

18732: Secure Software Systems

A Static Analysis Method for Buffer Overflow Detection

Anupam Datta
CMU

Fall 2010

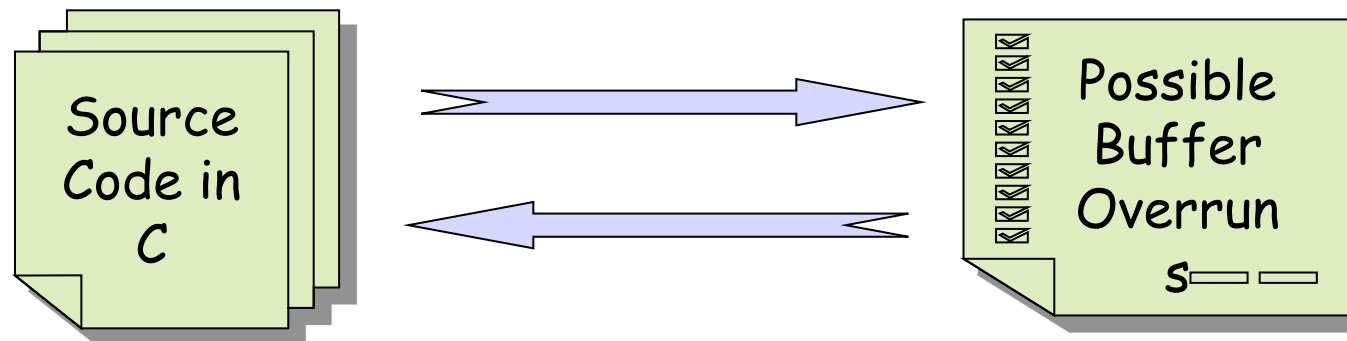
Model Checking: Final Comments

- Widely used in industry for hardware verification
- Beginning to be used for software verification including security
- Main problem: State space explosion
 - symmetry reduction, partial order reduction, symbolic model checking, bounded model checking, counter-example guided abstraction refinement

Big Picture

- Earlier lecture:
 - Low-level attacks on software systems, specifically, control hijacking attacks (buffer overflow, format string vulnerabilities)
 - Runtime enforcement methods for buffer overflow detection
- Issues with run-time enforcement
 - Performance hit
 - Functionality hit (typically program execution terminated if overflow detected)
- Today:
 - *Static analysis* of C source code for buffer overflow detection
 - Detect and fix potential buffer overflow *before* running code

Today's lecture

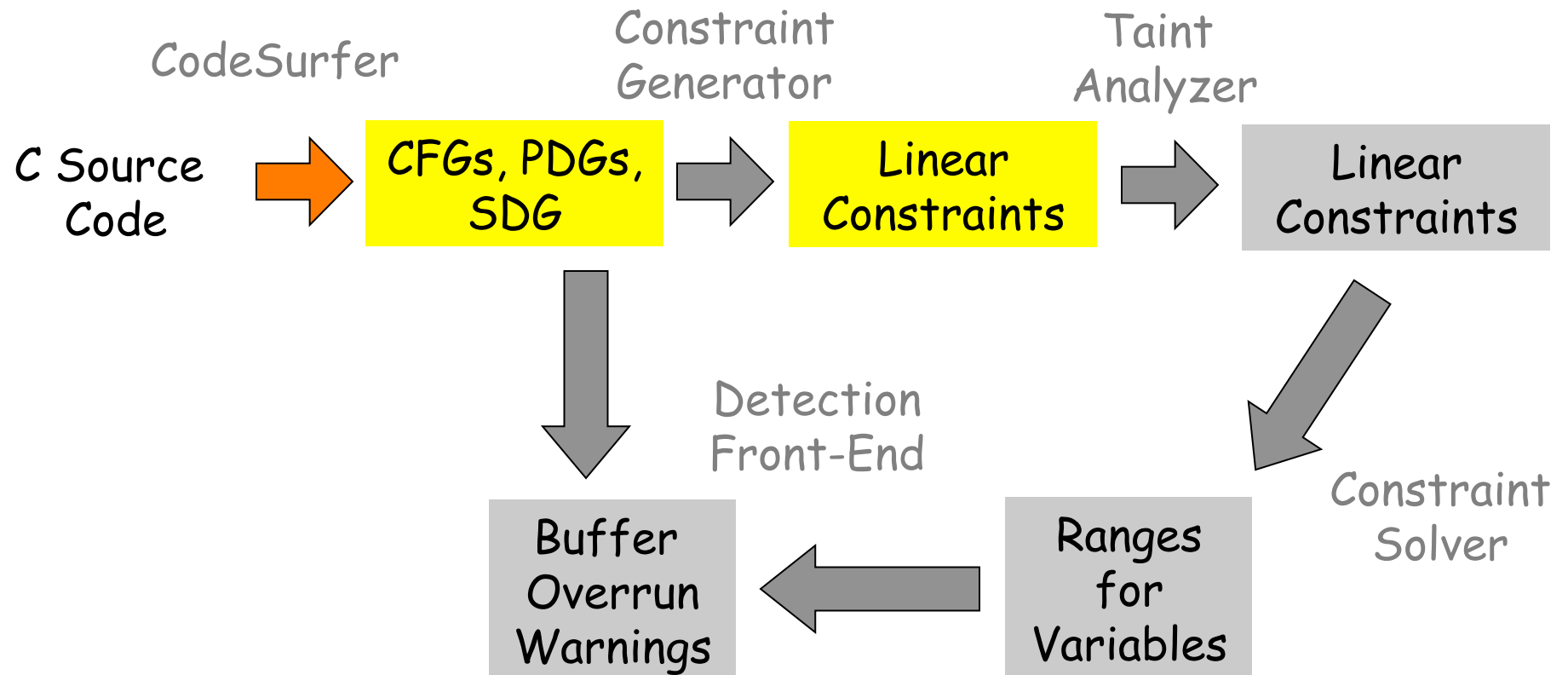


- Goal
 - Use static program analysis to produce a list of possibly vulnerable program locations
- Main idea
 - Model C string manipulations as a linear program
- General program analysis concept
 - Precision
 - False positives (mistakenly flag as a bug) & negatives (miss bugs)
 - Incomplete & unsound
- General program analysis challenges
 - Context-sensitive analysis (reduce false positives)
 - Pointer analysis (reduce false negatives)

Outline of lecture

- Tool Architecture
- Adding Context Sensitivity
- Pointer Analysis
- Results

Tool Architecture



Constraint Variables

- Four constraint variables for each string buffer
 - Overflow possible if more memory than allocated is used
 - `buf!alloc!max, buf!alloc!min`
 - `buf!used!max, buf!used!min`
- Two constraint variables for each integer variable
 - `i!max, i!min`

Constraint Generation

- Four kinds of program points result in constraints:
 - Declarations
 - Assignments
 - Function Calls
 - Function Return

Declaration

- Example code

```
char header[2048];
```

- Constraints

```
header!alloc!max ≥ 2048
```

```
header!alloc!min ≤ 2048
```

Assignment

- Example code

```
buf[i] = 'c';
```

- Constraints

```
buf!used!max  $\geq$  i!max
```

```
buf!used!min  $\leq$  i!min
```

Library Function Call

- Example code

```
strcpy(copy, buffer);
```

- Constraints

$copy!used!max \geq buffer!used!max$
 $copy!used!min \leq buffer!used!min$

User-defined functions (calls and returns) – next slide

A Larger Example

Focus on user-defined function call and return

```

(1) main(int argc, char* argv){
(2)   char header[2048], buf[1024],
      *cc1, *cc2, *ptr;
(3)   int counter;
(4)   FILE *fp;
(5)   ...
(6)   ptr = fgets(header, 2048, fp);
(7)   cc1 = copy_buffer(header);
(8)   for (counter = 0; counter < 10; counter++){
(9)     ptr = fgets (buf, 1024, fp);
(10)    cc2 = copy_buffer(buf);
(11)  }
(12) }
(13)
(14) char *copy_buffer(char *buffer){
(15)   char *copy;
(16)   copy = (char *) malloc(strlen(buffer));
(17)   strcpy(copy, buffer);
(18)   return copy;
(19) }

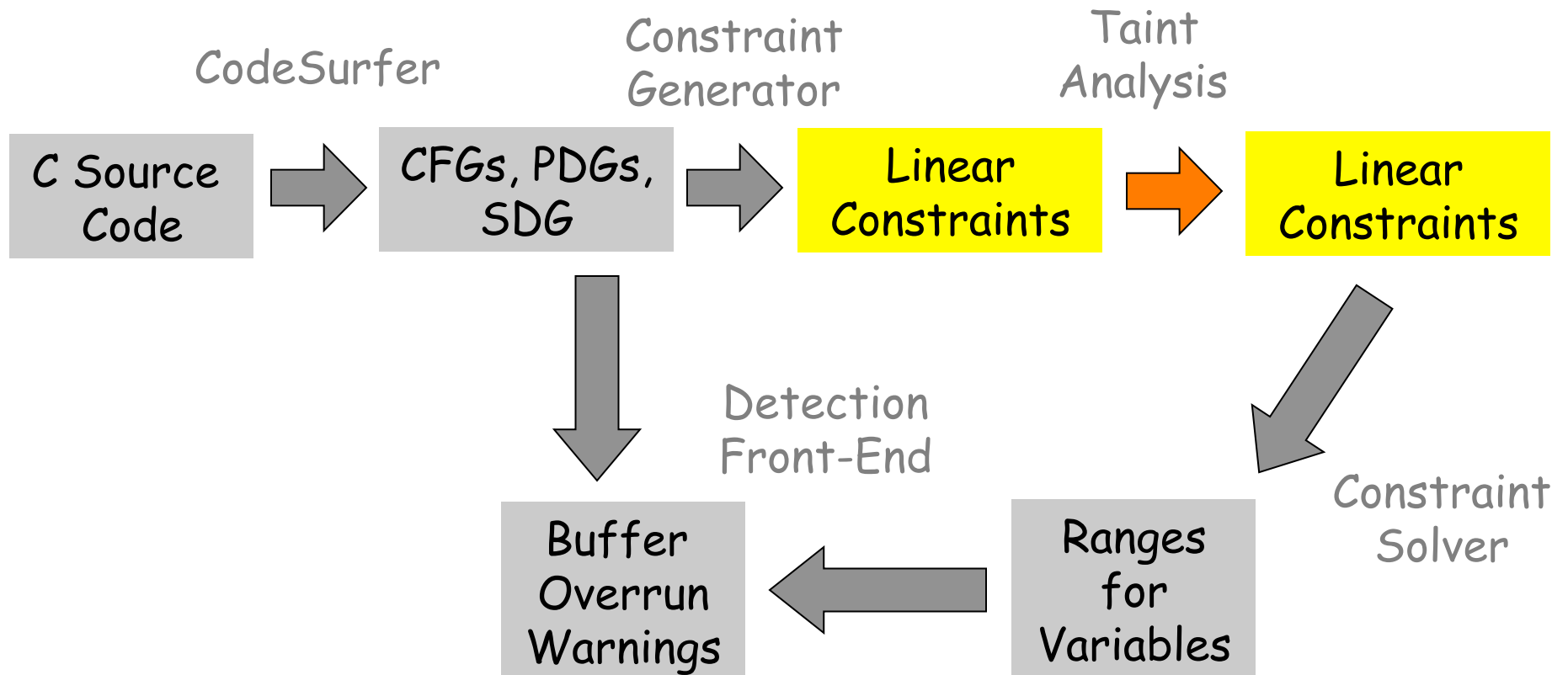
```

Constraint	Stmt.
header!used!max \geq 2048	6
header!used!min \leq 1	6
buffer!used!max \geq buf!used!max	10
buffer!used!min \leq buf!used!min	10
buffer!alloc!max \geq buf!alloc!max	10
buffer!alloc!min \leq buf!alloc!min	10
copy_buffer\$return!alloc!max \geq copy!alloc!max	18
copy_buffer\$return!alloc!min \leq copy!alloc!min	18
copy_buffer\$return!used!max \geq copy!used!max	18
copy_buffer\$return!used!min \geq copy!used!min	18
cc2!used!max \geq copy_buffer\$return!used!max	10
cc2!used!min \leq copy_buffer\$return!used!min	10
cc2!alloc!max \geq copy_buffer\$return!alloc!max	10
cc2!alloc!min \geq copy_buffer\$return!alloc!min	10
counter'!max \geq counter!max + 1	8
counter!max \geq counter'!max	8
counter'!min \leq counter!min + 1	8
counter!min \leq counter'!min	8

Function call

- Passing parameters to function
- Assigning function return value

Taint Analysis



Taint Analysis

- **Goals:**
 - Identify and remove constraint variables that get an infinite value
 - Example: user supplied integers, environment variables, uninitialized variables
- **Required:**
 - For linear programming solvers to function correctly
- **Main idea:**
 - Start from variables with infinite values; remove associated constraints; identify other variables that can take on infinite values and iterate.

This step leads to imprecision (may cause false positives & negatives)

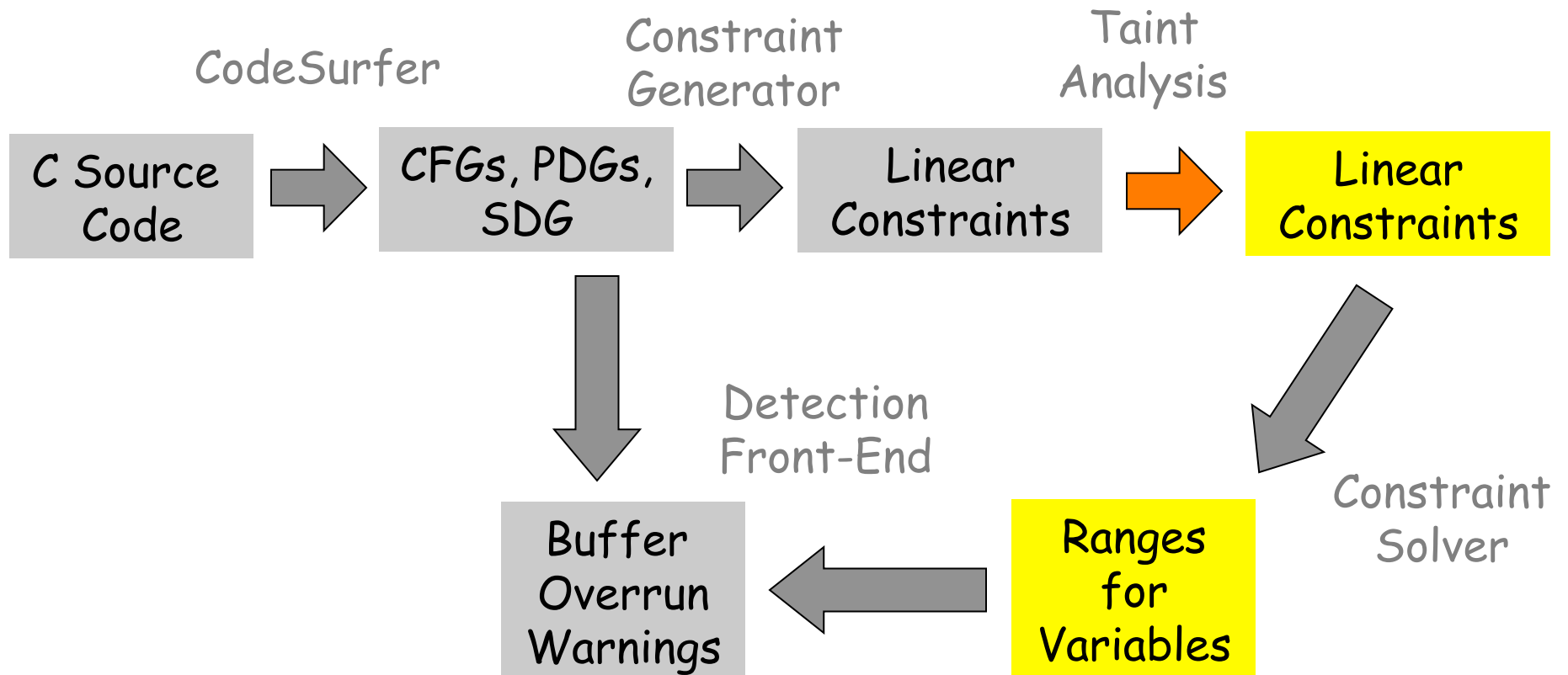
Handling Infeasible LPs

```
Minimize : counter!max
Subject To : counter'!max ≥ counter!max + 1
            counter!max ≥ counter'!max
```

- Remove “small” subset of constraints s.t. resulting constraint system is *feasible* (has a solution)
 - “Small” = a minimal set of inconsistent constraints

This step leads to imprecision (may cause false positives & negatives)

Solving Linear Constraints



Constraint Solvers

- Abstract problem:
 - Given a set of constraints on **max** and **min** variables
 - Get tightest possible fit satisfying all the constraints
- Approach:
 - Model as a linear program and solve

Linear Programming

- Optimization problem:
 - Maximize: $c^T x$
 - Subject to: $Ax \leq b$
 - Here: A is a matrix of constants, b & c are vectors of constants and x is a vector of variables, which can take finite real values
- Example:

Maximize: $x + 3y$

Subject to:

$$2x + 7y \leq 3$$

$$5x + 3y \leq 7$$

Why Linear Programming (LP)?

- Can support arbitrary linear constraints
- Well developed theory
- Commercial linear program solvers are highly optimized and fast; can scale to millions of constraints
 - Simplex algorithm [Dantzig 1947] works well in practice, although it is worst-case exponential time
 - LP can be solved in polynomial time [Karmakar 84]
 - Example tools: CPLEX, Ip_solve
 - Integer Linear Programming is NP-Complete

Google “linear programming” for more information about LP algorithms

Linear programming problem

- Goal: Obtain values for buffer bounds
- Modeling as a Linear Program

Minimize: max variable

subject to:

Set of Constraints

And

Maximize: min variable

subject to:

Set of Constraints

Least Upper Bound

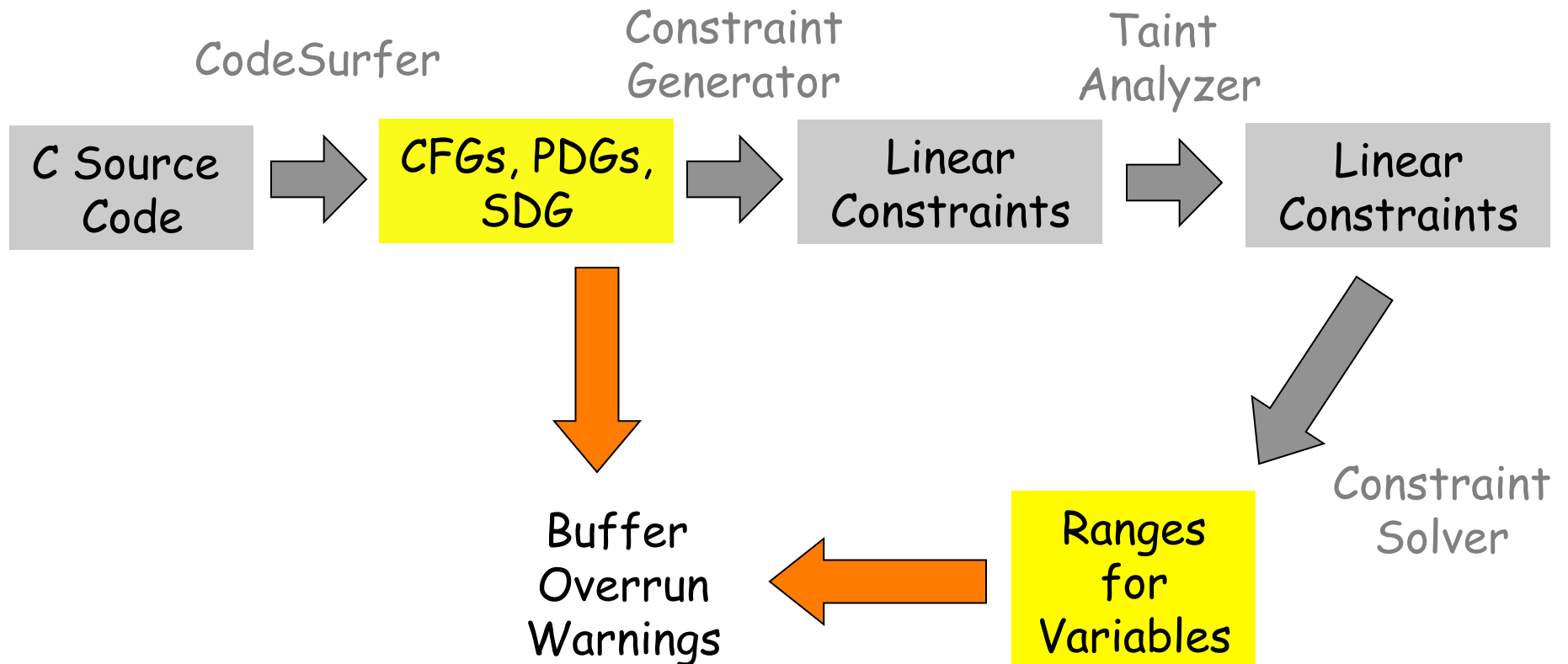
Greatest Lower Bound

Tightest possible fit

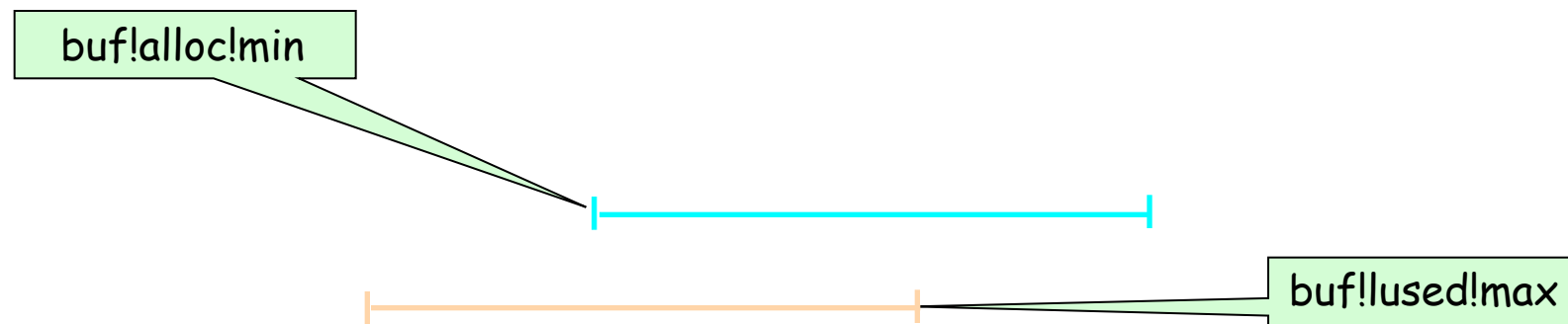
Improvement

- It can be shown that:
Min: Σ (max vars) – Σ (min vars)
Subject to: Set of Constraints
yields the same solution for each variable
- Solve just *one linear program* and get values for all variables

Detection Front-End



LP Solution \rightarrow Buffer Overflow?



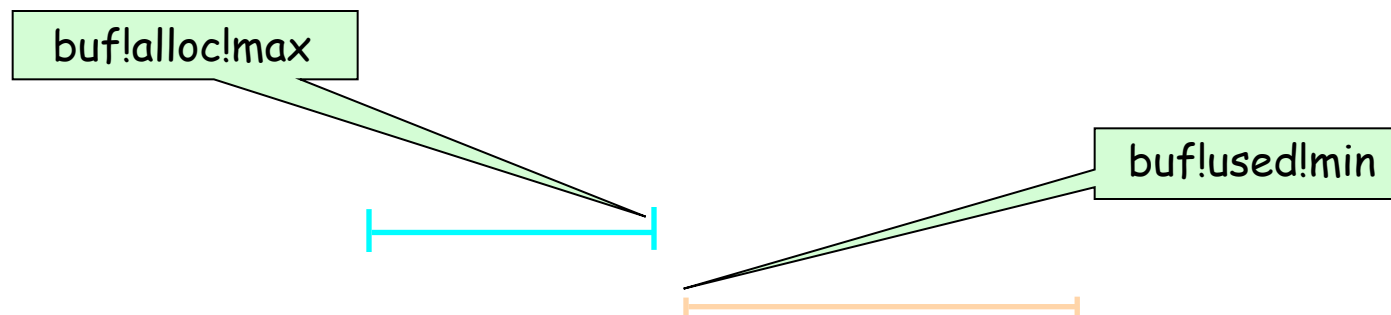
Scenario I: "Possible" buffer overflow

LP Solution \rightarrow Buffer Overflow?



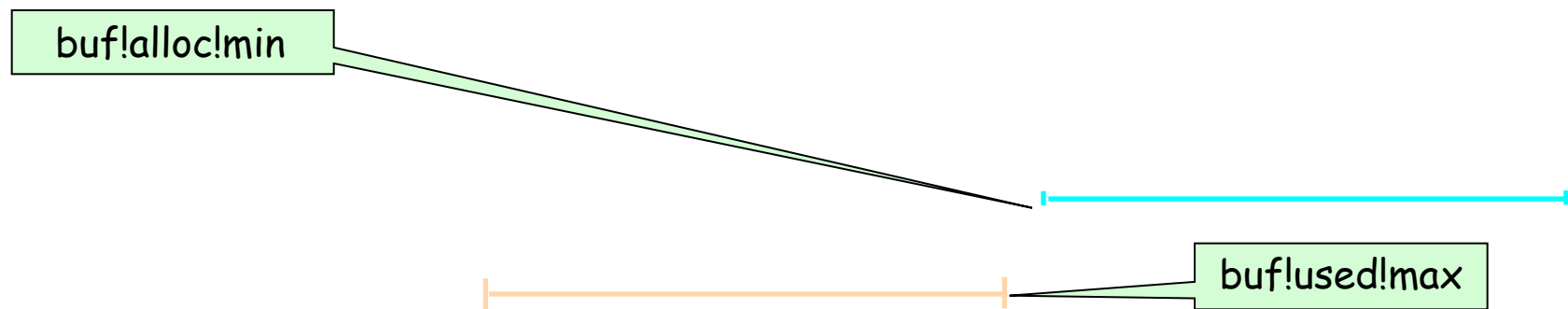
Scenario II: "More possible" buffer overflow

LP Solution \rightarrow Buffer Overflow?



Scenario III: Sure buffer overflow

LP Solution \rightarrow Buffer Overflow?

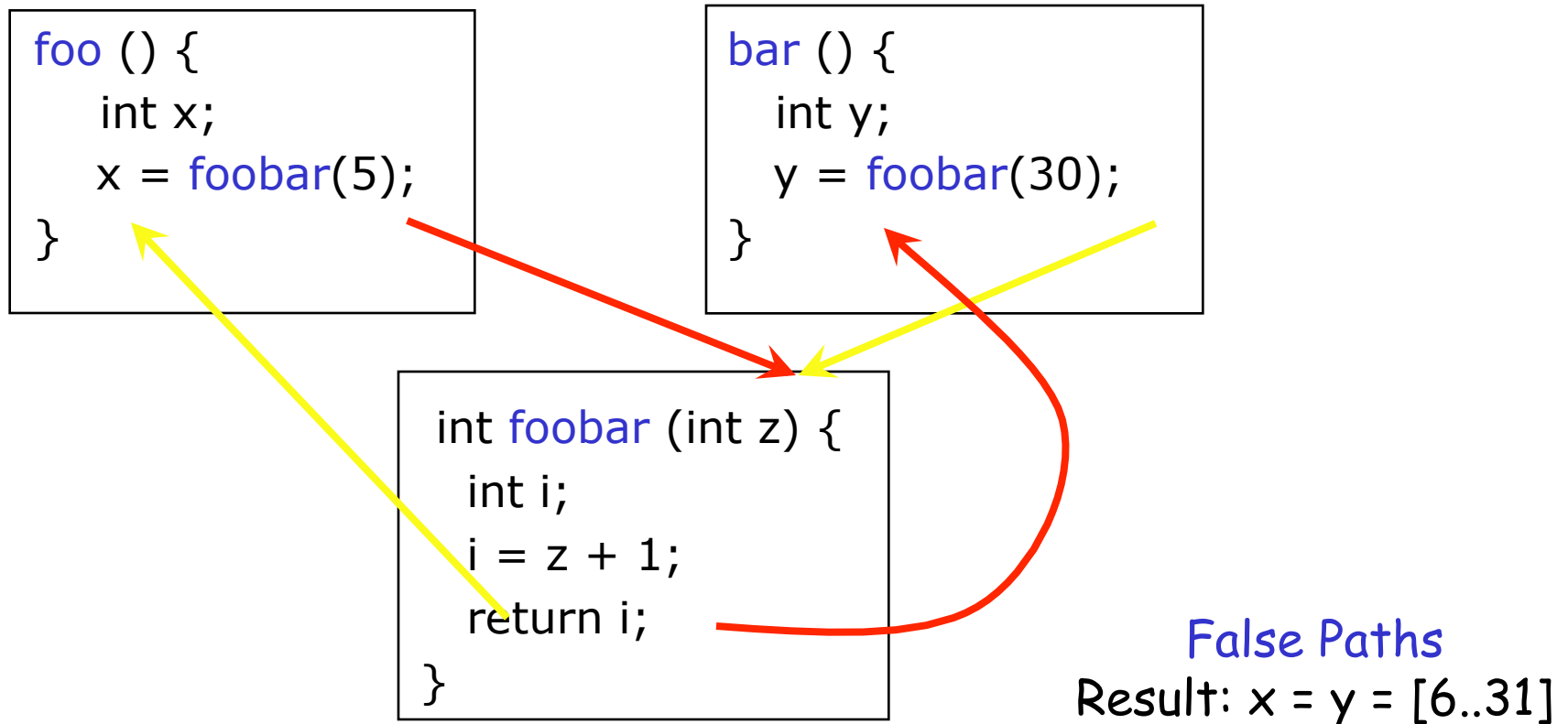


Scenario IV: No buffer overflow

Outline of lecture

- Tool Architecture
 - Constraint Generation
 - Constraint Resolution
 - Producing Warnings
- Adding Context Sensitivity
- Pointer Analysis
- Results

Context-Insensitive Analysis



Adding Context Sensitivity

- Make user functions context-sensitive
 - Improve precision: Aware of which call site called function

Method: Constraint inlining

- Formal variables of function are uniquely *renamed* while generating constraints for each call site

- Example

Note subscript 1

```
char *copy_buffer(char *buffer) {
    char *copy;
    copy = (char *) malloc(strlen(buffer));
    strcpy(copy, buffer);
    return copy;
}
```

<pre>copy!alloc!max₁ ≥ buffer!used!max₁ - 1 copy!used!max₁ ≥ buffer!used!max₁ copy!used!min₁ ≤ buffer!used!min₁ copy_buffer\$return!used!max₁ ≥ copy!used!max₁ copy_buffer\$return!used!min₁ ≤ copy!used!min₁</pre>

Constraint Inlining: Pros and Cons

- 😊 Precisely separates calling contexts
- 😞 Large number of constraint variables

Outline of lecture

- Tool Architecture
- Adding Context Sensitivity
- **Pointer Analysis**
- Results
- Summary

Pointer Analysis

- Example
 - `strcpy(p → f, buf)`; where `p` is a pointer to a structure `s`
 - Constraints relate constraint variables for `s.f` and `buf`
- Can handle arbitrary levels of dereferencing
- Example false negative
 - Without pointer analysis, tool misses vulnerability in `sendmail-8.7.6` (see paper)
- Significant amount of research on pointer analysis
 - This tool implements Anderson's method [And94]
 - Create "may point to" map

Pointer Analysis Example

- Code and safe point-to map

```
int main(void)
{
    int x, y, *p, **q, (*fp)(char *, char *);
    p = &x;
    q = &p;
    *q = &y;
    fp = &strcmp;
}
```

A safe point-to map is

$$[p \mapsto \{x, y\}, q \mapsto \{p\}, fp \mapsto \{strcmp\}]$$

and it is also a *minimal* map.

- Point-to map obtained by solving constraint system

$$\{T_p \supseteq \{x\}, T_q \supseteq \{p\}, *T_q \supseteq \{y\}, T_{fp} \supseteq \{strcmp\}\}$$

Outline of lecture

- Tool Architecture
- Adding Context Sensitivity
- Pointer Analysis
- Results

Results: Overruns Identified

Application	LOC	Warnings	Vulnerability	Detected?
WU-FTPD-2.5.0	16000	139	CA-1999-13	Yes
WU-FTPD-2.6.2	18000	178	None	14 New
Sendmail-8.7.6	38000	295	Identified by Wagner et al.	Yes
Sendmail-8.11.6	68000	453	CA-2003-07	Yes

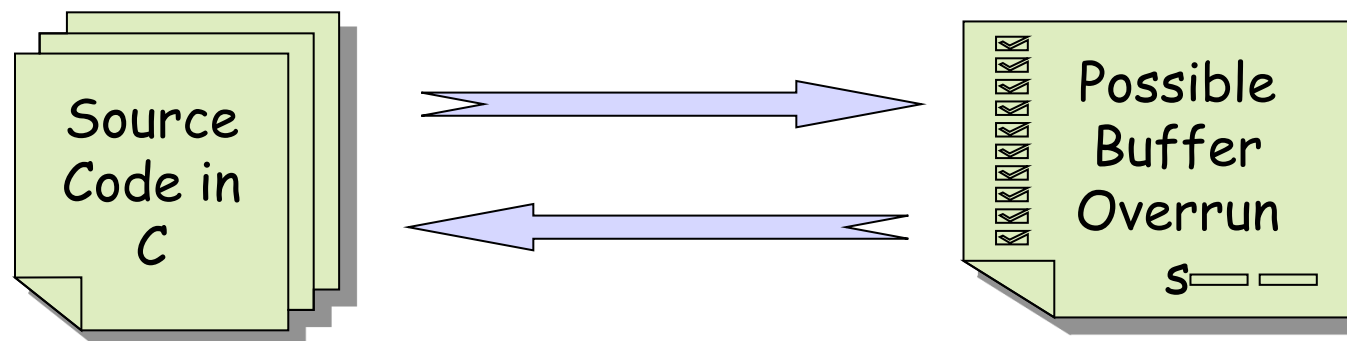
Note imprecision: Many false positives; false negatives harder to measure

Results: Performance

	wu-ftp-2.6.2	sendmail-8.7.6
CODESURFER	12.54 sec	30.09 sec
GENERATOR	74.88 sec	266.39 sec
TAINT	9.32 sec	28.66 sec
LPSOLVE	3.81 sec	13.10 sec
HIER.SOLVE	10.08 sec	25.82 sec
TOTAL (LP./HIER.)	100.55/106.82 sec	338.24/350.96 sec
Number of Constraints Generated		
PRE-TAINT	22008	104162
POST-TAINT	14972	24343

Table 1: Performance of the tool

Summary



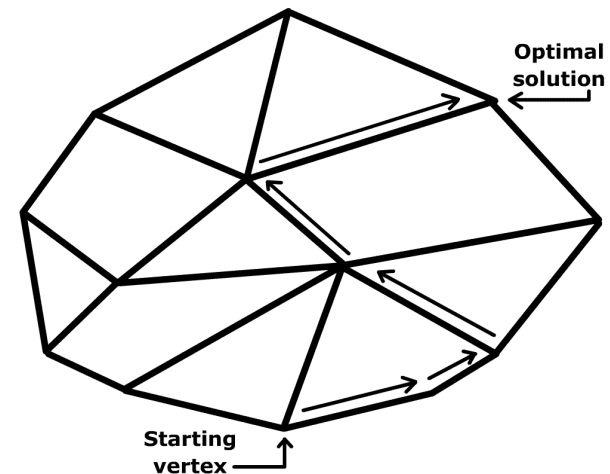
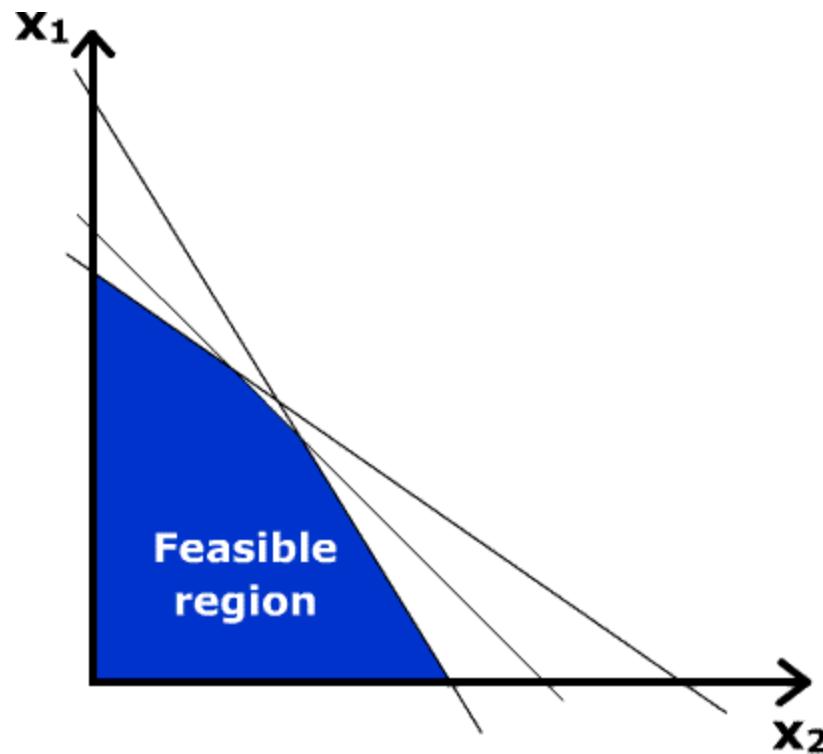
- Goal
 - Use static program analysis to produce a list of possibly vulnerable program locations
- Main idea
 - Model C string manipulations as a linear program
- General program analysis concept
 - Precision
 - False positives (mistakenly flag as a bug) & negatives (miss bugs)
- General program analysis challenges
 - Context-sensitive analysis (reduce false positives)
 - Pointer analysis (reduce false negatives)

Acknowledgement

- Ganapathy et al.,
[Buffer Overrun Detection using Linear Programming and Static Analysis](#), CCS 2003.
- Thanks to V. Ganapathy for helpful discussions and sharing his CCS 2003 talk slides, which were useful in preparing this lecture

Questions?

Simplex in a Picture



The simplex algorithm solves LP problems by constructing an admissible solution at a vertex of the polyhedron and then walking along edges of the polyhedron to vertices with successively higher values of the objective function until the optimum is reached.

Taint Analysis Algorithm

```

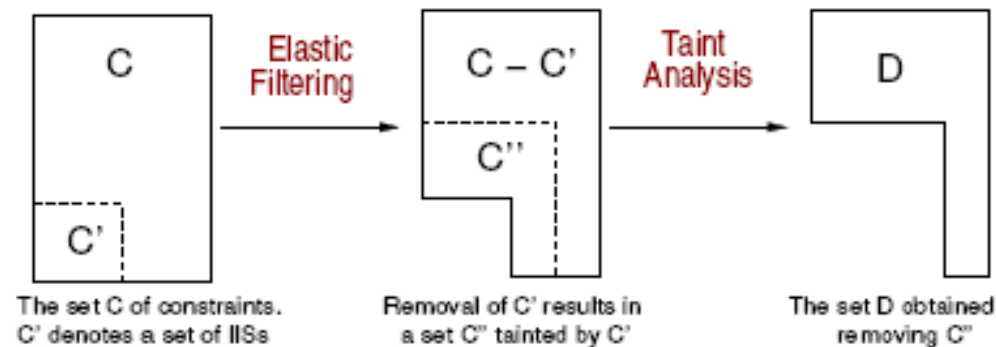
Input: Set of Constraints  $C$ 
Output: Subset of  $C$  with no uninitialized, or infinite variables
(1)   InfSet = {var | var  $\leq -\infty \vee$  var  $\geq \infty$ }  $\cup$  {var | var is un-initialized}
(2)   while InfSet  $\neq \phi$ 
(3)     Select and remove var from InfSet
(4)     foreach Constraint  $c \in C$  of the form MaxVar  $\geq$  RHS
(5)       if MaxVar is var
(6)         Drop  $c$  from  $C$ 
(7)       else if var appears in RHS
(8)         Set MaxVar to  $+\infty$  and add MaxVar to InfSet
(9)         Drop  $c$  from  $C$ 
(10)      endif
(11)    foreach Constraint  $c \in C$  of the form MinVar  $\leq$  RHS
(12)      if MinVar is var
(13)        Drop  $c$  from  $C$ 
(14)      else if var appears in RHS
(15)        Set MinVar to  $-\infty$  and add MinVar to InfSet
(16)        Drop  $c$  from  $C$ 
(17)      endif
(18)  Return  $C$ 

```

Figure 4: Algorithm for Taint Analysis

Main idea: Start from variables with infinite values; remove associated constraints; identify other variables that can take on infinite values and iterate.

Handling Infeasible LPs



- C' : IIS identified by Elastic Filtering
- Set max and min vars in C' to ∞ and $-\infty$
- Run taint analysis to identify vars that may be infinite and remove constraints in which they appear

This step leads to imprecision (may cause false positives & negatives)

Method 2: Procedure Summaries

- Basic Idea:
 - *Summarize* the called procedure
 - Insert the summary at the call-site in the caller
 - Remove false paths

Example 1

```
foo () {
  int x;
  x = foobar(5);
}
```

$x = 5 + 1$

```
bar () {
  int y;
  y = foobar(30);
}
```

$y = 30 + 1$

```
int foobar (int z) {
  int i;
  i = z + 1;
  return i;
}
```

Summary: $i = z + 1$

No false paths 😊

$x = [6..6]$
 $y = [31..31]$
 $i = [6..31]$

Example 2

Function

```
char *copy_buffer(char *buffer) {
    char *copy;
    copy = (char *) malloc(strlen(buffer));
    strcpy(copy, buffer);
    return copy;
}
```

Summary constraints

<pre>copy_buffer\$return!alloc!max ≥ buffer!used!max - 1 copy_buffer\$return!used!max ≥ buffer!used!max copy_buffer\$return!alloc!min ≤ buffer!used!min - 1 copy_buffer\$return!used!min ≤ buffer!used!min</pre>
--

Call Sites

```
cc1 = copy_buffer(header);
cc2 = copy_buffer(buf);
```

Replace formal parameters
 <buffer, copy_buffer\$return> in
 summary constraints by actual
 parameters (<header, cc1>,
 <buf,cc2>) at each call site

Procedure Summaries: Pros & Cons

- 😊 No increase in number of constraint variables *since formal parameters are not renamed*
- 😞 Not as precise as constraint inlining *since formal parameters are not renamed*

Results: Context Sensitivity

- WU-FTPD-2.6.2: 7310 ranges identified with context-insensitive analysis
- Constraint Inlining:
 - 5.8x number of constraints
 - 8.7x number of constraint variables
 - 406 ranges *refined* (tighter, more precise bounds) in at least one calling context
- Procedure Summaries:
 - 72 ranges refined
 - 1% increase in number of constraints