Scalable Context-Sensitive Pointer Analysis for LLVM

by

Jakub Kuderski

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2019

Author's Declaration

Statement of Contributions

Some figures, tables, and text are restarted from the TeaDsa paper [10]. Some of the described modifications to SeaDsa were implemented by Prof. Arie Gurfinkel and Dr. Jorge A. Navas.

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 for low-level languages that are moderately precise while scalable to large real-world programs. In this thesis, 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, with such an oversharing significantly reduced. 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.

Acknowledgements

I would like to thank all the people who helped me during this degree. Most notably, I thank my supervisor, Prof. Arie Gurfinkel, for hundreds of hours spent on explanations and discussions, for introducing me to verification, and continuous guidance through my degree at UWaterloo. His mentorship was vital to my success. I thank Dr. Jorge A. Navas who worked with me on several improvements to both SeaHorn and SeaDsa, and hosted me at SRI International. I also appreciate all the helpful comments and remarks to this thesis submitted by the readers: Prof. Werner Dietl and Prof. Ondřej Lhoták.

I also thank Dr. Omer Tripp for hosting me at Google. I am grateful for the funding and scholarships provided by NSERC, UWaterloo, and CMU-SEI. I thank the LLVM Foundation for funding my trips to conferences.

I appreciate all the effort that my former URAs and mentees put into their projects: Chijun Sima who worked on DomTreeUpdater in LLVM; Matvey Shvartszayd for the work on STL for verification; Xiaoxiao Wang and Qizhi Wen for the work on the SeaCut refactoring tool.

I thank my friends from my research group, Hari Govind V. K. and Nhâm Lê, for all the time spent together inside and outside the lab. Lastly, I am thankful to my family, especially: my mom Agnieszka, my brother Bartosz, my partner Ye; and to all my other friends for always supporting me.

Dedication

This is dedicated to my family who gave me enormous support throughout my whole education.

Table of Contents

List of Tables

List of Figures

Chapter 1

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., [24, 14, 4]).

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 [7] (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 such as C, C++, and LLVM 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.

The situation is better for higher-level languages such as JAVA [2, 8, 7, 18, 23] and GO[1]. In this thesis, we focus on the PTA problem for low-level languages.

[1]https://godoc.org/golang.org/x/tools/go/pointer.

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* [1]) or *unification-based* (a.k.a., *Steensgaard-style* [19]). 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 into a prohibitive analysis.

State-of-the-art implementations of unification-based, context-sensitive PTAs (such as DSA [11] and SeaDsa [5]) 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 Chapter 7 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 aggregates precisely, 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 thesis, we first present a modular formulation of unification-based pointer analyses and use it to describe the main sources of oversharing in a typical combination with context-sensitivity. Based on the properties of our formulation, we show that a class of such an oversharing can be avoided. Then, we present a new pointer analysis for LLVM, called TeaDsa, that eliminates this 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 and treating global variables context-insensitively. SeaDsa does not tackle this problem since it focuses on medium-size programs such as the SV-COMP benchmarks [3].

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. Flow-sensitivity is achieved by not propagating memory objects across functions if a more precise local analysis can infer that they do not alias parameters or returned values, or if the memory objects are not accessible outside of the current scope.

Third, we noted that another source of imprecision in SeaDsa is (partial) 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 [9]. 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.

Chapter 2

Overview

In this chapter, we illustrate our approach on a series of simple examples. Consider a C program P_1 in Fig. 2.1(a) and its corresponding context-insensitive and flow-insensitive points-to graph G_1 in Fig. 2.1(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. 2.1(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.

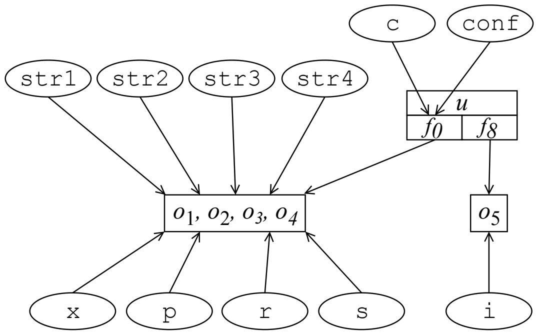
The edges of G_1 denote whether a pointer p may point to an abstract object o, written $p \mapsto o$. Whenever p may point to multiple abstract objects all of these objects are grouped into a single (rectangular) node. For instance, $\texttt{x} \mapsto o_1$, $\texttt{x} \mapsto o_2$, $\texttt{x} \mapsto o_3$, $\texttt{x} \mapsto o_4$, 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. 2.1(b) corresponds to the Steensgaard (or unification-based) PTA [19]. 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

```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; };
```

```c
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)

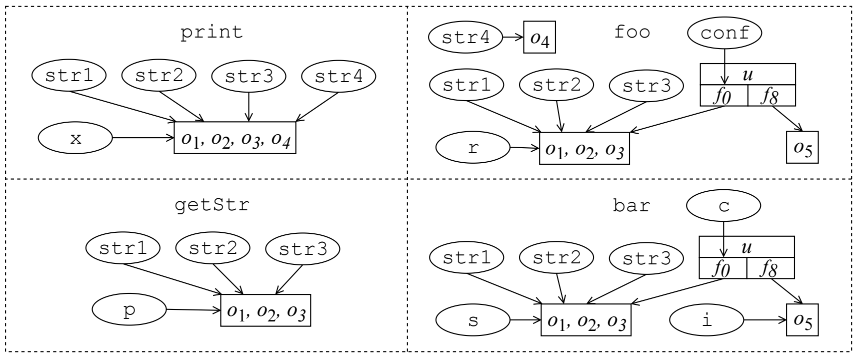


(b)

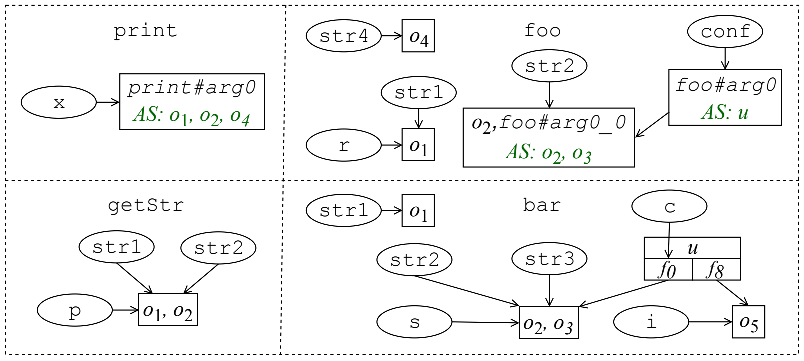
Figure 2.1: Sample C program P_1 (a) and its Context-insensitive Points-To Graph G_1 (b).

may alias, which is clearly not the case. For instance, `s` $\mapsto \{o_1, o_2, o_3, o_4\}$ in Fig. 2.1(b), even though there is no execution in which `s` $\mapsto o_1$ or `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) [11] is an example of a context-sensitive Steensgaard PTA. The results of a context-sensitive Steensgaard PTA on P_1 are shown in Fig. 2.2(a) as four separate points-to graphs – one for each procedure in P_1. An increase in precision (compared to the PTA in Fig. 2.1(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.



(a)



(b)

Figure 2.2: Context-sensitive Points-To Graphs for P_1.

In Chapter 7, 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. 2.1(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. 2.2(a), as computed by a DSA-like PTA. This performance issue was already observed in [11], but only a workaround that loses context-sensitivity for global objects was implemented.

In this thesis, 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. 2.2(a), in our proposed analysis, Fig. 2.2(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. 2.1(a), `str1` $\mapsto \{o_1, o_2\}$ in `getStr`, thus the returned value `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 `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 `r` $\mapsto o_1$ and `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 thesis, 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. 2.3: 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. In the P_2's points-to graph in Fig. 2.4, type tags are shown at the bottom of each abstract object. We maintain soundness by discovering type tags based only on memory accesses 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

```
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);
    }
}
```

Figure 2.3: Sample C program P_2.

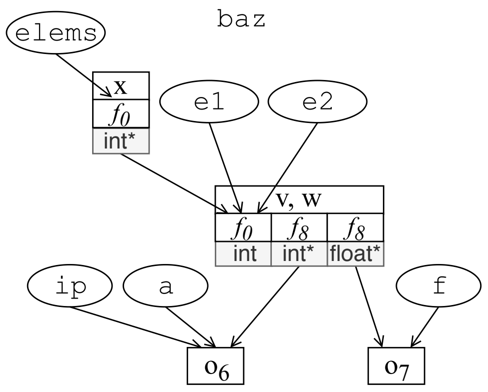


Figure 2.4: Type-aware Points-To Graph of P_2.

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, there 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 [21], 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.

Chapter 3

Background

In this chapter, we present the necessary background to understand the rest of the thesis. We summarize the basic concepts behind pointer analyses and their properties, such as flow-sensitivity and field-sensitivity. We establish a formal notation used to describe our analysis and show that the basic analysis is sound with respect to the semantics of our input language. We assume a basic understanding of pointer analysis. We refer interested readers to [15, 7] for additional exposition.

3.1 Language

For presentation, we use a simple LLVM-like language, whose syntax is shown in Fig. 3.1 and operational semantics in Fig. 3.2. Note that while the language is used to simplify the presentation, our implementation (in Chapter 7) supports full LLVM bitcode.

The semantics are defined using a system of *inference rules*, where premises appear above the bar and consequences below the bar – if all premises are met, the consequences can be derived. We define the program state to be a 6-tuple $\langle V, P, M, T, S, i\rangle$ where:

- V – *value environment* – a map from *value registers* to their values. Value registers can be evaluated under the current value environment, written: $[\![\mathtt{r}]\!]_V = v$, which means that the register $\mathtt{r}$ has a value v under the current value environment V. Value environment V can be modified by assigning a new value v to a value register $\mathtt{r}$, e.g., $V[\mathtt{r} := v]$. Although our grammar does not support constants, we allow registers to be initialized to constant values prior to program execution.

$$
\begin{array}{lcl}
P & ::= & F+ \\
F & ::= & \texttt{fun name(}\overline{\texttt{f}}\texttt{):}\ \ \overline{\texttt{r}}\ \texttt{\{}I+\texttt{\}} \\
I & ::= & \texttt{r = alloc()} \mid \texttt{r = cast } T\texttt{, p} \mid \\
 & & \texttt{r = load } PT\texttt{ p} \mid \texttt{store r, } PT\texttt{ p} \mid \\
 & & \texttt{r = gep } PT\texttt{ p, } \mathit{fld} \mid \overline{\texttt{r}}\texttt{ = callee(}\overline{\texttt{p}}\texttt{)} \mid \texttt{return } \overline{\texttt{z}} \\
T & ::= & BT \cup PT \\
BT & ::= & \texttt{int} \mid \texttt{float} \mid \texttt{char} \\
PT & ::= & BT\texttt{*} \mid BT\texttt{**} \\
\mathit{fld} & ::= & \texttt{a} \mid \texttt{b}
\end{array}
$$

Figure 3.1: A simple language.

- P – *pointer environment* – a map from *pointer registers* to memory objects. Pointer registers are evaluated under the current pointer and memory environments. For example, $[\![\texttt{r}]\!]_{P,M} \mapsto H$ means that the pointer register `r` points to the memory object H under the current pointer environment P and the current memory environment M. Similarly to the value environment, pointer environment P can be modified by making a register `r` point to a memory object H, e.g., $P\,[\texttt{r} \mapsto H]$.

- M – *memory environment* – a set of *memory objects*. A memory object H can be evaluated under the current memory environment M, yielding either a value v or a memory object I, written $[\![H]\!]_M = v$ and $[\![H]\!]_M \mapsto I$, respectively. A fresh, uninitialized memory object H can be added to an existing memory environment M, written $M \cup H$, forming a new memory environment.

- T – *type environment* – a map from *memory objects* to their types. A type environment T can be updated by assigning a type t to a memory object H, written $T\,[H := t]$.

- S – a stack of *function return instructions*. The current instruction on top of the stack S can be obtained with $\mathit{top}(S)$ and removed with $\mathit{pop}(S)$. An instruction i can be added to the top of the stack S with $\mathit{push}(S, i)$.

- i – indicates the next *instruction* to be executed. We denote the successor of an instruction i by writing $i + 1$.

We denote a program state transition using the $V, P, M, T, S, i \hookrightarrow V', P', M', T', S', i'$ notation, where the left-hand-side of $\hookrightarrow$ represents a pre-state and the right-hand-side represents the post-state.

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 a sequence of instructions. This simplified setting is sufficient because our PTA is flow-insensitive – it does not use control-flow information. All registers are of a static (either pointer or value) type. 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). Note that while our language has only two possible fields of constant offset, our implementation supports all LLVM `GetElementPointer` instructions, even when the offset is symbolic, as described in detail in [5]. Each time a `call` instruction ci is executed, ci is pushed onto the stack, and the formal arguments of the callee are assigned with the passed vector of parameters from the caller, s.t. the order and type of arguments is preserved. When the callee executes a `return` instruction, the execution resumes from the successor of the saved ci instruction.

Syntax	Semantics
`r = alloc()`	$\dfrac{i : \texttt{r = alloc()} \qquad H_a, H_b \notin M}{V, P, M, T, S, i \hookrightarrow V, P\left[\texttt{r} \mapsto H_a\right], M \cup H_a \cup H_b, T\left[H_a, H_b := \texttt{char}\right], S, i+1}$
`r = cast T, p`	$\dfrac{i : \texttt{r = cast T, p} \qquad [\![\texttt{p}]\!]_{P,M} \mapsto H}{V, P, M, T, S, i \hookrightarrow V, P\left[\texttt{r} \mapsto H\right], M, T, S, i+1}$ $\dfrac{i : \texttt{r = cast T, p} \qquad [\![\texttt{p}]\!]_{V} = v}{V, P, M, T, S, i \hookrightarrow V\left[\texttt{r} := v\right], P, M, T, S, i+1}$
`r = gep PT p, f`	$\dfrac{i : \texttt{r = gep T p, a} \qquad [\![\texttt{p}]\!]_{P,M} \mapsto H}{V, P, M, T, S, i \hookrightarrow V, P\left[\texttt{r} \mapsto H\right], M, T, S, i+1}$ $\dfrac{i : \texttt{r = gep T p, b} \qquad [\![\texttt{p}]\!]_{P,M} \mapsto H \qquad \mathit{siblingObj}(H) = I}{V, P, M, T, S, i \hookrightarrow V, P\left[\texttt{r} \mapsto I\right], M, T, S, i+1}$
`store r, PT p`	$\dfrac{i : \texttt{store r, T* p} \qquad [\![\texttt{p}]\!]_{P,M} \mapsto H \qquad [\![\texttt{r}]\!]_{P,M} \mapsto I}{V, P, M, T, S, i \hookrightarrow V, P, M\left[H \mapsto I\right], T\left[H := \texttt{T}\right], S, i+1}$ $\dfrac{i : \texttt{store r, T* p} \qquad [\![\texttt{p}]\!]_{P,M} \mapsto H \qquad [\![\texttt{r}]\!]_{V} = v}{V, P, M, T, S, i \hookrightarrow V, P, M\left[H := v\right], T\left[H := \texttt{T}\right], S, i+1}$
`r = load PT p`	$\dfrac{i : \texttt{r = load T* p} \qquad [\![\texttt{p}]\!]_{P,M} \mapsto H \qquad [\![H]\!]_{T} = \texttt{U} \qquad \texttt{T} \sqsubseteq \texttt{U} \qquad [\![H]\!]_{M} \mapsto I}{V, P, M, T, S, i \hookrightarrow V, P\left[\texttt{r} \mapsto I\right], M, T, S, i+1}$ $\dfrac{i : \texttt{r = load T* p} \qquad [\![\texttt{p}]\!]_{P,M} \mapsto H \qquad [\![H]\!]_{T} = \texttt{U} \qquad \texttt{T} \sqsubseteq \texttt{U} \qquad [\![H]\!]_{M} = v}{V, P, M, T, S, i \hookrightarrow V\left[\texttt{r} := v\right], P, M, T, S, i+1}$
$\texttt{fun name(}\overline{\texttt{f}}\texttt{):}\ \ \overline{\texttt{r}}$	$\dfrac{i : \texttt{fun name(}\overline{\texttt{f}}\texttt{):}\ \ \overline{\texttt{r}}}{V, P, M, T, S, i \hookrightarrow V, P, M, T, S, i+1}$
$\overline{\texttt{y}}\ \texttt{= callee(}\overline{\texttt{p}}\texttt{)}$	$\dfrac{\begin{array}{c} i : \overline{\texttt{y}}\ \texttt{= callee(}\overline{\texttt{p}}\texttt{)} \qquad j : \texttt{fun callee(}\overline{\texttt{f}}\texttt{):}\ \ \overline{\texttt{r}} \\ W = \{(k, v) \mid [\![\texttt{p}_k]\!]_V = v\} \qquad Q = \{(k, H) \mid [\![\texttt{p}_k]\!]_{P,M} \mapsto H\} \end{array}}{V, P, M, T, S, i \hookrightarrow V\left[\forall (k, v) \in W \cdot \texttt{f}_k := v\right], P\left[\forall (k, H) \in Q \cdot \texttt{f}_k \mapsto H\right], M, T, \mathit{push}(S, i), j}$
$\texttt{return}\ \overline{\texttt{z}}$	$\dfrac{\begin{array}{c} i : \texttt{return}\ \overline{\texttt{z}} \qquad j = \mathit{top}(S) \qquad j : \overline{\texttt{y}}\ \texttt{= callee(}\overline{\texttt{p}}\texttt{)} \\ W = \{(k, v) \mid [\![\texttt{z}_k]\!]_V = v\} \qquad Q = \{(k, H) \mid [\![\texttt{z}_k]\!]_{P,M} \mapsto H\} \end{array}}{V, P, M, T, S, i \hookrightarrow V\left[\forall (k, v) \in W \cdot \texttt{y}_k := v\right], P\left[\forall (k, H) \in Q \cdot \texttt{y}_k \mapsto H\right], M, T, \mathit{pop}(S), j+1}$

Figure 3.2: Operational semantics of the language in Fig. 3.1.

$$\frac{i:\texttt{r = alloc()}}{\texttt{r} \mapsto H_i}\ \textsc{Alloc} \qquad \frac{\begin{array}{c}\texttt{r = cast PT, p}\\ \texttt{p} \mapsto H\end{array}}{\texttt{r} \mapsto H}\ \textsc{Cast} \qquad \frac{\begin{array}{c}\texttt{r = load PT p}\\ \texttt{p} \mapsto H \quad H \mapsto I\end{array}}{\texttt{r} \mapsto I}\ \textsc{Load} \qquad \frac{\begin{array}{c}\texttt{store r, PT p}\\ \texttt{p} \mapsto I \quad \texttt{r} \mapsto H\end{array}}{I \mapsto H}\ \textsc{Store}$$

Figure 3.3: Inference rules for Inclusion-based PTA: $\mathbb{\Gamma}_{\mathrm{I}}$.

3.2 Pointer Analysis

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. We represent PTAs using inference rules that derive facts of the $\mapsto$ relation. A PTA is computed by applying these rules until saturation. Fig. 3.3 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. 3.3 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. 3.4.

A *unification-based (Steensgaard-style)* PTA is obtained by extending the analysis with additional unification rules $\mathbb{\Gamma}_{\mathrm{U}}$ shown in Fig. 3.5, 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 Chapter 2. 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. It is, however, possible to implement it directly using the presented inference rules in a Datalog framework (see [16]).

3.3 Soundness of the basic PTA rules

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 show that the presented rules $\mathbb{\Gamma}_{\mathrm{I}} \cup \mathbb{\Gamma}_{\mathrm{FLD}}$ are sound with respect to the semantics of our language in Fig. 3.2 when analyzing a single

$$\frac{\texttt{r = gep PT p, a} \quad \texttt{p} \mapsto H}{\texttt{r} \mapsto H}\ \text{GEP} \qquad \frac{\begin{array}{c}\texttt{r = gep PT p, b} \quad \texttt{p} \mapsto H \\ \mathit{fld}(H) = \texttt{a} \quad \mathit{siblingObj}(H) = I\end{array}}{\texttt{r} \mapsto I}\ \text{GEP}$$

Figure 3.4: Inference rules for Field-Sensitivity: $\mathbb{\Gamma}_{\textsc{Fld}}$.

$$\frac{\begin{array}{r}\texttt{r} \mapsto H \quad \texttt{r} \mapsto I \\ \texttt{p} \mapsto I\end{array}}{\texttt{p} \mapsto H}\ \textsc{Incoming} \qquad \frac{\begin{array}{r}H \mapsto I \quad H \mapsto J \\ L \mapsto J\end{array}}{L \mapsto I}\ \textsc{Incoming}$$

$$\frac{\begin{array}{c}\texttt{r} \mapsto H \quad H \mapsto J \\ \texttt{r} \mapsto I \quad I \mapsto K\end{array}}{H \mapsto K}\ \textsc{Outgoing} \qquad \frac{\begin{array}{c}H \mapsto I \quad I \mapsto K \\ H \mapsto J \quad J \mapsto L\end{array}}{I \mapsto L}\ \textsc{Outgoing}$$

Figure 3.5: Unification rules: $\mathbb{\Gamma}_{\textsc{U}}$.

function, and in turn so is Γ_{STEENS}. That is, we show that in every concrete execution of a program P, if p points to co, then $\Gamma_{\text{I}} \cup \Gamma_{\text{FLD}} \vdash_P p \mapsto o$, and whenever co_1 points to co_2 then $\Gamma_{\text{I}} \cup \Gamma_{\text{FLD}} \vdash_P o_1 \mapsto o_2$, where o, o_1, and o_3 are the corresponding abstract objects for co, co_1, and co_2, respectively.

Initially, when no instructions of P are yet executed, the concrete program state contains no memory objects, thus any set of $\mapsto$ facts computed by a PTA matches the concrete state. Then, for every subsequent instruction executed that introduces or modifies the concrete $\mapsto$ facts, there is a matching inference rule in $\Gamma_{\text{I}} \cup \Gamma_{\text{FLD}}$ that can derive it. Suppose that a sequence of instructions was executed and that the currently calculated set of $\mapsto$ facts by the PTA soundly overapproximates the concrete program state. When the next instruction i is executed, some of its operands may be pointer registers that were results of the previous instructions. Thus, if a pointer operand `p` points to some memory object, the corresponding $\mapsto$ fact must already be in the PTA results for $i - 1$. Because all instructions that modify the pointer environment P or the memory environment M have a corresponding inference rule in $\Gamma_{\text{I}} \cup \Gamma_{\text{FLD}}$ that matches the operation semantics, the PTA will compute sound results for i as well. This is because an appropriate inference rule will be eventually applied, even if the order is unspecified, as the derivation is exhaustive.

Chapter 4

Reducing Oversharing in DSA

This chapter is organized as follows: first, we describe how to extend the $\Gamma_{\textsc{Steens}}$ PTA to be *interprocedural* and explain *(calling) context-sensitivity*. Next, we show how to extend $\Gamma_{\textsc{Steens}}$ 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.

4.1 Context-sensitivity

The unification-based PTA $\Gamma_{\textsc{Steens}}$ from Chapter 3 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. Indirect calls can be supported by using PTA to identify potential callees. Recursion is supported by treating all strongly connected components in the call-graph intraprocedurally.

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. 2.1(b). A context-insensitive unification-based analysis is obtained by extending $\Gamma_{\textsc{Steens}}$ with rules for interprocedural assignments in Fig. 4.1.

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

$$\frac{i : \overline{\mathtt{y}} \texttt{ = callee(}\overline{\mathtt{x}}\texttt{)} \quad j : \texttt{return } \overline{\mathtt{z}} \quad \mathit{fun}(j) = \texttt{callee} \quad \mathtt{z}_k \mapsto H}{\mathtt{y}_k \mapsto H}\ \textsc{CI-Bottom-Up} \qquad \frac{i : \overline{\mathtt{y}} \texttt{ = callee(}\overline{\mathtt{x}}\texttt{)} \quad \mathtt{x}_k \mapsto H \quad j : \texttt{fun callee(}\overline{\mathtt{f}}\texttt{):} \quad \overline{\mathtt{r}}}{\mathtt{f}_k \mapsto H}\ \textsc{CI-Top-Down}$$

Figure 4.1: Inference rules for Context-insensitivity: $\mathbb{\Gamma}_{\mathrm{CI}}$.

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

Figure 4.2: Inference rules for formal arguments: $\mathbb{\Gamma}_{\textsc{Formals}}$.

$\overset{F}{\mapsto}$ relation for each function F in the analyzed program. DSA is an example of such an analysis [11]. Although not formally specified in [11], it is defined by adding rules to $\mathbb{\Gamma}_{\textsc{Steens}}$, $\mathbb{\Gamma}_{\mathrm{DSA}} = \mathbb{\Gamma}_{\mathrm{L}} \cup \mathbb{\Gamma}_{\mathrm{BU}} \cup \mathbb{\Gamma}_{\mathrm{TD}}$, where $\mathbb{\Gamma}_{\mathrm{L}} = \mathbb{\Gamma}_{\textsc{Steens}} \cup \mathbb{\Gamma}_{\textsc{Formals}}$. Soundness of $\mathbb{\Gamma}_{\mathrm{DSA}}$ in an interprocedural context is ensured by extending the basic rules $\mathbb{\Gamma}_{\mathrm{I}} \cup \mathbb{\Gamma}_{\mathrm{FLD}}$ with rules that handle interprocedural pointer assignments from callees to callers and callers to callees, at call- and return-sites, closely following our language semantics for function calls.

4.2 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. 4.2 shows inference rules $\mathbb{\Gamma}_{\textsc{Formals}}$ that specify how these abstract object may point to each other. The rules model precisely only two levels of indirection. Any memory object obtained by

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

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

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

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

Figure 4.3: Inference rules for Context-Sensitivity: $\mathbb{\Gamma}_{\mathrm{BU}}$ and $\mathbb{\Gamma}_{\mathrm{TD}}$.

a further dereference is mapped to the same second-level formal, adding a cycle. In practice, precision of analysis can be improved by computing the necessary levels of indirection (e.g., [22, 11, 5]), or creating formals on demand, like in our implementation.

4.3 Oversharing

While the local analysis $\mathbb{\Gamma}_{\mathrm{L}}$ only uses abstract objects from the analyzed function (i.e., coming from allocation sites in that function or its formals), the rules $\mathbb{\Gamma}_{\mathrm{BU}} \cup \mathbb{\Gamma}_{\mathrm{TD}}$, shown in Fig. 4.3, propagate $\mapsto$ facts across functions. They use a helper function *Resolve* from Fig. 4.4 to map between callee and caller 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 $\xmapsto{F_1}$ 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 [11], executes three phases of the analysis for a function F as

$$\frac{\begin{array}{c} i : \texttt{return}\ \overline{\texttt{z}} \qquad \mathit{fun}(i) = \texttt{callee} \\ \texttt{z}_k \xmapsto{\texttt{callee}} H \end{array}}{\mathit{Accessible}(\texttt{callee}, H)} \qquad \frac{\begin{array}{c} i : \texttt{fun callee(}\overline{\texttt{f}}\texttt{):}\ \ \overline{\texttt{r}} \\ \texttt{f}_k \xmapsto{\texttt{callee}} H \end{array}}{\mathit{Accessible}(\texttt{callee}, H)} \qquad \frac{\mathit{Accessible}(\texttt{callee}, H) \qquad H \xmapsto{\texttt{callee}} I}{\mathit{Accessible}(\texttt{callee}, I)}$$

$$\frac{\begin{array}{c} i : \overline{\texttt{y}}\ \texttt{=}\ \texttt{callee(}\overline{\texttt{x}}\texttt{)} \qquad \mathit{fun}(i) = \texttt{caller} \\ j : \texttt{fun callee(}\overline{\texttt{f}}\texttt{):}\ \ \overline{\texttt{r}} \qquad \texttt{p} \xmapsto{\texttt{callee}} H \\ \mathit{isAllocSite}(H) \end{array}}{\mathit{Resolve}(i, H, H)} \qquad \frac{\begin{array}{c} i : \overline{\texttt{y}}\ \texttt{=}\ \texttt{callee(}\overline{\texttt{x}}\texttt{)} \qquad \mathit{fun}(i) = \texttt{caller} \\ \texttt{x}_k \xmapsto{\texttt{caller}} H \qquad j : \texttt{fun callee(}\overline{\texttt{f}}\texttt{):}\ \ \overline{\texttt{r}} \end{array}}{\mathit{Resolve}(i, V^{\texttt{a}}_{j,k}, H)}$$

$$\frac{\begin{array}{c} i : \overline{\texttt{y}}\ \texttt{=}\ \texttt{callee(}\overline{\texttt{x}}\texttt{)} \qquad \mathit{fun}(i) = \texttt{caller} \\ H \xmapsto{\texttt{caller}} I \qquad \mathit{Resolve}(i, V^{f}_{j,k}, H) \end{array}}{\mathit{Resolve}(i, V^{f\texttt{a}}_{j,k}, I)} \qquad \frac{\begin{array}{c} i : \overline{\texttt{y}}\ \texttt{=}\ \texttt{callee(}\overline{\texttt{x}}\texttt{)} \qquad \mathit{fun}(i) = \texttt{caller} \\ H \xmapsto{\texttt{caller}} I \qquad \mathit{Resolve}(i, V^{fg}_{j,k}, H) \end{array}}{\mathit{Resolve}(i, V^{fg}_{j,k}, I)}$$

$$\frac{\begin{array}{c} \mathit{Resolve}(i, V^{\texttt{a}}_{j,k}, H) \\ \mathit{fld}(H) = \texttt{a} \qquad \mathit{siblingObj}(H) = I \end{array}}{\mathit{Resolve}(i, V^{\texttt{b}}_{j,k}, I)} \qquad \frac{\begin{array}{c} \mathit{Resolve}(i, V^{f\texttt{a}}_{j,k}, H) \\ \mathit{fld}(H) = \texttt{a} \qquad \mathit{siblingObj}(H) = I \end{array}}{\mathit{Resolve}(i, V^{f\texttt{b}}_{j,k}, I)}$$

Figure 4.4: Helper inference rules for Context-Sensitivity.

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 $\mathbb{\Gamma}_{\mathrm{L}}$ and $\mathbb{\Gamma}_{\mathrm{BU}}$ rules until saturation in a reverse-topological call-graph order, followed by $\mathbb{\Gamma}_{\mathrm{TD}}$ in a topological order until saturation. The rules can be soundly applied in this sequence and no new $\overset{F}{\mapsto}$ 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 Chapter 2, is a major source of oversharing in DSA. This form of oversharing led to a workaround in [11] 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 $(\mathbb{\Gamma}_{\mathrm{DSA}} \vdash_P x \overset{F}{\mapsto} H) \implies \exists y \cdot (\mathbb{\Gamma}_{\mathrm{L}} \cup \mathbb{\Gamma}_{\mathrm{BU}} \vdash_P y \overset{F}{\mapsto} H)$*, where x and y are registers or abstract objects.*

Theorem 1 states that no new foreign objects are ever introduced by $\mathbb{\Gamma}_{\mathrm{TD}}$. The derivable $\overset{F}{\mapsto}$ facts are always over abstract objects *resolved* from caller's abstract object to callee's abstract objects. The proof of Theorem 1 follows from the fact that $\mathbb{\Gamma}_{\mathrm{L}}$ models the operational semantics of our language (see Chapter 3), and that our formulation of interprocedural rules explicitly uses the callee-caller resolution of abstract objects. By contradiction, suppose that a new $x \overset{F}{\mapsto} H$ fact is derived during $\mathbb{\Gamma}_{\mathrm{TD}}$, s.t. H did not appear in analysis result after running $\mathbb{\Gamma}_{\mathrm{L}} \cup \mathbb{\Gamma}_{\mathrm{BU}}$ until saturation. For this to be the case there must be an inference rule that can derive facts about new abstract objects. $\mathbb{\Gamma}_{\mathrm{TD}}$, together with its helper rules for *Resolve*, only derive $\overset{F}{\mapsto}$ facts about (existing) abstract objects in callees (Top-Down-1) and formals in these callees (Top-Down-2), thus we reach a contradiction. □

The simplicity of Theorem 1 is solely due to our new formulation of DSA. Prior works ([11, 12]) miss this, now obvious, fact. With our formulation, it is clear that the role of Top-Down is to use $\overset{F}{\mapsto}$ at a call-site and use it to instantiate a fully-general summary for a callee by introducing necessary $\overset{F}{\mapsto}$ between function arguments and formals. If a client of a PTA requires to know not only $\overset{F}{\mapsto}$ 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, like in our implementation. Our evaluation (Chapter 7) demonstrates that this improves performance and precision.

4.4 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 Chapter 2. Due to the Γ_{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. 2.1(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 $\xmapsto{F@i}$ 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. 2.1(a), `str1` $\xmapsto{\texttt{getStr@11}} o_1$ at the return statement. We call this refinement *partial flow-sensitivity*. We present a set of rules, Γ_{PFS} in Fig. 4.5, that combine together with Γ_{DSA} to define an analysis called $\Gamma_{\text{PFS-DSA}}$. Note that Γ_{PFS} *replaces* the corresponding two rules from Γ_{DSA}. We assume that $\xmapsto{F@i}$ is externally defined and is a (sound) subset of $\xmapsto{F}$. BOTTOM-UP-1 rule of Γ_{PFS} propagates $\xmapsto{\texttt{callee}@j}$ (points-to information at the return-site) into $\xmapsto{\texttt{caller}}$, 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 $\xmapsto{F}$ 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 $p \xmapsto{F} o$, $p \not\xmapsto{F@i} o$. For instance, `str1` $\not\xmapsto{\texttt{getStr@11}} o_2$ 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.

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

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

Figure 4.5: Inference rules for Partial Flow-Sensitivity: Γ_{PFS}.

The only difference between $\Gamma_{\text{PDF-DSA}}$ and Γ_{DSA} is the use of the $\xmapsto{F@i}$ relation instead of $\overset{F}{\mapsto}$ in Bottom-Up and Top-Down rules, where $\xmapsto{F@i}$ is a subset of $\overset{F}{\mapsto}$. Assuming $\xmapsto{F@i}$ is sound at a call-site (return-site), every $\overset{F}{\mapsto}$ fact is correctly propagated by the interprocedural assignment rules.

Chapter 5

Extending DSA with Type-awareness

5.1 Strict Aliasing

The *effective type rules* of the C11 standard [9] 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 [9, Sec. 6.5 p. 6-7]. Other languages, including C++ and SWIFT, impose similar rules. Thus, pointers of incompatible types do not alias, and the result of a `load` instruction can only be affected by `store` instructions of compatible type. These rules are usually referred to as *strict aliasing* and are widely exploited in all major optimizing compilers. In this thesis, we use them to improve precision of the LOCAL phase of TEADSA.

5.2 Type Compatibility

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

$$\forall \tau \in T \cdot \tau \sqsubseteq \texttt{char} \qquad\qquad \forall \tau \in PT \cdot \tau \sqsubseteq \texttt{char*}$$

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).

5.3 Type-aware DSA

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 $\mathtt{r} \overset{T}{\mapsto} H$ 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. 2.3 is shown in Fig. 2.4.

Although the type of each register is known statically, we only require memory operation (`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 $\mathbb{\Gamma}_{\text{TY}}$ in Fig. 5.1. 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 $\xmapsto{\mathtt{char}}$ 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 abstract object may be written to. For example, consider a simple 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. Note that this technique does not require flow-sensitivity as we do not perform type inference and only ignore stores that definitely do not affect loads of incompatible types.

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

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

Theorem 2 says that $\mathbb{\Gamma}_{\text{TeaDsa}}$ *is not less* precise than $\mathbb{\Gamma}_{\text{PFS-DSA}}$, i.e., no points-to relation not present in analysis results for $\mathbb{\Gamma}_{\text{PFS-DSA}}$ is present in analysis results for $\mathbb{\Gamma}_{\text{TeaDsa}}$. This is because the type-aware rules for `load` and `store` are similar to $\mathbb{\Gamma}_{\text{I}}$, except that they

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

Figure 5.1: Type-awareness rules: $\mathbb{\Gamma}_{\textsc{Ty}}$.

prevent loads from deriving facts about stores of incompatible types. The addition of type compatibility checks in $\mathbb{\Gamma}_{\textsc{Ty}}$ is an extra premise (filter), and thus can only prevent some derivations, compared to $\mathbb{\Gamma}_{\textsc{I}}$. With the most conservative compatibility relation: $\forall \tau, \upsilon \in T \cdot \tau \sqsubseteq \upsilon$ (i.e., all types are compatible), the $\mathbb{\Gamma}_{\textsc{Ty}}$ would degrade to $\mathbb{\Gamma}_{\textsc{I}}$ and derive exactly the same $\mapsto$ facts. □

Chapter 6

Implementation

In this chapter, we describe our implementation of TeaDsa.

6.1 SeaDsa and Key Data-Structures

We implemented TeaDsa on top of SeaDsa – a context-, field-, and array-sensitive DSA-style PTA for LLVM [5]. 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; 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.

SeaDsa, just like DSA, maintains a separate *DS graph* (a DSA-sepcific name for a points-to graph) for each analyzed function. A group of abstract objects is represented by a *node*, s.t. all sibling objects are represented as different fields of the same node. A node maintains its point-to set, and each field in a node contains outgoing links, represented with *cells* – offset-identified fields within other nodes. One can think of a cell as an abstract pointer, as it is sufficient to identify both a pointee and an offset within it. Our implementation supports all `GetElementPointer` LLVM instructions. Constant offsets are handled in a precise field-sensitive manner, symbolic array offsets handled in an array-sensitive manner, while non-array symbolic offsets are handled soundly but not precisely. When a precise field cannot be determined for a memory access, or when unifying certain kinds of nodes together, a node may get *collapsed* and lose (its) field-sensitivity. Note that nodes and cells belong to exactly one graph at a time. To maintain efficiency of the data structure, links are only forward-traversable. A more detailed explanation of the DSA data structures is presented in the SeaDsa paper [5].

6.2 Top-Down Optimizations in TeaDsa

The Top-Down phase in SeaDsa processes a function at a time in a topological call-graph order. A graph of a caller is *cloned* (inlined) into each of its callees. Cloning starts by creating a new copy of each node reachable from the caller's parameters and globals, together with its links, inside the callee's graph. Next, interprocedural assignment from caller's parameters to callee's formals, including all global variables in the caller, is performed over the freshly cloned nodes. As a result some of the nodes corresponding to formals, global variables, and nodes reachable from them may get grouped (unified). This way of performing cloning causes Top-Down to be worst-case quadratic in the graph size of Bottom-Up and, in our experience, renders it not scalable for large programs, such as LLVM and Rippled.

In TeaDsa, we leverage Theorem 1 by performing cloning in a more efficient way. We know that a callee may perform only as many indirections of formals and globals as discovered during the Local and Bottom-Up analysis, and it is not necessary to consider caller's nodes above that, as they are inaccessible in the callee. For simplicity of implementation, we mark each to-be-cloned caller's node as *foreign*, and remove any foreign node that was not unified with a callee's node by the end of cloning. Even though the whole part of the caller's graph reachable from formals and globals is initially cloned, the pruning step ensures that inaccessible nodes are not overshared, and makes Top-Down linear in the size of Bottom-Up graph.

6.3 Partial Flow-sensitivity in TeaDsa

We perform two partial flow-sensitivity optimizations to increase precision of our implementation. First, we do not propagate stack-allocated abstract objects during cloning in Bottom-Up. This is because the LLVM language semantics, similar to most imperative languages, specify that memory created with an `alloca` stack allocation instruction is only valid when the function containing that instruction is being executed. Thus, callee's stack-allocated memory is not available in its callers, and it is not necessary to propagate nodes with only `alloca`s in their points-to sets bottom-up.

Second, we disambiguate pointers that *must alias* known allocation sites from other objects in their points-to sets. Because LLVM uses the *Static Single Assignment (SSA)* form, we can determine whether a pointer must point to a known allocation site by following its use-def chain until we reach an allocation site or an instruction that does not allow us to determine a precise allocation site. For each interprocedural assignment with a known allocation site, we clone the associated node with only this allocation site in its points-to set. We do not copy the other allocation sites if we can prove the cloned node is not reachable from some other pointer in the cloned-to graph. Determining so is however nontrivial, as the data-structures used in our implementation do not allow for checking what the incoming cells to a node are. Instead, we start cloning the source graph by processing all nodes corresponding to interprocedurally-assigned registers, and check whether some cloned cell reaches these nodes. If that happens to be the case, we copy all the other allocation sites from the source node into the node's points-to set and unify it with other nodes that share any of the same allocation sites.

A more sophisticated implementation is possible with a similar technique to the one described above for nodes that may alias a larger number of allocation sites. A more precise auxiliary flow-sensitive PTA could be used to filter points-to sets (e.g., LLVM's BasicAA or TBAA), as long as the notion of allocation sites that the auxiliary PTA uses is compatible with the one in TeaDsa.

6.4 Type-awareness in TeaDsa with LLVM Type Tags

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. If the sequence of nested types forms a cycle, we choose the least of these types as a representative. Two types are compatible if they have the same type tag. We use the LLVM `Type` pointers for type tags, as the in-memory bitcode representation guarantees each type object to have a single instance for the whole analyzed program.

To support typed abstract objects, we define a field of a node in terms of its offset and type tag, but leave the definition of a cell unchanged. This follows directly from the fact that in LLVM, just like in our language in Chapter 3, it is the memory that is typed, and not pointers, i.e., the content of the memory affected by a memory instruction depends on the accessed type.

Chapter 7

Evaluation

In this chapter, we compare TeaDsa, 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/tree/tea-dsa`.

7.1 Program Verification Task

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. 2.3 – 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 [4], and make it available at:
`https://github.com/seahorn/seahorn/tree/tea-dsa`.

The checker starts by running a PTA on a single bitcode file for the whole program. Next, for each memory instruction i and a pointer operand p, the PTA is used to identify all allocation sites that are in p's points-to set A. Based on the size of accessed memory region and the offsets applied, we determine what is the minimum required size s of a memory object o such that i is not accessing o outside of its bounds. Calculating s is only easy for some pointers that are created by constant offsets manipulations; we do not check for unsafe memory accesses with variable (symbolic) offsets. We then partition A into three disjoint sets: allocation sites of variable size V; allocation sites of sufficient, statically-known size $\geq s$, S; and allocation sites of insufficient, statically-known size $< s$, I. Next, for each $a \in I$ we check whether there is an execution s.t. the pointer p used in i comes from a and i gets executed.

In order to perform a single safety check for a memory instruction i and an allocation site $a \in I$, we instrument a program as follows:

- Add two new global variables G and initialize it with `false`.
- Partition memory into address spaces, X and Y.
- Instrument all allocation sites $b \in A \setminus \{a\}$ by adding assumptions: $\mathit{assume}(b \notin Y)$.
- Instrument a such that when G is `false` it nondeterministically returns a fresh memory object o from either X or Y, and add appropriate assumptions. If $o \in Y$, then G is set to `true`.
- Just before i, add an assertion: $\mathit{assert}(G \implies \texttt{p} \notin Y)$.

If there is no execution that violates the assertion in a program instrumented this way, we conclude that there are no field-overflow bugs for that pair of i and a, i.e., no pointers originating from a are used (unsafely) in i.

In the evaluation, we do not emit the described instrumentation, and only use the number of checks that this technique would generate as a proxy for the total verification time of the whole program.

7.2 Experimental Evaluation

We compare TeaDsa with two state-of-the-art interprocedural PTAs for LLVM: SVF [21] and SeaDsa [5]. SVF [21] 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 $\mathbb{F}_{\text{PFS-DSA}}$, and our implementation of $\mathbb{F}_{\text{TeaDsa}}$, 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 in Table 7.1 and precision on the field-overflow detection in Table 7.2. 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. For the performance comparison in Table 7.1, we only measure the time taken to load bitcode and run a PTA on it. In Table 7.2, 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 and assumptions 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).

Program	Bitcode Size [kB]	Results									
		Wave Diff		SVF Sparse		SeaDsa		PFS-SeaDsa		TeaDsa	
		Run-time [s]	Memory [MB]	Run-time [s]	Memory [MB]	Run-time [s]	Memory [MB]	Run-time [s]	Memory [MB]	Run-time [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 7.1: 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 7.2: Precision of different PTAs.

Chapter 8

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 thesis, we compare with the closest related work: DSA [11] and SeaDsa [5]. In Chapter 7, we compared empirically with two context-insensitive, inclusion-based implementations of SVF [21] – a state-of-the-art PTA framework for LLVM. In the rest of this chapter, we compare with other related works.

Sui et al. [22] 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. [6, 20]). 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 [2] extends a whole-program, inclusion-based PTA with types. The analysis is object and type-sensitive ([17]). 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 [13] 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.

Chapter 9

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 about memory accesses. 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.

References

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[2] George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for C and C++. In Xavier Rival, editor, *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, volume 9837 of *Lecture Notes in Computer Science*, pages 84–104. Springer, 2016.

[3] Dirk Beyer. Automatic verification of C and java programs: SV-COMP 2019. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, volume 11429 of *Lecture Notes in Computer Science*, pages 133–155. Springer, 2019.

[4] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015.

[5] Arie Gurfinkel and Jorge A. Navas. A context-sensitive memory model for verification of C/C++ programs. In Francesco Ranzato, editor, *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, volume 10422 of *Lecture Notes in Computer Science*, pages 148–168. Springer, 2017.

[6] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the CGO 2011, The 9th International Symposium on Code*

Generation and Optimization, Chamonix, France, April 2-6, 2011, pages 289–298, 2011.

[7] Michael Hind. Pointer analysis: haven't we solved this problem yet? In John Field and Gregor Snelting, editors, *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, pages 54–61. ACM, 2001.

[8] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In Debra J. Richardson and Mary Jean Harold, editors, *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2000, Portland, OR, USA, August 21-24, 2000*, pages 113–123. ACM, 2000.

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

[10] Jakub Kuderski, Jorge A. Navas, and Arie Gurfinkel. Unification-based pointer analysis without oversharing. In *FMCAD*, 2019.

[11] Chris Lattner and Vikram S. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 129–142. ACM, 2005.

[12] Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani. A framework for efficient modular heap analysis. *Foundations and Trends in Programming Languages*, 1(4):269–381, 2015.

[13] Zvonimir Rakamaric and Alan J. Hu. A scalable memory model for low-level code. In Neil D. Jones and Markus Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*, pages 290–304. Springer, 2009.

[14] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 693–706, 2018.

[15] Yannis Smaragdakis, George Balatsouras, et al. Pointer analysis. *Foundations and Trends® in Programming Languages*, 2(1):1–69, 2015.

[16] Yannis Smaragdakis and Martin Bravenboer. Using Datalog for Fast and Easy Program Analysis. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers, editors, *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*, pages 245–251. Springer, 2010.

[17] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM.

[18] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 22:1–22:26. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[19] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[20] Yulei Sui, Peng Di, and Jingling Xue. Sparse flow-sensitive pointer analysis for multithreaded programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, pages 160–170, 2016.

[21] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 265–266. ACM, 2016.

[22] Yulei Sui, Sen Ye, Jingling Xue, and Jie Zhang. Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation. *Softw., Pract. Exper.*, 44(12):1485–1510, 2014.

[23] Tian Tan. *Precise and Efficient Points-to Analysis via New Context-Sensitivity and Heap Abstraction.* PhD thesis, University of New South Wales, Sydney, Australia, 2017.

[24] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 327–337. ACM, 2018.