Unification-based Pointer Analysis without Oversharing

Jakub Kuderski\textsuperscript{1,3}, Jorge A. Navas\textsuperscript{2}, Arie Gurfinkel\textsuperscript{1}

\textsuperscript{1} University of Waterloo, Canada
\textsuperscript{2} SRI International, USA
\textsuperscript{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

- 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.

```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;
}
```
Pointers in Low-level Languages

**Definition:**

**Pointer** -- object identifier and offset within that object.

```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;
    return 0;
}
```
Pointers in Program Verification

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

- 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) -- determining whether a given pointer:

- aliases with another pointer (alias analysis)
  \[ \text{alias}(p_1, p_2) \]

- points to an object (points-to analysis)
  \[ p \mapsto o \]

- 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

Inclusion-based (Andersen-style):
- e.g., SVF

Unification-based (Steensgaard-style)
- e.g., DSA, SeaDsa

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

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

```c
int *ptr;
ptr = (int *) malloc(sizeof(int));  // o1
int **ptr_ptr = NULL;
if (nondet()) {
    MyClass *classPtr = new MyClass();  // o3
    ptr_ptr = (int *) classPtr;
} else {
    ptr_ptr = &ptr;
    **ptr_ptr = 42;
}
```
# Inclusion and Unification Constraints

## Inclusion-based
(Andersen-style):
- e.g., SVF.

### Instructions
- `p = malloc(n)`
  - Inclusion constraint: `p ⊇ loc(malloc)`
  - Unification constraint: `p ≈ loc(malloc)`
- `p = q`
  - Inclusion constraint: `p ⊇ q` (may also be `p = q` if `p` and `q` are the same location)
  - Unification constraint: `p ≈ q`
- `*p = q`
  - Inclusion constraint: `pts(p) ⊇ q`
  - Unification constraint: `pts(p) = q`
- `p = *q`
  - Inclusion constraint: `p ⊇ pts(q)`
  - Unification constraint: `p ≈ pts(q)`
- `p = &x`
  - Inclusion constraint: `p ⊇ loc(x)`
  - Unification constraint: `p ≈ loc(x)`

## Unification-based
(Steensgaard-style)
- e.g., DSA, SeaDsa.

### Diagrams

![Diagram showing inclusion and unification constraints](image_url)
Conventional Wisdom

Inclusion-based (Andersen-style):
- e.g., SVF.

Unification-based (Steensgaard-style)
- e.g., DSA, SeaDsa.

<table>
<thead>
<tr>
<th>Property</th>
<th>Inclusion-based</th>
<th>Unification-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision?</td>
<td>Precise ✓</td>
<td>Imprecise x</td>
</tr>
<tr>
<td>Speed?</td>
<td>Slow x</td>
<td>Fast ✓</td>
</tr>
<tr>
<td>Memory consumption?</td>
<td>Large x</td>
<td>Small ✓</td>
</tr>
<tr>
<td>Patent issues?</td>
<td>No x</td>
<td>Yes ✓</td>
</tr>
</tbody>
</table>

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

A Context-insensitive Points-To Graph:

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

void print(const char *x) {}

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

struct Config {
    const char *label;
    int *val;
};

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

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

A Context-sensitive Points-To Graph:

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

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

struct Config {
    const char *label;
    int *val;
};

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

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

**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.

Contribution #1

DSA -- a Formulation with Inference Rules

A simple LLVM-like Low-level language.

PTA inference rules.

\[
\begin{align*}
\text{i:r = alloc()} & \quad \frac{}{r \rightarrow H} \\
\text{r = cast T, p} & \quad \frac{p \rightarrow H, r \rightarrow H}{r \rightarrow H} \\
\text{r = gep PT p, t} & \quad \frac{p \rightarrow H, t \rightarrow I}{r \rightarrow H} \\
\text{store r, PT p} & \quad \frac{p \rightarrow H, r \rightarrow H}{r \rightarrow H} \\
\text{i:fun \text{ fn}(\bar{x}): \bar{y}} & \quad \frac{0 \leq k < |\bar{x}|}{\text{FORMALS}}
\end{align*}
\]
### Contribution #2

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

**Theorem 1** \( (\Gamma_{\text{DSA}} \vdash_P x \xrightarrow{F} H) \quad \implies \quad \exists y \cdot (\Gamma_L \cup \Gamma_{\text{BU}} \vdash_P y \xrightarrow{F} H) \)
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

Contribution #3

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

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

```c
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`
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.

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.

```c
const int BASE_TAG = 0, INT_TAG = 1;
struct Element { int tag; };
struct IEElement { Element e; int d };

void baz() {
    Element e1 = {BASE_TAG};
    IEElement e2 = {{INT_TAG}, 42};
    Element *elems[2] = {&e1, &e2.e};

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

Only safe for 2
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

<table>
<thead>
<tr>
<th>Program</th>
<th>Bitcode Size [kB]</th>
<th>Wave Diff</th>
<th>SVF Sparse</th>
<th>Results</th>
<th>PFS-SEADSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqlite</td>
<td>140</td>
<td>&lt;1</td>
<td>106</td>
<td>&lt;1</td>
<td>135</td>
</tr>
<tr>
<td>bftpd</td>
<td>268</td>
<td>&lt;1</td>
<td>114</td>
<td>&lt;1</td>
<td>133</td>
</tr>
<tr>
<td>htop</td>
<td>320</td>
<td>&lt;1</td>
<td>216</td>
<td>7</td>
<td>483</td>
</tr>
<tr>
<td>cass</td>
<td>1,384</td>
<td>&lt;1</td>
<td>399</td>
<td>1</td>
<td>453</td>
</tr>
<tr>
<td>wasm-dis</td>
<td>1,420</td>
<td>6</td>
<td>1,041</td>
<td>122</td>
<td>4,594</td>
</tr>
<tr>
<td>openssl</td>
<td>1,504</td>
<td>1</td>
<td>706</td>
<td>2</td>
<td>792</td>
</tr>
<tr>
<td>wasm-as</td>
<td>1,824</td>
<td>10</td>
<td>1,428</td>
<td>195</td>
<td>8,249</td>
</tr>
<tr>
<td>h264ref</td>
<td>2,468</td>
<td>6</td>
<td>1,655</td>
<td>7</td>
<td>1,784</td>
</tr>
<tr>
<td>tmux</td>
<td>2,996</td>
<td>1</td>
<td>586</td>
<td>3</td>
<td>696</td>
</tr>
<tr>
<td>wasm-opt</td>
<td>3,520</td>
<td>36</td>
<td>2,784</td>
<td>960</td>
<td>33,138</td>
</tr>
<tr>
<td>llvm-dis</td>
<td>11,232</td>
<td>1,640</td>
<td>9,964</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>llvm-as</td>
<td>14,012</td>
<td>4,892</td>
<td>15,377</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>llvm-opt</td>
<td>16,012</td>
<td>9,104</td>
<td>19,633</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>rippled</td>
<td>157,804</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
## Evaluation -- Precision

<table>
<thead>
<tr>
<th>Program</th>
<th>Bitcode Size [kB]</th>
<th>SVF Sparse</th>
<th>SEADSA</th>
<th>PFS-SEADSA</th>
<th>TEADSA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Checks</td>
<td>Aliases</td>
<td>Checks</td>
<td>Aliases</td>
</tr>
<tr>
<td>sqlite</td>
<td>140</td>
<td>&lt;1k</td>
<td>&lt;1k</td>
<td>1k</td>
<td>3k</td>
</tr>
<tr>
<td>bftpd</td>
<td>268</td>
<td>&lt;1k</td>
<td>&lt;1k</td>
<td>&lt;1k</td>
<td>1k</td>
</tr>
<tr>
<td>htop</td>
<td>320</td>
<td>24k</td>
<td>26k</td>
<td>110k</td>
<td>110k</td>
</tr>
<tr>
<td>cass</td>
<td>1,384</td>
<td>1k</td>
<td>7k</td>
<td>12k</td>
<td>14k</td>
</tr>
<tr>
<td>wasm-dis</td>
<td>1,420</td>
<td>132k</td>
<td>241k</td>
<td>616k</td>
<td>634k</td>
</tr>
<tr>
<td>openssl</td>
<td>1,504</td>
<td>&lt;1k</td>
<td>2k</td>
<td>&lt;1k</td>
<td>4k</td>
</tr>
<tr>
<td>wasm-as</td>
<td>1,824</td>
<td>243k</td>
<td>412k</td>
<td>933k</td>
<td>957k</td>
</tr>
<tr>
<td>h264ref</td>
<td>2,468</td>
<td>&lt;1k</td>
<td>37k</td>
<td>21k</td>
<td>174k</td>
</tr>
<tr>
<td>tmux</td>
<td>2,996</td>
<td>8k</td>
<td>17k</td>
<td>403k</td>
<td>422k</td>
</tr>
<tr>
<td>wasm-opt</td>
<td>3,520</td>
<td>718k</td>
<td>1,174k</td>
<td>8,851k</td>
<td>8,637k</td>
</tr>
<tr>
<td>llvm-dis</td>
<td>11,232</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>llvm-as</td>
<td>14,012</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>llvm-opt</td>
<td>16,012</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>rippled</td>
<td>157,804</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

* 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?