Buffer Overrun Detection using Linear Programming and Static Analysis

Vinod Ganapathy, Somesh Jha
{vg, jha}@cs.wisc.edu
University of Wisconsin-Madison

David Chandler, David Melski, David Vitek
{chandler, melski, dvitek}@grammatech.com
Grammatech Inc., Ithaca, New York
The Problem: Buffer Overflows

• Highly exploited class of vulnerabilities
  - Legacy programs in C are still vulnerable
  - “Safe” functions can be used unsafely

• Need:
  - Automatic techniques that will assure code is safe before it is deployed
The Solution

- Use static program analysis
- Produce a list of possibly vulnerable program locations
- Couple buffer overrun warnings with code understanding techniques
Our Contributions

• Program Analysis:
  – Incorporated buffer overrun detection in a program understanding tool
    • Program slicing, Data predecessors,...
  – Use of procedure summaries to make buffer overrun analysis context-sensitive

• Constraint Resolution:
  – Use of linear programming to solve a range analysis problem
Roadmap of Talk

• Tool Architecture
  - Constraint Generation
  - Constraint Resolution
  - Producing Warnings
• Adding Context Sensitivity
• Results
• Future work and Conclusions
Tool Architecture

CodeSurfer

C Source Code → CFGs, PDGs, SDG → Linear Constraints

Constraint Generator

Detection Front-End

Buffer Overrun Warnings ↘

Ranges for Variables ←

Linear Constraints

Linear Constraints

Taint Analyzer

Constraint Solver
CodeSurfer

C Source Code → CFGs, PDGs, SDG → Linear Constraints → Linear Constraints → Buffer Overrun Warnings → Ranges for Variables → Constraint Solver

Constraint Generator

Taint Analyzer

Detection Front-End
CodeSurfer

• Commercial tool from Grammatech Inc.
• Code understanding framework
  – Inter-procedural slicing, chopping…
• Internally constructs:
  – Control Flow Graph (CFG)
  – Program Dependence Graphs (PDG)
  – System Dependence Graph (SDG)
• Incorporates results of pointer analysis
  – Helps reduce the number of warnings
The Role of CodeSurfer

- **Program Analysis Framework:**
  - Use internal data structures to generate constraints

- **Detection Front-end:**
  - Link each warning to corresponding lines of source code through the System Dependence Graph
  - Interactive front-end
Constraint Generation

- C Source Code
- CFGs, PDGs, SDG
- Linear Constraints
- Buffer Overrun Warnings
- Ranges for Variables
- Linear Constraints
- Constraint Solver
- Taint Analyzer
- Detection Front-End

CodeSurfer

Constraint Generator
Constraint Generation

- Four kinds of program points result in constraints:
  - Declarations
  - Assignments
  - Function Calls
  - Function Return
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 (tgt, src)
  tgt_len_max ≥ src_len_max
  tgt_len_min ≤ src_len_min
Constraint Generation

if(…)

`strcpy(tgt, srcA)`

tgt_len_max ≥ srcA_len_max
tgt_len_min ≤ srcA_len_min

`strcpy(tgt, srcB)`

tgt_len_max ≥ srcB_len_max
tgt_len_min ≤ srcB_len_min
Constraint Generation Methods

• Order of statements:
  - Flow-Sensitive Analysis:
    • Respects program order
  - Flow-Insensitive Analysis:
    • Does not respect program order

• Function Calls:
  - Context-Sensitive modeling of functions:
    • Respects the call-return semantics
  - Context-Insensitive modeling of functions:
    • Ignores call-return semantics => imprecise
Constraint Generation

• Constraints generated by our tool:
  - Flow-insensitive
  - Context-sensitive for some library functions
  - Partly Context-sensitive for user defined function
    • Procedure summaries
Taint Analysis

C Source Code → CFGs, PDGs, SDG → Linear Constraints → Linear Constraints

Detection Front-End

Buffer Overrun Warnings → Ranges for Variables

Constraint Solver

CodeSurfer

Constraint Generator

Taint Analyzer
Taint Analysis

- Removes un-initialized constraint variables
  - Un-modeled library calls
  - Un-initialized program variables
- Required for solvers to function correctly
Constraint Solvers

CodeSurfer

C Source Code → CFGs, PDGs, SDG

Constraint Generator

Linear Constraints

Taint Analyzer

Linear Constraints

Detection Front-End

Buffer Overrun Warnings

Ranges for Variables

Constraint Solver
Constraint Solvers

• Abstract problem:
  - Given a set of constraints on max and min variables
  - Get tightest possible fit satisfying all the constraints

• Our approach:
  - Model and solve as a linear program
Constraint Solvers

• We have developed two solvers:
  - Vanilla solver
  - Hierarchical solver

- Speed of Analysis:
  - fast (Vanilla)
  - slow (Hierarchical)

- Precision of results:
  - low (Vanilla)
  - high (Hierarchical)
Linear Programming

- An objective function $F$
- Subject to: A set of constraints $C$

**Example:**

Maximize: $x$

Subject to:

$x \leq 3$
Why Linear Programming?

- Can support arbitrary linear constraints
- Commercial linear program solvers are highly optimized and fast
- Use of *linear programming duality* for diagnostics
  - Can be used to produce a “witness” trace leading to the statement causing the buffer overrun
Vanilla Constraint Solver

- **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
Vanilla Constraint Solver

• However, it can be shown that:
  Min: \( \Sigma (\text{max vars}) - \Sigma (\text{min vars}) \)
  Subject to: Set of Constraints
  yields the same solution for each variable

• Solve just one Linear Program and get values for all variables!
Vanilla Constraint Solver

- Why is this method imprecise?
  - *Infeasible* linear programs
  - Why do such linear programs arise?
- Deals with infeasibility using an approximation algorithm
- See paper for details
Detection Front-End

C Source Code → CFGs, PDGs, SDG → Linear Constraints → Linear Constraints → Detection Front-End

CodeSurfer
Constraint Generator
Taint Analyzer

Detection Front-End

Ranges for Variables
Buffer Overrun Warnings

Linear Constraints

Constraint Solver
Detection Front-End

Scenario I: "Possible" buffer overflow

buf_alloc_min

buf_len_max
Detection Front-End

Scenario II: Sure buffer overflow

buf_alloc_min

buf_len_max

0 1
Roadmap of Talk

• Tool Architecture
  - Constraint Generation
  - Constraint Resolution
  - Producing Warnings

• Adding Context Sensitivity

• Results

• Future work and Conclusions
Context-Insensitive Analysis

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

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

int foobar (int z) {
    int i;
    i = z + 1;
    return i;
}
Context-Insensitive Analysis

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

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

int foobar (int z) {
    int i;
    i = z + 1;
    return i;
}
```
Context-Insensitive Analysis

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

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

int foobar (int z) {
    int i;
    i = z + 1;
    return i;
}
```
Context-Insensitive Analysis

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

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

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

Result: \( x = y = [6..31] \)
Adding Context Sensitivity

• **Make user functions context-sensitive**
  - e.g. wrappers around library calls
• **Inefficient method: Constraint inlining**
  ☺ Can separate calling contexts
  ☹ Large number of constraint variables
  ☹️ Cannot support recursion
Adding Context Sensitivity

- Efficient method: Procedure summaries
- Basic Idea:
  - Summarize the called procedure
  - Insert the summary at the call-site in the caller
  - Remove false paths
Adding Context Sensitivity

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

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

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

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

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

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

Summary: $i = z + 1$

$x = 5 + 1$

$y = 30 + 1$
Adding Context Sensitivity

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

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

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

Jump Functions
Adding Context Sensitivity

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

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

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

No false paths 😊  
$x = [6..6]$  
$y = [31..31]$  
$i = [6..31]$
Adding Context Sensitivity

• Computing procedure summaries:
  - In most cases, reduces to a shortest path problem
  - Other cases, Fourier-Motzkin variable elimination
Roadmap of Talk

• Tool Architecture
  - Constraint Generation
  - Constraint Resolution
  - Producing Warnings
• Adding Context Sensitivity
• Results
• Future work and Conclusions
# Results: Overruns Identified

<table>
<thead>
<tr>
<th>Application</th>
<th>LOC</th>
<th>Warnings</th>
<th>Vulnerability</th>
<th>Detected?</th>
</tr>
</thead>
<tbody>
<tr>
<td>WU-FTPD-2.5.0</td>
<td>16000</td>
<td>139</td>
<td>CA-1999-13</td>
<td>Yes</td>
</tr>
<tr>
<td>WU-FTPD-2.6.2</td>
<td>18000</td>
<td>178</td>
<td>None</td>
<td>14 New</td>
</tr>
<tr>
<td>Sendmail-8.7.6</td>
<td>38000</td>
<td>295</td>
<td>Identified by Wagner et al.</td>
<td>Yes</td>
</tr>
<tr>
<td>Sendmail-8.11.6</td>
<td>68000</td>
<td>453</td>
<td>CA-2003-07</td>
<td>Yes, but...</td>
</tr>
</tbody>
</table>
Results: Context Sensitivity

• WU-FTPD-2.6.2: 7310 ranges identified
• Constraint Inlining:
  - 5.8x number of constraints
  - 8.7x number of constraint variables
  - 406 ranges refined in at least one calling context

• Function Summaries:
  - 72 ranges refined
Roadmap of Talk

• Tool Architecture
  - Constraint Generation
  - Constraint Resolution
  - Producing Warnings
• Adding Context Sensitivity
• Results
• Future work and Conclusions
Conclusions

• Built a tool to detect buffer overruns
• Incorporated in a program understanding framework
• Current work:
  - Adding Flow Sensitivity
  - Reducing the number of false warnings while still maintaining scalability
Buffer Overrun Detection using Linear Programming and Static Analysis

Vinod Ganapathy, Somesh Jha
{vg,jha}@cs.wisc.edu
University of Wisconsin-Madison

David Chandler, David Melski, David Vitek
{chandler,melski,dvitek}@grammatech.com
Grammatech Inc., Ithaca, New York