Leveraging Test Generation and Specification Mining for Automated Bug Detection without False Positives

Michael Pradel and Thomas R. Gross

Department of Computer Science
ETH Zurich
Motivation

Class X

API

API usage protocol
Motivation

API usage protocol

Client → Class X → API
Motivation

X must ensure safe API usage
class X {
    private Stack s = new Stack();
    public String report() {
        return get().toString();
    }
    private Object get() {
        s.peek();
    }
}
Example from Apache Xalan

Stack has a protocol ...

```java
class X {
    private Stack s = new Stack();

    public String report() {
        return get().toString();
    }

    private Object get() {
        s.peek();
    }
}
```
Example from Apache Xalan

Stack has a protocol ...

... but X fails to ensure it:

```
X x = new X();
x.report();
```

```
class X {
    private Stack s = new Stack();
    public String report() {
        return get().toString();
    }

    private Object get() {
        s.peek();
    }
}
```

EmptyStackException
Unsafe API Usage

Client → Class X → API

API usage protocol
Unsafe API Usage

Client → Class X → API

API usage protocol

exception
Unsafe API Usage

Unsafe API usage:
X exposes client to unexpected protocol exception
Unsafe API Usage

How to find unsafe API usages?

Unsafe API usage:
X exposes client to unexpected protocol exception
Goal

Program → Tool → Unsafe API usages
Goal

Program \rightarrow Tool \rightarrow Unsafe API usages

Automatic and precise bug detection
Goal

Automatic and precise bug detection

Tests

Program

Formal specs

Tool

Unsafe API usages

False positives
State of the Art (1)

(Tests)

Program

Anomaly detection

Unsafe API usages

False positives

Nguyen et al. ’09
Wasylkowski + Zeller ’09
Thummalapenta + Xie ’09
Monperrus et al. ’10
Gabel + Su ’10
State of the Art (2)

Program → Type state checking → Unsafe API usages → Formal specs → False positives

- DeLine + Fähndrich ’04
- Bierhoff + Aldrich ’07
- Fink et al. ’08
- Naeem + Lhotak ’08
- Bodden ’10
State of the Art (3)

Program → Static code checker → (Unsafe API usages) → False positives

FindBugs
PMD
Goal

Program → Tool → Unsafe API usages

Automatic and precise bug detection
Approach

Dynamic protocol mining

Test generation

Runtime protocol verification
Approach

- Dynamic protocol mining
  - Provides protocols
  - Requires input to run program

- Test generation

- Runtime protocol verification
Approach

- **Dynamic protocol mining**
  - ✔️ provides protocols
  - ✗ requires input to run program

- **Runtime protocol verification**
  - ✔️ finds protocol violations
  - ✗ requires protocols and input

**Test generation**
Approach

Dynamic protocol mining

- Provides protocols
- Requires input to run program

Test generation

- Provides input
- Requires test oracle to find bugs

Runtime protocol verification

- Finds protocol violations
- Requires protocols and input
Approach

Program

Test generation

tests

tests

Dynamic protocol mining

protocols

Runtime protocol verification

Unsafe API usages
Approach

Program

Test generation

tests

tests

Dynamic protocol mining

protocols

tests

Runtime protocol verification

Unsafe API usages
Test Generation

Feedback-directed, random test generation [Randoop, Pacheco 2007]

Two kinds of tests:

```java
class Test {
    ...
    \(\times\)
}
```

Failing (exception or assertion violation)

```java
class Test {
    ...
    \(\checkmark\)
}
```

Passing
Approach

Program

Test generation

tests

tests

dynamic protocol mining

protocols

tests

Runtime protocol verification

Unsafe API usages
Approach

Program

Test generation

tests

dynamic protocol mining

Runtime protocol verification

Unsafe API usages

tests

protocols
Protocol Mining

[ASE’09 and ICSM’10, Pradel et al.]
Example

```java
new Stack()
→
s
s.push()

s.removeAllElements()

s.push()

s.pop()

s.peek()
```
Approach

Program

Test generation

Dynamic protocol mining

Runtime protocol verification

Tests

Protocols

Unsafe API usages
Approach

Program

Test generation

Dynamic protocol mining

Runtime protocol verification

Unsafe API usages

Tests

Protocols

Tests
Runtime Protocol Verification

- Check all instances of protocol
- Warn if non-existing transition is taken
Example

Test:

```java
class X {
    private Stack s = new Stack();
    public String report() {
        return get().toString();
    }

    private Object get() {
        s.peek();
    }
}

X x = new X();
x.report();
```

1. New Stack()
2. S.push()
3. S.pop()
4. S.peek()
5. S.removeAllElements()
Example

Test:

```java
X x = new X();
x.report();
```

class X {
    private Stack s = new Stack();
    public String report() {
        return get().toString();
    }
    private Object get() {
        s.peek();
    }
}

1

2

3

new Stack()  s.push()  s.push()  s.push()  s.pop()  s.peek()  s.removeElements()

1  s  2  s  3  s  3  s  1  s  2  s
Example

Test:

```java
class X {
    private Stack s = new Stack();
    public String report() {
        return get().toString();
    }
    private Object get() {
        s.peek();
    }
}
X x = new X();
x.report();
```

1. `new Stack()`
2. `s.push()`
3. `s.push()`
4. `s.push()`
5. `s.pop()`
6. `s.peek()`
7. `s.removeAllElements()`
Example

Test:

```java
class X {
    private Stack s = new Stack();
    public String report() {
        return get().toString();
    }
    private Object get() {
        s.peek();
    }
}

X x = new X();
x.report();
```

Protocol violation
False Positives

Challenge:
Incomplete protocols (depend on mining)

1. new Stack() → s
2. s.push()
3. s.removeAllElements() → s
4. s.push()
5. s.pop()
6. s.peek()
False Positives

Challenge:
Incomplete protocols (depend on mining)

Program:
Stack s = new Stack();
s.push(..);
s.push(..);
s.push(..);
False Positives

Challenge:
Incomplete protocols (depend on mining)

Program:
Stack s = new Stack();
s.push(..);
false positive
s.push(..);

False positive protocol violation
Challenge:
Incomplete protocols (depend on mining)

Program:
Stack s = new Stack();
s.push(..);
s.push(..);

Protocol violation ≠ Bug

False positive protocol violation
Warnings without False Positives

Protocol violation

Program crash

Undeclared exception
Warnings without False Positives

- Protocol violation
- Program crash
- Undeclared exception

May be due to incomplete protocol
Warnings without False Positives

- Protocol violation
- Undeclared exception

- Program crash
  - May be due to illegal use of class
Warnings without False Positives

- Protocol violation
- Program crash
- Undeclared exception

Warnings to report
Warnings without False Positives (2)

Only report problem if:

- protocol violated
  and
- protocol-violating call fails the test
  and
- protocol-violating method does not declare the exception
Example (again)

Program:

```
Stack s = new Stack();
s.push(..);
s.push(..);
```

No warning, since protocol violation doesn’t raise exception
Approach

Program → Test generation

Test generation → Dynamic protocol mining
Dynamic protocol mining → protocols
protocols → Runtime protocol verification
Runtime protocol verification → Unsafe API usages
Evaluation

Implemented into fully automatic tool

Main questions:

1. Effectiveness in finding unsafe API usages
2. Comparison with existing work
3. Performance
Setup

Programs:
- DaCapo benchmarks (5,012 classes)

APIs:
- Collection+Iterator
- Vector+Enumeration

Stopping criterion:
- Generate 10,000 tests per program
Unsafe API Usages

54 unsafe API usages

0 false positives
Example from Jython

```java
public class X {
    protected Iterator iter;
    public void _beginCanonical() {
        iter = classes.values().iterator();
    }
    public Object _next() {
        if (iter.hasNext()) return iter.next();
        else return null;
    }
    public void _flushCurrent() {
        iter.remove();
    }
}
```
public class X {

    protected Iterator iter;

    public void _beginCanonical() {
        iter = classes.values().iterator();
    }

    public Object _next() {
        if (iter.hasNext()) return iter.next();
        else return null;
    }

    public void _flushCurrent() {
        iter.remove();
    }

}
Example from Jython

```java
public class X {

    protected Iterator iter;

    public void _beginCanonical() {
        iter = classes.values().iterator();
    }

    public Object _next() {
        if (iter.hasNext()) return iter.next();
        else return null;
    }

    public void _flushCurrent() {
        iter.remove();
    }

}
```

Safe API usage
Example from Jython

```java
public class X {
    protected Iterator iter;
    public void _beginCanonical() {
        iter = classes.values().iterator();
    }
    public Object _next() {
        if (iter.hasNext()) return iter.next();
        else return null;
    }
    public void _flushCurrent() {
        iter.remove();
    }
}
```

- Protocol violation
- Crash through exception
- Not declared
Kinds of Bugs

Diverse kinds of unsafe API usages

- Invalid indexing of lists and vectors
- Iterators: Illegal next() and remove()
- Accessing non-existing elements: E.g., pop()

All unsafe API usages for download:
http://mp.binaervarianz.de/icse2012-leveraging/
Comparison with Prior Work

JDK-API usage in DaCapo:

<table>
<thead>
<tr>
<th>OCD</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 potential bug</td>
<td>54 crashing bugs</td>
</tr>
<tr>
<td>2 false positives</td>
<td>0 false positives</td>
</tr>
</tbody>
</table>

OCD [Gabel+Su, ICSE’10]
Comparison with Prior Work

JDK-API usage in DaCapo:

<table>
<thead>
<tr>
<th>OCD [Gabel+Su, ICSE’10]</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 potential bug</td>
<td>54 crashing bugs</td>
</tr>
<tr>
<td>2 false positives</td>
<td>0 false positives</td>
</tr>
</tbody>
</table>

DaCapo input vs. generated input
**Comparison with Prior Work**

**JDK-API usage in DaCapo:**

<table>
<thead>
<tr>
<th>OCD [Gabel+Su, ICSE’10]</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 potential bug</td>
<td>54 crashing bugs</td>
</tr>
<tr>
<td>2 false positives</td>
<td>0 false positives</td>
</tr>
</tbody>
</table>

Avoid false positives by construction
Performance

Between less then a minute and several minutes per program-API pair

Optimization: Find bugs with 5x less tests

- Static analysis: Prioritize methods
- Guide random test generator towards API-relevant parts of program
Summary

- Benefits of dynamic analysis without providing input
- Find bugs with mined specifications without false positives
- Guide test generator towards API

Program → Automatic and precise analysis → Unsafe API usages
Conclusion

Don’t waste precious developer time

Lots of testing with little effort

Photo: Alfonso Silóniz
Leveraging Test Generation and Specification Mining for Automated Bug Detection without False Positives

Michael Pradel, Thomas R. Gross

Department of Computer Science
ETH Zurich

Thank you!