Topic 17: Memory Analysis

COS 320

Compiling Techniques

Princeton University
Spring 2016

Lennart Beringer

Motivation

Is this string sanitized?

Do pointers p and q alias?

Can pointer x be NULL here?

Can I eliminate this load instruction?

Can I eliminate this store instruction?

Can these three store instructions be scheduled for the same cycle?

Does a pointer to my private key leak to code outside the crypto library?

Can I swap the order of this load-store pair?

Is my program memory-safe?

Can we exploit the distinctness of pointers to improve precision of analyses and used for constant propagation, common subexpression elimination, etc.?

Flavors of memory analyses I

Alias analyses concern potential /definitive (non-) equality of **two** items:

- Can x and y point to the same address? Or are they definitely separate?
- Are x.f1 and y.f2 always pointing to the same object?
- May versus must:
  - **May** alias: is it **possible** that x and y point to the same address?
  - **Must** alias: do x and y **necessarily/always** point to the same address?

Flavors of memory analyses I

Alias analyses concern potential /definitive (non-) equality of **two** items:

- Can x and y point to the same address? Or are they definitely separate?
- Are x.f1 and y.f2 always pointing to the same object?
- May versus must:
  - **May** alias: is it **possible** that x and y point to the same address?
  - **Must** alias: do x and y **necessarily/always** point to the same address?

Points-to analyses concern a **single** item:

- can x be NULL?
- what objects might this method m be invoked upon?
- again, both questions have **may** and **must** variants

Flavors of memory analyses I

Alias analyses concern potential /definitive (non-) equality of **two** items:

- Can x and y point to the same address? Or are they definitely separate?
- Are x.f1 and y.f2 always pointing to the same object?
- May versus must:
  - **May** alias: is it **possible** that x and y point to the same address?
  - **Must** alias: do x and y **necessarily/always** point to the same address?

Points-to analyses concern a **single** item:

- can x be NULL?
- what objects might this method m be invoked upon?
- again, both questions have **may** and **must** variants

Can obtain **alias** information from **points-to** analysis:

- PtsTo(x) ∩ PtsTo(y) = Ø ➔ x and y **don't** alias
- PtsTo(x) ∩ PtsTo(y) ≠ Ø ➔ x and y **may** alias
- PtsTo(x) = {a} = PtsTo(y) ➔ x and y **must** alias, to location x

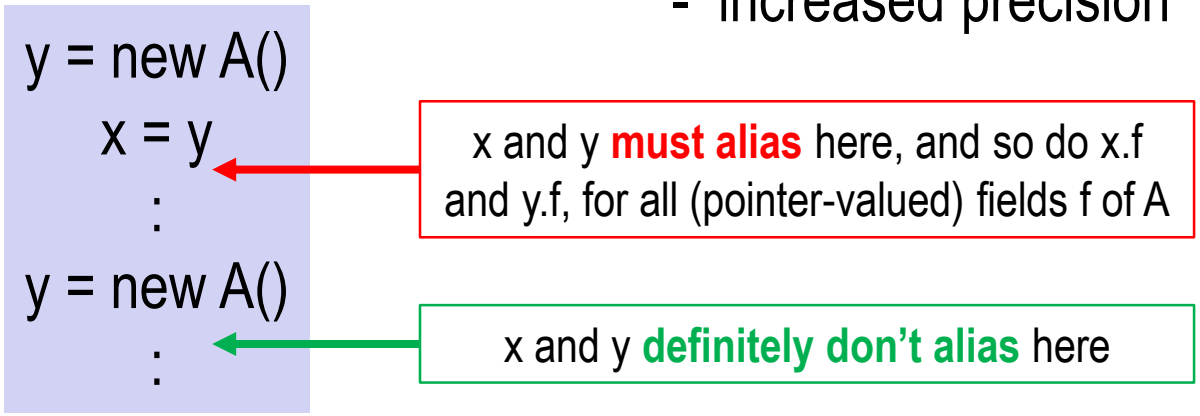
Flavors of memory analyses II

Flow-insensitive

- points-to/alias information holds globally / in entire method
- simplifies analysis: order of instructions irrelevant, conditionals, loop conditions can be ignored
  + speeds up analysis
  - reduces precision

Flow-sensitive

- points-to/alias information given for each program point
- order of instructions relevant; code traversal a la data flow analysis, with join operation at control flow merge points
  + more costly analysis
  - increased precision



SSA disambiguates the 2 defs of y, but object references may have been copied elsewhere, and SSA doesn't cover fields.

Flavors of memory analyses III

Information can be collected for

- **program variables**: PtsTo(x) = {a, b, null} , MustAlias(x, y) = true
- **fields**: PtsTo(C.f) = {a, null}, MayAlias (C.f, D.g) = false

captures the f fields of all (static? dynamic?) C-objects

- **access paths:** items of the form $b.f_1.f_2.\ldots.f_n$ where f_i are field names and "base" b can again be
  - a variable: MustAlias ($x.f_1.g_2$, $y.g_1.f_3$)
  - a class name: MayAlias ($C.f_1.g_2$, $D.g_1.f_3$)
  - a symbolic identifier of an object (specified by allocation site)

Fields f of A-objects allocated **here**
and g-fields of B-objects allocated
here definitely don't alias here

```
x = new A(…)
:
y = new B(..)
:
if (z.f == v.g) …
```

So need to know where the objects held in z and y were allocated…(next slide)

Flavors of memory analyses IV

What information is associated with variables / fields / access paths?

- other variables / fields / access paths
- abstract locations (eg allocation sites)

Presence of merge points and need for conservativeness / approximation suggests using **sets** of abstract locations etc, or other lattice structure

Flavors of memory analyses IV

What information is associated with variables / fields / access paths?

- other variables / fields / access paths
- abstract locations (eg allocation sites)

Presence of merge points and need for conservativeness / approximation suggests using **sets** of abstract locations etc, or other lattice structure

How can different objects allocated at the same allocation site be distinguished?

```
method A m () {
    return new A(…)
}
```

clearly, different calls to m yield different objects!

```
for (int i = 0; i < N; i++) {
    x = new A(…)
    myArray[i] =x
}
```

clearly, this loop yields N different objects!

Flavors of memory analyses V

Example: “k-CFA”

One solution: (call) context abstraction:

- refine **allocation site** by adding a static approximation of the frame stack that is valid when the allocation is executed

list of method names
(reverse order, truncated)

list of method names **and the (abstractions of) the objects they were invoked upon**
(reverse order, truncated)

list of method names and the (abstractions of) the objects they were invoked upon,
and the (abstractions of) the arguments of these method calls
(reverse order, truncated)

Often, truncation necessary at length 2 or 3 ;-)

Challenges of memory analyses

- precision, precision, precision, precision!
- speed, speed, speed!

Yes, but not blindly: need to balance precision and speed

- which code sections need to analyzed in detail?
- which transformations need /can exploit precision, and how much do these transformations speed up the code **you are interested in**
- is the application in a JIT compiler? Or are you compiling code running in a data center, on millions of VM's?

- modularity: how are analysis results of methods / classes / libraries best communicated to the outside (procedure summaries, types, …)

Indeed, memory analyses often inter-procedural: worst case-assumptions on (non-)aliasing of method parameters too approximate.

M. Hind: Pointer Analysis: Haven't We Solved This Problem Yet? Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)}, pages = {54--61}, ACM 2001.

Compiler construction – done and dusted?

New architectures:

- Multi-core: parallelization eg of array intensive scientific code (DSWP)
- Multi-core: coordination between threads on machines with weak memory models
- GPU, FPGA, …

New applications

- Mobile (energy optimization)
- Data centers (energy)
- IoT / embedded systems

New languages

- Domain specific, eg networking (SDN)
- General purpose (Go, Rust, …)

New requirements: high-assurance

- Correctness: preservation of source language meaning – mathematically proven, mechanically checkable

Compiler construction – done and dusted?

New architectures:

- Multi-core: parallelization eg of array intensive scientific code (DSWP)
- Multi-core: coordination between threads on machines with weak memory models
- GPU, FPGA, …

New applications

- Mobile (energy optimization)
- Data centers (energy)
- IoT / embedded systems

New languages

- Domain specific, eg networking (SDN)
- General purpose (Go, Rust, …)

New requirements: high-assurance

- Correctness: preservation of source language meaning – mathematically proven, mechanically checkable

Thanks!