Breadcrumbs: Efficient Context Sensitivity for Dynamic Bug Detection Analyses

Bond, Baker and Guyer

Basic Contribution

- Decoding a PCC value
  - Human readable sequence of calls
- Evaluation
  - Dynamic Race Detector
  - Origin Tracking – Null Pointer Exception Diagnosis

Basic PCC

- PCC

$$p' = f(p, c) = (3p + c)\, mod\ 2^{32}$$

» $p_o = 0$ [main]

» $p_1 = f(p_o, co)$

» $p_2 = f(p_1, c_1)$

» $p_3 = f(p_2, c_2)$

.

.

» $p_i = f(pi_{-1}, ci_{-1})$

.

.

» $p_n = f(pn_{-1}, cn_{-1})$ [return main]

Decoding PCC…

- Meaning

» $p_o = 0$ [main]

» $p_1 = f(p_{o,} c_o)$

» $p_2 = f(p_{1,} c_1)$

» $p_3 = f(p_{2,} c_2)$

.

.

» $p_i = f(pi_{-1,} c_{i-1})$

.

.

» $\boldsymbol{p_n} = f(pn_{-1,} \boldsymbol{c_{n-1}})$ [return main]

Inverse, $f^{-1}()$

- Given p’ in
  $p' = f(p, c) = (3p + c)\, mod\ 2^{32}$
  - Find *p and* ***c***
  - *for a given c and p’ … p is unique.*
  - err… we want to find ***c***
  - but**,** in order to **track back,** ***p*** is required

Inverse, *f* $^{-1}$*()*

- Given p’ in $p' = f(p, c) = (3p + c)\, mod\, 2^{32}$
  - Find *p and* ***c***
  - *Choose a* ***c****, then* $p = f^{-1}(p', c)$
    - » p_n
    - » $p_{n-1} = f^{-1}(p_n, \boldsymbol{c_{n-1}})$
    - » $p_{n-2} = f^{-1}(pn_{-1,}\, c_{n-2})$

      .

      .
    - » $p_i \;_{=}\; f^{-1}(pi_{+1,}\, c_i)$

      .

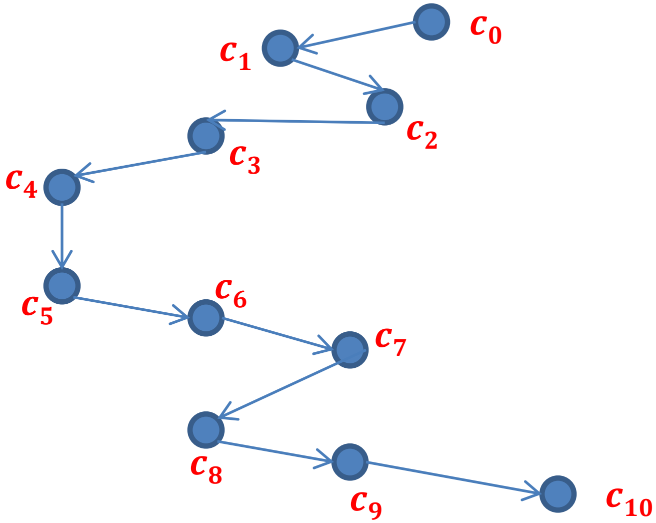
      .
    - » $p_0 = f^{-1}(p_{1,}\, c_0)$

[return main]

[main]

Challenges?

- Difficult search problem
  - Many Call sites **c** to choose from (1000s)
    - $p_{n-1} = f^{-1}(p_n, \boldsymbol{c_{n-1}})$
      - Accurately choosing the right $\boldsymbol{c_{n-1}}$ will be difficult.
      - Compounds the problem of deriving the right sequence.

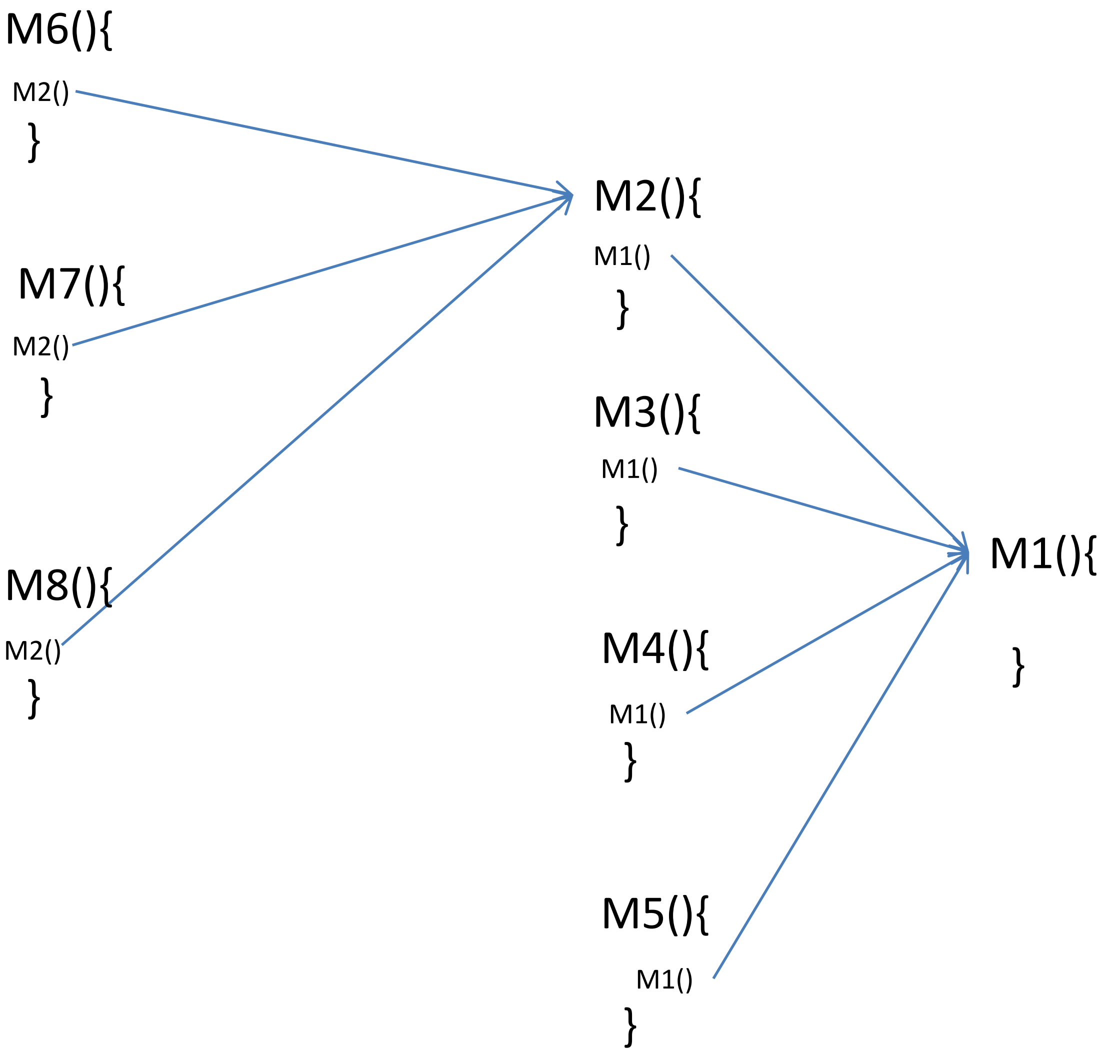


$1000^{10} = 10^{30}$ possible calling sequences. (this is minimum)

Reducing the Search Space

- == reducing the probable Call sites
- Static
- Dynamic

Static



M6(){
M2()
}

M7(){
M2()
}

M8(){
M2()
}

M2(){
M1()
}

M3(){
M1()
}

M4(){
M1()
}

M5(){
M1()
}

M1(){

}

Issues with Static

```
class A {
static { methodA(); }
}

public methodA(){
System.out.println(“helloworld”);
}

public static void main(String[] args ) {
A objecta = new A();
}
```

JVM

```
registerKeyPressEvent(e);

void HandleKeyPressEvent(e, arg)
{
  Display(“hello!”);
}

void Display()
{


}
```

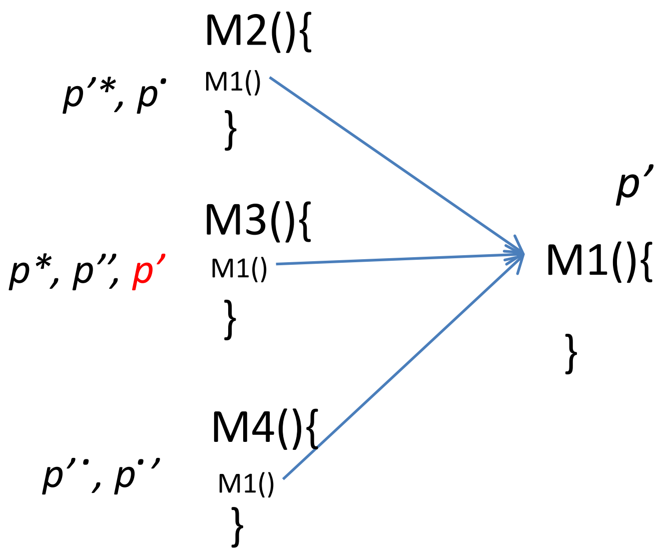
JAVA/SWING

The possible call sites are incomplete.

Dynamic Analysis is then used to find the missing links.

Dynamic

- Calculate **and store** all PCC values at specific call sites.
- $p = f^{-1}(p', c)$



M2(){
M1()
}
p′*, p˙

M3(){
M1()
}
p*, p′′, p′

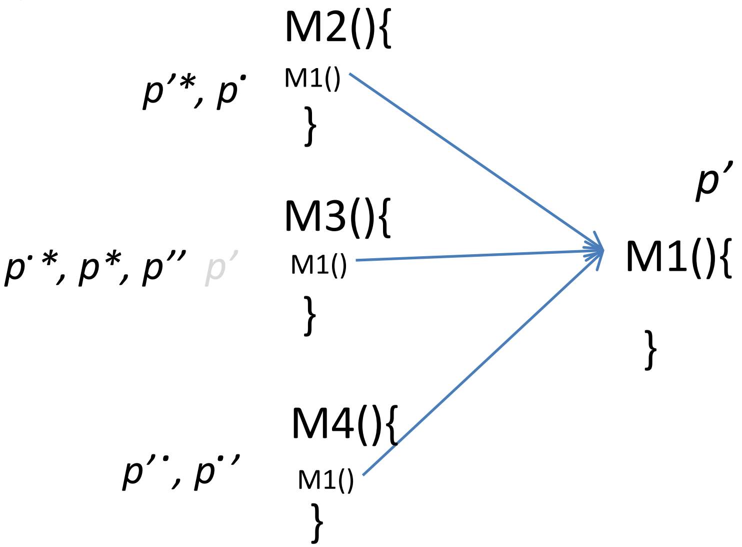
M4(){
M1()
}
p′˙, p˙′

M1(){
}
p′

a. 3 out of 1000 call sites (static)
b. Find all Per call site PCC values. See where *p′* is. (dynamic)

Issues with Dynamic

- As always, too expensive.
- Solution-
  - *hotThreshold*
    - Stop recording the PCC values after the threshold.
- Issues with Solution
  - You can’t guess accurately anymore.
- As always, the Accuracy –Performance Tradeoff



M2(){
$p'*, p^{\cdot}$
M1()
}

M3(){
$p^{\cdot}*, p*, p''$ p'
M1()
}

M4(){
$p'^{\cdot}, p^{\cdot}{}'$
M1()
}

p'
M1(){
}

PCC Values are Client sites - Extensibility

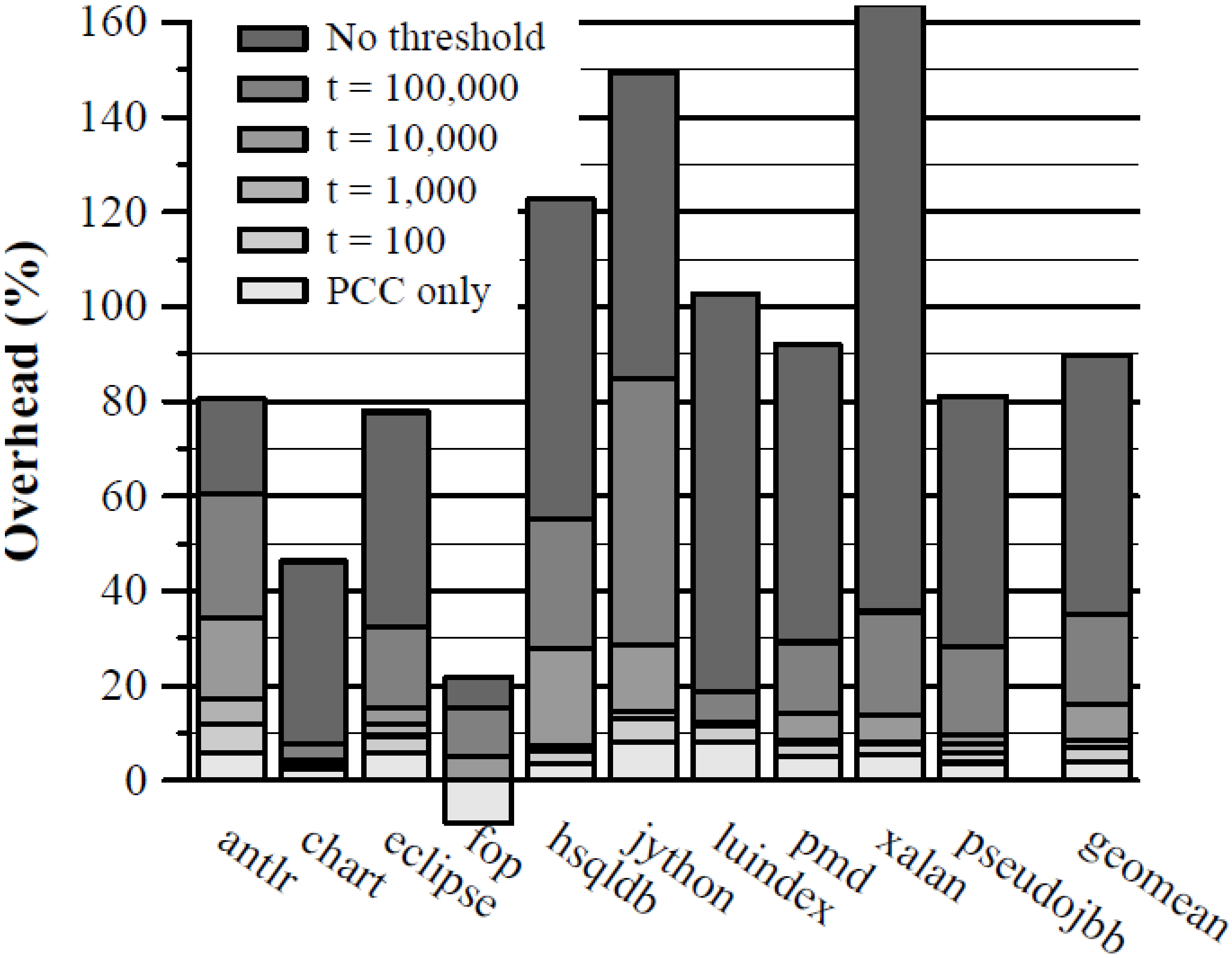
- The PCC values are generally calculated at callsites.
- Thus, you can’t look at the program flow at all points.
- So, you start storing the information at the client sites (sites which are of interest to the client, like suspicious bug locations, or memory operations).

Evaluations

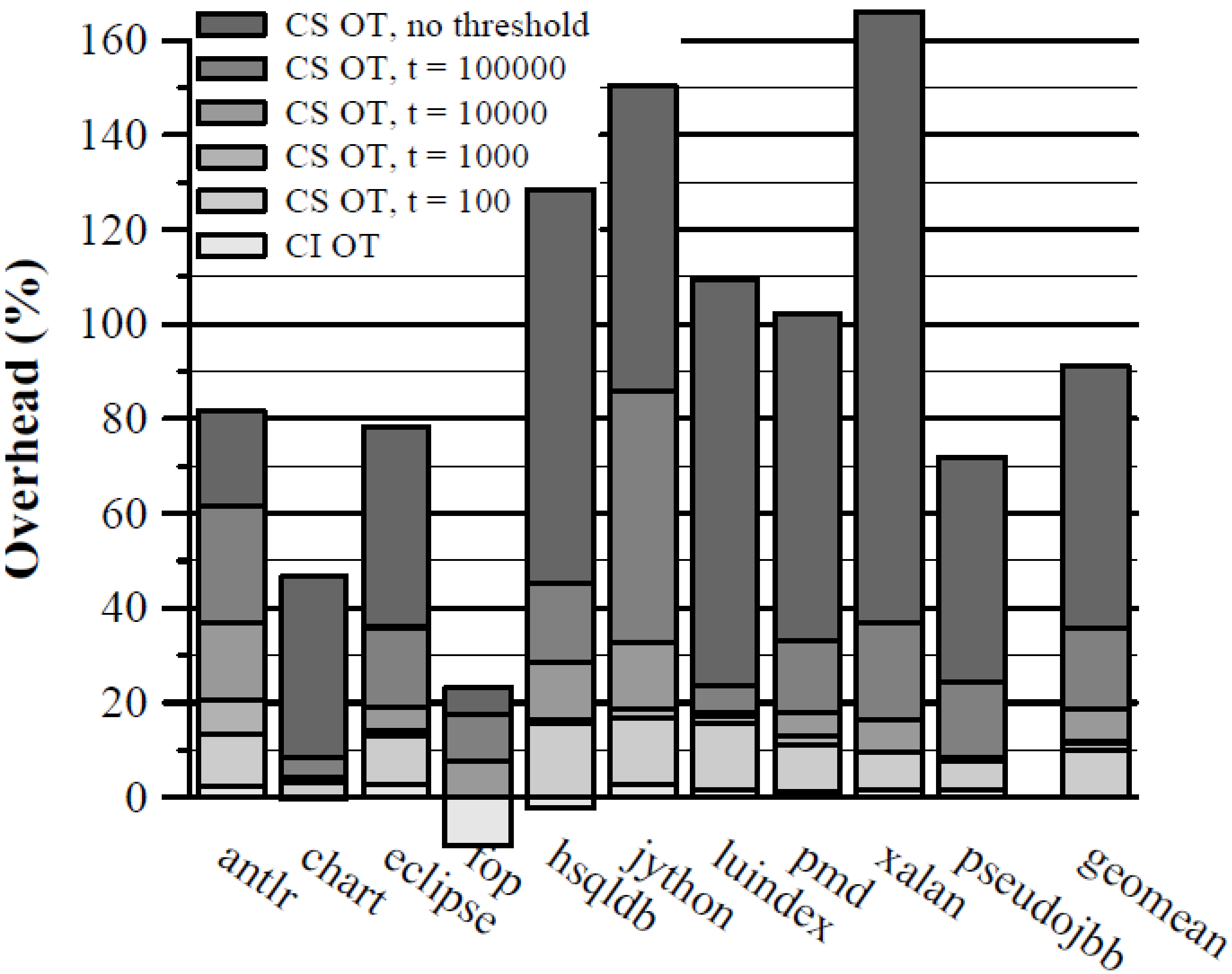
- No client
  - PCC only
  - T= 100; 1,000; 10,000; 100,000; inf.
- Origin Tracking
  - OT only
  - T= 100; 1,000; 10,000; 100,000; inf.
- Race Detection
  - RD only
  - T= 100; 1,000; 10,000; 100,000; inf.

contd.

- PCC only – No Client
  - “No threshold” adds as high as 90% overhead.
  - T = 100 to 1000, adds about 10 to 20%. Still too high for production.
- Origin Tracking
  - Direct application of PCC.
  - Propagation of null values.
  - The overheads are very similar to PCC only.



Overhead (%)
160
140
120
100
80
60
40
20
0
No threshold
t = 100,000
t = 10,000
t = 1,000
t = 100
PCC only
antlr
chart
eclipse
fop
hsqldb
jython
luindex
pmd
xalan
pseudojbb
geomean


CS OT, no threshold
CS OT, t = 100000
CS OT, t = 10000
CS OT, t = 1000
CS OT, t = 100
CI OT
Overhead (%)
160
140
120
100
80
60
40
20
0
antlr
chart
eclipse
fop
hsqldb
jython
luindex
pmd
xalan
pseudojbb
geomean

contd.

- Race Detector - Pacer
  - FastTrack Algorithm
    - Significant Runtime and Space
  - Calling contexts of all memory operations
  - Overhead of PCC Decoding is very small compared to the overhead of Pacer.

Take Away

- Add-on to the original PCC work
- Significant runtime overhead
  - 10 to 20 % at the minimum. (if you want accurate reconstruction of graphs.)
- Reconstruction not easy even at t = 10,000 sometimes.

Observations

- **Space overhead was not talked about.**
- Did not specify what call-depth is practically useful
  - Do you need 10+ levels of depth to debug?
  - Would give a more practical picture.
- Can we use an arithmetic encoding function instead, like in compression techniques?