Developing the Clang Static Analyzer

Artem Dergachev, Apple
Clang Static Analyzer

- Finds bugs at compile time – by inspecting your source code
- Bugs it finds are more sophisticated than warnings or Clang-Tidy
int foo(int x) {
    int y = x;
    if (y == 0)
        return 24 / x;
    return 0;
}
Clang Static Analyzer

- Natural!
- Mimics normal program execution
- Easy to understand why it “thinks” there is a bug
- Takes all source code information into account
- Explains bugs in terms of the source code
Clang Static Analyzer

• Natural!
• Deals with a lot of open problems

• Researchy!
• People publish articles, defend BS/MS/Ph.D. theses on it
• Fully Open Source – lives in Clang repo
Clang Static Analyzer

• Natural!
• Researchy!
• Practical!

• Used in industry
• Shipped with IDEs
• Finds bugs in your code before your users do!
Clang Static Analyzer

• Natural!
• Researchy!
• Practical!
• Extensible!

• Finds over 50 kinds of bugs!
  - Memory leaks
  - Null dereferences
  - Use-after-free
  - Use-after-move
  - …

• “Building a Checker in 24 hours” – LLVM DevMtg 2012
  https://youtu.be/kdxIsP5QVPw
Clang Static Analyzer

- Natural!
- Researchy!
- Practical!
- Extensible!
- Spooky!

💰 Symbolic Execution
💰 Dead, Undead, Zombie and Schrödinger Symbols, The Reaper
💰 Body Farms
Clang Static Analyzer

- Natural!
- Researchy!
- Practical!
- Extensible!
- Spooky!
- Exciting!
Plan For Today!

• Algorithms and Data Structures of the Static Analyzer

• How to Fix a Static Analyzer Bug in 24 minutes
Algorithms and Data Structures of the Static Analyzer

Abstract Syntax Tree → Control Flow Graph

Exploded Graph ← Path Diagnostics

Plain Source Code ←
AST: How Compiler Sees Your Code

- **Nodes**: statements, declarations, types – annotated and cross-referenced
- **Edges**: “is-part-of” relation
AST: How Compiler Sees Your Code

- **Nodes**: statements, declarations, types – annotated and cross-referenced
- **Edges**: “is-part-of” relation
CFG: Order in which Statements are Executed

- **Nodes**: usually AST statements
- **Edges**: “executed-after” relation

![Diagram showing the execution order of statements](image)
CFG: Order in which Statements are Executed

- **Nodes**: usually AST statements
- **Edges**: “executed-after” relation
Program Points

Stmt_1

Stmt_2

Point

Stmt

Point_1

Point_2
Exploded Graph: Paths Through CFG

- **Nodes**: $(\text{Point, State})$ pairs
  - **Program Point**: A point between statements (usually)
  - **Program State**: A record of effects of statements evaluated so far

- **Edges**: An edge from $(\text{Point}_1, \text{State}_1)$ to $(\text{Point}_2, \text{State}_2)$ means that the statement between $\text{Point}_1$ and $\text{Point}_2$ updates $\text{State}_1$ to $\text{State}_2$
Exploded Graph Edges

\[ \text{Node}_1 \quad \text{State}_1 \quad \text{Stmt} \quad \text{Point}_1 \]

\[ \text{Node}_2 \quad \text{State}_2 \quad \text{Point}_2 \]

\[ \text{Edge}_{12} \]

- $\text{Node}_1$ to $\text{Node}_2$ via $\text{Stmt}$
Effects of Assignments: Store

Statement: $x = 7$

Program State: Nothing Yet!

Program State: Store: $x \rightarrow 7$
Values of Expressions: Environment

Program State: Store:
\[ x \rightarrow 7 \]

Statement:
\[ x + 5; \]

Program State: Store:
\[ x \rightarrow 7 \]
Exprs:
\[ x + 5 \rightarrow 12 \]
Focus on One Operation at a Time

Program State:
Exprs:
\[ x + 5 \rightarrow 12 \]

Statement:
\[ \frac{x + 5}{2}; \]

Program State:
Exprs:
\[ \frac{x + 5}{2} \rightarrow 6 \]
What If It’s Not In The Store?

Program State: Nothing Yet!

Statement: $x$

Program State: Exprs:
$x \rightarrow \text{reg}_0\langle\text{int } x\rangle$

// example:
int foo(int x) {
    return x;
}
Effects of Branches: Constraints

Program State:
Exprs:
\( x \rightarrow \text{reg}_0<\text{int } x> \)

Statement:
\[
\text{if } (x > 5) \ldots
\]

Program State:
Ranges:
\[
\text{reg}_0<\text{int } x> > 5
\]

Program State:
Ranges:
\[
\text{reg}_0<\text{int } x> \leq 5
\]
Symbolic Execution Recipe

• Just execute the program as you normally would
• Don’t know the value? – Denote it with a symbol
• Branch depends on a symbol? – Split up, record constraints
• Don’t explore paths on which constraints contradict each other
Demo:
How to Fix a Static Analyzer Bug in 24 minutes!
Summary!

- Static Analyzer finds bugs by exploring sequences of events that may occur during the execution of the program.

- You can understand and study the internal logic of the static analyzer by looking at exploded graph dumps and setting conditional breaks on individual nodes.

- Sometimes these graphs are huge, so you should use `utils/analyzer/exploded-graph-rewriter.py` with various flags to extract useful information from the dump.

- See `clang-analyzer.llvm.org` for more information!