Unification-based Pointer Analysis without Oversharing

Jakub Kuderski*, Jorge A. Navas†, and Arie Gurfinkel*

*University of Waterloo, Canada
{jakub.kuderski, arie.gurfinkel}@uwaterloo.ca

†SRI International, USA
jorge.navas@sri.com

***Abstract*—Pointer analysis is indispensable for effectively verifying heap-manipulating programs. Even though it has been studied extensively, there are no publicly available pointer analyses that are moderately precise while scalable to large real-world programs. In this paper, we show that existing context-sensitive unification-based pointer analyses suffer from the problem of *oversharing* – propagating too many abstract objects across the analysis of different procedures, which prevents them from scaling to large programs. We present a new pointer analysis for LLVM, called TEADSA, without such an oversharing. We show how to further improve precision and speed of TEADSA with extra contextual information, such as flow-sensitivity at call- and return-sites, and type information about memory accesses. We evaluate TEADSA on the verification problem of detecting unsafe memory accesses and compare it against two state-of-the-art pointer analyses: SVF and SEADSA. We show that TEADSA is one order of magnitude faster than either SVF or SEADSA, strictly more precise than SEADSA, and, surprisingly, sometimes more precise than SVF.**

I. INTRODUCTION

Pointer analysis (PTA) – determining whether a given pointer aliases with another pointer (*alias analysis*) or points to an allocation site (*points-to analysis*) are indispensable for reasoning about low-level code in languages such as C, C++, and LLVM bitcode. In compiler optimization, PTA is used to detect when memory operations can be lowered to scalar operations and when code transformations such as code motion are sound. In verification and bug-finding, PTA is often used as a pre-analysis to limit the implicit dependencies between values stored in memory. This is typically followed by a deeper, more expensive, path-sensitive analysis (e.g., [1], [2], [3]). In both applications, the efficiency of PTA is crucial since it directly impacts compilation and verification times, while precision of the analysis determines its usability. Moderately precise and efficient PTA is most useful, compared to precise but inefficient or efficient but imprecise variants.

The problem of pointer analysis is well studied. A survey by Hind [4] (from 2001!) provides a good overview of techniques and precision vs cost trade-offs. Despite that, very few practical implementations of PTA targeting low-level languages are available. In part, this is explained by the difficulty of soundly supporting languages that do not provide memory safety guarantees, allow pointers to fields of aggregates, and allow arbitrary pointer arithmetic. In this paper, we focus on the PTA problem for low-level languages.

There are many dimensions that affect precision vs cost trade-offs of a PTA, including path-, flow-, and (calling) context-sensitivity, modeling of aggregates, and modularity of the analysis. From the efficiency perspective, the most significant dimension is whether the analysis is *inclusion-based* (a.k.a., *Andersen-style* [5]) or *unification-based* (a.k.a., *Steensgaard-style* [6]). All other things being equal, a unification-based analysis is significantly faster than an inclusion-based one at the expense of producing very imprecise results. To improve further precision while retaining its efficiency, a unification-based PTA can be extended with (calling) context-sensitivity in order to separate local aliasing created at different call sites. Unfortunately, the combination of a unification-based analysis with context-sensitivity can quickly degenerate in a prohibitive analysis.

State-of-the-art implementations of unification-based, context-sensitive PTA (e.g., DSA [7] and SEADSA [8]) perform the analysis in phases. First, each function is analyzed in an intra-procedural manner (LOCAL). Second, a BOTTOM-UP phase inlines callees' points-to graphs into their callers. Third, a TOP-DOWN phase inlines callers' points-to graphs into their callees. We observed that both BOTTOM-UP and TOP-DOWN often copy too many *foreign objects*, memory objects allocated by other functions that cannot be accessed by the function at hand, increasing dramatically both analysis time and memory usage. In fact, we show in Sec. VI that the majority of analysis runtime is spent on copying foreign objects. Even worse, due to the imprecise nature of unification-based PTA and difficulty of analyzing accurately aggregates, foreign objects can be aliased with other function objects affecting negatively the precision of the analysis. We refer to *oversharing* as the existence of large number of inaccessible foreign objects during the analysis of a particular function.

In this paper, we present a new pointer analysis for LLVM, called TEADSA, that eliminates a class of such an oversharing. TEADSA is a new unification-based PTA implemented on top of SEADSA. Since TEADSA builds on SEADSA, it remains modular (i.e., analysis of each function is summarized and the summary is used at call sites), context-, field-, and array-

sensitive. The first main difference is that TEADSA does not add oversharing during TOP-DOWN while retaining full context-sensitivity. This is achieved by not copying foreign objects coming from callers. This is a major improvement compared to previous implementations. DSA mitigates the oversharing problem by partially losing context-sensitivity. SEADSA does not tackle this problem since it focuses on medium-size programs such as the SV-COMP benchmarks [9].

Second, we observed that oversharing can also come from the LOCAL phase. This is mainly because the local analysis is flow-insensitive. To mitigate this, we make TEADSA flow-sensitive but only at call- and return sites. This preserves the efficiency of the analysis while improving its precision.

Third, we noted that another source of imprecision in SEADSA is loss of field-sensitivity during analysis of operations in which determining the exact field being accessed is difficult and merging that is inherent to its unification nature. Crucially, in many cases where field-sensitivity is lost, it is still clear that pointers do not alias if their types are taken into account. Under *strict aliasing* rules of the C11 standard, two pointers cannot alias if they do not have compatible types [10]. By following strict aliasing, we further improve the precision of TEADSA.

We have evaluated TEADSA against SEADSA and SVF, a state-of-the-art inclusion-based pointer analysis in LLVM, on the verification problem of detecting unsafe memory accesses. Our evaluation shows that TEADSA is one order of magnitude faster than SEADSA or SVF, strictly more precise than SEADSA, and sometimes more precise than SVF.

II. OVERVIEW

In this section, we illustrate our approach on a series of simple examples. Consider a C program P_1 in Fig. 1(a) and its corresponding context-insensitive and flow-insensitive points-to graph G_1 in Fig. 1(b). The nodes of G_1 correspond to registers (ellipses) and *groups* of abstract memory objects (rectangles), and edges of G_1 represent the points-to relation between them. As usual, a *register* is a program variable whose address is not taken. For example, the local variable `s` is a register. Similarily, an *abstract object* represents concrete memory objects allocated at a static allocation site, such as an address-taken global or local variable, or a call to an allocating function like `malloc`. For field sensitivity, `struct` fields are associated with their own abstract objects. In Fig. 1(a), we denote corresponding abstract objects in comments. For example, the local integer variable `i` is associated with an abstract object o_5, while the `struct` variable `c` is associated with abstract objects $u.f_0$ and $u.f_8$ for its `label` and `val` fields at offset 0 and 8, respectively.

The edges of G_1 denote whether a pointer p may point to an abstract object o, written $p \mapsto o$. Whenever p may point to multiple abstract objects all of these objects are grouped into a single (rectangular) node. For instance, $\texttt{x} \mapsto o_1$, $\texttt{x} \mapsto o_2$, $\texttt{x} \mapsto o_3$, $\texttt{x} \mapsto o_4$, or $\texttt{x} \mapsto \{o_1, o_2, o_3, o_4\}$ for brevity. We say that two pointers p_1 and p_2 *alias* when they may point to the same abstract object, written $alias(p_1, p_2)$.

The graph G_1 in Fig. 1(b) corresponds to the Steensgaard (or unification-based) PTA [6]. This style of PTA ensures an invariant (**I1**): whenever there is a pointer p_1 and objects o_a and o_b such that $p_1 \mapsto o_a$ and $p_1 \mapsto o_b$, then for any other pointer p_2 if $p_2 \mapsto o_a$ then $p_2 \mapsto o_b$. On one hand, (**I1**) implies that Steensgaard PTA can be done in linear time using a union-find data structure to group objects together. On the other, Steensgaard PTA is quite imprecise. In our running example, it deduces that almost all registers of P_1 may alias, which is clearly not the case. For instance, $\texttt{s} \mapsto \{o_1, o_2, o_3, o_4\}$ in Fig. 1(b), even though there is no execution in which $\texttt{s} \mapsto o_1$ or $\texttt{s} \mapsto o_4$.

A standard way to make the Steensgaard PTA more precise is to perform the analysis separately for each procedure. This is referred to as (calling) context-sensitivity. The main idea is to distinguish local aliasing created at different call sites. Data Structure Analysis (DSA) [7] is an example of a context-sensitive Steensgaard PTA. The results of a context-sensitive Steensgaard PTA on P_1 are shown in Fig. 2(a) as four separate points-to graphs – one for each procedure in P_1. An increase in precision (compared to the PTA in Fig. 1(b)) is visible in procedures `foo`, `bar`, and `getStr`: the string `str4` does not alias all the other strings. The improvement comes at a cost – some abstract objects appear in the analysis results of multiple procedures. For instance, o_1, o_2, and o_3 appear in all 4 graphs. In the worst case, DSA can grow quadratically in the program size, which prevents it from scaling to large programs.

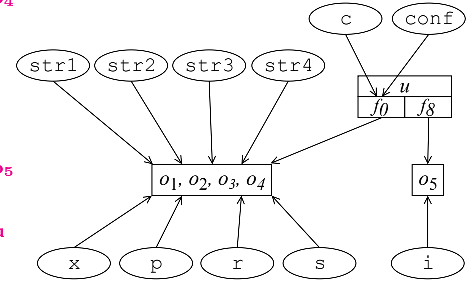
In Sec. VI, we show that in DSA the majority of runtime is often spent on copying *foreign abstract objects* coming from other procedures. For example, consider the abstract objects $u.f_8$ and o_5: the procedure `foo` never accesses the `val` field of `conf`. As shown in Fig. 1(a), $u.f_8$ and o_5 are only accessible in `foo` through `conf` and thus should not appear in the analysis for `foo` or any of its callees. However, both $u.f_8$ and o_5 are present in the points-to graph for `foo` in Fig. 2(a), as computed by a DSA-like PTA. This performance issue was already observed in [7], but only a workaround that loses context-sensitivity for global objects was implemented.

In this paper, we show that points-to analysis should refer only to abstract objects actually used by a procedure. This includes abstract objects in a procedure and its callees, abstract objects derived from function arguments, and used global variables. Thus, foreign abstract objects coming from callers are not only unnecessary in the final analysis results of their callees, but needless in the first place. Compared to Fig. 2(a), in our proposed analysis, Fig. 2(b), function argument accesses are given separate abstract objects, instead of referring to (foreign) abstract objects of callees.

Furthermore, we observed that DSA maintains the following invariants (**I2**): if a procedure F_1 with $p_1 \mapsto o$ calls a procedure F_2, and there is an interprocedural assignment to a function argument p_2 of F_2, $p_2 := p_1$, then $p_2 \mapsto o$; (**I3**): if F_1 calls F_2 and $p_2 \mapsto o$ in F_2, and there is an interprocedural assignment to a pointer p_1 in F_1 by returning p_2 from F_2, $p_1 := p_2$, then $p_1 \mapsto o$. For example, `foo` calls `getStr` in Fig. 1(a), $\texttt{str1} \mapsto \{o_1, o_2\}$ in `getStr`, thus

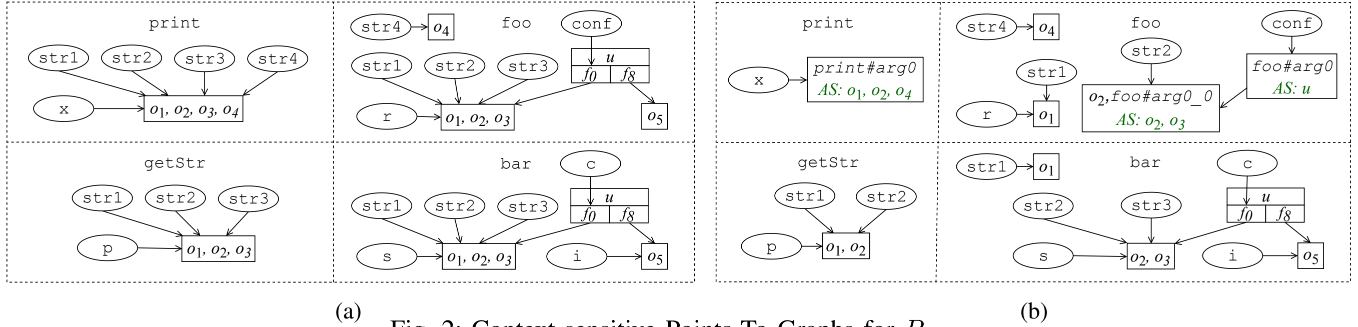
```c
const char *str1 = "Str1";   // o1
const char *str2 = "Str2";   // o2
const char *str3 = "Str3";   // o3

void print(const char *x) {}

const char *getStr() {
  const char *p = nondet() ?
                  str1 : str2;
  print(p);
  return str1;
}

struct Config
{ const char *label; int *val; };
int foo(struct Config *conf) {
  const char str4[5] = "Str4";// o4
  print(str4);
  const char *r = getStr();
  print(r);
  return conf->label == r;
}

int bar() {
  int i = 42;                  // o5
  const char *s = nondet() ?
                  str2 : str3;
  struct Config c = {s, &i}    // u
  return foo(&c);
}
```

(a)



(b)

Fig. 1: Sample C program P_1 (a) and its Context-insensitive Points-To Graph G_1 (b).



(a) (b)

Fig. 2: Context-sensitive Points-To Graphs for P_1.

the returned value $\mathtt{r} \mapsto \{o_1, o_2\}$. (**I2**) and (**I3**) are useful to argue that adding context-sensitivity to Steensgaard preserves soundness. However, they cause unnecessary propagations of foreign abstract objects. For instance, even though according to (**I1**) it must be that locally $\mathtt{str1} \mapsto \{o_1, o_2\}$ in `getStr`, `getStr` can only return a pointer to o_1, as `str1` is used in the return statement, so $\mathtt{r} \mapsto o_1$ and $\mathtt{r} \not\mapsto o_2$ – that violates (**I3**).

In addition to not introducing foreign abstract object for arguments, many propagations caused by a local imprecision are avoided by not maintaining (**I2**) and (**I3**). Breaking (**I2**) allows the analysis to propagate fewer foreign abstract objects from callers to callees (i.e., top-down), while breaking (**I3**) at return sites to reduces the number of maintained foreign abstract objects coming from callees (i.e., bottom-up).

In this paper, we show that a context-sensitive unification-based PTA that does not maintain (**I2**) and (**I3**) can be refined with extra contextual information to reduce the number of foreign abstract objects, as long as the information is valid for a given source location in the current calling context.

The *strict aliasing* rules of the C11 standard specify that at any execution point every memory location has a type, called *effective type*. A read from a memory location can only access a type compatible with its effective type. Consider the program P_2 in Fig. 3: dereferencing the integer pointer `ip` is only allowed when the last type written was `int`. We use *strict aliasing* to improve precision of our PTA.

In order to use types as an additional context, we add an extra abstract object for any type used with the corresponding allocation site or its field. As a result, every abstract object has an associated type tag. Following *strict aliasing*, two objects o_1 and o_2 can alias, only when their type tags are compatible.

```c
const int INT_TAG = 0, FLOAT_TAG = 1;
typedef struct { int tag; } Element;
typedef struct
{ Element e; int *d; } IElement;
typedef struct
{ Element e; float *d; } FElement;

void print_int(int);
void baz() {
  int a = 1;                          // o6
  float f;                            // o7
  IElement e1 = {{INT_TAG}, &a};      // v
  FElement e2 = {{FLOAT_TAG}, &f};    // w
  Element *elems[2] = {&e1, &e2};     // x

  for (int i = 0; i < 2; ++i)
    if (elems[i]->tag == INT_TAG) {
      IElement *ie = elems[i];
      int *ip = (int *) ie->d;
      print_int(*ip);
    }
}
```

Fig. 3: Sample C program P_2.

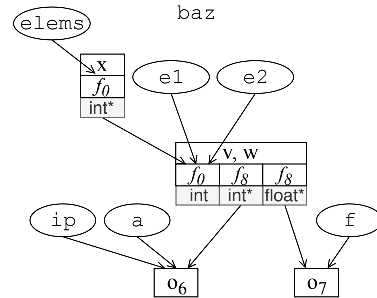


Fig. 4: Type-aware Points-To Graph of P_2.

In the P_2's points-to graph in Fig. 4, type tags are shown at the bottom of each abstract object. We maintain soundness by discovering type tags based only on memory accesses

P	::=	$F+$
F	::=	`fun name(`$\overline{\texttt{f}}$`):` $\overline{\texttt{r}}$ $\{I+\}$
I	::=	`r = alloc()` | `r = cast` T`, p` |
		`r = load` PT `p` | `store r,` PT `p` |
		`r = gep` PT `p,` fld | $\overline{\texttt{r}}$ `= callee(`$\overline{\texttt{p}}$`)` | `return` $\overline{\texttt{z}}$
T	::=	$BT \cup PT$
BT	::=	`int` | `float` | `char`
PT	::=	$BT\star$ | $BT\star\star$
fld	::=	`a` | `b`

Fig. 5: A simple language.

$$\frac{i : \texttt{r = alloc()}}{\texttt{r} \mapsto H_i}\ \textsc{Alloc} \qquad \frac{\texttt{r = cast PT, p} \quad \texttt{p} \mapsto H}{\texttt{r} \mapsto H}\ \textsc{Cast}$$

$$\frac{\texttt{r = load PT p} \quad \texttt{p} \mapsto H \quad H \mapsto I}{\texttt{r} \mapsto I}\ \textsc{Load} \qquad \frac{\texttt{store r, PT p} \quad \texttt{p} \mapsto I \quad \texttt{r} \mapsto H}{I \mapsto H}\ \textsc{Store}$$

Fig. 6: Inference rules for Inclusion-based PTA: $\mathbb{\Gamma}_{\mathrm{I}}$.

$$\frac{\texttt{r = gep PT p, a} \quad \texttt{p} \mapsto H}{\texttt{r} \mapsto H}\ \textsc{GEP} \qquad \frac{\texttt{r = gep PT p, b} \quad \texttt{p} \mapsto H \quad fld(H) = \texttt{a} \quad siblingObj(H) = I}{\texttt{r} \mapsto I}\ \textsc{GEP}$$

Fig. 7: Inference rules for Field-Sensitivity: $\mathbb{\Gamma}_{\mathrm{FLD}}$.

performed, instead of relying on casts or type declarations. Alternatively, it is also possible to use externally supplied type tags (e.g., emitted from a C compiler's frontend).

Although types increase the number of abstract objects, they improve the precision of our analysis. For example, consider the structs `e1` and `e2` defined in lines 12 and 13 of P_2. The `d` field of `e1` is assigned a pointer to `a`, while the `d` field of `e2` is assigned a pointer to `f`. Because of these memory writes, we know that `e1.d` is of type `int*` and `e2.d` is `float*`. Even though v and w are grouped according to (**I1**) as `elems` $\mapsto \{v.f_0.int, w.f_0.int\}$, o_6 and o_7 do not alias, as the abstract objects for `e1.d` and `e2.d` differ in type tags: $v.f_8.int*$ vs $w.f_8.float*$.

In summary, our enhancements to the standard context-sensitive unification-based PTA not only dramatically improve the performance, but also the precision of the analysis. This is due to the interaction between improved local reasoning at call- and return-sites, and the reduction on propagating foreign abstract objects across functions. We also show that while the added type-awareness increases the number of abstract objects, the analysis scales better than a type-unaware one (on our benchmarks). Interestingly, our proposed PTA is much faster and usually as precise as the SVF PTA [11], and sometimes even significantly more precise. Note that SVF is a state-of-the art, inclusion-based PTA that chooses not to maintain (**I1**) for more precision, but is not context-sensitive in order to scale.

III. Background

In this section, we present the necessary background to understand the rest of the paper. We assume a basic understanding of pointer analysis. We refer interested readers to [12], [4] for additional exposition.

For presentation, we use a simple LLVM-like language shown in Fig. 5. The language is used to simplify the presentation, but our implementation (in Sec. VI) supports full LLVM bitcode. Our language supports standard pointer and memory operations, but has no control flow constructs, such as conditional statements or loops, and defines a function by an unordered bag of instructions. This simplified setting is sufficient because our PTA is flow-insensitive – it does not use control-flow information. Although there are no global variables, they are modeled by explicitly passing them between functions. We allow passing and returning multiple values, modeled as a vector of function arguments and returns, respectively. For simplicity of presentation, we assume that all allocations create structures with exactly two fields, and that the size of an allocation is big enough to store any scalar type, including integers and pointers. New memory objects are created using the `alloc` instruction that allocates two fresh memory objects (one for each field) and returns a pointer to the first one. The result is saved in a register of type `char*`, that can be cast to a desired type with the `cast` instruction. Contents of a register is written to memory using `store` and read back with `load`. A *sibling* memory object I of an object H corresponding to field `a` is obtained with the `gep` (`GetElementPointer`) instruction with `b` as its field operand; applying the `gep` to H (or I) with a field operand `a` yields H (or I).

In PTA, the potentially infinite set of concrete memory object is mapped to a finite set of abstract objects. A standard way to identify abstract objects is by their *allocation site* – an `alloc` instruction that created them. A *points-to analysis (PTA)* of a program P computes a relation $\cdot \mapsto \cdot$, called points-to, between pointers and abstract objects. A PTA is sound if whenever $p \not\mapsto o$ then there is no execution of P in which p points to a concrete memory object corresponding to o. We represent PTAs using inference rules that derive facts of the $\mapsto$ relation. A PTA is computed by applying these rules until saturation. Fig. 6 contains a set of standard inference rules for the inclusion-based (Andersen-style) context-insensitive analysis in our language. We let $\mathbb{\Gamma}_{\mathrm{I}}$ represent the rules in Fig. 6 and denote a $\mapsto$ fact derivable by applying them exhaustively on a program P, written: $\mathbb{\Gamma}_{\mathrm{I}} \vdash_P x \mapsto H$, where x is a pointer and H is an abstract object. To support the `gep` instruction and make the PTA *field-sensitive*, we extend $\mathbb{\Gamma}_{\mathrm{I}}$ with additional rules $\mathbb{\Gamma}_{\mathrm{FLD}}$ shown in Fig. 7.

A *unification-based (Steensgaard-style)* PTA is obtained by extending the analysis with additional unification rules $\mathbb{\Gamma}_{\mathrm{U}}$ shown in Fig. 8, such that $\mathbb{\Gamma}_{\mathrm{STEENS}} = \mathbb{\Gamma}_{\mathrm{I}} \cup \mathbb{\Gamma}_{\mathrm{FLD}} \cup \mathbb{\Gamma}_{\mathrm{U}}$. The rules $\mathbb{\Gamma}_{\mathrm{U}}$ enforce the invariant (**I1**) from Sec. II. Note that $\mathbb{\Gamma}_{\mathrm{STEENS}}$ is less precise than $\mathbb{\Gamma}_{\mathrm{I}} \cup \mathbb{\Gamma}_{\mathrm{FLD}}$, because altering a PTA by adding extra inference rules never derives fewer $\mapsto$ facts. A *unification-based* PTA like $\mathbb{\Gamma}_{\mathrm{STEENS}}$ is typically implemented using the *Union-Find* data structure that allows to perform the abstract objects grouping in (almost) linear time.

$$\frac{\mathtt{r} \mapsto H \quad \mathtt{r} \mapsto I \quad \mathtt{p} \mapsto I}{\mathtt{p} \mapsto H}\ \textsc{Incoming} \qquad \frac{H \mapsto I \quad H \mapsto J \quad L \mapsto J}{L \mapsto I}\ \textsc{Incoming}$$

$$\frac{\mathtt{r} \mapsto H \quad H \mapsto J \quad \mathtt{r} \mapsto I \quad I \mapsto K}{H \mapsto K}\ \textsc{Outgoing} \qquad \frac{H \mapsto I \quad I \mapsto K \quad H \mapsto J \quad J \mapsto L}{I \mapsto L}\ \textsc{Outgoing}$$

Fig. 8: Unification rules: $\mathbb{\Gamma}_{\mathrm{U}}$.

IV. Keep Your Objects to Yourself

This section is organized as follows: first, we describe how to extend the $\mathbb{\Gamma}_{\textsc{Steens}}$ PTA to be *interprocedural* and explain *(calling) context-sensitivity*. Next, we show how to extend $\mathbb{\Gamma}_{\textsc{Steens}}$ to a DSA-style analysis. Using this formulation, we define the *oversharing* that happens in DSA, and show a way to reduce it. Finally, we show how to make the PTA partially *flow-sensitive* to further improve both precision and efficiency.

Context-sensitivity. The unification-based PTA $\mathbb{\Gamma}_{\textsc{Steens}}$ from Sec. III is an *intraprocedural analysis*. It analyzes a single function at a time and does not reason about other functions. *Interprocedural* reasoning requires propagating $\mapsto$ between *callers* and *callees* at all call-sites. For simplicity of explanation, we assume that calls are direct, i.e., callees are statically known, and that functions are not recursive.

A PTA is *(calling) context-insensitive* when it is interprocedural, but does not distinguish between calls to a function at different call-sites. For example, a context-insensitive unification-based PTA would not be able to tell apart `str4` and `r` passed to `print` in P_1, as illustrated in Fig. 1(b). A context-insensitive unification-based analysis is obtained by extending $\mathbb{\Gamma}_{\textsc{Steens}}$ with rules for interprocedural assignments.

A *(calling) context-sensitive* PTA provides $\mapsto$ facts relative to the requested calling context. In unification-based analyses, this is usually achieved by calculating a separate $\overset{F}{\longmapsto}$ relation for each function F in the analyzed program. DSA is an example of such an analysis [7]. Although not formally specified in [7], it is defined by adding rules to $\mathbb{\Gamma}_{\textsc{Steens}}$, $\mathbb{\Gamma}_{\text{DSA}} = \mathbb{\Gamma}_{\text{L}} \cup \mathbb{\Gamma}_{\text{BU}} \cup \mathbb{\Gamma}_{\text{TD}}$, where $\mathbb{\Gamma}_{\text{L}} = \mathbb{\Gamma}_{\textsc{Steens}} \cup \mathbb{\Gamma}_{\textsc{Formals}}$.

Formal arguments. To perform a local analysis of a function F, DSA calculates $\overset{\mathtt{F}}{\mapsto}$ based on instructions in F, including function calls. These instructions may access memory derived from *formal arguments*. Thus, it is necessary to introduce additional abstract objects for them. We refer to this kind of abstract objects as *formals*, and provide them for each defined function. Every formal argument of a function i : `fun fn(f): r`, $\mathtt{f}_k$, has six associated formals: $V^{\mathtt{a}}_{i,k}, V^{\mathtt{b}}_{i,k}, V^{\mathtt{aa}}_{i,k}, V^{\mathtt{ab}}_{i,k}, V^{\mathtt{ba}}_{i,k}, V^{\mathtt{bb}}_{i,k}$, corresponding to abstract objects for fields `a` and `b`, and abstract objects reachable by dereferencing each of these two fields. Fig. 9 shows inference rules $\mathbb{\Gamma}_{\textsc{Formals}}$ that specify how these abstract objects may point to each other. The rules model precisely only two levels of indirection. Any memory object obtained by a further dereference is mapped to the same second-level formal, adding a cycle. In practice, precision of analysis can

$$\frac{i : \mathtt{fun\ fn(}\overline{\mathtt{f}}\mathtt{):\ }\overline{\mathtt{r}} \quad 0 \leq k < |\overline{\mathtt{f}}|}{\begin{array}{c}\mathtt{f}_k \overset{\mathtt{fn}}{\longmapsto} V^{\mathtt{a}}_{i,k} \quad V^{\mathtt{a}}_{i,k} \overset{\mathtt{fn}}{\longmapsto} V^{\mathtt{aa}}_{i,k} \quad V^{\mathtt{b}}_{i,k} \overset{\mathtt{fn}}{\longmapsto} V^{\mathtt{ba}}_{i,k} \\ V^{\mathtt{aa}}_{i,k} \overset{\mathtt{fn}}{\longmapsto} V^{\mathtt{aa}}_{i,k} \quad V^{\mathtt{ba}}_{i,k} \overset{\mathtt{fn}}{\longmapsto} V^{\mathtt{ba}}_{i,k} \\ V^{\mathtt{ab}}_{i,k} \overset{\mathtt{fn}}{\longmapsto} V^{\mathtt{ab}}_{i,k} \quad V^{\mathtt{bb}}_{i,k} \overset{\mathtt{fn}}{\longmapsto} V^{\mathtt{bb}}_{i,k}\end{array}}\ \textsc{Formals}$$

Fig. 9: Inference rules for formal arguments: $\mathbb{\Gamma}_{\textsc{Formals}}$.

be improved by computing the necessary levels of indirection (e.g., [13], [7], [8]).

Oversharing. While the local analysis $\mathbb{\Gamma}_{\text{L}}$ only uses abstract objects from the analyzed function (i.e., coming from allocation sites in that function or its formals), the rules $\mathbb{\Gamma}_{\text{BU}} \cup \mathbb{\Gamma}_{\text{TD}}$, shown in Fig. 10, propagate $\mapsto$ facts across functions. They use a helper function *Resolve* to map between caller and callee abstract objects. For any pair of functions F_1 and F_2, we refer to the abstract objects defined by F_2 and present in $\overset{F_1}{\longmapsto}$ as *foreign*. A foreign object is *overshared* in F_1 if it is inaccessible by F_1, but needlessly appears in the analysis results of F_1.

DSA, as presented in [7], executes three phases of the analysis for a function F as follows: (a) Local phase for F; (b) Bottom-Up for each callee of F; and (c) Top-Down for each caller of F. This is equivalent to applying the $\mathbb{\Gamma}_{\text{L}}$ and $\mathbb{\Gamma}_{\text{BU}}$ rules until saturation in a reverse-topological call-graph order, followed by $\mathbb{\Gamma}_{\text{TD}}$ in a topological order until saturation. The rules can be soundly applied in this sequence and no new $\overset{F}{\longmapsto}$ facts can be derived by running any of the phases again. The original DSA implementation performs foreign object propagation during both Bottom-Up and Top-Down: Bottom-Up copies foreign abstract objects *accessible* from formal arguments and returned values from a callee to its callers, while Top-Down copies *all* abstract objects *accessible* (directly or transitively) from function parameters (actual arguments) in a caller to its callees, even if they are unused. We notice that the copying of foreign objects in Top-Down, required to maintain (**I2**) from Sec. II, is a major source of oversharing in DSA. This form of oversharing led to a workaround in [7] that improves performance at expense of precision by treating all global variables (major source of foreign objects) context-insensitively.

Our first contribution is to show that such an oversharing of foreign abstract objects is unnecessary. All abstract objects of a function are known after Local and Bottom-Up phases:

Theorem 1 $(\mathbb{\Gamma}_{\text{DSA}} \vdash_P x \overset{F}{\longmapsto} H) \implies \exists y \cdot (\mathbb{\Gamma}_{\text{L}} \cup \mathbb{\Gamma}_{\text{BU}} \vdash_P y \overset{F}{\longmapsto} H)$*, where x and y are registers or abstract objects.*

Theorem 1 states that no new foreign objects are ever introduced by $\mathbb{\Gamma}_{\text{TD}}$. The derivable $\overset{F}{\longmapsto}$ facts are always over abstract objects *resolved* from callee's abstract object to caller's abstract objects. The proof of Theorem 1 follows from the fact that $\mathbb{\Gamma}_{\text{L}}$ models the operational semantics of our language, and that our formulation of interprocedural rules explicitly uses the callee-caller resolution of abstract objects[1].

[1] All proofs are available in the extended version of the paper [14].

The simplicity of Theorem 1 is solely due to our new formulation of DSA. Prior works ([7], [15]) miss this, now obvious fact. With our formulation, it is clear that the role of TOP-DOWN is to use $\overset{F}{\longmapsto}$ at a call-site and use it to instantiate a fully-general summary for a callee by introducing necessary $\overset{F}{\longmapsto}$ between function arguments and formals. If a client of a PTA requires to know not only $\overset{F}{\longmapsto}$ but also all the mapping from formals to allocation sites each formal may originate from, it is possible to maintain such information separately, without introducing oversharing during TOP-DOWN. Our evaluation (Sec. VI) demonstrates that this improves performance and precision.

Partial Flow-sensitivity. Our second contribution is to identify additional opportunities to reduce oversharing by increasing the precision of the analysis at interprocedural assignments – call- and return-sites. Overall precision of a PTA can be improved by making the LOCAL phase more precise, or by not propagating the local imprecision interprocedurally. In DSA, a function with an instruction that operates on two abstract objects can cause these abstract objects to be grouped in any subsequent function, provided enough interprocedural assignments. The source of the problem is that DSA preserves any local grouping of abstract objects by maintaining (**I2**) and (**I3**) from Sec. II. Due to the $\mathbb{\Gamma}_{\mathrm{U}}$ rules, such confusion can reduce the precision of the whole PTA. For example, once o_1 and o_2 are grouped together in `getStr` from Fig. 1(a), in DSA the grouping is propagated bottom-up to `foo` and `bar`.

Flow-sensitivity is a simple way to increase precision of a PTA at a cost of performance. A flow-sensitive analysis computes a relation $\overset{F@i}{\longmapsto}$ not only at the function level (F), but also relative to a particular instruction (i) within F. To improve precision for interprocedural assignments, we need to know where each function parameter points to at a particular call- or return-site. For example, in P_1 from Fig. 1(a), $\mathtt{str1} \overset{\mathtt{getStr@11}}{\longmapsto} o_1$ at the return statement. We call this refinement *partial flow-sensitivity*. We present a set of rules, $\mathbb{\Gamma}_{\mathrm{PFS}}$ in Fig. 11, that combine together with $\mathbb{\Gamma}_{\mathrm{DSA}}$ to define an analysis called $\mathbb{\Gamma}_{\mathrm{PFS\text{-}DSA}}$. Note that $\mathbb{\Gamma}_{\mathrm{PFS}}$ *replaces* the corresponding two rules from $\mathbb{\Gamma}_{\mathrm{DSA}}$. We assume that $\overset{F@i}{\longmapsto}$ is externally defined and is a (sound) subset of $\overset{F}{\longmapsto}$. BOTTOM-UP-1 rule of $\mathbb{\Gamma}_{\mathrm{PFS}}$ propagates $\overset{\mathtt{callee@}j}{\longmapsto}$ (points-to information at the return-site) into $\overset{\mathtt{caller}}{\longmapsto}$, by *resolving* abstract objects across these two functions; formals from `callee` get matched with abstract objects passed into it at the call-site, while allocation sites from `callee` are resolved to themselves. Similarly, TOP-DOWN-1 resolves abstract objects reachable from parameters at a call-site into appropriate formals for the `callee`.

Partial flow-sensitivity is much cheaper than a (full) flow-sensitivity, as we do not even need to maintain a separate flow-sensitive $\overset{F}{\longmapsto}$ at call and return sites. This is because it is often enough to perform a very cheap local reasoning to determine that given a local fact $p \overset{F}{\longmapsto} o$, $p \overset{F@i}{\not\longmapsto} o$. For instance, $\mathtt{str1} \overset{\mathtt{getStr@11}}{\not\longmapsto} o_2$ because the variable name `str1` is used explicitly at the return-site, and the variable `str1` is never reassigned, it must only point to o_1 at line 11.

The only difference between $\mathbb{\Gamma}_{\mathrm{PDF\text{-}DSA}}$ and $\mathbb{\Gamma}_{\mathrm{DSA}}$ is the use of the $\overset{F@i}{\longmapsto}$ relation instead of $\overset{F}{\longmapsto}$ in BOTTOM-UP and TOP-DOWN rules, where $\overset{F@i}{\longmapsto}$ is a subset of $\overset{F}{\longmapsto}$. Assuming $\overset{F@i}{\longmapsto}$ is sound at a call-site (return-site), every $\overset{F}{\longmapsto}$ fact is correctly propagated by the interprocedural assignment rules.

V. BE AWARE OF YOUR TYPE

The *effective type rules* of the C11 standard [10, Sec. 6.5] say that memory is dynamically strongly typed: roughly, a memory read (`load`) of an object *co* is valid only when the last write (`store`) to *co* was of a compatible type. Thus, pointers of incompatible types do not alias. Other languages, including C++ and SWIFT, impose similar rules typically called *strict aliasing*. Strict aliasing is widely exploited in all major optimizing compilers. In this paper, we use it to improve precision of the LOCAL phase of TEADSA.

We assume that a *type compatibility* relation, $\sqsubseteq$, on types, is provided as an *input* to our analysis. For our simple language, the compatibility relation is defined as a *partial order* s.t.:

$$\forall \tau \in T \cdot \tau \sqsubseteq \mathtt{char} \qquad \forall \tau \in PT \cdot \tau \sqsubseteq \mathtt{char\star}$$

That is, `char` is compatible with all other types, `char⋆` is compatible with all pointer types, and every type is compatible with itself, but `int` and `float` are not compatible. In our implementation, we use a more sophisticated type lattice to handle LLVM's structure types. It is also possible to use the type lattice of a compiler frontend (e.g., CLANG's TBAA tags).

Due to the low-level nature of our language, allocations and function definitions do not specify the types of objects. To allow untyped allocations and function arguments, we extend our notion of abstract objects to include object type. For example, an i `: r = alloc()` instruction has $|fld| \times |T|$ allocation sites of a form H_i^T – one for each field of any possible type. Similarly, each formal function argument has $|\{\mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}\}| \times |T|$ formals. In our implementation, we discover abstract object types on demand. We modify the basic $\mapsto$ relation to include the type of the pointed-to abstract object and disambiguate it from abstract objects of other types. For example, a fact $\mathtt{r} \overset{T}{\longmapsto} H$ means: the register `r` may point to the abstract object H of type T. A sample points-to graph for a type-aware PTA of a program in Fig. 3 is shown in Fig. 4.

Although the type of each register is known statically, we only require memory operations (`load` and `store`) to access objects using compatible types, while types used in function calls, `cast`, and `gep` instructions are ignored. Instead of relying on declared types, we discover them at memory accesses, as shown in type-awareness rules $\mathbb{\Gamma}_{\mathrm{TY}}$ in Fig. 12. We say that a pointer returned by an `alloc` may point to *any* abstract object for the field `a` created at this allocation site, and express that with a $\overset{\mathtt{char}}{\longmapsto}$ fact, as `char` is compatible with all types. A `load` accesses only the abstract object pointed-to by the pointer operand if they are of a compatible type. Similarly, the type of the destination register of a `store` dictates which

$$\frac{\begin{array}{c} i : \overline{\mathtt{y}} \texttt{ = callee(}\overline{\mathtt{x}}\texttt{)} \quad \mathit{fun}(i) = \texttt{caller} \quad j : \texttt{return } \overline{\mathtt{z}} \\ \mathit{fun}(j) = \texttt{callee} \quad \mathtt{z}_k \xmapsto{\texttt{callee}} H \quad \mathit{Resolve}(i, H, I) \end{array}}{\mathtt{y}_k \xmapsto{\texttt{caller}} I} \textsc{Bottom-Up-1}$$

$$\frac{\begin{array}{c} i : \overline{\mathtt{y}} \texttt{ = callee(}\overline{\mathtt{x}}\texttt{)} \quad \mathit{fun}(i) = \texttt{caller} \quad \mathit{Accessible}(\texttt{callee}, J) \\ J \xmapsto{\texttt{callee}} K \quad \mathit{Resolve}(i, J, H) \quad \mathit{Resolve}(i, K, I) \end{array}}{H \xmapsto{\texttt{caller}} I} \textsc{Bottom-Up-2}$$

$$\frac{\begin{array}{c} i : \overline{\mathtt{y}} \texttt{ = callee(}\overline{\mathtt{x}}\texttt{)} \quad \mathit{fun}(i) = \texttt{caller} \quad \mathtt{x}_k \xmapsto{\texttt{caller}} H \\ j : \texttt{fun callee(}\overline{\mathtt{f}}\texttt{) : } \overline{\mathtt{r}} \quad \mathit{Resolve}(i, I, H) \end{array}}{\mathtt{f}_k \xmapsto{\texttt{callee}} I} \textsc{Top-Down-1}$$

$$\frac{\begin{array}{c} i : \overline{\mathtt{y}} \texttt{ = callee(}\overline{\mathtt{x}}\texttt{)} \quad \mathit{fun}(i) = \texttt{caller} \quad H \xmapsto{\texttt{caller}} I \\ \mathit{Resolve}(i, J, H) \quad \mathit{isFormal}(J) \quad \mathit{Resolve}(i, K, I) \quad \mathit{isFormal}(K) \end{array}}{J \xmapsto{\texttt{callee}} K} \textsc{Top-Down-2}$$

Fig. 10: Inference rules for Context-Sensitivity: $\mathbb{\Gamma}_{\mathrm{BU}}$ and $\mathbb{\Gamma}_{\mathrm{TD}}$.

$$\frac{\begin{array}{c} i : \overline{\mathtt{y}} \texttt{ = callee(}\overline{\mathtt{x}}\texttt{)} \quad \mathit{fun}(i) = \texttt{caller} \\ j : \texttt{return } \overline{\mathtt{z}} \quad \mathit{fun}(j) = \texttt{callee} \\ \mathtt{z}_k \xmapsto{\texttt{callee@}j} H \quad \mathit{Resolve}(i, H, I) \end{array}}{\mathtt{y}_k \xmapsto{\texttt{caller}} I} \textsc{Bottom-Up-1}$$

$$\frac{\begin{array}{c} i : \overline{\mathtt{y}} \texttt{ = callee(}\overline{\mathtt{x}}\texttt{)} \quad \mathit{fun}(i) = \texttt{caller} \\ \mathtt{x}_k \xmapsto{\texttt{caller@}i} H \\ j : \texttt{fun callee(}\overline{\mathtt{f}}\texttt{) : } \overline{\mathtt{r}} \quad \mathit{Resolve}(i, I, H) \end{array}}{\mathtt{f}_k \xmapsto{\texttt{callee}} I} \textsc{Top-Down-1}$$

Fig. 11: Inference rules for Partial Flow-Sensitivity: $\mathbb{\Gamma}_{\mathrm{PFS}}$.

$$\frac{i : \texttt{r = alloc()}}{\mathtt{r} \xmapsto{\texttt{char}} H_i} \textsc{Alloc} \qquad \frac{\begin{array}{c} \texttt{r = load T* p} \\ \mathtt{p} \overset{\mathtt{U}}{\mapsto} H \quad \mathtt{T} \sqsubseteq \mathtt{U} \\ H \overset{\mathtt{X}}{\mapsto} I \end{array}}{\mathtt{r} \overset{\mathtt{X}}{\mapsto} I} \textsc{Load} \qquad \frac{\begin{array}{c} \texttt{store r, T* p} \\ \mathtt{p} \overset{\mathtt{U}}{\mapsto} H \quad \mathtt{U} \sqsubseteq \mathtt{T} \\ \mathtt{r} \overset{\mathtt{X}}{\mapsto} I \end{array}}{H \overset{\mathtt{X}}{\mapsto} I} \textsc{Store}$$

Fig. 12: Type-awareness rules: $\mathbb{\Gamma}_{\mathrm{TY}}$.

abstract object may be written to. For example, consider a simple class hierarchy $C \sqsubseteq B \sqsubseteq A$, where A is a *superclass* of both B and C, while B is a *superclass* of C. In our formalization, a `load B*` p can access both abstract objects of type A and B, whereas a `store` v, `B*` p writes to abstract objects of type B and C. Such a conservative handling of memory operations, consistent with the *strict aliasing rules* of C11, guarantees soundness of a PTA extended with type-awareness rules.

The $\mathbb{\Gamma}_{\mathrm{TY}}$ rules *replace* the rules in $\mathbb{\Gamma}_{\mathrm{I}}$; we omit the remaining replacement rules that use $\overset{T}{\mapsto}$ instead of $\mapsto$, as the modification is straightforward. Finally, we define $\mathbb{\Gamma}_{\mathrm{TEADSA}}$ to be the modified set rules $\mathbb{\Gamma}_{\mathrm{PFS\text{-}DSA}}$ based on $\mathbb{\Gamma}_{\mathrm{TY}}$ and the $\overset{T}{\mapsto}$ relation. Type-awareness improves both the local and global analysis precision, and in turn further reduces oversharing:

Theorem 2 $\mathbb{\Gamma}_{\mathrm{PFS\text{-}DSA}} \vdash_P x \not\mapsto H \implies \mathbb{\Gamma}_{\mathrm{TEADSA}} \vdash_P x \overset{T}{\not\mapsto} H$

Theorem 2 says that $\mathbb{\Gamma}_{\mathrm{TEADSA}}$ is not less precise than $\mathbb{\Gamma}_{\mathrm{PFS\text{-}DSA}}$, i.e., no points-to relation not present in analysis results for $\mathbb{\Gamma}_{\mathrm{PFS\text{-}DSA}}$ is present in analysis results for $\mathbb{\Gamma}_{\mathrm{TEADSA}}$. This is because the type-aware rules for `load` and `store` are similar to $\mathbb{\Gamma}_{\mathrm{I}}$, except that they prevent loads from deriving facts about stores of incompatible types. With the most conservative compatibility relation (i.e., all types are compatible), the $\mathbb{\Gamma}_{\mathrm{TY}}$ would derive exactly the same $\mapsto$ facts as $\mathbb{\Gamma}_{\mathrm{I}}$.

VI. Implementation and Evaluation

In this section, we describe our implementation of TEADSA and compare its scalability and precision against other state-of-the-art PTAs. To meaningfully compare precision, we developed a checker for a class of memory safety violations, and use it to evaluate the PTAs on a set of C and C++ programs. Our implementation, benchmarks, and experiments are available at https://github.com/seahorn/sea-dsa/releases/tag/tea-dsa-fmcad19.

Implementation. We implemented TEADSA on top of SEADSA – a context-, field-, and array-sensitive DSA-style PTA for LLVM [8]. Our implementation inherits many of the advantages of SEADSA, including: an effective representation of $\mapsto$ using a union-find data-structure; three analysis passes (local, bottom-up, top-down); modular analysis of each function; support for `gep` instructions with fixed and symbolic offsets; handling recursion by losing context sensitivity for strongly connected components in the call graph; and, on-demand discovery of abstract objects for fields, formals, as well as their corresponding types. In the evaluation, we devirtualize indirect calls. For partial flow-sensitivity, we disambiguate pointers that must alias known allocation sites from other objects in their points-to sets, and do not propagate stack-allocated abstract objects bottom-up. We use the type compatibility relation $\sqsubseteq$ based on the type tags in the code such that the type of each structure is the same as the type of its first (innermost) field. Two types are compatible if they have the same type tag.

The client. We chose a problem of statically detecting *field overflow* bugs. A field-overflow happens when an instruction accesses a nonexistent field of an object, such that the memory access is outside of the allocated memory object. For example, consider the field access in line 19 in Fig. 3 – loading the value of the field `d` is not safe if the pointer `ie` is pointing to an object of an insufficient size, e.g, o_6. To determine whether a field access through a pointer p causes a field overflow, we identify the set A of all the allocation sites that p might point to. Then, any allocation site $a \in A$ of an insufficient size might cause a field overflow. We have implemented such a field-overflow-checker in SEAHORN [3].

Evaluation. We compare TEADSA with two state-of-the-art interprocedural PTAs for LLVM: SVF [11] and SEADSA [8]. SVF [11] is a flow-sensitive, context-insensitive, inclusion-based PTA. We compare against two variants of SVF: the most precise Sparse Flow-sensitive analysis (*SVF Sparse*), and the

same analysis with the *Wave Diff* pre-analysis. As for DSA-style analyses, we use SEADSA, PFS-SEADSA, and TEADSA to denote SEADSA, our implementation of $\mathbb{\Gamma}_{\text{PFS-DSA}}$, and our implementation of $\mathbb{\Gamma}_{\text{TEADSA}}$, respectively. Note that we do not use the DSA implementation from LLVM's POOL-ALLOC, as it is not maintained and crashes on many of our examples.

We perform the evaluation on a set of C and C++ programs. The programs vary in size, ranging from 140kB to 158MB of LLVM bitcode. All experiments are done on a Linux machine with two Intel Xeon E5-2690v2 10-core processors and 128GB of memory. We present performance results of running PTAs in Table I and precision on the field-overflow detection in Table II. In the tables, – denotes that an experiment did not finish within 3 hours or exceeded the 80GB memory limit and was terminated. To ensure that all PTAs are working in a consistent environment, we modified SVF to use the same notion of allocation sites that is used by TEADSA. We asses the precision of the PTAs using our field-overflow checker. In Table II, we use *Aliases* to denote the number of reported $\langle \text{allocation site}, \text{accessed pointer} \rangle$ pairs, and *Checks* as the number of assertions necessary to show that the analyzed program is free of field overflow bugs. The lower the numbers, the more precise a PTA is.

In our experiments, TEADSA is almost always the most scalable PTA, both in terms of runtime and memory use, closely followed by PFS-SEADSA. These two analyses scaled an order of magnitude better than the plain version of SEADSA. TEADSA was faster than SVF, especially on large programs like LLVM tools (prefix llvm-), where it finished in seconds instead of hours. As for precision, TEADSA and SVF achieved similar results on most of the smaller programs. TEADSA is strictly more precise than SEADSA, and, surprisingly, more precise than SVF on C++ programs such as cass, WEBASSEMBLY tools (prefix wasm-), LLVM tools, and on the C program htop that uses a C++-like coding style. When performing a closer comparison of PFS-SEADSA vs SEADSA, we noticed that the performance improvement can be attributed to not copying foreign objects during TOP-DOWN (up to 96% shorter running time on wasm-opt), while partial flow-sensitivity explains most of the increase in precision (up to 25% fewer aliases on h264ref).

VII. RELATED WORK

There is a large body of work on points-to analysis, both for low-level languages and for higher-level languages like Java. Throughout the paper, we compare with the closest related work: DSA [7] and SEADSA [8]. In Sec. VI, we compared empirically with two context-insensitive, inclusion-based implementations of SVF [11] – a state-of-the-art PTA framework for LLVM. In the rest of this section, we compare with other related works.

Sui et al. [13] present a context-sensitive, inclusion-based pointer analysis, called ICON. The fact that ICON is an inclusion-based PTA and SEADSA is unification-based makes it hard to compare them without an experimental evaluation. Unfortunately, ICON is not part of the SVF framework and its implementation is not publicly available. Therefore, comparing experimentally is not possible.

The precision of inclusion-based pointer analyses can be improved by flow-sensitivity (e.g. [16], [17]). However, unification-based PTA are always flow-insensitive to retain their efficiency. In our work, we improve a context-sensitive, unification-based PTA by making it flow-sensitive only at call and return statements. This allows us to improve the precision of the analysis without jeopardizing its efficiency.

Using types to improve precision of a PTA is not new. Structure-sensitive PTA [18] extends a whole-program, inclusion-based PTA with types. The analysis is object and type-sensitive ([19]). This work is orthogonal to ours. The main purpose of type sensitivity is to distinguish multiple abstract memory objects from a given (untyped) heap allocation (e.g., `malloc`) based on their uses. This avoids aliasing among objects that are originated from the same allocation wrapper or a factory method. We do not tackle this problem. Instead, we use types to avoid unrealized aliasing under the strict aliasing rules. We mitigate the problem of using allocation wrappers by inlining memory allocating functions.

Rakamaric and Hu [20] use DSA ability to track types for an efficient encoding of verification conditions (VC) for program analysis. Their approach differs significantly from ours. They do not tackle the problem of improving the precision of a pointer analysis using types. Instead, they extract useful type information from a PTA to produce more efficient VCs.

VIII. CONCLUSION

We identify a major deficiency of context-sensitive unification-based PTA's, called *oversharing*, that affects both scalability and precision. We present TEADSA– a DSA-style PTA that eliminates a class of oversharing during the TOP-DOWN analysis phase and further reduces it using flow-sensitivity at call- and return-sites, and typing information. Our evaluation shows that avoiding such an oversharing makes the analysis much faster than DSA, as well as more precise than DSA on our program verification problem. The results are very promising – TEADSA compares favorably against SVF in scalability in the presented benchmarks, and sometimes shows even better precision results.

Acknowledgments. This material is based upon work supported by US NSF grants 1528153 and 1817204 and the Office of Naval Research under contract no. N68335-17-C-0558 and by an Individual Discovery Grant from the Natural Sciences and Engineering Research Council of Canada. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research.

REFERENCES

[1] H. Yan, Y. Sui, S. Chen, and J. Xue, "Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities," in *ICSE*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 327–337.

Program	Bitcode Size [kB]	Results									
		Wave Diff		SVF Sparse		SEADSA		PFS-SEADSA		TEADSA	
		Runtime [s]	Memory [MB]	Runtime [s]	Memory [MB]	Runtime [s]	Memory [MB]	Runtime [s]	Memory [MB]	Runtime [s]	Memory [MB]
sqlite	140	**<1**	106	**<1**	135	**<1**	57	**<1**	**19**	**<1**	**19**
bftpd	268	**<1**	114	**<1**	133	**<1**	50	**<1**	**24**	**<1**	**24**
htop	320	**<1**	216	7	483	**<1**	242	**<1**	49	**<1**	**37**
cass	1,384	**<1**	399	1	453	1	375	**<1**	**45**	**<1**	46
wasm-dis	1,420	6	1,041	122	4,594	1	618	**<1**	169	**<1**	**98**
openssl	1,504	1	706	2	792	1	683	**<1**	59	**<1**	**58**
wasm-as	1,824	10	1,428	195	8,249	2	1,162	**<1**	248	**<1**	**149**
h264ref	2,468	6	1,655	7	1,784	5	2,323	**<1**	**183**	**<1**	197
tmux	2,996	1	586	3	696	1	649	**<1**	144	**<1**	**125**
wasm-opt	3,520	36	2,784	960	33,138	51	23,339	**1**	1,507	**1**	**308**
llvm-dis	11,232	1,640	9,964	–	–	–	–	18	4,587	**16**	**3,254**
llvm-as	14,012	4,892	15,377	–	–	–	–	24	7,130	**19**	**4,100**
llvm-opt	16,012	9,104	19,633	–	–	–	–	55	20,555	**27**	**8,319**
rippled	157,804	–	–	–	–	–	–	379	55,691	**308**	**25,626**

TABLE I: Performance of different PTAs.

Program	Bitcode Size [kB]	Results									
		Wave Diff		SVF Sparse		SEADSA		PFS-SEADSA		TEADSA	
		Checks	Aliases	Checks	Aliases	Checks	Aliases	Checks	Aliases	Checks	Aliases
sqlite	140	**<1k**	**<1k**	**<1k**	**<1k**	1k	3k	1k	3k	1k	3k
bftpd	268	<1k	<1k	**<1k**	**<1k**	<1k	1k	<1k	1k	<1k	<1k
htop	320	24k	26k	24k	26k	110k	110k	109k	109k	**9k**	**11k**
cass	1,384	1k	7k	1k	7k	12k	14k	3k	12k	**<1k**	**3k**
wasm-dis	1,420	136k	253k	132k	241k	616k	634k	539k	558k	**119k**	**132k**
openssl	1,504	**<1k**	2k	**<1k**	**2k**	<1k	4k	<1k	4k	<1k	4k
wasm-as	1,824	248k	424k	**243k**	412k	933k	957k	823k	849k	293k	**317k**
h264ref	2,468	**<1k**	38k	**<1k**	37k	21k	174k	15k	148k	3k	**34k**
tmux	2,996	8k	17k	**8k**	**17k**	403k	422k	391k	410k	333k	350k
wasm-opt	3,520	724k	1,196k	718k	1,174k	8,851k	8,637k	7,632k	7,466k	**603k**	**645k**
llvm-dis	11,232	6,107k	6,842k	–	–	–	–	4,358k	4,391k	**1,097k**	**1,404k**
llvm-as	14,012	12,198k	13,866k	–	–	–	–	8,992k	9,017k	**2,138k**	**2,470k**
llvm-opt	16,012	16,346k	17,140k	–	–	–	–	47,174k	47,421k	**9,551k**	**13,878k**
rippled	157,804	–	–	–	–	–	–	130,957k	129,910k	**47,415k**	**47,848k**

TABLE II: Precision of different PTAs.

[2] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang, "Pinpoint: fast and precise sparse value flow analysis for million lines of code," in *PLDI*, 2018, pp. 693–706.

[3] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The SeaHorn verification framework," in *CAV*, 2015, pp. 343–361.

[4] M. Hind, "Pointer analysis: haven't we solved this problem yet?" in *PASTE*, 2001, pp. 54–61.

[5] L. O. Andersen, "Program Analysis and Specialization for the C Programming Language," Ph.D. dissertation, DIKU, University of Copenhagen, 1994.

[6] B. Steensgaard, "Points-to analysis in almost linear time," in *POPL*, 1996, pp. 32–41.

[7] C. Lattner and V. S. Adve, "Automatic pool allocation: improving performance by controlling data structure layout in the heap," in *PLDI*, 2005, pp. 129–142.

[8] A. Gurfinkel and J. A. Navas, "A context-sensitive memory model for verification of C/C++ programs," in *SAS*, 2017, pp. 148–168.

[9] D. Beyer, "Automatic verification of C and Java programs: SV-COMP 2019," in *TACAS*, 2019, pp. 133–155.

[10] ISO, *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, Dec. 2011.

[11] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *CC*, 2016, pp. 265–266.

[12] Y. Smaragdakis, G. Balatsouras *et al.*, "Pointer analysis," *Foundations and Trends® in Programming Languages*, vol. 2, no. 1, pp. 1–69, 2015.

[13] Y. Sui, S. Ye, J. Xue, and J. Zhang, "Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation," *Softw., Pract. Exper.*, vol. 44, no. 12, pp. 1485–1510, 2014.

[14] J. Kuderski, "Scalable context-sensitive pointer analysis for LLVM," Master's thesis, University of Waterloo, 2019. [Online]. Available: https://hdl.handle.net/10012/14875

[15] R. Madhavan, G. Ramalingam, and K. Vaswani, "A framework for efficient modular heap analysis," *Foundations and Trends in Programming Languages*, vol. 1, no. 4, pp. 269–381, 2015.

[16] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *CGO*, 2011, pp. 289–298.

[17] Y. Sui, P. Di, and J. Xue, "Sparse flow-sensitive pointer analysis for multithreaded programs," in *CGO*, 2016, pp. 160–170.

[18] G. Balatsouras and Y. Smaragdakis, "Structure-sensitive points-to analysis for C and C++," in *SAS*, 2016, pp. 84–104.

[19] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: Understanding object-sensitivity," in *POPL*, 2011, pp. 17–30.

[20] Z. Rakamaric and A. J. Hu, "A scalable memory model for low-level code," in *VMCAI*, 2009, pp. 290–304.