Unification-based Pointer Analysis without Oversharing

Jakub Kuderski[1,3], Jorge A. Navas[2], Arie Gurfinkel[1]

[1] University of Waterloo, Canada
[2] SRI International, USA
[3] Currently Google Canada

FMCAD 2019, San Jose, CA, USA, October 23 2019

TeaDsa -- a new Pointer Analysis for LLVM

A state-of-the-art PTA for LLVM, based on SeaDsa

- Unification-based (Steensgaard-style);
- Context-, field-, and array-sensitive.

Contributions:

1. A modular formulation of DSA;
2. Elimination of abstract object copying in the Top-Down phase of DSA;
3. Improved inter-procedural reasoning with partial flow-sensitivity;
4. Improved intra-procedural reasoning with type-awareness.

Evaluation based on a program verification task: detecting field-overflow bugs.

Outline

1. Verification Challenges for Low-level Programs
2. Pointer Analysis
3. Oversharing in Existing Unification-based Pointer Analyses
4. Analyzing Pointer Analyses
5. TeaDsa -- a Scalable Context-Sensitive Pointer Analysis for LLVM
6. Evaluation and Conclusions

Pointers in Low-level Languages

```c
struct Data {
  float f;
  int i;
  struct Data* next;
};

int main() {
  int x;
  int *px = &x;
  *px = 42;

  struct Data *data = malloc(sizeof(struct Data));
  px = &data->i;
  scanf("%d", px);

  data->next = data;

```

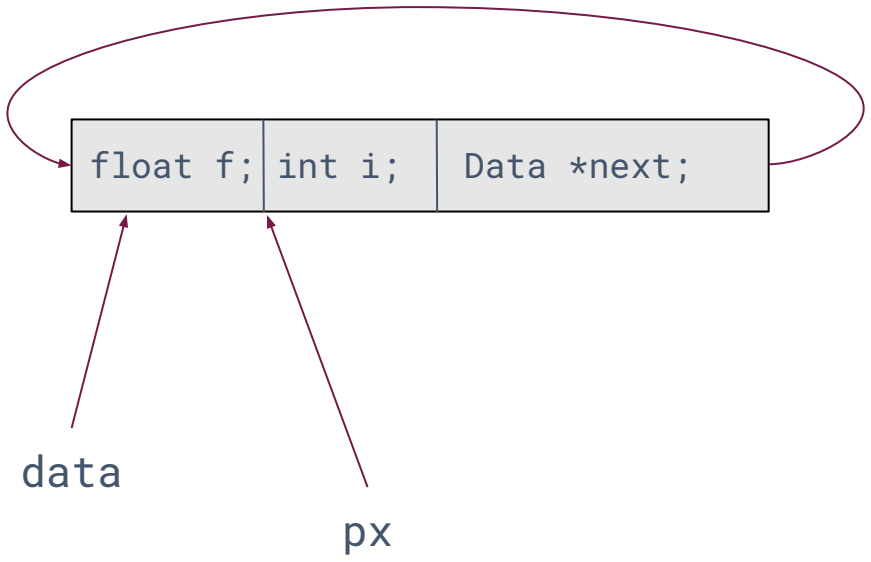
- Used for strings, arrays, passing function parameters, return values.
- Pointers to fields of aggregates (e.g., structs, arrays).
- Pointer arithmetic, integer-to-pointer conversions, type casts.

Pointers in Low-level Languages

```c
struct Data {
  float f;
  int i;
  struct Data* next;
};

int main() {
  int x;
  int *px = &x;
  *px = 42;

  struct Data *data = malloc(sizeof(struct Data));
  px = &data->i;
  scanf("%d", px);

  data->next = data;

```

Definition:

Pointer -- object identifier and offset within that object.



Pointers in Program Verification

```
1   const char *str1 = "Str1";
2   const char *str2 = "Str2";
3   const char *str3 = "Str3";
4
5   void print(const char *x) {}
6
7   const char *getStr() {
8     const char *p = nondet() ? str1 : str2;
9     print(p);
10    return str1;
11  }
12
13  struct Config {
14    const char *label;
15    int *val;
16  };
17
18  int foo(struct Config *conf) {
19    const char str4[5] = "Str4";
20    print(str4);
21    const char *r = getStr();
22    print(r);
23    return conf->label == r;
24  }
25
```

- What strings can line 9 print?
- What is the result of the comparison on line 23?
- Can `foo` overwrite the `label` field of `conf`?
- Is accessing the `label` field of `conf` safe in `foo`?

Pointer Analysis (PTA)

Pointer Analysis (PTA) -- determining whether a given pointer:

a. aliases with another pointer (alias analysis)

   alias(p1, p2)

b. points to an object (points-to analysis)

   $p \longmapsto o$

- Indispensable in reasoning about programs:
  - Static Program Analysis, Program Verification, Compiler Optimizations.
- Undecidable -- we need approximate solutions.
- Numerous publications about Pointer Analysis, yet very few quality open-source implementations for LLVM:
  - e.g., DSA, SeaDsa, SVF.

Inclusion- and Unification-based PTAs

```
1   int *ptr;                                  // o1
2   ptr = (int *) malloc(sizeof(int *));       // o2
3   int **ptr_ptr = NULL;
4
5   if (nondet()) {
6     MyClass *classPtr = new MyClass();       // o3
7     ptr_ptr = (int *) classPtr;
8   }
9   else {
10    ptr_ptr = &ptr;
11    **ptr_ptr = 42;
12  }
13
```

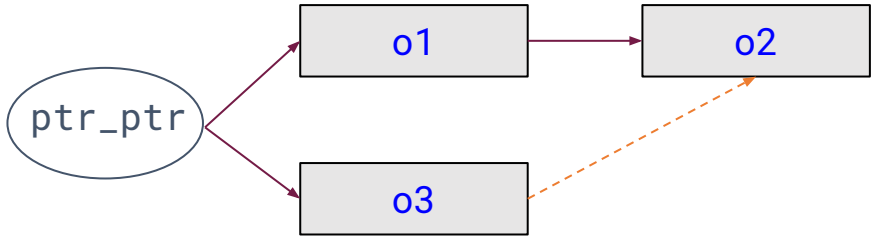
Definitions:
Objects distinguished by their *Allocation Site*, e.g., calls to allocating functions, declarations of address-taken variables.

Soundnes:
If a PTA says that two pointers do not alias, there must be no program execution where they point to the same object.

Inclusion-based

(Andersen-style):

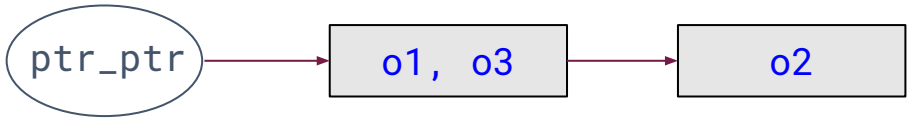
- e.g., SVF



Unification-based

(Steensgaard-style)

- e.g., DSA, SeaDsa

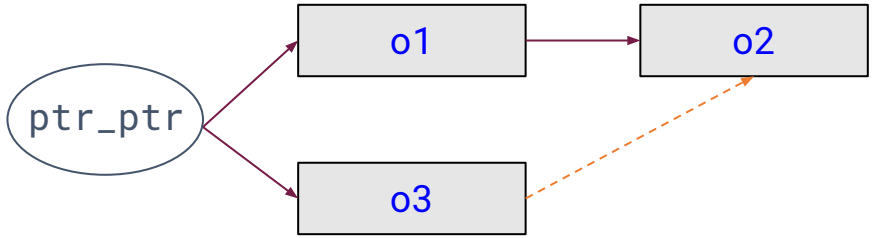


Inclusion and Unification Constraints

Inclusion-based

(Andersen-style):

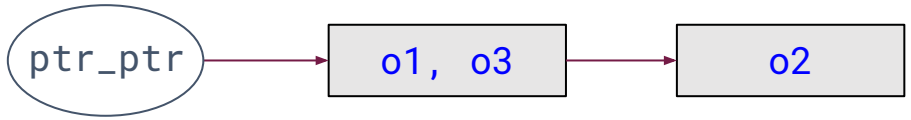
- e.g., SVF.



Unification-based

(Steensgaard-style)

- e.g., DSA, SeaDsa.



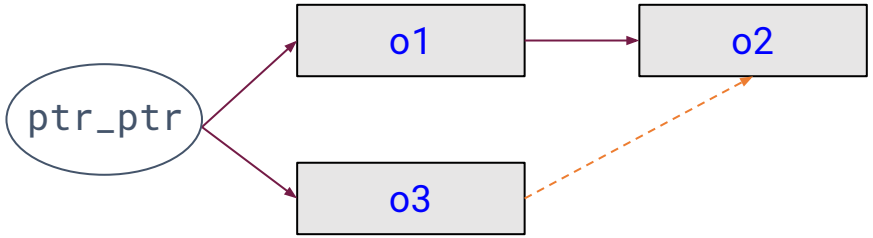
Instruction	Inclusion (subset) constraint	Unification constraint
p = malloc(n)	$p \supseteq loc(malloc)$	$p \approx loc(malloc)$
p = q	$p \supseteq q$	$p \approx q$
**p = q*	$pts(p) \supseteq q$	$pts(p) \approx q$
*p = *q*	$p \supseteq pts(q)$	$p \approx pts(q)$
p = &x	$p \supseteq loc(x)$	$p \approx loc(x)$

Conventional Wisdom

Inclusion-based

(Andersen-style):

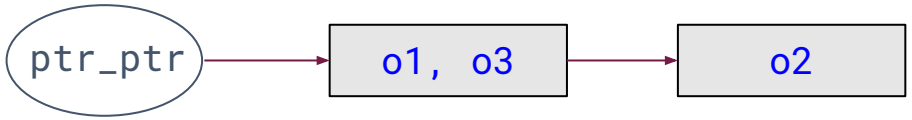
- e.g., SVF.



Unification-based

(Steensgaard-style)

- e.g., DSA, SeaDsa.



Property	Inclusion-based	Unification-based
Precision?	Precise ✓	Imprecise ✗
Speed?	Slow ✗	Fast ✓
Memory consumption?	Large ✗	Small ✓
Patent issues?	No ✗	Yes ✓

Definition:

Precision -- roughly, the fewer points-to facts a PTA derives the more precise it is.

Dimensions of PTAs

1. Flow-sensitivity -- separate results for each program instruction. (e.g., SVF)
2. Field-sensitivity -- distinguishing fields of aggregates. (e.g., SVF, SeaDsa)
3. Context-sensitivity -- distinguishing different calling contexts. (e.g., SeaDsa)
4. More...

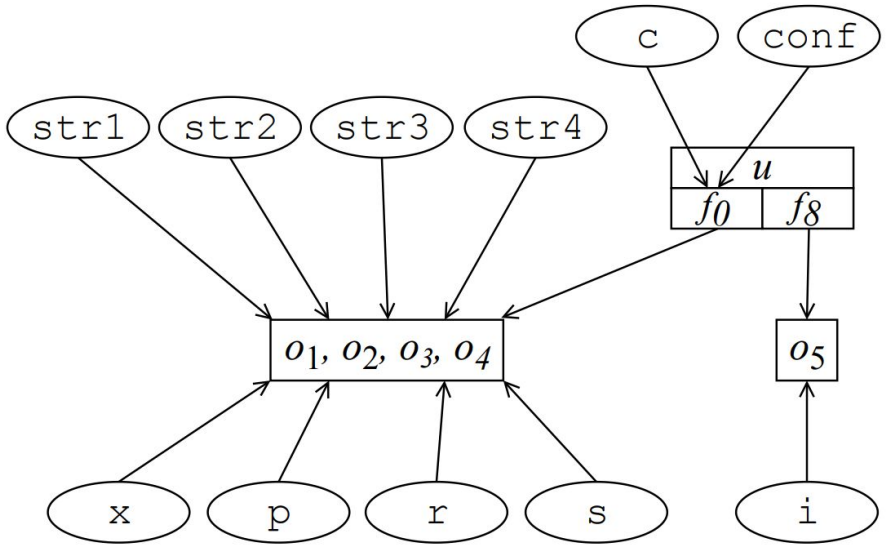
Inclusion-based PTAs are typically flow-sensitive but context-insensitive.

Unification-based PTAs are typically context-sensitive but flow-insensitive.

Unification-based PTA -- an example

```
1  const char *str1 = "Str1";                // o1
2  const char *str2 = "Str2";                // o2
3  const char *str3 = "Str3";                // o3
4
3  void print(const char *x) {}
6
7  const char *getStr() {
1    const char *p = nondet() ? str1 : str2;
9    print(p);
10   return str1;
11 }
12
13 struct Config {
14   const char *label;
15   int *val;
16 };
17
18 int foo(struct Config *conf) {
19   const char str4[5] = "Str4";            // o4
20   print(str4);
21   const char *r = getStr();
22   print(r);
23   return conf->label == r;
24 }
25
26 int bar() {
27   int i = 42;                             // o5
2    const char *s = nondet() ? str2 : str3;
29   struct Config c = {s, &i}               // u
30   return foo(&c);
31 }
32
```

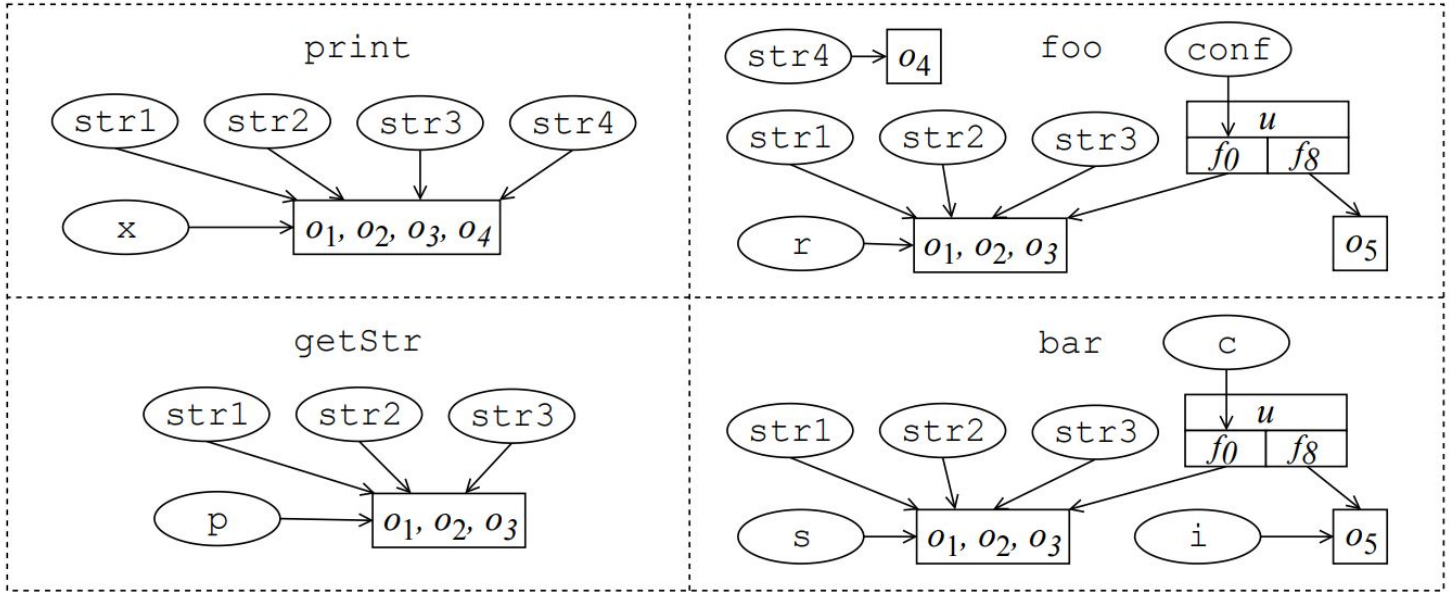
A Context-insensitive Points-To Graph:



Unification-based PTA -- an example

```
1   const char *str1 = "Str1";                    // o1
2   const char *str2 = "Str2";                    // o2
3   const char *str3 = "Str3";                    // o3
4
5   void print(const char *x) {}
6
7   const char *getStr() {
8     const char *p = nondet() ? str1 : str2;
9     print(p);
10    return str1;
11  }
12
13  struct Config {
14    const char *label;
15    int *val;
16  };
17
18  int foo(struct Config *conf) {
19    const char str4[5] = "Str4";                // o4
20    print(str4);
21    const char *r = getStr();
22    print(r);
23    return conf->label == r;
24  }
25
26  int bar() {
27    int i = 42;                                 // o5
28    const char *s = nondet() ? str2 : str3;
29    struct Config c = {s, &i}                   // u
30    return foo(&c);
31  }
32
```

A Context-sensitive Points-To Graph:



Definition:

Oversharing -- existence of large number of inaccessible foreign objects during the analysis of a particular function.

Data Structure Analysis (DSA)

A state-of-the-art PTA for LLVM [1].

- Unification-based (Steensgaard-style), context- and field-sensitive.
- Uses a Union-Find data structure for efficient abstract object grouping.

- Analysis performed in 3 phases:
  - Local -- resolves local points-to information;
  - Bottom-Up -- inlines points-to information from callees to callers;
  - Top-Down -- inlines points-to information from callers to callees.

- Works around the problem of having too many abstract object by maintaining a separate context-insensitive points-to graph for global variables.

SeaDsa -- an implementation of DSA used by the SeaHorn verification framework [2]:

- Context-, field-, and array-sensitive;
- Designed to work on (small) SVComp benchmarks, no workaround for global variables.

[1] C. Lattner, V. S. Adve: Automatic pool allocation: improving performance by controlling data structure layout in the heap. PLDI 2005
[2] A. Gurfinkel, J. A. Navas: A Context-Sensitive Memory Model for Verification of C/C++ Programs. SAS 2017

Contribution #1

DSA -- a Formulation with Inference Rules

A simple LLVM-like Low-level language.

Syntax	Semantics
`r = alloc()`	$\dfrac{i:\texttt{r = alloc()} \quad H_a, H_b \notin M}{V, P, M, T, S, i \hookrightarrow V, P[\texttt{r} \mapsto H_a], M \cup H_a \cup H_b, T[H_a, H_b := \texttt{char}], S, i+1}$
`r = cast T, p`	$\dfrac{i:\texttt{r = cast T, p} \quad [\![\texttt{p}]\!]_{P,M} \mapsto H}{V, P, M, T, S, i \hookrightarrow V, P[\texttt{r} \mapsto H], M, T, S, i+1}$ $\dfrac{i:\texttt{r = cast T, p} \quad [\![\texttt{p}]\!]_{V} = v}{V, P, M, T, S, i \hookrightarrow V[\texttt{r} := v], P, M, T, S, i+1}$
`r = gep PT p, f`	$\dfrac{i:\texttt{r = gep T p, a} \quad [\![\texttt{p}]\!]_{P,M} \mapsto H}{V, P, M, T, S, i \hookrightarrow V, P[\texttt{r} \mapsto H], M, T, S, i+1}$ $\dfrac{i:\texttt{r = gep T p, b} \quad [\![\texttt{p}]\!]_{P,M} \mapsto H \quad siblingObj(H) = I}{V, P, M, T, S, i \hookrightarrow V, P[\texttt{r} \mapsto I], M, T, S, i+1}$
`store r, PT p`	$\dfrac{i:\texttt{store r, T* p} \quad [\![\texttt{p}]\!]_{P,M} \mapsto H \quad [\![\texttt{r}]\!]_{P,M} \mapsto I}{V, P, M, T, S, i \hookrightarrow V, P, M[H \mapsto I], T[H := \texttt{T}], S, i+1}$ $\dfrac{i:\texttt{store r, T* p} \quad [\![\texttt{p}]\!]_{P,M} \mapsto H \quad [\![\texttt{r}]\!]_{V} = v}{V, P, M, T, S, i \hookrightarrow V, P, M[H := v], T[H := \texttt{T}], S, i+1}$
`r = load PT p`	$\dfrac{i:\texttt{r = load T* p} \quad [\![\texttt{p}]\!]_{P,M} \mapsto H \quad [\![H]\!]_{T} = \texttt{U} \quad \texttt{T} \sqsubseteq \texttt{U} \quad [\![H]\!]_{M} \mapsto I}{V, P, M, T, S, i \hookrightarrow V, P[\texttt{r} \mapsto I], M, T, S, i+1}$ $\dfrac{i:\texttt{r = load T* p} \quad [\![\texttt{p}]\!]_{P,M} \mapsto H \quad [\![H]\!]_{T} = \texttt{U} \quad \texttt{T} \sqsubseteq \texttt{U} \quad [\![H]\!]_{M} = v}{V, P, M, T, S, i \hookrightarrow V[\texttt{r} := v], P, M, T, S, i+1}$

PTA inference rules.

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

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

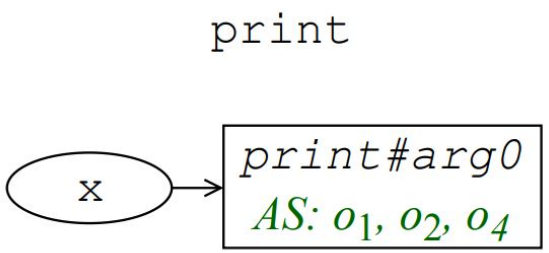
$$\frac{\texttt{r = gep PT p, a} \quad \texttt{p} \mapsto H}{\texttt{r} \mapsto H}\ \text{GEP} \qquad \frac{\texttt{r = gep PT p, b} \quad \texttt{p} \mapsto H \quad fld(H) = \texttt{a} \quad siblingObj(H) = I}{\texttt{r} \mapsto I}\ \text{GEP}$$

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

DSA -- an Improved Example

```
1   const char *str1 = "Str1";                    // o1
2   const char *str2 = "Str2";                    // o2
3   const char *str3 = "Str3";                    // o3
4
5   void print(const char *x) {}
6
7   const char *getStr() {
8     const char *p = nondet() ? str1 : str2;
9     print(p);
10    return str1;
11  }
12
13  struct Config {
14    const char *label;
15    int *val;
16  };
17
18  int foo(struct Config *conf) {
19    const char str4[5] = "Str4";                // o4
20    print(str4);
21    const char *r = getStr();
22    print(r);
23    return conf->label == r;
24  }
25
26  int bar() {
27    int i = 42;                                 // o5
28    const char *s = nondet() ? str2 : str3;
29    struct Config c = {s, &i}                   // u
30    return foo(&c);
31  }
32
```

A better Points-To Graph for `print`:



Contribution #2

Based on the formulation, we show that no abstract objects should be copied during the Top-Down phase of DSA.

Theorem 1 $(\Gamma_{\mathrm{DSA}} \vdash_P x \overset{F}{\mapsto} H)$

$$\implies \exists y \cdot (\Gamma_{\mathrm{L}} \cup \Gamma_{\mathrm{BU}} \vdash_P y \overset{F}{\mapsto} H)$$

DSA -- Improving Precision

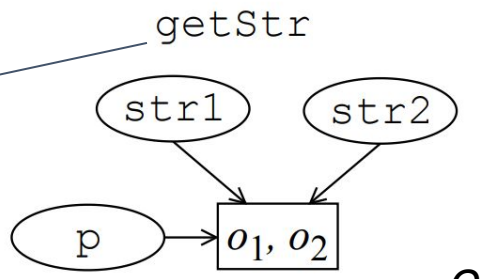
Precision can be improved by:

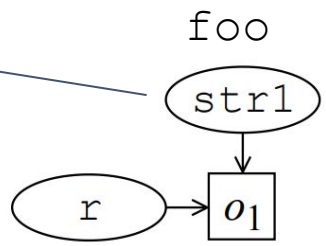
1. More precise intraprocedural (local) analysis
   - Less confusion locally and less local confusion propagated to analyses of other functions.

2. More precise interprocedural analysis
   - Less confusion propagated across functions.

DSA -- Improving Interprocedural Rules

```
1   const char *str1 = "Str1";                    // o1
2   const char *str2 = "Str2";                    // o2
3   const char *str3 = "Str3";                    // o3
4
5   void print(const char *x) {}
6
7   const char *getStr() {
8     const char *p = nondet() ? str1 : str2;
9     print(p);
10    return str1;
11  }
12
```





Observation:
Abstract objects that do not alias the passed parameters and returned values do not have to be propagated.

Contribution #1

DSA -- a formulation with inference rules

A simple LLVM-like Low-level language.

	Semantics
	$\frac{i:\texttt{r = alloc()} \quad H_a, H_b \notin M}{V,P,M,T,S,i \hookrightarrow V,P[\texttt{r} \mapsto H_a],M \cup H_a \cup H_b,T[H_a,H_b := \texttt{char}],S,i+1}$
	$\frac{i:\texttt{r = cast T, p} \quad [\texttt{p}]_{P,M} \mapsto H}{V,P,M,T,S,i \hookrightarrow V,P[\texttt{r} \mapsto H],M,T,S,i+1}$ $\quad \frac{i:\texttt{r = cast T, p} \quad [\texttt{p}]_V = v}{V,P,M,T,S,i \hookrightarrow V[\texttt{r} := v],P,M,T,S,i+1}$
f	$\frac{i:\texttt{r = gep T p, a} \quad [\texttt{p}]_{P,M} \mapsto H}{V,P,M,T,S,i \hookrightarrow V,P[\texttt{r} \mapsto H],M,T,S,i+1}$ $\frac{i:\texttt{r = gep T p, b} \quad [\texttt{p}]_{P,M} \mapsto H \quad siblingObj(H) = I}{V,P,M,T,S,i \hookrightarrow V,P[\texttt{r} \mapsto I],M,T,S,i+1}$
	$\frac{i:\texttt{store r, T* p} \quad [\texttt{p}]_{P,M} \mapsto H \quad [\texttt{r}]_{P,M} \mapsto I}{V,P,M,T,S,i \hookrightarrow V,P,M[H \mapsto I],T[H := \texttt{T}],S,i+1}$ $\frac{i:\texttt{store r, T* p} \quad [\texttt{p}]_{P,M} \mapsto H \quad [\texttt{r}]_V = v}{V,P,M,T,S,i \hookrightarrow V,P,M[H := v],T[H := \texttt{T}],S,i+1}$
	$\frac{i:\texttt{r = load T* p} \quad [\texttt{p}]_{P,M} \mapsto H \quad [H]_T = \mathtt{U} \quad \texttt{T} \sqsubseteq \mathtt{U} \quad [H]_M \mapsto I}{V,P,M,T,S,i \hookrightarrow V,P[\texttt{r} \mapsto I],M,T,S,i+1}$ $\frac{i:\texttt{r = load T* p} \quad [\texttt{p}]_{P,M} \mapsto H \quad [H]_T = \mathtt{U} \quad \texttt{T} \sqsubseteq \mathtt{U} \quad [H]_M = v}{V,P,M,T,S,i \hookrightarrow V[\texttt{r} := v],P,M,T,S,i+1}$

PTA inference rules.

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

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

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

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

15

Contribution #3

Improved global reasoning with Partial Flow-Sensitivity at call- and return-sites.

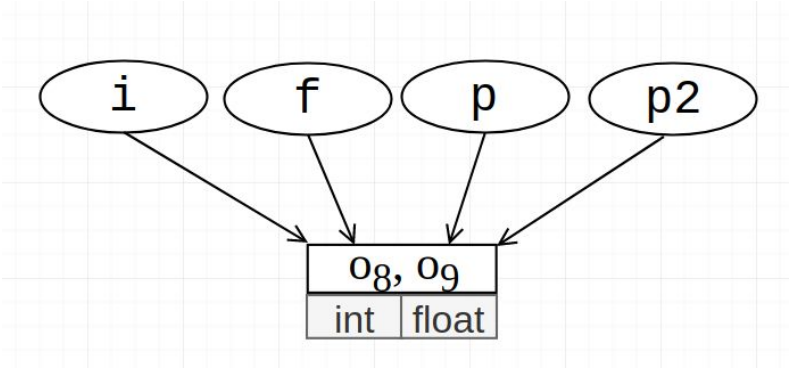
DSA -- Improving Local Rules

The C11 programming language in Section 6.5 introduces *effective type rules:*

- Roughly, every memory location has a type determined by the last write and all reads from that memory location must be of compatible types.

When analyzing memory reads in PTA, we can exploit it and ignore writes of incompatible types that definitely do not affect the read values.

```
int main() {
  int i = 42;              // o8
  float f = 3.14f;         // o9
  void *p = &i;

  if (tricky_condition())
    p = &f;

  int *p2 = (int *) p;
  print_int(*p2);
}
```



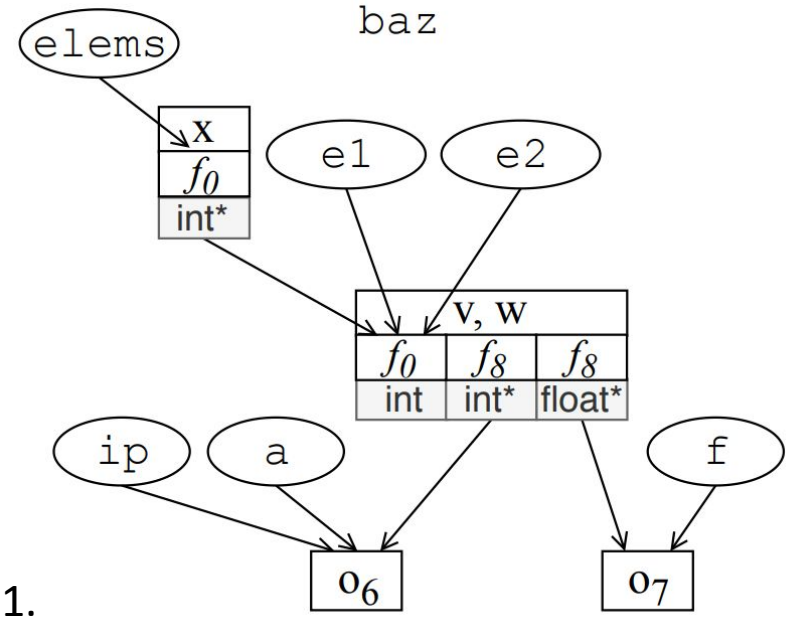
Must be an `int`

DSA -- Improving Local Rules

The C11 programming language in Section 6.5 introduces *effective type rules:*

- Roughly, every memory location has a type determined by the last write and all reads from that memory location must be of compatible types.

When analyzing memory reads in PTA, we can exploit it and ignore writes of incompatible types that definitely do not affect the read values.



Contribution #4

Improved local reasoning,
based on the *effective type rules* of C11.

Evaluation -- a Program Verification Task

A program verification task: detecting a class of memory-safety bugs, called field-overflow bugs:

- A field-overflow happens when a field not present in an object is tried to be accessed, causing an access outside of the allocated object.

```
  1  const int BASE_TAG = 0, INT_TAG = 1;
  2  struct Element { int tag; };
  3  struct IElement { Element e; int d }
  4
  5  void baz() {
1 6    Element e1 = {BASE_TAG};
2 7    IElement e2 = {{INT_TAG}, 42};
  8    Element *elems[2] = {&e1, &e2.e};
  9
 10    for (int i = 0; i < 2; ++i)
 11      if (elems[i]->tag == INT_TAG) {
 12        IElement *ie = (IElement *) elems[i];
 13        print_int(ie->d);
 14      }
 15  }
 16
```

Only safe for 2

To know if an access is safe or not, we need to identify all potential Allocation Sites of the accessed pointer.

If the Allocation Site the pointer originates from is too small, the access is not safe.

Evaluation -- Simple Memory Checker

- A checker for the Program Verification Task, implemented in the SeaHorn verification framework.

- For all memory accesses, identifies all potential allocation sites and checks if the accesses pointer comes from an allocation site of insufficient size.

  a. All allocation sites of variable size are discarded.
  b. Allocation sites of statically-known insufficient size need to be checked.
  c. Allocation sites of statically-known sufficient size are safe.

Evaluation -- Setup

Based on the Simple Memory Checker analysis.

- Comparison against the vanilla SeaDsa, SeaDsa with the Top-Down optimization and Partial Flow-Sensitivity.

- Comparison against two PTAs from SVF: the WaveDiff pre-analysis and the Sparse Value-Flow PTA.
  - Inclusion-based flow-sensitive state-of-the-art PTAs.
  - Allocation site detection modified to match the one from SeaDsa and TeaDsa.

- All target programs linked into a single LLVM bitcode file (whole-program analysis).
  - Popular C and C++ programs.
  - Program size ranges from 140 kB to 157 MB of bitcode.

Evaluation -- Performance

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

Evaluation -- Precision

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

* Lower is better

Conclusions

1. Reducing oversharing in DSA-style PTAs improves both performance and precision.
   a. Most performance gained by improving the Top-Down phase and not copying abstract objects.
   b. Most precision gained by introducing partial flow-sensitivity and type-awareness.

2. New optimizations were possible thanks to a new formulation of DSA.
   a. Formal mechanism to ask questions about properties of PTAs.
   b. Provably better performance and precision.

3. Time to re-evaluate tradeoffs between Inclusion- and Unification-based PTAs for real-world Low-level programs?

Thank you

Questions?