

Buffer Overrun Detection via Static Analysis

Vinod Ganapathy

University of Wisconsin

Introduction

- Buffer Overruns:
 - Easily exploitable class of vulnerabilities
 - Large number of systems are vulnerable
- Inadequate bounds checking
- CERT:
 - 9 out of 19 vulnerabilities since July '02
 - BIND, Kerberos, SSH, OpenSSL

WiSA BO-Tool

- Addresses the Buffer Overrun Problem
- Features:
 - *Statically* analyzes code for vulnerabilities
 - Vulnerabilities can be caught before deployment
 - Uses *points-to* information
 - Complicated dependencies can be tracked
 - Is designed to *scale* to large programs

Overview of Talk

- Related Work
- Tool Architecture
 - Constraint Generation
 - Constraint Solving
- Results
- Goals

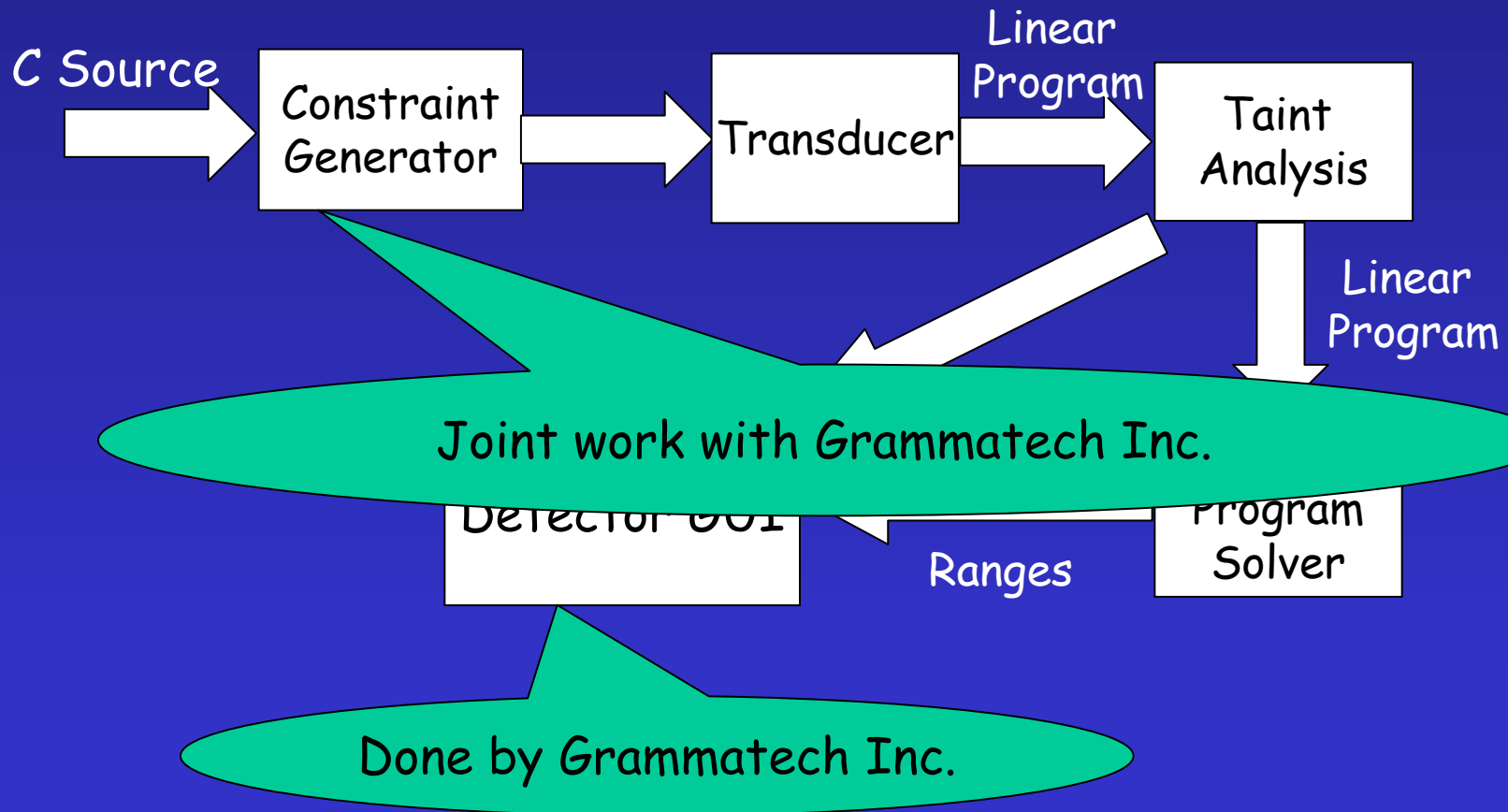
Related Work

- 'Fat' Pointers:
 - Static + Dynamic Analysis
 - SafeC (Wisconsin: Austin et.al.)
 - CCured (Berkeley: Necula et.al.)
- StackGuard:
 - Place 'canary' on the stack
 - Dynamic Analysis: High runtime overhead

Related Work

- BOON: (Berkeley: Wagner et.al.)
 - Closest relative to our work
 - Static Analysis
 - But, no points-to information used
 - Yet, good results

BO-Tool Architecture



Enhancements since July'02

- Taint Analysis and Pre-solve
- Ability to handle all kinds of Linear Programs
- Detector GUI with trace-back
- Other kinds of solvers

Constraint Generation

- Constraint Generator + Transducer
- Input: C source code
- Output: Linear Program
- Basic Idea:
 - Treat buffers as abstract data types
 - Reflect changes in buffers by changing associated buffer variables

Constraint Generation

- Four variables for each string buffer
 - `buf_len_max`, `buf_len_min`
 - `buf_alloc_max`, `buf_alloc_min`
- Operations on a buffer
 - `strcpy(target, source)`
 - `target_len_max >= source_len_max`
 - `target_len_min <= source_len_min`

Constraint Generation

- Source code fed to Codesurfer
- Analysis is done by Codesurfer
- Various options available for program analysis

Constraint Generation

- Options Available
 - *Flow Sensitive Analysis*:
 - Respect Program order
 - *Flow Insensitive Analysis*:
 - Do not respect program order
 - *Context-Sensitive* modeling of functions:
 - Differentiate Information between call-sites
 - *Context-Insensitive* modeling of functions:
 - Merge Information across call-sites

Constraint Generation

- Current Model:
 - Flow Insensitive Analysis
 - Context-sensitive modeling for some library functions
 - Context-insensitive for the rest
- Pros and Cons:
 - ☺ Faster and Easier Analysis
 - ☺ Smaller space requirements
 - ☹ Lower Precision => Higher False Positives

The Solver

- Abstract Problem:
 - Given a set of constraints on min and max variables
 - Get tightest possible fit satisfying the constraints
- Our approach:
 - Model and solve as a linear program

Why Linear Programming?

- Rich literature available
 - Solutions to problems readily available
- Commercial solvers available
 - No need to build our own solver
 - Highly optimized code => faster solves
- Known to scale to large problem sizes
 - One of our initial goals

The Solver

- Consists of various phases:
 - Taint Analysis
 - Pre-solve value inference
 - Obtain solution based on constraint analysis
 - $a \geq 4$ and $a \geq 3$ imply $a \geq 4$
 - Mainly an optimization to speed up LP solver
 - Linear Program Solver

Taint Analysis

- Objective: serve as a pre-solve step
- Search constraints for variables that
 - Are entered by the user:
 - `sprintf(buf, "%s", argv[1])`
 - Are un-initialized (incomplete modeling)
 - e.g. Library function that has not been modeled
- Helps reduce the Linear Program size

Linear Programming

- A set of constraints C
- Subject to: An objective function F
- Example:
 Maximize: x
 Subject to:
 $x \leq 3$

Linear Program Solver

- In our case:
 - Constraints are available
 - Goal: Obtain values for buffer bounds
- Modeling as a Linear Program

Minimize: max variable

Least Upper Bound

Subject to:

Set of Constraints

And

Maximize: min variable

Greatest Lower Bound

Subject to:

Set of Constraints

Linear Program Solver

- The Solution to an LP can be:
 - Optimal
 - Unbounded
 - Infeasible (constraint set is infeasible)

Linear Program Solver

- Optimal:
 - All constraints are satisfied
 - Objective function is optimized
- Value of buffer variable = solution
- Example:
 - Minimize: `buf_len_max`
 - `buf_len_max >= 3`

Linear Program Solver

- Unbounded
 - Infinitely many solutions exist
- Example:
 - Minimize: var_max
 - $\text{var_max} - \text{var2_max} \geq 4$
- Solution: set variable value to $\infty/-\infty$

Linear Program Solver

- Infeasible:
 - No solution exists => Bad news for us
- Example:
 - Minimize: var
 - var ≥ 5
 - var ≤ 3
- Does this case arise?
 - Yes! And very often!

Infeasible LPs

- Common Program construct:
 $i=i+1$ -> loop iteration, pointer arithmetic
- Convert this to an LP constraint:

$$i'_{\max} \geq i_{\max} + 1$$

$$i_{\max} \geq i'_{\max}$$

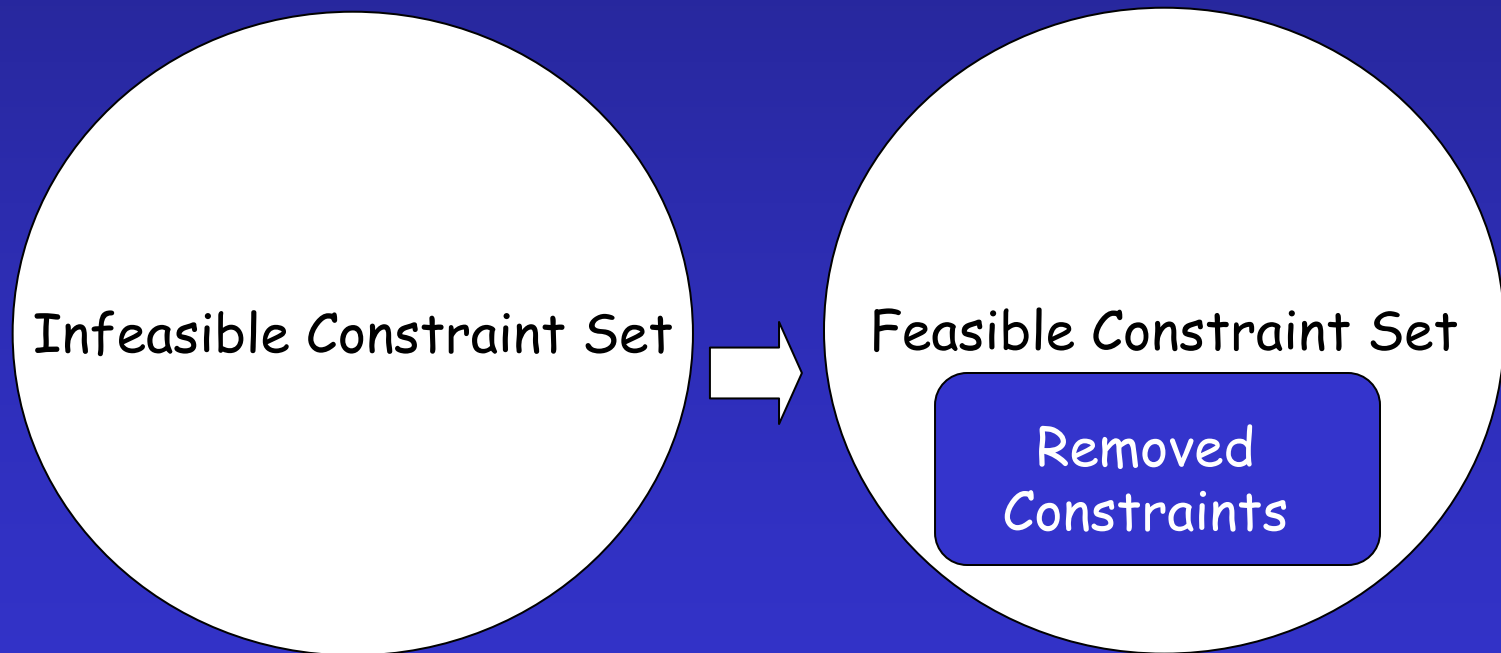
$$i'_{\min} \leq i_{\min} + 1$$

$$i_{\min} \leq i'_{\min}$$



Infeasible Set

Solving Infeasible LPs



Solving Infeasible LPs

- Optimization literature to the rescue
- Problem of IIS detection
 - IIS = Irreducibly Inconsistent Set
 - Smallest set of constraints such that
 - The constraint set is infeasible
 - Any subset of the constraint set is feasible
- Algorithms available to identify IISs

Solving Infeasible LPs

$i_max \geq i_min + 1$
 $i_min \geq i_max$
 $i_max \leq i_min + 1$
 $i_min \leq i_max$
 $a_max \geq i_max + 2$

Propagate SS

$i_max = \infty$
 $i_min = \infty$
 $a_max = \infty$

Solving Infeasible LPs

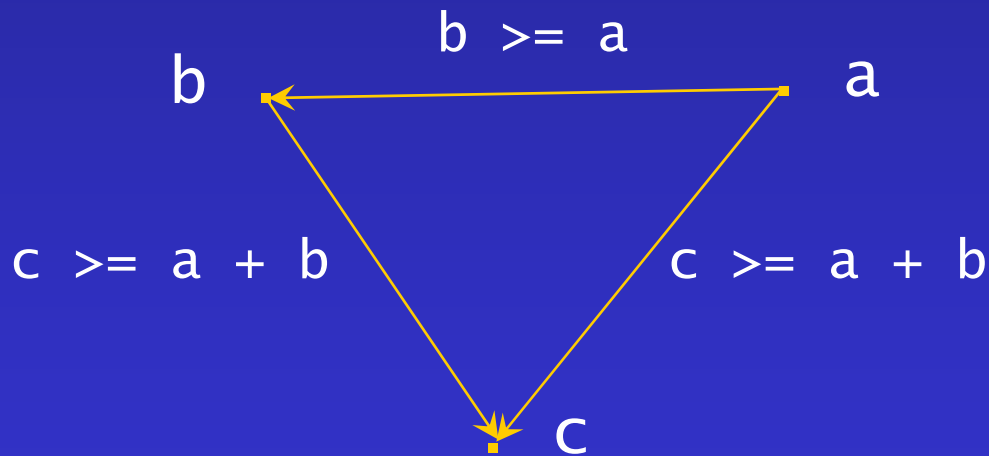
- Heuristic:
 - Identify IISs
 - Set variable values to $\infty/-\infty$
 - Ripple effect through constraint set
- Investigation underway (with Michael Ferris)
 - How effective is this heuristic?
 - Do we set more variables to $\infty/-\infty$ than required?

Other kinds of Solvers

- Hierarchical Solver
- Draw constraint dependency graph
- Identify *SCCs*
- Solve each *SCC*
- Propagate values

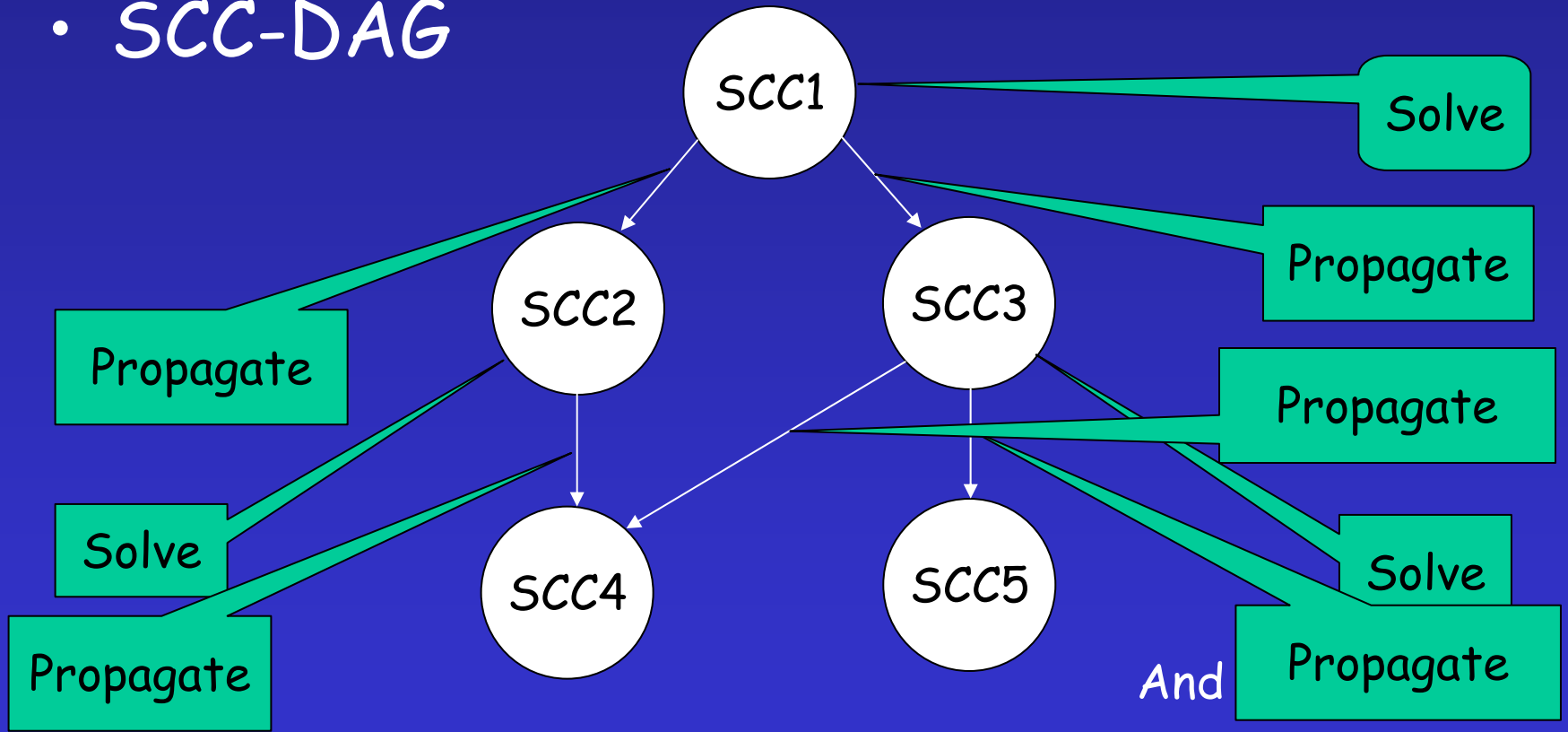
Dependency Graph

$$c \geq a + b, \quad b \geq a, \quad a \geq 2$$



Hierarchical Solver

- SCC-DAG

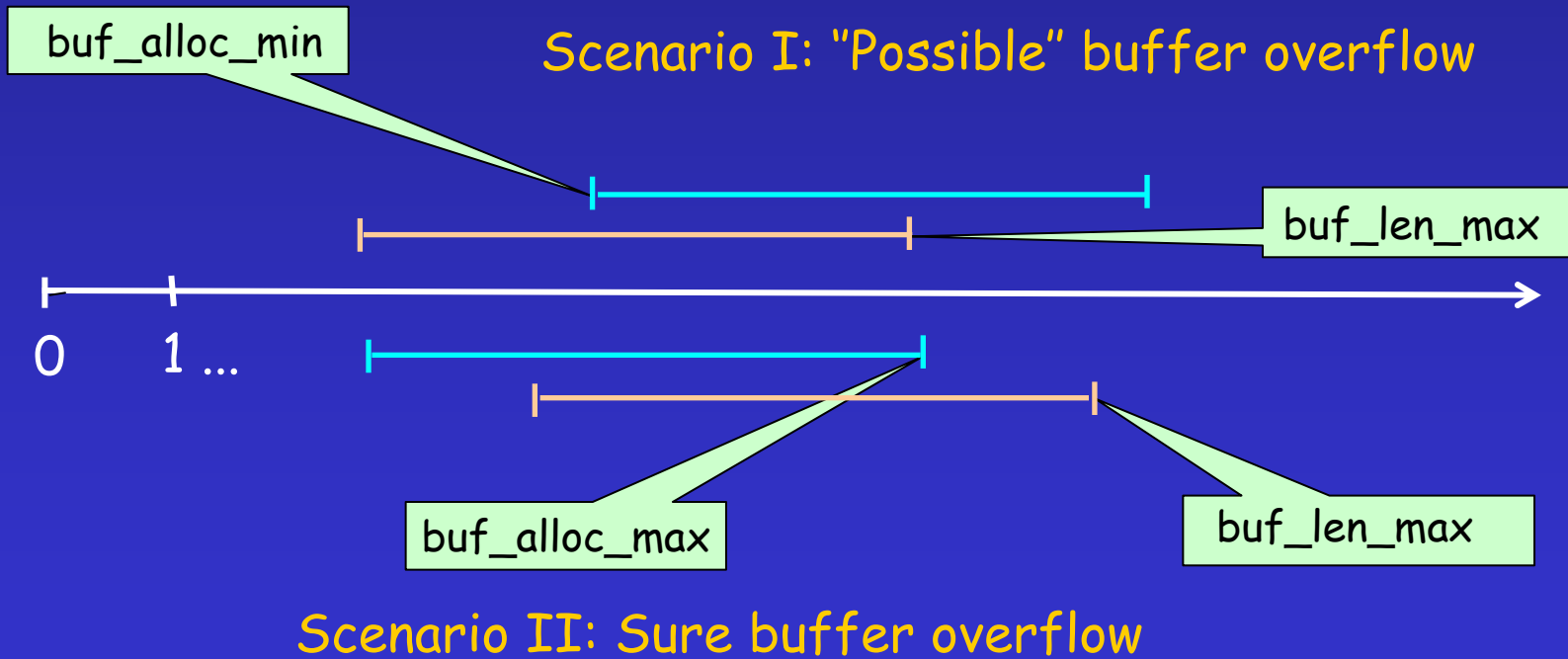


Hierarchical Solver

- Big LP broken down into smaller ones
- Can use different solvers for different SCCs
- Can solve in parallel (?)
- Status:
 - Most of the Infrastructure in place
 - To test on benchmarks

Detector: Basic Idea

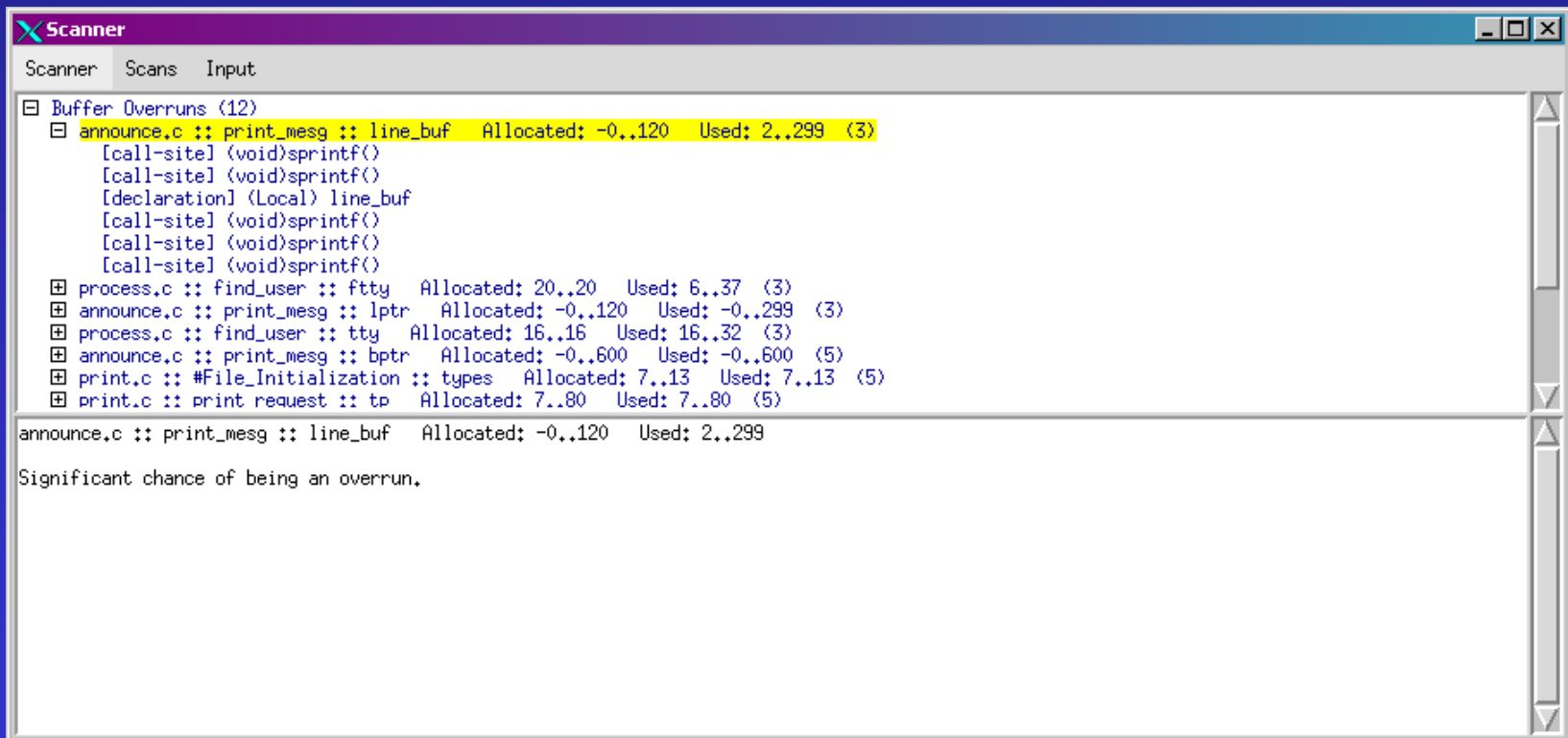
- Takes values from the LP solver
- Detects overruns based on the values



Detector Front End

- GUI built at Grammatech Inc.
- Allows trace-back:
 - Click on warning
 - Get to offending line on source code
 - Constraints also available for the more informed debugger
- Currently compiled for Linux

Detector Front End



The screenshot shows a window titled "Scanner" with a menu bar containing "Scanner", "Scans", and "Input". The main content area displays a tree view of memory overruns. The top-level item is "Buffer Overruns (12)", which is expanded to show a list of items. The first item, "announce.c :: print_mesg :: line_buf", is highlighted in yellow and shows "Allocated: -0..120 Used: 2..299 (3)". Below it are several "[call-site] (void)sprintf()" entries. Other items in the list include "process.c :: find_user :: ftty", "announce.c :: print_mesg :: lptr", "process.c :: find_user :: tty", "announce.c :: print_mesg :: bptr", "print.c :: #File_Initialization :: types", and "print.c :: print request :: tp". At the bottom of the window, there is a summary line for the highlighted item: "announce.c :: print_mesg :: line_buf Allocated: -0..120 Used: 2..299" and a note: "Significant chance of being an overrun."

```
Scanner
Scanner Scans Input
[-] Buffer Overruns (12)
  [-] announce.c :: print_mesg :: line_buf Allocated: -0..120 Used: 2..299 (3)
    [call-site] (void)sprintf()
    [call-site] (void)sprintf()
    [declaration] (Local) line_buf
    [call-site] (void)sprintf()
    [call-site] (void)sprintf()
    [call-site] (void)sprintf()
  [+] process.c :: find_user :: ftty Allocated: 20..20 Used: 6..37 (3)
  [+] announce.c :: print_mesg :: lptr Allocated: -0..120 Used: -0..299 (3)
  [+] process.c :: find_user :: tty Allocated: 16..16 Used: 16..32 (3)
  [+] announce.c :: print_mesg :: bptr Allocated: -0..600 Used: -0..600 (5)
  [+] print.c :: #File_Initialization :: types Allocated: 7..13 Used: 7..13 (5)
  [+] print.c :: print request :: tp Allocated: 7..80 Used: 7..80 (5)
announce.c :: print_mesg :: line_buf Allocated: -0..120 Used: 2..299
Significant chance of being an overrun.
```

Results

- 3 Benchmarks:
 - BSD Talk Daemon-4.2 (1000 lines)
 - WuFTP Daemon-2.5.0 (17000 lines)
 - Sendmail-8.7.6 (40000 lines)
- WuFTP Daemon: CERT-1999-13
- Sendmail-8.7.6: 1 known bug (BOON)
- Talk Daemon: ??

Results: Talk Daemon

- `line_buf`: [120..120] [2..299]
- Offending source code:

```
sprintf(line_buf[i], "...", var1, var2)
```

Could be as large as 256 bytes

- `snprintf` will solve the problem

Results: WuFTP Daemon

- `strcat(mapped_path, dir)`
- `mapped_path`: global array: 4096
- `dir`: there is a path to user input
- Result:
 `mapped_path` : [4096..4096] $[-\infty, \infty]$

Results: Sendmail

- Unreported overrun: caught by BOON
- Off by one bug:
 - BOON gets it as:
 - **dfname**: [20..20] [-∞..257]
 - We get it as:
 - **dfname**: [20..20] [-∞..∞]

Current Status

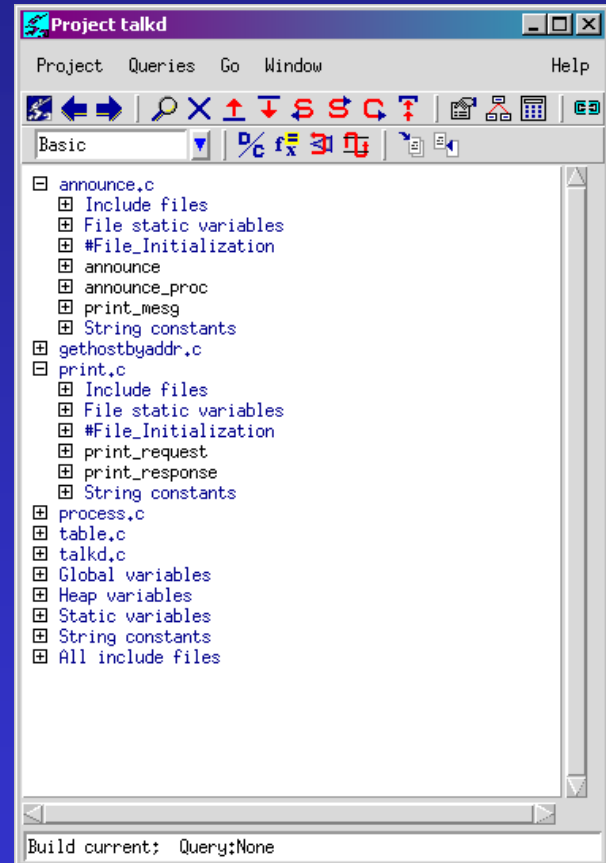
- Alpha version ready and working
- Acceptably quick:
 - Sendmail ~2 hours
 - Wuftpd, TalkD < 5 minutes
- User friendly GUI for trace-back

Next in line...

- The challenge: BIND ~50000 lines
 - Highly vulnerable
 - 4 CERT advisories in 2 years
- Hierarchical Solver results
- Context Sensitivity through summary functions
- Timeline: completion by mid-April

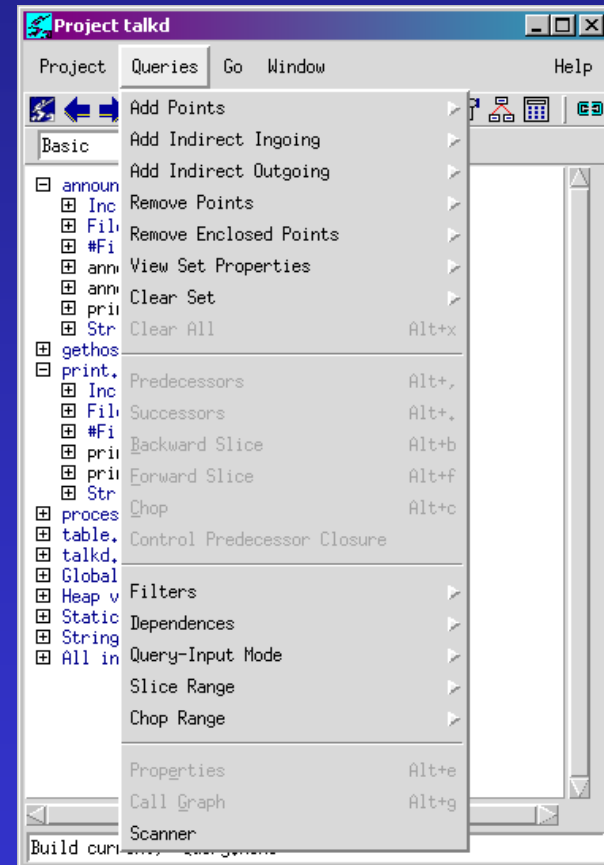
Tool Demo: TalkD

Step 1: Build the source code
using Codesurfer



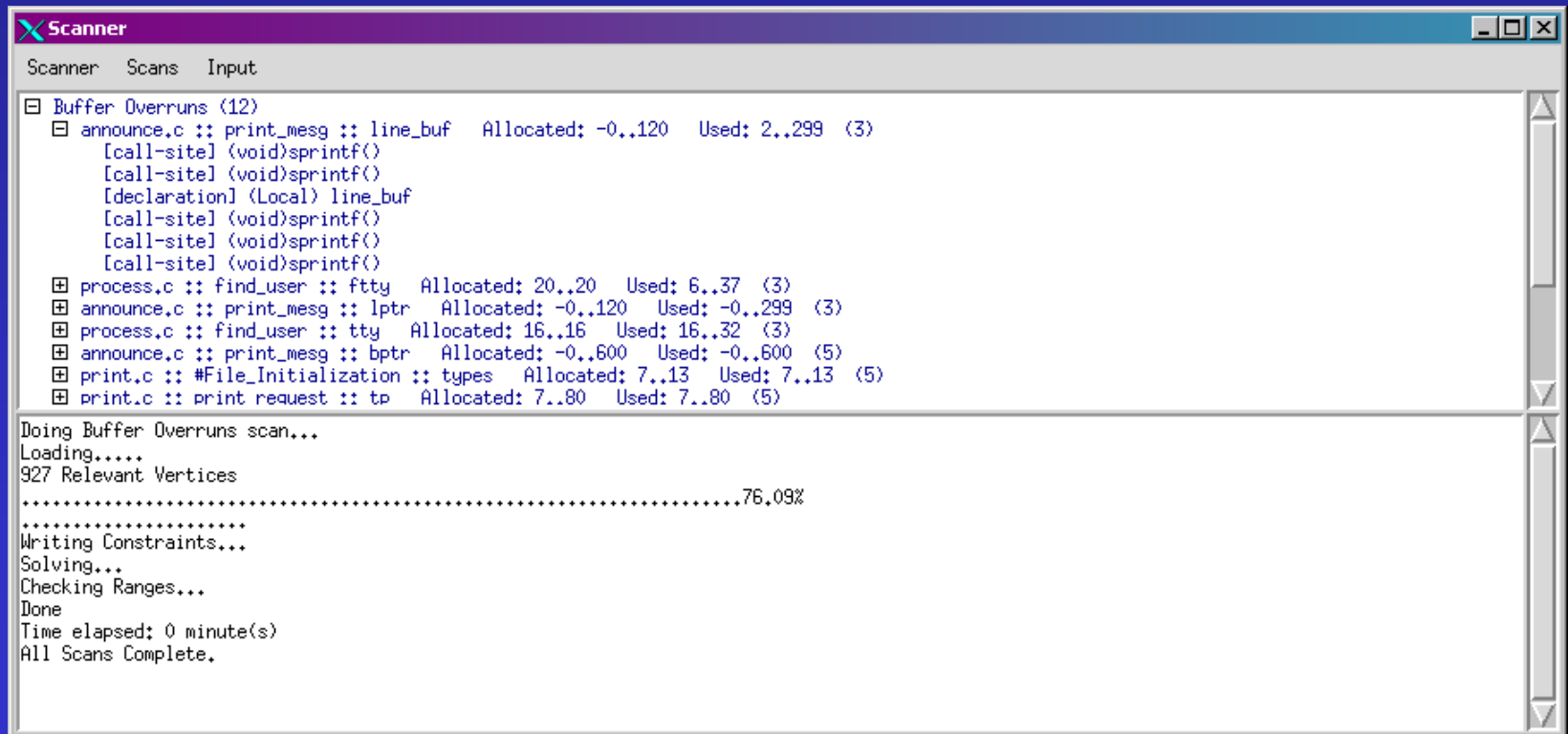
Tool Demo: TalkD

Step 2: Invoke the
Buffer Overrun Analyzer



Tool Demo: TalkD

Step 3: Follow the warnings to source code lines



```
Scanner Scans Input
[+] Buffer Overruns (12)
  [-] announce.c :: print_mesg :: line_buf Allocated: -0..120 Used: 2..299 (3)
    [call-site] (void)sprintf()
    [call-site] (void)sprintf()
    [declaration] (Local) line_buf
    [call-site] (void)sprintf()
    [call-site] (void)sprintf()
    [call-site] (void)sprintf()
  [+] process.c :: find_user :: fty Allocated: 20..20 Used: 6..37 (3)
  [+] announce.c :: print_mesg :: lptr Allocated: -0..120 Used: -0..299 (3)
  [+] process.c :: find_user :: tty Allocated: 16..16 Used: 16..32 (3)
  [+] announce.c :: print_mesg :: bptr Allocated: -0..600 Used: -0..600 (5)
  [+] print.c :: #File_Initialization :: types Allocated: 7..13 Used: 7..13 (5)
  [+] print.c :: print request :: tp Allocated: 7..80 Used: 7..80 (5)

Doing Buffer Overruns scan...
Loading.....
927 Relevant Vertices
.....76.09%
.....
Writing Constraints...
Solving...
Checking Ranges...
Done
Time elapsed: 0 minute(s)
All Scans Complete.
```

Tool Demo: TalkD

The image shows a development environment with two windows. The top window, titled 'Project talkd', displays a project tree on the left and a code editor on the right. The code editor shows the source file 'announce.c' with the following code:

```
(void)sprintf(line_buf[i], "talk: connection requested by %s@%s.",
    request->l_name, remote_machine);
sizes[i] = strlen(line_buf[i]);
max_size = max(max_size, sizes[i]);
i++;
(void)sprintf(line_buf[i], "talk: respond with: talk %s@%s",
    request->l_name, remote_machine);
sizes[i] = strlen(line_buf[i]);
max_size = max(max_size, sizes[i]);
i++;
(void)sprintf(line_buf[i], " ");
sizes[i] = strlen(line_buf[i]);
max_size = max(max_size, sizes[i]);
i++;
bptr = big_buf;
*bptr++ = '^'; /* send something to wake them up */
*bptr++ = '\n'; /* add a \n in case of raw mode */
*bptr++ = '\n';
for (i = 0; i < N_LINES; i++) {
    /* copy the line into the big buffer */
    lptr = line_buf[i];
    while (*lptr != '\0')
        *(bptr++) = *(lptr++);
    /* pad out the rest of the lines with blanks */
```

The bottom window, titled 'Scanner', shows the output of a memory scanner. It lists memory usage for various variables and functions:

```
Buffer Overruns (12)
  announce.c :: print_mesg :: line_buf Allocated: -0..120 Used: 2..299 (3)
    [call-site] (void)sprintf()
    [call-site] (void)sprintf()
    [declaration] (Local) line_buf
    [call-site] (void)sprintf()
    [call-site] (void)sprintf()
    [call-site] (void)sprintf()
  process.c :: find_user :: tty Allocated: 20..20 Used: 6..37 (3)
  announce.c :: print_mesg :: lptr Allocated: -0..120 Used: -0..299 (3)
  process.c :: find_user :: tty Allocated: 16..16 Used: 16..32 (3)
  announce.c :: print_mesg :: bptr Allocated: -0..600 Used: -0..600 (5)
  print.c :: #File_Initialization :: types Allocated: 7..13 Used: 7..13 (5)
  print.c :: print request :: tp Allocated: 7..80 Used: 7..80 (5)
```

Below the scanner output, the source code for the selected call-site is shown:

```
[call-site] (void)sprintf()
Source Code:
162: (void)sprintf(line_buf[i], "talk: connection requested by %s@%s.",
163: request->l_name, remote_machine);
```

Finally, the constraints for the selected call-site are listed:

```
Constraints:
{ 3} line_buf!len = + 31 + remote_machine!len + request.l_name!len = [43..299]
    remote_machine!len = [0..256]
    request.l_name!len = [12..12]
```

Thank You!

Questions?