Automatic Recovery from Runtime Failures
The Problem

Deployed applications.

- Always have some bugs
- Really hard to understand and reproduce
- Deliver the patch
- Customer dissatisfaction
The Problem

Deployed applications.

- Always have some bugs
- Really hard to understand and reproduce
- Deliver the patch
- Customer dissatisfaction

What can we do?
The Problem

Deployed applications.

- Always have some bugs
- Really hard to understand and reproduce
- Deliver the patch
- Customer dissatisfaction

What can we do?
Natural answer: Prevent field failures
Key new idea

Make two assumptions about the software:
Key new idea

Make two assumptions about the software:

1. Applications use libraries
Key new idea

Make two assumptions about the software:

1. Applications use libraries
2. Intrinsic redundancy
Key new idea

Make two assumptions about the software:

1. Applications use libraries
2. Intrinsic redundancy

Exploit these observations to deliver a system that allows deployed software to recover from failures related to libraries.
ARMOR

System that implements the automatic workaround technique.
Analyze the application

```javascript
this.onEnterFrame = function(){
  // get current time.
  cTime = new Date();
  hour = cTime.getHours();
  minute = cTime.getMinutes();
  second = cTime.getSeconds();
  // Converting to 12 hour clock.
  if(hour > 12) hour -= 12;
  // Determine the angle of each hand.
  hourAngle = 360 * hour / 12;
  minAngle = 360 * minute / 60;
  secAngle = 360 * second / 60;
  // Visually set angle of the hands.
  hour_hand._rotation = hourAngle;
  minute_hand._rotation = minAngle;
  second_hand._rotation = secAngle;
}
```
Analyze the application

```javascript
this.onEnterFrame = function(){
    // get current time.
    cTime = new Date();
    hour = cTime.getHours();
    minute = cTime.getMinutes();
    second = cTime.getSeconds();
    // Converting to 12 hour clock.
    if(hour > 12) hour -= 12;
    // Determine the angle of each hand.
    hourAngle = 360 * hour / 12;
    minAngle = 360 * minute / 60;
    secAngle = 360 * second / 60;
    // Visually set angle of the hands.
    hour_hand._rotation = hourAngle;
    minute_hand._rotation = minAngle;
    second_hand._rotation = secAngle;
}
```
ARMOR - Big Picture

Instrument the code with alternatives.
During this phase, ARMOR identifies the Roll-Back Areas.

- Calls operation of any one of the libraries
- Failure can be detected and reported

Applied to method bodies and field initialization.
Precompile alternatives (2)

Where do we get the alternatives?
Precompile alternatives (2)

Where do we get the alternatives?

- Requires a library to come with a specification of the equivalences!
Precompile alternatives (2)

Each RBA comes with an alternative that is based on the specification of the library.

Creates several versions of the class that contains RBA with different versions for a particular method.

Pre-compiles these alternatives into classes that can be loaded at failure.
Initialization expressions are encapsulated into a method.

A proxy method is created for every method.

A proxy method stores check point, attempts the standard method call and responds to failures if needed.
Example

```java
public void addToCollection(String s) {
    arrayList.add(s);
}

/*
   add(E o)
   add(index, E o)
   addAll(Collection<? extends E> c)
*/
```
Example

can be substituted with...

```java
public void addToCollection(String s){
    arrayList.add(s);
}
```

```java
public void addToCollection(String s){
    arrayList.add(arrayList.size(), s);
}
```
Example

can be substituted with...

```java
class Example {
    public void addToCollection(String s){
        arrayList.add(s);
    }
}
```

or something as bizarre as this...

```java
class Example {
    public void addToCollection(String s){
        SomeCollection<String> collection = new SomeCollection<String>();
        collection.put(s);
        arrayList.addAll(collection);
    }
}
```
public void addToCollection_original(String s) {
    arrayList.add(s);
}

public void addToCollection(String s) {
    try {
        create_checkpoint();
        addToCollection_original(s);
    } catch (Exception e) {
        boolean success = false;
        while (!success && more_rb_variants_available) {
            try {
                restore_checkpoint();
                load_new_rba_variant();
                addToCollection_original(s);
                success = true;
            } catch (Exception e2) {
                // record and adjust
            }
        }
        if (!success) throw ex;
    } finally {
        discard_checkpoint();
    }
}
Evaluation - Measure

2 libraries: JodaTime and Guava.
4 applications: Fb2pdf, Carrot2, Caliper, Closure.

Effectiveness: Mutation testing. How many times did the application successfully recover from a mutant. (With correct output)

Overhead: Compare total execution time and allocated memory in instrumented non failing run and original application.
Evaluation - Effectiveness

Effectiveness ranges from 19% - 48%

<table>
<thead>
<tr>
<th></th>
<th>Caliper</th>
<th>Carrot2</th>
<th>Closure</th>
<th>Fb2pdf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total mutants</td>
<td>21297</td>
<td>21297</td>
<td>21297</td>
<td>16858</td>
</tr>
<tr>
<td>Relevant mutants</td>
<td>309</td>
<td>187</td>
<td>344</td>
<td>2200</td>
</tr>
<tr>
<td></td>
<td>210</td>
<td>120</td>
<td>177</td>
<td>1805</td>
</tr>
<tr>
<td>success</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>non-equivalent</td>
<td>0</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>detected</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>not detected</td>
<td>12</td>
<td>9</td>
<td>15</td>
<td>47</td>
</tr>
<tr>
<td>execution</td>
<td>87</td>
<td>47</td>
<td>149</td>
<td>347</td>
</tr>
<tr>
<td>error</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total mutants run with ARMOR

<table>
<thead>
<tr>
<th></th>
<th>Caliper</th>
<th>Carrot2</th>
<th>Closure</th>
<th>Fb2pdf</th>
</tr>
</thead>
<tbody>
<tr>
<td>(28%)</td>
<td>24</td>
<td>(48%)</td>
<td>(47%)</td>
<td>(19%)</td>
</tr>
<tr>
<td>Mutants where ARMOR is successful</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Evaluation - Overhead

Overhead ranges from 2% - 194%
Discussion questions

● What can we build upon this idea?
Discussion questions

- What are the possible downsides?
Discussion questions

What are the possible downsides?

● Requires specification requirement.
● Overhead is high.
● Relies on redundancies in libraries.
● Deep library failure.
Discussion questions

- How to avoid downsides?
- Why it is useful?
- How can this idea improve the way that developers write code?
- What is the benefit of ARMOR in the context of field failure.
Conclusion

1. An analysis of the effectiveness of the automatic-workaround technique in resolving software faults at runtime.

2. Automatic workarounds exploit the property of intrinsic redundancy within libraries that the application employs to attempt recovery from a crash.

3. The tool ARMOR implements the automatic-workaround technique.