A Crash Course In Compiler Certification

Language-based Security

September 25, 2017

A Famous Compiler Attack

• Ken Thompson’s 1984 Turing Award Speech
  – awarded (in part) for inventing B language
  – “Reflections on Trusting Trust”
  – described an experimental “compiler Trojan”

• Compiler Trojan Attack
  – compiler modified to embed a Trojan horse in all compiled object code
  – compiler then recompiled
  – Result: Trojan horse no longer visible in ANY source code, yet infects all binaries!
A Modern Compiler Trojan

Apple scrambles after 40 malicious "XcodeGhost" apps haunt App Store
Outbreak may have caused hundreds of millions of people to download malicious apps.

by Dan Goodin - Sep 21, 2015 9:40am CDT
Anatomy of a Compiler

Example: GCC

FRONT END
- C Parser
- C++ Parser
- Java Parser
- VB Parser
- F95 Parser

MIDDLE “END”
- GENERIC
- GIMPLE
- GIMPLE - SSA
- Various Optimizations

BACK END
- RTL
- x86
- PowerPC
- x64
What is “Compiler Correctness”?

• Each source language has a well-defined semantics (we hope)
  – Large-step: \( <c, \sigma> \downarrow \sigma' \)
  – Small-step: \( <c, \sigma> \rightarrow <c', \sigma'> \)
  – Denotational: \( C[[c]]\sigma = \sigma' \)
• Each object language has a well-defined binary semantics
  – See the Intel x86 architecture manual, for example
• High-level idea: Semantic Transparency
  – Correct compilers preserve program behavior.
  – Equivalently: Each compiler stage is semantically transparent.
• Problem:
  – What is “equivalent behavior”?
  – Different domain models for source vs. object language!
Labeled Transition Systems

• Project out “observable” events

\[ \langle c, \sigma \rangle \xrightarrow{\tau} \langle c', \sigma' \rangle \]

• Correct compilers preserve sequences of observable events (traces)

• Still many challenges:
  – What about non-determinism?
  – What about (unsafe) source languages for which some operations are “undefined”?
  – What about operations that admit several different “correct” behaviors?
Case Study: Register Allocation

• Map program variables (actually intermediate language pseudo-registers) to machine registers
  – Use registers as much as possible (fast)
  – Avoid memory storage (slow)

• Example:

  **Inputs:** b, c, d
  a := c + d
  e := a + b
  f := e − 1

  **Outputs:** f

  **Inputs:** r₁, r₂, r₃
  r₂ := r₂ + r₃
  r₁ := r₂ + r₁
  r₁ := r₁ − 1

  **Outputs:** r₁
Algorithm for Register Allocation

• Famous result due to Chaitin (IBM, 1980)
  – **Liveness**: x is *live* if it contains a value that is later used in at least one ensuing control-flow path
  – **Interference**: x and y *interfere* if they are ever both live
  – **Interference Graph**:
    • nodes are variables
    • edges between variables that interfere
  – register allocation = *k*-coloring problem!
    • *k* = # machine registers = # colors
    • color all nodes with no edge having endpoints of same color
Step 1: Compute Liveness

• Construct control-flow graph
• Working backwards along edge CFG edge...
  – remove assigned variables from live-set
  – then add read variables to live-set
• Iterate along CFG until a fixed point is reached
  – Question: How do we know this process eventually terminates?
\[ a := b + c \]
\[ d := -a \]
\[ e := d + f \]
\[ f := 2 \times e \]
\[ b := d + e \]
\[ e := e - 1 \]
\[ b := f + c \]
\[ \text{return } b \]
Step 2: Construct Interference Graph

- Graph with one node per variable
- Insert edges between variables that appear together in any live-set
a := b + c

d := -a

e := d + f

f := 2 * e

b := d + e

e := e - 1

b := f + c

return b
Step 3: Color Nodes

• Find a k-coloring of the nodes
  – no edge’s endpoints may have same color
  – # of colors (k) = # of machine registers

• k-coloring is NP-complete
  – no known algorithm better than brute search
  – very good heuristic algorithms though

![Graph Diagram]
Step 3: Color Nodes

• Find a k-coloring of the nodes
  – no edge’s endpoints may have same color
  – # of colors (k) = # of machine registers

• k-coloring is NP-complete
  – no known algorithm better than brute search
  – very good heuristic algorithms though
How to Prove Correctness of a Register Allocator?

• Correctness:
  – Result is a register mapping $R$ satisfying $\text{val}(v) = \text{val}(R(v))$ for all possible traces

• Proof approach #1:
  – Prove that all 3 steps always yield such an $R$
  – Proving the k-coloring part is hard because the heuristics can be quite complex

• Proof approach #2:
  – Add a small validator to the compiler that double-checks the $R$ for correctness.
  – Prove that the validator is correct.
  – Much easier (k-coloring is NP, so $R$ is easier to check than to produce!)
One Drawback of Validator Approach

- Proving validator correct does NOT prove that the algorithm whose output it checks is always correct!
  - If algorithm is wrong, validator catches error and stops.
  - Compilation fails with error.
- Why is this okay?
  - From a security perspective, our primary concern is preventing compiler from *silently* generating *incorrect* code.
  - Failing with an error has less severe consequences.
  - Can be addressed with unit testing, quality assurance processes, etc.
Next Time

• Machine Code Validation
  – How can we formally prove things about raw machine code programs?