Alias Analysis of Executable Code

S. Debray, et al. (POPL ‘98)

Presented by Xin Qi

What is Special about Executables

- We no longer have
  - Types – can’t do type filtering
  - Structures – jump all around
- We have
  - Pointer arithmetics – a lot!
  - Normally whole-program information
- In addition
  - Compilers can do something unexpected
    - Tom Reps’ example about uninitialized variables

Introduction to the Analysis

- Works on RISC instruction set
  - Memory accessed only through load & store
  - Three-operator integer instructions:
    - Basically only add & mult (sub & mov modeled by add)
    - Bitwise operators?
- Properties of the analysis
  - May alias analysis
  - Flow-sensitive, context-insensitive, interprocedural

Naïve Approach

- Local Alias Analysis
  - Within a basic block
  - Two references are not aliasing each other if
    - Either they use distinct offsets from the same base register, and the register is not redefined in between
    - Or one points to stack and the other points to global data area
  - Not working across basic block boundaries

Residue-based Approach

- Want to know the set of possible addresses referenced by a memory access
  - Basically the set of possible values in a register
- Impractical to consider all possible integer values in registers
- For instruction add & mult, a very natural thing is to consider mod-k residues
  - Very easy to compute the new residue
  - $k = 2^m$ – The set of $\{0, 1, \ldots, k - 1\}$ is called Z_k

Residue-based Approach (cntd)

- Not always possible to compute a set of actual values for a register
  - User inputs
  - Read from memory
- Can't just say that it is Z_k
  - Too imprecise

Example

```
load r1, addr
…
add r1, 3, r2
add r1, 5, r3
…
```

Address Descriptors

- The idea of “being relative to a common value” is captured in address descriptors
- Address descriptors <I, M>
  - I – defining instruction, abstract away the unknown part
  - M – residue set, as before

Address Descriptors (cntd)

- Defining instruction I
  - Can be an instruction, NONE, or ANY
  - <NONE, *> represents absolute addresses
  - <ANY, *> is essentially $\perp$
- Residue set M
  - Set of mod-k addresses relative to the value defined in the instruction
  - <*, Z_k> is also $\perp$

Address Descriptors (cntd^2)

- $val_P(I)$ = set of values that some execution path of P would make I evaluate to
- Concretization function
  - $conc_P(\langle I, M \rangle) =$
    $\{w + ik + x \mid w \in val_P(I), x \in M, i \geq 0\}$
  - Why should $i \geq 0$?

Address Descriptors (cntd^3)

- A preorder relation <I_1, M_1> $\leq$ <I_2, M_2>
  - I_1 = ANY or $M_1 = Z_k$
  - $M_2 = \varnothing$
  - $I_1 = I_2$ and $M_1 \subseteq M_2$
- An equivalence relation
  - <*, Z_k> = <ANY, *> = $\perp$
  - <*, $\varnothing$> = $\top$
- We hence have a lattice

The Algorithm

- Transfer function
  - Load r, addr
    - <NONE, $\{val \bmod k\}$> if addr is read-only with val
    - <I, {0}>
  - Add src_a, src_b, dest (<I_a, M_a> and <I_b, M_b>)
    - If one of I_a and I_b is NONE, say I_a
      - A' = <I_b, $\{(x_a + x_b) \bmod k \mid x_a \in M_a, x_b \in M_b\}$>
      - A' if $A' \neq \perp$; <I, {0}> otherwise
    - Otherwise, <I, {0}>

The Algorithm (cntd)

- For each program point, only keep a single address descriptor for each register
  - Take glb if there are more
- Reasoning alias relationships
  - For different I's. can't say much but assume may alias
  - For same I, need to check it is the same value computed by I

Experimental Results

- Benchmarks
  - SPEC-95, and 6 others
- k = 64
- Precision measurement
  - Number of memory references that some information is obtained
    - 30% ~ 60%
- Cost
  - Time and space: almost linear

Experimental Results (cntd)

- Reason for loss of precision & for low cost
  - Memory is not modeled
    - No information for something that is saved in memory, and read out later
  - Multiple address descriptors are merged for every program point
  - Context insensitivity

Experimental Results (cntd^2)

- Utility of the analysis
  - Reducing the number of load instructions
    - Naïve algorithm improves by almost always $\leq$ 1%
    - This algorithm improves often close to 2%, sometimes even higher
      - Not very impressive still
  - Because …
    - Compiler has done a good job
    - Not many free registers to use

Conclusion

- It is an interesting problem to analyze executable code
- The algorithm is
  - Simple and elegant
  - Scalable
  - Somewhat useful

Discussion

- Weakness?
- Possible improvements?