Paper Review: TIE: Principled Reverse Engineering of Types in Binary Programs
Authors: JongHyup Lee, Thanassis Avgerinos, David Brumley
Carnegie Mellon University
Presented by: Charles Fauman, Andrew Ma, Johnson Liu

Overview
Reverse engineers binary code to get high level types
Implement novel techniques in a system called TIE
Infers types using the code, but doesn’t guess

Problem
Abstractions are removed as code gets translated into binary code
Current solutions are inefficient or inaccurate

Result
Type reconstruction techniques are promising alternatives
Provides more precise and conservative types than competitors
Challenges notion that type inference is impractical

Competitors
Hex-rays decompiler
REWARDS system

Example Input => Output
Three key principles

1. Variables have a certain type in a certain environment
   \[ \varphi : \text{list} \in \Gamma \]
   \[ \Gamma \vdash x : \text{list} \]

2. Bounded via lattice
   Inference rules: We specify typing rules as inference rules of the form:
   \[ \alpha \rightarrow \beta \]

3. Logical inference

Type definitions

Base Types in Lattice

Inference Rules

Method: Type Inference on Executables (TIE)

Conversion to Assembly and then to Binary Intermediate Language (BIL)
Variables are recovered using DVSA
([David?] Variable State Analysis)
- infer variable locations in memory
- infer aliases

DVSA
- A variable is represented by a series of memory loads and stores
- BIL is temporarily translated to single state assignment (SSA) form
  - Different uses of the same register are resolved
  - For control flow, variables must be resolved by a function \( \Phi \) either before each branch, e.g., \( \Phi(v_1, v_2) \)
  - Value points to a register
- A type of Value Set Analysis (VSA) is used to examine patterns in memory
  - One of possible memory locations for each variable
- This set of memory assignments \( \Phi(v) \) is a disjunction
  - \( \Phi(v) \) is a type of Value Set Analysis (VSA)
- Each set of possible memory locations for a variable is approximated to a range \([lb, ub]\)
  - This is an over-approximation, e.g., \([1, 4]\) for \([1, 2, 4]\)
- \( \Phi \) is a more accurate generalization of VSA
  - Based on \( \Phi \) with respect to any number of reference registers

Fresh Variables and some Base Types
- each variable is assigned an unknown type \( \tau \)
- BIL provides low-level typing for all registers and memory cells
  - \( \tau \) updates if loaded into eax (reg32_t)
  - eax is a 32-bit long register

Constraint Generation from Functions
- The use of variables in known functions provides constraints
  - \( \Phi \) is a disjunction over a given function
  - Values must be a valid type for the function

Constraints from simple functions

<table>
<thead>
<tr>
<th>Type</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Phi )</td>
<td>( \Phi(v) ) is a disjunction</td>
</tr>
<tr>
<td>( \Phi(v) )</td>
<td>( \Phi(v) ) is a disjunction</td>
</tr>
</tbody>
</table>

Constraints Solved and Types Inferred
- Lower and upper bounds for each type
  - \( \Phi(v) \) is a disjunction over a given function
  - Values must be a valid type for the function
Constraint Solving is Complicated

- Decomposition Rules: “Before solving a subtype relation constraint, we try to simplify it by applying decompositional rules.”
- Conjunctive constraints (C1 \& C2): C1 and C2 separated and added to constraints.
- Disjunctive constraints (C1 \lor C2): Produces a solution for each constraint separately.
- Composition Rules: Sometimes, constraints can be simplified together to create a stronger constraint. This especially saves resources when we can do it with disjunctive constraints.

Complete Example

General registers, 32 bits:
- eax, edx
Index and pointers:
- ebp, esp

http://www.eecg.toronto.edu/~amza/
www.mindsec.com/files/x86regs.html

Results

- Used several metrics to measure their results:
  - Distance: Distance between two types in the lattice
  - Conservativeness: A measure how conservative a type is
  - Precision: How close a inferred type to original type
- Very promising results!
  - TIE was more precise than Hex-rays
  - TIE was more precise and conservative than REWARDS.

Results Graph

Critique

- Uses a lot of knowledge from class!
- The DCGA algorithm wasn’t explained too much
- Uses known functions to type variables:
  - Wouldn’t it be nice to have a list
- Is it possible to correct user errors?
- How would this be different if it had C’s assembly code instead of binary code.
Evaluation of Results

- Converts lower bound to C type
  - Would like to look at other heuristics
- Uses the output’s upper and lower bound for each variable
  - Used to measure conservativeness
- Doesn’t explain how
- However the results were very nice
- They tested in many different ways

Future Work

- Cited 151 times.
- Try applying this to other languages.
  - Will have to deal with data structures more complicated than struct.
  - As mentioned before try direct assembly code.
  - Note differences between different compilations of code.
  - As mentioned before, try some different heuristics out.

Questions?

Thank you