TeaDsa: Type-aware DSA-style Pointer Analysis for Low Level Code

Jakub Kuderski, Nhâm Lê, Arie Gurfinkel (UWaterloo); Jorge Navas (SRI International)

Pointer Analysis (PTA)

- Determines whether two pointer expressions refer to the same memory location
- Essential for any non-trivial reasoning about programs

Aliasing

Pointer expressions p_1 and p_2 alias, alias(p_1, p_2), iff there is an execution Π such that:

p_1, p_2 are valid at a program location l and

$$\Pi[p_1@l] = \Pi[p_2@l]$$

Observation

alias(p_1, p_2) does not imply $p_1 = p_2$

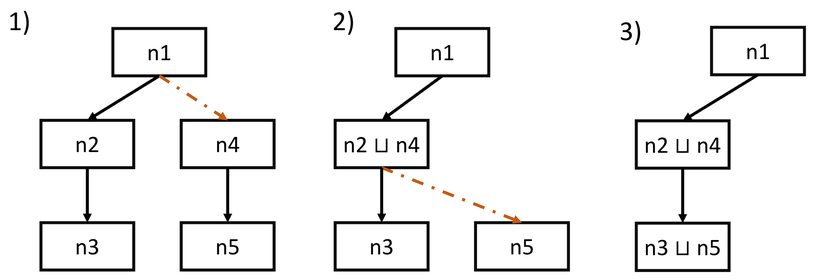
Existing PTAs for LLVM are Inadequate

1. SeaDsa
   - Context-, field-, and array-sensitive
   - Unification-based (Steensgaard-style)
   - Scalable but not precise enough
2. SVF
   - Context and array-insensitive, quasi-field-sensitive
   - Inclusion-based (Andersen-style)
   - Does not scale

Statement	Inclusion-based	Unification-based
$p = malloc(n)$	$p \supseteq \text{loc(malloc)}$	$p \approx loc(malloc)$
$p = q$	$p \supseteq q$	$p \approx q$
$*p = q$	$\text{pts(p)} \supseteq q$	$\text{pts(p)} \approx q$
$p = *q$	$p \supseteq \text{pts}(q)$	$p \approx \text{pts}(q)$
$p = \&x$	$p \supseteq loc(x)$	$p \approx loc(x)$

Source of Imprecision in Unification-based PTA

- A points-to graph node can have at most one outgoing edge
- If a node were to have two outgoing edges, the destination nodes need to be merged together, forming a single node



Idea: TeaDsa

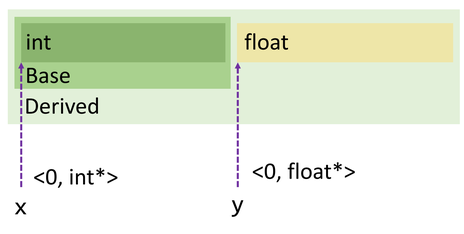
Improve SeaDsa's precision by using type-based field-sensitivity!

- Fields as abstract objects
- Types discovered by accesses
- Fields identified by <Offset, Type>
- Allows to separate fields based on types, even if offsets are the same

```
struct Base { int val; };
struct Derived : Base { float size; };

void *buffer = malloc(sizeof(Derived));
__construct_Base((Base *) buffer);
__construct_Derived((Derived *) buffer);

Derived *obj = (Derived *) buffer;
int *x = &(obj->val);
float *y = &(obj->size);
```



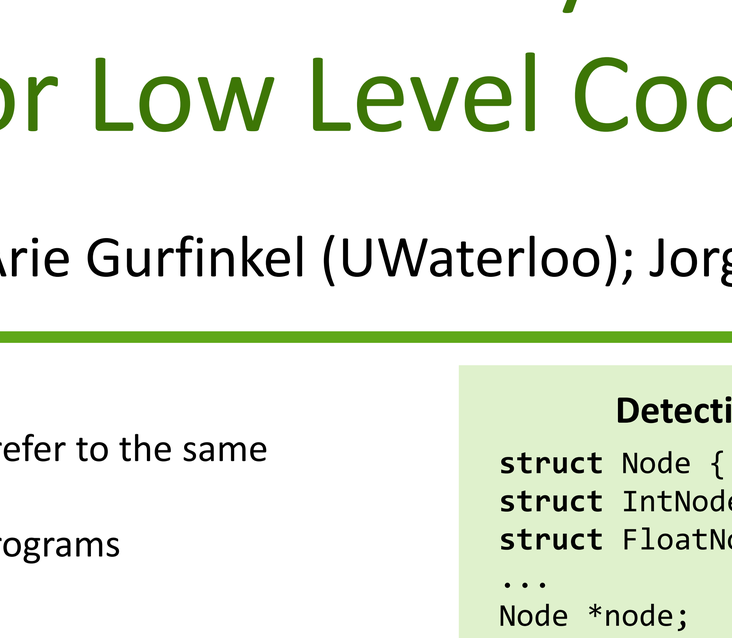
Composite types identified by the type of the innermost field.

Detecting Field Overflow Memory Safety Bugs

```
struct Node { Node *next = nullptr; int TAG; };
struct IntNode : Node { int *i; };
struct FloatNode : Node { float *f; };
...
Node *node;

node = getNode();
if (node->TAG == INT_TAG)
  *(((IntNode *) node)->i) = 123;     // SAFE?

node = getNode();
*(((FloatNode *) node)->f) = 3.14f;  // SAFE?
```



Is Relying on Types Sound?

- Low-level language features (casts, memcpy, type punning)
- Potential memory faults

Consistent Dynamic Typing Assumption

Every program object must have a unique effective type during its life time. The type is known at first assignment and can never change while the object is alive.

1. Stronger than Strict Aliasing in C and C++ standards
2. Consistent Dynamic Typing Checker:
   - Use the results of TeaDsa to statically check if the CDT assumption holds
   - Reasoning appears to be circular, but is not [4]

Evaluation

Program	Source Language	Size [kB of bitcode]	SVF Time [s]	SeaDsa Time [s]	TeaDsa Time [s]	% Checks Discharged with Types
bzip2	C	29	173	0.19	0.19	0
mcf	C	37	1.98	0.02	0.03	--
libquantum	C	80	8.66	0.08	0.09	--
sjeng	C	308	260	0.44	0.45	0
CASS	C++	765	5390	6.20	5.85	65
htop	C	800	--	5.02	3.80	71
hmmer	C	859	2548	3.51	3.60	1
h264ref	C	1784	11525	9.44	10	26

References:

[1] A. Gurfinkel, J. A. Navas: A Context-Sensitive Memory Model for Verification of C/C++ Programs. SAS 2017

[2] Y. Sui, J. Xue: SVF: interprocedural static value-flow analysis in LLVM. CC 2016

[3] C. Lattner, A. Lenharth, V. S. Adve: Making context-sensitive points-to analysis with heap cloning practical for the real world. PLDI 2007

[4] C. L. Conway et al.: Pointer Analysis, Conditional Soundness, and Proving the Absence of Errors. SAS 2008

Available on GitHub!
github.com/seahorn/sea-dsa/tree/types