2$ are interpreted by the points-to map $\tau_1^G \overset{G}{\rightharpoonup} \tau_2^G$; contains constraints $\tau_1 \hookrightarrow_{i,j} \tau_2$ are interpreted by the containment relation $\tau_1^G \overset{G}{\hookrightarrow}_{i,j} \tau_2^G$; and source and scalar are similarly interpreted by *source* and *scalar*.

Intuitively, a cast predicate $\mathsf{cast}(k, \tau_1, \tau_2)$ states that cell τ_2 is of size k and is obtained by a pointer cast from cell τ_1. Thus, any source cell that contains τ_1 at offset i must also contain τ_2 at that offset. That is, $G \models \mathsf{cast}(k, \tau_1, \tau_2)$ iff:

$$\forall\, c \in C, i, j \in \mathbb{N}.\, c \in \mathit{source} \land c \overset{G}{\hookrightarrow}_{i,j} \tau_1^G \implies c \overset{G}{\hookrightarrow}_{i,i+k} \tau_2^G \ .$$

The predicate $\mathsf{collapsed}(\tau)$ indicates that τ points to a cell c that may be accessed in a type-unsafe manner, e.g., due to pointer arithmetic. Then, $G \models \mathsf{collapsed}(\tau)$ iff

$$\forall\, c, c' \in C, i, j \in \mathbb{N}.\, \tau^G \overset{G}{\rightharpoonup} c \land c' \overset{G}{\hookrightarrow}_{i,j} c \implies c' = c \ .$$

The predicate $\tau_1 \trianglelefteq \tau_2$ (taken from [26]) is used to state the equivalence of the points-to content of τ_1 and τ_2. Formally, $G \models \tau_1 \trianglelefteq \tau_2$ iff

$$\forall c \in C.\, \tau_1^G \xrightarrow{G} c \implies \tau_2^G \xrightarrow{G} c \enspace .$$

5.2 Constraint Generation

The first phase of our analysis generates constraints from the target program in a syntax-directed bottom-up fashion. The constraint generation is described by the inference rules in Fig. 11. Recall that each program expression e is labeled with a cell variable τ. The judgment form $e : \tau|\phi$ means that for the expression e labeled by the cell variable τ, we infer the constraint ϕ over the cell variables of e (including τ). A box with annotated ϕ in the premise of an inference rule is used to indicate that ϕ is the conjunction of the formulas contained in the box. Thus, the formula in the box is the constraint inferred in the conclusion of that rule.

For simplicity, we assume the target program is well-typed. Our analysis relies on the type system to infer the byte sizes of expressions and the field layout within records (e.g. structures or unions). To this end, we assume a type environment $\mathcal{T}$ that assigns C types to program variables. Moreover, we assume the following functions: $\mathit{typeof}(\mathcal{T}, e)$ infers the type of an expression following the standard type inference rules in the C language; $|t|$ returns the byte size of the type t; and $\mathit{offset}(t, \mathsf{f})$ returns the offset of a field f from the beginning of its enclosing record type t. Finally, $\mathit{isScalar}(t)$ returns true iff the type t is an integer or pointer type.

The inference rules are inspired by the formulation of Steensgaard's field-insensitive analysis due to Forster and Aiken [12]. We adapt them to our cell-based field-sensitive analysis. Note that implications of the form $\mathit{isScalar}(t) \implies \mathsf{scalar}(\tau)$, which we use in some of the rules, are directly resolved during the rule application and do not yield disjunctions in the generated constraints.

We only discuss some of the rules in detail. The rule Malloc generates the constraints for a malloc operation. We assume that each occurrence of malloc in the program is tagged with a unique identifier l and labeled with a unique cell variable τ representing the memory allocated by that malloc. The return value of malloc is a pointer with associated cell variable τ'. Thus, τ' points to τ.

The rules Dir-Sel, Arith-Op, and Cast are critical for the field-sensitive analysis. In particular, Dir-Sel generates constraints for field selections. A field f within a record expression e is associated with a cell variable τ_f. The rule states there must be a contains-edge from the cell variable τ associated with e to τ_f with appropriate offsets. Rule Arith-Op is for operations that may involve pointer arithmetic. The cell variables τ_1, τ_2 and τ are associated with e_1, e_2, and $e_1 \odot e_2$, respectively. Any cells pointed to by τ_1 and τ_2 must be equal, which is expressed by the constraints $\tau_1 \trianglelefteq \tau$ and $\tau_2 \trianglelefteq \tau$. Moreover, if τ points to another cell τ', then pointer arithmetic collapses all relevant cells containing τ', since we can no longer guarantee structured access to the memory represented by τ'. Rule Cast handles pointer cast operations. A cast can change

$$\textsc{Const}\ \frac{\boxed{\tau = \tau}^{\phi}}{n : \tau \mid \phi} \qquad \textsc{Seq}\ \frac{e_1 : \tau_1 \mid \phi_1 \quad e_2 : \tau_2 \mid \phi_2 \quad \boxed{\tau = \tau_2}^{\phi}}{e_1, e_2 : \tau \mid \phi_1 \wedge \phi_2 \wedge \phi}$$

$$\textsc{Assg}\ \frac{e_1 : \tau_1 \mid \phi_1 \quad e_2 : \tau_2 \mid \phi_2 \quad \boxed{\tau_2 \tau_1 \quad \tau = \tau_2}^{\phi}}{e_1 = e_2 : \tau \mid \phi_1 \wedge \phi_2 \wedge \phi} \qquad \textsc{Var}\ \frac{\boxed{\begin{array}{c} t = \mathsf{x} \quad (\tau) \\ t \implies (\tau) \\ 0|t|\tau\tau \end{array}}^{\phi}}{\mathsf{x} : \tau \mid \phi}$$

$$\textsc{Addr}\ \frac{e : \tau \mid \phi_1 \quad \boxed{\begin{array}{c} (\tau') \\ \tau'\tau \quad 0|\mathsf{ptr}|\tau'\tau' \end{array}}^{\phi}}{\&e : \tau' \mid \phi_1 \wedge \phi} \qquad \textsc{Deref}\ \frac{e : \tau \mid \phi_1 \quad \boxed{\begin{array}{c} t = *e \\ t \implies (\tau') \\ \tau\tau' \quad 0|t|\tau'\tau' \end{array}}^{\phi}}{*e : \tau' \mid \phi_1 \wedge \phi}$$

$$\textsc{Malloc}\ \frac{\mathsf{malloc}_l : \tau \quad \boxed{\begin{array}{c} t \implies (\tau) \\ (\tau) \quad (\tau') \\ \tau'\tau \quad 0|\mathsf{ptr}|\tau'\tau' \quad 0|t|\tau\tau \end{array}}^{\phi}}{(t*)\mathsf{malloc}_l(e) : \tau' \mid \phi} \qquad \textsc{Dir-Sel}\ \frac{e : \tau \mid \phi_1 \quad \boxed{\begin{array}{c} t = e \\ o = tf \\ t.\mathsf{f} \implies (\tau_f) \\ oo + |t.\mathsf{f}|\tau\tau_f \quad 0|t.\mathsf{f}|\tau_f\tau_f \end{array}}^{\phi}}{e.\mathsf{f} : \tau_f \mid \phi_1 \wedge \phi}$$

$$\textsc{Cast}\ \frac{e : \tau_1 \mid \phi_1 \quad (t*)_l : \tau_2' \quad \boxed{\begin{array}{c} t \implies (\tau_2') \\ \tau_1\tau_1' \quad \tau_2\tau_2' \\ 0|\mathsf{ptr}|\tau_2\tau_2 \quad 0|t|\tau_2'\tau_2' \\ |t|\tau_1'\tau_2' \end{array}}^{\phi}}{(t*)_l e : \tau_2 \mid \phi_1 \wedge \phi} \qquad \textsc{Arith-Op}\ \frac{e_1 : \tau_1 \mid \phi_1 \quad e_2 : \tau_2 \mid \phi_2 \quad \boxed{\begin{array}{c} t = e_1\ op_b\ e_2 \quad t \implies (\tau) \\ \tau_1\tau \quad \tau_2\tau \quad 0|t|\tau\tau \quad (\tau) \end{array}}^{\phi}}{e_1\ op_b\ e_2 : \tau \mid \phi_1 \wedge \phi_2 \wedge \phi}$$

Fig. 11. Constraint generation rules.

the points-to range of a pointer. In the rule, τ_1 and τ_2 represent the operand and result pointer, respectively. τ_1' and τ_2' represent their points-to contents. Similar to malloc, each cast $t*$ has a unique identifier l and is labeled with a unique cell variable τ_2' that represents the points-to content of the result. The constraint $\mathsf{cast}(s, \tau_1', \tau_2')$ specifies that both τ_1' and τ_2' are within the same source containers with the same offsets. In particular, the size of τ_2' must be consistent with $|t|$ (the size of the type t).

5.3 Constraint Resolution

We next explain the constraint resolution step that computes a CFPG G from the generated constraint ϕ such that $G \models \phi$. The procedure must be able to reason about containment between cells, a transitive relation. Inspired by a procedure for the reachability in function graphs [15], we propose a rule-based procedure for this purpose.

The procedure is defined by a set of inference rules that infer new constraints from given constraints. The rules are shown in Figure 12. They are derived directly from the semantics of the constraints and the consistency properties of

CFPGs. Some of the rules make use of the following syntactic shorthand:

$$\mathsf{overlap}(\tau, \tau_1, i_1, j_1, \tau_2, i_2, j_2) \equiv \tau \hookrightarrow_{i_1,j_1} \tau_1 \wedge \tau \hookrightarrow_{i_2,j_2} \tau_2 \wedge i_1 \leq i_2 \wedge i_2 < j_1$$

We omit the rules for reasoning about equality and inequality constraints, as they are straightforward. We also omit the rules for detecting conflicts. The only possible conflicts are inconsistent equality constraints such as $i = \top$ and inconsistent inequality constraints such as $i < \top$.

Our procedure maintains a *context* of constraints currently asserted to be true. The initial context is the set of constraints collected in the first phase. At each step, the rewrite rules are applied on the current context. For each rule, if the antecedent formulas are matched with formulas in the context, the consequent formula is added back to the context. The rules are applied until a conflict-free saturated context is obtained. The rule Split branches on disjunctions. Note that the rules do not generate new disjunctions. All disjunctions come from the constraints of the form $i \preceq \eta$ and $i \sqsubseteq \eta$ in the initial context. Each disjunction in the initial context has at least one satisfiable branch. Our procedure uses a greedy heuristic that first chooses for each disjunction the branch that preserves more information and then backtracks on a conflict to choose the other branch. For example, for a disjunct $i \sqsubseteq \eta$, we first try $i = \eta$ before we choose $\eta = \top$. Once a conflict-free saturated context has been derived, we construct the CFPG using the equivalence classes of cell variables induced by the derived equality constraints as cells of the graph. The other components can be constructed directly from the constraints.

Termination. To see that the procedure terminates, note that none of the rules introduce new cell variables τ. Moreover, the only rules that can increase the offsets i, j in containment constraints $\tau_1 \hookrightarrow_{i,j} \tau_2$ are Cast and Trans. The application of these rules can be restricted in such a way that the offsets in the generated constraints do not exceed the maximal byte size of any of the types in the input program. With this restriction, the rules will only generated a bounded number of containment constraints.

Soundness. The soundness proof of the analysis is split into three steps. First, we prove that the CFPG resulting from the constraint resolution indeed satisfies the original constraints that are generated from the program. The proof shows that the inference rules are all consequences of the semantics of the constraints and the consistency properties of CFPGs. The second step defines an abstract semantics of programs in terms of abstract stores. These abstract stores only keep track of the partition of the byte-level memory into alias groups according to the computed CFPG. We then prove that the computed CFPG is a safe inductive invariant of the abstract semantics. The safety of the abstract semantics is defined in such a way that it guarantees that the computed CFPG describes a valid partition of the reachable program states into alias groups. Finally, we prove that the abstract semantics simulates the concrete byte-level semantics of programs. The details of the soundness proof are omitted due to space restrictions.

$$\textsc{Size1}\ \frac{\tau_1 \hookrightarrow_{i,j} \tau_2 \quad size\ (\tau_1) = k}{j \le k} \qquad \textsc{Size2}\ \frac{\tau_1 \hookrightarrow_{i,j} \tau_2 \quad size\ (\tau_2) = k}{j - i = k}$$

$$\textsc{Refl}\ \frac{size\ (\tau) = i}{\tau \hookrightarrow_{0,i} \tau} \qquad \textsc{Trans}\ \frac{\tau_1 \hookrightarrow_{i_1,j_1} \tau_2 \qquad \tau_2 \hookrightarrow_{i_2,j_2} \tau_3}{\tau_1 \hookrightarrow_{i_1+i_2, i_1+j_2} \tau_3}$$

$$\textsc{AntiSym}\ \frac{\tau_1 \hookrightarrow_{i_1,j_1} \tau_2 \qquad \tau_2 \hookrightarrow_{i_2,j_2} \tau_1}{\tau_1 = \tau_2}$$

$$\textsc{Collapse1}\ \frac{\tau_1 \hookrightarrow_{i,j} \tau_2 \qquad scalar\ (\tau_1)}{\tau_1 = \tau_2} \qquad \textsc{Linear}\ \frac{\begin{array}{c}\tau \hookrightarrow_{i_1,j_1} \tau_1 \qquad \tau \hookrightarrow_{i_2,j_2} \tau_2 \\ i_1 \le i_2 < j_2 \le j_1\end{array}}{\tau_1 \hookrightarrow_{i_2-i_1, j_2-i_1} \tau_2}$$

$$\textsc{Collapse2}\ \frac{(\tau) \quad \tau\tau_1 \quad \tau' \hookrightarrow_{i,j} \tau_1}{\tau' = \tau_1} \qquad \textsc{Cast}\ \frac{\begin{array}{c}\mathsf{cast}(k, \tau_1, \tau_2) \qquad source\ (\tau) \\ \tau \hookrightarrow_{i,j} \tau_1\end{array}}{\tau \hookrightarrow_{i,i+k} \tau_2}$$

$$\textsc{Scalar}\ \frac{size\ (\tau) = \top}{scalar\ (\tau)}$$

$$\textsc{Overlap}\ \frac{scalar\ (\tau_1) \quad scalar\ (\tau_2) \quad overlap(\tau, \tau_1, i_1, j_1, \tau_2, i_2, j_2)}{\tau_1 = \tau_2}$$

$$\textsc{Points}\ \frac{\tau\tau_1 \qquad \tau\tau_2}{\tau_1 = \tau_2} \qquad \textsc{PtrEq}\ \frac{\tau_1\tau_2 \qquad \tau_1\tau}{\tau_2\tau} \qquad \textsc{Split}\ \frac{\phi_1 \vee \phi_2}{\phi_1 \qquad \phi_2}$$

Fig. 12. Constraint resolution rules

6 Experiments

To assess the impact of different memory models in a verification setting, we implemented both the flat memory model and the partitioned memory model in the Cascade verification framework [29], a competitive[6] C verification tool that uses bounded model checking and SMT solving. For the partitioned memory model, a points-to analysis is run as a preprocessing step, and the resulting points-to graph is used to: (i) determine the element size of the memory arrays; (ii) select which memory array to use for each read or write (as well as for each memory safety check); and (iii) add disjointness assumptions where needed (for distinct locations assigned to the same memory array). We implemented several points-to analyses, including Steensgaard's original and field-sensitive analyses, the data structure analysis (DSA), and our cell-based points-to analysis.

Benchmarks. For our experiments, we used a subset of the SVCOMP'16 benchmarks [4], specifically the Heap Data Structures category (consisting of two sub-categories HeapReach and HeapMemSafety) as these contained many programs with heap-allocated data structures. For HeapReach, we checked for reachability of the ERROR label in the code. For HeapMemSafety, we checked for invalid memory dereferences, invalid memory frees, and memory leaks.

[6] Cascade placed 3rd in the Heap Data Structures category of SV-COMP 2016 [4].

Configuration. Like other bounded model checkers, Cascade relies on function inlining and loop unrolling. It takes as parameters a function-inline depth d and a loop-unroll bound b. It then repeatedly runs its analysis, inlining all functions up to depth d, and using a set of successively larger unrolls until the bound b is reached. There are four possible results: *unknown* indicates that no result could be determined (for our experiments this happens only when the depth of function calls exceeds d); *unsafe* indicates that a violation was discovered; *safe* indicates that no violations exist *within the given loop unroll bound*; and *timeout* indicates the analysis could not be completed within the time limit provided. For the reachability benchmarks, we set $d = 6$ and $b = 1024$; for the memory safety benchmarks, we set $d = 8$ and $b = 200$ (these values were empirically determined to work well for SV-COMP). Note that Cascade may report *safe* even if a bug exists, if this bug requires executing a loop more times than is permitted by the unroll limit (the same is true of other tools relying on bounded model checking, like CBMC and LLBMC). For other undiscovered bugs, it reports *unknown* or *timeout*.

	HeapReach(81)						HeapMemSafety(190)					
	False(25)			True(56)			False(83)			True(107)		
	#solved	time(s)	ptsTo(s)	#solved	time(s)	ptsTo(s)	#solved	time(s)	ptsTo(s)	#solved	time(s)	ptsTo(s)
Flat	19	112.8	-	29	228.7	-	50	357.3	-	38	567.5	-
St-fi	19	96.4	0.06	33	168.3	0.08	54	306.2	0.04	39	548.0	0.03
St-fs	19	92.8	0.14	33	168.5	0.17	58	274.8	0.05	42	527.4	0.05
DSA-local	19	94.6	0.09	33	168.1	0.11	54	305.2	0.07	39	550.4	0.07
CFS	19	**69.4**	0.11	33	168.3	0.14	**66**	**182**	0.05	**50**	**461.8**	0.05

Table 1. Comparison of various memory models in Cascade. Experiments were performed on an Intel Core i7 (3.7GHz) with 8GB RAM. The timeout was 850 seconds. "#solved" is the number of benchmarks correctly solved within the given limits. In the columns labeled "False", a benchmark is solved if Cascade found a bug that violates the specified property. In the columns labeled "True", a benchmark is solved if Cascade completed its analysis up to the maximum unroll and inlining limits without finding a bug. "time" is the average time spent on the benchmarks (solved and unsolved) in each category. "ptsTo" is the average time spent on the points-to analysis in each category.

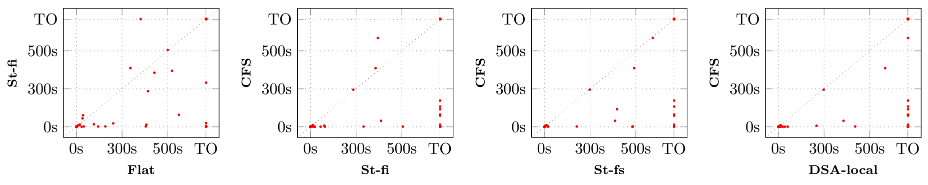

Fig. 13. Scatter plots showing a benchmark-by-benchmark comparison of various memory models over all the 271 benchmarks. The timeout (TO) was set to 850 seconds.

Table 1 reports results for the flat model (Flat) and partitioned models using the original (field-insensitive) Steensgaard analysis (St-fi), the field-sensitive

Steensgaard analysis (St-fs), the context-insensitive DSA (DSA-local),[7] and our cell-based field-sensitive points-to analysis (CFS). More detail is shown in the scatter plots in Fig. 13. These results show that the baseline partitioned model (St-fi) already improves significantly over the flat model. Of the partitioned models, the cell-based model (CFS) is nearly uniformly superior to the others.

To compare the precision of the different points-to analyses, we also computed the number of alias groups computed by each algorithm. Over the 190 benchmarks in HeapMemSafety, CFS always computes more or the same number of alias groups than the other analyses (it is better than DSA-local on 10 benchmarks, St-fs on 67 benchmarks, and St-fi on 165 benchmarks). The same is true of the 81 benchmarks in HeapReach, (better than DSA-local on 2 benchmarks, St-fs on 28 benchmarks and St-fi on 52 benchmarks). The precision advantage of CFS over DSA-local is somewhat limited because field aliasing does not occur too frequently in these programs.

The other advantage of CFS is that it tracks the access size of objects in each alias group. The only other analysis that does this is St-fs, and as a result, St-fs solves more benchmarks than DSA-local, even though the alias information computed by DSA-local is much more precise. This shows the advantage of tracking size information for memory modeling applications.

7 Related Work

Several memory models have been proposed as alternatives to the flat model. Cohen *et al.* [8] simulate a typed memory model over untyped C memory by adding additional disjointness axioms. Their approach introduces quantified axioms which can sometimes be challenging for SMT solvers. Böhme *et al.* [5] propose a variant of Cohen's memory model. CCured [21] separates pointers by their usage (not alias information) and skips bounds checking for "safe" pointers. Quantified disjointness axioms are also introduced. Rakamarić *et al.* [25] propose a variant of the Burstall model that employs type unification to cope with type-unsafe behaviors. However, in the presence of type casting, the model can easily degrade into the flat model. Havoc [9, 16] refines the Burstall model with field-sensitivity that is only applicable to field-safe code fragments. Lal *et al.* [17] split memory in order to reason about the bitvector operations and integer operations separately. Frama-C [11] develops various models at different abstraction levels. However, no attempt is made to further partition the memory.

For points-to analyses in general, we refer the reader to the survey by Hind [14]. Yong *et al.* [30] propose a framework for a spectrum of analyses from complete field-insensitivity through various levels of field-sensitivity, but none of these analyses support field-sensitive analysis of heap data structures. Pearce *et al.* [22, 23] extends field-sensivity on heap data structures with an inclusion-

[7] We used a context-insensitive version of DSA to make a fair comparison because the other analyses are also context-insensitive. Context sensitivity could be added to any of them, improving the results.

based approach, and Balatsouras *et al.* [2] further improve the precision. However, none of them are precise enough to analyze field-aliasing.

8 Conclusion

In this paper, we introduced partitioned memory models and showed how to use various points-to analyses to generate coarser or finer partitions. A key contribution was a new cell-based points-to analysis algorithm, which improves on earlier field-sensitive points-to analyses, more precisely modeling heap data structures and type-unsafe operations in the C language.

In SV-COMP 2016, Cascade won the bronze medal in the Heap Data Structures category using the cell-based partitioned memory model We conclude that the cell-based partitioned memory model is a promising approach for obtaining both scalability and precision when modeling memory.

References

1. Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, University of Copenhagen, May 1994.
2. George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for C and C++. In *Static Analysis Symposium (SAS)*, 2016.
3. Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard – version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (SMT)*, 2010.
4. Dirk Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2016.
5. Sascha Böhme and MichałMoskal. Heaps and data structures: A challenge for automated provers. In *Conference on Automated Deduction (CADE)*, 2011.
6. Rodney M. Burstall. Some techniques for proving correctness of programs which alter data structures. In *Machine Intelligence*, 1972.
7. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.
8. Ernie Cohen, Michal Moskal, Stephan Tobies, and Wolfram Schulte. A precise yet efficient memory model for C. In *Electronic Notes in Theoretical Computer Science (ENTCS)*, 2009.
9. Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In *Principles of Programming Languages (POPL)*, 2009.
10. Christopher L. Conway, Dennis Dams, Kedar S. Namjoshi, and Clark Barrett. Pointer analysis, conditional soundness, and proving the absence of errors. In *Static Analysis Symposium (SAS)*, 2008.
11. Pascal Cuoq, Florent Kirchner, Nikolaï Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C a software analysis perspective. In *Software Engineering and Formal Methods (SEFM)*, 2012.

12. Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Flow-insensitive points-to analysis with term and set constraints. Technical Report CSD-97-964, University of California, Berkeley, 1997.
13. Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In *Computer Aided Verification (CAV)*, 2015.
14. Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Program Analysis for Software Tools and Engineering (PASTE)*, 2001.
15. Shuvendu K. Lahiri and Shaz Qadeer. Back to the future: Revisiting precise program verification using SMT solvers. In *Principles of Programming Languages (POPL)*, 2008.
16. Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. Static and precise detection of concurrency errors in systems code using SMT solvers. In *Computer Aided Verification (CAV)*, 2009.
17. Akash Lal and Shaz Qadeer. Powering the static driver verifier using Corral. In *Foundations of Software Engineering (FSE)*, 2014.
18. Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Programming Language Design and Implementation (PLDI)*, 2007.
19. Antoine Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Language, Compilers, and Tool Support for Embedded Systems (LCTES)*, 2006.
20. Jeremy Morse, Mikhail Ramalho, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. ESBMC 1.22 (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2014.
21. George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Principles of Programming Languages (POPL)*, 2002.
22. David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for C. In *Program Analysis for Software Tools and Engineering (PASTE)*, 2004.
23. David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of C. *ACM Trans. Program. Lang. Syst.*, 2007.
24. Zvonimir Rakamarić and Michael Emmi. SMACK: Decoupling source language details from verifier implementations. In *Computer Aided Verification (CAV)*, 2015.
25. Zvonimir Rakamarić and Alan J. Hu. A scalable memory model for low-level code. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2009.
26. Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Compiler Construction (CC)*, 1996.
27. Bjarne Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages (POPL)*, 1996.
28. Wei Wang. *Partition Memory Models for Program Analysis.* PhD thesis, New York University, May 2016.
29. Wei Wang, Clark Barrett, and Thomas Wies. Cascade 2.0. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2014.
30. Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. In *Programming Language Design and Implementation (PLDI)*, 1999.