Structure-Sensitive Points-To Analysis for C and C++

George Balatsouras and Yannis Smaragdakis

Department of Informatics, University of Athens
Athens, 15784, Greece,
{gbalats, smaragd}@di.uoa.gr

Abstract. We present a points-to analysis for C/C++ that recovers much of the available high-level structure information of types and objects, by applying two key techniques: (1) It records the type of each abstract object and, in cases when the type is not readily available, the analysis uses an *allocation-site plus type* abstraction to create multiple abstract objects per allocation site, so that each one is associated with a single type. (2) It creates separate abstract objects that represent (a) the fields of objects of either struct or class type, and (b) the (statically present) constant indices of arrays, resulting in a limited form of array-sensitivity.
We apply our approach to the full LLVM bitcode intermediate language and show that it yields much higher precision than past analyses, allowing accurate distinctions between subobjects, v-table entries, array components, and more. Especially for C++ programs, this precision is invaluable for a realistic analysis. Compared to the state-of-the-art past approach, our techniques exhibit substantially better precision along multiple metrics and realistic benchmarks (e.g., 40+% more variables with a single points-to target).

1 Introduction

Points-to analysis computes an abstract model of the memory that is used to answer the following query: *What can a pointer variable point-to, i.e., what can its value be when dereferenced during program execution?* This query serves as the cornerstone of many other static analyses aiming to enhance program understanding or assist in bug discovery (e.g., deadlock detection), by computing higher-level relations that derive from the computed points-to sets. In the literature, one can find a multitude of points-to analyses with varying degrees of precision and speed.

One of the most popular families of pointer analysis algorithms, *inclusion-based* analyses (or Andersen-style analyses [1]), originally targeted the C language, but has been extended over time and successfully applied to higher-level object-oriented languages, such as Java [3, 4, 17, 22, 24]. Surprisingly, precision-enhancing features that are common practice in the analysis of Java programs, such as field-sensitivity or online call-graph construction are absent in many analyses of C/C++ [5, 7, 8, 11, 12, 25].

In the case of field-sensitivity, the reason behind its frequent omission when analyzing C is that it is much harder to implement correctly than in Java. As noted by Pearce et al. [21], the crucial difference is that, in C/C++, it is possible to have the address of a field taken, stored to some pointer, and then dereferenced later, at an arbitrarily distant program point. In contrast, Java does not permit taking the address of a field; one can only load or store to some field directly. Hence, `load/store` instructions in Java bytecode (or any equivalent IR) need an extra field specifier, whereas in C/C++ intermediate representations (e.g., LLVM bitcode) `load/store` requires only a single address operand. The precise field affected is not explicit, but only possibly computed by the analysis itself.

The effect of such difference in the underlying IRs, as far as pointer analysis is concerned, is far from trivial. In C, the computed points-to sets have an expanded domain, since now the analysis must be able to express that a variable `p` *at some offset* `i` may point-to another variable `q` *at some offset* `j`, with these offsets corresponding to either field components or array elements.

The best-documented approach on how to incorporate field-sensitivity in a C/C++ points-to analysis is that of Pearce et al. [20, 21]. The authors extend the constraint-graph of the analysis by adding (positive) weights to edges; the weights correspond to the respective field indices. For instance, the instruction "`q = &(p->f`$_i$`)`" would be encoded as a constraint $q \supseteq p + i$. However, this approach does not take types into account. In fact, types are not even statically available at all allocation sites, since most standard C allocation routines are type-agnostic and return byte arrays that are cast to the correct type at a later point (e.g., `malloc()`, `realloc()`, `calloc()`). Thus, field i is represented with no regard to the type of its base object, even when this base object abstracts a number of concrete objects of different types. As we shall see, the lack of type information for abstract objects is a great source of imprecision, since it results in a prohibitive number of spurious points-to inferences.

We argue that type information is an essential part in increasing analysis precision, even when it is not readily available. The abstract object types should be rigorously recorded in all cases, especially when indexing fields, and used to filter the points-to sets. In this spirit, we present a *structure-sensitive* analysis for C/C++ that employs a number of techniques in this direction, aiming to retrieve high-level structure information for abstract objects in order to increase analysis precision:

1. First, the analysis records the type of an abstract object when this type is available at the allocation site. This is the case with stack allocations, global variables, and calls to C++'s `new()` heap allocation routine.
2. In cases where the type is not available (as in a call to `malloc()`), the analysis deviates from the allocation-site abstraction and creates multiple abstract objects per allocation site: one for every type that the object could have. Thus, each abstract object of type `T` now represents the set of all concrete objects of type `T` allocated at this site. To determine the possible types for a given allocation site, the analysis creates a special type-less object and records the cast instructions it flows to (i.e., the types it is cast to), using the

existing points-to analysis. This is similar to the use-based *back-propagation* technique used in past work [15, 16, 23], in a completely different context—handling Java reflection.

3. The field components of abstract objects are represented as abstract objects themselves, as long as their type can be determined. That is, an abstract object `SO` of struct type `S` will trigger the creation of abstract object $\texttt{SO.f}_\texttt{i}$, for each field $\texttt{f}_\texttt{i}$ in `S`. (The aforementioned special objects trigger no such field component creation, since they are typeless.) Thus, the recursive creation of subobjects is bounded by the type system, which does not allow the declaration of types of infinite size.
4. Finally, the analysis treats array elements similarly to field components (i.e., by representing them as distinct abstract objects, if we can determine their type), as long as their respective indices statically appear in the source code. That is, an abstract object `AO` of array type `[T×N]` will trigger the creation of abstract object `AO[c]`, if the constant `c` is used to index into type `[T×N]`. The object `AO[*]` is also created, to account for indexing at unknown (variable) indices.

As we shall see, the last point offers some form of array-sensitivity as well and is crucial for analyzing C++ code, lowered to an intermediate representation such as LLVM bitcode, in which all the object-oriented features have been translated away. To be able to resolve virtual calls, an analysis must precisely reason about the exact v-table index that a variable may point to, and the method that such an index may itself point-to. That is, a precise analysis should not merge the points-to sets of distinct indices of v-tables.

In summary, our work makes the following contributions:

- It presents a structure-sensitive pointer analysis that employs key techniques, essential in retrieving high-level structure information of heap objects, thus significantly increasing the precision of the analysis.
- The analysis is implemented and evaluated in `cclyzer`[1], a new pointer analysis framework that operates on LLVM Bitcode. The pointer analysis is expressed in a fully declarative manner, using Datalog.
- We evaluate the precision of our structure-sensitive analysis by comparing to a re-implementation of the Pearce et al. [20, 21] analysis, also operating over the full LLVM bitcode language. We show that our techniques provide a major precision enhancement for realistic programs.

2 Background and Motivation

We next discuss essential aspects of precise pointer analysis for C and C++, as well as the key features of the LLVM bitcode intermediate language.

[1] `cclyzer` is publicly available at `https://github.com/plast-lab/cclyzer`

2.1 C/C++ Intricacies and Issues

Research on pointer analysis in the last decade has shifted much of its focus from the low-level C language to higher-level object-oriented (OO) languages, such as Java [3, 4, 17, 22, 24]. To a large extent, the industry's paradigm shift to object oriented programming and Java's rising popularity naturally ignited a similar interest shift in the research community.

In points-to analysis, however, one could argue that object-oriented languages in general, and Java, in particular, are better targets than C, for a number of reasons. First, the points-to abstraction [6] is more suited to OO programming, where dynamic object allocations are more common. Furthermore, Java offers a clear distinction: only variable *references* are allocated on the stack, whereas the allocated objects themselves are stored on the heap. Also, class fields can only contain references to other objects, not entire subobjects. Thus, variables point to (heap) objects and objects can only point to each other through their fields. This leads to a clear memory abstraction as well, where objects are commonly represented by their allocation site. A points-to analysis in Java has to compute two sets of edges: (i) a set of unlabeled edges from variables to abstract heap objects, and (ii) a set of field-labeled edges between abstract objects.

This is not the case for C/C++, where:

1. Objects can be allocated both on the stack and on the heap.
2. An object can contain another *subobject* as a field component. In fact, a field may even contain a fixed-size array of subobjects.
3. Any such subobject can have its address taken and stored to some variable, which can be dereferenced later (as can any normal pointer variable) to return the subobject's exact address (i.e., the address of the base object plus the relative byte offset of the given subobject).

Figure 1 illustrates the above points. The `Outer` struct type contains a 3-element array of `Inner` subobjects via its field `in`. Unlike in Java, all these subobjects are stored inside the `Outer` instance's allocation; no dereference is needed to access them. On Figure 1b, variable `ptr` will hold the address of some subobject of variable (or stack-allocated object) `obj` of the `Outer` type. Variable `ptr` is then used later to store to this field of `obj`. (Note that the two instructions, the store instruction at line 4 and the instruction that returns the field address at line 3, can even reside in different functions.) In a precise analysis, this should establish that the `in[1].x` field of abstract object $\widehat{o_1}$ (representing the stack allocation for `obj` at line 1), may point to abstract object $\widehat{o_2}$ (representing the heap allocation of line 2).

In contrast, a *field-insensitive* approach (which is common among C/C++ analyses [5, 7, 8, 11, 12, 25]) is to not record offsets at all. This affords simplicity, at the expense of significant loss of precision. A field-insensitive analysis would disregard any offsets of any field or array accesses it encounters and simply compute that $\widehat{o_1}$ points-to (somewhere inside) $\widehat{o_2}$. Any subsequent instruction that accesses *any* field of $\widehat{o_1}$ would have to consider $\widehat{o_2}$ as a possible target. In the case of line 5, the field-insensitive analysis would (over-)conservatively infer that variable `q` may point to $\widehat{o_2}$.

```
1  typedef struct Inner {
2     int **x;
3     int *y;
4  } Inner;
5
6  typedef struct Outer {
7     void *x;
8     Inner in[3];
9  } Outer;
```

(a) Nested struct declaration

```
1  Outer obj;              // alloc: ô1
2  int *g = malloc(...); // alloc: ô2
3  int ***ptr = &(obj.in[1].x); ...
4  *ptr = &g;
5  void *q = obj.x;
```

(b) Complex Field Access

```
1  Inner i;
2  Inner *ip = &i;
3  ip = (Inner *) &ip->y;
```

(c) Positive Weight Cycles

Fig. 1: C example with nested struct types

The line of work by Pearce et al. [20,21] introduces a form of *field-sensitivity*, such that the analysis differentiates between different fields of an object by representing them with distinct symbolic offsets. For instance, the i-th field of `p` is encoded as $p + i$. Thus, the effect of an *address-of-field* instruction such as "`q = &(p->f_i)`"— f_i being the name of the i-th field of `p`—would add the edge (p, q) labeled with i to a constraint graph, to encode that $q \supseteq p+i$: the points-to set of variable q is a superset of that of the i-th field of any object pointed-to by p.

There are several issues with this approach:

1. First, it is not clear how the approach generalizes to nested structures, as in Figure 1a. Had a heap allocation $\widehat{o}$ (of unknown type) flowed to the points-to set of variable `p`, how could an expression like $p+i$ differentiate between the i-th field of $\widehat{o}$ and the i-th field of $\widehat{o}$'s first subobject? (Note that the two fields could be of entirely incompatible types.)
2. As Pearce et al. note, imprecision in the analysis may introduce positive weight cycles that lead to infinite derivations, if no other action is taken. For instance, in Figure 1c:
   i. Due to the instruction "`ip = &i;`", the points-to set of `ip` should include at least `i`: $ip \supseteq \{i\}$.
   ii. Due to instruction "`ip = (Inner *) &ip->y;`", the corresponding constraint, $ip \supseteq ip + 1$, would induce: $ip \supseteq \{i, i.y, i.y.y, i.y.y.y, \ldots\}$. Of course, an object like $i.y.y$ would make no sense given that no such field exists.

   As a way to overcome this, Pearce et al. assign unique indices to all (local) program variables and their fields, and also record their symbolic ranges (that is, the index where the enclosing lexical scope of each variable ends). Then, they ensure that field accesses only reference memory locations within the same enclosing scope. However, this does not prohibit all redundant derivations: $ip + 1$ may still add to the points-to set irrelevant variables or fields that happen to be in the same enclosing scope.

Also, this does not work well for heap allocations, since their type, and hence the number of their fields, is unknown. Instead, they are assumed to define as many fields as the largest struct in the program, which will also lead to many redundant derivations.

3. This approach greatly decreases the analysis precision in the presence of factory methods or wrapper functions for allocation routines. Consider the `xmalloc()` function of *GNU Coreutils* in Figure 2, which is consistently used instead of `malloc()` to check if the allocation succeeded and abort the program otherwise. The allocation site it contains will represent the union of all struct types, dynamically allocated via `xmalloc()`, by the same abstract object. The i-th field of this abstract object will then represent the i-th field of this union type, losing essential type information by merging unrelated fields (whose types we statically know to be completely different).

```
1  /* Allocate N bytes of memory dynamically, with error checking. */
2  void * xmalloc (size_t n) {
3    void *p = malloc (n);
4    if (!p && n != 0) xalloc_die ();
5    return p;
6  }
```

Fig. 2: Generic `malloc()` wrapper with error checking that aborts the program when allocation fails

The common denominator of all these limitations is that they lose any association between abstract objects and their types, due to cases in which type information is not readily available (as in heap allocations). What we propose instead is that the analysis strictly record types for all abstract objects (any abstract object must have a *single* type) and use this type information to filter redundant derivations that arise from analysis imprecision. For heap allocations specifically, where a single allocation site could be used to allocate objects of many different types, we propose a deviation from the standard allocation-site abstraction that creates multiple abstract objects per allocation site (one for each different type allocated there).

2.2 The LLVM IR

Our analysis targets C/C++ programs translated to LLVM bitcode. LLVM bitcode is a low-level intermediate representation, similar to an abstract assembly language, and forms the core of the LLVM umbrella project. It defines an extensive *strongly-typed* RISC instruction set, and has the following distinguishing features:

– Instead of a fixed set of registers, it uses an infinite set of temporaries, called *virtual registers*. At the register allocation phase, some of the virtual registers

will be replaced by physical registers while the rest will be spilled to memory. All virtual registers are kept in SSA form.
- Program variables are divided into two categories:
  i. variables whose address is taken and can be referenced by pointers
  ii. variables that can never be referenced by pointers.

  The latter are converted to SSA, whereas the former are kept in memory by using: (i) `alloca` instructions to allocate the required space on stack, and (ii) load/store instructions to access or update, respectively, the variable contents, at any point (hence escaping SSA form). This technique has been termed "*partial SSA*" [10].
- Like address-taken variables, global variables are also kept in memory and are always represented by a pointer to their "content" type. However, their space is allocated using a global initializer instead of an `alloca` instruction.

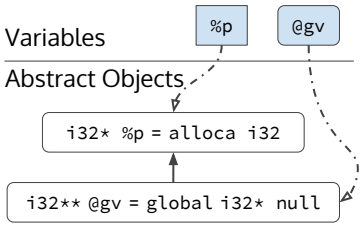
The example of Figure 3 illustrates these points regarding the LLVM translation. Figure 3a shows the original source code, while Figure 3b shows the corresponding LLVM bitcode. Local variable `p` is stored in memory (since its address is taken) and virtual register `%p` holds its address. `%p`'s value can be updated multiple times, using `store` instructions. Likewise, global variable `gv` (of type `int*`) is also kept in memory and pointer `@gv` (of type `int**`) is used to access it. As will be clear later, our analysis follows the variable representation conventions of LLVM and decouples memory allocations from virtual registers (or global variable references). Figure 3c depicts the relevant points-to relationships, which capture that `gv` points to `p`. Dashed edges are used to represent *variable points-to edges* (whose source is a virtual register), while solid edges are *dereference edges* between abstract objects.

```
int *gv;

void f()
{
  int p = 3;
  gv = &p;
}
```

(a) C source

```
i32** @gv = global i32* null

define void @f() {
  i32* %p = alloca i32
  store i32 3, i32* %p
  store %p, i32** @gv
}
```

(b) LLVM translation



(c) Points-to graph

Fig. 3: Partial SSA Example

3 Approach

Our analysis approach adds more detail to object abstractions, which serve both as sources and as targets of points-to edges, allowing a more detailed represen-

tation of the heap. Although our approach is applicable to C/C++ analysis in general, it is best to see it in conjunction with the LLVM bitcode intermediate language. Just as LLVM bitcode is a strongly-typed intermediate language, we assign types and offsets to every abstract object value and its points-to relationships. The challenge is that, unlike in the LLVM bitcode type system, such information is not readily available by local inspection of the code—it needs to be propagated by the analysis reasoning itself.

We next discuss the various abstractions of our analysis, in representing its input and output relations. Then, we express the main aspects of our analysis as a set of inference rules.

3.1 Abstractions

Figure 4 presents the input and output domains of our analysis. We represent functions as a subset of global entities. Thus, G contains all symbols referencing global entities—everything starting with symbol "`@`" in LLVM bitcode. Set V holds temporaries only (i.e., virtual registers), and not global variables. We represent the union of these two sets with P, which stands for *pointer variables* (i.e., any entity whose value may hold some memory address). Our analysis only introduces the set of abstract objects O, that correspond to memory locations.

T	set of program types
L	set of instruction labels
C	program (integer) constants
V	set of virtual registers
G	set of global variables
$F \subseteq G$	program functions
$P = V \cup G$	pointer variables
O	set of abstract objects

Fig. 4: Analysis Domains

The LLVM IR defines an extensive instruction set. However, only a small subset is relevant for the purposes of pointer analysis. Figure 5 presents a simplified version of these relevant instructions. The first two instructions are used to allocate memory on the stack and on the heap, respectively. As previously discussed, `alloca` instructions are used for address-taken variables. They accept an extra type argument (absent in `malloc` instructions), which specifies the exact type of the allocation (virtual registers are strongly typed), and the allocation size is a constant. Next, we have `cast` instructions, used solely to satisfy LLVM's type checker since they do not change any memory contents, and `phi` instructions that choose a value depending on the instruction's predecessor. Apart from the standard `load`/`store` instructions, we have two more instructions that, given a memory address operand, return a new address by adding a

relative offset that corresponds to either a field or an array element. (Only `load` instructions dereference memory, however.) Finally, we have call and return instructions. Call instructions may also accept a variable (function pointer), as their first argument.

LLVM Instruction	*Operand Types*	*Description*
`p` = `alloca T`, *nbytes*	$V \times T \times C$	Stack Allocations
`p` = `malloc` *nbytes*	$V \times (V \cup C)$	Heap Allocations
`p` = `(T) q`	$V \times T \times (P \cup C)$	(No-op) Casts
`p` = `phi`$(l_1 : a_1, l_2 : a_2)$	$V \times (L \mapsto (P \cup C))^2$	SSA Phi Node
`p` = `*q`	$V \times P$	Load from Address
`*p` = `q`	$P \times (P \cup C)$	Store to Address
`p` = `&q->`f	$V \times P \times C$	Address-of-field
`p` = `&q`$[idx]$	$V \times P \times (V \cup C)$	Address-of-array-index
`p` = $a_0(a_1, a_2, \ldots, a_n)$	$V \times (F \cup V) \times (P \cup C)^n$	Function Call
`return p`	$P \cup C$	Function Return

Fig. 5: LLVM IR Instruction Set. We also prepend a label $l \in L$ to each instruction (that we omit in this figure). Each such label can be used to uniquely identify its instruction.

Abstract Objects. Our analysis defines several different kinds of abstract objects that express the exact nature of the allocation. Any abstract object must fall into one of the following categories:

- $\widehat{o_i}$ A stack or heap allocation for instruction (allocation site) $i \in L$.
- $\widehat{o_{i,\mathtt{T}}}$ A (heap) allocation for instruction $i \in L$, specialized for type $\mathtt{T} \in T$.
- $\widehat{o_g}$ A global allocation for global variable or function $g \in G$.
- $\widehat{o.fld}$ A field subobject that corresponds to field "fld" of base object $\widehat{o} \in O$.
- $\widehat{o[c]}$ An array subobject that corresponds to the element at constant index $c \in C$ of base object $\widehat{o} \in O$.
- $\widehat{o[*]}$ An array subobject that corresponds to any elements at unknown indices of base object $\widehat{o} \in O$.

When not using any special notation, we shall refer to a generic abstract object that could be of any of the above forms.

By representing field and array subobjects as separate abstract objects themselves, the handling of instructions that return addresses anywhere but at the beginning of some allocation becomes straightforward. As we shall see at Section 3.2, all our analysis has to do is return the relevant abstract object that represents the given subobject of its base allocation. This abstract subobject will have its own distinct points-to set, which will be tracked separately from that of its base allocation or any of the rest of its fields. Thus, it will allow the analysis to retain a certain degree of precision that would be otherwise impossible.

Our analysis computes four main relations:

Variable points-to edges. Edge $\mathtt{p} \mapsto \widehat{o} \in P \times O$ records that pointer variable (either virtual register or global variable) `p` may point to abstract object $\widehat{o}$. Note that virtual registers that correspond to source variables will always point to a single object: the corresponding stack allocation. Temporaries introduced by LLVM bitcode, though, may point to many abstract objects.

Dereference edges. Edge $\widehat{po} \rightsquigarrow \widehat{o} \in O \times O$ records that abstract object $\widehat{po}$ may point to abstract object $\widehat{o}$. Any object that has a non-empty points-to set (i.e., the object has outgoing dereference edges) may represent a pointer. Dereference edges can only be established by `store` instructions.

Abstract object types. The partial function $\mathit{type} : O \nrightarrow T$ records the type of an abstract object. An abstract object can be associated with one type at most, or none at all. Since our analysis uses types to filter redundant derivations, the more types it establishes for abstract objects, the more points-to edges it will compute.

Call-graph edges. Edge $i \xrightarrow{calls} f \in L \times F$ records that invocation site i may call function f. This also accounts for indirect calls that use function pointers.

3.2 Techniques - Rules

Figure 6 presents the main aspects of the analysis as a set of inference rules. The first two rules handle stack and heap allocation instructions. All they do is create a new abstract object representing the given allocation site, and assign it to the target variable. In the case of stack allocation, we also record the type of the object, since it is available at the allocation site. The next pair of rules handle global allocations for global variables and functions, respectively, in a similar way. In contrast to the previous rules, we create abstract objects for all global entities, regardless of any instructions (since their allocation in LLVM bitcode is implicit), and record their types.

For cast instructions, we copy any object that flows in the points-to set of the source variable to the points-to set of the target variable. Phi instructions are treated similarly, but we have to consider both of the instruction's operands, regardless of their corresponding labels, since our result must be an over-approximation.

Store instructions are the only way in which the analysis establishes dereference edges. For a store instruction, $*\mathtt{p} = \mathtt{q}$, we have to perform the following:

1. First, find the corresponding abstract objects that the two instruction operands point to, by following their outgoing variable points-to edges. Namely: (i) the memory allocation of the value to be stored (abstract object $\widehat{o}$), and (ii) the memory allocation that $\widehat{o}$ is going to be stored into (abstract object $\widehat{po}$).
2. Then, establish a dereference edge between any two such abstract objects returned, expressing that object $\widehat{po}$ may point to object $\widehat{o}$.

$$\textsc{Stack}\ \frac{i:\ \mathtt{p} = \mathtt{alloca\ T},\ nbytes}{\mathtt{p} \mapsto \widehat{o_i} \qquad type(\widehat{o_i}) = \mathtt{T}} \qquad \textsc{Heap}\ \frac{i:\ \mathtt{p} = \mathtt{malloc}\ nbytes}{\mathtt{p} \mapsto \widehat{o_i}}$$

$$\textsc{Global}\ \frac{f \in F}{f \mapsto \widehat{o_f} \qquad type(\widehat{o_f}) = type(f)} \quad \frac{\mathtt{g} \in (G \setminus F)}{\mathtt{g} \mapsto \widehat{o_g} \qquad type(\widehat{o_g}) = type(\mathtt{g})}$$

$$\textsc{Cast}\ \frac{i:\ \mathtt{p} = (\mathtt{T})\,\mathtt{q} \qquad \mathtt{q} \mapsto \widehat{o}}{\mathtt{p} \mapsto \widehat{o}} \qquad \textsc{Phi}\ \frac{i:\ \mathtt{p} = \mathtt{phi}(l_1 : a_1,\ l_2 : a_2)}{\forall j:\ a_j \mapsto \widehat{o}\ \Rightarrow\ p \mapsto \widehat{o}}$$

$$\textsc{Load}\ \frac{i:\ \mathtt{p} = *\mathtt{q} \qquad \mathtt{q} \mapsto \widehat{po} \qquad \widehat{po} \rightsquigarrow \widehat{o}}{\mathtt{p} \mapsto \widehat{o}} \qquad \textsc{Store}\ \frac{i:\ *\mathtt{p} = \mathtt{q} \qquad \mathtt{p} \mapsto \widehat{po} \qquad \mathtt{q} \mapsto \widehat{o}}{\widehat{po} \rightsquigarrow \widehat{o}}$$

$$\textsc{Field}\ \frac{i:\ \mathtt{p} = \mathtt{\&q\text{->}}f \qquad \mathtt{q} \mapsto \widehat{o} \qquad type(\widehat{o}) = S \qquad type(q) = S}{\mathtt{p} \mapsto \widehat{o.f} \qquad type(\widehat{o.f}) = type(S.f)}$$

$$\textsc{Array} - \textsc{Const}\ \frac{i:\ \mathtt{p} = \mathtt{\&q}\,[c] \qquad \mathtt{q} \mapsto \widehat{o} \qquad type(\widehat{o}) = [\mathtt{T}] \qquad type(q) = [\mathtt{T}]}{\mathtt{p} \mapsto \widehat{o[c]} \qquad type(\widehat{o[c]}) = \mathtt{T}}$$

$$\textsc{Array} - \textsc{Var}\ \frac{i:\ \mathtt{p} = \mathtt{\&q}\,[\mathtt{j}] \qquad \mathtt{q} \mapsto \widehat{o} \qquad type(\widehat{o}) = [\mathtt{T}] \qquad type(q) = [\mathtt{T}]}{\mathtt{p} \mapsto \widehat{o[*]} \qquad type(\widehat{o[*]}) = \mathtt{T}}$$

$$\textsc{Call}\ \frac{i:\ \mathtt{p} = a_0(a_1,\ a_2,\ \ldots,\ a_n) \qquad a_0 \mapsto \widehat{o_f}}{i \xrightarrow{calls} f(p_1,\ p_2,\ \ldots,\ p_n) \qquad \forall j:\ a_j \mapsto \widehat{o}\ \Rightarrow\ p_j \mapsto \widehat{o}}$$

$$\textsc{Ret}\ \frac{i:\ \mathtt{p} = a_0(\ldots) \qquad i \xrightarrow{calls} f(\ldots) \qquad j:\ \mathtt{return\ q} \qquad j \in \mathrm{body}(f) \qquad \mathtt{q} \mapsto \widehat{o}}{\mathtt{p} \mapsto \widehat{o}}$$

$$\textsc{Heap-bp}\ \frac{i:\ \mathtt{p} = \mathtt{malloc}\ nbytes \qquad j:\ \mathtt{w} = (\mathtt{T})\,\mathtt{q} \qquad \mathtt{q} \mapsto \widehat{o_i}}{\mathtt{p} \mapsto \widehat{o_{i,\mathtt{T}}} \qquad type(\widehat{o_{i,\mathtt{T}}}) = \mathtt{T}}$$

Fig. 6: Inference Rules

The first step simply bypasses the indirection introduced by LLVM bitcode, where operands are represented as virtual registers that point to memory locations. Load instructions perform the opposite operation, and thus are treated symmetrically. For instruction $\mathtt{p} = *\mathtt{q}$, we first (i) find the corresponding abstract object that the address operand may point to (abstract object $\widehat{po}$), (ii) then follow any outgoing dereference edge of object $\widehat{po}$ to get any memory location $\widehat{po}$ may point to (object $\widehat{o}$), and finally (iii) establish a new variable points-to edge for target variable $\mathtt{p}$, recording that $\mathtt{p}$ may now also point to object $\widehat{o}$.

The next three rules (Field, Array–Const, Array–Var) model field-sensitivity. The rule handling field accesses, such as $\mathtt{p} = \mathtt{\&q\text{->}}f$, finds any object $\widehat{o}$ that base variable $\mathtt{q}$ may point to, and returns $\widehat{o}$'s relevant field subobject $\widehat{o.f}$. However, a key element is that $\widehat{o}$ is only considered as a base object if its type matches the declared (struct) type of $\mathtt{g}$ (recall that LLVM bitcode is strongly typed). This precludes any untyped heap allocations as possible base objects.

Otherwise, the analysis would end up creating untyped field subobjects too, further fueling imprecision. Thus, we are able to maintain an important invariant of our structure-sensitive analysis: *only create field (or array) subobjects whose types we are able to determine*. Effectively, LLVM bitcode imposes *strong typing on variables*, while our analysis extends the treatment to *abstract objects*.

Array element accesses are treated similarly and they, too, maintain this invariant. However, we distinguish array accesses using a constant index from those using a variable (i.e., unknown) index. In the former case, we return the array subobject $\widehat{o[c]}$, which represents the subobject at index c. In the latter case, we return $\widehat{o[*]}$, which represents the *unknown* index. Essentially, this treatment allows our analysis to track independently the points-to sets of array indices that are statically known to be different, yielding a form of *array-sensitivity*.

Call and return instructions as modeled as assignments: (i) from any actual argument a_j to its respective formal parameter f_j, and (ii) from any returned value `q` to the target variable of the call instruction `p`. Like cast instructions, they simply copy the points-to sets from the assignment's source to its target. However, the rule that handles call instructions also records call-graph edges. When the function operand a_0 may point to abstract object o_f, representing function f, we record an edge from the given call site to function f. This handles both direct and indirect calls (i.e., via function pointers).

How to produce type information for unknown objects. Our analysis only allows taking the address of fields of objects whose type is known. This prevents loading and storing from/to fields of objects without types. Such objects can only be used as identity markers. Yet C and C++ allow the creation of untyped objects. Their handling is a key element of the analysis.

The HEAP-BP rule implements the *use-based back-propagation* technique [15, 16, 23], which creates multiple abstract objects per (untyped) allocation site. The rule states that when an (untyped) heap object $\widehat{o_i}$ (allocated at instruction i) flows to some cast instruction j, where it is cast to type `T`, we augment the points-to set of i's target variable `p` with a new abstract object $\widehat{o_{i,\texttt{T}}}$, specialized for the given type. The insight behind this rule is that, even when the program performs an allocation via a type-agnostic routine like `malloc()`, the allocation will be later cast to its intended type before being used. By using this technique, the original untyped allocation will be prevented from creating any untyped subobjects, but as soon as the possible type of the allocation is discovered, the new abstract typed object will succeed where the untyped one has failed. Note that instructions i and j could occur in distant parts of the program, as long as the analysis can establish that the object allocated at instruction i flows to j.

This treatment successfully deals with generic allocation wrappers or factory methods. In this case, the wrapped allocation will flow to multiple cast instructions, and thus create multiple typed variations of the original object. However, in each case, only the object with the correct matching type will be used as a base for any subsequent address-of-field instructions. The rest of the objects will be filtered, since they are indeed irrelevant.

3.3 Partial Order of Abstract Objects

As the observant reader may have noticed, the rules of Figure 6 about accesses or array elements are not sound. Consider the example of Figure 7. Variable `p` points to a heap allocation. Three different store instructions take place: (i) one that stores `&i` to index 1, (ii) one that stores `&j` to index 3, and (iii) one that stores `&k` to some variable index. When loading from index 1, the analysis has to return both `&i` and `&k` (since the value of variable `idx` may be equal to 1), but not `&j`, which is stored to a different index. Conversely, when loading from a variable index, the analysis has to return all three addresses, since the index could be equal to any constant.

```
int i, j, k, idx;
...
int **p = malloc(...);
p[1] = &i;
p[3] = &j;
p[idx] = &k;
int *x = p[1];  // yields { i, k }
int *y = p[2];  // yields { k }
int *z = p[j];  // yields { i, j, k }
```

Fig. 7: Accessing array elements.

Using our array-sensitive approach, we ensure that indices 1, 3, and "$*$" (unknown) are associated with separate points-to sets that are not merged. To handle loads correctly, though, we have to be able to reason about implicit associations of abstract objects, due to possible index aliases. Thus, we say that object $\widehat{o[*]}$ "generalizes" object $\widehat{o[c]}$ (for the same base object $\widehat{o}$), since loading from $\widehat{o[*]}$ must always return a superset of the objects returned by loading from $\widehat{o[c]}$, for any constant c. This concept extends even to deeply nested subobjects. For instance, an object $o.f_1\widehat{[*][2].}f_2[*]$ generalizes object $o.f_1\widehat{[4][2].}f_2[*]$.

We can think of this binary relation between abstract objects as a partial order over domain O and define it appropriately.

Definition 1 *Abstract Object Generalization Order.* An abstract object $\widehat{y} \in O$ *generalizes* an abstract object $\widehat{x}$, denoted $\widehat{x} \sqsubseteq \widehat{y}$, if and only if:

$$\widehat{x} = \widehat{y}$$
$$\vee$$
$$(\,\widehat{x} = \widehat{p[*]} \;\vee\; \widehat{x} = \widehat{p[c]}\,) \;\wedge\; \widehat{y} = \widehat{q[*]} \;\wedge\; \widehat{p} \sqsubseteq \widehat{q}$$
$$\vee$$
$$(\widehat{x} = \widehat{p.f} \;\wedge\; \widehat{y} = \widehat{q.f} \;\wedge\; \widehat{p} \sqsubseteq \widehat{q}) \vee (\widehat{x} = \widehat{p[c]} \;\wedge\; \widehat{y} = \widehat{q[c]} \;\wedge\; \widehat{p} \sqsubseteq \widehat{q})$$

Intuitively, $\widehat{o_1} \sqsubseteq \widehat{o_2}$ holds when $\widehat{o_1}$ can be turned to $\widehat{o_2}$ by substituting any of its constant array indices with "$*$". Figure 8 gives an example of such ordering. The direction of the edges is from the less to the more general object.

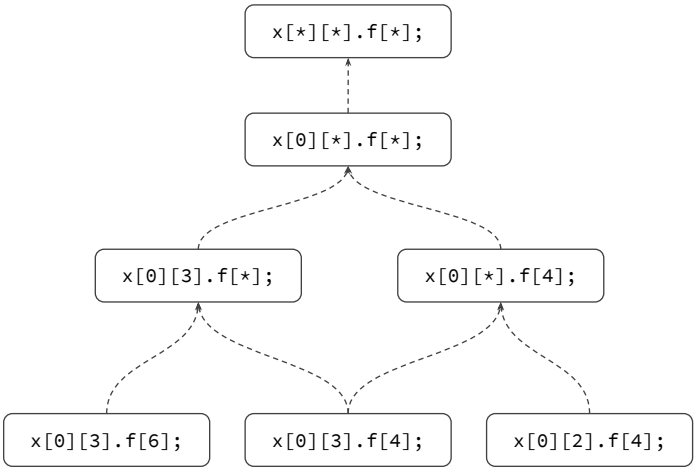


Fig. 8: Abstract Object Ordering – Example: Nodes are abstract objects. An edge $(\widehat{s}, \widehat{t})$ denotes that object $\widehat{s}$ is generalized by object $\widehat{t}$ (i.e., $\widehat{s} \sqsubseteq \widehat{t}$).

Given this partial order, it suffices to add the two rules of Figure 9 to account for possible index aliases. The first rule states that the points-to set of a (less general) object, such as $\widehat{o[c]}$, is a superset of the points-to set of any object that generalizes it, such as $\widehat{o[*]}$. The second rule modifies the treatment of load instructions, so that they may return anything in the points-to set of not just the object we load from (such as $\widehat{o}[*]$), but also of objects that it generalizes (such as $\widehat{o[c]}$). In this way, the general and specific points-to sets are kept distinct, while their subset relationship is maintained.

3.4 Soundness

As stated by Avots et al. [2]: "*A C pointer alias analysis cannot be strictly sound, or else it would conclude that most locations in memory may point to any memory location.*" As in the PCP points-to analysis [2], our approach tries to maintain precision at all times, even if this means that the analysis is not sound in some cases. Instead of trying to be as conservative as possible, we choose to opt for precision and increase soundness by selectively supporting well-established code patterns or idioms (such as using `malloc()` to allocate many objects of different types).

$$\textsc{Match}\ \frac{\widehat{o_1} \sqsubseteq \widehat{o_2} \qquad \widehat{o_2} \rightsquigarrow \widehat{o}}{\widehat{o_1} \rightsquigarrow \widehat{o}} \qquad \textsc{Load II}\ \frac{i:\ \mathtt{p} = *\mathtt{q} \qquad \mathtt{q} \mapsto \widehat{o_2} \qquad \widehat{o_1} \sqsubseteq \widehat{o_2} \qquad \widehat{o_1} \rightsquigarrow \widehat{o}}{\mathtt{p} \mapsto \widehat{o}}$$

Fig. 9: Associating array subobjects via their partial order.

The soundness assumptions of our analysis are that: (i) objects are allocated in separate memory spaces [2], and (ii) every (concrete) object has a single type throughout its lifetime. Hence, our analysis would be unsound when a union type is used to modify the same concrete object using two different types, since this violates the second assumption. However, our analysis would be a good fit for programs that use *discriminated unions* (e.g., unions that depend on a separate *tag* field to determine the exact type of the object), since it would create a different abstract object for every type of the union, so that each such abstract object would represent the subset of concrete objects with the same tag value.

In general, the single-type-per-lifetime assumption is reasonable for most objects, but would be prohibitive in some cases—especially so when the code relies on low-level assumptions about the byte layout of the objects. For instance, our base approach would not be able to meaningfully analyze code that uses a custom memory allocator. Instead, the analysis would need to be extended so that it models calls to the allocator by creating new abstract objects.

Finally, the analysis must be able to discover all associated types for any given object, to retain its soundness. For simplicity, we have only considered cast instructions as places where the analysis discovers new types, but it is easy to supply additional type hints by considering more candidates. For instance, an exception object of unknown type may be allocated and then thrown, by calling the `cxa::throw()` function in the C++ exception handling ABI, without any intervening cast. However, we can use the accompanying *typeinfo* object (always supplied as the second argument to `cxa::throw()`) to recover its true type and hence create a typed abstract exception object. To the best of our knowledge, such special treatment is needed only in rare cases, and the analysis can be easily extended to handle them.

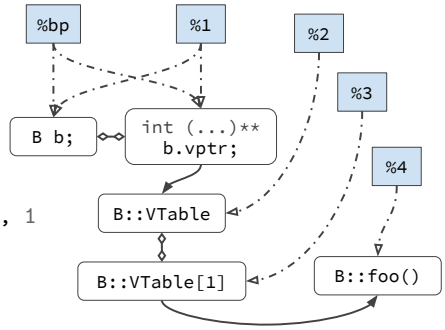
4 Analyzing C++

LLVM bitcode is a representation well-suited for C. However, for OO languages such as C++, high-level features are translated to low-level constructs. A classic example is dynamic dispatch, through virtual methods. Virtual-tables are represented as constant arrays of function pointers, and virtual calls are, in turn, translated to a series of indirect access instructions.

Figure 10a presents (a simplified version of) the LLVM bitcode for such a translation. A virtual call has to (i) load the v-pointer of the class instance (at offset 0), (ii) index into the returned v-table (at the corresponding offset of the function being called), (iii) then load the returned function pointer to get the exact address of the function, and (iv) finally call the function. By employing

```
%class.B = type { i32 (...)**, ...}

;; translation of bp->foo(), for
;; B *bp;
%1=bitcast %bp to i32 (%class.B*)***
%2=load i32 (%class.B*)** %1
%3=getelementptr i32 (%class.B*)** %2, 1
%4=load i32 (%class.B*)* %3
call i32 %4 (%class.B* %bp)
```

(a) C++ virtual call compiled to LLVM bitcode



(b) Points-to graph

Fig. 10: C++ Virtual Call Example

the techniques we have described so far, our structure-sensitive analysis is well-equipped to deal with such an involved pattern, and precisely resolve the function to be called.

Figure 10b shows what our analysis computes (assuming `%bp` points to variable `b`). Only a minor addition is required: anything that points to an object should also point to its first field (at byte offset 0). Hence, both `%bp` and `%1` (after the cast) will point both to (stack-allocated) object $\widehat{b}$, and to its v-pointer field subobject $\widehat{b.vptr}$. The first load instruction will return the v-table. Indexing into the v-table will return the corresponding array element subobject, which will maintain its own independent (singleton) points-to set, due to array-sensitivity. Finally, the second load instruction will return the exact function that the v-table points to, at the given offset.

5 Evaluation

We compare our structure-sensitive analysis to a re-implementation of the Pearce et al. [20,21] analysis in `cclyzer`, that also operates over the full LLVM bitcode language. We will refer to this analysis as *Pearce*^{c}. Both analyses were implemented using Datalog, and include many enhancements to deal with various features (such as `memcpy` instructions, hidden copies of struct instances due to pass-by-value semantics, type compatibility rules, etc.) that arise in practice.

For our benchmark suite, we use the 8 largest programs (in terms of bitcode size) in *GNU Coreutils*,[2] and 14 executables from *PostgreSQL*. We use a 64-bit machine with two octa-core Intel Xeon E5-2667 (v2) CPUs at 3.30GHz and 256GB of RAM. The analysis is single-threaded and occupies a small portion of the RAM. We use the LogicBlox Datalog engine (v.3.10.14) and LLVM v.3.7.0.

[2] Our original selection included the 10 largest coreutils, but `dir` and `vdir` turned out to be identical to `ls` and are maintained mostly for backwards-compatibility reasons.

			Structure-sensitive		*Pearce*^{c}	
Benchmark	Size	call-graph edges	abstract objects	running time	abstract objects	running time
cp	720K	3205	68166	29.25s	3380	13.62s
df	456K	1812	38919	20.68s	2236	11.09s
du	608K	2424	49592	29.77s	3008	21.96s
ginstall	692K	3185	59893	25.12s	3207	14.32s
ls	604K	2654	66469	22.43s	2783	13.35s
mkdir	384K	1466	21900	17.35s	1641	11.43s
mv	648K	2932	55619	25.50s	3015	12.20s
sort	608K	2480	75360	34.25s	2955	21.40s
clusterdb	528K	1390	167605	33.90s	4461	11.89s
createdb	528K	1412	168068	30.58s	4480	11.07s
createlang	572K	1928	133869	25.67s	4275	12.68s
createuser	532K	1435	171115	31.07s	4569	9.31s
dropdb	524K	1361	165966	31.26s	4399	12.72s
droplang	572K	1936	133912	24.38s	4278	12.55s
dropuser	524K	1356	165615	30.45s	4386	12.15s
ecpg	1.2M	5713	59252	38.47s	5219	29.11s
pg-ctl	488K	1615	118689	23.36s	3655	9.14s
pg-dumpall	572K	2110	184276	32.18s	5153	11.95s
pg-isready	464K	1302	108622	21.54s	3343	11.25s
pg-rewind	556K	1943	136915	25.56s	4301	11.48s
pg-upgrade	604K	2501	151967	26.49s	4965	11.80s
psql	1.4M	5925	460522	67.76s	14025	25.28s

Fig. 11: Input and Output Metrics. The first column is benchmark bitcode size (in bytes). The second column is the number of call-graph edges (as computed by our analysis). The third (resp. fifth) column is the number of abstract objects created. The fourth (resp. sixth) column is the analysis running time.

Figure 11 presents some general metrics on the input and output of each analysis: (i) number of call-graph edges (allocation site to function), (ii) number of abstract objects created by the analysis, and (iii) running time (excluding constant overhead that bootstrap both analyses).

Figure 12 compares the two analyses in terms of the degree of resolving variable points-to targets. The first column of each analysis lists the percentage of fully resolved variables (virtual registers): *how many point to a single abstract object*. This is the main metric of interest for most analysis clients. The next two columns list the percentage of variables that point to two/three objects.

It is evident that our structure-sensitive analysis fares consistently better in fully resolving variable targets. Our analysis resolves many more variables than *Pearce*^{c} does, for any of the available benchmarks, with an average increase of 36% across all coreutil benchmarks and 58% in the PostgreSQL benchmarks. This is *despite using a finer-grained object abstraction than* Pearce^c: The "abstract objects" column of Figure 11 shows that our analysis abstraction has *one*

	Structure-sensitive			*Pearce*^{c}		
Benchmark	(%) $\|pt(v)\| \to 1$	2	3	(%) $\|pt(v)\| \to 1$	2	3
cp	35.42	11.56	9.03	24.02	2.91	3.51
df	35.98	13.15	8.37	26.28	1.98	4.38
du	37.06	10.51	7.54	25.60	2.00	2.95
ginstall	36.31	14.24	8.28	27.15	7.44	3.14
ls	33.23	6.09	8.81	26.90	3.57	2.67
mkdir	36.11	8.43	9.65	23.02	2.00	4.35
mv	35.09	13.71	8.97	24.58	6.78	3.04
sort	29.20	5.25	9.65	22.37	1.47	2.53
average	34.49	9.51	8.79	25.37	3.53	3.19
clusterdb	40.86	8.42	7.93	24.46	2.79	3.85
createdb	40.82	9.11	7.95	24.54	2.83	4.31
createlang	42.72	8.87	11.89	25.62	4.10	4.78
createuser	40.33	8.85	8.75	24.07	3.18	4.44
dropdb	40.59	8.69	7.96	23.97	2.91	4.00
droplang	42.68	8.86	11.88	25.67	4.10	4.75
dropuser	40.36	8.72	8.01	23.86	2.86	4.02
ecpg	16.72	1.22	0.52	15.14	0.30	42.64
pg-ctl	41.31	8.46	8.50	25.31	3.31	4.05
pg-dumpall	40.52	7.10	7.21	27.74	3.10	4.61
pg-isready	39.89	8.12	7.87	23.59	2.92	4.03
pg-rewind	44.74	7.55	8.56	31.39	2.75	3.76
pg-upgrade	41.12	8.35	9.34	27.73	2.95	3.70
psql	38.62	5.81	9.33	25.61	2.31	3.20
average	39.38	7.72	4.55	24.91	2.89	6.87

Fig. 12: Variable points-to sets. Proportion of resolved variables (that point to one abstract object), as well as variables with two or three points-to targets.

to two orders of magnitude more abstract objects than *Pearce*^{c}. Yet it succeeds at resolving many more variables to a single (and much finer-grained) abstract object. (The only benchmark instance in which *Pearce*^{c} somewhat benefits from its coarse abstract object granularity is ecpg: a full 42.64% of variables point to 3, much coarser than ours, abstract objects.) Note also that the *Pearce*^{c} analysis appears much better than it actually is for meaningful cases, due to large amounts of low-hanging fruit—e.g., global or address-taken variables, which are the single target of some virtual register, due to the SSA representation.

6 Related Work

We discussed some closely related work throughout the paper. Most C and C++ analyses in the past have focused on scalability, at the expense of precision. Several (e.g., [7,14,25]) do not model more than a small fraction of the functionality of modern intermediate languages.

One important addition is the DSA work of Lattner et al. [13], which was the original points-to analysis in LLVM. The analysis is no longer maintained, so comparing experimentally is not possible. In terms of a qualitative comparison, the DSA analysis is a sophisticated but ad hoc mix of techniques, some of which add precision, while others sacrifice it for scalability. For instance, the analysis is field-sensitive using byte offsets, at both the source and the target of points-to edges. However, when a single abstract object is found to be used with two different types, the analysis reverts to collapsing all its fields. (Our analysis would instead create two abstract objects for the two different types.) Furthermore, the DSA analysis is unification-based (a Steensgaard analysis), keeping coarser abstract object sets and points-to sets than our inclusion-based analysis. Finally, the DSA analysis uses deep context-sensitivity, yet discards it inside a strongly connected component of methods.

The field-sensitive inclusion-based analysis of Avots et al. [2] uses type information to improve its precision. As in this work, they explicitly track the types of objects and their fields, and filter out field accesses whose base object has an incompatible type (which may arise due to analysis imprecision). However, their approach is array-insensitive and does not employ any kind of type back-propagation to create more (fine-grained) abstract objects for polymorphic allocation sites. Instead, they consider objects used with multiple types as possible type violations. Finally, they extend type compatibility with a form of structural equivalence to mark types with identical physical layouts as compatible. The implementation of `cclyzer` applies a more general form of type compatibility, which has been omitted from this paper for space reasons.

Miné [18] presents a highly precise analysis, expressed in the abstract interpretation framework, that translates any field and array accesses to pointer arithmetic. By relying on an external numerical interval analysis, this technique is able to handle arbitrary integer computations, and, thus, any kind of pointer arithmetic. However, the precision comes with scalability and applicability limitations: the technique can only analyze programs without dynamic memory allocation or recursion.

There are similarly other C/C++-based analyses that claim field-sensitivity [9, 10], but it is unclear at what granularity this is implemented. Existing descriptions in the literature do not match the precision of our structure-sensitive approach, which maintains maximal structure information (with typed abstract objects and full distinction of subobjects), at both sources and targets of points-to relationships. Nystrom et al. [19] have a fine-grained heap abstraction that corresponds to standard use of "heap cloning" (a.k.a. "context-sensitive heap").

7 Conclusions

We presented a structure-sensitive points-to analysis for C and C++. The analysis attempts to always distinguish abstract objects and assign them a unique type (even when none is known at the point of object creation) as well as to discriminate between subobjects of a single object (array or structure instance).

We describe the analysis in precise terms and show that its approach succeeds in maintaining precision when analyzing realistic programs. In our experience, the techniques we described are essential for analyzing C/C++ programs at the same level of precision as programs in higher-level languages.

Acknowledgments

We gratefully acknowledge funding by the European Research Council under grant 307334 (Spade). We thank Kostas Ferles and Eirini Psallida for their early contributions to `cclyzer`; and also the anonymous reviewers of this paper, for their insightful comments and suggestions.

References

1. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. Ph.D. thesis, DIKU, University of Copenhagen (May 1994)
2. Avots, D., Dalton, M., Livshits, B., Lam, M.S.: Improving software security with a C pointer analysis. In: Proc. of the 27th International Conf. on Software Engineering. pp. 332–341. ICSE '05, ACM, New York, NY, USA (2005)
3. Berndl, M., Lhoták, O., Qian, F., Hendren, L.J., Umanee, N.: Points-to analysis using BDDs. In: Proc. of the 2003 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 103–114. PLDI '03, ACM, New York, NY, USA (2003)
4. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications. OOPSLA '09, ACM, New York, NY, USA (2009)
5. Das, M.: Unification-based pointer analysis with directional assignments. In: Proc. of the 2000 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 35–46. PLDI '00, ACM, New York, NY, USA (2000)
6. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Proc. of the 1994 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 242–256. PLDI '94, ACM, New York, NY, USA (1994)
7. Hardekopf, B., Lin, C.: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In: Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 290–299. PLDI '07, ACM, New York, NY, USA (2007)
8. Hardekopf, B., Lin, C.: Exploiting pointer and location equivalence to optimize pointer analysis. In: Proc. of the 14th International Symp. on Static Analysis. pp. 265–280. SAS '07, Springer (2007)
9. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: Proc. of the 36th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. pp. 226–238. POPL '09, ACM, New York, NY, USA (2009)
10. Hardekopf, B., Lin, C.: Flow-sensitive pointer analysis for millions of lines of code. In: Proc. of the 9th International Symp. on Code Generation and Optimization. pp. 289–298. CGO '11, IEEE Computer Society (2011)

11. Heintze, N., Tardieu, O.: Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In: Proc. of the 2001 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 254–263. PLDI '01, ACM, New York, NY, USA (2001)
12. Hind, M., Burke, M.G., Carini, P.R., Choi, J.: Interprocedural pointer alias analysis. ACM Trans. Program. Lang. Syst. 21(4), 848–894 (1999)
13. Lattner, C., Lenharth, A., Adve, V.S.: Making context-sensitive points-to analysis with heap cloning practical for the real world. In: Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 278–289. PLDI '07, ACM, New York, NY, USA (2007)
14. Lhoták, O., Chung, K.C.A.: Points-to analysis with efficient strong updates. In: Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. pp. 3–16. POPL '11, ACM, New York, NY, USA (2011)
15. Li, Y., Tan, T., Sui, Y., Xue, J.: Self-inferencing reflection resolution for Java. In: Proc. of the 28th European Conf. on Object-Oriented Programming. pp. 27–53. ECOOP '14, Springer (2014)
16. Livshits, B., Whaley, J., Lam, M.S.: Reflection analysis for Java. In: Proc. of the 3rd Asian Symp. on Programming Languages and Systems. pp. 139–160. APLAS '05, Springer (2005)
17. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to and side-effect analyses for Java. In: Proc. of the 2002 International Symp. on Software Testing and Analysis. pp. 1–11. ISSTA '02, ACM, New York, NY, USA (2002)
18. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: Proc. of the 2006 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems. pp. 54–63. LCTES '06, ACM (2006)
19. Nystrom, E.M., Kim, H., Hwu, W.W.: Importance of heap specialization in pointer analysis. In: Proc. of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. pp. 43–48. PASTE '04, ACM, New York, NY, USA (2004)
20. Pearce, D.J., Kelly, P.H.J., Hankin, C.: Efficient field-sensitive pointer analysis for C. In: Proc. of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. pp. 37–42. PASTE '04, ACM, New York, NY, USA (2004)
21. Pearce, D.J., Kelly, P.H.J., Hankin, C.: Efficient field-sensitive pointer analysis of C. ACM Trans. Program. Lang. Syst. 30(1) (2007)
22. Rountev, A., Milanova, A., Ryder, B.G.: Points-to analysis for Java using annotated constraints. In: Proc. of the 16th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications. pp. 43–55. OOPSLA '01, ACM, New York, NY, USA (2001)
23. Smaragdakis, Y., Balatsouras, G., Kastrinis, G., Bravenboer, M.: More sound static handling of Java reflection. In: Proc. of the 13th Asian Symp. on Programming Languages and Systems. pp. 485–503. APLAS '15, Springer (2015)
24. Whaley, J., Rinard, M.C.: Compositional pointer and escape analysis for Java programs. In: Proc. of the 14th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications. pp. 187–206. OOPSLA '99, ACM, New York, NY, USA (1999)
25. Zheng, X., Rugina, R.: Demand-driven alias analysis for C. In: Proc. of the 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. pp. 197–208. POPL '08, ACM, New York, NY, USA (2008)