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

Abstract

Large-scale cloud services deploy hundreds of configuration changes to production systems daily. At such velocity, configuration changes have inevitably become prevalent causes of production failures. Existing misconfiguration detection and configuration validation techniques only check configuration values. These techniques cannot detect common types of failure-inducing configuration changes, such as those that cause code to fail or those that violate hidden constraints.

We present ctests, a new type of tests for detecting failure-inducing configuration changes to prevent production failures. The idea behind ctests is simple—connecting production system configurations to software tests so that configuration changes can be tested in the context of code affected by the changes. So, ctests can detect configuration changes that expose dormant software bugs and diverse misconfigurations.

We show how to generate ctests by transforming the many existing tests in mature systems. The key challenge that we address is the automated identification of test logic and oracles that can be reused in ctests. We generated thousands of ctests from the existing tests in five cloud systems.

Our results show that ctests are effective in detecting failure-inducing configuration changes before deployment. We evaluate ctests on real-world failure-inducing configuration changes, injected misconfigurations, and deployed configuration files from public Docker images. Ctests effectively detect real-world failure-inducing configuration changes and misconfigurations in the deployed files.

1 Introduction

1.1 Motivation

Large-scale cloud and Internet services evolve rapidly and deploy hundreds to thousands of configuration changes to production systems daily [35, 38, 53, 55]. For example, at Facebook, thousands of configuration changes are committed daily, outpacing the frequency of code changes [55]. Other cloud services such as Google and Azure also frequently deploy configuration changes [9, 10, 38].

The high velocity of configuration changes has led to prevalent configuration-induced failures. For example, faulty configurations are the second largest cause of service disruptions in a main Google production service [5]. At Facebook, 16% of service-level incidents, including major outages [54], are induced by configuration changes [55]. Similar levels of severity and prevalence of configuration-induced failures occur in other cloud systems [19, 34, 40, 42, 74].

Based on our experience from analyzing hundreds of configuration-induced incidents, failure-inducing configuration changes are rarely caused by trivial mistakes (e.g., typos). This rarity is attributed to the DevOps practices that enforce change review and validation [6, 27, 55]. As a result, the root causes of configuration-induced failures are often non-trivial; they commonly reside in the program and not in the changed configurations. Failures typically occur when *valid* configuration changes expose dormant software bugs [55] and when configuration changes violate undocumented, hidden configuration constraints. The root causes of the former are in the program, while the latter are often due to configuration design or implementation flaws [69]. Review and validation of configuration changes alone can hardly detect failures resulting from these root causes.

Researchers have proposed several configuration validation and misconfiguration detection techniques [70]. These include new languages and frameworks for implementing validators [6, 27, 55], detection techniques that use machine learning and document analysis to infer correctness rules on configuration values [38, 43, 44, 49, 50, 59, 61, 75, 77], and type or constraint checkers [48, 67]. These techniques are successful, but they are limited:

- Existing techniques only check configuration values and cannot detect configuration changes that cause code to fail.
- Very few existing techniques can detect "legal misconfigurations" [71], which have syntactically and semantically valid values but result in unexpected behavior.
- It is costly and hard for human-written or machine-learned rules to check the often subtle, version-specific [78], and inconsistent [69] configuration requirements.

1.2 Contributions

We present ctests, a new type of tests for detecting failure-inducing configuration changes to prevent production failures.

[*]Co-primary authors

Ctests take a simple and effective approach—connecting software tests with production system configurations. In this way, ctests can test configuration changes in the context of code that is affected by the changes. A ctest is parameterized by a set of system configuration parameters. Running a ctest instantiates each of its input parameters with a configuration value from production or a value to be deployed to production. Like regular software tests, ctests exercise system code and assert that program behavior satisfies certain properties (correctness, performance, security, etc). Ctests can be unit, integration, or system tests.

Existing software testing techniques do not connect tests to *actual* production system configurations. Rather, existing testing techniques sample *possible* configurations through systematic or random exploration of the enormous space of configuration value combinations [37]. Systematic exploration can be prohibitively expensive due to combinatorial explosion [39], while random exploration can have a low probability of covering all offending values that can cause production failures. Ctests have neither the cost of systematic exploration nor the low coverage of random exploration. By connecting tests to production system configurations, ctests can effectively detect failure-inducing configurations.

Ctests can test entire system configurations or incremental configuration changes in the form of configuration file "diffs." Our ctest infrastructure (see §3) supports *selectively* running only the ctests that are relevant to a configuration change, instead of re-running all ctests. Selectively running ctests saves testing time—most real-world configuration changes modify only a few configuration values [55].

We show how to generate ctests by transforming the existing and abundant tests in mature software projects in an automated fashion that reuses well-engineered test logic and oracles. The main challenge that we address is the automated identification of test logic and oracles that can be transformed into ctests. Existing test logic may assume specific configuration values. Such assumptions can be implicit (assuming default values) or explicit (hardcoding certain values). Thus, naïve parameterization will not always generate valid ctests.

Our transformation identifies and respects the intent of existing tests that assume specific configuration values. First, configuration parameters whose values are explicitly reassigned in the test code are excluded from the input parameter set of a ctest. Then, the values of the parameters used in candidate ctests are varied to observe the corresponding test output. We exclude parameters whose values are hardcoded in a test because such tests will fail on different but valid values. Our tests-to-ctests transformation is mechanized in a toolchain and we successfully generated 7,974 ctests by transforming the existing test suites in five cloud systems.

Ctests address the following limitations of existing configuration validation and misconfiguration detection techniques:

- Ctests can detect failure-inducing configuration changes where the root cause of the failure is in the code.
- Ctests can detect legal misconfigurations by capturing the resulting unexpected system behavior.
- Ctests can be generated from existing tests, without incurring the high cost of learning or codifying rules.

Our results show that ctests can effectively detect failure-inducing configuration changes before deploying them to production. We evaluated ctests using 64 real-world configuration-induced failures, 1,055 diverse misconfigurations generated by error injection rules, and 92 deployed configuration files from publicly-available Docker images.

Ctests detected the failure-inducing configurations in 96.9% of the real-world failures. The ctests that detected these real-world failures were transformed from the tests in the older version of the systems on which the failures were reported. That is, ctests could have detected these failures earlier. Ctests also detected 10 misconfigurations in 7 deployed files. Additionally, our ctest generation process exposed 14 previously unknown bugs, including a bug that users encountered after we reported it [24]. Developers confirmed 12 of these 14 bugs and fixed 10 of them.

In summary, this paper makes the following contributions:

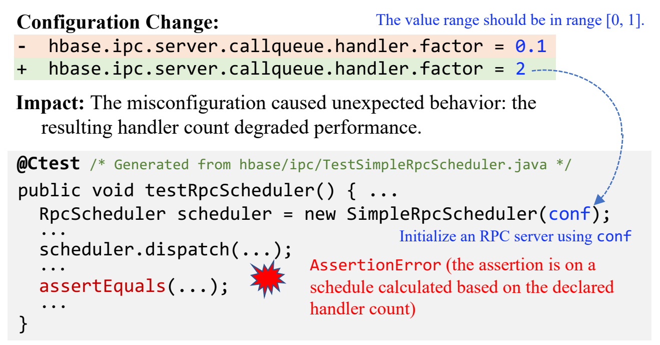
- Ctests enable a simple and effective approach for detecting failure-inducing configuration changes.
- We present how to generate ctests by transforming the many existing tests in mature systems.
- We show that ctests can effectively detect real-world configuration-induced failures early, during testing.

2 Background and Examples

We describe how ctests address the limitations of existing techniques for validating configuration values [6,27,48,55,67] and techniques for detecting specific types of misconfigurations [38,43,44,49,50,59,61,62,75,77].

Checking configurations based on program behavior. A key capability of ctests is to check how actual configuration values impact program behavior. This capability is essential for detecting configuration changes that result in code failures or expose dormant bugs. In our experience, checking program behavior can be more effective in capturing failure-inducing configuration changes than checking configuration values against rules (which are usually incomplete).

Figure 1 uses a real-world issue from HBase [21] to illustrate the capability of ctests. There, a ctest detects a misconfiguration that degrades performance ("*too many handlers can be counter-productive* [56]"). The ctest is generated from an existing test in the reported HBase version. It asserts on the computed schedule with the expected behavior that handler counts are not affected by configuration changes. The offending value is "legal" [71] but the reported version had no validation code to check the expected behavior.



Configuration Change:

The value range should be in range [0, 1].

```
- hbase.ipc.server.callqueue.handler.factor = 0.1
+ hbase.ipc.server.callqueue.handler.factor = 2
```

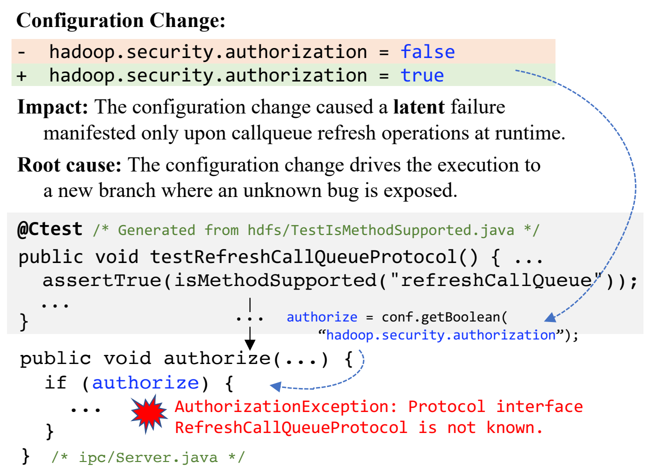
Impact: The misconfiguration caused unexpected behavior: the resulting handler count degraded performance.

```
@Ctest /* Generated from hbase/ipc/TestSimpleRpcScheduler.java */
public void testRpcScheduler() { ...
  RpcScheduler scheduler = new SimpleRpcScheduler(conf);
  ...
  scheduler.dispatch(...);
  ...
  assertEquals(...);
  ...
}
```

Initialize an RPC server using conf

AssertionError (the assertion is on a schedule calculated based on the declared handler count)

Figure 1: A ctest that detects a real-world misconfiguration in HBase [21] by checking expected system behavior. The ctest is generated from a test available in the reported HBase version.



Configuration Change:

```
- hadoop.security.authorization = false
+ hadoop.security.authorization = true
```

Impact: The configuration change caused a **latent** failure manifested only upon callqueue refresh operations at runtime.

Root cause: The configuration change drives the execution to a new branch where an unknown bug is exposed.

```
@Ctest /* Generated from hdfs/TestIsMethodSupported.java */
public void testRefreshCallQueueProtocol() { ...
  assertTrue(isMethodSupported("refreshCallQueue"));
  ...
}                     ...  authorize = conf.getBoolean(
                                "hadoop.security.authorization");
public void authorize(...) {
  if (authorize) {
    ...    AuthorizationException: Protocol interface
           RefreshCallQueueProtocol is not known.
  }
} /* ipc/Server.java */
```

Figure 2: A ctest that detects a dormant bug exposed by a configuration change in Hadoop [20]. The ctest is generated from a test available in the reported Hadoop version.

Detecting dormant bugs exposed by valid configuration changes. Ctests can detect not only misconfigurations but also software bugs exposed by valid configuration changes. Such bugs are common root causes of configuration-related incidents (§1.1). Existing configuration validation and misconfiguration detection techniques only check for erroneous configuration values; they are fundamentally limited to detecting failures with root causes outside the changed configurations. Such software bugs inevitably occur, despite extensive testing and static analysis. Some bugs can only be exposed under specific configurations. Figure 2 shows a real-world failure from Hadoop [20]. A failure-inducing configuration change caused Hadoop to traverse new execution paths and exposed a dormant software bug.

Detecting diverse misconfigurations. Many existing techniques focus on detecting specific kinds of misconfigurations. Ctests are generic. They can detect configuration changes that lead to any kind of unexpected program behavior. So ctests can detect misconfigurations that are hard for state-of-the-art techniques to detect. Such misconfigurations involve (1) custom regular expressions, user commands, and URIs (statistical analyses and machine learning can detect outliers but cannot deal with custom values [67]), (2) invalid content referred to by path-related configurations (most existing techniques only check metadata), (3) violations of undocumented constraints that cannot be found by text-based document analysis [44, 59, 65], and (4) dependencies among multiple configuration parameters [12]. Figures 8 and 9 show examples of misconfigurations detected by ctests that are hard to detect using existing techniques.

Incremental pre-deployment testing for every configuration change. Ctests can help *prevent* failure-inducing configuration changes from being deployed to production systems. The goal of ctests is to test every configuration change early, during testing. Ctests can be run selectively on configuration file "diffs" to save testing time (§3.2). Ctests do not suffer from limitations of post-deployment configuration checking (e.g., disallowing operations with side effects to avoid corrupting production system states as in PCheck [67]).

3 Ctest Overview

The idea behind ctests is to connect production system configurations to software tests, enabling the checking of configuration changes against program properties in the context of code affected by the configuration changes. Ctests detect *both* misconfigurations caused by assigning invalid values to configuration parameters and bugs in the code that are exposed by changing configuration parameters to new valid values.

3.1 Ctest Definition

A ctest, $\hat{t}(\hat{P})$, is a test $\hat{t}$ that is parameterized by a set of system configuration parameters $\hat{P}$. Running a ctest instantiates each parameter $p \in \hat{P}$ with a concrete value. In particular, each $p \in \hat{P}$ in a ctest can be instantiated with a value from the production system configuration or a configuration change (in the form of a configuration file "diff") to be deployed. Note that $\hat{P}$ is typically only a very small subset of all system configuration parameters, denoted as $\mathbb{P}$. That is, $|\hat{P}| \ll |\mathbb{P}|$.

Ctests can be unit, integration, or system tests. Like regular software tests, a ctest can assert on different kinds of program properties: correctness, performance, security, etc. Ctests can be written from scratch by developers, or they can be generated from existing software tests (see §4). Our generation procedure reuses test logic and assertions in existing tests during transformation to ctests.

3.2 Ctest Usage

Ctests can check an entire system configuration, a configuration change, or a configuration file. So, ctests can be used both as a traditional configuration file checker and as an enabler of configuration checking during continuous integration and deployment [52]. Ctests are complementary to configuration validation, similar to how software testing complements static analysis for bug detection.

Ctests for checking entire system configurations. We define a system configuration as the values of all configuration parameters in the system denoted as $C = \bigcup_{i=1..|\mathbb{P}|}\{(p_i \mapsto v_i)\}$ (it assigns a value v_i to every parameter p_i in $\mathbb{P}$). Running a ctest, $\hat{t}(\hat{P})$, instantiates each parameter $p_i \in \hat{P}$ with its value in the system configuration v_i such that $(p_i \mapsto v_i) \in C$. To test the system configuration, C, all available ctests are run. C passes if all ctests pass and fails if any ctest fails.

Ctests for checking configuration changes. In modern continuous integration and deployment, a configuration change has the form of a configuration file *"diff"*. A diff typically only changes the values of a small set of configuration parameters, P_D [55]. It updates the system configuration from C to C' by changing each $p_d \in P_D$'s value from v_d to v'_d. Formally, we define a configuration diff $D = \{(p_d \mapsto v'_d) \mid p_d \in P_D \textit{ and } (p_d \mapsto v_d) \in C \textit{ and } v_d \neq v'_d\}$.

For a given diff D, a ctest $\hat{t}(\hat{P})$ can be used to test D if at least one configuration parameter in D is in its input parameter set $\hat{P}$ (i.e., if $P_D \cap \hat{P} \neq \emptyset$). We use this *test selection criterion* to re-run only the subset of ctests whose outcome could be altered by D, instead of re-running all ctests after every configuration change.

A selected ctest $\hat{t}(\hat{P})$ can be run before deploying D to production by assigning values in D to the ctest's parameters that are in D and assigning values in C to the ctest's parameters that are not in D. Precisely, assign v'_d to each $p_d \in \hat{P} \cap P_D$, where $(p_d \mapsto v'_d) \in D$; then, assign v to each $p \in \hat{P} - P_D$, where $(p \mapsto v) \in C$. Ctests with $\hat{P} \cap P_D = \emptyset$ do not need to be run when testing D. A configuration diff, D, passes if all selected ctests pass and fails if any selected ctest fails.

Ctests for checking configuration files. A configuration file typically only assigns values to a subset of $\mathbb{P}$. Parameters whose values are not assigned in the configuration file receive their default values. So, ctests treat a configuration file as a diff which updates the default system configuration with the configuration values that are set in the file.

Locating offending configuration values. If a ctest is newly failing on a configuration diff, D, then the offending parameters must be in $\hat{P} \cap P_D$, unless the tests are flaky [8]. Parameters in D are typically very few, e.g., 49.5% of configuration changes have two-line revisions [55]. We discuss our experience on inspecting ctest failures in §7.

3.3 Creating a Ctest Infrastructure

Ctest infrastructure can be built on top of existing software testing frameworks. Specifically, a ctest can be run in the same way as a regular software test by instantiating the test's input parameters with system configuration values. We built our current ctest infrastructure on top of the Maven build system [36]—all the systems that we study use Maven to compile and run their test suites (§5.1). It should be straightforward to extend our infrastructure to support other build systems such as Gradle [16], Bazel [7], and Buck [11].

Ctests should be run in a hermetic test environment (a common software testing practice [41]). Ctests are best run in the same environmental setup as in production because ctests can capture environment-specific, configuration-induced failures (e.g., Figure 8). Our current infrastructure supports running ctests in Linux containers.

Ctest selection. Ctest selection is critical for utilizing ctests during continuous integration and deployment of configuration diffs. Regression test selection, which reruns tests that are affected by code changes [17], does not work for configuration changes. We build our ctest selection mechanism using the test selection criterion described in §3.2; it only runs ctests that are parameterized by parameters in D.

Configuration versioning. We store the latest version of the system configuration C to be updated after a configuration diff passes ctest and is deployed (§3.2). So, our infrastructure can instantiate ctests with updated parameter values in C.

4 Ctest Generation

Ctests can be generated by transforming existing tests in mature software projects with reasonable manual effort. The generated ctests inherit test logic and assertions from the original tests. The inherited assertions hold for *all* correct configuration values.

Ctest generation proceeds in two steps. First, the existing tests are *parameterized* by system configuration parameters, so that they can be run against different system configurations (§3.2). We describe in §4.1 how to parameterize an existing test t to obtain $\hat{t}(\mathbb{P})$ (or $\hat{t}$ in short), where $\mathbb{P}$ represents all the configuration parameters of the target system. Second, the parameterized tests are transformed into ctests.

A parameterized test $\hat{t}$ may not be directly usable as a ctest if the original test t contains test logic or oracles that assume specific configuration values. The resulting parameterized test $\hat{t}$ may fail incorrectly on valid configuration values if the subsequently resulting ctest is run against new values that are not the assumed values. So, if $\hat{t}$ assumes specific values of a configuration parameter $p \in \mathbb{P}$, $\hat{t}$ cannot be a ctest for p. But $\hat{t}$ can still be a ctest for another independent parameter, say $q \in \mathbb{P}$, if $\hat{t}$ does not assume a value for q. In short, if $\hat{t}$ assumes a value for p but not for q, $\hat{t}$ can result in a ctest for q but not for p. We address the challenge of identifying, among all configuration parameters exercised by $\hat{t}$, those that can be included in the input parameter set $\hat{P}$ of the resulting ctest $\hat{t}(\hat{P})$. In this example, $q \in \hat{P}$ and $p \notin \hat{P}$. We describe in §4.2 how to identify $\hat{P}$ from $\mathbb{P}$ when generating a ctest $\hat{t}(\hat{P})$ from $\hat{t}$.

One can optionally rewrite generated ctests to allow generated ctests check more configuration parameters or to generate

```
    static {
      ...
      addDefaultResource("core-default.xml");
      addDefaultResource("core-site.xml");
+     addDefaultResource("core-ctest.xml");
    }
    /* conf/Configuration.java */
```

Figure 3: Parameterization by intercepting the configuration APIs of Hadoop. After the interception, test code reads configuration values from `core-ctest.xml` which is managed by our ctest infrastructure. In this way, the test code can be instantiated with values in `core-ctest.xml`.

new ctests. §4.3 presents two simple rewriting rules for dealing with hardcoded parameter values and assertions.

In summary, given tests $T = \{t_i \mid i = 1, 2, ..., Nf\}$, we generate a set of ctests $\hat{T} = \{\hat{t}_i(\hat{P}_i)\}$, where $|\hat{T}| \le |T|$. For each ctest $\hat{t}_i(\hat{P}_i)$, $\hat{t}_i$ is the parameterized test and $\hat{P}_i$ is the set of configuration parameters that can be tested by the ctest. Each ctest is generated from an existing test and checks one or more parameters. To test to-be-deployed configurations, a ctest instantiates all its input parameters.

Developer effort. To generate ctests from existing tests, developers need to instrument the configuration APIs of the system. We discuss instrumentation in §4.1 and §4.2.1. After instrumentation, ctest generation is mechanized.

4.1 Parameterization

The first step in generating ctests is to parameterize an existing test t into $\hat{t}$. so that t can be run by instantiating the parameters with actual system configuration values. Parameterization requires changing test code to read configuration values at runtime, as provided by ctest infrastructure (§3.3), instead of from default configuration files or other test files. To generate large numbers of ctests, parameterization is automated.

We find that systematic parameterization can be done by intercepting the configuration APIs that existing tests use for reading configuration values. Figure 3 exemplifies our interception of Hadoop's configuration API. The idea is to overwrite configuration values as the final step of configuration loading. Thus, when the test code reads configuration values from configuration APIs, the values come from the configurations maintained by the ctest infrastructure (§3.3). Our parameterization approach minimizes the changes needed and avoids changing individual tests. Our approach is applicable to many (if not all) modern cloud systems, but its implementation is project-specific. We implemented parameterization for five cloud systems (§5.1) and validated its applicability to other systems including Spark and OpenStack.

The parameterization step produces a parameterized test, $\hat{t}(\mathbb{P})$, for each test t, where $\mathbb{P}$ is the set of all system configuration parameters. Parameterization is oblivious of the set of configuration parameters exercised by each $\hat{t}$; these are automatically identified in §4.2.1.

4.2 Transformation

A parameterized test $\hat{t}(\mathbb{P})$ may not be a valid ctest—a ctest's parameter set $\hat{P}$ should include only configuration parameters that can be checked by the ctest—the test logic and oracles should not assume specific parameter values.

Transforming a parameterized test into a ctest consists of (1) identifying the set of configuration parameters that are exercised by each test t, denoted as P (§4.2.1), and (2) for each $p \in P$, determining whether the test logic and oracle of t assume any specific value of p; if so, $p \notin \hat{P}$ (§4.2.2). Figure 4 shows ctest generation process that transforms from t to $\hat{t}(\hat{P})$.

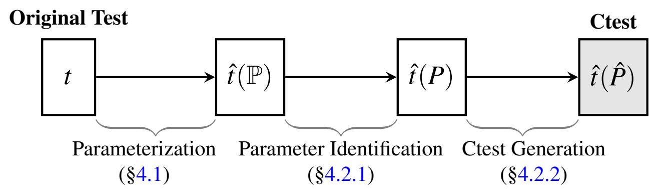


Figure 4: Steps in the ctest generation process.

4.2.1 Identifying Parameters Exercised in Tests

Static or dynamic analysis can be used to identify P for each test t. We implemented and experimented with both. Our static analysis taints the statements that can be reached by t and searches for configuration API usage (§4.1) among the tainted statements. It was straightforward to identify configuration API usages in test code. But, since test code commonly passes configuration values to system code initialization, it is hard to precisely collect the exact configuration API usage in system code that may be reachable from tests. So, static analysis often imprecisely produces a parameter set larger than P.

We chose dynamic analysis under the assumption that most test cases are relatively deterministic [15]. Our dynamic analysis requires developers to instrument configuration `GET` and `SET` APIs for reading and writing configuration values in the target system, respectively.[1] Our instrumentation inserts code to log the stack trace of each API invocation and the configuration parameter involved. Figure 5 is an example of our instrumentation for Hadoop. With instrumentation in place, our dynamic analysis runs all existing tests and post-processes the log for each test t to automatically identify (1) the set of configuration parameters P exercised by t, and (2) parameters written (via the `SET` API) in t (needed in §4.2.2).

Log processing is automated, as our instrumentation produces easily-parsed output. Our dynamic approach is simple,

[1]The `GET` and `SET` APIs are common configuration abstractions used in cloud systems written in Java and Python [33, 48, 67, 69]. `GET` APIs are of the form, "`<T> get(Class<T> class, String parameter)`"; they take a parameter name and return a value. `SET` APIs are of the form, "`void set(Class<T> class, String parameter, <T> value)`"; they reset the original value of the given parameter with the input `value`. Typically, `get` and `set` are declared in wrapper classes such as `java.util.Properties` for Java and `configparser` for Python projects.

```
public String get(String name) {
+   String ctestParam = name;
    String[] names = handleDeprecation(
        deprecationContext.get(), name);
    String value = null;
    for(String n : names) {
+     ctestParam = n;
      value = substituteVars(
          getProps().getProperty(n));
    }
+   LOG.warn("[CTEST][GET-API] " + ctestParam);
+   CTestUtils.printStackTrace(ctestParam);
    return value;
}
/* conf/Configuration.java */
```

Figure 5: Example Instrumentation for a GET API in HCommon. The get method is the lowest level API used by high-level GET APIs, e.g., getInt and getBool. handleDeprecation handles deprecated parameters. SET APIs are instrumented similarly.

general, and reliable, requiring modest instrumentation effort (§5.1). Instrumentation of configuration APIs is performed during ctest generation. Instrumentation is neither performed when running ctests nor added to the production system.

For completeness, we consider a test to *exercise* a parameter if the test uses the parameter's value as it executes. We do not exclude tests based on potential effectiveness for exposing misconfigurations or bugs. In general, such effectiveness is hard to define or model. Our decision is also justified because GET API invocations alone can expose misconfigurations (e.g., those due to type-casting errors [26]) or bugs (e.g., those caused by failing to trim white space [25]).

4.2.2 Generating Parameter Sets for Ctests

For each test $\hat{t}(P)$ that is parameterized after the steps in §4.1 and §4.2.1, our toolchain automatically generates the ctest $\hat{t}(\hat{P})$ by filtering out configuration parameters in $P - \hat{P}$:

Respecting intended configuration resets. If a test explicitly resets a configuration parameter to a specific value, then the test logic or its oracle depends on the new value. So, the test cannot be applied to other valid values of the configuration parameter. Our tool automatically identifies all configuration parameters whose values are reset in a test t. It does so by parsing the logs generated by instrumented SET APIs (§4.2.1) and excluding parameters that are reset from P. Note that we do not exclude configuration parameters from P that are reset in the system (not test) code. System code can reset configuration values in ways that should not impact ctests, e.g., during dynamic configuration tuning.

Detecting implicit assumptions on configuration values. In practice, not all parameter resets are performed using SET APIs. Some tests *implicitly* assume specific parameter values. Most tests with such implicit assumptions expect default parameter values and do not set them explicitly. If the default value is unchanged, then the tests pass. Although such assumptions constitute bad software engineering practice ("brittle assertions" in the literature [28]), we observe many such cases in the existing test code. Therefore, we automatically identify and exclude from $\hat{P}$ the parameters on which tests have implicit assumptions.

Our intuition is that, if a test assumes specific values, then it will fail on different but valid values. That is, if $p \in \hat{P}$, then $\hat{t}$ should pass on all valid values of p. So, a configuration parameter on which a test makes an implicit assumption can be identified by assigning a different valid value to the parameter and observing the outcome of the existing test.

Our implementation validates whether $\hat{t}$ makes implicit assumptions on each $p \in P$ by running $\hat{t}$ with p instantiated with a few valid values. If $\hat{t}$ fails on a valid value, then $\hat{t}$ makes an assumption on the value of p, i.e., $p \notin \hat{P}$. In our experience in generating thousands of ctests (§5), using up to three values for validation is sufficient to identify configuration parameters on which tests make implicit assumptions.

We use heuristics to automatically generate values for validation from the default value of each configuration parameter, based on the parameter types. For numeric types, we halve and double the original value. For Boolean values, we use the negation. For environment-related values (e.g., path, address, and port), we generate a similar but different value (e.g., a different port number). We use the regular expression described in [77] to infer parameter value types.

These heuristics are not sound; they do not guarantee the validity of generated values. However, the heuristics are simple and practical—only 1.6% of the generated values were invalid due to hidden constraints (§5.3). Our heuristics could not generate valid values for about 16% of parameters: enum options, class names, and commands. We manually selected valid values in these cases. Our future work includes integrating advanced inference tools [44, 47, 69] to infer valid values for these parameter types.

The validation yields $\hat{P}$ for each parameterized $\hat{t}$ transformed from t. If $\hat{P} \neq \emptyset$, $\hat{t}(\hat{P})$ is a ctest for all $p \in \hat{P}$.

4.3 Rewriting

In addition to the generated ctests, one can optionally manually rewrite an existing test to create a new ctest or rewrite a generated ctest to check more configuration parameters. We find two common patterns for rewriting, exemplified in Figure 6. First, many configuration resets in test code are used for setting up the test environment, e.g., a test file, address, port, etc. Those resets are not needed in ctests which are run with actual environment variables. Figure 6a shows this rewriting pattern. There, by simply removing the reset, the ctest can check alluxio.master.rpc.port's values. Note that removing hardcoded resets may require changing how the test reads the configuration values, if the test code does not use standard APIs (discussed in §5.4). Second, some assertions in the test code assume the default configuration values (§4.2.2) which can be safely removed or rewritten to the actual values

```
    @Ctest
    void testStartStopPrimary() {
-     conf.set("alluxio.master.rpc.port",
-         TEST_PORT);
      master = new AlluxioMasterProcess(conf);
      master.start();
      ...
    }
    /* master/AlluxioMasterProcessTest.java */
```

(a) Removing hardcoded resets. After removing the `conf.set()` call, the `alluxio.master.rpc.port` parameter's value comes from system configuration. The rewritten ctest can then test `alluxio.master.rpc.port`.

```
   @Ctest
   void testNameNodeXFrameOptionsEnabled() {
     ...
     header = conn.getHeaderField(
         "X-FRAME-OPTIONS");
     ...
     assertTrue(header.endsWith(
-        HttpServer.XFrameOption.SAMEORIGIN));
+        conf.getTrimmed("dfs.xframe.value")));
   }
/* namenode/TestNameNodeHttpServerXFrame.java */
```

(b) Rewriting hardcoded assertions. The rewritten ctest asserts on the actual value of the `dfs.xframe.value` parameter not its default value (`SAMEORIGIN`).

Figure 6: Two common patterns of test rewrites (§4.3).

being tested, as shown in Figure 6b. For both patterns, test code is rewritten to read values from the system configuration without changing the test logic.

5 Generating Thousands of Ctests

We share our experience in generating over 7900 ctests by transforming existing tests in five mature and widely-used open-source cloud systems: HCommon (Hadoop runtime and core utilities), HDFS, HBase, ZooKeeper, and Alluxio. We chose these projects for our evaluation (§6) because they are widely studied, their configuration APIs represent the state-of-the-art in modern cloud systems, and they expose many configuration parameters (Table 1). We discuss the feasibility of, and opportunities for, generating ctests in practice.

5.1 Evaluated Systems and their Test Suites

Table 1 shows the characteristics of the cloud systems that we studied: tests, configuration parameters, and how much instrumentation we performed.

Instrumentation effort. Our system-specific instrumentation is modest because each system uses a few classes to implement the configuration APIs. In the worst case, we changed only three classes each in ZooKeeper and Alluxio ("# Class" column in Table 1). It takes more lines of instrumentation for ZooKeeper than the others, because the `GET` and `SET` APIs

Software	Test Coverage		# Config.	Instrum.	
	Stmt Cov.	Meth Cov.	Params	LoC	# Class
HCommon (2.8.5)	73.1%	74.0%	269	24	1
HDFS (2.8.5)	80.3%	79.6%	296	24	2
HBase (2.2.2)	69.5%	80.1%	205	29	2
ZooKeeper (3.5.6)	75.8%	84.3%	32	130	3
Alluxio (2.1.0)	70.8%	72.6%	515	34	3

Table 1: Characteristics of studied systems (test suites, configuration parameters, and instrumentation efforts). The instrumentation includes both parameterization (§4.1) and logging (§4.2.1).

Software	Module	# Tests		# Config. Param.	
		Total	Using Config.	Total	Used in Tests
HCommon	hadoop-common	3268	1923 (58.8%)	269	232 (86.2%)
HDFS	hadoop-hdfs	3957	3293 (83.2%)	296	284 (95.9%)
HBase	hbase-server	2630	2035 (77.4%)	205	169 (82.4%)
ZooKeeper	zookeeper-server	881	180 (20.4%)	32	32 (100.0%)
Alluxio	core	1648	1117 (67.8%)	515	423 (82.1%)

Table 2: Characteristics of configuration parameters exercised in software tests of the studied systems.

are implemented per configuration parameter;[2] the other four systems implement generic APIs as exemplified in Figure 5.

Test suites. The studied systems all have a good number of tests, mostly at the unit- and integration-test levels. System-level tests are rare, reflecting a common testing practice in modern systems engineering [60]. Further, all five projects enforce rigorous code commit policies that require every code change to be covered by a test. Code coverage is high ("Test Coverage" in Table 1), with at least 70% statement coverage. Proprietary systems report even higher test coverage [29, 51].

We focus on the core modules of the studied systems ("Module" in Table 2), which are likely to be used in production. In the rest of this paper, we only use the tests in the studied modules, even though tests in the other modules can also be leveraged during ctest generation.

Table 2 shows the percentage of existing tests per module that exercise configuration values ("Using Config.") and the percentage of configuration parameters exercised by tests ("Used in Tests"). We collected these percentages from instrumented configuration API logs (see §4.2.1). Clearly, many tests exercise configuration values and are candidates that can be transformed into ctests. Furthermore, 82.1%–100.0% of configuration parameters across the studied systems are exercised by existing tests. So, most configuration parameters have a chance of being checked by a generated ctest (§5.2).

5.2 Ctest Generation Results

We apply the automated approach in §4.2 to generate ctests from the existing tests in the evaluated systems. We select all 32 configuration parameters in ZooKeeper. For the other

[2]We are helping ZooKeeper to improve their APIs (e.g., [81]); using ZooKeeper shows applicability of ctests across configuration APIs.

Software	Existing Tests			Generated Ctests	
	# Param.	Tests	→	Ctests	#Param. (Cov.)
HCommon	90	1870	→	1846 (98.7%)	90 (100.0%)
HDFS	90	3191	→	3148 (98.7%)	90 (100.0%)
HBase	90	1909	→	1687 (88.4%)	90 (100.0%)
ZooKeeper	32	180	→	176 (97.8%)	32 (100.0%)
Alluxio	90	1117	→	1117 (100.0%)	90 (100.0%)

Table 3: Ctest generation results. The results include only generated ctests (§4.2). The generated ctests have 100% coverage of the configuration parameters.

systems, we randomly select 90 configuration parameters that are exercised by the tests ("Used in Tests" in Table 2). Note that we sampled 90 parameters mainly to bound our manual inspection effort for analyzing effectiveness and false negatives (Tables 8 and 9). The generation process is mostly automated after API instrumentation.

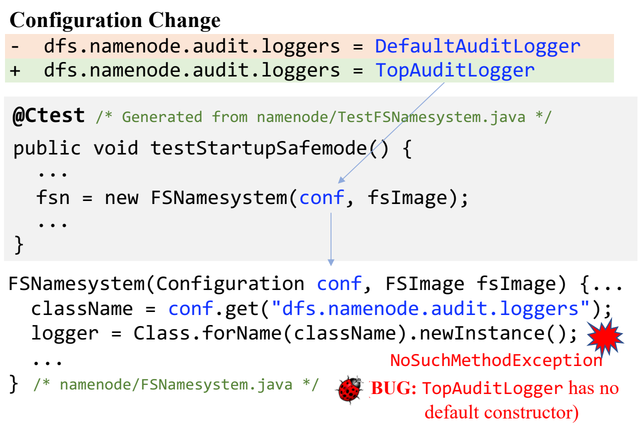
Table 3 shows ctest generation results. Overall, 88.4%–100% of existing tests that exercise the selected configuration parameters were successfully transformed into ctests. Furthermore, the generated ctests cover 100% of the selected parameters, i.e., each parameter is checked by at least one ctest. The small percentage of tests that could not be transformed as ctests were hardcoded to specific values of *all* the parameters that they exercise—a ctest is generated as long as it can check at least one configuration parameter. Section 6 discusses the effectiveness of the generated ctests for detecting failure-inducing configurations in different settings.

5.3 Detecting Bugs and Hidden Constraints

Some *valid* configuration values caused ctests (§4.2.2) to unexpectedly throw runtime exceptions instead of the failed assertions that are typical manifestations of hardcoded tests. We analyze these exceptions and find, surprisingly, that most are caused by (1) previously unknown bugs in the code exposed by configuration changes, or (2) hidden constraints which made seemingly valid configuration values erroneous. We include these ctests which are effective in capturing bugs and misconfigurations in our evaluation.

Dormant bugs exposed by configuration changes. We find 14 previously unknown bugs in the latest versions of the five evaluated systems. 12 of those bugs are confirmed and 10 were fixed by the developers after we reported them; 9 bugs are considered "Major" or "Critical". Real users encountered a bug after we reported it [24]. 12 of the 14 bugs existed for more than five years in these projects that routinely run static analyses and perform testing. Figure 7 shows one of these bugs, in which changing the value of the parameter to a valid option `TopAuditLogger` will crash the NameNode of HDFS because a default constructor is required but not implemented.

Hidden configuration constraints. We also discovered 11 hidden constraints that cause the generated values to result in errors. We say these constraints are "hidden" because they



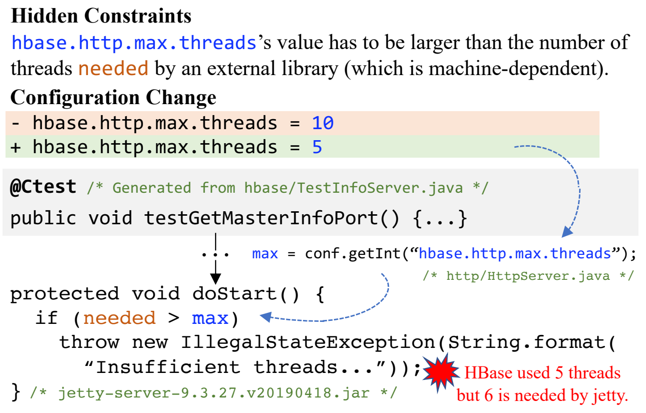
Configuration Change

```
-  dfs.namenode.audit.loggers = DefaultAuditLogger
+  dfs.namenode.audit.loggers = TopAuditLogger
```

```
@Ctest /* Generated from namenode/TestFSNamesystem.java */
public void testStartupSafemode() {
  ...
  fsn = new FSNamesystem(conf, fsImage);
  ...
}
```

```
FSNamesystem(Configuration conf, FSImage fsImage) {...
  className = conf.get("dfs.namenode.audit.loggers");
  logger = Class.forName(className).newInstance();
  ...
} /* namenode/FSNamesystem.java */
```

Figure 7: A new bug that was exposed by a *valid* configuration change and was captured by a ctest [23]. The bug crashes HDFS NameNode due to missing a default constructor. The bug has been fixed after we reported it.



Hidden Constraints

`hbase.http.max.threads`'s value has to be larger than the number of threads `needed` by an external library (which is machine-dependent).

Configuration Change

```
- hbase.http.max.threads = 10
+ hbase.http.max.threads = 5
```

```
@Ctest /* Generated from hbase/TestInfoServer.java */
public void testGetMasterInfoPort() {...}
```

```
  ...   max = conf.getInt("hbase.http.max.threads");
                        /* http/HttpServer.java */
protected void doStart() {
  if (needed > max)
    throw new IllegalStateException(String.format(
      "Insufficient threads..."));
} /* jetty-server-9.3.27.v20190418.jar */
```

Figure 8: A hidden constraint exposed by a ctest. The configuration of HBase is constrained by an external library (Jetty).

were not documented and are not intuitive to discover. Figure 8 is an example of a hidden constraint—the configuration parameter of HBase is constrained by an external library, Jetty. Any configuration value that is smaller than the `needed` variable's value in Jetty will cause a runtime exception.

5.4 Rewriting Ctests

We further study the intended configuration resets in test code (§4.2.2) to understand the opportunities and challenges of rewriting tests. We focus on environment-related configuration parameters—as discussed in §4.3, tests often reset configuration values to set up test environments, which are not needed by ctests. For this study, we selected 44 configuration parameters with hardcoded environment settings, including all four from ZooKeeper and 10 from the other four systems. There are altogether 263 tests that reset at least one of the 44 parameters; 233 of these tests were transformed to generate ctests but those ctests cannot check the reset parameters. The 233 generated ctests cover all 44 parameters (Table 3).

We manually applied the two test rewriting rules described in Figure 6 to these 263 tests. Removing hardcoded resets alone (Figure 6a) can enhance 86 tests for ctests to cover 8 parameters. Further, by removing or rewriting hardcoded assertions (Figure 6b), we can enhance 16 more tests. In total, the two test rewriting rules can cover 102 (38.8%) tests for 18 out of 44 parameters. The remaining tests either cannot benefit from rewriting, or require significant changes beyond the two simple patterns in Figure 6.

The test rewriting effort was small in HCommon, HDFS, HBase, and Alluxio for which we rewrote 33 tests for 16 parameters using a total of 90 changed lines. Rewriting a test in these four systems takes only two or three lines of test code (Figure 6). The rewriting effort was much larger in ZooKeeper, mainly because ZooKeeper does not utilize similar configuration APIs (§5.1) as other systems—the test code does not use SET APIs to reset the parameter value as in Figure 6a. So, we wrote a new API to load actual configuration values into the tests; our implementation has 14 lines of code. With our new API, we were able to rewrite 69 tests for two parameters, which takes a total of 103 changed lines.

6 Evaluation of Ctest Effectiveness

We used three experimental settings to extensively evaluate ctests' effectiveness for testing configurations in context:

1. real-world configuration-induced failures documented in issue tracking databases;
2. diverse injected misconfigurations for configuration parameters that have different value types and semantics;
3. non-default configuration files collected from Docker images hosted at DockerHub [14].

6.1 Evaluating Ctests on Real-world Failures

We evaluate the effectiveness of ctests for detecting failure-inducing configurations that caused real-world failures. Our goal is to see how many of these failures ctests could have been detected earlier.

Configuration-induced failures used. We reproduced 64 real-world configuration-induced failures from the issue-tracking database of the five systems (Table 4). Each failure was reported by real system users and was caused by a configuration change (i.e., a value different from the default was used). These 64 failures have diverse root causes, including 51 misconfigurations and 13 software bugs exposed by *valid* configuration changes.[3] We collected failures from issue-tracking systems instead of user forums or mailing lists because: (1) failures recorded in issue-tracking databases tend to have had large impact, and (2) issue-tracking databases rigorously record the version of the systems on which the failures were reported, which is critical for reproducing failures. Importantly, we only generate ctests from the tests in the reported version, *not* from tests in later versions.

[3]For seven failures, misconfigurations triggered bugs in the code. We categorize them as "misconfigurations" in Table 4.

Software	Misconfigs	Bugs (Valid Configs)	Total
HCommon	11 (84.6%)	2 (15.4%)	13
HDFS	21 (95.5%)	1 (4.5%)	22
HBase	8 (61.5%)	5 (38.5%)	13
ZooKeeper	8 (66.7%)	4 (33.3%)	12
Alluxio	3 (75.0%)	1 (25.0%)	4
Total	51 (76.9%)	13 (20.3%)	64

Table 4: Statistics on real-world configuration-induced failures from issue-tracking databases used in ctest evaluation.

Root Cause	# Failures	# (%) Detected by Ctests	
		Gen Only	Gen + Rewrite
Misconfigurations	51	41 (80.4%)	51 (100.0%)
⊢ Corrupt config files	3	3 (100.0%)	3 (100.0%)
⊢ Value type errors	3	3 (100.0%)	3 (100.0%)
⊢ Out-of-range values	12	11 (91.7%)	12 (100.0%)
⊢ Value semantic errors	22	16 (72.7%)	22 (100.0%)
⊢ Dependency violations	10	7 (70.0%)	10 (100.0%)
⊢ Resource violations	1	1 (100.0%)	1 (100.0%)
Bugs exposed by valid config	13	10 (76.9%)	11 (84.6%)
Total	64	51 (79.7%)	62 (96.9%)

Table 5: Ctest effectiveness in detecting real-world configuration-induced failures of various root-cause types. Most types are self-explanatory; value semantic errors refer to misconfigurations that violate the semantics of the configuration parameter, including invalid file paths, URI, IP addresses, permission masks, etc.

Ctests evaluated. For each failure, we identify each configuration parameter p_i and its value v_i in the failure-inducing configuration change (13 of 64 failures involve more than one configuration parameter). We then generate ctests using the method in §4 for p_i. Further, we apply the two rewriting rules in §5.4 to enhance 11 generated ctests.

6.1.1 Effectiveness

Table 5 shows the effectiveness of ctests in detecting the 64 real-world failures and the root causes of those failures.

The results are promising. 96.9% (62/64) of the failure-inducing configurations are detected by ctests. *All* failures due to misconfigurations are detected. Specifically, 79.7% (51/64) of all failures are detected by using only generated ctests; the other 17.2% (11/64) require rewriting of ctests (§5.4). In 9 of the 11 failures that require rewriting, we only remove unnecessary value resets (like in Figure 6a). In the other two, we also change an assertion (like in Figure 6b). The results show that existing tests contain effective test logic and oracles needed to expose failure-inducing configuration changes. By leveraging those test logic/oracles, ctests can effectively detect failure-inducing configuration changes and prevent them from being deployed to production.

Failure Mode	**Count (Pct)**
Unexpected runtime exceptions	31 (50.0%)
Exceptions thrown by configuration-checking code	27 (43.5%)
Failing assertions in ctest code	3 (4.8%)
Test timeout (the system hangs)	1 (1.6%)

Table 6: Failure modes of ctests when detecting the failures.

By checking the behavior of code that exercise configuration parameters, ctests have generic ability to detect diverse types of misconfigurations, as well as bugs exposed by valid configuration changes (Table 5). That is, ctests are not designed to detect specific types of misconfigurations or bugs. We exemplified failures detected by ctests in Figures 1 and 2. Table 6 shows the failure modes of ctests on the 62 detected configuration-induced failures. Most failures manifested as unexpected runtime exceptions (division by zero, array index out of bound exceptions, etc.) or exceptions thrown by configuration-checking code. We show examples in Figures 2 and 7. Both types of exceptions would have the same impact on production systems if the failure-inducing changes were deployed. In three failures, test assertions fail because of unexpected behavior. The last failure was a test timeout that occurred because the configuration change caused the system to hang (similar to Figure 9a).

Two of the 64 failures were not detected by ctests [2,80]. In ALLUXIO-9810 [2], the root cause is a buggy shell script that no test invoked. The root cause of ZOOKEEPER-2299 [80] is a bug in a method that no test in the reported ZooKeeper version exercised. Both bugs can be detected by extending the test suite. In fact, for ZOOKEEPER-2299, the latest ZooKeeper version includes a test from which we have now generated a ctest that detects this bug.

6.1.2 Comparison with State-of-the-Art Techniques

Table 7 compares ctests with two state-of-the-art configuration checking techniques, PCheck [67] and Spellcheck [48]. Both PCheck and Spellcheck are designed for cloud systems and do not require additional training data or rule sets.

None of the 13 failures induced by *valid* configuration changes triggering bugs in code can be detected by existing configuration validation or automatic misconfiguration detection techniques, because those techniques only check whether configuration values are valid.

Ctests detected *all* misconfigurations among the real-world failures, including many that are challenging for state-of-the-art checking and detection techniques to detect. Spellcheck only detects value-type errors. In our real-world configuration-induced failure dataset (Table 5), only three failures were caused by value-type errors.

The following misconfigurations detected by ctests cannot be detected by PCheck: (1) two misconfigurations leading to non-crashing behavior (e.g., Figure 1), (2) five misconfigurations involving operations that have side effects (e.g.,

	# Failures	**Spellcheck**	**PCheck**	**Ctest**	
				Gen Only	**Gen+Rewrite**
Misconfigs	51	3	41	41	51
Bugs	13	0	0	10	11
Total	64	3	41	51	62

Table 7: A comparison of Ctests, PCheck, and Spellcheck in detecting misconfigurations and bugs exposed by valid configuration changes (Table 5). We assume sound PCheck and Spellcheck static analyses—these are upper bounds for PCheck and Spellcheck.

writing files), and (3) three misconfigurations that require client-side interactions to expose. Note that PCheck performs *post-deployment* configuration validation; PCheck does not run tests but instruments deployed systems. Ctests detect misconfigurations early, during *pre-deployment* testing. PCheck has two limitations that ctests do not have: (1) PCheck cannot have side effects in the production environment, and (2) PCheck cannot deal with external dependencies and events (e.g., client operations) [67]. Moreover, unlike PCheck, ctests can find bugs resulting from valid configuration changes.

6.2 Evaluating Ctests on Diverse Misconfigurations

We ran ctests on injected misconfigurations to (1) systematically evaluate ctests' effectiveness on many diverse configuration parameters with different value types and semantics, and (2) experimentally evaluate ctests on misconfigurations that were not in the failures from issue-tracking databases.

Injected misconfigurations. We generate up to three erroneous values for each of the 392 configuration parameters in §5. We use the misconfiguration generation rules proposed for misconfiguration injection testing [31,32,69]. But we exclude rules such as case alternation and random fuzzing, which lead to many false errors. Note that the misconfiguration generation rules are different from the heuristics for generating *valid* values in §4.2.2. Specifically, we generate misconfigurations based on the types and semantics of each configuration parameter. For Boolean or enum types, we generate invalid options. For numeric types, we generate values containing alphabetic characters, and out-of-range values (smaller/larger than the min/max value). For parameters without explicit data ranges specified in the configuration file, we use the range of their data type, e.g., `INT_MAX` as the maximum value of integers. For strings, erroneous values are generated based on the semantics of the parameter. We follow the fine-grained rules defined in [32,69]. For example, for file-path parameters, we generate non-existent files, incorrect file content, and incorrect file types. We reviewed each generated erroneous value to reduce false errors.

Ctests evaluated. We use the generated ctests from §5. For each erroneous value e generated for p, we create a configuration diff $D_e = \{(p \mapsto e)\}$ that sets p's value to e. We run all

Software	Complete	Partial	None	N/A
HCommon	43 (48.3%)	22 (24.7%)	24 (27.0%)	1
HDFS	67 (77.9%)	12 (14.0%)	7 (8.1%)	4
HBase	52 (61.9%)	23 (27.4%)	9 (10.7%)	6
ZooKeeper	20 (90.9%)	2 (9.1%)	0 (0.0%)	10
Alluxio	43 (47.8%)	15 (16.7%)	32 (35.6%)	0

Table 8: Ctest effectiveness in detecting injected misconfigurations *per parameter*. "Complete", "Partial", and "None" refer to number of parameters with all, some (but not all), and none of the injected misconfigurations detected, respectively. "N/A" refers to the number of parameters in which all the generated misconfigurations turned out to be valid due to the imprecision of error generation.

Misconfiguration

```
hadoop.security.random.device.file.path = INVALID_RANDEV
```

The random device is used by the object to get random bytes.

```
@Ctest /* Generated from TestOsSecureRandom.java */
public void testRandomBytes() {
  ...
  OsSecureRandom rand = new OsSecureRandom(conf);
  // checkRandomBytes will timeout if secure random
  // implementation always returns a constant value
  checkRandomBytes(rand, ...); RuntimeException (not readable file)
                               TimeoutException (not rand device)
}
```

(a) Invalid file content. The ctest detects the misconfigurations by testing the functionality of the random device.

Misconfiguration

```
hbase.regionserver.hlog.reader.impl = ProtobufLogReader
hbase.regionserver.hlog.writer.impl = SecureProtobufLogWriter

@Ctest /* Generated from wal/AbstractTestProtobufLog.java */
public void testWALTrailer() {
  ...
  // Appends entries in the WAL and reads it.
  doRead(...);  IOException (the log written by the hlog writer
                cannot be read by the hlog reader on the region server)
}
```

The misconfiguration is latent (causing runtime exception) and undocumented.

(b) Non-interoperability (undocumented [22]). The ctest detects the misconfigurations by testing the reader and writer together.

Figure 9: Non-trivial misconfigurations detected by ctests.

the ctests for p on each D_e and check whether any ctest fails on e. Unlike in §6.1, we do not rewrite ctests in this evaluation due to the larger size of experiments. So, our effectiveness results are a lower bound.

6.2.1 Effectiveness on Injected Misconfigurations

Table 8 shows the effectiveness of ctests in detecting the injected misconfigurations. Ctests detect *all* injected errors for 47.8%–90.9% of parameters and *at least one* injected error for 64.4%–100% of the parameters across the five systems.

Figure 9 shows two non-trivial misconfigurations detected by ctests. In Figure 9a, a ctest detects an invalid random device file path in HCommon by using the referred device to generate random bytes. Very few existing misconfiguration detection tools check file content; they mostly just check file paths and metadata. In Figure 9b, most `reader` and `writer` implementations of HBase are interoperable, but a few are not. Ctests checked the interoperability of a specific (`reader`, `writer`) pair and detected this non-trivial misconfiguration. The non-interoperability was neither documented nor checked in the system code before we reported it [22]. Using the non-interoperability configurations will fail HBase region servers.

Software	No Observable Symptom		Test Inadequacy	
	Correction	Mask	No Exposure	No Oracle
HCommon	14 (14.0%)	10 (10.0%)	56 (56.0%)	20 (20.0%)
HDFS	4 (11.8%)	8 (23.5 %)	9 (26.5%)	13 (38.2%)
HBase	25 (46.3%)	8 (14.8%)	19 (35.2%)	2 (3.7%)
ZooKeeper	2 (100.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
Alluxio	3 (2.7%)	0 (0.0%)	100 (90.9%)	7 (6.4%)
Total	48 (16.0%)	26 (8.7%)	184 (61.3%)	42 (14.0%)

Table 9: Root causes of false negatives among the injected misconfiguration *values*.

```
if (snapRetainCount < 3) {
  LOG.warn(
    "Invalid autopurge.snapRetainCount: "
    + snapRetainCount + ". Defaulting to 3");
  snapRetainCount = 3;
}
/* quorum/QuorumPeerConfig.java */
```

(a) An example of error correcting code in ZooKeeper

```
try { ...
  paths = conf.get("dfs.datanode.shared.file.
      descriptor.paths")
  fdFac = FileDescFactory.create(..., paths);
  ...
} catch (IOException e) {
  LOG.debug(
      "Disabling ShortCircuitRegistry", e);
}
/* datanode/ShortCircuitRegistry.java */
```

(b) An example of partial-failure masking in HDFS

Figure 10: Two patterns that lead to false negatives during misconfiguration injection.

The generated ctests failed to detect 28.4% (300 of 1055) injected misconfigurations, i.e., false negatives. The results are consistent with the evaluation of misconfigurations without rewriting in §6.1. Recall that we do not rewrite tests in this evaluation, which could improve ctest adequacy (§6.1).

We inspected the 300 false negatives. Table 9 shows root causes of false negatives and their distribution. 75.3% of false negatives are due to inadequacy of ctests that either does not expose the effects of the misconfigurations or does not have oracles to check the effects. Many of these effects are non-functional (e.g., performance issues). Moreover, unlike real-world failures (§6.1), many injected misconfigurations are expected to be uncommon in practice. So, the systems have no error-checking logic or test code. For example, in HDFS, negative `io.seqfile.compress.blocksize` values cause se-

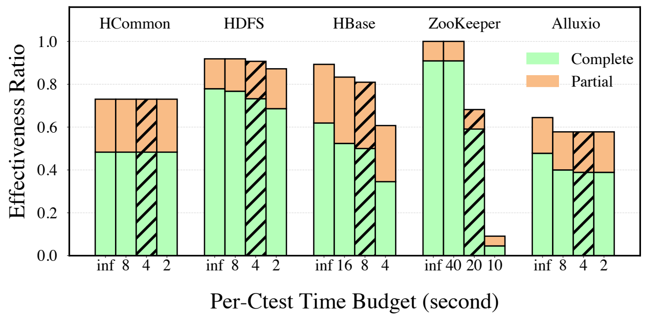


Figure 11: Time-budget analysis results. The results show the effectiveness of detecting misconfigurations if only ctests that finish under each time budget are run. The `inf` budget is equivalent to times from Table 8 where "Complete" and "Partial" are defined. We use the shaded budgets for experiments in §6.3.

vere performance degradation: every `append` triggers data compression. However, HDFS does not check against negative values nor have a test with performance-based oracles.

The remaining 24.7% of false negatives have no observable effects because of the presence of error-correcting code (e.g., Figure 10a), or because the consequences were masked (e.g., Figure 10b) by the system. Ctests cannot detect misconfigurations that have no observable effects.

6.2.2 Time-Budget Analysis

The per-parameter evaluation enables us to analyze the tradeoff between effectiveness and running time of ctests. To analyze this tradeoff, we performed a time-budget analysis. Our time-budget analysis excludes ctests that do not finish under a specified time budget and measures the effectiveness of the remaining ctests for detecting misconfigurations. We have not yet designed a test prioritization [45,72] scheme for ctests (§7), so we use per-test budgets (the amount of time *each* ctest is allowed to run) rather than a total-test-time budget (the amount of time *all* ctests are allowed to run). Per-test time budgets are well suited to test-suite parallelization, where each test is run in a separate process. Ctests for time-budget analysis run on an 8-core Intel i7-9700 CPU with 32 GB memory and Ubuntu 18.04.

Figure 11 shows the results of time-budget analysis. We observe that different budget ranges are needed for different systems given their different test characteristics. For example, ZooKeeper does not have many unit tests but relies mostly on integration tests. So, ZooKeeper needs larger per-test time budgets than other systems. Further, all ctests in HCommon finish under two seconds, so there is no decline in effectiveness across the time budgets shown. The key result from Figure 11 is that smaller per-test budgets can still achieve similar levels of effectiveness as running all the ctests for all projects except for Zookeeper. We use the shaded budgets in Figure 11 for evaluating ctests on real-world configuration files in §6.3.2 because they achieve good time-effectiveness tradeoff. We use a minimal per-test budget of 4 seconds to leave room for performance variability.

Software	# Files Tested	# Files that Fail Ctests		# False Alarms
		Version/Env.	Misconfig.	
HCommon	20	16	4	0
HDFS	20	15	2	0
HBase	20	12	0	0
ZooKeeper	20	14	0	0
Alluxio	12	3	1	0

Table 10: Results of running ctests on real-world configuration files collected from Docker images.

6.3 Evaluating Ctests on Configuration Files

We evaluate the effectiveness of ctests using configuration files collected from public Docker images. The experiments also enable us to measure the false positives and overhead of ctests on real-world configuration files (these are hard to systematically evaluate in §6.1 and §6.2).

Evaluated configuration files. We extract 92 configuration files from Docker images hosted on DockerHub [14] using the method described in [68]. We randomly sample 20 Docker images from the most popular 300 image repositories on DockerHub for the five systems. We only find 12 Alluxio image repositories that use non-default configuration files on DockerHub. We use the most recent image in each repository. The average number of configuration parameters in these files is 5.8 (the minimum is one and the maximum is 29).

Ctests evaluated. We generate ctests using the method in §4. For each configuration file f, we create a diff $D_f = \{(p \mapsto v_f)\}$ for all v_f explicitly set in f and run all the ctests that cover at least one parameter in D_f (see §3.2). We use the selected per-test budget from §6.2.2 to run ctests on each configuration file. We run ctests against the configuration files on our server, rather than deploying the ctest infrastructure in each image's container to reduce the cost of resolving dependencies and setting up environments (many images are built from old OS distributions with incompatible dependencies).

6.3.1 Ctests Effectiveness on Configuration Files

Table 10 presents the effectiveness of ctests on real-world configuration files. Surprisingly, many configuration files did not pass the ctests. We inspected all failed ctests and found 85 of 537 values to be erroneous. 76 of 85 erroneous values are correct in the container, but fail ctests because (1) certain files, IP addresses and ports in the containers do not exist or are not available on our server, and (2) the ctests are generated from tests from a newer version of the system—the configuration values in the images are no longer correct. We reported one such case, ALLUXIO-3402 [1], where the configuration parameter `alluxio.user.file.metadata.load.type` has the value "*Always*" in an image, `scality/alluxio`. But, in the latest Alluxio, an all-capitalized parameter value is required.

Software	Ctests with Budget		# All Ctests		
	# Ctests	Runtime	# Ctests	Runtime	Baseline
HCommon	1014.20	1.90	1019.55	3.42	3.84
HDFS	1850.75	37.48	2310.20	126.35	120.43
HBase	842.75	47.70	1053.65	99.47	140.86
ZooKeeper	39.55	6.79	76.95	26.29	19.98
Alluxio	796.5	1.66	796.5	1.66	1.44

Table 11: The number of ctests and their running time (in minutes) per configuration file using all ctests and ctests within time budget (selected in §6.2.2). The numbers are averaged over all evaluated files. "Baseline" is the time for running the corresponding original tests (not ctests).

So a ctest generated for the latest Alluxio fails. These results show that ctests effectively detect misconfigurations caused by version and environment changes [76].

Ctests also detect 9 misconfigurations of various types in seven configuration files (Table 10), including malformed files, value-type errors, and dependency violations, which are misconfigurations in the native container. Based on our inspection on the seven Docker images, we suspect that some of these configuration files may be managed by custom scripts that overwrite those files. Unfortunately, we find no documentation for five of the seven Docker images on DockerHub.

Zero false positives found. We expected a few false positives due to tests that assume some values but were not identified when generating parameter sets for ctests—the heuristics for generating valid values for validation are unsound (§4.2.2). However, we found no false positives (Table 10).

6.3.2 Ctest Running Time on Configuration Files

We measure the ctest-running time per configuration file. Table 11 shows the average total ctest-running time per configuration file when running ctests that finish within the per-test time budgets selected in §6.2.2. HCommon, ZooKeeper and Alluxio take less than ten minutes. HDFS and HBase have longer-running tests and take few tens of minutes.

We run all ctests with the `inf` budget and compare it with running the ctests under the time budget. There is no difference in the effectiveness of ctests, showing that the budgets are sufficient. We also compare total running time of all ctests with a baseline total time for running all the original software tests from which the ctests are generated. The results show that the running times of the ctests are similar to those of the original software tests ("Baseline" in Table 11). The running time for HBase is about 70% of its baseline because many tests in HBase aborted the execution and failed quickly due to the exceptions triggered by misconfigurations.

7 Discussion and Limitations

There is no silver bullet against configuration-induced failures. Ctests offer a simple, effective way to detect failure-inducing configurations, and are complementary to existing techniques.

The effectiveness of generated ctests depends on the adequacy of the original tests. On the evaluated systems, which have abundant tests, ctests outperform state-of-the-art tools. However, ctests cannot be generated if there are no existing tests, which is why no ctest exposed the two bugs in the evaluation (§6.1.1). Mature software systems will likely benefit from ctests because they have comprehensive test suites [29, 51]. For newer projects or projects with less comprehensive test suites, the generation of ctests could be limited. Note that the concept of ctests is not limited by the generation method discussed in §4. Ctests can be implemented by developers, just like they implement regular software tests.

Ctests cannot localize the root causes of configuration-induced failures. Based on our analysis of ctest results (§6.2 and §6.3), root cause localization can usually be done efficiently by analyzing the stack traces. However, a few failures take considerable time to understand, due to (1) complexity of configuration value propagation and transformation, or (2) unexpected, hidden configuration constraints (e.g., Figure 8). Fault localization [64] for configuration-induced failures can be developed to automate root cause analysis.

Ctests can increase the cost of regression testing, which is already expensive. Section 6.3.2 shows that running ctests for the evaluated systems takes reasonable time. On the other hand, we believe that the cost of running ctests can be significantly reduced by developing ctest reduction, prioritization and minimization techniques, as was done for regression testing [72]. One direction is to analyze ctest code and to understand how each ctest exercises configuration changes, towards reducing and prioritizing ctests. Ctests running time can also be further reduced by running ctests in parallel.

The ctest generation described in §4 is neither sound nor complete. First, the heuristics for detecting implicit test assumptions (§4.2.2) are unsound and could lead to false negatives in detecting bugs. Our heuristics minimize false positives. Second, dynamically tracing test executions to identify parameters exercised in tests (§4.2.1) is incomplete, because configuration changes could lead to different execution paths. Like any other form of testing, we do not claim completeness.

Like any other pre-deployment testing, ctests are fundamentally limited by a possible mismatch between the test environment and the production environment. Such a mismatch could lead to both false positives and false negatives.

8 Related Work

The severity and prevalence of configuration-induced failures [18, 19, 34, 35, 40, 42, 55, 74] has resulted in novel techniques for misconfiguration troubleshooting and debugging [3, 4, 46, 61–63, 73]. Advanced techniques have also been developed for diagnosing production failures [10, 13, 30, 79]. Ctests proactively detect failure-inducing configuration changes to *prevent* production failures in the first place.

Ctests is complementary to our prior work, PCheck [67]. We compared ctests with PCheck [67] in §6.1.2, despite

PCheck being a *post-deployment* technique. Note that PCheck can only detect misconfigurations, because it considers only statements on the data-flow path of each configuration value. Differently, ctests can detect valid configuration changes that expose bugs in the code, a common type of failure-inducing configuration changes [55]. Techniques designed for pre- and post-deployment have fundamentally different opportunities and challenges. In our experience, it is difficult (if not impossible) for auto-generated checking code to deal with many sophisticated real-world misconfigurations. This was the main motivation behind ctests which can exercise code and configurations together. But post-deployment techniques such as PCheck do not have problems caused by the mismatches between the test and the production environments.

We mentioned in §3.2 that ctests are complementary to configuration validation and misconfiguration detection [6, 27, 38, 43, 44, 49, 50, 55, 59, 61, 62, 66, 67, 75, 77], similar to how software testing complements static bug detection tools. Ctests can detect failure-inducing configuration changes that are challenging for existing techniques to detect, e.g., valid configuration changes that expose bugs. Most automated detection techniques only focus on specific types of misconfigurations. For example, Rex [38] detects dependency violations between source-code files and configuration files which should be updated together. Ctests are not specific to any type of misconfigurations or softwre bugs—they detect failure-inducing configuration changes based on the resulting program behavior. A common class of existing validation/detection techniques requires validation rules or training data that either do not exist (e.g., for systems that we evaluate) or are not available (we found no rule sets or training data online). In contrast, ctests do not rely on external rule sets or training data—they leverage existing abundant test cases.

Ctests complement software and system testing. In essence, ctests enhance existing testing techniques to focus on the actual configurations in production or configurations to be deployed, given that testing all possible configurations is infeasible. A ctest is a parameterized test. But ctests differ from traditional parameterized unit tests (PUTs) [57, 58] in goal, parameter source, and generation method. The goal of PUTs is to allow developers rerun the same test against different inputs, to cover more program paths. The goal of ctests is to connect production system configurations to software tests, to find failure-inducing configuration changes. The inputs to PUTs are either specified by developers or automatically generated by symbolic execution, but the inputs to ctests are read from the system configuration files or diffs.

9 Conclusion

This paper proposes ctests to connect software testing with production system configurations to enable detecting failure-inducing configurations during testing. We present how to generate ctests from existing software tests that are abundant in mature cloud systems. We show that ctests are effective in detecting real-world failure-inducing configurations, including both misconfigurations and dormant software bugs exposed by valid configuration changes. Our goal of ctests is to make testing of configuration changes a key component of configuration management and fill the missing piece in the practice of treating configuration as code. We have made all the code and datasets available at: https://github.com/xlab-uiuc/openctest.

Acknowledgement

We thank the anonymous reviewers and our shepherd, Haryadi Gunawi, for their insightful comments and feedback. We thank Darko Marinov, Lalith Suresh, Neil Zhao, Madhusudan Parthasarathy, Yongle Zhang, David Chou, and Justin Meza for the invaluable discussions. We thank Qingrong Chen, Andrew Yoo, Angello Astorga, Liia Butler, and Jonathan Osei-Owusu for helping proofread the paper. This work was funded in part by CCF-1816615, CCF-2029049, CNF-1956007, CCF-2019277, a Facebook Distributed Systems Research award, Microsoft Azure credits, and Google Cloud credits.

References

[1] ALLUXIO-3402. Backward compatibility for enum-typed configuration. https://alluxio.atlassian.net/browse/ALLUXIO-3402, 2020.

[2] ALLUXIO GITHUB ISSUE #9810. Alluxio worker fails to start when using multiple storage media in single tier on EMR. https://github.com/Alluxio/alluxio/issues/9810, 2019.

[3] ATTARIYAN, M., CHOW, M., AND FLINN, J. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)* (October 2012).

[4] ATTARIYAN, M., AND FLINN, J. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)* (October 2010).

[5] BARROSO, L. A., HÖLZLE, U., AND RANGANATHAN, P. *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2018.

[6] BASET, S., SUNEJA, S., BILA, N., TUNCER, O., AND ISCI, C. Usable Declarative Configuration Specification and Validation for Applications, Systems, and Cloud. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware'17), Industrial Track* (December 2017).

[7] Bazel: a fast, scalable, multi-language and extensible build system. https://bazel.build/, 2020.

[8] BELL, J., LEGUNSEN, O., HILTON, M., ELOUSSI, L., YUNG, T., AND MARINOV, D. DeFlaker: Automatically Detecting Flaky Tests. In *In Proceedings of the 40th International Conference on Software Engineering (ICSE'18)* (May 2018).

[9] BEYER, B., MURPHY, N. R., RENSIN, D. K., KAWAHARA, K., AND THORNE, S. *Site Reliability Workbook: Practical Ways to Implement SRE*. O'Reilly Media Inc., August 2018.

[10] Bhagwan, R., Kumar, R., Maddila, C. S., and Philip, A. A. Orca: Differential Bug Localization in Large-Scale Services. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)* (October 2018).

[11] Buck: A fast build tool. https://buck.build/, 2020.

[12] Chen, Q., Wang, T., Legunsen, O., Li, S., and Xu, T. Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems. In *In Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)* (November 2020).

[13] Cui, W., Ge, X., Kasikci, B., Niu, B., Sharma, U., Wang, R., and Yun, I. REPT: Reverse Debugging of Failures in Deployed Software. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)* (October 2018).

[14] Docker Hub. https://www.docker.com/products/docker-hub, 2020.

[15] Fowler, M. Eradicating Non-Determinism in Tests. https://martinfowler.com/articles/nonDeterminism.html, April 2011.

[16] Gradle Build Tool. https://gradle.org/, 2020.

[17] Graves, T. L., Harrold, M. J., Kim, J.-M., Porter, A., and Rothermel, G. An Empirical Study of Regression Test Selection Techniques. *ACM Transactions on Software Engineering and Methodology 10*, 2 (April 2001), 184–208.

[18] Gunawi, H. S., Hao, M., Leesatapornwongsa, T., Patana-anake, T., Do, T., Adityatama, J., Eliazar, K. J., Laksono, A., Lukman, J. F., Martin, V., and Satria, A. D. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC'14)* (November 2014).

[19] Gunawi, H. S., Hao, M., Suminto, R. O., Laksono, A., Satria, A. D., Adityatama, J., and Eliazar, K. J. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)* (October 2016).

[20] HADOOP-10508. RefreshCallQueue fails when authorization is enabled. https://issues.apache.org/jira/browse/HADOOP-10508, 2014.

[21] HBASE-22559. [RPC] set guard against CALL_QUEUE_HANDLER_FACTOR_CONF_KEY. https://issues.apache.org/jira/browse/HBASE-22559, 2019.

[22] HBASE-23962. Improving the documentation for 'hbase.regionserver.hlog.reader, writer.impl'. https://issues.apache.org/jira/browse/HBASE-23962, 2020.

[23] HDFS-15124. Crashing bugs in NameNode when using a valid configuration for 'dfs.namenode.audit.loggers'. https://issues.apache.org/jira/browse/HDFS-15124, 2020.

[24] HDFS-15250. Setting 'dfs.client.use.datanode.hostname' to true can crash the system because of unhandled UnresolvedAddressException. https://issues.apache.org/jira/browse/HDFS-15250, 2020.

[25] HDFS-7684. The host:port settings of the daemons should be trimmed before use. https://issues.apache.org/jira/browse/HDFS-7684, 2015.

[26] HDFS-7727. Check and verify the auto-fence settings to prevent failures of auto-failover. https://issues.apache.org/jira/browse/HDFS-7727, 2015.

[27] Huang, P., Bolosky, W. J., Sigh, A., and Zhou, Y. ConfValley: A Systematic Configuration Validation Framework for Cloud Services. In *Proceedings of the 10th ACM European Conference in Computer Systems (EuroSys'15)* (April 2015).

[28] Huo, C., and Clause, J. Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)* (November 2014).

[29] Ivanković, M., Petrović, G., Just, R., and Fraser, G. Code Coverage at Google. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)* (August 2019).

[30] Kasikci, B., Schubert, B., Pereira, C., Pokam, G., and Candea, G. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-Production Failures. In *Proceedings of the 25th ACM Symposium on Operating System Principles (SOSP'15)* (October 2015).

[31] Keller, L., Upadhyaya, P., and Candea, G. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In *Proceedings of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'08)* (June 2008).

[32] Li, S., Li, W., Liao, X., Peng, S., Zhou, S., Jia, Z., and Wang, T. ConfVD: System Reactions Analysis and Evaluation Through Misconfiguration Injection. *IEEE Transactions on Reliability 67*, 4 (December 2018), 1393–1405.

[33] Lillack, M., Kästner, C., and Bodden, E. Tracking Load-time Configuration Options. *IEEE Transactions on Software Engineering (TSE) 44*, 12 (December 2018), 1269–1291.

[34] Liu, H., Lu, S., Musuvathi, M., and Nath, S. What bugs cause production cloud incidents? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS'19)* (May 2019).

[35] Maurer, B. Fail at Scale: Reliability in the Face of Rapid Change. *Communications of the ACM 58*, 11 (November 2015), 44–49.

[36] Apache Maven. http://maven.apache.org/, 2020.

[37] Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., and Apel, S. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)* (May 2016).

[38] Mehta, S., Bhagwan, R., Kumar, R., Ashok, B., Bansal, C., Maddila, C., Bird, C., Asthana, S., and Kumar, A. Rex: Preventing Bugs and Misconfiguration in Large Services using Correlated Change Analysis. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)* (February 2020).

[39] MUKELABAI, M., NEŠIĆ, D., MARO, S., BERGER, T., AND STEGHÖFER, J.-P. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)* (September 2018).

[40] NAGARAJA, K., OLIVEIRA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (December 2004).

[41] NARLA, C., AND SALAS, D. Hermetic Servers. https://testing.googleblog.com/2012/10/hermetic-servers.html, October 2012. Google Testing Blog.

[42] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why Do Internet Services Fail, and What Can Be Done About It? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)* (March 2003).

[43] PALATIN, N., LEIZAROWITZ, A., SCHUSTER, A., AND WOLFF, R. Mining for Misconfigured Machines in Grid Systems. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06)* (August 2006).

[44] POTHARAJU, R., CHAN, J., HU, L., NITA-ROTARU, C., WANG, M., ZHANG, L., AND JAIN, N. ConfSeer: Leveraging Customer Support Knowledge Bases for Automated Misconfiguration Detection. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'15)* (August 2015).

[45] QU, X. Configuration Aware Prioritization Techniques in Regression Testing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)* (May 2009).

[46] RABKIN, A., AND KATZ, R. Precomputing Possible Configuration Error Diagnosis. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)* (November 2011).

[47] RABKIN, A., AND KATZ, R. Static Extraction of Program Configuration Options. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)* (May 2011).

[48] RABKIN, A. S. *Using Program Analysis to Reduce Misconfiguration in Open Source Systems Software*. PhD thesis, University of California, Berkeley, 2012.

[49] SANTOLUCITO, M., ZHAI, E., DHODAPKAR, R., SHIM, A., AND PISKAC, R. Synthesizing Configuration File Specifications with Association Rule Learning. In *Proceedings of 2017 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'17)* (October 2017).

[50] SANTOLUCITO, M., ZHAI, E., AND PISKAC, R. Probabilistic Automated Language Learning for Configuration Files. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV'16)* (July 2016).

[51] SAVOIA, A. Code coverage goal: 80% and no less! https://testing.googleblog.com/2010/07/code-coverage-goal-80-and-no-less.html, July 2010. Google Testing Blog.

[52] SAVOR, T., DOUGLAS, M., GENTILI, M., WILLIAMS, L., BECK, K., AND STUMM, M. Continuous Deployment at Facebook and OANDA. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering (ICSE'16)* (May 2016).

[53] SHERMAN, A., LISIECKI, P., BERKHEIMER, A., AND WEIN, J. ACMS: Akamai Configuration Management System. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)* (May 2005).

[54] SHIEBER, J. Facebook blames a server configuration change for yesterday's outage. https://techcrunch.com/2019/03/14/facebook-blames-a-misconfigured-server-for-yesterdays-outage/, March 2019.

[55] TANG, C., KOOBURAT, T., VENKATACHALAM, P., CHANDER, A., WEN, Z., NARAYANAN, A., DOWELL, P., AND KARL, R. Holistic Configuration Management at Facebook. In *Proceedings of the 25th ACM Symposium on Operating System Principles (SOSP'15)* (October 2015).

[56] THE APACHE HBASE REFERENCE GUIDE. Default Configuration. https://hbase.apache.org/book.html#hbase_default_configurations, 2020.

[57] TILLMANN, N., AND SCHULTE, W. Parameterized Unit Tests. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05)* (September 2005).

[58] TILLMANN, N., AND SCHULTE, W. Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution. *IEEE Software 23*, 4 (July 2006), 38–47.

[59] TUNCER, O., BILA, N., ISCI, C., AND COSKUN, A. K. ConfEx: An Analytics Framework for Text-Based Software Configurations in the Cloud. Tech. Rep. RC25675 (WAT1803-107), IBM Research, March 2018.

[60] WACKER, M. Just Say No to More End-to-End Tests. https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html, April 2015. Google Testing Blog.

[61] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (December 2004).

[62] WANG, Y.-M., VERBOWSKI, C., DUNAGAN, J., CHEN, Y., WANG, H. J., YUAN, C., AND ZHANG, Z. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *Proceedings of the 17th Large Installation Systems Administration Conference (LISA'03)* (October 2003).

[63] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (December 2004).

[64] WONG, W. E., GAO, R., LI, Y., ABREU, R., AND WOTAWA, F. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering (TSE) 42*, 8 (August 2016), 707–740.

[65] XIANG, C., HUANG, H., YOO, A., ZHOU, Y., AND PASUPATHY, S. PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC'20)* (July 2020).

[66] XIANG, C., WU, Y., SHEN, B., SHEN, M., HUANG, H., XU, T., ZHOU, Y., MOORE, C., JIN, X., AND SHENG, T. Towards Continuous Access Control Validation and Forensics. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19)* (November 2019).

[67] XU, T., JIN, X., HUANG, P., ZHOU, Y., LU, S., JIN, L., AND PASUPATHY, S. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)* (November 2016).

[68] XU, T., AND MARINOV, D. Mining Container Image Repositories for Software Configurations and Beyond. In *In Proceedings of the 40th International Conference on Software Engineering (ICSE'18), New Ideas and Emerging Results* (May 2018).

[69] XU, T., ZHANG, J., HUANG, P., ZHENG, J., SHENG, T., YUAN, D., ZHOU, Y., AND PASUPATHY, S. Do Not Blame Users for Misconfigurations. In *Proceedings of the 24th ACM Symposium on Operating System Principles (SOSP'13)* (November 2013).

[70] XU, T., AND ZHOU, Y. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Computing Surveys (CSUR) 47*, 4 (July 2015).

[71] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L. N., AND PASUPATHY, S. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)* (October 2011).

[72] YOO, S., AND HARMAN, M. Regression Testing Minimisation, Selection and Prioritization: A Survey. *Software Testing, Verification, and Reliability 22*, 2 (March 2012), 67–120.

[73] YUAN, C., LAO, N., WEN, J.-R., LI, J., ZHANG, Z., WANG, Y.-M., AND MA, W.-Y. Automated Known Problem Diagnosis with Event Traces. In *Proceedings of the 1st ACM European Conference on Computer Systems (EuroSys'06)* (April 2006).

[74] YUAN, D., LUO, Y., ZHUANG, X., RODRIGUES, G., ZHAO, X., ZHANG, Y., JAIN, P. U., AND STUMM, M. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)* (October 2014).

[75] YUAN, D., XIE, Y., PANIGRAHY, R., YANG, J., VERBOWSKI, C., AND KUMAR, A. Context-based Online Configuration Error Detection. In *Proceedings of 2011 USENIX Annual Technical Conference (USENIX ATC'11)* (June 2011).

[76] ZHANG, G., AND LIU, L. Why Do Migrations Fail and What Can We Do about It? In *Proceedings of the 25th USENIX Large Installation System Administration Conference (LISA'11)* (December 2011).

[77] ZHANG, J., RENGANARAYANA, L., ZHANG, X., GE, N., BALA, V., XU, T., AND ZHOU, Y. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'14)* (March 2014).

[78] ZHANG, S., AND ERNST, M. D. Which Configuration Option Should I Change? In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)* (May 2014).

[79] ZHANG, Y., RODRIGUES, K., LUO, Y., STUMM, M., AND YUAN, D. The Inflection Point Hypothesis: A Principled Debugging Approach for Locating the Root Cause of a Failure. In *Proceedings of the 26th ACM Symposium on Operating System Principles (SOSP'19)* (October 2019).

[80] ZOOKEEPER-2299. NullPointerException in LocalPeerBean for ClientAddress. https://issues.apache.org/jira/browse/ZOOKEEPER-2299, 2015.

[81] ZOOKEEPER-3721. PR #1266: ZOOKEEPER-3721: Making the boolean configuration parameters consistent. https://github.com/apache/zookeeper/pull/1266, 2020.