Basic Program Analysis

Dongdong She

*some slides are borrowed from Baishakhi Ray and Ras Bodik
Our Goal

Program analyzer must be able to understand program properties (e.g., can a variable be NULL at a particular program point?)

Must perform control and data flow analysis

Source code → Program Analyzer → Security bugs
Do we need to implement control and data flow analysis from scratch?

• Most modern compilers already perform several types of such analysis for code optimization
  • We can hook into different layers of analysis and customize them
  • We still need to understand the details

• LLVM (http://llvm.org/) is a highly customizable and modular compiler framework
  • Users can write LLVM passes to perform different types of analysis
  • Clang static analyzer can find several types of bugs
  • Can instrument code for dynamic analysis
Compiler Overview

- **Abstract Syntax Tree**: Source code parsed to produce AST
- **Control Flow Graph**: AST is transformed to CFG
- **Data Flow Analysis**: operates on CFG
The Structure of a Compiler

1. Source code (stream of characters)
   - Scanner
   - Stream of tokens
   - Parser
   - Abstract Syntax Tree (AST)
   - Checker
   - AST with annotations (types, declarations)
   - Code gen
   - Machine/byte code
Syntactic Analysis

• **Input**: sequence of tokens from scanner
• **Output**: abstract syntax tree

• Actually,
  • parser first builds a **parse tree**
  • AST is then built by translating the parse tree
  • parse tree rarely built explicitly; only determined by, say, how parser pushes stuff to stack
Example

• Source Code
  \[4 \times (2 + 3)\]

• Parser input
  \texttt{NUM(4) TIMES LPAR NUM(2) PLUS NUM(3) RPAR}

• Parser output (AST):
  \[
  \begin{array}{c}
  * \\
  + \\
  \text{NUM(4)} \\
  \text{NUM(2)} \quad \text{NUM(3)}
  \end{array}
  \]
Parse tree for the example: $4 \times (2 + 3)$

leaves are tokens
Another example

• Source Code

    if (x == y) { a=1; }

• Parser input

    IF LPAR ID EQ ID RPAR LBR ID AS INT SEMI RBR

• Parser output (AST):

    IF-THEN

         ==  =

    ID    ID   ID    INT
Parse tree for example: if (x==y) {a=1;}

leaves are tokens
Parse Tree

• Representation of grammars in a tree-like form.

A parse tree pictorially shows how the start symbol of a grammar derives a string in the language. ... Dragon Book

• Is a one-to-one mapping from the grammar to a tree-form.
C Statement: \texttt{return a + 2}

A very formal representation that strictly shows how the parser understands the statement \texttt{return a + 2};
Abstract Syntax Tree (AST)

- Simplified syntactic representations of the source code, and they're most often expressed by the data structures of the language used for implementation

ASTs differ from parse trees because superficial distinctions of form, unimportant for translation, do not appear in syntax trees... … Dragon Book

- Without showing the whole syntactic clutter, represents the parsed string in a structured way, discarding all information that may be important for parsing the string, but isn't needed for analyzing it.
C Statement: return a + 2

AST
Disadvantages of ASTs

• AST has many similar forms
  • E.g., for, while, repeat...until
  • E.g., if, ?, switch

• Expressions in AST may be complex, nested
  • \((x \times y) + (z > 5 \ ? \ 12 \times z : z + 20)\)

• Want simpler representation for analysis
  • ...at least, for dataflow analysis
Parse Tree vs. AST
Control Flow Graph & Analysis
Representing Control Flow

High-level representation
– Control flow is implicit in an AST

Low-level representation:
– Use a Control-flow graph (CFG)
  – Nodes represent statements (low-level linear IR)
  – Edges represent explicit flow of control
What Is Control-Flow Analysis?

1. \( a := 0 \)
2. \( b := a \times b \)
3. \( L1: \ c := b/d \)
   - \( \text{if } c < x \text{ goto } L2 \)
4. \( e := b / c \)
5. \( f := e + 1 \)
6. \( L2: \ g := f \)
7. \( h := t - g \)
   - \( \text{if } e > 0 \text{ goto } L3 \)
8. \( \text{goto } L1 \)
9. \( L3: \text{ return} \)
A **basic block** is a sequence of straight line code that can be entered only at the beginning and exited only at the end.

### Building basic blocks

- **Identify leaders**
  - The first instruction in a procedure, or
  - The target of any branch, or
  - An instruction immediately following a branch (implicit target)
**Basic Block Example**

```
1. a := 0
2. b := a * b
3. L1: c := b/d
   if c < x goto L2
4. e := b / c
5. f := e + 1
6. L2: g := f
7. h := t - g
   if e > 0 goto L3
8. goto L1
9. L3: return

Leaders?
- {1, 3, 5, 7, 10, 11}

Blocks?
- {1, 2}
- {3, 4}
- {5, 6}
- {7, 8, 9}
- {10}
- {11}
```

Adopted From U Penn CIS 570: Modern Programming Language Implementation (Autumn 2006)
Building a CFG From Basic Block

Construction
– Each CFG node represents a basic block
– There is an edge from node i to j if
  – Last statement of block i branches to the first statement of j, or
  – Block i does not end with an unconditional branch and is immediately followed in program order by block j (fall through)
backedges indicate that we might need to traverse the CFG more than once for data flow analysis
Dominators

- **d dom i** if all paths from entry to node i include d

- **Strict Dominator (d sd dom i)**
  - If d dom i, but d != i

- **Immediate dominator (a idom b)**
  - a sd dom b and there does not exist any node c such that c != a, c != b, a dom c, c dom b

- **Post dominator (p pdom i)**
  - If every possible path from i to exit includes p
AST vs. CFG

CFGs are much simpler than ASTs
- Fewer forms, less redundancy, only simple expressions

So for AST
- Easier to report error + other messages
- Easier to explain to programmer
- Easier to unpars to produce readable code