Automatically Identifying Critical Input and Code Regions in Applications

Michael Carbin and Martin Rinard

Massachusetts Institute of Technology

Computer Science and Artificial Intelligence Laboratory
Open Challenges in Program Analysis

- **Reliability**
  - Memory Safety, Memory Leaks, Data Structure Corruption, Error Recovery

- **Performance**
  - Excess Running Times, Excess Power Consumption

- **Security**
  - Code Injection Attacks
New Class of Solutions

- **Memory Safety** (OSDI’04)
- **Data Structure Repair** (ICSE’05, ISSTA’06)
- **Bounded Memory Consumption** (ISMM’07)
- **Automatic Error Recovery** (ASPLOS’09)
- **Automatic Patching** (SOSP’09)
- **Performance Profiling** (ICSE’10)
- **Reduced Power Consumption** (PLDI10, sub OSDI’10)

**UNSOUND**
Automatic Patching Against Exploits (SOSP '09)

- **Patch** restores learned invariants.
- Skip call to a never before seen function.
- Set variable to a previously seen value.
Visualizing Execution

- Program may transition to previously unreachable program states.

- Solution is to identify regions that won’t get us into trouble.
The Takeaway

• **Critical** Input and Code Regions
  
  • **Hard** functional correctness requirements - must.

• **Forgiving** Input and Code Regions
  
  • **Soft** functional correctness requirements - may.

• Regions are characterized by application’s response to change.
  
  • **Critical** - intolerant to change.

  • **Forgiving** - tolerant to change.

• We can automatically determine regions by modeling application response.
Critical and Forgiving Regions
Critical Input Regions in GIF Image Conversion

- metadata
- configuration
- palette
- data
- comments
Modeling Application Response
Modeling Response

• Behavioral Distance

• Change in input region causes change in the behavior of the application.

\[
\text{distance} = \frac{8}{9}
\]
Modeling Response (cont.)

• **Computation Influence**

  - Contribution of input region to computation’s intermediate results.

<table>
<thead>
<tr>
<th>input</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
</table>

**operations**

- \(x = \text{add}\{3,4\}, \{2\}\)
- \(y = \text{add}\ (x, \{1\})\)
- \(z = \text{neg}\{5,6,7,8\}\)
- \(a = \text{sub}\{3,4\}, z\)
- \(b = \text{mult}\{1\}, \{2\}\)

\[
\begin{align*}
\text{operations} & = \text{influence}\{3,4\}, \\
\text{Influence Trace} & = 3
\end{align*}
\]
Classification Scheme

1. Exploration

2. Input Region Classification

3. Code Region Classification
### Exploration Phase

#### Inputs

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>char</th>
<th>char</th>
<th>short</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

#### Traces

- ET
- IT
- ET
- ET
- ET
- ET
- ET
- ET
Input Region Classification

• For each region:

\[
\text{if } \text{distance( ET, ET) is large:}
\]
return “critical”

\[
\text{if } \text{influence(○, IT) is large:}
\]
return “critical”

else
return “forgiving”

- determined by clustering
- parameterized threshold
Code Region Classification

• Given input region classifications

  • For each basic block, identify accessed input regions and aggregate input region classifications.

    • Majority are critical => critical code region

    • Majority are forgiving => forgiving code region.

    • No majority => mixed code region.
Evaluation Methodology

• Input Region Classification
  • Compare automatic classifications to golden test oracle.

• Code Region Classification
  • Manually determine if code classifications are sensible.
Benchmarks

• Three image processing libraries
  - gif (5KLOC), png (36KLOC), jpeg (35KLOC)

• One Task
  - Convert a image (gif, png, or jpeg) to a bitmap file.

• Five inputs per benchmark
  - Each input exercises different functionality.
Constructing Golden Classifications

- Given input of length n. Run the program to produce the de facto output.

- For each of the n bytes of the given input, generate m fuzzed inputs by replacing the value of the byte with a random value.

- For each of the \( n \times m \) fuzzed inputs, run the program to produce \( n \times m \) fuzzed outputs.

- Compute the dissimilarity between the de facto output and each of the \( n \times m \) fuzzed outputs.

- For each byte, if one of the m fuzzed outputs is more than 10% dissimilar, classify as critical. Otherwise, classify as forgiving.
## Input Region Classification Results

- Precision: % of critical classification’s that were correct.
- Recall: % of critical classifications that were identified.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>CC*</th>
<th>IC**</th>
<th>CF*</th>
<th>IF**</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>png</td>
<td>9580</td>
<td>5</td>
<td>451</td>
<td>18</td>
<td>99%</td>
<td>99%</td>
</tr>
<tr>
<td>gif</td>
<td>6951</td>
<td>23</td>
<td>2149</td>
<td>1412</td>
<td>99%</td>
<td>83%</td>
</tr>
<tr>
<td>jpeg</td>
<td>5123</td>
<td>27</td>
<td>542</td>
<td>1831</td>
<td>99%</td>
<td>73%</td>
</tr>
</tbody>
</table>

*higher is better  **lower is better
Role of Behavioral Distance (GIF)

- Clean Separation
- Large behavioral distance implies large dissimilarity
- Behavioral distance does not account for all dissimilarity.
• Computation influence correlates strongly with dissimilarity.
PNG Code Classification Results

B-Critical
- png_handle_IHDR
- png_memcpy_check
- png_handle_tRNS
- png_do_read_transformations
- png_read_start_row

C-Critical
- inflate_table
- inflate_fast
- inflate_table
- png_read_row
- png_read_finish_row
- updatewindow

Forgiving
- png_handle_tIME
- png_handle_gAMA
- png_handle_IEND
- png_handle_pHYs

Mixed
- png_crc_read
- png_crc_error
- png_get_int_31
- png_read_data
GIF Code Classification Results

- B-Critical
  - DGifGetLine
  - DGifGetImageDesc

- C-Critical
  - DGifDecompressLine
  - DGifDecompressInput

- Mixed
  - DGifGetWord
  - DGifDecompressInput

• Few functions because of small size of the benchmark
## JPEG Code Classification Results

<table>
<thead>
<tr>
<th>B-Critical</th>
<th>C-Critical</th>
<th>Forgiving</th>
<th>Mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>jpeg_calc_output_dims</td>
<td>decode_mcu</td>
<td>examine_app0</td>
<td>get_interesting_appn</td>
</tr>
<tr>
<td>jinit_d_main_controller</td>
<td>jpeg_huff_decode</td>
<td>examine_app14</td>
<td>next_marker</td>
</tr>
<tr>
<td>free_pool</td>
<td>jpeg_fill_bit_buffer</td>
<td></td>
<td>read_markers</td>
</tr>
<tr>
<td>alloc_large</td>
<td>jpeg_idct_islow</td>
<td></td>
<td>skip_input_data</td>
</tr>
<tr>
<td>alloc_sarray</td>
<td></td>
<td></td>
<td>skip_variable</td>
</tr>
<tr>
<td>start_pass_huff_decoder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>process_data_simple_main</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jpeg_finish_decompress</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Misclassifications in Mixed category due to lower recall.
Limitations and Future Work

• Benchmarks
  • all image conversion

• Behavioral Influence
  • Does not capture all behaviors of interest.

• Computation Influence
  • Does not track indirect (pointer arithmetic) influence.
Conclusion

• New approaches to program analysis are enabled by the distinction between:

  • **Critical** Input and Code Regions - must.
  
  • **Forgiving** Input and Code Regions - may.

• Input and Code Regions are determined by application’s response to change.

  • **Critical** - intolerant to change.
  
  • **Forgiving** - tolerant to change.

• We can automatically determine regions by modeling application response.
Thanks
Related Work

• Perturbation Analysis (Voas ’92)

• Critical and Forgiving (Rinard ’05)
  • Definition and manual exploration.

• Critical Memory (Pattabiraman ’08)
  • Programmers manually allocate memory in a critical heap that provides probabilistic memory safety

• Continuity (Chaudhuri ’10)
Implementation

• LLVM-based static bitcode instrumentor and dynamic runtime.

• Currently requires source code.

  • C, C++, Java, Ada, MSIL

  • x86 -> LLVM would eliminate need for source.

• Runtime tracks influence (like taint tracing) of input bytes on each operand and memory location.

  • Shadow Execution (registers, stack, memory, filesystem).

  • External library model
Input Specification Generator

- Groups input bytes by *affinity*: #together/#total

### Influence Trace

<table>
<thead>
<tr>
<th>Op</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1,2}</td>
</tr>
<tr>
<td>2</td>
<td>{3,4}</td>
</tr>
<tr>
<td>3</td>
<td>{1}</td>
</tr>
<tr>
<td>4</td>
<td>{2}</td>
</tr>
<tr>
<td>5</td>
<td>{1}</td>
</tr>
<tr>
<td>6</td>
<td>{5,6,7,8}</td>
</tr>
<tr>
<td>7</td>
<td>{5,6,7,8}</td>
</tr>
<tr>
<td>8</td>
<td>{2}</td>
</tr>
<tr>
<td>9</td>
<td>{3,4}</td>
</tr>
<tr>
<td>10</td>
<td>{3,4}</td>
</tr>
</tbody>
</table>

### Affinity

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1,2)</td>
<td>1/5 = .2</td>
<td>N</td>
</tr>
<tr>
<td>A(2,3)</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>A(3,4)</td>
<td>3/3 = 1</td>
<td>Y</td>
</tr>
<tr>
<td>A(4,5)</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>A(5,6)...A(7,8)</td>
<td>2/2 = 1</td>
<td>Y</td>
</tr>
</tbody>
</table>
Evaluating Input Region Classifications

• Precision-Recall:
  
  • **True Positive**: Correct Critical (CC)
  
  • **False Positive**: Incorrect Critical (IC)
  
  • **True Negatives**: Correct Forgiving (CF)
  
  • **False Negatives**: Incorrect Forgiving (IF)