DELTAPATH: PRECISE AND SCALABLE CALLING CONTEXT ENCODING

Qiang Zeng*, Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, Peng Liu*

NEC Laboratories America
*Penn State University

www.nec-labs.com
Calling Context

- Calling Context is a sequence of active function/method invocations that lead to a program location (i.e., call stack status).

- Wide range of applications
  - Debugging, event logging, error reporting, testing, anomaly detection, performance optimization, profiling, security.
How to Collect Calling Contexts?

- Stack Walking
- Probabilistic Calling Context [OOPSLA’07]
- Precise Calling Context Encoding [ICSE’10]
Stack Walking

• Walk stack and collect context
  • Stack walking collects a set of return addresses from the stack.
  • Commonly used in debuggers (e.g., gdb) and error reporting

```
1 A() {
2     B();
3 }
4 B() {
5     C();
6     D();
7 }
8 C() {
9     D();
10 }
11 D() {
12 // Context?
13 }
```

Advantage: simple
Disadvantage: performance overhead
Probabilistic Calling Context

- Compute probabilistic calling context at runtime

\[
f (V, cs) := 3 \times V + cs
\]

```
A() {
  B(); (cs1)
}

B() {
  C(); (cs2)
  D();
}

C() {
  D(); (cs3)
}

D() {
  // Context?
}
```

Advantage: simple & fast encoding scheme
Disadvantage: decoding is not guaranteed.
Precise Calling Context Encoding

• Use unique numbering to represent a path in a CFG

```c
1 A() {
2   B(); (cs1)
3 }
4 B() {
5   C(); (cs2)
6   D();
7 }
8 C() {
9   D(); (cs3)
10 }
11 D() {
12   // Context?
13 }
```

Advantage: Precise call context encoding and decoding
class Shape { void draw() {};
}
class Rectangle extends Shape {
  void draw() {}
}
class Triangle extends Shape {
  void draw() {}
}
class D {
  static void main() {
    Shape a;
    if (input) a = new Rectangle();
    else a = new Triangle();
    a.draw();
  }
}
Precise Calling Context Encoding

• PCCE maps each unique context into an integer.
• The integer space is insufficient for large programs.
  • Object oriented programs tend to have many small functions leading to a large context space.

Disadvantage 2: PCCE addresses this problem using profiling and identifying hot and cold edges.
DeltaPath Features

• New precise and scalable calling context encoding algorithm for both procedural and object oriented programs
  • Overcome dynamic dispatch
  • Address encoding space pressure systematically

• Practical Issues
  • Dynamic class loading is handled.
  • Flexible encoding scope
Technique – Inflated Calling Context

• Basic properties of Precise Calling Context Encoding
  • Ensure the invariant that for a given node, its encoding space is divided into disjoint sub-ranges for unique numbering.
  • AV : addition value, CC : calling context count

$$AV[P_1] = 0$$
$$AV[P_2] = CC[P_1]+AV[P_1]$$
$$AV[P_m] = CC[P_{m-1}]+AV[P_{m-1}]$$

Context Encoding ID  0  1  2  3  4  5  6  7  8  9


Invariants:
$$AV[P_i] = CC[P_{i-1}]+AV[P_{i-1}]$$
for $$i = 2, ..., m$$
$$CC[n] >= CC[P_m] + AV[P_m]$$

Encoding ID space is partitioned using AV and CC
Technique – Inflated Calling Context

• **Idea: Inflated Calling Context**
  • While PCCE processes the nodes one by one, DeltaPath needs to take into account the current addition value for another node so that all nodes involved in dynamic dispatch can agree on the common addition value. This is achieved by the *inflation of calling context*.

1. AV for a call from D.main to Rectangle.draw
2. \( A = \text{Max}(AV[\text{Rectangle.draw()}], \text{AV}[\text{Triangle.draw()}]) \)
3. \( \text{AV[Rectangle.draw()]} \) and \( \text{AV[Triangle.draw()]} \) are inflated as \( CC[D\.main()] + A \).
Technique – Resolving Context Explosion

• Encoding for large-scale object-oriented programs
  • Systematically divides the CFG into **territories** whose contexts fit the limit of integer space.
  • On the detection of overflow, the node is added into the set of **anchor** nodes and static analysis is restarted (iterative approach).
  • At runtime an anchor flushes current context onto stack and the context variable is reset.

Challenge: The problem would be simple if each territory has only one entry point but in fact there are **cross-territory calls**.
Technique – Resolving Context Explosion

• Multiplexing the contexts of multiple territories
  • The common addition value is used for all multiplexed territories. Thus the context variable should afford the context of all multiplexed territories.
  • Use **two dimensional states** in the algorithm to track contexts from multiple overlapped territories.
  • Use **inflation** to meet the invariants for multiple territories simultaneously.

\[
\text{ICC[node][anchor]} = \text{inflated calling context count and addition value at the node relative to the anchor}
\]

\[
A = \text{Max}(\text{CAV[E][D]}, \text{CAV[F][D]}, \text{CAV[F][C]})
\]
Practical Issues

- **Dynamic Class Loading**
  - Java loads and combines code at runtime. Such code cannot be pre-analyzed causing **unexpected call paths (UCPs)**.

- **Solution: Calling Context Tracking**
  - We adopted control flow integrity (CFI) technique to detect UCPs.
  - For each call site, finds out the dispatch target nodes. Merge the sets that contain any overlap and assign unique set identifiers (SID).
  - Expected SIDs are stored at callers and checked at callees.
Practical Issues

- Do we need to track all code?
  - Java has large library code base which may be little of interest for debugging etc.
  - PCC redefines application only calling context and encodes it.
  - Also including all code inevitably will slow down execution.

- Solution: Flexible Encoding
  - Leveraging call path tracking we can skip encoding components of little interest the same way we handle dynamically loaded classes.
  - Call paths through skipped nodes are detected as UCPs.

Application code fully covered

No overhead in numerous libraries
Implementation and Evaluation

• Static Analysis
  • WALA (T.J. Watson Libraries for Analysis)
  • Analysis: Context Insensitive Control Flow Analysis (0-CFA)
  • Input: Binary only, No source code

• Runtime Module and Dynamic Instrumentation
  • A Java agent based on Javassist
  • Support Sun JVM (Version >= JDK 5.0)

• Evaluation
  • SPECjvm2008 Benchmark Suite
  • Intel Core i7 CPU, 8GB RAM
  • Ubuntu Linux 10.04
  • Sun JDK 1.6.0.24
Evaluation

- Static Program Characteristics

<table>
<thead>
<tr>
<th>program</th>
<th>size (bytes)</th>
<th>encoding-all</th>
<th>max. ID</th>
<th>encoding application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nodes</td>
<td>edges</td>
<td>CS</td>
<td>VCS</td>
</tr>
<tr>
<td>compiler.compiler</td>
<td>114K</td>
<td>2308</td>
<td>7329</td>
<td>7003</td>
</tr>
<tr>
<td>compiler.sunflow</td>
<td>85K</td>
<td>1846</td>
<td>4185</td>
<td>5511</td>
</tr>
<tr>
<td>compress</td>
<td>59K</td>
<td>1298</td>
<td>2675</td>
<td>3391</td>
</tr>
<tr>
<td>crypto.aes</td>
<td>133K</td>
<td>2656</td>
<td>8201</td>
<td>8369</td>
</tr>
<tr>
<td>crypto.rsa</td>
<td>133K</td>
<td>2656</td>
<td>8204</td>
<td>8386</td>
</tr>
<tr>
<td>crypto.signverify</td>
<td>135K</td>
<td>2694</td>
<td>8290</td>
<td>8548</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>261K</td>
<td>3132</td>
<td>9734</td>
<td>9579</td>
</tr>
<tr>
<td>scimark.fft.large</td>
<td>57K</td>
<td>1279</td>
<td>2636</td>
<td>3321</td>
</tr>
<tr>
<td>scimark.lu.large</td>
<td>57K</td>
<td>1273</td>
<td>2616</td>
<td>3304</td>
</tr>
<tr>
<td>scimark.monte_carlo</td>
<td>56K</td>
<td>1260</td>
<td>2590</td>
<td>3262</td>
</tr>
<tr>
<td>scimark.sor.large</td>
<td>57K</td>
<td>1269</td>
<td>2614</td>
<td>3303</td>
</tr>
<tr>
<td>scimark.sparse.large</td>
<td>57K</td>
<td>1265</td>
<td>2614</td>
<td>3291</td>
</tr>
<tr>
<td>sunflow</td>
<td>458K</td>
<td>7727</td>
<td>25485</td>
<td>27135</td>
</tr>
<tr>
<td>xml.transform</td>
<td>752K</td>
<td>9766</td>
<td>38010</td>
<td>44266</td>
</tr>
<tr>
<td>xml.validation</td>
<td>478K</td>
<td>6703</td>
<td>23092</td>
<td>28333</td>
</tr>
</tbody>
</table>

- Encoding all setting
  - 13 out of 15 need encoding space larger than a million
  - Two benchmarks have overflow of the 64bit integer (1.8 X 10^19).
  - Overflow is resolved by introducing 6~7 anchor nodes.
Evaluation

• Performance Comparison

![Graph showing performance comparison between Native, DeltaPath wo/CPT, PCC, and DeltaPath w/CPT across various benchmarks.](graph.png)

• DeltaPath without Call Path Tracking: 32.51% (geometric mean)
• Call Path Tracking adds extra 6.79% slow down.
• Comparable with PCC (0.5% slower)
Evaluation

- Dynamic Program Characteristics (Application only)

<table>
<thead>
<tr>
<th>program</th>
<th>collected calling contexts</th>
<th>PCC</th>
<th>DeltaPath</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total contexts</td>
<td>max. depth</td>
<td>avg. depth</td>
</tr>
<tr>
<td>compiler.compiler</td>
<td>92634</td>
<td>15</td>
<td>5.1</td>
</tr>
<tr>
<td>compiler.sunflow</td>
<td>63705</td>
<td>12</td>
<td>5.4</td>
</tr>
<tr>
<td>compress</td>
<td>3243640985</td>
<td>12</td>
<td>10.0</td>
</tr>
<tr>
<td>crypto.aes</td>
<td>14431</td>
<td>9</td>
<td>5.6</td>
</tr>
<tr>
<td>crypto.rsa</td>
<td>538625</td>
<td>9</td>
<td>6.0</td>
</tr>
<tr>
<td>crypto.signverify</td>
<td>541682</td>
<td>9</td>
<td>6.0</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>2489700943</td>
<td>17</td>
<td>13.4</td>
</tr>
<tr>
<td>scimark.fft.large</td>
<td>566237360</td>
<td>12</td>
<td>10.0</td>
</tr>
<tr>
<td>scimark.lu.large</td>
<td>188838329</td>
<td>10</td>
<td>10.0</td>
</tr>
<tr>
<td>scimark.monte_carlo</td>
<td>5033167760</td>
<td>11</td>
<td>10.0</td>
</tr>
<tr>
<td>scimark.sor.large</td>
<td>293603875</td>
<td>10</td>
<td>10.0</td>
</tr>
<tr>
<td>scimark.sparse.large</td>
<td>252002429</td>
<td>11</td>
<td>10.0</td>
</tr>
<tr>
<td>sunflow</td>
<td>2840077292</td>
<td>39</td>
<td>21.8</td>
</tr>
<tr>
<td>xml.transform</td>
<td>92333406</td>
<td>55</td>
<td>15.5</td>
</tr>
<tr>
<td>xml.validation</td>
<td>12900727</td>
<td>11</td>
<td>9.0</td>
</tr>
</tbody>
</table>

- Average stack depth is 1~4.4 (5.1~21.8 call stack depth)
- PCC collects less unique contexts due to hash collision.
- DeltaPath offers precise decoding compared to PCC.
Conclusion

- DeltaPath provides precise and scalable calling context encoding for procedural and object-oriented programs.
- DeltaPath provides high efficiency similar to PCC with the advantage of precise encoding and decoding.
- DeltaPath deals with dynamic class loading and supports selective encoding.

<table>
<thead>
<tr>
<th>Feature</th>
<th>PCC</th>
<th>PCCE</th>
<th>DeltaPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support both procedural and OO</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Reliable decoding</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Scalability</td>
<td>Y*</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

PCC: Probabilistic calling context, PCCE: Precise Calling Context Encoding

* Hash collision may become a problem in very large-scale software.
Thank you