Disassembling and Control Flow Reconstruction

- Code vs. data separation problem while disassembling binaries
- Resolve indirect jumps (e.g. from switch construct, function pointers)
- """" value analysis with disassembling

RREIL (Relational Reverse Engineering Intermediate Language)

Example loop in C:
```c
for (int i = -1; i > -100; i--)
{
    /* body */
}
```

RREIL translation:
```c
mov rax, -1
mov rax, 0
sub rax, 100
cmp rax, 0
jg 0
```

x86-64 translation:
```c
mov eax, 0xffffffff
```

- use small architecture independent, intermediate language for analysis
- introduce virtual flags `[I, L, LE]` to associate comparison operations
  with test used at the jump (precision analysis discards unneeded flags)
- use test and relational information to infer bounds for variable values

The Analyzer Structure

- fixpoint engine is the driver for the disassembler and the domain hierarchy
- disassembles a chunk of bytes and translates them to RREIL instructions
- RREIL instructions are processed by the domain hierarchy (down channel)
- indirect jumps are resolved by querying the hierarchy (up channel)
- fixpoint continues disassembly at jump target

Memory Domain

From operations on registers/memory \( m \in \mathbb{M} \) to operations on numeric variables \( x \in \mathbb{X} \).

Consider x86-64: \( \text{mov} \ rax, 0xffffffff \)

- 32-bit `mov` implicitly clears upper 32 bits of `rax`
- cannot store `-1` in numeric domain since then `rax=0xffffffffffffff`

- tracking `rax` as one numeric variable leads to precision loss
- `idea`: use fields with size and offset in RREIL

Dealing with Finite Integer Arithmetic

We have a special view on numeric information:

- we store e.g. \( a1 \in \mathbb{Z}^{[254, 257]} \) (\( a1 \) is 8 bits big)
- this means:
  ```
  [254] = 11111110
  [255] = 11111111
  [256] = 00000000
  [257] = 00000001
  ```
- which is also \( [254, 255] \cup [0, 1] = [0, 255] \)
- we call this conversion *wrapping*

The finite domains associate a bit-size with each \( x \in \mathbb{X} \).

- `idea`: wrap operand \( i \) before each operation; wrapping is no-op if \( i \in [-2^{31}, 2^{31} - 1] \)
- `problem`: precision loss;
  ```
  add, sub carry no sign information (signedness-agnostic)
  ```
- `idea`: only wrap when unavoidable, e.g. before executing the test
  ```
  i > -100
  ```
- `problem`: \( i \in [-1, -1], [-2, -1], [-3, -1], [-4, -1] \ldots \) is inferred during fixpoint computation; *wrapping* applied to \( i \overset{\text{to}}{\rightarrow} [-\infty, -1] \)
- wrapping of widened value \([-\infty, -1]\) gives \([-2^{31}, 2^{31} - 1]\)
- cannot infer that \( i = \text{negative} \)

Narrowing Domain

Avoiding precision loss incurred by widening:

- have a *narrowing* domain that tracks all tests that don’t affect the state (redundant tests)
- ````: the test \( i > -100 \) is stored when analyzing the loop
- after widening \( i \) to \([-\infty, -1]\) apply all stored tests
- ````: \( i \in [-99, -1] \) follows
- wrapping to positive values avoided

Numeric Domain

Numeric domains map variables \( x \in \mathbb{X} \) to a subset of \( \mathbb{Z} \).

- the affine tracks equalities \( c1 \cdot x1 + c2 \cdot x2 \)
- `````: no need to store \( x2 \) in child domains; some linear assignments need not be propagated to child
- maintains equalities between memory and register locations when inferring numerical values
- the interval domain maps \( x_k \) to \([l_k, u_k]\)

Tool Implementation

The mentioned techniques have been implemented as an analyzer tool:
- able to disassemble Linux (ELF) and Windows (PE) binaries
- displays RREIL and native Control Flow Graphs for each discovered procedure
- shows warnings that occurred during the program analysis

Publications

