Towards a Better Collaboration of Static and Dynamic Analyses for Testing Concurrent Programs

Jun Chen and Steve MacDonald

David R. Cheriton School of Computer Science
University of Waterloo
Introduction

- Finding concurrent bugs statically
  - Alias analysis, escape analysis, may happen in parallel analysis
    - Chord - false positives

- Finding concurrent bugs dynamically
  - Lockset algorithm
    - Eraser – false negatives/positives
  - Instrumenting
    - Contest – false negatives
  - Explicit State Model Checking
    - JPF – state space explosion
Static + Dynamic Analysis

- Static analysis prunes out irrelevant program artefacts. Dynamic analysis tests rest of problem space.
  - Advantage: Reduces the problem space of dynamic analysis.
  - Static analysis tends to be conservative. The remaining problem space might still remain large.

- What if static analysis is used to actively guide the dynamic analysis to test potential bugs?
Rationale

- Explore interleaving (state space) DFS/BFS. You may run out of memory or CPU time before hitting a bug.

- Instead of searching exhaustively, the computational resources should always be dedicated to actively finding a possible bug.
Finding a Concurrent Bug

- What you could learn from static analysis:
  - Point of interests:
    - MHP Reads/Writes (possible data races)
    - Competing monitor entries (possible general races)
- What you could not know from static analysis:
  - Accessibility of the read/write assignments.
    - A feasible path could enable concurrent accesses
  - Precise alias relationship among reads/writes & monitor entries
    - If such feasible path exists, are the read/writes or monitor entries still relevant?
Finding a Concurrent Bug

Strategy:
- Scan the program using static analysis
  - Identify a potential pair of MHP read/write.
  - Compute a path to enable them.
  - Drive the program execution along the path.
- Adjust/append the path along the way.
  - Which branch or polymorphism call edge is taken at runtime.
  - After concurrent accesses have been reached, are they relevant (true alias)? If not, pick another possible pair to test.
Finding a Concurrent Bug

- **Advantage**
  - Avoids exhaustive search (either DFS or BFS)

- **Supporting mechanism**
  - Tight collaboration between static and dynamic analysis
    - Iterative information exchange between two components.
An Example

Thread t1:
Manager (Account P1, Account B1)
  P1.deposit(20);
  P1.transfer(20, B1);

Thread t2:
Manager (Account B1, Account B2)
  B1.deposit(20);
  B1.transfer(20, B2);

Thread t3:
Manager (Account B2, Account P1)
  B2.deposit(20);
  B2.transfer(20, P1);

3 accounts:
  2 BusinessAccount B1, B2
  1 PersonalAccount P1
An Example

<table>
<thead>
<tr>
<th>Personal Account</th>
<th>Business Account</th>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync transfer(amt, account) { bal -= amt; //unprotected account.bal += amt; }</td>
<td>sync transfer(amt, account) { bal -= amt; sync(account) { account.bal += amt; } }</td>
<td>sync deposit(amt) { bal += amt; }</td>
</tr>
</tbody>
</table>

Data Race:
A unprotected access of account’s bal field from the transfer() method of the Personal Account, while account is depositing.
Static Analysis

Static analysis:

- MHP read/writes
  - PersonalAccount::transfer(): account.bal += amt;
  - AND
    - Account::deposit(): bal += amt;

- This pair might happen between any two threads due to imprecise alias and call graph analysis.
  - Alias analysis will resolve every reference of account in each thread to all three instances of account created in main().
  - Call graph analysis will report every account.transfer() might produce two call edges for two different subtype of Account.
Using Dynamic Information

- Analysis of: `PersonalAccount::transfer()`: `account.bal +=amt;`  
  AND  
  `Account::deposit(): bal +=amt;`

<table>
<thead>
<tr>
<th></th>
<th>T1(P.deposit)</th>
<th>T2(B.deposit)</th>
<th>T3(B.deposit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1(P.transfer)</td>
<td></td>
<td>True Alias</td>
<td>False Alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td>True CFG</td>
<td>True CFG</td>
</tr>
<tr>
<td>T2(B.transfer)</td>
<td>False CFG</td>
<td></td>
<td>False CFG</td>
</tr>
<tr>
<td>T3(B.transfer)</td>
<td>False CFG</td>
<td>False CFG</td>
<td>False CFG</td>
</tr>
</tbody>
</table>
Static Analysis

Instead of having dynamic analysis run randomly, we statically compute a path to test a read/write pair.

Let us assume, we pick pair

\[ T1: \text{PersonalAccount::transfer(): account.bal} += \text{amt}; \]

\[ T2: \text{Account::deposit(): bal} += \text{amt}; \]

The path will enable \( \text{w(account.bal)} \) and \( \text{r(bal)} \) to become next-to-execute instruction in \( t1 \) and \( t2 \) respectively.
Static Analysis + Dynamic Method Resolving

T1:
```
P1.deposit(20);
P1.transfer(20, B1);
```

deposit(amt) {
    bal += amt;
}

transfer(amt, account)

Execute program to here, and look into the heap to determine the runtime type of the callee.

PersonalAccount::transfer()  BusinessAccount::transfer()
Static Analysis + Dynamic Alias Checking

T1:

```
P1.deposit(20);
P1.transfer(20, B1);
```

```
deposit(amt) {
    bal += amt;
}
```

```
transfer(amt, account) {
    bal -= amt;
    //unprotected
    tmp = r(account.bal);
    tmp += amt;
    w(account.bal);
}
```

T2:

```
B1.deposit(20);
B1.transfer(20, B2);
```

```
deposit(amt) {
    tmp = r(bal)
    tmp += amt;
    w(bal);
}
```

Execute threads up to blue statements, and get the runtime memory address of bal.

In this case, aliased.
Static Analysis + Dynamic Alias Checking

T1:
```
deposit(amt) {
    bal += amt;
}
```
```
transfer(amt, account) {
    bal -= amt;
    //unprotected
    tmp = r(account.bal);
    tmp += amt;
    w(account.bal);
}
```

T3:
```
B1.deposit(20);  
P1.transfer(20, B1);
```
```
B2.deposit(20);  
P2.transfer(20, P1);
```
```
T1:
```
B2.deposit(20);  
P2.transfer(20, P1);
```
```
execute threads up to blue statements, and get the runtime memory address of bal.
In this case, not aliased.
Implementation

Static Modules
- Soot
- MHP
- CFG
- Value Schedule Generator

Dynamic Modules
- Model Checker JPF

Runtime control flow edge
Fulfilling paths
Runtime feasibility
Runtime alias relationships
A Simple Experiment

- Test Case:
  - AccountSubtype (4 Business, 1 Personal)

- Environment:
  - Intel P4 2.0; Memory 1.5GB

<table>
<thead>
<tr>
<th></th>
<th>Overall states</th>
<th>Dynamic method resolving states</th>
<th>Permutation</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPF 4.1</td>
<td>70279</td>
<td></td>
<td></td>
<td>1,783,821</td>
</tr>
<tr>
<td>Alias, cfg</td>
<td>193</td>
<td>186</td>
<td>7</td>
<td>34,056</td>
</tr>
<tr>
<td>no-alias, cfg</td>
<td>240</td>
<td>186</td>
<td>54</td>
<td>70,316</td>
</tr>
<tr>
<td>no-cfg, alias</td>
<td>7</td>
<td></td>
<td>7</td>
<td>58,992</td>
</tr>
<tr>
<td>no-alias, no-cfg</td>
<td>95</td>
<td></td>
<td>95</td>
<td>104,613</td>
</tr>
</tbody>
</table>
Conclusions

We’ve presented a collaboration scheme between static and dynamic analysis
- Static analysis guides dynamic analysis
- Correct static analysis using dynamic analysis
- Iterative scheme

Preliminary results show some improvement over existing JPF
Questions?