Compiler Security

Tech Talk #03

Charles Averill

Introduction to Compiler Design
The University of Texas at Dallas

Spring 2023

What is it?

- No unsafe code generation
  - Open-source development
  - Preventing unsafe behavior from compiling
  - Prevent safe behavior from optimizing into unsafe behavior
- Semantically equivalent code generation
  - PL Theory :)

Open-source compiler vulnerabilities

- GCC vulns
- LLVM vulns
- Ken Thompson's compiler backdoor (thought experiment)
- (buffer overflow, uncontrolled recursion, RNG manipulation)

Semantics

What does this language do?

```
gleeble: glorp
       | gloozle
       | gleeble "+" gleeble
       | "glonzagle" gleeble
```

Semantics

You have no idea! We need to describe, mathematically, how a programming language works. Some groundwork:

- "Store": σ - a function from variable names to values
- "Typing context": Γ - a function from variable names to types
- "Judgement": $\langle x, \sigma, \Gamma \rangle \Downarrow \sigma', \Gamma'$ - "When the command x runs with store σ and typing context Γ, it results in a new program state defined by the new store σ' and the new typing context Γ'
  - Judgements can have conditions: $\dfrac{A \quad B}{\langle c, \sigma, \Gamma \rangle \Downarrow \sigma', \Gamma'}$
  - "This judgement holds if some propositions A and B hold"

$$\frac{\Gamma \vdash 5 : int \quad \Gamma' := \Gamma[x := int] \quad \sigma' := \sigma[x := 6]}{\langle \texttt{int x = 5;}, \sigma, \Gamma \rangle \Downarrow \sigma', \Gamma'}$$

Simply-Typed Lambda Calculus Semantics

Static Semantics of $\lambda^{\rightarrow}$

$$\Gamma \vdash n : \mathit{int} \tag{9}$$

$$\Gamma \vdash v : \Gamma(v) \tag{10}$$

$$\frac{\Gamma[v \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash (\lambda v{:}\tau_1 \,.\, e) : \tau_1 \rightarrow \tau_2} \tag{11}$$

$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'} \tag{12}$$

$$\Gamma \vdash \mathtt{true} : \mathit{bool} \tag{13}$$

$$\Gamma \vdash \mathtt{false} : \mathit{bool} \tag{14}$$

$$\frac{\Gamma \vdash e_1 : \mathit{int} \qquad \Gamma \vdash e_2 : \mathit{int}}{\Gamma \vdash e_1 \ \mathit{aop} \ e_2 : \mathit{int}} \tag{15}$$

$$\frac{\Gamma \vdash e_1 : \mathit{bool} \qquad \Gamma \vdash e_2 : \mathit{bool}}{\Gamma \vdash e_1 \ \mathit{bop} \ e_2 : \mathit{bool}} \tag{16}$$

$$\frac{\Gamma \vdash e_1 : \mathit{int} \qquad \Gamma \vdash e_2 : \mathit{int}}{\Gamma \vdash e_1 \ \mathit{cmp} \ e_2 : \mathit{bool}} \tag{17}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \tag{18}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2 \qquad i \in \{1, 2\}}{\Gamma \vdash \pi_i e : \tau_i} \tag{19}$$

$$\Gamma \vdash () : \mathit{unit} \tag{20}$$

$$\frac{\Gamma \vdash e : \tau_i \qquad i \in \{1, 2\}}{\Gamma \vdash \mathtt{in}_i^{\tau_1 + \tau_2} e : \tau_1 + \tau_2} \tag{21}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma[v_1 \mapsto \tau_1] \vdash e_1 : \tau \qquad \Gamma[v_2 \mapsto \tau_2] \vdash e_2 : \tau}{\Gamma \vdash (\mathtt{case}\ e\ \mathtt{of}\ \mathtt{in}_1(v_1) \rightarrow e_1 \mid \mathtt{in}_2(v_2) \rightarrow e_2) : \tau} \tag{22}$$

So What?

If we have a model of input C semantics, and we have a model of output ASM semantics, we can mathematically determine if the input of our compiler does the same thing as the output!

Sort of...

There are some issues:

- C has typing contexts (sort of) and stores (yes)
- Assemble does not have types (sort of) has stores but limited variables (registers)

We've invented concepts like register mapping to abstract the idea of stores to assembly code. Question: how do we map typing contexts?

Reality

Answer: We don't!

Semantic equivalence states:

$$\begin{aligned} \forall B &: \text{ program behaviors,} \\ S &: \text{ source programs,} \\ C &: \text{ compiled programs,} \\ S \Downarrow B &\iff C \Downarrow B. \end{aligned}$$

This is way too hard to deal with. No realistic compiler ever written for a production language to a different production language satisfies this condition.

Reality

For example, C semantics do not specify the order of operations between addition and subtraction. Therefore, C compilers choose an order. But, that means that the C semantics allow for more program behaviors than the output assembly semantics. So how can the condition hold (it's bidirectional!)?

Another example: `if (5 < 3) printf("%d\n", 1 / 0);`

The compiler can optimize out the unsafe division by zero because the condition is trivially never true. So the division by zero might not throw a compiler error. Therefore, if the source program can semantically "go wrong," the compiled program can still not "go wrong."

Useful semantic equivalence states:

$$\forall BSC, B \text{ not wrong } \implies S \Downarrow B \iff C \Downarrow B.$$