Testing Configuration Changes in Context to Prevent Production Failures

Xudong Sun*, Runxiang Cheng*, Jianyan Chen, Elaine Ang, Owolabi Legunsen, Tianyin Xu

University of Illinois at Urbana-Champaign  Cornell University

*Co-primary authors
Configurations are continuously deployed

• Configurations are changed in frequent "diffs"
  • Updated hundreds to thousands times a day

• Configuration changes induce **two types of production system failures**
  1. Dormant bugs in the system exposed by valid configuration changes
  2. Erroneous configuration values causing unexpected behavior

• Ctest: detect both types of failure-inducing configuration changes
Ctest complements existing software testing

• Testing is hard to cover all possible configuration value combinations
  • Workaround: test a few representative ones (e.g., default values)

• But, production system configurations may not be tested
  • Production configurations are the ones that matter

• Ctest: focus on testing production system configurations
  • When configuration changes, test it against the code
Ctest complements static configuration validation

• Configuration validation frameworks are based on correctness rules
  • Automatic techniques learn correctness rules

• Static validation cannot detect bugs exposed by valid value changes
  • The root cause is outside the configuration

• It is hard to codify and maintain all correctness rules

• Ctest: check program behavior without the need for rules
Contributions

• Ctest: a new perspective for detecting failure-inducing configurations
  • **Key idea: connect production system configurations to software tests**
  • Ctest checks program behavior against configurations to be deployed
  • Ctest detects both types of failure-inducing configuration changes

• A methodology of generating ctests from existing software tests

• Evaluations on the effectiveness of ctests in different scenarios
Ctest definition

• A ctest is a software test parameterized by configuration parameters
  • Run by instantiating input configuration parameters with concrete values
  • Exercise system code and assert program behavior
  • Can be a unit, an integration, or a system test
A ctest that detects bugs exposed by valid changes

```java
@Ctest
public void testRefreshCallQueueProtocol() {
    Configuration conf = new Configuration();
    Server server = new Server(conf);
    server.authorize(...);
    ...
}

void authorize(...) {
    if (authz) {
        acls = protocolToAcls.get(...);
        if (acls == null)
            throw new AuthorizationException(...);
    }
}
```
A ctest that detects erroneous configuration values

```java
@Ctest
class TestRpcScheduler {
    public void testRpcScheduler() {
        Configuration conf = new Configuration();
        RpcScheduler scheduler = new RpcScheduler(conf);
        ...
        int totalTime = scheduler.dispatch(tasks);
        assertEquals(930, totalTime);
    }
}
```

AssertionError

- factor = 0.1
+ factor = 2

Out-of-range

Error!
Ctests can be used for ...

• Checking entire system configuration

• Checking a configuration diff
  • For each configuration diff, only rerun relevant ctests

• Checking a configuration file
  • A configuration file is a "diff" over the default system configuration
Using ctests to check a configuration diff

• Rerun only ctests that exercise changed configuration parameters

<table>
<thead>
<tr>
<th>Config File</th>
<th>Config Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1 = 0.1</td>
<td>- p1 = 0.1</td>
</tr>
<tr>
<td>p2 = false</td>
<td>- p2 = false</td>
</tr>
<tr>
<td>p3 = /data</td>
<td>+ p1 = 2</td>
</tr>
<tr>
<td>p4 = foo</td>
<td>+ p2 = true</td>
</tr>
<tr>
<td>p5 = bar</td>
<td></td>
</tr>
</tbody>
</table>

All Ctests
- t1(p1, p2)
- t2(p2, p3)
- t3(p3)
- t4(p4, p5)

Selected Ctests
- t1(p1, p2)
- t2(p2, p3)
- t3(p3)
- t4(p4, p5)

> ctest.py config.diff
mvn test -Dtest=t1(p1=2,p2=true)
mvn test -Dtest=t2(p2=true,p3=/data)

ctest.py is implemented on top of Maven
Ctests can be transformed from existing tests

• Software and DevOps engineers can also write new ctests

• Mature software projects have high-quality test code
  • 70+% statement and method coverage in the evaluated systems
  • Higher coverage is reported in commercial projects

• **Insight:** reuse well-engineered test logic and oracles
Transforming existing tests into ctests

• **Step 1**: Identify configuration parameters exercised by an existing test
  • Instrument configuration APIs

• **Step 2**: Connect existing tests to the production configuration
  • Intercept configuration APIs

• **Step 3**: Respect explicit and implicit test assumptions on configurations
Step 1: Instrumentation (example from Hadoop)

- Configuration interfaces **(used in test code)**
  
  ```java
  String get(String parameter);
  void set(String parameter, String value);
  ```

- Test example
  
  ```java
  @Test
  void testRefreshCallQueueProtocol {
    ...
    authorize = conf.get("hadoop.security.authorization");
    ...
  }
  ```
Step 1: Instrumentation (example from Hadoop)

• Configuration interfaces (used in test code)

```java
String get(String parameter) {
    + LOG.info("[GET] " + parameter);
    ...
}
```

• Test example

```java
@Test
void testRefreshCallQueueProtocol {
    ...
    authorize = conf.get("hadoop.security.authorization");
    ...
}
```
Step 2: Interception (example from Hadoop)

- Configuration interfaces (**used in test code**)
  ```java
  String get(String parameter) {
    + LOG.info("[GET] " + parameter);
    ...
  }
  ```

- Configuration initialization
  ```java
  static {
    ...
    addDefaultResource("core-default.xml");
    addDefaultResource("core-site.xml");
    + addDefaultResource("core-ctest.xml");
  }
  ```
Step 3: Respect test assumptions on configurations

- Tests can **explicitly** reset configuration values
- Tests can also **implicitly** assume configuration values (w/o explicit reset)

```java
@Test
public void testNameNodeXFrameOptionsEnabled() {
    conf.set("dfs.xframe.enabled", "true");
    String xfoHeader = conn.getHeaderField("X-FRAME-OPTIONS");
    Assert.assertTrue(xfoHeader.endsWith(XFrameOption.SAMEORIGIN));
}
```
Our experience on generating ctests

• Selected 392 configuration parameters in five cloud systems
  • Hadoop Common, HDFS, HBase, Alluxio, and ZooKeeper

• Instrumentation effort
  • 24–130 lines of code (in 1–3 classes)

• Generated 7000+ ctests for all the selected configuration parameters

• Rewrote 102 ctests by changing 190 lines of code
  • Assess the opportunity of manual rewriting efforts
Evaluation

• How effectively do ctests prevent configuration-induced failures?
  • 64 real-world configuration-induced failures
  • Ctests detected 62 out of 64 failure-inducing configurations

• How effectively do ctests detect diverse types of misconfigurations?
  • 1,055 synthesized misconfiguration values
  • 72% were detected by the generated ctests

• How do ctests detect misconfigurations in the wild?
  • 92 configuration files collected from public docker images
  • 10 misconfigurations in 7 configuration files
### Ctest effectiveness

<table>
<thead>
<tr>
<th>Root cause</th>
<th># Failures</th>
<th>Ctest gen-only</th>
<th>Ctest gen+rewrite</th>
<th>Spellcheck</th>
<th>PCheck</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software bugs</td>
<td>13</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Erroneous values</td>
<td>51</td>
<td>41</td>
<td>51</td>
<td>3</td>
<td>41</td>
</tr>
<tr>
<td>• Value type errors</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>• Corrupt config files</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>• Out-of-range values</td>
<td>12</td>
<td>11</td>
<td>12</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>• Value semantic errors</td>
<td>22</td>
<td>16</td>
<td>22</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>• Dependency violations</td>
<td>10</td>
<td>7</td>
<td>10</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>• Resource violations</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>64</td>
<td><strong>51 (79.7%)</strong></td>
<td><strong>62 (96.9%)</strong></td>
<td><strong>3 (4.8%)</strong></td>
<td><strong>41 (66.1%)</strong></td>
</tr>
</tbody>
</table>
Limitations

• Ctest effectiveness relies on the quality of tests
  • The two missing cases are due to lack of effective tests
  • One missing case can be detected by a ctest generated in the latest test suite

• Ctest generation methods are neither sound nor complete
  • It could have both false positives and false negatives
  • We did not observe false positives

• Ctests do not bridge the gap between testing env. and production env.
Conclusion

• Ctest: a new perspective for detecting failure-inducing configurations
  • Key idea: connect production system configurations to software tests
  • Ctest checks program behavior against configurations to be deployed
  • Ctest detects both types of failure-inducing configuration changes

• A methodology of generating ctests from existing software tests

• Evaluations on the effectiveness of ctests in different scenarios

• Code and datasets: https://github.com/xlab-uiuc/opencstest