The Type Inference on Executables (TIE) System

An analysis of “TIE: Principled Reverse Engineering of Types in Binary Programs” By JongHyup Lee, Thanassis Avgerinos, and David Brumley, Carnegie Mellon University, NDSS, 2011

Presentation by Kevin Sanita, April 16, 2020
Schedule

- Background
- Solution
- Testing
- Conclusion
- Questions
Background
Summary

• Description of Type Inference on Executables (TIE) system

• Reverse-engineers binary code to get variables and their type

• Can be applied to static or dynamic analysis

• Tested on 87 programs and is more conservative and precise than previous solutions
Motivation

• Recurring step in security
  • Fuzzing
  • COTS software understanding
  • Binary program analysis

• Two main tasks
  • Variable Recovery
  • Type Recovery
Current Solutions

- Dynamic approach
  - Poor program coverage
  - Cannot handle control flow
  - REWARDS

- Static approach
  - Potentially incorrect results
  - IDA Pro - Hex Rays
Solution
TIE Concept

• Principled inference-based approach

• Can handle control flow

• Works with both static and dynamic analysis
How It Works - Overview

- Lifts to Binary Intermediate Language (BIL) using Binary Analysis Program (BAP)

- Algorithm takes in BIL, recovers variables, then reconstructs types
How It Works - Binary to BIL

- Static analysis
  - Identifies functions using heuristics
  - Outputs BIL
- Dynamic analysis
  - Outputs list of instructions along path
  - Instructions are converted to BIL
How It Works - Binary to BIL (Static)

- BAP disassembles binary into assembly instructions
- Implements a linear sweep disassembly
  - Other disassemblers possible
How It Works - Binary to BIL (Dynamic)

• Executes program within emulator to produce instruction trace

• Two options in BAP
  • PIN-based implementation
  • TEMU (whole-system emulator)
How It Works - Variable Recovery

- Input is BIL
  - Converts to static assignment form (SSA)

- Uses DVSA to infer variable locations from memory addresses
  - Builds on value set analysis (VSA)
How It Works - Type Reconstruction

- Step 1: Assign Type Variables
- Step 2: Constraint Generation
  - Based on how the variables are used
- Step 3: Constraint Solving
  - Solves for most precise but conservative types
  - Uses TIE type system, then translates into C
• Heavily-subtyped

• Lattice-style

• Uses upper and lower bounds
  • “How much does the binary tell us”

• Includes data, addresses, intersection and union types, etc.
How It Works - Type Reconstruction (Step 1)

- Creates type variables for every term
  - \(\text{Tau}_x\) for variable \(x\), \(v\) for integer \(v\), etc.

- Unary and binary operations handle multiple cases
How It Works - Type Reconstruction (Step 1.5)

- Create context F for known functions
  - Matches arguments and return value

- Also handles unknown parameter types
  - Adds calls to context F
  - Generates constraints for functions
How It Works - Type Reconstruction (Step 2)

- Uses set of constraint patterns
- Uses pointers as hint of structural data
How It Works - Type Reconstruction (Step 3)

- Extended unification algorithm
- Uses working set for current state
How It Works - Output

- Returns upper and lower bound of type for all variables
- Structs are just handled as pointers
- Combines members for same address as union type of them
Testing
Overview

• Tested 87 programs from coreutils

• Compared against Hex-Rays and REWARDS

• Some issues with typing
  • Hex-Rays and REWARDS restricted to c-types
  • TIE types had to be converted
Static Analysis (vs. Hex-Rays)

• Precision
  • Intra-procedural: +28%
  • Inter-procedural: +38%

• Conservatism (on structural types)
  • 90% on TIE vs. <45% on Hex-Rays

• Overall TIE is more conservative and precise
Dynamic Analysis (vs. REWARDS)

• Less coverage than static analysis

• REWARDS initially mislabeled types
  • Some edits were made to make it more precise

• TIE is more precise and conservative in all cases
<table>
<thead>
<tr>
<th>Program</th>
<th>Coverage</th>
<th>Conserv.</th>
<th>Distance</th>
<th>Conserv.</th>
<th>Distance</th>
<th>Conserv.</th>
<th>Distance</th>
<th>Conserv.</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>chroot</td>
<td>9.23 %</td>
<td>1.0</td>
<td>0.88</td>
<td>0.56</td>
<td>2.3</td>
<td>1.0</td>
<td>1.42</td>
<td>0.87</td>
<td>1.76</td>
</tr>
<tr>
<td>df</td>
<td>6.9 %</td>
<td>1.0</td>
<td>1.39</td>
<td>0.46</td>
<td>2.73</td>
<td>1.0</td>
<td>1.62</td>
<td>0.942</td>
<td>1.42</td>
</tr>
<tr>
<td>groups</td>
<td>11.11 %</td>
<td>1.0</td>
<td>1.47</td>
<td>0.48</td>
<td>2.22</td>
<td>0.96</td>
<td>1.7</td>
<td>0.93</td>
<td>1.52</td>
</tr>
<tr>
<td>hostid</td>
<td>6.89 %</td>
<td>1.0</td>
<td>0.92</td>
<td>0.71</td>
<td>1.82</td>
<td>1.0</td>
<td>1.25</td>
<td>0.97</td>
<td>1.63</td>
</tr>
<tr>
<td>users</td>
<td>9.52 %</td>
<td>1.0</td>
<td>1.09</td>
<td>0.73</td>
<td>1.87</td>
<td>0.95</td>
<td>1.64</td>
<td>0.97</td>
<td>1.51</td>
</tr>
<tr>
<td>average</td>
<td>8.73 %</td>
<td>1.0</td>
<td>1.15</td>
<td>0.59</td>
<td>2.19</td>
<td>0.98</td>
<td>1.53</td>
<td>0.93</td>
<td>1.57</td>
</tr>
</tbody>
</table>

Table 2. TIE vs. REWARDS with dynamic analysis. TIE is both more precise and more conservative.
Conclusion
Conclusion

• TIE is an effective system for reverse-engineering binary code

• Core of system is the novel type reconstruction algorithm

• Is more conservative and precise than research (REWARDS) and leading industry product (IDA-Pro Hex-Rays)
Thoughts

• Seems like a significant improvement

• Like the type-lattice system and reduction in guessing

• Struggling to find impactful implementations
  • 183 citations
  • If it’s such an improvement, why?
Questions
Binary Intermediate Language

\[
\begin{align*}
\text{program} & ::= (\text{label stmt}^*) \text{ (i.e., functions)} \\
\text{stmt} & ::= \text{var} \colonequals \text{exp} \mid \text{goto exp} \mid \text{if exp then goto exp else goto exp} \mid \text{return} \\
& \quad \mid \text{halt exp} \mid \text{assert exp} \mid \text{label label.kind} \mid \text{call exp with argument ret var} \mid \text{special string} \\
\text{exp} & ::= \text{exp} \diamond_b \text{exp} \mid \diamond_u \text{exp} \mid \text{var} \mid \text{lab string} \mid \text{integer} \mid \text{load(\text{exp, exp, exp, } \tau_{\text{reg}})} \mid \text{store(\text{exp, exp, exp, } \tau_{\text{reg}})} \\
& \quad \mid \text{cast(\text{cast.kind, } \tau_{\text{reg}}, \text{exp})} \mid \Phi(\text{var}^*) \\
\text{label.kind} & ::= \text{integer} \mid \text{string} \\
\text{cast.kind} & ::= \text{unsigned} \mid \text{signed} \mid \text{high} \mid \text{low} \\
\text{var} & ::= (\text{string, id}_v, \tau) \\
\diamond_b & ::= +, -, \ast, /, \div, \ldots \\
\diamond_u & ::= - \text{ (unary minus)}, \sim \text{ (bit-wise not)} \\
\text{memory} & ::= \{ \text{integer} \rightarrow \text{integer}, \text{integer} \rightarrow \text{integer}, \ldots \} (\tau_{\text{mem}}) \\
\text{argument} & ::= (\text{var})^+ \\
\tau & ::= \tau_{\text{reg}} \mid \tau_{\text{mem}} \\
\tau_{\text{mem}} & ::= \text{mem.t(}\tau_{\text{reg}}) \\
\tau_{\text{reg}} & ::= \text{reg1.t, reg8.t, reg16.t, reg32.t, reg64.t}
\end{align*}
\]
TIE types and constraints

\[
T := \tau_{\text{data}} \mid \tau_{\text{fun}} \mid \top \mid \bot \mid T \land T \mid T \lor T \mid T \to T
\]

\[
\tau_{\text{data}} := \tau_{\text{base}} \mid \tau_{\text{mem}}
\]

\[
\tau_{\text{base}} := \tau_{\text{reg}} \mid \tau_{\text{refined}}
\]

\[
\tau_{\text{reg}} := \text{reg1}_t \mid \text{reg8}_t \mid \text{reg16}_t \mid \text{reg32}_t
\]

\[
\tau_{\text{refined}} := \text{numn}_t \mid \text{uintn}_t \mid \text{intn}_t \ (n = 8, 16, 32) \mid \text{ptr}(T) \mid \text{code}_t
\]

\[
\tau_{\text{mem}} := \{\forall \text{ addresses } i | l_i : T_i\}
\]

\[
\tau_R := \{\text{var}_1 : T_1, \ldots, \text{var}_n : T_n\}
\]

\[
\tau_{\text{fun}} := \tau_{\text{mem}} \to \tau_R \to T
\]

\[
\text{constraints} := T = T \mid T <: T
\]

\[
\mid \text{constraints} \land \text{constraints}
\]

\[
\mid \text{constraints} \lor \text{constraints}
\]
# Type Constraint Rules

<table>
<thead>
<tr>
<th>Statement</th>
<th>Generated constrains</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := e )</td>
<td>( \tau_x = \tau_e )</td>
</tr>
<tr>
<td>( \text{goto } e )</td>
<td>( \tau_e = \text{ptr(code.t)} )</td>
</tr>
<tr>
<td>( \text{if } e \text{ then goto } e_t \text{ else goto } e_f )</td>
<td>( \tau_e = \text{reg1.t} \land \tau_t = \text{ptr(code.t)} \land \tau_f = \text{ptr(code.t)} )</td>
</tr>
</tbody>
</table>
| \( \text{call } f \text{ with } m v^* \text{ ret } r \) | \( \tau'_m = \tau_{mf} \land \bigwedge_v (\tau_v = \tau_{vf} . [v]) \land \tau_r = \tau_{rf} \)  
  (where \( F \vdash f : \tau_{mf} \rightarrow \tau_{vf} \rightarrow \tau_{rf} \), \( \tau'_m = \text{update}(\tau_m) \)) |
## Type Constraint Rules

<table>
<thead>
<tr>
<th>Expression</th>
<th>Generated constraint for term with type variable $\tau$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$ (variable)</td>
<td>$\tau_x$</td>
</tr>
<tr>
<td>$v$ (integer)</td>
<td>$\tau_v$</td>
</tr>
<tr>
<td>$-n \cdot e$ (unary neg)</td>
<td>$\tau_e &lt;: \text{intn}<em>.t \land \tau :&gt; \text{intn}</em>.t$</td>
</tr>
</tbody>
</table>
| $e_1 + 32 e_2$ | $(\tau_{e_1} <: T_\gamma \land \tau_{e_2} <: T_\gamma \land \tau :> T_\gamma \land \tau <: \text{num32}_.t)$  
$\lor (\tau_{e_1} <: \text{ptr}(T_\alpha) \land \tau_{e_2} <: \text{num32}_.t \land \tau :> \text{ptr}(T_\beta))$  
$\lor (\tau_{e_1} <: \text{num32}_.t \land \tau_{e_2} <: \text{ptr}(T_\alpha) \land \tau :> \text{ptr}(T_\beta))$ |
| $e_1 + n \cdot e_2$ | $\tau_{e_1} <: T_\gamma \land \tau_{e_2} <: T_\gamma \land \tau :> T_\gamma \land \tau <: \text{numn}_.t$ |
| $\sim_n e$ | $\tau_e <: \text{uintn}_.t \land \tau :> \text{uintn}_.t$ |
| $e_1 < s_n e_2$ | $\tau_{e_1} <: \text{intn}_.t \land \tau_{e_2} <: \text{intn}_.t \land \tau :> \text{regl}_.t$ |
| load($m, i, d, \text{regn}_.t$) | $\tau_i = \text{ptr}(\tau_m[i]) \land \tau = \tau_m[i] \land \tau <: \text{regn}_.t$ |
| store($m, i, v, d, \text{regn}_.t$) | $\tau_i = \text{ptr}(\tau_v) \land \tau = \tau_m(i : \tau_v) \land t[i] <: \text{regn}_.t$ |
| $\Phi(e_1, \ldots, e_n)$ | $\tau' <: \tau_{e_1} \cap \ldots \cap \tau_{e_n}$ |
Figure 10. Summary of precision (left) and conservativeness (right)
(a) Conservativeness for structural types.

(b) Distance for structural types.

Figure 11. Conservativeness and distance for structural types