Efficient Detection of All Pointer and Array Access Errors

Todd M. Austin (presenting)
Scott E. Breach
Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin – Madison
Outline

- Memory Access Errors
- Motivation
- Basic Methodology
- Check Optimization
- Experimental Evaluation
- Summary and Future Work
Memory Access Errors

A memory access error is any dereference which accesses storage outside of the intended referent.

- *spatial access error*, outside of referent address bounds
- *temporal access error*, outside of referent lifetime
- *intended referent*, the memory object used to create the pointer value (in C, reference created with `malloc()` or `&`)
Motivation

- programming errors are costly
- aggravated by abstraction, parallel programming, and programming in the large
- memory access errors are difficult to find and fix
  - arise under exceptional conditions
  - difficult to reproduce
  - difficult to correlate fault to error
- therefore: need both efficiency and complete coverage
Safe Pointer Abstraction

Valid Dereference:

\[(\text{capability} \in \text{capStore}) \land (\text{base} \leq \text{value} \leq \text{base} + \text{size} - \text{sizeof(*value)})\]
### Safe Pointer Creation, Manipulation, and Use

\[
\text{malloc}(\text{sizeof(struct Foo)}) \quad \Rightarrow \quad \begin{array}{c|c|c|c} \text{value} & \text{base/size} & \text{storage class} & \text{capability} \\ \hline 2000 & 2000/32 & \text{Heap} & 52 \end{array}
\]

\[
\&p->\text{next}->\text{status} \quad \Rightarrow \quad \begin{array}{c|c|c|c} \text{value} & \text{base/size} & \text{storage class} & \text{capability} \\ \hline 4032 & 4032/4 & \text{Local} & 36 \end{array}
\]

<table>
<thead>
<tr>
<th>value</th>
<th>base/size</th>
<th>storage class</th>
<th>capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>1000/100</td>
<td>Heap</td>
<td>46</td>
</tr>
</tbody>
</table>

\[
\llbracket \text{value} + 6 \rrbracket \quad \Rightarrow \quad \begin{array}{c|c|c|c} \text{value} & \text{base/size} & \text{storage class} & \text{capability} \\ \hline 1016 & 1000/100 & \text{Heap} & 46 \end{array}
\]

\[
\text{deref} \quad \begin{array}{c|c|c|c} \text{value} & \text{base/size} & \text{storage class} & \text{capability} \\ \hline 1010 & 1000/100 & \text{Heap} & 46 \end{array} \quad \Rightarrow \quad \text{perform access check}
\]
Program Transformations

C Program

Parse/Semantic Analysis

Pointer Conversion

Operator Conversion

Check Insertion

Code Generation

Implemented with C++ compiler

Run-time Library

Compile/Link

Safe Executable
Requisites of Complete Coverage

- storage management must be apparent
  - systems programmer can assist with API
- pointer constants must be well defined
  - NULL, functions, strings are ok, others use API
- object attributes must be preserved
  - cannot be manipulated by program
  - cannot be lost
  - "well behaved" programs can be checked efficiently
Check Optimization

A check at a dereference of pointer value $v$ may be elided at program pointer $p$ if the previous, equivalent check executed on $v$ has not been invalidated by some program action.

- run-time check optimization
  - more flexible, but more run-time overhead
  - eliding capability store searches: free counter
  - eliding range checks: memoization

- compile-time check optimization
  - less flexible, no run-time overhead
  - similar to common subexpression elimination
Execution Overheads

Normalized Instruction Count

<table>
<thead>
<tr>
<th></th>
<th>Anagram</th>
<th>Backprop</th>
<th>BC</th>
<th>Min-Span</th>
<th>Partition</th>
<th>YACR-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unopt Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unopt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unopt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unopt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unopt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unopt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unopt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unopt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unopt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unopt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unopt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unopt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unopt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unopt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unopt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Results

- execution overheads are low enough for development environments (130-540%), not low enough for in-field releases
- greatest slowdown factors:
  - check insertion breaks traditional optimizations
  - C++ templates are problematic
  - safe pointers are not register allocated
- run-time optimization of spatial checks is ineffective
- run-time optimization of temporal checks works well
- text and data size overheads are quite low
Comparison to Other Checking Techniques

• How are intended referents tracked?
  – “fat” pointers vs. no tracking

• How is the state of active memory represented?
  – capability store vs. memory state map

• How is the program instrumented?
  – object- vs. source-level

• What optimizations are applied?
  – spatial and temporal check optimization
Summary

- technique capable of detecting **ALL** memory access errors
- source level program transformations
- works on existing codes, e.g., those written in C/C++
- employs safe pointers $\rightarrow$ pointer value + referent details
- compile- and run-time access check optimization possible
- 540% overhead for *Partition* (3.7 insts/deref)
Current and Future Work

- re-target compiler from C++ to C
- compile-time check optimization
- run-time check improvements
- back-end integration
- parallel checking
- storage leak detection
- interface issues (libraries, system calls, unsafe code)

- availability: e-mail austin@cs.wisc.edu
char *p, *q;
p = malloc(10);
q = p+6;
*q;  /* no error */
free(p);
p = malloc(10);
*q;  /* error!!! */

[value, base, size, storageClass, capability]
## Analyzed Programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Instructions</th>
<th>Instructions per Dereference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Static</td>
<td>Dynamic</td>
<td>Static</td>
</tr>
<tr>
<td>Anagram</td>
<td>10.0K</td>
<td>19.4M</td>
<td>106.3</td>
</tr>
<tr>
<td>Backprop</td>
<td>10.8K</td>
<td>122.4M</td>
<td>148.5</td>
</tr>
<tr>
<td>GNU BC</td>
<td>19.5K</td>
<td>12.2M</td>
<td>15.5</td>
</tr>
<tr>
<td>Min-Span</td>
<td>11.9K</td>
<td>13.3M</td>
<td>48.7</td>
</tr>
<tr>
<td>Partition</td>
<td>13.5K</td>
<td>21.1M</td>
<td>62.4</td>
</tr>
<tr>
<td>YACR-2</td>
<td>18.5K</td>
<td>546.2M</td>
<td>37.1</td>
</tr>
</tbody>
</table>
Comparison to Other Checking Techniques

- How are intended referents tracked?
  - “fat” pointers (Safe-C, CodeCenter, RTCC, Bcc, UW-Pascal)
  - intended referents are not tracked (Purify)

- How is the state of active memory represented?
  - capability store (Safe-C)
  - memory state map (Purify, CodeCenter)
  - keys stored in heap node headers (UW-Pascal)
  - memory state is not tracked (RTCC, Bcc)

- How is the program instrumented?
  - object-level: (Purify)
  - source-level/compile-time: (Safe-C, RTCC, Bcc, UW-Pascal)
  - run-time: (CodeCenter)