CS 6V81-05: System Security and Malicious Code Analysis
Principles of Program Analysis

Zhiqiang Lin

Department of Computer Science
University of Texas at Dallas

February 1\textsuperscript{st}, 2012
1. Program Analysis

2. Data Flow Analysis
   - Available Expressions
   - Liveness Analysis
   - Reaching Definitions
   - Very Busy Expressions

3. Theory Behind

4. Sensitivity

5. Summary
In computer science, program analysis is the process of automatically analysing the behavior of computer programs.
In computer science, program analysis is the process of automatically analysing the behavior of computer programs. Two main approaches in program analysis are static program analysis and dynamic program analysis.
Program Analysis

In computer science, program analysis is the process of automatically analysing the behavior of computer programs.

Two main approaches in program analysis are static program analysis and dynamic program analysis.

Main applications of program analysis are program correctness (including security) and program optimization.
Basic Techniques

Techniques related to program analysis include:

- control-flow and data-flow analysis
- constraint-based analysis
- abstract interpretation
- type and effect systems
Basic Techniques

Techniques related to program analysis include

- control-flow and data-flow analysis
- constraint-based analysis
- abstract interpretation
- type and effect systems

A technique that is applied for certain kinds of program analysis is program slicing.
## Basic Techniques

Techniques related to program analysis include:

- control-flow and data-flow analysis
- constraint-based analysis
- abstract interpretation
- type and effect systems

A technique that is applied for certain kinds of program analysis is **program slicing**.

Related fields include performance analysis and program verification.
1. Program Analysis

2. Data Flow Analysis
   - Available Expressions
   - Liveness Analysis
   - Reaching Definitions
   - Very Busy Expressions

3. Theory Behind

4. Sensitivity

5. Summary
Data Flow Analysis

Definition

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program.
Definition

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program.

A program’s control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate.
**Definition**

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program.

A program’s control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate.

The information gathered is often used by compilers when optimizing a program. A canonical example of a data-flow analysis is reaching definitions.
A simple way to perform data-flow analysis of programs is to set up data-flow equations for each node of the control flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes, i.e., it reaches a fixpoint.
Basic Approaches

A simple way to perform data-flow analysis of programs is to set up data-flow equations for each node of the control flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes, i.e., it reaches a fixpoint.

The above general approach was developed by Gary Kildall while teaching at the Naval Postgraduate School.
Outline

1. Program Analysis

2. Data Flow Analysis
   - Available Expressions
   - Liveness Analysis
   - Reaching Definitions
   - Very Busy Expressions

3. Theory Behind

4. Sensitivity

5. Summary
Analysis: Available Expressions

- An expression $e$ is available at program point $p$ if
  - $e$ is computed on every path to $p$, and
  - the value of $e$ has not changed since the last time $e$ was computed on every path to $p$
Analysis: Available Expressions

- An expression $e$ is available at program point $p$ if
  - $e$ is computed on every path to $p$, and
  - the value of $e$ has not changed since the last time $e$ was computed on every path to $p$

- Optimization
  - If an expression is available, need not be recomputed
    - (At least, if it is still in a register somewhere)
Is expression $e$ available?

Facts:
- $a + b$ is available
- $a \times b$ is available
- $a + 1$ is available

The following data flow examples are in courtesy of Prof. Mike Hicks.
**Gen and Kill**

- What is the effect of each statement on the set of facts?

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := a + b</td>
<td>a + b</td>
<td></td>
</tr>
<tr>
<td>y := a * b</td>
<td>a * b</td>
<td></td>
</tr>
<tr>
<td>a := a + 1</td>
<td>a + 1, a + b, a * b</td>
<td></td>
</tr>
</tbody>
</table>

**Diagram**

```
entry
  x := a + b
  y := a * b
  y > a
  a := a + 1
  x := a + b
exit
```
Computing Available Expressions

entry

x := a + b

y := a * b

y > a

a := a + 1

x := a + b

exit
Computing Available Expressions

\[ x := a + b \]
\[ y := a \times b \]
\[ y > a \]
\[ a := a + 1 \]
\[ x := a + b \]
Computing Available Expressions

- Computing Available Expressions
- \( \{a + b\} \)
- \( x := a + b \)
- \( y := a * b \)
- \( y > a \)
- \( a := a + 1 \)
- \( x := a + b \)
- \( \emptyset \)
- entry
- exit

Tuesday, September 20, 2011
Computing Available Expressions

∅

entry

{x := a + b}

{a + b}

y := a * b

y > a

a := a + 1

x := a + b

∅

exit

entry

{x := a + b}

y := a * b

y > a

a := a + 1

x := a + b

∅
Computing Available Expressions

\[ \emptyset \]

entry

\[ x := a + b \]

\{a + b\}

\[ y := a \times b \]

\{a + b, a \times b\}

\[ y > a \]

\[ a := a + 1 \]

\[ x := a + b \]

exit

Tuesday, September 20, 2011
Computing Available Expressions

∅

entry

{x := a + b}

{a + b}

{a + b, a * b}

y := a * b

y > a

{a + b, a * b}

a := a + 1

x := a + b

exit

Tuesday, September 20, 2011
Computing Available Expressions

0

{a + b}

{a + b, a * b}

{a + b, a * b}

x := a + b

y := a * b

y > a

a := a + 1

x := a + b

entry

exit

Tuesday, September 20, 2011
Computing Available Expressions

- Entry
  - \( \emptyset \)

- \( x := a + b \)
- \( y := a \times b \)

- \( y > a \)

- \( a := a + 1 \)
- \( x := a + b \)

- Exit
Computing Available Expressions

- $\emptyset$
- $\{a + b\}$
- $\{a + b, a * b\}$
- $\{a + b, a * b\}$
- $\emptyset$
Computing Available Expressions

∅

entry

x := a + b

y := a * b

y > a

a := a + 1

x := a + b

entry

exit

∅

{a + b}

{a + b, a * b}

{a + b, a * b}

∅
Computing Available Expressions

entry

∅

{x := a + b}

{y := a * b, a + b}

{y > a, a + b, a * b}

∅

{x := a + b, a * b, a + b}

entry

∅

{x := a + b, a * b, a + b}

y := a * b

y > a

a := a + 1

x := a + b

exit

Tuesday, September 20, 2011
Computing Available Expressions

```
∅
```

```
{a + b}
```

```
{a + b, a * b}
```

```
{a + b, a * b}
```

```
Ø
```

```
{a + b}
```

```
x := a + b
```

```
y := a * b
```

```
y > a
```

```
a := a + 1
```

```
x := a + b
```

```
entry
```

```
exit
```

```
Tuesday, September 20, 2011
```

```
CMSC 631 17
```

```
Computing Available Expressions
```

```
∅
```

```
{a + b}
```

```
{a + b, a * b}
```

```
{a + b, a * b}
```

```
Ø
```

```
{a + b}
```

```
x := a + b
```

```
y := a * b
```

```
y > a
```

```
a := a + 1
```

```
x := a + b
```

```
entry
```

```
exit
```

```
Tuesday, September 20, 2011
```
Computing Available Expressions

- Entry state: \( \emptyset \)
- \( x := a + b \)
- \( y := a \cdot b \)
- \( y > a \)
- \( a := a + 1 \)
- Exit state: \( \{a + b\} \)
Computing Available Expressions

\[ \emptyset \]

entry

\[ x := a + b \]

\[ \{a + b\} \]

\[ y := a \times b \]

\[ \{a + b, a \times b\} \]

\[ y > a \]

\[ a := a + 1 \]

\[ \emptyset \]

\[ \{a + b\} \]

\[ x := a + b \]

\[ \{a + b\} \]

exit

Tuesday, September 20, 2011
Computing Available Expressions

\[
\emptyset \rightarrow \text{entry} \rightarrow \{a + b\} \rightarrow \{a + b, a \times b\} \rightarrow \emptyset \rightarrow \{a + b\} \rightarrow \text{exit}
\]

\[
x := a + b \\
y := a \times b \\
y > a \\
a := a + 1 \\
x := a + b
\]
Computing Available Expressions

∅ → entry

{a + b} → x := a + b

{a + b, a * b} → y := a * b

∅ → exit

∅ → x := a + b

∅ → y > a

∅ → a := a + 1

∅ → {a + b}

∅ → {a + b}

∅ → {a + b}
Computing Available Expressions

∅ → entry

{x := a + b} → y := a * b

{a + b, a * b} → y > a

∅ → a := a + 1

∅ → x := a + b

∅ → exit

{a + b} → entry
A join point is a program point where two branches meet
Available expressions is a forward must problem

- Forward = Data flow from in to out
- Must = Property must hold on all paths to the join

Dataflow analysis requires facts that summarize all paths to a join point

- Symbolic execution analyzes each path separately
Data Flow Equations

Let $s$ be a statement

- $\text{succ}(s) = \{\text{immediate successor statements of } s\}$
- $\text{pred}(s) = \{\text{immediate predecessor statements of } s\}$
- $\text{In}(s) = \text{program point just before executing } s$
- $\text{Out}(s) = \text{program point just after executing } s$

- $\text{In}(s) = \bigcap_{s' \in \text{pred}(s)} \text{Out}(s')$

- $\text{Out}(s) = \text{Gen}(s) \cup (\text{In}(s) - \text{Kill}(s))$
1. Program Analysis

2. Data Flow Analysis
   - Available Expressions
   - Liveness Analysis
   - Reaching Definitions
   - Very Busy Expressions

3. Theory Behind

4. Sensitivity

5. Summary
Liveness Analysis

A variable v is live at program point p if v will be used on some execution path originating from p... before v is overwritten.

Optimization
If a variable is not live, no need to keep it in a register.
If variable is dead at assignment, can eliminate assignment.
A variable $v$ is live at program point $p$ if
- $v$ will be used on some execution path originating from $p$...
- before $v$ is overwritten
Liveness Analysis

- A variable $v$ is live at program point $p$ if
  - $v$ will be used on some execution path originating from $p$...
  - before $v$ is overwritten

- Optimization
  - If a variable is not live, no need to keep it in a register
  - If variable is dead at assignment, can eliminate assignment
Data Flow Equations

Available Expressions
- Available expressions is a forward must analysis
  - Data flow propagate in same dir as CFG edges
  - Expr is available only if available on all paths

Liveness Analysis
- Liveness is a backward may problem
  - To know if variable live, need to look at future uses
  - Variable is live if used on some path

\begin{align*}
\text{Out}(s) &= \bigcup_{s' \in \text{succ}(s)} \text{In}(s') \\
\text{In}(s) &= \text{Gen}(s) \cup (\text{Out}(s) - \text{Kill}(s))
\end{align*}
### Gen and Kill

What is the effect of each statement on the set of facts?

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := a + b</td>
<td>a, b</td>
<td>x</td>
</tr>
<tr>
<td>y := a * b</td>
<td>a, b</td>
<td>y</td>
</tr>
<tr>
<td>y &gt; a</td>
<td>a, y</td>
<td></td>
</tr>
<tr>
<td>a := a + 1</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

- $x := a + b$
- $y := a * b$
- $y > a$
- $a := a + 1$
- $x := a + b$
Computing Live Variables

x := a + b

y := a * b

y > a

a := a + 1

x := a + b
Computing Live Variables

```
x := a + b
y := a * b
y > a
a := a + 1
x := a + b
```
Computing Live Variables

\[
\begin{align*}
x &:= a + b \\
y &:= a \times b \\
y > a \\
a &:= a + 1 \\
x &:= a + b
\end{align*}
\]
Computing Live Variables

{x}

{x, y, a}

y := a * b

y > a

a := a + 1

{x, y, a}

x := a + b

Tuesday, September 20, 2011
Computing Live Variables

```
x := a + b
y := a * b
y > a
a := a + 1
x := a + b
```

Initial state: \{x, y, a\}

Intermediate state: \{y, a, b\}

Final state: \{x\}
Computing Live Variables

\[
x := a + b
\]

\[
y := a \times b
\]

\[
y > a
\]

\[
a := a + 1
\]

\[
x := a + b
\]

\[
\{x, y, a\}
\]

\[
\{y, a, b\}
\]

\[
\{x, y, a\}
\]

\[
\{y, a, b\}
\]

\[
\{y, a, b\}
\]

\[
\{x\}
\]
Computing Live Variables

- \( x := a + b \)
- \( y := a \times b \)
- \( y > a \)
- \( a := a + 1 \)
- \( x := a + b \)
Computing Live Variables

x := a + b

y := a * b

y > a

a := a + 1

x := a + b

{x, y, a, b}

{y, a, b}

{y, a, b}

{x, y, a}

{x, y, a, b}

{y, a, b}

{y, a, b}

{x}
Computing Live Variables

1. \( x := a + b \)
2. \( y := a \times b \)
3. \( y > a \)
4. \( a := a + 1 \)
5. \( x := a + b \)
Computing Live Variables

\[
x := a + b
\]
\[
y := a \times b
\]
\[
y > a
\]
\[
a := a + 1
\]
\[
x := a + b
\]
Computing Live Variables

{x, a, b} → {x, y, a, b} → {y, a, b} → {x, y, a, b} → {y, a, b} → {x, y, a, b} → {x} → {x, y, a, b} → {x, y, a, b} → {x}

x := a + b
y := a * b
y > a
a := a + 1
x := a + b
Computing Live Variables

\{a, b\} → x := a + b

\{x, a, b\} → y := a * b

\{x, y, a, b\} → y > a

\{y, a, b\} → a := a + 1

\{y, a, b\} → x := a + b

\{x, y, a, b\} → x := a + b

\{x\} →
Outline

1. Program Analysis

2. Data Flow Analysis
   - Available Expressions
   - Liveness Analysis
   - Reaching Definitions
   - Very Busy Expressions

3. Theory Behind

4. Sensitivity

5. Summary
Reaching Definitions

- A definition of a variable $v$ is an assignment to $v$
- A definition of variable $v$ reaches point $p$ if
  - There is no intervening assignment to $v$
- Also called def-use information
- What kind of problem?
  - Forward or backward?
  - May or must?
The assignment $[x := a]^{\ell}$ reaches $\ell'$ if there is an execution where $x$ was last assigned at $\ell$.

Example: Reaching Definitions

```
[y := x]^{1}
[z := 1]^{2}
[y > 0]^{3}
[z := z \times y]^{4}
[y := y - 1]^{5}
[y := 0]^{6}
```
Reaching Definitions

\[ y := x \]
\[ z := 1 \]
while \[ y > 0 \] do
\[ z := z \times y \]
\[ y := y - 1 \]
od;
Reaching Definitions

\[ y := x \]

\[ z := 1 \]

\[ \text{while } y > 0 \text{ do} \]

\[ z := z \times y \]

\[ y := y - 1 \]

\[ \text{od;} \]

\[ y := 0 \]
Reaching Definitions

\[
\begin{align*}
[y := x] & \quad \{(x, ?), (y, ?), (z, ?)\} \\
[z := 1] & \quad \{(x, ?), (y, 1), (z, ?)\} \\
\text{while } [y > 0] & \quad \{(x, ?), (y, 1), (y, 5), (z, 2), (z, 4)\} \cup \{(y, 5), (z, 4)\} \\
[z := z \times y] & \quad \{(x, ?), (y, 1), (y, 5), (z, 2), (z, 4)\} \\
[y := y - 1] & \quad \{(x, ?), (y, 5), (z, 4)\} \\
\text{od;} & \quad \{(x, ?), (y, 1), (y, 5), (z, 2), (z, 4)\} \\
[y := 0] & \quad \{(x, ?), (y, 1), (y, 5), (z, 2), (z, 4)\}
\end{align*}
\]
Reaching Definitions

\[y := x\]
\[z := 1\]
while \(y > 0\) do
\[z := z \times y\]
\[y := y - 1\]
end;
Outline

1. Program Analysis
2. Data Flow Analysis
   - Available Expressions
   - Liveness Analysis
   - Reaching Definitions
   - Very Busy Expressions
3. Theory Behind
4. Sensitivity
5. Summary
Very Busy Expressions

- An expression $e$ is very busy at point $p$ if
  - On every path from $p$, expression $e$ is evaluated before the value of $e$ is changed

- Optimization
  - Can hoist very busy expression computation

- What kind of problem?
  - Forward or backward?
  - May or must?
Outline

1 Program Analysis

2 Data Flow Analysis
   - Available Expressions
   - Liveness Analysis
   - Reaching Definitions
   - Very Busy Expressions

3 Theory Behind

4 Sensitivity

5 Summary
Most data flow analyses can be classified this way:
- A few don’t fit: bidirectional analysis
- Lots of literature on data flow analysis

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>Reaching definitions</td>
<td>Available expressions</td>
</tr>
<tr>
<td>Backward</td>
<td>Live variables</td>
<td>Very busy expressions</td>
</tr>
</tbody>
</table>
Typically, data flow facts form a lattice

Example: Available expressions

\[
\begin{align*}
\text{a+b, a*b, a+1} \\
\text{a+b, a*b} \\
\text{a*b, a+1} \\
\text{a+b} \\
\text{a*b} \\
\text{a+1} \\
\text{(none)} \\
\text{“top”} \\
\text{“bottom”}
\end{align*}
\]
Partial Orders

A partial order is a pair \((P, \leq)\) such that

- \(\leq \subseteq P \times P\)
- \(\leq\) is reflexive: \(x \leq x\)
- \(\leq\) is anti-symmetric: \(x \leq y \text{ and } y \leq x \Rightarrow x = y\)
- \(\leq\) is transitive: \(x \leq y \text{ and } y \leq z \Rightarrow x \leq z\)
A partial order is a lattice if □ and □ are defined on any set:

- □ is the meet or greatest lower bound operation:
  - $x \sqcap y \leq x$ and $x \sqcap y \leq y$
  - if $z \leq x$ and $z \leq y$, then $z \leq x \sqcap y$

- □ is the join or least upper bound operation:
  - $x \leq x \sqcup y$ and $y \leq x \sqcup y$
  - if $x \leq z$ and $y \leq z$, then $x \sqcup y \leq z$
Forward Must Data Flow Algorithm

Out(s) = Top for all statements s
Forward Must Data Flow Algorithm

Out(s) = Top for all statements s

//Slight acceleration: Could set Out(s) = Gen(s) \cup (Top - Kill(s))
Forward Must Data Flow Algorithm

Out(s) = Top for all statements s

//Slight acceleration: Could set Out(s) = Gen(s) ∪ (Top - Kill(s))

W := { all statements } (worklist)
Forward Must Data Flow Algorithm

Out(s) = Top for all statements s

//Slight acceleration: Could set Out(s) = Gen(s) ∪ (Top - Kill(s))

W := { all statements } (worklist)
repeat
Forward Must Data Flow Algorithm

Out(s) = Top for all statements s

//Slight acceleration: Could set Out(s) = Gen(s) \cup (Top - Kill(s))

W := \{ all statements \} (worklist)

repeat
    Take s from W
Forward Must Data Flow Algorithm

Out(s) = Top for all statements s

//Slight acceleration: Could set Out(s) = Gen(s) ∪ (Top - Kill(s))

W := { all statements } (worklist)

repeat
  Take s from W
  In(s) := \bigcap_{s' \in \text{pred}(s)} Out(s')
  \text{temp} := \text{Gen}(s) \cup (\text{In}(s) - \text{Kill}(s))
  \text{if} (\text{temp} \neq \text{Out}(s)) {
    \text{Out}(s) := \text{temp}
    W := W \cap \text{succ}(s)
  }

until W = \emptyset
Forward Must Data Flow Algorithm

\[ \text{Out}(s) = \text{Top for all statements } s \]

//Slight acceleration: Could set \( \text{Out}(s) = \text{Gen}(s) \cup (\text{Top} - \text{Kill}(s)) \)

\[ W := \{ \text{all statements} \} \quad \text{(worklist)} \]

repeat

Take \( s \) from \( W \)

\[ \text{In}(s) := \bigcap_{s' \in \text{pred}(s)} \text{Out}(s') \]

\[ \text{temp} := \text{Gen}(s) \cup (\text{In}(s) - \text{Kill}(s)) \]

if (temp \neq \text{Out}(s)) {

\[ \text{Out}(s) := \text{temp} \]

\[ W := W \cap \text{succ}(s) \]

}\until W = \emptyset
Out(s) = Top for all statements s

//Slight acceleration: Could set Out(s) = Gen(s) ∪ (Top - Kill(s))

W := { all statements } (worklist)

repeat
    Take s from W
    In(s) := \bigcap_{s' \in \text{pred}(s)} \text{Out}(s')
    temp := Gen(s) ∪ (In(s) - Kill(s))
    if (temp != Out(s)) {
    
}
Forward Must Data Flow Algorithm

Out(s) = Top for all statements s

//Slight acceleration: Could set Out(s) = Gen(s) ∪ (Top - Kill(s))

W := { all statements }  (worklist)

repeat
  Take s from W
  In(s) := \( \bigcap_{s' \in \text{pred}(s)} \text{Out}(s') \)
  temp := Gen(s) ∪ (In(s) - Kill(s))
  if (temp != Out(s)) {
    Out(s) := temp
  }

until W = ∅
Out(s) = Top for all statements s

//Slight acceleration: Could set Out(s) = Gen(s) ∪ (Top - Kill(s))

W := { all statements } (worklist)
repeat
  Take s from W
  In(s) := \bigcap_{s' \in \text{pred}(s)} \text{Out}(s')
  temp := Gen(s) ∪ (In(s) - Kill(s))
  if (temp != Out(s)) {
    Out(s) := temp
    W := W ∩ \text{succ}(s)
  }
until W = ∅
**Forward Must Data Flow Algorithm**

Out(s) = Top for all statements s

//Slight acceleration: Could set Out(s) = Gen(s) \(\cup\) (Top - Kill(s))

\[ W := \{ \text{all statements} \} \quad \text{(worklist)} \]

repeat

Take s from W

In(s) := \(\bigcap_{s' \in \text{pred}(s)} \text{Out}(s')\)

temp := Gen(s) \(\cup\) (In(s) - Kill(s))

if (temp \(!=\) Out(s)) {

Out(s) := temp

W := W \(\cap\) succ(s)

}


Forward Must Data Flow Algorithm

Out(s) = Top for all statements s

//Slight acceleration: Could set Out(s) = Gen(s) ∪ (Top - Kill(s))

W := { all statements } (worklist)
repeat
    Take s from W
    In(s) := \bigcap_{s' \in \text{pred}(s)} \text{Out}(s')
    temp := Gen(s) \cup (\text{In}(s) - \text{Kill}(s))
    if (temp != Out(s)) {
        Out(s) := temp
        W := W \cap \text{succ}(s)
    }
until W = \emptyset
Forward vs. Backward

**Out(s) = Top for all s**

\[ W := \{ \text{all statements} \} \]

repeat
  Take s from W
  temp := \( f_s(s' \in \text{pred}(s) \ Out(s')) \)
  if (temp ≠ Out(s)) {
    Out(s) := temp
    W := W \cup \text{succ}(s)
  }
until W = \ø

**In(s) = Top for all s**

\[ W := \{ \text{all statements} \} \]

repeat
  Take s from W
  temp := \( f_s(s' \in \text{succ}(s) \ In(s')) \)
  if (temp ≠ In(s)) {
    In(s) := temp
    W := W \cup \text{pred}(s)
  }
until W = \ø
Outline

1. Program Analysis
2. Data Flow Analysis
   - Available Expressions
   - Liveness Analysis
   - Reaching Definitions
   - Very Busy Expressions
3. Theory Behind
4. Sensitivity
5. Summary
Flow-Sensitivity

- Data flow analysis is flow-sensitive
  - The order of statements is taken into account
  - I.e., we keep track of facts per program point
- Alternative: Flow-insensitive analysis
  - Analysis the same regardless of statement order
  - Standard example: types
    - /* x : int */ x := ... /* x : int */
Path-Sensitivity

Path-sensitive Analysis

A path-sensitive analysis computes different pieces of analysis information dependent on the predicates at conditional branch instructions.
Path-Sensitivity

Path-sensitive Analysis

A path-sensitive analysis computes different pieces of analysis information dependent on the predicates at conditional branch instructions.

For instance, if a branch contains a condition $x>0$, then on the fall-through path, the analysis would assume that $x\leq 0$ and on the target of the branch it would assume that indeed $x>0$ holds.
A context-sensitive analysis is an inter-procedural analysis that considers the calling context when analyzing the target of a function call.
Context-Sensitivity

Context-sensitive Analysis

A context-sensitive analysis is an inter-procedural analysis that considers the calling context when analyzing the target of a function call.

In particular, using context information one can jump back to the original call site, whereas without that information, the analysis information has to be propagated back to all possible call sites, potentially losing precision.
Terminology Review

- Must vs. May
- Forwards vs. Backwards
- Flow-sensitive vs. Flow-insensitive
Data Flow Analysis and Functions

- What happens at a function call?
  - Lots of proposed solutions in data flow analysis literature
- In practice, only analyze one procedure at a time
- Consequences
  - Call to function kills all data flow facts
  - May be able to improve depending on language, e.g., function call may not affect locals
More Terminology

- An analysis that models only a single function at a time is intraprocedural.
- An analysis that takes multiple functions into account is interprocedural.
Data Flow Analysis and The Heap

- Data Flow is good at analyzing local variables
  - But what about values stored in the heap?
  - Not modeled in traditional data flow

- In practice: `*x := e`
  - Assume all data flow facts killed (!)
  - Or, assume write through x may affect any variable whose address has been taken

- In general, hard to analyze pointers
Outline

1. Program Analysis

2. Data Flow Analysis
   - Available Expressions
   - Liveness Analysis
   - Reaching Definitions
   - Very Busy Expressions

3. Theory Behind

4. Sensitivity

5. Summary
Moore’s Law: Hardware advances double computing power every 18 months.
Data Flow Analysis and Optimization

- Moore’s Law: Hardware advances double computing power every 18 months.
- Proebsting’s Law: Compiler advances double computing power every 18 years.
DF Analysis and Defect/Vulnerability Detection

- LCLint - Evans et al. (UVa)
- METAL - Engler et al. (Stanford, now Coverity)
- EXE/KLEE - Engler et al. (Stanford)
- ESP - Das et al. (MSR)
- Whitebox Fuzzing (MSR)
- Gamma Tech (Cornell and WiSc)
- IDA-Pro
- ...

...
Summary

- Program analysis
- Data flow analysis
  - Available Expression
  - Liveness Analysis
  - Reaching Definitions
  - Very Busy Expressions
- The theory behind the program analysis
- Sensitivity
References

- Program analysis and understanding by Prof. Mike Hicks at UMD
- Principles of Program Analysis
  http://www2.imm.dtu.dk/ riis/PPA/ppasup2004.html