Heap Cloning: Enabling Dynamic Symbolic Execution of Java Programs

Saswat Anand
Mary Jean Harrold

Supported by NSF (CCF-0725202, CCF-0541048) and IBM (Software Quality Innovation Award)

Symbolic Execution

Estimated number of publications related to symbolic execution. The list of papers available at: http://sites.google.com/site/symexbib
Symbolic Execution

Goal of this work is to enable correct symbolic execution of real-world Java programs with minimal manual effort.

Estimated number of publications related to symbolic execution. The list of papers available at: http://sites.google.com/site/symexbib

Outline

- Problem
- Heap Cloning
- Empirical evaluation
- Conclusion
Symbolic Execution of Real-World Programs

P’ not symbolically executed because:
1. P’ uses difficult-to-handle Java features such as native methods.
2. Symbolic execution of P’ is mostly unnecessary.

Outline

• Problem
• Heap Cloning
• Empirical evaluation
• Conclusion
Symbolic Execution with User-specified Models

Replace $P'$ with user-specified models $P_M$

Impractical! Requires significant manual effort.

Dynamic Symbolic Execution

$P$: to be symbolically executed
$P'$: not to be symbolically executed

No manual effort to write models
**Example**

```java
class A {
    int f;
    A(int x) {
        this.f = x;
    }
    native void negate();
    void negate() {
        this.f = -this.f;
    }
    void main() {
        x = read();
        m = new A(x);
        if(m.f < 0)
            m.negate();
        if(m.f < 0)
            error();
        exit();
    }
}
```

**Task:**
Perform dynamic symbolic execution along the path (1-2-3-4-5-7) that program takes for concrete input -10.

**Requirement:**
Native method `negate()` not to be symbolically executed
Dynamic Symbolic Execution

Path Constraint (PC) of a path is a constraint of input values.

If PC is unsatisfiable, the path is infeasible.

If PC is satisfiable, any solution of PC is a program input that takes the corresponding program path.
Dynamic Symbolic Execution

1. \( x = \text{read}() \);
2. \( m = \text{new } A(x) \);
3. if(m.f < 0)
4. \( m.\text{negate}() \);
5. if(m.f < 0)
6. error();
7. exit();

Continued on next slide

Differences between two successive states are shown in Red.

---

Dynamic Symbolic Execution

1. \( x = \text{read}() \);
2. \( m = \text{new } A(x) \);
3. if(m.f < 0)
4. \( m.\text{negate}() \);
5. if(m.f < 0)
6. error();
7. exit();

Continued on next slide

Differences between two successive states are shown in Red.
Dynamic Symbolic Execution

Continued from prev. slide

Without model of negate native method

1 x = read();
2 m = new A(x);
3 if(m.f < 0)
4   m.negate();
5 if(m.f < 0)
6   error();
7   exit();

Differences between two successive states are shown in Red.

Dynamic Symbolic Execution

Continued from prev. slide

Without model of negate native method

1 x = read();
2 m = new A(x);
3 if(m.f < 0)
4   m.negate();
5 if(m.f < 0)
6   error();
7   exit();

Differences between two successive states are shown in Red.
Dynamic Symbolic Execution

Concrete value of the field changed from -10 to 10. But, symbolic value $X_0$ did not change.

Differences between two successive states are shown in Red.

Dynamic Symbolic Execution

Solve UNSAT

Differences between two successive states are shown in Red.
Dynamic Symbolic Execution

1. \( x = \text{read}(); \)
2. \( m = \text{new} \ A(x); \)
3. if(m.f < 0)
4. \( m.\text{negate}(); \)
5. if(m.f < 0)
6. \( \text{error}(); \)
7. \( \text{exit}(); \)

Differences between two successive states are shown in Red.

Dynamic Symbolic Execution with Models

void negate()
{ \[ \text{this.f = -this.f;} \]
}

User-specified model of negate

Differences between two successive states are shown in Red.
Dynamic Symbolic Execution with Models

1. x = read();
2. m = new A(x);
3. if(m.f < 0)
4. m.negate();
5. if(m.f < 0)
6. error();
7. exit();

Differences between two successive states are shown in Red.

void negate() {
    this.f = -this.f;
}

User-specified model of negate

Differences between two successive states are shown in Red.
But, Models Not Always Needed for Soundness!

void negate() {
    this.f = -this.f;
}

Original example

1   x = read();
2   m = new A(x);
3   if(m.f < 0)
4      m.negate();
5   if(m.f < 0)
6      error();
7   exit();

Modified example

1   x = read();
2   m = new A(x);
3   if(x < 0)
4      error();
5   exit();

Stmt at line 5 uses the value defined inside negate method.
Thus, negate introduces imprecision.
Thus, model for negate is needed.

Stmt at line 5 does not use the value defined inside negate method.
Thus, negate does not introduce imprecision.
Thus, no need for model for negate.
Goal and Approach

Goal:
Identify where imprecision is introduced and report to user. User then provides models for only a subset methods in $P'$ to eliminate imprecision.

Approach:
1. Transform the program using Heap Cloning.
2. Perform dynamic symbolic execution of the transformed program without any models
3. Detect and report where imprecision might be introduced.

Outline

• Problem
• Heap Cloning
• Empirical evaluation
• Conclusion
Heap Cloning

1. Symbolic execution of $P_T$ generates same results as symbolic execution of $P$.
2. However, if imprecision is introduced during symbolic execution of $P_T$, it can be detected and reported to user.

Heap Cloning

For every class $A$, Heap Cloning generates $HC.A$.

```java
class HC.java.lang.Object {
    ...
}
class HC.A extends HC.java.lang.Object {
    ...
}
```
Heap Cloning

For every class A, Heap Cloning generates HC.A.

For each field (method) of A, HC.A has a corresponding field (method).

class HC.java.lang.Object {
   ...
}
class HC.A extends HC.java.lang.Object {
   int f;
   ...
   void main() {
      ...
   }
}

Heap Cloning

Effects of the statement $m = \text{new } A$ on program state

Effect in the original program P

Effect in the transformed program $P_T$

For every object of class A in P, $P_T$ allocates two objects: one of class A, other of class HC.A
Heap Cloning

Effects of the statement \( m.f = -10 \) in \( P_T \)

Every heap location is cloned. Values of both locations remain “consistent” if \( P' \) does not modify value of shadow location.
Heap Cloning

Effects of the statement $m.f = -10$ in $P_T$

Every heap location is cloned. Values of both locations remain “consistent” if $P'$ does not modify value of shadow location.
### Heap Cloning

**Effects of the statement**

\( m.f = -10 \) in \( P_T \)

**Effects of the statement**

\( m.f = v \) (\( v \) is a reference type variable) in \( P_T \)

Inconsistency implies imprecision

Two conditions are satisfied and indicate potential imprecision.

1. Concrete values are different.
2. \( m.f \) has a symbolic value
Outline

• Problem
• Heap Cloning
• Empirical evaluation
• Conclusion

Empirical Study

Implementation: Cinger

* our prior work from TACAS'07
**Empirical Study**

**Subjects**

<table>
<thead>
<tr>
<th>Subject</th>
<th>Covered lines of code</th>
<th>Avg. no. of conjuncts in path constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Application</td>
<td>Library</td>
</tr>
<tr>
<td>NanoXML</td>
<td>1,230</td>
<td>14,604</td>
</tr>
<tr>
<td>JLex</td>
<td>6,566</td>
<td>13,702</td>
</tr>
<tr>
<td>Sat4J-Dimacs</td>
<td>3,908</td>
<td>17,195</td>
</tr>
<tr>
<td>Sat4J-CSP</td>
<td>4,125</td>
<td>39,617</td>
</tr>
<tr>
<td>BCEL</td>
<td>2,321</td>
<td>12,659</td>
</tr>
<tr>
<td>Lucene</td>
<td>20,821</td>
<td>56,622</td>
</tr>
</tbody>
</table>

**Goal**

Evaluate the reduction in number of methods for which user must write models, when $P'$ is the set of all native methods that the program uses.

**Method**

1. Estimate the number of native methods that may cause side effects
2. Count the number of native methods that introduce imprecision when symbolic execution is performed on Heap-Cloning transformed program.
3. Compare the two
Empirical Study

Results

<table>
<thead>
<tr>
<th>Subject</th>
<th>Number of unique native methods that were executed and that took reference-type parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>NanoXML</td>
<td>4</td>
</tr>
<tr>
<td>JLex</td>
<td>6</td>
</tr>
<tr>
<td>Sat4J-Dimacs</td>
<td>0</td>
</tr>
<tr>
<td>Sat4J-CSP</td>
<td>9</td>
</tr>
<tr>
<td>BCEL</td>
<td></td>
</tr>
<tr>
<td>Lucene</td>
<td></td>
</tr>
</tbody>
</table>

Over all subjects, only one native method System.arraycopy introduced imprecision, and thus needed model.

Summary and Future Work

1. Heap Cloning worked well for selected subjects. In future, use more and diverse set of subjects.

2. Heap Cloning reduced manual effort when P' consisted of all native methods. In future, treat framework (e.g., Google’s Android framework) code as part of P'.

3. Heap Cloning could identify where models are necessary. In future, check for correctness of user-specifed models.
Contributions

1. Heap Cloning technique that enables correct dynamic symbolic execution, but requires minimal manual effort to specify models
2. Implementation of Cinger system that is capable of correct dynamic symbolic execution of real-world Java programs
3. Empirical studies that shows Heap Cloning identifies a small number of methods that cause imprecision and need to be manually modeled