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

```
--- hdfs-site.xml
+++ hdfs-site.xml
- authz = false
+ authz = true
```

- 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

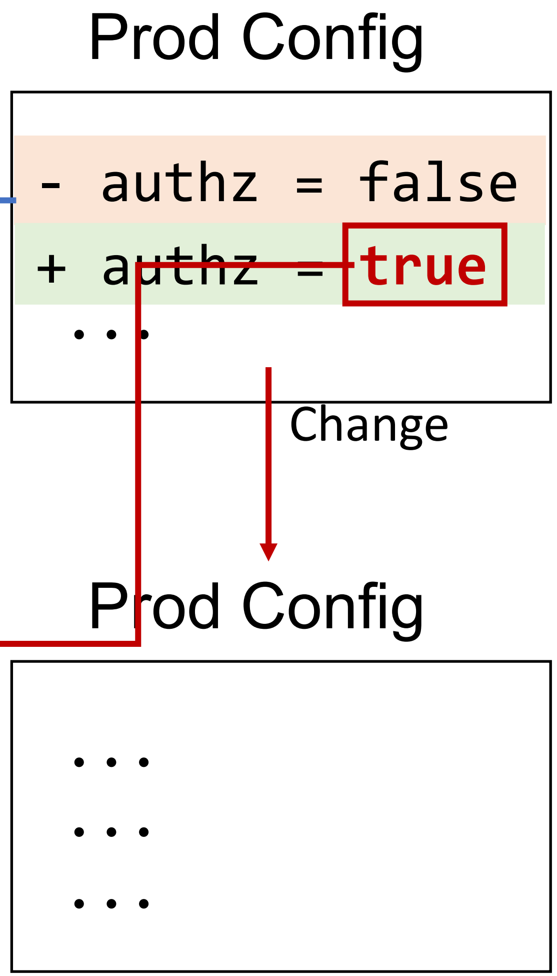
Ctest definition

- 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

```
@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(...);
  }
}
```



Prod Config

```
- authz = false
+ authz = true
 ...
```

Change

Prod Config

```
...
...
...
```

A ctest that detects erroneous configuration values

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

AssertionError

Prod Config

```
- factor = 0.1
+ factor = 2
```

Change

Prod Config

```
...
Out-of-range
...
... Error!
```

Ctests can be used for …

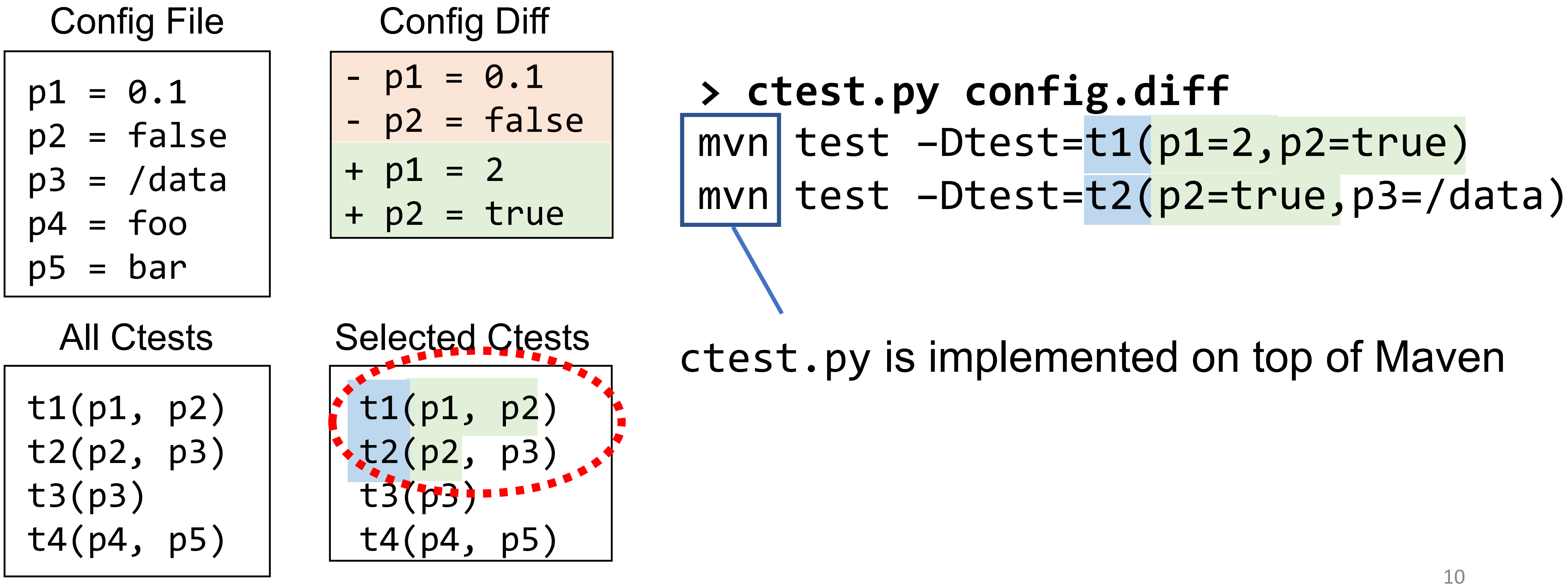
- Checking entire system configuration

- 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



Config File

```
p1 = 0.1
p2 = false
p3 = /data
p4 = foo
p5 = bar
```

Config Diff

```
- p1 = 0.1
- p2 = false
+ p1 = 2
+ p2 = true
```

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**)

```
String get(String parameter);
void set(String parameter, String value);
```

read

write

Key-value configuration store

- Test example

```
@Test
void testRefreshCallQueueProtocol {
  ...
  authorize = conf.get(“hadoop.security.authorization”);
  ...
}
```

Step 1: Instrumentation (example from Hadoop)

- Configuration interfaces (**used in test code**)

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

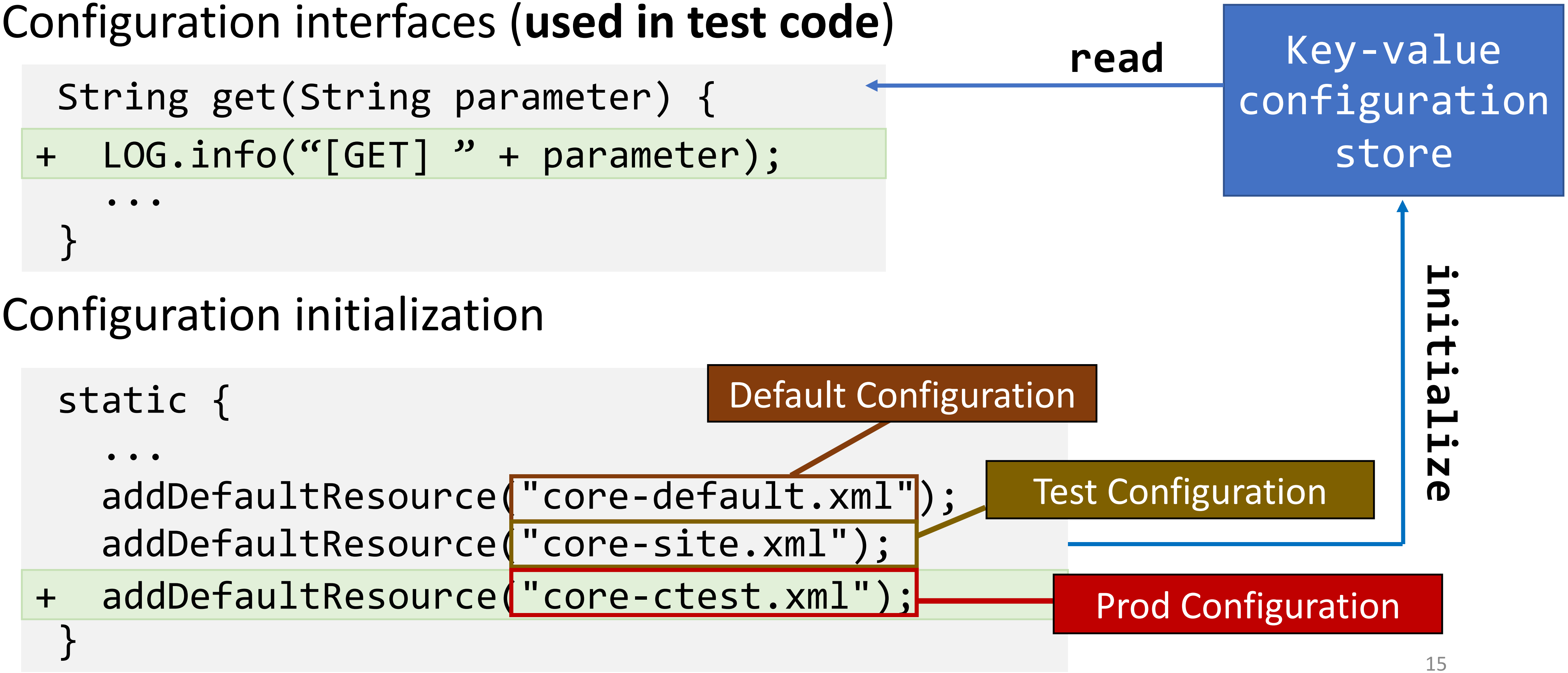
read

Key-value configuration store

- Test example

```
 @Test
 void testRefreshCallQueueProtocol {
   ...
   authorize = conf.get(“hadoop.security.authorization”);
   ...
 }
```

Step 2: Interception (example from Hadoop)



- Configuration interfaces (**used in test code**)

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

- Configuration initialization

```
 static {
    ...
    addDefaultResource("core-default.xml");
    addDefaultResource("core-site.xml");
+   addDefaultResource("core-ctest.xml");
 }
```

read

Key-value configuration store

initialize

Default Configuration

Test Configuration

Prod Configuration

Step 3: Respect test assumptions on configurations

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

```
@Test
public void testNameNodeXFrameOptionsEnabled() {
   ...
   conf.set(“dfs.xframe.enabled”, “true”);
   ...
   String xfoHeader = conn.getHeaderField("X-FRAME-
   ...
   Assert.assertTrue(xfoHeader.endsWith( conf.get("dfs.xframe.value") ));
}
```

Explicit assumption

Implicit assumption

Default value of dfs.xframe.value

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

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

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/openctest GitHub