Type Recovery in a Binary Rewriter

Khaled ElWazeer
Term project for CMSC631
Motivation

• Why Type Recovery?
  - More Optimizations.
    • Ex; Stack Splitting – Automatic parallelization
    • Better Alias Analysis
  - Reducing Rewriting Overhead.
    • Ex; Unneeded type casts
  - Better identification of indirect call targets
    • Ex; using function pointers
  - Code Understanding
    • Higher level intermediate representation
SecondWrite – The Framework

- Takes any input binary
- Convert that into initial disassembly represented in LLVM
- LLVM memory to register promotion runs to convert to SSA form
- Internal passes run on the IR to get higher level representation Ex; Function Prototypes
- LLVM optimization passes run to enhance IR
Type Recovery

- A new internal pass added to the framework
- Tries to prove types for all symbols
- Runs on a functional IR to produce better and more understandable IR
- Will require LLVM optimizations to run again after modifying the IR to make full use of enabled optimizations.
Current Typing Problems

- Everything is created as byte pointers
  - For the purpose of indirect call targets identification
- Casts are all over the place
  - To convert between different types according to the uses
- Stack frames are allocated as arrays of byte pointers
Example – Sum of integers

```assembly
%sum = phi i8* [ %"sum_in_loop", %"loop" ], [ null, %"in" ]
%i = phi i8* [ %"i_in_loop", %"loop" ], [ null, %"in" ]
%"i_int" = ptrtoint i8* %i to i32
%"offset" = shl i32 %"i_int", 2
%"array_int" = ptrtoint i8* %"in_array" to i32
%"element_add" = add i32 %"i_int", %"array_int"
%"element_ptr" = inttoptr i32 %"element_add" to i8**
%"element_val" = load i8** %"element_ptr"
%"element_int" = ptrtoint i8* %"element_val" to i32
%"sum_int" = ptrtoint i8* %sum to i32
%"sum_in_loop_int" = add i32 %"sum_int", %"element_int"
%"sum_in_loop" = inttoptr i32 %"sum_in_loop_int" to i8*
%i_in_loop" = getelementptr i8* %i, i32 1
%"comparison" = icmp eq i8* %"i_in_loop", %"num"
```
Type Detection Methodology

- Detect real pointers, and prove that they are really pointers
- Detect pointer arithmetic, and possibly declare more pointers
- Repeat the above steps until no further pointers detected
- Prove types for all symbols given pointer information
A pointer is really a pointer if:

- **Base cases:**
  - There is some **load** from that pointer
  - There is some **store** to that pointer
  - If it is an argument/return to a known function **call** that takes/returns pointers

- **Derived -propagated- cases:**
  - If at least one operand to a **compare** / **select** / **PHI** / **cast** / **GEP** instruction is a pointer
Pointer Detection Analysis

- Walk the code instruction by instruction
- For each instruction, derive all possible pointers
- Iterate until no further pointers proved
- Analysis is done in reverse call graph order - callees, then callers -
- Inter procedural analysis.
- NOT ENOUGH; Ex; Arrays / Structures
Sum of integers - Revisited

%sum = phi i8* [ %"sum_in_loop", %"loop" ], [ null, %"in" ]

%i = phi i8* [ %"i_in_loop", %"loop" ], [ null, %"in" ]

%"i_int" = ptrtoint i8* %i to i32
%"offset" = shl i32 %"i_int", 2

%"array_int" = ptrtoint i8* %"in_array" to i32
%"element_add" = add i32 %"i_int", %"array_int"
%"element_ptr" = inttoptr i32 %"element_add" to i8**
%"element_val" = load i8** %"element_ptr"
Pointer Arithmetic Elimination

- Assumptions -
  - Two pointers added to each other can't produce a valid pointer
  - A pointer can't be multiplied/divided by some integer producing a valid pointer

- If one of those assumptions was not valid, fall back to the current working IR – Rare
- If they are valid, pointer bases and offsets are detected
Pointer Arithmetic Elimination

- For every detected pointer, with integer addition/subtraction origin:
  - Look at origin of operands
  - If one is a pointer, and the other is not, replace arithmetic with GEP
  - If one is NOT a pointer “result of mult / shift”, declare the other as pointer
  - If weird cases, not satisfying assumptions, fall back without declaring more pointers
Sum of integers - Revisited

%sum = phi i8* [ %"sum_in_loop", %"loop" ], [ null, %"in" ]

%i = phi i8* [ %"i_in_loop", %"loop" ], [ null, %"in" ]

%"i_int" = ptrtoint i8* %i to i32
%"offset" = shl i32 %"i_int", 2
%"array_int" = ptrtoint i8* %"in_array" to i32
%"element_add" = add i32 %"i_int", %"array_int"
%"element_ptr" = inttoptr i32 %"element_add" to i8**
%"element_val" = load i8** %"element_ptr"
Pointer Detection – Complete View

- Run basic pointer detection algorithm
- Run pointer arithmetic elimination
- Repeat until no further pointers are discovered
- Linear time algorithm, detecting (93)% of the pointers on average -on simple tests-
Type Detection using Pointers

- Because we know pointers, we can perform type detection based on that.
- Rules are created to detect what pointers point to, and what non-pointers are.
- Assumption: All originally non-pointer in the IR have correct types – Except for pointer arithmetic.
Basic Type Detection Rules

- Base cases - Examples:
  - Calling a function with a known prototype
    - Update arguments, return value types
  - Y = GEP X, offset
    - If Y and X are pointers, X is a possible array/structure
    - If both are non pointers, both are integers of offset type – replace by add
  - “y = inttoptr x to ptr_type”
    - If y is not a proved pointer, y takes type of x unless proved wrong, if a pointer, fall back
Basic Type Detection Rules

• Derived cases - Examples:
  - Y = load x_ptr / “same for store”
    • x_ptr is pointing to type of Y
  - If at least one operand to a compare / select / PHI instruction has a known type, propagate type to other operands
  - A return instruction
    • If return type was detected, propagate to all call sites
  - A call instruction
    • Propagate types for arguments/return type
Results – Floating point vs. Integer SPEC2006 Benchmarks

Bzip2
“Integer Benchmark”
- %64 initial pointers
- %79 proved pointers
- %72 detected types

Lbm
“Floating pt. Benchmark”
- %54 initial pointers
- %95 proved pointers
- %75 detected types
Small sum of int program results

- %63 initial pointers in the IR
- %57 proved pointers
- %79 detected types
- %93 pointer detection accuracy (2 pointers were not detected out of 27 correct pointers)
- %91 correctness in detected types (5 wrong types detected out of 55)
Conclusion

- Initial algorithm implemented for pointer detection
- Enhanced pointer detection through pointer arithmetic elimination
- Type detection based on detected pointer information and initial non-pointer types in the IR