Static Analysis Techniques to Detect Buffer Overrun Vulnerabilities

Vinod Ganapathy
University of Wisconsin

Introduction

- Buffer Overruns
  - An important class of vulnerabilities
  - CERT advisories:
    - 10 out of 37 in 2001
    - 8 out of 19 till July 2002
      - Eg. Microsoft IE [2002-04], Yahoo! Messenger [2002-16]
  - C is highly vulnerable
    - Systems programming mainly done in C
    - Easy to exploit

Goal

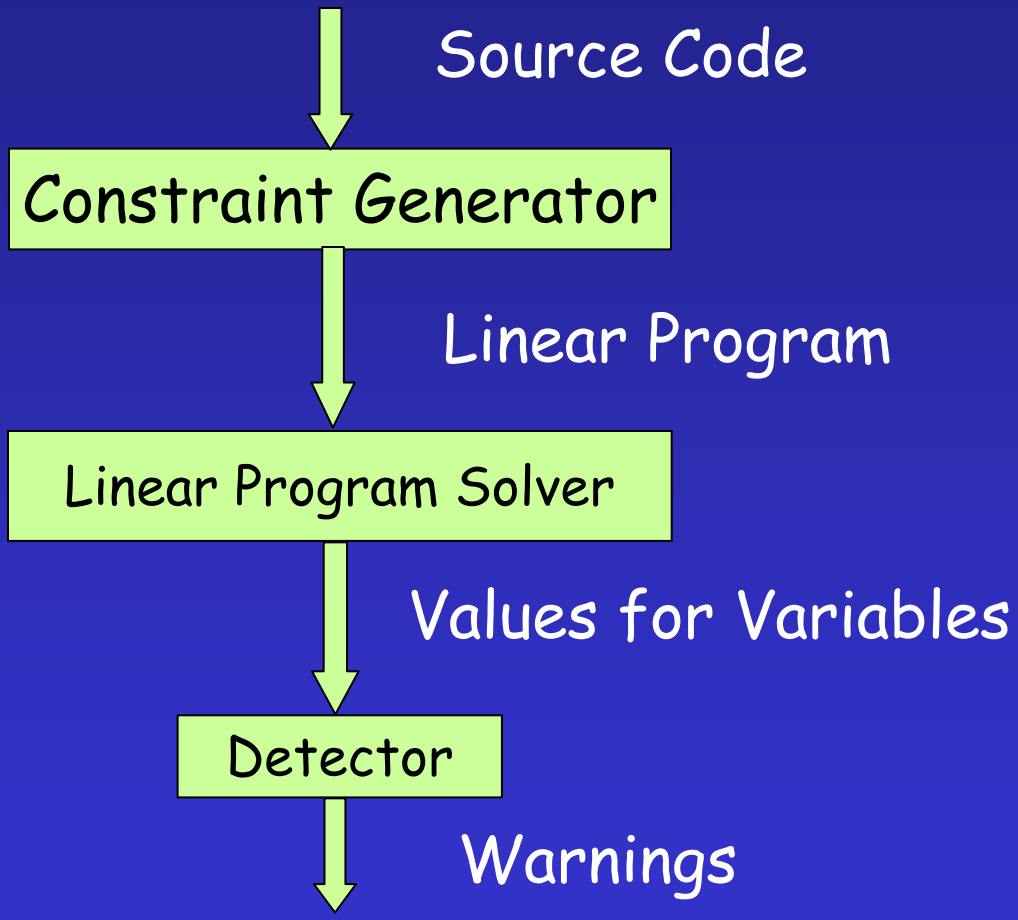
- Build a tool that will detect overruns in C source code
- Static Analysis or Run-time Analysis?
  - Run-time: *Avoids* vulnerabilities, tools available
    - Examples: CCured [Berkeley] , SafeC [Wisconsin]
  - Static Analysis: *Eliminates* vulnerabilities
    - Examples: Wagner's tool [Berkeley] , Our tool
- Our Tool:
  - Statically analyzes source code
  - Uses points to information
  - Uses Linear Programming (LP) for analysis

Idea

- Treat strings as Abstract Data Types
  - Each string buffer `buf` associated with four variables:
    - `buf_len_max, buf_len_min, buf_alloc_max, buf_alloc_min`
    - These define an interval of values
  - Each integer `int` associated with two variables:
    - `int_max, int_min`

- Model the semantics as constraints on variables
  - Eg. `strcpy(a, b)` `a_len_max >= b_len_max`
    `a_len_min <= b_len_min`

Tool Layout – An Overview



Constraint Generation

- Consists of the following phases:
  - *Program Analysis* : using Codesurfer
  - *Constraint Generation* : Codesurfer + Transducer
  - *Constraint format converter* [optional]

- Program Analysis with Codesurfer:
  - *Input* : Source program
  - *Internally Constructs* : CFG, PDG, Points to information
  - *Front End* : Traverse the constructs and produce constraints

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

- Our 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

Constraint Generation

- Transducer:
  - Produces constraints from an intermediate format produced by Codesurfer
- Constraint Format Converter:
  - Converts constraints into the format required by the LP solver
  - MPS format is popular
    - Column based format
  - Row based format ⟶ Column based format

Constraint Generation

- An Example

```
param2 = ID("hi");
strcpy(a, param2);
…
```

```
Char *ID (char *formal){
    return formal;
}
```

- A few Constraints:

call site :

```
formal_len_max      >=  3 ("h"+"i"+"\0")
formal_len_min      <=  3
ID_return_len_max   >=  formal_len_max
ID_return_len_min   <=  formal_len_min
```

assignment:

```
param2_len_max      >=  ID_return_len_max
param2_len_min      <=  ID_return_len_min
```

call to strcpy:

```
a_len_max           >= param2_len_max
a_len_min           <= param2_len_min
```

Linear Program Solver

- A Linear Program:
  - A set of constraints
  - An objective function
  - *Goal*: Maximize/Minimize the value of the objective function subject to the constraints
- In our case:
  - Constraints are available
  - Goal should be to:
    - Maximize the `min` variables [greatest lower bound]
    - Minimize the `max` variables [least upper bound]

Some LP Terminology

- Constraint set `C`, objective: `minimize F`
- *Optimal* solution: A finite assignment satisfying `C` and giving `F` its lowest finite value
- *Unbounded*
  - Give me an arbitrary finite value `M`
  - I'll find an assignment so that the value of `F < M`
  - e.g.: `Minimize -a`, subject to `a >= 5`
- *Infeasible*
  - No finite assignment satisfying `C` exists
  - eg: `C` is `a >= 5; a <= 2`

Linear Program Solver

- What should the objective function be?
  - Option 1:
    - Minimize: Σ (buf_len_max - buf_len_min) +
      Σ (buf_alloc_max - buf_alloc_min) +
      Σ (int_max - int_min)
    - Solve one LP
  - Option 2:
    - Minimize: buf_alloc_max - buf_alloc_min +
      buf_len_max - buf_len_min
    - Solve as many LPs as there are buffers

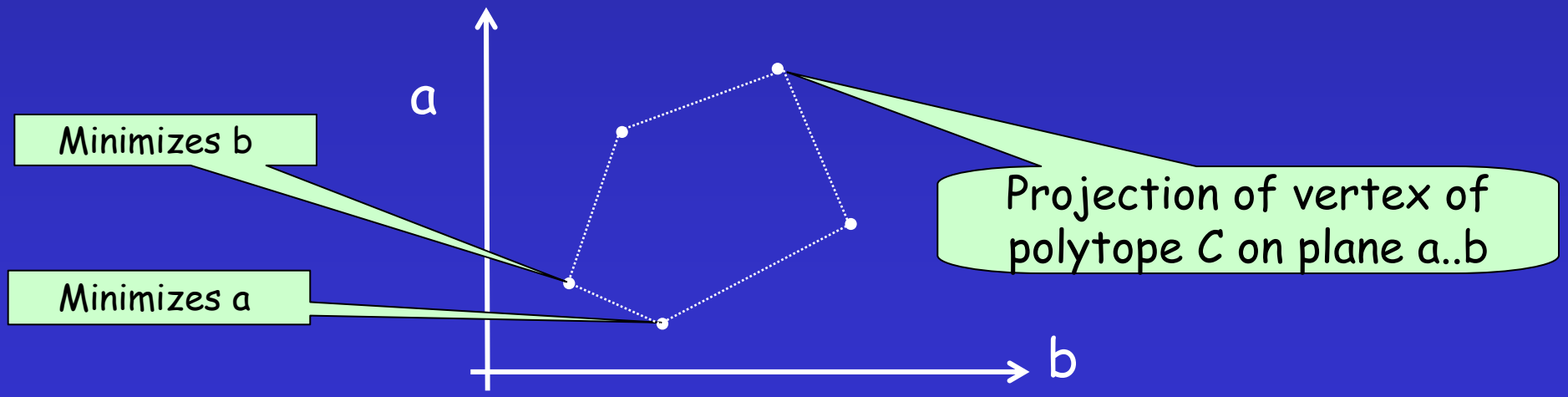
Linear Program Solver

- Option 3:
  - Minimize: (`buf_len_max` - `buf_len_min`)
  - Minimize: (`buf_alloc_max` - `buf_alloc_min`)
  - Solve twice as many LPs as there are buffers
- Option 4:
  - Minimize: `buf_alloc_max`
  - Minimize: `buf_len_max`
  - Maximize: `buf_alloc_min`
  - Maximize: `buf_len_min`
  - Solve four times as many LPs as there are buffers

Linear Program Solver

- Precision of the results obtained:
  - Option 1 < Option 2 < Option 3 < Option 4
- Reason:
  - Goal: Minimize a and b, subject to C
  - Objective function: Minimize: a + b

Most Precise

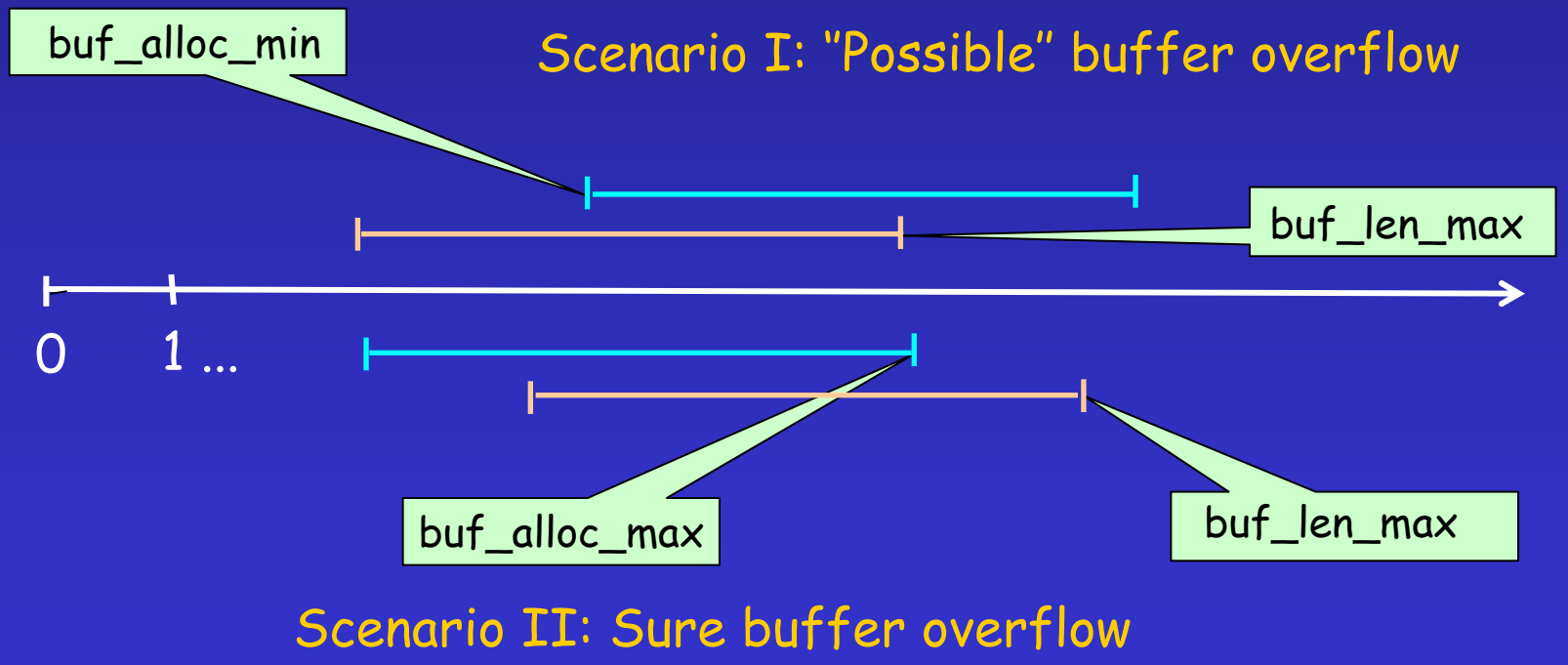


Linear Program Solver

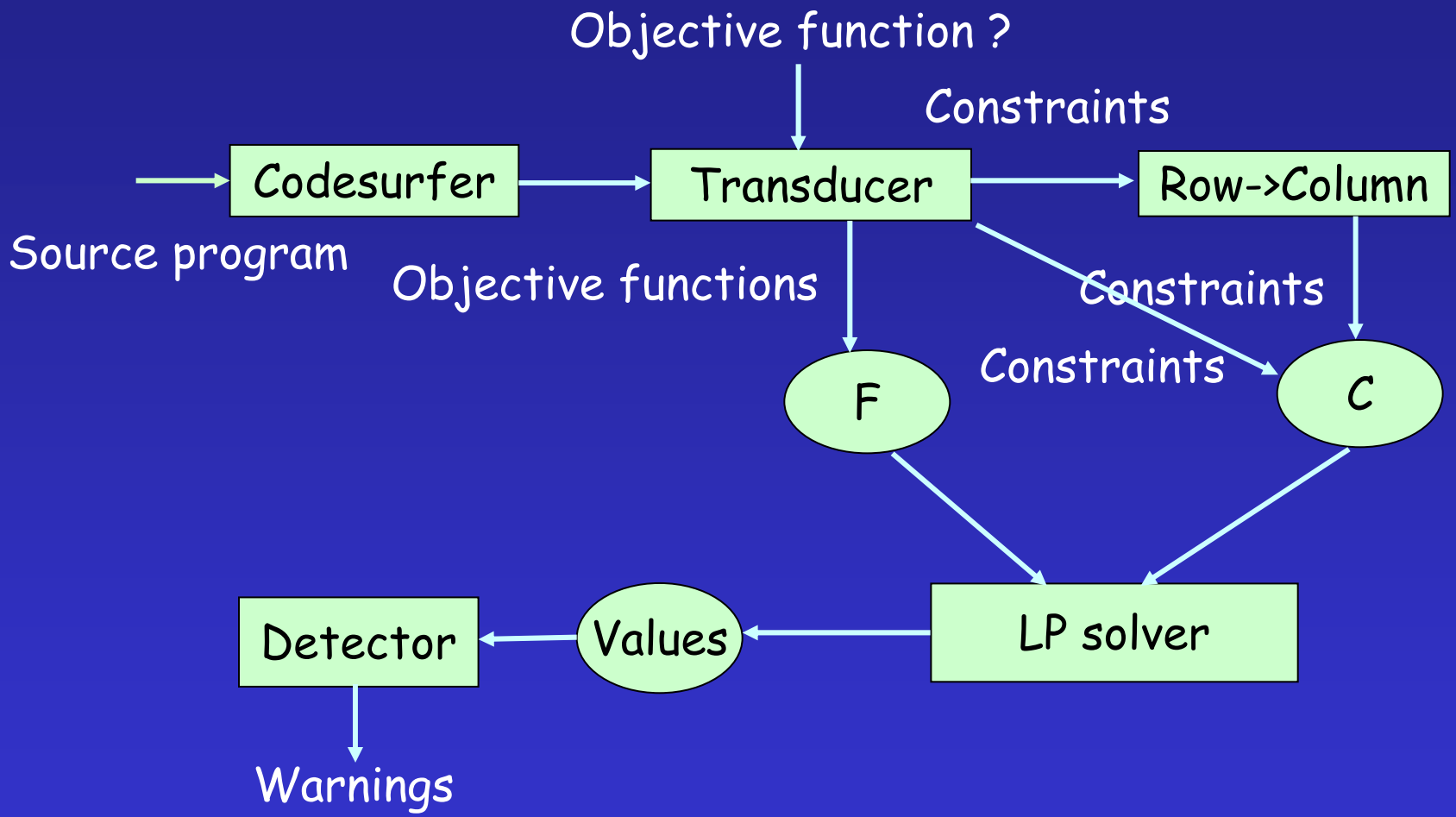
- All 4 options are in place
  - Lets you choose the level of precision desired
  - Way out when the LP is Unbounded
  - More precision => More LPs => Greater analysis time
- We use SoPlex
  - Sequential Object Oriented Simplex
  - Accepts Column based and Row based formats
  - May also try out CPLEX

Detector

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



Tool Layout - Summary



Some Problems !

- A few problems:
  - LP can only work with finite values
  - All problems need not have an optimal solution
  - What if your source program had `i = i + 1` ?
    - `i_max >= i_max + 1` is not accepted by any LP solver
  - What if your source program had:
    - `a = b + 5; b = a` ? (pointer arithmetic)
    - A subset of the constraints generated:

      ```
      a_len_min <= b_len_min - 5
      b_len_min <= a_len_min
      ```

    - This LP is infeasible!

Our solutions

- Optimal: We are done.
- Unbounded:
  - Some variable goes to infinity.
  - Produce objective function using choice 3 or 4
  - Still unbounded => those variables are infinite

- Infeasible (work in progress):
  - Remove the constraints that caused infeasibility
  - CPLEX has the capability to do so

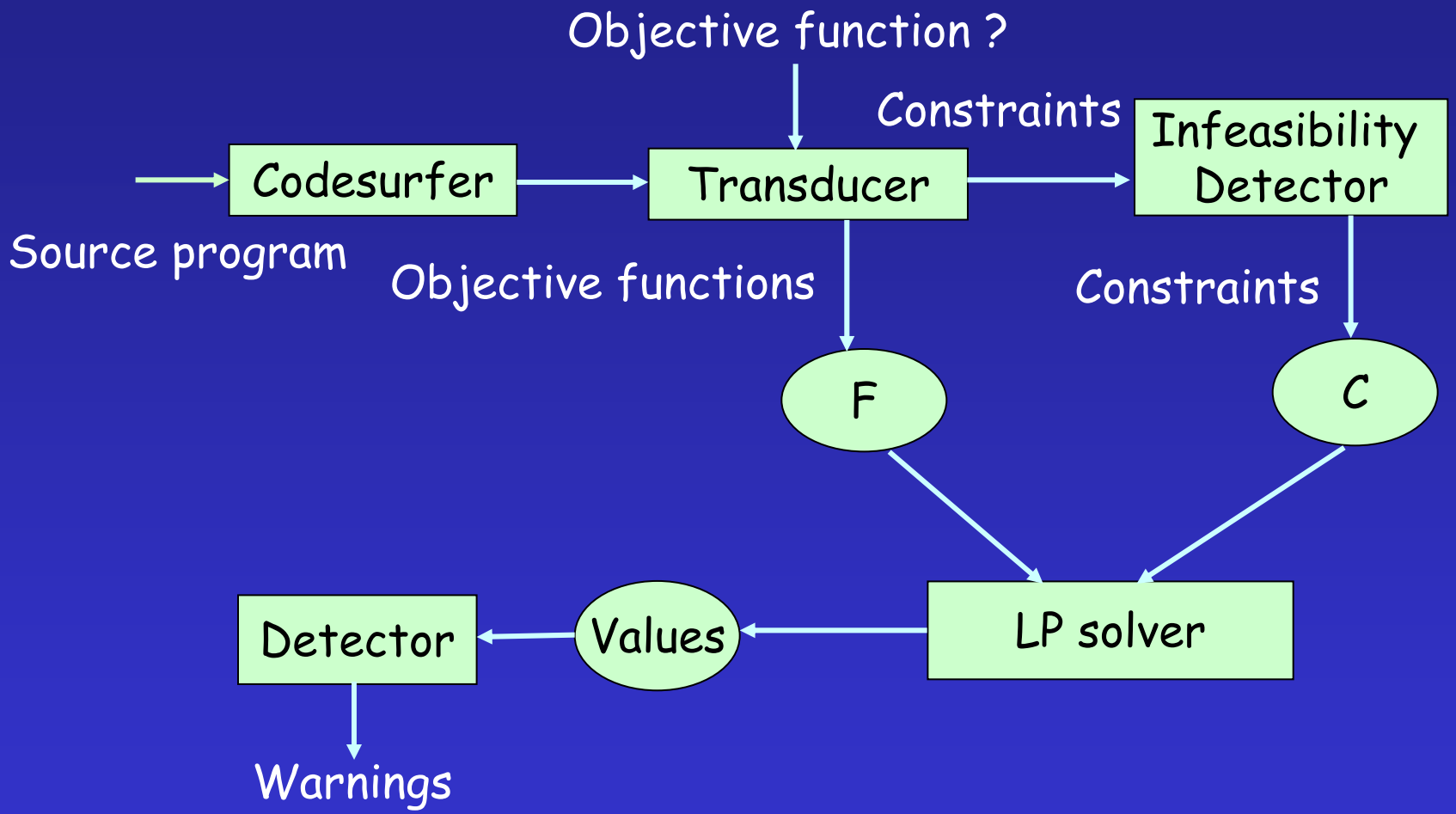
Infeasible constraints

- Some preliminary ideas:
  - Remove a subset C' of constraints from C
  - So that C – C' is feasible
  - Once C' is removed:
    - Have to set values of variables in C' to ∞ / -∞
    - May have to ripple this through C – C'
  - E.g.
    - Remove b_len_min <= a_len_min
    - Set both their values to -∞

Our solutions

- `i = i + 1`
  - Convert to `i_prime = i + 1`
  - Use `i_prime` in places where `i` is used
  - Problem here:
    - Increased value of `i` not being fed back to itself
    - We could miss potential overruns
    - What if I add `i = i_prime`? *Infeasible*!
  - Possible way out:
    - Use the above conversion
    - Remove the infeasible set; Set `i` to ∞

Tool Layout with Infeasibility Detector



Status

- Done:
  - Constraint Generation
  - Capability to solve multiple linear programs
  - Capability to analyze unbounded problems
  - Capability to reuse basis from previous solve
    - Faster solves as a result
- Doing
  - Working on the infeasibilty problem

Goals

- Biggest program tested SendMail 8.7.6
  - Constraint set is infeasible
- Target:
  - Immediate (next 1 month) :
    - The infeasibility problem
    - Performance issues
  - Short term :
    - Diagnostics: *Which statement* caused the overrun ?
  - Long term :
    - Alternatives to Linear Programming based analysis
    - Constraint Generator for binaries

Demo with BSD Talk Daemon

Constraint Generation

Analysis

Warnings

Thank You!

Questions?