

# 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



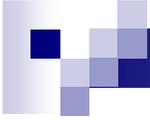
# Naïve Approach

- 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  $\{0, 1, \dots, k - 1\}$  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  $\langle I, M \rangle$ 
  - $I$  – 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
- $\langle \text{NONE}, * \rangle$  represents absolute addresses
- $\langle \text{ANY}, * \rangle$  is essentially  $\perp$

## ■ Residue set $M$

- Set of mod- $k$  addresses relative to the value defined in the instruction
- $\langle *, Z_k \rangle$  is also  $\perp$



# Address Descriptors (cntd<sup>2</sup>)

- $val_P(I)$  = 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<sup>3</sup>)

- A preorder relation  $\langle I_1, M_1 \rangle \leq \langle I_2, M_2 \rangle$ 
  - $I_1 = \text{ANY}$  or  $M_1 = Z_k$
  - $M_2 = \emptyset$
  - $I_1 = I_2$  and  $M_1 \subseteq M_2$
- An equivalence relation
  - $\langle *, Z_k \rangle = \langle \text{ANY}, * \rangle = \perp$
  - $\langle *, \emptyset \rangle = \top$
- We hence have a lattice



# The Algorithm

## ■ Transfer function

### □ Load $r$ , $\text{addr}$

■  $\langle \text{NONE}, \{val \bmod k\} \rangle$  if  $\text{addr}$  is read-only with  $val$

■  $\langle I, \{0\} \rangle$

### □ Add $\text{src}_a$ , $\text{src}_b$ , $\text{dest}$ ( $\langle I_a, M_a \rangle$ and $\langle I_b, M_b \rangle$ )

■ If one of  $I_a$  and  $I_b$  is NONE, say  $I_a$

□  $A' = \langle I_b, \{(x_a + x_b) \bmod k \mid x_a \in M_a, x_b \in M_b\} \rangle$

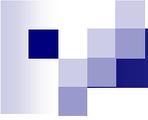
□  $A'$  if  $A' \neq \perp$ ;  $\langle I, \{0\} \rangle$  otherwise

■ Otherwise,  $\langle I, \{0\} \rangle$



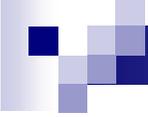
# The Algorithm (cntd)

- 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<sup>2</sup>)

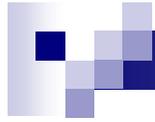
## ■ Utility of the analysis

- 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?