A Context-Sensitive Memory Model for Verification of C/C++ Programs[⋆]

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. Verification of low-level C/C++ requires a precise memory model that supports type unions, pointer arithmetic, and casts. We present a new memory model that splits memory into a finite set of disjoint regions based on a pointer analysis. The main contribution is a field-, array- and context-sensitive pointer analysis tailored to verification. We have implemented our memory model for the LLVM bitcode and used it on a C++ case study and on SV-COMP benchmarks. Our results suggests that our model can reduce verification time by producing a finer-grained partitioning in presence of function calls.

1 Introduction

Verification of low-level C and C++ programs (e.g., OS drivers, flight control systems) often requires a low-level modeling of the heap that supports *type-unsafe* features such as type unions, pointer arithmetic, and pointer casts. The more detailed the memory model the more precise the analysis. However, extra details often increase the computational cost of the analysis.

A memory model defines the semantics of pointers. The standard C/C++ memory model (also called *block-level*) interprets a pointer as a pair (id, o) where id is an identifier that uniquely defines a memory region and o defines the byte in the region being point to. The number of regions is unbounded. On the other hand, in the *flat* or *byte-level* model used by most execution platforms there is a single memory region (i.e., the memory) and every allocation returns a new offset in that region. For verification purposes, we adopt a memory model similar to that of C/C++ but ensuring a finite number of memory regions.

We say that a memory model is *context-sensitive* (*context-insensitive*) if memory is divided into finitely many global regions and every pointer points

[⋆] This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center Pacific (SSC Pacific) under Contract No. N66001-15-C-4061. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DARPA or SSC Pacific. This work has been also supported in part by NSF grant 1528153.

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}
void g(int* p,int* q,int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

(a) C-like fragment

$f(x, y, A_x, A_y, A'_x, A'_y)$ $A'_x = A_x[x \leftarrow 1]$ $A'_y = A_y[y \leftarrow 2]$ $g(\ldots)$ $f(p, q, A_{pq}, A_{pq}, A'_{pq}, A'_{pq})$ $f(r, s, A_r, A_s, A'_r, A'_s)$	$f(x, y, A_{xy}, A''_{xy})$ $A'_{xy} = A_{xy}[x \leftarrow 1]$ $A''_{xy} = A'_{xy}[y \leftarrow 2]$ $g(\ldots)$ $f(p, q, A_{pqrs}, A'_{pqrs})$ $f(r, s, A'_{pqrs}, A''_{pqrs})$	$f(x, y, A_{xy}, A''_{xy})$ $A'_{xy} = A_{xy}[x \leftarrow 1]$ $A''_{xy} = A'_{xy}[y \leftarrow 2]$ $g(\ldots)$ $f(p, q, A_{pq}, A'_{pq})$ $f(r, s, A_{rs}, A'_{rs})$
(b) Context-sensitive	(c) Context-insensitive	(d) Our approach

Fig. 1: Several translations to a side-effect free language if `p` and `q` may alias.

to a region that does (not) depend on the call paths leading to the allocation of the region. A context-sensitive memory model might scale more than its context-insensitive counterpart because it induces a finer-grained memory partitioning so that verification conditions can be solved more efficiently.

This paper presents a new context-sensitive memory model for verification of C/C++ programs. Like other memory models (e.g., [19, 21]) our model divides memory into disjoint regions based on the information computed by a pointer analysis. Unlike these works, our memory model produces a finer-grained partitioning in the presence of function calls.

Our main technical contribution is a context, field, and array-sensitive pointer analysis for C/C++ which disambiguates the heap in a way that a side-effect free version of the program can be obtained from which modular verification conditions can be generated. A standard approach (e.g., SMACK [18], SEAHORN [11], CBMC [7], ESBMC [10], and CASCADE [22]) to transforming an imperative program to a side-effect-free form is to map each pointer p to a symbolic memory region M_p using a pointer analysis. Then, each M_p is replaced by a logical array A_p and memory accesses are replaced by array stores and selects to A_p. Each array store on A_p produces a new version of A'_p representing the array after the execution of the memory write. This encoding is straightforward for intra-procedural code. However, a precise modular encoding is more challenging.

Consider the snippet of C on Fig. 1(a) and three translations to a side-effect free program[3] on Fig. 1(b-d) using three flow-insensitive pointer analyses: (b) a

[3] Logic-based verifiers require to generate verification conditions in a side-effect free form so that they can be solved by SMT solvers. In this paper, we focus on how to provide precise points-to information to produce a sound translation to such a form. The syntax and semantics of the language and construction of VCs are beyond the scope of this paper. We refer readers to e.g., [11, 18] and their references for details.

context-sensitive analysis, (c) context-insensitive, and (d) our technique. Assume that at the entry of the procedure `g` variables `p` and `q` may alias.

Memory has been disambiguated such that each accessed memory region is passed explicitly to each call. Each logical array A_S denotes a memory region to which all pointers in S point to. Non-primed and primed names denote input and output versions, respectively. Intuitively, the larger the number of non-primed array variables the more efficient the process of solving verification conditions may be because more disjointness information between regions is available to the solver. Let us focus on the procedure g. The first translation produces three non-primed array variables: A_{pq}, A_r, and A_s, the second translation one array: A_{pqrs}, and the third translation two arrays: A_{pq} and A_{rs}.

However, closer inspection to the first translation in Fig. 1(b) reveals that the encoding is not sound. Since `p` and `q` may alias, the same array A_{pq} is passed to the 3rd and 4th arguments at the first callsite of f in g. The encoding is unsound because at the callsite we obtain a similar effect to:

$$H'_{pq} = H_{pq}[p \leftarrow 1] \text{ and } H'_{pq} = H_{pq}[q \leftarrow 2]$$

i.e., the update of p is lost, instead of the correct

$$H'_{pq} = H_{pq}[p \leftarrow 1] \text{ and } H''_{pq} = H'_{pq}[q \leftarrow 2]$$

where both updates of p and q are preserved.

This example shows that an arbitrary context-sensitive pointer analysis cannot be directly leveraged for modular verification without being unsound unless the analysis ensures the following correctness condition **CC**: *"no two disjoint memory regions modified in a function can be aliased at any particular call site"*.

A simple solution adopted by verifiers such as SMACK and SEAHORN is to give up the precision of a context-sensitive pointer analysis and use a context-insensitive one. The resulting translation is shown in Fig. 1(c). A more precise but incomplete approach adopted by Hubert and Marché [12] exploits context-sensitivity if **CC** holds and returns inconclusive results otherwise. Moy [17] refines Hubert and Marché's approach by generating function contracts that ensure that **CC** holds, rejecting programs for which it does not.

We argue that none of these solutions is fully satisfactory. Instead, our approach consists of reusing existing pointer analysis technology and adapting it to verification. Pointer analyses have been studied for decades and, thus, we want to leverage existing advances as much as possible. For this reason, our pointer analysis is inspired by *Data Structure Analysis (DSA)* [15]. DSA is a context, field-sensitive pointer analysis that represents the heap explicitly. Moreover, DSA supports *type-unsafe* C/C++ code and it scales to large code bases. However, context-sensitivity cannot be directly exploited because DSA does not ensure **CC**. We also observed that DSA is very imprecise when modeling consecutive sequences of bytes (e.g., C arrays) which complicates the verification of some programs such as our C++ case study. Last, but not less important, our experience with the public implementation of DSA [1] is that it is full of corner cases and it is very hard to reason about its correctness. Our goal is to develop

```
class X {
   X() {....}
};

class Y: public X {
   Y(): X() {....}
};

class Z: public X {
   Z(): X() {....}
};

Y* y = new Y();
Z* z = new Z();
```

```
% Constructor for Y
_Y_c(this) {
   _X_c(this);
   ...
}
% Constructor for Z
_Z_c(this) {
   _X_c(this);
   ...
}

% Y y = new Y();
y = _Znwm(sizeof(Y));
_Y_c(y);

% Z z = new Z();
z = _Znwm(sizeof(Z));
_Z_c(z);
```

Fig. 2: A C++ fragment and its translation to a low-level IR form.

a new pointer analysis that enjoys the same benefits as DSA while producing a sound and context-sensitive static partitioning of the heap required for verification. Unlike [15], we provide a more rigorous formalization of the analysis from which a proof of correctness can follow.

Coming back to our example, the side-effect free program obtained with our analysis is shown on Fig. 1(d). The translation is sound yet more precise than using a context-insensitive analysis since we do not merge the regions A_{pq} and A_{rs} passed to each call to `f`, and, thus, it is still context-sensitive. It is worth mentioning that the more different heap call patterns there are for a function, the fewer opportunities to partition the heap into smaller regions. This problem can be addressed by cloning functions with different call patterns. However, this is orthogonal to our approach and was not necessary during our experimental evaluation.

Although it is folklore that context-sensitivity is beneficial for analysis, it is particularly important for verification of C++ programs. Consider the snippet of C++ on the left in Fig. 2, where classes `Y` and `Z` are sub-classes of `X`. This fragment allocates memory for two objects of class `Y` and `Z`. We omit all class methods except the constructors. The constructors of `Y` and `Z` call the constructor of the base class `X`. We show on the right in Fig. 2 a translation to a low-level intermediate representation (IR) without C++ features that resembles LLVM [14] IR. Each object allocation (`_znwm` is the C++ mangled name for the `new` operator) is followed by a call to the corresponding constructor `_Y_c` and `_Z_c` which in turn calls `_X_c`. As a result, `_X_c` is called twice: one from `_Y_c` and another from `_Z_c`. The key observation is that a context-insensitive pointer analysis will merge `y` and `z` into the same alias set, collapsing the whole hierarchy into a large alias set. However, a context-sensitive pointer analysis will not merge them since it can distinguish between different calls to the constructor `_X_c`.

In summary, the paper makes the following contributions: (1) It presents a new context- and field-sensitive pointer analysis that (a) is based on a new concept of a *simulation relation* between graphs that allows to produce sound

$$
\begin{array}{lll}
S & ::= & p = \&\mathrm{x} \mid p = \mathbf{malloc}(...) \\
 & & p = q + (c \times m) + d \quad (c \geq 0) \\
 & & p = *\mathrm{q} \mid *\mathrm{p} = q \\
 & & S; S \\
 & & \mathsf{if}\ (Cond)\ \mathsf{then}\ S\ \mathsf{else}\ S \\
 & & \mathsf{while}\ (Cond)\ \mathsf{do}\ S \\
Cond & ::= & p = q \mid p \neq q
\end{array}
$$

Fig. 3: Syntax of pointer operations

modular verification conditions and, unlike DSA, reason formally about proof of correctness, and (b) is *array-sensitive*; (2) The analysis is implemented[4] operating on LLVM bitcode and integrated in SeaHorn; and (3) It has been evaluated on the flight control component of the Core Autonomous Safety Software (CASS) of an Autonomous Flight Safety System written in C++ and on C benchmarks from the Software Verification Competition (SV-COMP). We show that it reduces verification times by producing a finer-grained memory partitioning.

2 Syntax and Concrete Memory Model

Consider a simple imperative language shown in Fig. 3. It captures the core pointer arithmetic of C/C++ at the function level. The variables $p, q \in \mathcal{V_P}$ denote pointer variables, $m \in \mathcal{V_I}$ denotes integer variables, the symbols c, d denote integer constants, and x denotes either a pointer or integer variable. The set of program variables is denoted by $\mathcal{V}$. We restrict variables to pointer and integer types $\mathcal{V} = \mathcal{V_P} \cup \mathcal{V_I}$ and we assume that they are disjoint, $\mathcal{V_P} \cap \mathcal{V_I} = \emptyset$. The set of statements is denoted by $\mathbb{S}$. Each statement is assigned a unique label $\ell \in \mathbb{L}$.

A *cell* c is a pair (id, o) where id is a unique identifier of a memory object of size sz and o is byte offset in id, $0 \leq o < sz$. The set of all concrete objects is denoted by $\mathcal{O}_\mathbb{C}$ and we use $\mathcal{C}_\mathbb{C}$ to denote the set of all possible concrete cells. Note that the cardinality of $\mathcal{C}_\mathbb{C}$ is unbounded since the number of concrete objects in the heap is unbounded. Memory is represented by a *concrete points-to graph.* A concrete points-to graph is a triple $\langle V, E, \sigma \rangle$, where:

- $V \subseteq \mathcal{C}_\mathbb{C}$ is a set of concrete cells;
- $E \subseteq \mathcal{C}_\mathbb{C} \times \mathcal{C}_\mathbb{C}$ is a set of edges denoting that a source cell points to target cell.
- The environment $\sigma : \mathcal{V_P} \mapsto \mathcal{C}_\mathbb{C}$ maps pointer variables to cells. We write $\mathsf{dom}(\sigma)$ for the domain of σ, that is, the set of variables for which it is defined.

The symbol $\mathcal{G}_\mathbb{C}$ denotes the set of all concrete points-to graphs. Concrete graphs are *functional* in the sense that for each source cell there is at most one target cell. For convenience, we will write $c_2 = E(c_1)$ and $E(c_3) = c_4$ to refer to $c_1 \to c_2 \in E$ and $E \setminus \{c_3 \to \_\} \cup \{c_3 \to c_4\}$, respectively.

[4] It is publicly available at `https://github.com/seahorn/sea-dsa`

```
typedef struct list{
  struct list *n;
  int e;
} ll;
ll* mkList(int s,int e){
 if (s <= 0)
   return NULL;
 ll*p=malloc(sizeof(ll));
 p->e=e;
 p->n=mkList(s-1,e);
 return p;
}
void main(){
 ll* a[4]; int i;
 for(i=0;i<4;i++)
   a[i] = mkList(2,0);
}
```

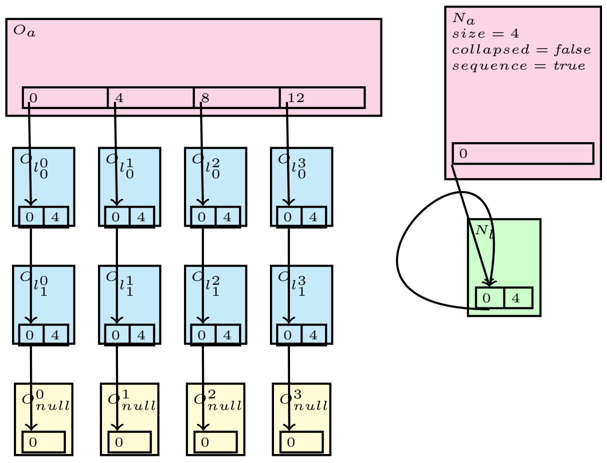


Fig. 4: C snippet (left), concrete (center), and abstract points-to graphs (right).

Example 1 (Concrete points-to graph). The middle of Fig. 4 depicts the cells and edges of a concrete points-to graph from a simplified fragment extracted from the CASS code. The program initializes each array element of size four to a new linked list of two memory objects which is dynamically allocated by the function mkList. The array is represented by a concrete object O_a of size 16 (assuming four bytes per pointer). The variable a points to the first cell $(O_a, 0)$. For each cell (O_a, i) $(i \in \{0, 4, 8, 12\})$ there is an edge to the first cell of each linked list: $(O^0_{l_0}, 0)$, $(O^1_{l_0}, 0)$, $(O^2_{l_0}, 0)$, and $(O^3_{l_0}, 0)$. The size of each list object is 8 (assuming integers occupy 4 bytes as well). Finally, each of the lists has an edge to a special concrete object $O^{i\in\{0,1,2,3\}}_{null}$ that represents a null terminator. ■

A *concrete state* is a triple $\langle g, \pi, l\rangle$ where $g \in \mathcal{G}_{\mathbb{C}}$ is a concrete points-to graph, $\pi : \mathcal{V}_{\mathcal{I}} \mapsto \mathbb{Z}$ is an environment mapping integer variables to values, and $l \in \mathbb{L}$ is the label of the next statement to be executed.

We do not give the concrete semantics of our language since it is standard. We only point out that the semantics of a *taken-address* variable (i.e., `p = &x`) creates a new memory object but always the same for the same variable, and **malloc** returns non-deterministically a fresh memory object from the (infinite) set of unallocated objects. Note that, *re-incarnation* of freed memory cannot be modeled in our semantics. For instance, in the following sequence of statements: `p = malloc(...); q = p; free(p); r=malloc(...)`, the pointer variable `r` cannot be aliased with `q`.

3 Abstract Memory Model

In this section, we present our abstract memory model via *abstract points-to graphs*. An abstract graph looks almost identical to a concrete graph with the major difference that the set of abstract objects is finite. The symbol $\mathcal{C}_{\mathbb{A}}$ denotes the set of abstract cells. An abstract cell is a pair of an identifier of an abstract object and a byte offset. The set of all abstract objects is denoted by $\mathcal{O}_{\mathbb{A}}$. An abstract points-to graph is a triple $\langle V^\sharp, E^\sharp, \sigma^\sharp \rangle$ where:

– $V^\sharp \subseteq \mathcal{C}_{\mathbb{A}}$ is a finite set of abstract cells.
– $E^\sharp \subseteq \mathcal{C}_{\mathbb{A}} \times \mathcal{C}_{\mathbb{A}}$ is a set of edges denoting points-to relations.
– The environment $\sigma^\sharp : \mathcal{V}_{\mathcal{P}} \mapsto \mathcal{C}_{\mathbb{A}}$ maps pointer variables to abstract cells.

An *abstract state* is represented by an abstract points-to graph. We will use the symbol $\mathbb{A}$ to denote the set of all abstract states. Note that our abstract semantics over-approximates the concrete semantics in three ways: (a) the set of objects is finite, (b) the abstract semantics is flow insensitive, and (c) it does not keep track of an environment for integer variables. To keep the number of objects finite, each abstract object maintains the following information:

– whether the abstract object is *collapsed* meaning that all cells from this object have been merged into a single one (i.e., field-sensitivity is lost). The function $\mathsf{isCollapsed} : \mathcal{O}_{\mathbb{A}} \mapsto \mathbb{B}$ serves to denote such objects;
– whether an abstract object represents a sequence of an unknown number of consecutive bytes (e.g., C arrays). The function $\mathsf{isSeq} : \mathcal{O}_{\mathbb{A}} \mapsto \mathbb{B}$ denotes such objects;
– a *size* that over-approximates the size of every concrete object represented by the abstract object. The function $\mathsf{size} : \mathcal{O}_{\mathbb{A}} \mapsto \mathbb{N}$ maps objects to their sizes:
  • if $\mathsf{isCollapsed}(n)$ then $\mathsf{size}(n) = 1$;
  • if $\mathsf{isSeq}(n)$ then $\mathsf{size}(n) = k$ represents that the actual size is some value in the set $\{k \times N \mid N \in \mathbb{N}, k > 0\}$ (i.e., positive multiple of k);
  • otherwise, it returns the value of the largest offset accessed so far plus the size of the field indexed by that offset.

  It is worth mentioning that the size of an abstract object is computed based on its *use* and not based on the allocation. Therefore, the values returned by size change during the analysis.

Example 2 (Abstract points-to graph). The right of Fig. 4 depicts the cells and edges of an abstract points-to graph. The whole array is abstracted to an object N_a marked as a sequence of consecutive bytes whose unknown size is multiple of 4 (e.g., $4, 8, 12, 16, 20, \ldots$). The abstraction loses the fact that the original array had exactly four elements of four bytes each. The linked lists are also abstracted. Each list is abstracted by a single object N_l that has two fields at offsets 0 and 4. There is a loop edge at $(N_l, 0)$ that means we have lost track of the exact list size. Moreover, the abstraction loses the fact that the linked lists are null-terminated. Note that in spite of this loss of precision, this abstraction is able to preserve the fact that the array and the linked lists point to two disjoint memory regions which is a very valuable information while solving the verification conditions. ■

Simulation relation between graphs. We now introduce the fundamental concept of a simulation relation between points-to graphs. First, we define two helper functions. Given an object n' and a set of edges E, $\mathsf{Links}(n', E)$ returns a sequence of numerical offsets $(o_n)_{n\in\mathbb{N}}$ such that $(o_n) = o$ if $(n', o) \to c \in E$. Function $\oplus_n : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$ adds numerical offsets and adjusts them depending on object n flags as follows:

$$o_1 \oplus_n o_2 = \begin{cases} 0 & \text{if } \mathsf{isCollapsed}(n) \\ (o_1 + o_2) \ \% \ \mathsf{size}(n) & \text{if } \mathsf{isSeq}(n) \\ o_1 + o_2 & \text{otherwise} \end{cases}$$

We say that there is a *simulation relation* between two abstract graphs $\langle V_1^\sharp, E_1^\sharp, \sigma_1^\sharp\rangle$ and $\langle V_2^\sharp, E_2^\sharp, \sigma_2^\sharp\rangle$ if $\exists \rho \subseteq \mathcal{C}_\mathbb{A} \times \mathcal{C}_\mathbb{A}.\ \forall p \in \mathsf{dom}(\sigma_1^\sharp).\ (\sigma_1^\sharp(p), \sigma_2^\sharp(p)) \in \rho$, and for all $((n_1, o_1), (n_2, o_2)) \in \rho$:

– if $(o_1 \leq o_2 \wedge o_1 > 0)$ then $((n_1, 0), (n_2, o_2 - o_1)) \in \rho$

– else

$$\begin{array}{ll} ((o_1 \leq o_2) \wedge o_1 = 0) & \wedge \\ \mathit{compatible}(n_1, n_2, o_2) & \wedge \\ \forall o \in \mathsf{Links}(n_1, E_1^\sharp).(E_1^\sharp((n_1, o)), E_2^\sharp((n_2, o_2 \oplus_{n_2} o))) \in \rho & \end{array}$$

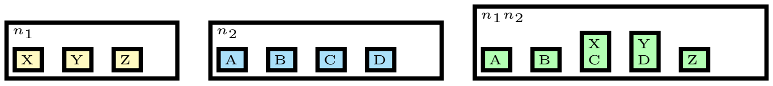
where

$$\mathit{compatible}(n_1, n_2, o) = \begin{cases} \mathit{true} & \text{if } (\mathsf{isCollapsed}(n_2)) \\ \mathit{false} & \text{if } (\mathsf{isCollapsed}(n_1)) \\ \mathit{false} & \text{if } (\mathsf{isSeq}(n_1) \wedge \neg\, \mathsf{isSeq}(n_2)) \\ (o = 0 \wedge & \text{if } (\mathsf{isSeq}(n_2)) \\ \ \mathsf{size}(n_1) \geq \mathsf{size}(n_2) \wedge & \\ \ \mathsf{size}(n_1) \ \% \ \mathsf{size}(n_2) = 0) & \\ n_1 = n_2 \implies o = 0 & \text{otherwise} \end{cases}$$

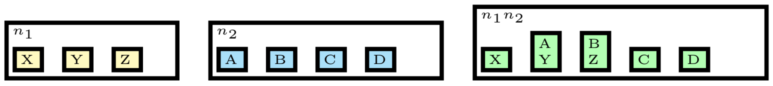
We can adapt the definition of a simulation relation to a relation between a concrete and abstract graph. For that, we only need to extend $\mathsf{isCollapsed}(n)$ and $\mathsf{isSeq}(n)$ to return *false* for any concrete object n, and let $\mathsf{size}(n)$ denote the allocated size of n[5]. Given a concrete graph $g_c \in \mathcal{G}_\mathbb{C}$ and an abstract graph $g_a \in \mathcal{G}_\mathbb{A}$, we use the notation $g_c \preceq g_a$ to say that there is a simulation relation between g_c and g_a.

Concretization and ordering of abstract graphs. The meaning of an abstract graph is given by the function $\gamma : \mathcal{G}_\mathbb{A} \mapsto 2^{\mathcal{G}_\mathbb{C}}$ and it is defined as $\gamma(g_a) = \{g_c \in \mathcal{G}_\mathbb{C} \mid g_c \preceq g_a\}$. Moreover, a simulation relation defines an ordering between abstract graphs such that $g \sqsubseteq_{\mathcal{G}_\mathbb{A}} g'$ if and only if there exists a simulation relation between g and g'.

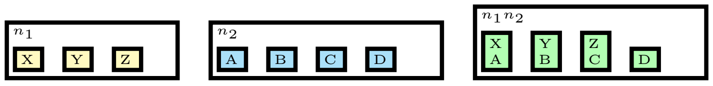
[5] For simplicity, we choose not to modify the definition of a concrete object to include its size.



(a) Unify $Y = (n_1, 1)$ and $D = (n_2, 3)$



(b) Unify $Z = (n_1, 2)$ and $B = (n_2, 1)$



(c) Unify $Y = (n_1, 1)$ and $B = (n_2, 1)$

Fig. 5: Unification examples with all fields of size one.

Remark. The concept of a simulation relation between abstract graphs also plays an essential role for the analysis of procedures (see Section 4).

Example 3 (Simulation relation between a concrete and abstract graph). Coming back to Fig. 4, let the environments $\{p \mapsto (O_a, 0)\}$ and $\{p \mapsto (N_a, 0)\}$ together with the cells and edges shown in Fig. 4 (center) and (right), form the concrete g_C and abstract g_A graphs, respectively. There is a simulation relation ρ between g_C and g_A, defined as follows:

$$\begin{aligned}\rho = \{ & ((O_a, 0), (N_a, 0)),\\ & ((O_{l_0^0}, 0), (N_l, 0)), \ldots, ((O_{l_0^3}, 0), (N_l, 0)),\\ & ((O_{l_1^0}, 0), (N_l, 0)), \ldots, ((O_{l_1^3}, 0), (N_l, 0)),\\ & ((O^0_{null}, 0), (N_l, 0)), \ldots, ((O^3_{null}, 0), (N_l, 0)) \}\end{aligned}$$

To show that ρ is a simulation relation, it suffices to show that $((O_a, 0), (N_a, 0)) \in \rho$. Since $o_1 = 0$ and $o_2 = 0$ the "else" branch is triggered. The first conjunct holds trivially. Then, we need to check whether *compatible*$(O_a, N_a, 0)$ holds. Since O_a is a concrete object, we have that $\mathsf{isSeq}(O_a) = false$, but $\mathsf{isSeq}(N_a) = true$. Thus, the forth "if" in the definition of *compatible* is applicable. Since $\mathsf{size}(O_a) = 16$ and $\mathsf{size}(N_a) = 4$ the condition holds. Next, we check that ρ includes the pairs of cells $(E((n_1, 0)), E((n_2, 0 \oplus_{n_2} 0)))$,..., $(E((n_1, 12)), E((n_2, 12 \oplus_{n_2} 0)))$, which, after substitution, become $((O_{l_0^0}, 0), (N_l, 0))$,..., $((O_{l_0^3}, 0), (N_l, 0))$. We only show that $((O_{l_0^0}, 0), (N_l, 0)) \in \rho$ since the other pairs are identical. Again, the "else" branch is triggered together with the forth "if" from the *compatible* definition but with $\mathsf{isSeq}(O_{l_0^0}) = false$, $\mathsf{size}(O_{l_0^0}) = 8$, $\mathsf{isSeq}(N_l) = true$, and $\mathsf{size}(N_l) = 8$. Thus, the conditions hold. Finally, we need to check that $E((O_{l_0^0}, 0)) \equiv (O_{l_1^0}, 0)$ is simulated by $E((N_l, 0)) \equiv (N_l, 0)$. Hence, we can conclude that g_C is a concretization of g_A. ■

Another key concept in our memory model is an operation $\mathsf{unifyCells}$ that given two cells (n_1, o_1) and (n_2, o_2) creates a new cell (and possibly a new object) (n_3, o_3) that abstracts them both (i.e., (n_3, o_3) simulates both (n_1, o_1) and

```
unifyCells($(n_1, o_1), (n_2, o_2), g$)
  if ($o_1 < o_2$)
      unifyNodes ($n_1, n_2, o_2 - o_1, g$)
  elif ($o_2 < o_1$)
      unifyNodes ($n_2, n_1, o_1 - o_2, g$)
  else
      unifyNodes ($n_1, n_2, 0, g$)

% Embed in-place object $n_1$ into object $n_2$
% at offset $o$
unifyNodes($n_1, n_2, o, g$)
  $o' = o \oplus_{n_2} 0$
  if (isCollapsed($n_1$) $\wedge \neg$ isCollapsed($n_2$))
      return collapse($n_1, n_2, o', g$)
  elif $\neg$ isCollapsed($n_1$) $\wedge \neg$ isCollapsed($n_2$)
      if (isSeq($n_1$) $\wedge \neg$ isSeq($n_2$))
          if ($o' = 0$)
            return unifyNodes($n_2, n_1, 0, g$)
          else
            return collapse($n_1, n_2, o', g$)
      elif ($\neg$ isSeq($n_1$) $\wedge$ isSeq($n_2$))
          if (size($n_1$) % size($n_2$) $= 0$)
              isSeq($n_1$) $=$ *true*
              return unifyNodes($n_1, n_2, o, g$)
          elif (size($n_1$) $+ o' >$ size($n_2$))
              return collapse($n_1, n_2, o', g$)
      elif (isSeq($n_1$) $\wedge$ isSeq($n_2$))
          if (size($n_1$) $<$ size($n_2$))
              return unifyNodes($n_2, n_1, 0, g$)
          elif ((size($n_2$) % size($n_1$) $\neq 0$) $\vee$
                ($o \oplus_{n_1} 0 > 0$))
              return collapse($n_1, n_2, o', g$)
  if ($n_1 = n_2 \wedge o' > 0$) collapseNode($n_2, g$)
  redirectEdges($n_1, n_2, o', g$)

% Move all edges from/to $n_1$'s cells
% to $n_2$ and destroy $n_1$
redirectEdges($n_1, n_2, o, \langle V^\sharp, E^\sharp, \sigma^\sharp \rangle$)
  foreach $c_1 \to (n_1, i) \in E^\sharp$
      $E^\sharp(c_1) = (n_2, o \oplus_{n_2} i)$
      foreach $p \in$ dom($\sigma^\sharp$)$\wedge$
                  $\sigma^\sharp(p) = (n_1, i)$
          $\sigma^\sharp[p \mapsto (n_2, o \oplus_{n_2} i)]$
  foreach $(n_1, i) \to c \in E^\sharp$
      if $(n_2, o \oplus_{n_2} i) \to c' \in E^\sharp$
          unifyCells($c, c', \langle V^\sharp, E^\sharp, \sigma^\sharp \rangle$)
      else
          $E^\sharp((n_2, o \oplus_{n_2} i)) = c$
  $V^\sharp = V^\sharp \setminus \{n_1\}$
  $E^\sharp = E^\sharp \setminus \{e \in E^\sharp \mid e$ and $n_1$
                  are incident$\}$

% $n$ loses all its field-sensitivity
collapseNode(n, $\langle V^\sharp, E^\sharp, \sigma^\sharp \rangle$)
  $c =$ (mkNode(), 0)
  foreach $o \in$ Links($n, E^\sharp$)
      unifyCells($c, (n, o), \langle V^\sharp, E^\sharp, \sigma^\sharp \rangle$)
      $E^\sharp = E^\sharp \setminus \{(n, o) \to \_\}$
  size($n$) $= 1$
  isCollapsed($n$) $=$ *true*
  $E^\sharp((n, 0)) = c$

% Collapse $n_2$ and redirect $n_1$ to $n_2$
collapse($n_1, n_2, o, g$)
  collapseNode($n_2, g$)
  redirectEdges($n_1, n_2, o, g$)
```

Fig. 6: Unification of two cells

$(n_2, o_2))$. Conceptually, the operation replaces the objects n_1 and n_2 by a new object that is an abstraction of both of them. In practice, this is done by destructively updating the objects in the points-to graph. The pseudo-code for unifyCells is given in Fig. 6. We use unifyCells to combine cells in the graph and to combine points-to graphs as needed by the abstract semantics.

The idea is to *embed* one cell into another, possibly modifying them and all their reachable cells in the graph. If this is not possible then the cells are collapsed and field-sensitivity is lost. Given two cells $c_1 = (n_1, o_1)$ and $c_2 = (n_2, o_2)$ let us assume that $o_1 < o_2$ as shown in Fig. 5(a). The solution is to embed object n_1 in n_2 such that the memory location referred to by (n_2, o_2) is the same as that referred to by (n_1, o_1). This is achieved by adjusting $(n_1, 0)$ to $(n_2, o_2 - o_1)$. Then, each (n_1, o_i) can be mapped to $(n_2, o_2 - o_1 + o_i)$. In particular, (n_1, o_1) becomes (n_2, o_2). The case where $o_2 < o_1$ is symmetric (Fig. 5(b)), otherwise, if $o_1 = o_2$ the we can choose arbitrarily to embed one into the other (Fig. 5(c)). unifyCells at lines 1-6 decides which cell is embedded into and the offset adjustments.

unifyNodes unifies in place all cells of n_1 starting at offset 0 with all cells of n_2 starting at offset o. If possible it will redirect all the incoming edges of n_1 to n_2 by shifting them by o and unify recursively all the outgoing edges (see redirectEdges). After that, it destroys n_1 (and all its cells) and keeps n_2. The pseudo-code is more involved because it also checks for conditions where the objects must be collapsed:

1. If n_1 is collapsed but n_2 is not (Line 8): we need to collapse n_2 and redirect edges.
2. If n_1 is a sequence but n_2 is not (Line 11): check if we can embed the non-sequence n_2 into the sequence object n_1, which is handled by the next case.
3. If n_1 is a non-sequence and n_2 is a sequence (Line 16): try to modify n_1 into a sequence object and continue with the unification. We will explain this case through an example. Assume the size of n_2 is 4, meaning that it represents a memory region of size $4 \times C$ for some constant C. If the size of n_1 is divisible by 4, then we can convert n_1 into a sequence. If the size of n_1 is smaller than 4, we can embed it into n_2 by redirecting all the edges. If neither of the two conditions hold, we collapse n_1 and n_2.
4. if n_1 and n_2 are both sequences (Line 22): try to embed the larger one into the smaller one. We might collapse if the sizes of the sequences are not compatible or if we try to unify at non-zero offsets.

Assume a functional version of unifyCells denoted as $\mathsf{unifyCells}^{\mathsf{F}}$ that copies the input graph into g', perform unifyCells on g', and returns g'. The correctness of our approach follows from the following result.

Lemma 1. *For all abstract graphs $g \equiv \langle V^\sharp, E^\sharp, \sigma^\sharp \rangle \in \mathcal{G}_{\mathbb{A}}$ and for all $c, c' \in V^\sharp$. Let g' be the result of $\mathsf{unifyCells}^{\mathsf{F}}(c, c', g)$. Then, there always exists a simulation relation between g and g'.*

Proof. By structural induction over unifyCells and unifyNodes functions.

This lemma says that the result of unifying two cells produces always a new abstract graph whose concretization is always a superset of the concretization of graph before the unification took place. Moreover, note that once our analysis decides to merge two cells they can never be split again. This is also true during the analysis of function calls.

We can now define the abstract semantics for our core pointer language in Fig. 7 by means of the function $[\![\cdot]\!]_{\mathbb{A}} : \mathbb{S} \mapsto (\mathbb{A} \mapsto \mathbb{A})$. We only show the abstract semantics of the atomic pointer operations and postpone the analysis of function calls to the next section.

The analysis creates a new object the first time a taken-address variable (line 50) is accessed or when a new allocation occurs (line 54). In this case, a

```
$[\![S]\!]_{\mathbb{A}}(\langle V^\sharp, E^\sharp, \sigma^\sharp\rangle)$
49: switch $(S)$
50:     $l : p = \&\mathrm{x}$:
51:         $n = \mathsf{mkNode}(x)$
52:         $\mathsf{unify}(p, (n, 0), \langle V^\sharp, E^\sharp, \sigma^\sharp\rangle)$
53:         return $\langle V^\sharp \cup \{n\}, E^\sharp, \sigma^\sharp\rangle$
54:     $l : p =$ malloc$(\ldots)$:
55:         $n = \mathsf{mkNode}(l)$
56:         $\mathsf{unify}(p, (n, 0), \langle V^\sharp, E^\sharp, \sigma^\sharp\rangle)$
57:         return $\langle V^\sharp \cup \{n\}, E^\sharp, \sigma^\sharp\rangle$
58:     $l : p = q + (c \times m) + d$:
59:         $(n, o) = \sigma^\sharp(q)$
60:         if $(\mathsf{isCollapsed}(n))$
61:             $\mathsf{collapsePointer}(p, \langle V^\sharp, E^\sharp, \sigma^\sharp\rangle)$
62:         elif $(c = 0 \wedge \neg\, \mathsf{isSeq}(n))$
63:             $sz_{field} = \mathsf{sizeof}(\mathsf{typeof}(n, o + d))$
64:             $sz = o + d + sz_{field}$
65:             $\mathsf{updateSize}(n, sz, \langle V^\sharp, E^\sharp, \sigma^\sharp\rangle)$
66:             $\mathsf{unify}(p, (n, o + d), \langle V^\sharp, E^\sharp, \sigma^\sharp\rangle)$
67:         else
68:             $\mathsf{isSeq}(n) = \mathit{true}$
69:             $sz_{gcd} = \mathsf{gcd}(\mathsf{size}(n), c)$
70:             $\mathsf{updateSize}(n, sz_{gcd}, \langle V^\sharp, E^\sharp, \sigma^\sharp\rangle)$
71:             $\mathsf{unify}(p, (n, o), \langle V^\sharp, E^\sharp, \sigma^\sharp\rangle)$
72:         break
73:     $l : q = *\mathrm{p}$:
74:     $l : *\mathrm{p} = q$:
75:         $\mathsf{unifyCells}(E^\sharp(\sigma^\sharp(p)), \sigma^\sharp(q), \langle V^\sharp, E^\sharp, \sigma^\sharp\rangle)$
76:     default:
77: return $\langle V^\sharp, E^\sharp, \sigma^\sharp\rangle$

$\mathsf{unify}(p, c, \langle V^\sharp, E^\sharp, \sigma^\sharp\rangle)$
78: if $p \notin \mathsf{dom}(\sigma^\sharp)$
79:     $\sigma^\sharp[p \mapsto c]$
80: else
81:     $\mathsf{unifyCells}(\sigma^\sharp(p), c, \langle V^\sharp, E^\sharp, \sigma^\sharp\rangle)$

$\mathsf{collapsePointer}(p, g)$
82: $n = \mathsf{mkNode}()$
83: $\mathsf{collapseNode}(n, g)$
84: $\mathsf{unify}(p, (n, 0), g)$

$\mathsf{updateSize}(n, sz, g)$
85: if $(\mathsf{isSeq}(n) \wedge \mathsf{size}(n) \neq sz)$
86:     $\mathsf{collapseNode}(n, g)$
87: elif $(\neg\mathsf{isSeq}(n) \wedge sz > \mathsf{size}(n))$
88:     $\mathsf{size}(n) = sz$
```

Fig. 7: Abstract semantics of the atomic pointer operations.

fresh cell consisting of the new object and a zero-offset is unified with the cell of the left-hand side (if any). The analysis of a memory load (line 73) or store (line 74) is done in the same way: the dereference of a pointer p requires us to find the target cell of the edge whose source is the cell of p and unify it with the cell of the non-dereferenced pointer (line 75). The analysis of pointer casts and pointer arithmetic at line 58 is more involved and consists of three main cases:

1. *(collapse)* if the object of the base pointer q is already collapsed then the object of the left-hand side is also collapsed;
2. *(pointer casts, address of a struct field or a constant array index)* if the object of q is not a sequence and its offset can be statically determined (i.e., the statement can be simplified to $p = q + d$) then the cell of p is unified to the cell $(n, o + d)$ where (n, o) is the cell of q. Moreover, the size of n might

grow if the accessed offset $o + d$ plus the size of the indexed field is greater than its current size.

3. *(address of a symbolic array index)* the cell of q is unified with the cell of p after marking the object of q as a sequence. Again, we might need to update the size of the object of q. Since the object is a sequence, the new size is the *greatest common divisor (gcd)* of the old size and c.

Finally, we lift $[\![S]\!]_{\mathbb{A}}(g)$ to a function F, denoted by $[\![F]\!]_{\mathbb{A}}(g)$, as the application of $[\![S]\!]_{\mathbb{A}}$ to each statement S in F starting from the abstract graph g.

Example 4 (Comparing with DSA). Consider the program in Fig. 4. The DSA algorithm described in [15] is array-insensitive. DSA creates an object N_a after the stack allocation `ll* a[4]` and since the allocation size is statically known it decides that the size of N_a is 16 (assuming 4 bytes per pointer). Then, during the analysis of `a[i]=mkList(...)` it notices that there is a symbolic access to object N_a at some unknown offset. Although it knows that the size of the accessed element is 4, it loses all field-sensitivity since $4 \neq 16$.

After the array allocation our analysis creates a cell $(N_a, 0)$ where $\mathsf{isSeq}(N_a) = \mathsf{isCollapsed}(N_a) = \mathit{false}$ and $\mathsf{size}(N_a) = 16$. Next, the analysis `a[i]=mkList(...)` requires us to compute the address p_a of the array index which is translated to $p_a = a + (4 \times i) + 0$. Since the size of the array index is $c = 4 \neq 0$, the else branch at line 67 in Fig. 7 is applicable. As a result, $\mathsf{isSeq}(N_a) = \mathit{true}$ and $\mathsf{size}(N_a) = \mathsf{gcd}(16, 4) = 4$. Thus, our analysis first assigns a fixed size to N_a after its allocation (similar to DSA) but then it decides by its use that N_a is a non-empty sequence of bytes whose size is divisible by 4.

The same result would be obtained if the loop was unrolled. However, if the size of the array was not known then DSA would not have collapsed the abstract object. However, the pattern exemplified by our snippet of having a small array simulating a struct is actually common in our C++ case study and there the impact of collapsing is much worse since this kind of arrays are embedded in complex C++ objects. ■

4 A Context-Sensitive Abstract Memory Model

In this section, we describe how we extend the intra-procedural pointer analysis described in Section 3 to support more precisely function calls so that it can be leveraged to produce a side-effect free form useful for verification.

We do not define the syntax and concrete semantics of function calls as they are standard. We fix some notation and define some helper functions needed by our analysis. For a given function call (or *callsite*) cs we refer to f_{callee} and f_{caller} as the callee function and the function where cs is executed, respectively. We assume functions $\mathsf{callee}(cs)$ and $\mathsf{caller}(cs)$ that return f_{callee} and f_{caller}, respectively. We also assume functions $\mathsf{formals}(cs)$ and $\mathsf{actuals}(cs)$ that return the formal parameters of f_{callee} and the actual parameters of the call, respectively.

Our algorithm works in three phases. We start by describing the first two phases of the algorithm which are more straightforward and postpone the third

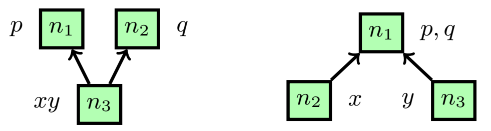


Fig. 8: Problematic aliasing patterns when `f(p,q)` calls to `f(x,y)`

one to the end of the section. The procedure cloneSummaries on the bottom left in Fig. 9 describes the first and second phase of our inter-procedural algorithm. First, each function f is analyzed in isolation while its abstract graph (a.k.a *function summary*) is computed. This phase is the only one that analyzes f's statements. If there is a recursive call then all the functions in the same *Strongly Connected Component (SCC)* will be analyzed in an intra-procedural manner. This has not been a problem in practice since our benchmarks have few recursive functions (e.g., our C++ case study has no recursive calls). Second, the *call graph* is traversed in reverse topological order exploring all callees before their callers. At each callsite the callee's graph is *cloned* into the caller's abstract graph while the objects of the formal and actual parameters are unified. This is done by the procedure cloneAndUnifyCells.

Therefore, *heap cloning* is how our analysis achieves context-sensitivity while being fully modular. Coming back to Fig. 1(a). Although `p` and `q` are aliased, they are disjoint from `r` and `s`. Thus, our analysis can distinguish each call to `f` without merging the cells of the first call to `f` (`p` and `q`) with the cells of the second call (`r` and `s`).

After completion of cloneSummaries, each cell pointed by the callee's formal parameters can be aliased with at most one cell from the caller's actual parameters and thus, scenarios such as the one illustrated on the left in Fig. 8 are not possible[6]. The pattern on the right is still possible: a cell in the caller can be the target of two different cells in the callee (i.e., **CC** does not hold). The example in Fig. 1(b) illustrated how this situation precludes pointer analyses to be used for obtaining sound side-effect-free programs.

The main purpose of the third phase of our algorithm is to ensure that the two patterns depicted in Fig. 8 are not possible. This phase is done by the **while** loop in procedure CSAnalysis on the left in Fig. 9.

At this point, the analysis has cloned all summaries by calling the procedure cloneSummaries (line 1). As a result, each cell originated from the callee's formal parameters cannot be mapped to two distinct cells in the caller graph. The reason is that cloneSummaries calls unifyCells for each pair of formal-actual parameters. On the left in Fig. 8, the cell $(n_3, 0)$ is unified with $(n_1, 0)$ from the pair (x, p). Then, the cell $(n_3, 0)$ is unified with $(n_2, 0)$ from the other pair (y, q). After these unifications, x, y, p, q point to the same cell.

Next, we scan all the function calls in the program and build a simulation relation $\mathcal{R}$ between the f_{callee}'s graph and f_{caller}'s graph but only considering

[6] For simplicity, we assume in Fig. 8 all cells have zero offsets.

```
CSAnalysis($cg$)
1:  cloneSummaries($cg$)
2:  $W = \{cs \mid cs \in cg\}$
3:  while $W \neq \emptyset$
4:      $W = W \setminus \{cs\}$
5:      $g \equiv \langle \_, \_, \sigma^\sharp_{callee} \rangle = G[\mathsf{callee}(cs)]$
6:      $g' \equiv \langle \_, \_, \sigma^\sharp_{caller} \rangle = G[\mathsf{caller}(cs)]$
7:      $C = \{\sigma^\sharp_{callee}(p) \mid p \in \mathsf{formals}(cs)\}$
8:      $C' = \{\sigma^\sharp_{caller}(p) \mid p \in \mathsf{actuals}(cs)\}$
9:      if (propagate(formals($cs$), $g, g'$) $=\downarrow$)
10:         cloneAndUnifyCells($C', g', C, g$)
11:         $W \cup=$ uses(callee($cs$)) $\cup$
                      defs(callee($cs$))
12:     elif (propagate(formals($cs$), $g, g'$) $=\uparrow$)
13:         cloneAndUnifyCells($C, g, C', g'$)
14:         $W \cup=$ uses(caller($cs$)) $\cup$
                      defs(caller($cs$))

cloneSummaries($cg$)
15: foreach $scc$ in reverse topological
            order of $cg$
16:     $g_{scc} = \langle \emptyset, \emptyset, [] \rangle$
17:     foreach ($f \in scc$)   $[\![f]\!]_{\mathbb{A}}(g_{scc})$
18:     foreach ($f \in scc$)   $G[f] = g_{scc}$
19:     foreach callsite $cs \in scc$
20:         $\langle V^\sharp_1, E^\sharp_1, \sigma^\sharp_1 \rangle = G[\mathsf{callee}(cs)]$
21:         $\langle V^\sharp_2, E^\sharp_2, \sigma^\sharp_2 \rangle = G[\mathsf{caller}(cs)]$
22:         $C_1 = \{\sigma^\sharp_1(p) \mid p \in \mathsf{formals}(cs)\}$
23:         $C_2 = \{\sigma^\sharp_2(p) \mid p \in \mathsf{actuals}(cs)\}$
24:         cloneAndUnifyCells($C_1, \langle V^\sharp_1, E^\sharp_1, \sigma^\sharp_1 \rangle,$
                               $C_2, \langle V^\sharp_2, E^\sharp_2, \sigma^\sharp_2 \rangle$)

cloneAndUnifyCells($C_1, g_1, C_2, g_2$)
25: foreach $((n_1, o_1) \in C_1)$
26:     cloneNode($n_1, g_1, g_2$)
27: foreach $(c_1 \in C_1 \wedge c_2 \in C_2)$
28:     unifyCells($c_1, c_2, g_2$)

% Clone object $n$ into graph $tg$
cloneNode($n, \langle V^\sharp, E^\sharp, \sigma^\sharp \rangle, \langle V^\sharp_{tg}, E^\sharp_{tg}, \sigma^\sharp_{tg} \rangle$)
29: if $(n \in V^\sharp_{tg})$ return
30: $V^\sharp_{tg} = V^\sharp_{tg} \cup \{n\}$
31: foreach $(p \in \mathsf{dom}(\sigma^\sharp) \wedge \sigma^\sharp(p) = (n, o))$
32:     $\sigma^\sharp_{tg}[p \mapsto (n, o)]$
33: foreach $((n, o) \rightarrow (n', o')) \in E^\sharp$
34:     cloneNode($n', \langle V^\sharp, E^\sharp, \sigma^\sharp \rangle,$
                  $\langle V^\sharp_{tg}, E^\sharp_{tg}, \sigma^\sharp_{tg} \rangle$)
35:     $E^\sharp_{tg}(n, o) = (n', o')$

propagate($roots, g, g'$)
36: Let $\mathcal{R}$ be a simulation relation between
    $g$ and $g'$ projected onto $roots$
37: if $\mathcal{R}$ is an injective function
38:     return none
39: elif $\mathcal{R}$ is a function
40:     return $\downarrow$
41: else
42:     return $\uparrow$
```

Fig. 9: Our Context-Sensitive Pointer Analysis for Verification. cg is the program call graph and G is a global variable that maps functions to abstract graphs.

cells[7] that are reachable from f_{callee}'s actual parameters. A key insight is that we can use $\mathcal{R}$ to find out whether two distinct cells in the callee can be mapped to the same cell in the caller ($\mathcal{R}$ is not an injective function). This is the case depicted on the right in Fig. 8. If this is not possible (i.e., $\mathcal{R}$ is an injective function) then we are done with this callsite. Otherwise, we require a *top-down propagation* by unifying cells from f_{caller}'s actual parameters with the ones from f_{callee}'s formals parameters (line 9). For instance, in Fig. 1(a) the call `f(p,q)` forces our analysis to unify the cells of `x` and `y` with the cell of `p` and `q` resulting in a single array variable A_{xy} in the translation in Fig. 1(d). After a top-down propagation occurs, it is possible that a cell in the callee is now mapped to more

[7] In fact, we only need to consider cells that can be modified. Our implementation considers this optimization.

than one cell in the caller. The simulation relation between callee and caller is then not a function anymore. This can be fixed by performing a *bottom-up propagation* that unifies the cells of f_{callee}'s formal parameters and f_{caller}'s actual parameters (line 12). Either way, after unification occurs during the analysis of a callsite, we must update our worklist by adding other callsites that might be affected. If a top-down (bottom-up) propagation was performed then we need to revisit all the function calls where f_{callee} (f_{caller}) is the callee (uses) and all callsites defined in f_{callee} (f_{caller}) denoted by defs.

Finally, it is worth mentioning that this propagation phase does not affect the modularity of our analysis since only cells from summary graphs and those involved at the callsites are considered. Each function is still analyzed only once by the cloneSummaries procedure.

Correctness and termination. Upon completion for every callsite the simulation relation between the callee and caller graph is an injective function. This condition suffices to ensure that no two cells in a function can be equal for any call. Termination is straightforward since in the worst case the algorithm will merge all the cells between callee's formal parameters and caller's actual parameters. Since the number of cells is finite, this process must terminate.

Limitations. We assume that programs are *complete* and thus, every callee function is known (or *resolved*) at compile time. Although resolving callsites is non-trivial, we consider it an orthogonal problem. We can apply the solution adopted by DSA [15] based on a top-down traversal of the call graph after the bottom-up one. A less precise but simpler solution is to replace each unresolved call by a non-deterministic call to one of the possible functions whose type signature match. For our experiments, the latter solution was precise enough.

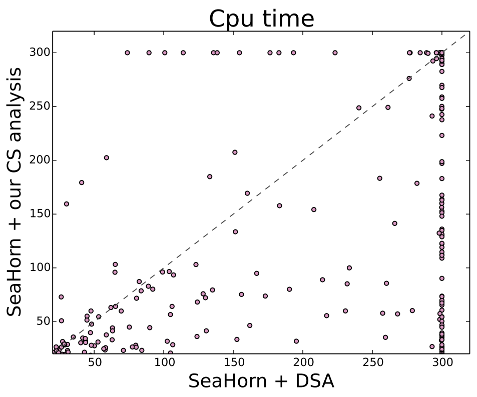
5 Experimental Evaluation

We have implemented our pointer analysis[8] with full support for LLVM bitcode and integrated it in SeaHorn [2]. For comparison, we have also integrated in SeaHorn the public implementation of DSA [1]. We use the context-insensitive version of DSA since we cannot use its context-sensitivity without being unsound. Note that the precision of SeaHorn (i.e., number of false positives) does not depend on the underlying pointer analysis. However, its scalability will be greatly affected as our results show. All experiments were carried out on a 3.5GHz Intel Xeon processor with 16 cores and 64GB on a Linux machine.

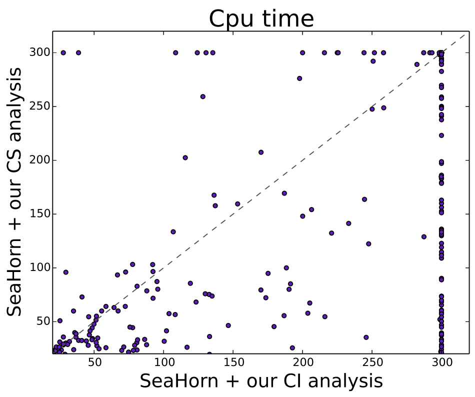
Results on C SV-COMP benchmarks. We ran SeaHorn on all 2,326 programs from the SV-COMP'17 sub-category DeviceDrivers64[9]. This collection of programs corresponds to Linux device drivers and its verification requires low-level modeling of pointers.

[8] The pointer analysis is available from `https://github.com/seahorn/sea-dsa`.

[9] Accessed `https://github.com/sosy-lab/sv-benchmarks` with sha `879e141f11348e49591738d3e11793b36546a2d5`



(a)



(b)

Fig. 10: CPU time spent by SeaHorn on DeviceDrivers64: (a) comparing DSA (SeaHorn + DSA) and our analysis (SeaHorn + our CS analysis), and (b) comparing our analysis without (CI) context-sensitivity with (CS) context-sensitivity.

Fig. 10(a) compares the impact of using DSA (x-axis) and our analysis (y-axis) on the CPU time spent by SeaHorn with a timeout of 5 minutes and 4GB memory limit. The scattered plot shows that in the majority of the cases, our analysis speeds up verification. Moreover, using our analysis SeaHorn was able to prove 81 more programs. Nevertheless, the plot indicates that our analysis is not always more beneficial than DSA. We investigated whether this occasional negative impact was due to timeouts in our pointer analysis. We compared the analysis time of DSA and our pointer analysis and differences were negligible. Both analyses were able to analyze each program in less than two seconds. Another possible explanation is in the use of Spacer [13] as back end in SeaHorn. Spacer is an SMT-based model checker and, thus, regardless of the memory model it always depends on the unpredictable nature of SMT solvers. We suspect that having more array variables (produced by our analysis) may have a negative impact if they are irrelevant to the property.

Since DSA and our analysis are different implementations, we also compare our analysis with and without context-sensitivity. Fig 10(b) shows the results of this comparison. We observe again that context-sensitivity often boosts SeaHorn during the verification process. Moreover, this comparison suggests our implementation is robust since the results are consistent with the previous experiment.

Case study: checking buffer overflow in C++ CASS code. We have evaluated our analysis to verify absence of buffer overflows on the flight control system of the Core Autonomous Safety Software (CASS) of an Autonomous

	Nodes	Collapsed	Max Node Density	Safe Alloc. Sites	Time (s)
SeaHorn + DSA	258	49%	0.8	49	33,119
SeaHorn + our CS	12,789	4%	0.13	73	31,102

Table 1: Preliminary results on proving absence of buffer overflows in CASS.

Flight Safety System. CASS[10] is written in C++ using standard C++ 2011 and following MISRA C++ 2008. It follows an object-oriented style and makes heavy use of dynamic arrays and singly-linked lists. CASS uses two custom libraries: one to manipulate strings and another for using associative containers (`map`). We did not consider these two libraries for our evaluation.

Instead of proving that all memory accesses are in-bounds by running SeaHorn once on an instrumented program[11] with all the properties, we split the set of accesses into multiple subsets so that we can run in parallel multiple instances of SeaHorn on a smaller number of memory accesses. We first identify all heap and stack allocation sites in the program. Each allocation site is a unique identifier of each LLVM instruction that allocates memory (e.g., `alloca`, `malloc`, etc.). Let k be the (finite) number of allocation sites. We run k instances of SeaHorn on an instrumented program that only checks for memory accesses from regions that are allocated by the allocation site of interest. This information is provided conservatively by the pointer analysis. The purpose of this methodology is twofold: (1) we can exploit parallelism, and (2) more importantly, we group memory accesses by allocation sites hoping that they can share the same proof.

Table 1 shows the results of running SeaHorn on CASS. Column Nodes is the number of nodes in the graph. If the pointer analysis is context-insensitive there is only one graph, otherwise it is the sum of all graphs. The larger is the number of nodes the finer is the partitioning induced by the pointer analysis. Column Collapsed is the percentage of the nodes for which the analysis lost all the field-sensitivity. We define *density* of a node n as the number of memory accesses from pointers that point to n divided by the total number of accesses. Column Max Node Density is the maximum density value: the smaller, the better. Column Safe Alloc. Sites is the number of allocation sites proven safe by SeaHorn. An allocation site is considered proven if all its memory accesses are proven safe. Finally, Time is the accumulated time in seconds of proving all allocation sites. We set a timeout of 100 seconds per allocation site and 4GB memory limit. The total number of allocation sites is 357 and the number of memory accesses is 3,946.

[10] CASS is owned NASA and is not publicly available. It is 13,460 LOC (excluding blanks/comments).

[11] How to instrument effectively a program for proving memory safety is beyond the scope of this paper. SeaHorn provides several LLVM bitcode transformations that insert assertions such that the transformed bitcode is free of buffer overflows if all assertions hold. For our experiments, we used one that stores non-deterministically the offset and size of a pointer. This instrumentation is simple and relies on the solver to resolve the non-determinism to make sure all pointers are properly checked.

The results show that context-sensitivity is extremely important for precise memory analysis of C++ programs. In particular, the number of collapsed nodes for which field sensitivity is lost has decreased significantly. The results also show a positive impact on verification, both in number of memory accesses proven safe and on the time. Yet, there is still a significant number of memory access that remains unproved within our time limit.

6 Related Work

Rakamaric et al. [19] and Wang et al. [21] present memory models that resemble ours. These models used in Smack and Cascade, respectively, are based on a static partitioning of memory using an unification-based pointer analysis. [19] rely on DSA and propose a variant of the Burstall model [5] by combining unification with a static analysis that can infer types conservatively. [21] propose a cell-based model similar to [19] but unification and type inference are performed simultaneously producing finer-grained partitions. Both [19] and [21] identify arrays based on their allocations rather than their uses as our model does. Therefore, at the function level these models and ours are incomparable. More importantly, both [19] and [21] are context-insensitive. cbmc [7] and esbmc [10] similarly split memory into a finite set of regions using a pointer analysis. However, the model is field- and context-insensitive and it does not support symbolic array accesses.

HAVOC [6, 9] uses a byte-level memory model augmented with another map that assigns types to memory offsets. This memory model is more precise than ours but much less efficient. VCC [8] is based on a typed object memory model that is sound and complete for type-unsafe C programs. However, the model introduces *quantified* axioms that can be challenging for the verifier. Frama-C[12] provides the Jessie plugin[13] that translates C programs into verification conditions using a weakest precondition calculus. Jessie is based on a *"byte-level block"* memory model [17] which can be seen as a hybrid between the byte-level model and ours by replacing the pair (id, o) with (a, o) where a is an address. This model relies on a pointer analysis to partition memory and it is context-sensitive, but it requires to add extra axioms to ensure that the analysis of function calls is sound. Unlike VCC and Jessie models our model does not add any extra axioms. Ours is at much higher level of abstraction than these models. Abstract graph objects become logical arrays only during the generation of verification conditions and it is the underlying solver the one that will introduce select/store axioms as needed during the solving of those verification conditions.

Venet [20] proposes a model by combining Andersen's pointer analysis [3] with a numerical abstraction of offsets. This model supports dynamic mem-

[12] https://frama-c.com/

[13] Frama-C provides another plugin called VC for C programs, complementary to Jessie, with three different memory models: Hoare (unsound with pointers), Typed based on Burstall's model that does not support casts, and Byte which is a byte-level memory model.

ory allocation by mapping allocations to *timestamps*. Miné [16] presents a precise cell-based memory model (limited to programs without dynamic allocation) which represents pointers flow-sensitively as well as precise numerical relationships between offsets. This model can also reason about memory *contents*. Although more precise than ours, these two models rely on expensive numerical abstractions. Moreover, their designs fulfill a different purpose. Our goal is to produce a static memory partitioning from which we can generate verification conditions that can be solved efficiently. Instead, they produce an accurate modeling of memory from which they can prove directly properties without external solvers. More recently, Balatsouras and Smaragdakis [4] propose a new structure-sensitive pointer analysis for C/C++ programs based on LLVM. Similar to ours, this analysis can be used to perform static memory partitioning. However, the analysis is context-insensitive and its field and array-sensitivity is limited to constant pointer offsets.

7 Conclusion

The paper presents a new context-sensitive memory model for verification of C/C++. This model relies on a *field-*, *array-*, and *context-sensitive* pointer analysis tailored for generating verification conditions. The notion of *simulation relation* between points-to graphs plays a major role during the analysis of function calls. Our results suggest that our memory model can often produce a finer-grained partition of memory for programs with procedures and that this results in faster verification times.

References

1. Data Structure Analysis (DSA) implementation. Available from `https://github.com/seahorn/llvm-dsa`.
2. SeaHorn Verification Framework. Available from `http://seahorn.github.io/`.
3. L. O. Andersen. Program Analysis and Specialization for the C Programming Language. Technical report, 1994.
4. G. Balatsouras and Y. Smaragdakis. Structure-sensitive points-to analysis for C and C++. In *SAS*, pages 84–104, 2016.
5. R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. In *Machine Intelligence*, 1972.
6. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. In *TACAS*, pages 19–33, 2007.
7. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176, 2004.
8. E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A precise yet efficient memory model for C. *Electr. Notes Theor. Comput. Sci.*, 254:85–103, 2009.
9. J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *POPL*, pages 302–314, 2009.
10. L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.*, 38(4):957–974, 2012.

11. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn Verification Framework. In *CAV*, pages 343–361, 2015.
12. T. Hubert and C. Marche. Separation analysis for deductive verification. In *HAV*, 2007.
13. A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke. Automatic abstraction in SMT-based unbounded software model checking. In *CAV*, pages 846–862, 2013.
14. C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.
15. C. Lattner and V. S. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *PLDI*, pages 129–142, 2005.
16. A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *LCTES*, pages 54–63, 2006.
17. Y. Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, 2009.
18. Z. Rakamaric and M. Emmi. SMACK: decoupling source language details from verifier implementations. In *CAV*, pages 106–113, 2014.
19. Z. Rakamaric and A. J. Hu. A scalable memory model for low-level code. In *VMCAI*, pages 290–304, 2009.
20. A. Venet. A scalable nonuniform pointer analysis for embedded programs. In *SAS*, pages 149–164, 2004.
21. W. Wang, C. Barret, and T. Wies. Partitioned memory models for program analysis. In *VMCAI*, 2017.
22. W. Wang, C. Barrett, and T. Wies. Cascade 2.0. In *VMCAI*, pages 142–160, 2014.