Precise Calling Context Encoding

Nick Sumner
Yunhui Zheng
Dasarath Weeratunge
Xiangyu Zhang
What Are Calling Contexts?

• Calling Contexts
  » Sequence of active functions on call stack
  » Precisely capture sequence of active call sites
Calling Contexts

• Calling Contexts
  » Sequence of active functions on call stack
  » Precisely capture sequence of active call sites

```python
1)def a():
2)    print('Here')
3)def b():
4)    a()
5)    a()
6)def main():
7)    b()
```
Calling Contexts

- Calling Contexts
  - Sequence of active functions on call stack
  - Precisely capture sequence of active call sites

```
1) def a():
   2)   print('Here')
3) def b():
   4)   a()
   5)   a()
6) def main():
   7)   b()
```
Why Calling Contexts?

- **Context sensitive profiling**
  - Identify subtle program behaviors
Why Calling Contexts?

- Context sensitive profiling
  - Identify subtle program behaviors
- Failure location
  - For bug reports and debugging tools
Why Calling Contexts?

- Context sensitive profiling
  - Identify subtle program behaviors
- Failure location
  - For bug reports and debugging tools
- Reverse engineering input formats
  - Contexts identify substructures
Why Calling Contexts?

- Context sensitive profiling
  - Identify subtle program behaviors
- Failure location
  - For bug reports and debugging tools
- Reverse engineering input formats
  - Contexts identify substructures
- Security
  - Tracking the sources of information
Existing Approaches

- Full Contexts
  - stack walking, calling context trees, ...
Existing Approaches

- Full Contexts
  - stack walking, calling context trees, ...

- Context IDs
  - probabilistic contexts, profile inferred contexts, ...
Existing Approaches

- Problems
  - Full contexts are too expensive
Existing Approaches

- Problems
  - Full contexts are too expensive
  - IDs don't allow reverse lookup

Given an ID, to what context does it belong?
Precise Context Features

- Encode many contexts to 1 integer
  - Uses multiple integers as necessary
Precise Context Features

- Encode many contexts to 1 integer
  - Uses multiple integers as necessary

- Reversible encoding
Precise Context Features

- Encode many contexts to 1 integer
  - Uses multiple integers as necessary

- Reversible encoding

- Robust
  - Recursion, indirection, exceptions, ...
Precise Context Features

- Encode many contexts to 1 integer
  - Uses multiple integers as necessary

- Reversible encoding

- Robust
  - Recursion, indirection, exceptions, ...

- Optimized using stack sizes and profiling
  - 1.9% - 3% overhead
Precise Context Encoding

Each context is a path in the call graph
Precise Context Encoding

Each context is a path in the call graph.
Precise Context Encoding

Use unique path numbering over the call graph
Precise Context Encoding

Use unique path numbering over the call graph
Precise Context Encoding

Use unique path numbering over the call graph
Precise Context Encoding

• Encode each context in a number
  ▪ Compute the current context number online
  ▪ Similar to Ball-Larus CFG path numbering
Basic Context Encoding

- Paths start at the root
- They may end **anywhere**

```
a() ≈ main()
```
Basic Context Encoding

- Paths start at the root
- They may end anywhere
Basic Context Encoding

- Paths start at the root
- They may end anywhere

```
0
b

0
0

c

0

0 & 1
d

0 & 1
e

0 & 1
f
```
Basic Context Encoding

- Paths start at the root
- They may end **anywhere**
Basic Context Encoding

- Paths start at the root
- They may end anywhere
- We reuse the solutions for common subproblems
Basic Context Encoding

Maintain the current ID online

def c():
    ...
    contextID += 1
    d()
    contextID -= 1
    ...

0 & 1
Basic Context Encoding

- Count # of contexts per function
Basic Context Encoding

- Count # of contexts per function

for each function:
\[ \sum \text{ # contexts for each caller} \]
Basic Context Encoding

- Count # of contexts per function

IDs:

0 1
Basic Context Encoding

- Use instrumentation to partition ID space

IDs

\[ \begin{array}{c|c}
  b & c \\
  \hline
  0 & 1
\end{array} \]
Basic Context Encoding
Basic Context Encoding
Basic Context Encoding

IDs

0 1 2 3 4 5 6 7 8

a a a a
Basic Context Encoding
Basic Context Encoding

IDs

<table>
<thead>
<tr>
<th>a</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>b</th>
<th>b</th>
<th>b</th>
<th>b</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Basic Context Encoding

![Diagram showing a basic context encoding with nodes a, b, c and IDs 0 to 8. The IDs are assigned based on the context.]
## Basic Context Encoding

![Diagram of Basic Context Encoding]

### IDs

<table>
<thead>
<tr>
<th>IDs</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>b</th>
<th>b</th>
<th>b</th>
<th>b</th>
<th>c</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Basic Context Encoding

- Use instrumentation to partition ID space
- Decoding simply reverses the process
Recursion

- With recursion cycles, numbering is unbounded.
Recursion

- With recursion cycles, numbering is unbounded.
  - Transform them into acyclic graphs.
Recursion

- With recursion\cycles, numbering is unbounded.
  - Transform them into acyclic graphs.

- Each back edge has a corresponding edge in the new acyclic graph.
  - Each cyclic path becomes a list of acyclic paths
Recursion

- Push the current ID onto a context stack before recursive calls.

Instrumentation:

```python
def d():
    ...
    push(d, contextID)
    contextID = 0
    c()
    contextID = pop()
    ...
```
Recursion

• In the series of calls: a→c→d→c→d

<table>
<thead>
<tr>
<th>Last Called</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
</tbody>
</table>
Recursion

- In the series of calls:  \( a \rightarrow c \rightarrow d \rightarrow c \rightarrow d \)

<table>
<thead>
<tr>
<th>Last Called</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
</tr>
</tbody>
</table>
Recursion

- In the series of calls: a→c→d→c→d

<table>
<thead>
<tr>
<th>Last Called</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
</tr>
</tbody>
</table>
Recursion

- In the series of calls: \( a \rightarrow c \rightarrow d \rightarrow c \rightarrow d \)

<table>
<thead>
<tr>
<th>Last Called</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
</tr>
</tbody>
</table>

ID
Context Stack

head

b
+1

d
+1

c

a
Recursion

• In the series of calls: \( a \rightarrow c \rightarrow d \rightarrow c \rightarrow d \)

<table>
<thead>
<tr>
<th>Last Called</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
</tr>
</tbody>
</table>
Precise Implicit Encoding

- Some contexts can be precisely identified by stack sizes

Call Stack

Call Stack

Call a

Call b

Call c

size: 15

Call a

Call c

size: 10
Precise Implicit Encoding

- Some contexts can be precisely identified by stack sizes
  - We can use these when possible and fall back on explicit encoding when necessary.

Call Stack

<table>
<thead>
<tr>
<th>Call a</th>
<th>Call b</th>
<th>Call c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size: 15

Call Stack

<table>
<thead>
<tr>
<th>Call a</th>
<th>Call c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size: 10
Precise Implicit Encoding

- Some contexts can be precisely identified by stack sizes
  - We can use these when possible and fall back on explicit encoding when necessary.

Call Stack

<table>
<thead>
<tr>
<th>Call a</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call b</td>
<td>5</td>
</tr>
<tr>
<td>Call c</td>
<td></td>
</tr>
</tbody>
</table>

size: 15

Call Stack

<table>
<thead>
<tr>
<th>Call a</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call d</td>
<td></td>
</tr>
<tr>
<td>Call c</td>
<td>5</td>
</tr>
</tbody>
</table>

size: 15
Precise Implicit Encoding

- Some contexts can be precisely identified by stack sizes
  - We can use these when possible and fall back on explicit encoding when necessary.

- Fall back on explicit encoding for contexts w/:
  - Variable stack allocation
  - Recursive paths
  - Conflicting contexts with the same size
Evaluation

- Implemented prototype using CIL
- Examined results on SPEC 2000 and a set of real world programs
- 32-bit IDs
## Evaluation: Context Attributes

<table>
<thead>
<tr>
<th>Program</th>
<th>Max Size</th>
<th>90% Size</th>
<th># Contexts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ours</td>
<td>Full</td>
<td>Ours</td>
</tr>
<tr>
<td>164.gzip</td>
<td>1</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>175.vpr</td>
<td>1</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>176.gcc</td>
<td>20</td>
<td>136</td>
<td>3</td>
</tr>
<tr>
<td>181.mcf</td>
<td>15</td>
<td>42</td>
<td>1</td>
</tr>
<tr>
<td>186.crafty</td>
<td>35</td>
<td>41</td>
<td>11</td>
</tr>
<tr>
<td>197.parser</td>
<td>37</td>
<td>73</td>
<td>12</td>
</tr>
<tr>
<td>255.vortex</td>
<td>8</td>
<td>43</td>
<td>3</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>2</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>300.twolf</td>
<td>5</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>Program</td>
<td>Max Size</td>
<td>90% Size</td>
<td># Contexts</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>------------</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>Full</td>
<td>Ours</td>
</tr>
<tr>
<td>cmp 2.8.7</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>diff 2.8.7</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>sdiff 2.8.7</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>find 4.4.0</td>
<td>3</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>locate 4.4.0</td>
<td>1</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>grep 2.5.4</td>
<td>1</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>tar 1.16</td>
<td>4</td>
<td>40</td>
<td>3</td>
</tr>
<tr>
<td>make 3.80</td>
<td>7</td>
<td>82</td>
<td>4</td>
</tr>
<tr>
<td>alpine 2.0</td>
<td>12</td>
<td>29</td>
<td>7</td>
</tr>
<tr>
<td>vim 6.0</td>
<td>11</td>
<td>31</td>
<td>6</td>
</tr>
</tbody>
</table>
Context Stack Size Sufficiency

Make

Our Contexts

Full Contexts

Dynamic Contexts vs. # IDs
## Evaluation: Context Attributes

<table>
<thead>
<tr>
<th>Program</th>
<th>Max Size</th>
<th>90% Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ours</td>
<td>Full</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>8.7</td>
<td>39.2</td>
</tr>
</tbody>
</table>
Evaluation: Runtime

Basic Normalized Overhead
Evaluation: Runtime

Implicit Normalized Overhead

Graph showing the implicit normalized overhead for various applications and programs.
Evaluation

• Our method
  » Basic: 3.6% overhead
  » Hybrid: 1.9% overhead
  » Reversible
  » Multiple integers (1-3 in most cases)

• Compared to Probabilistic:
  » 3% overhead
  » One way
  » One integer
Related Work

**Probabilistic Calling Context**
[Bond, McKinley OOPSLA'07]

**Breadcrumbs**
[Bond, Baker, Guyer PLDI'10]

**Inferred Call Path Profiling**
[Mytkowicz, Coughlin, Diwan OOPSLA'09]

**Efficient Path Profiling**
[Ball, Larus MICRO'96]
## Conclusions

<table>
<thead>
<tr>
<th>Partial Context Info</th>
<th>Lower Overhead</th>
<th>Higher Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PCC (Hashing)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Breadcrumbs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inferred Call Paths</td>
<td></td>
</tr>
<tr>
<td>Full Context Info</td>
<td></td>
<td>Stack Walking Calling Context Trees</td>
</tr>
</tbody>
</table>
## Conclusions

<table>
<thead>
<tr>
<th>Partial Context Info</th>
<th>Lower Overhead</th>
<th>Higher Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PCC (Hashing)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Breadcrumbs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inferred Call Paths</td>
<td></td>
</tr>
<tr>
<td>Full Context Info</td>
<td>Precise Calling Context Encoding</td>
<td>Stack Walking Calling Context Trees</td>
</tr>
</tbody>
</table>
Thank You