REUSING SOFTWARE TESTS FOR CONFIGURATION TESTING: A CASE STUDY OF THE HADOOP PROJECT

BY

RAN ANG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Assistant Professor Tianyin Xu

ABSTRACT

Configuration is an inseparable piece of today's software development. Due to its dynamic nature and lack of standardized checking procedure, misconfiguration has been one of the dominant factors contributing to software failures in production and wide spread outages. Existing techniques (static validation, system tests, canaries, etc.) for detecting misconfigurations have limitations, either being too coarse grained and unable to precisely exercise the changed configuration value, or not considering the interaction between configuration and software source code. Configuration testing takes pages from traditional software testing practices to test software configurations in a similar vein as how software code is tested.

This thesis makes a first step exploration towards configuration testing, focusing on analyzing feasibility and effectiveness of reusing existing source code tests to test configuration values. Through building a semi-automated infrastructure associating unit tests with configuration parameters, and later run the associated tests against injected correct and incorrect configuration values for each parameter, this thesis shows that reusing existing software test for configuration testing directly could yield an effective rate of 41.7% and 48.8%, for Hadoop Common and HDFS, respectively. Further, this thesis conducts in-depth analysis on tests that cannot be effectively reused, categorizes code patterns of these tests that have false positives or negatives, and provides examples on rewriting these tests.

To my parents and Mandy, for their love and support.

ACKNOWLEDGMENTS

This thesis is a joint effort with my fellow students Sam Cheng and Xudong Sun, under the supervision of Owolabi Legunsen and my advisor Assistant Professor Tianyin Xu. I am massively indebted to these inspiring co-workers, without whom this thesis would not have been possible. Owolabi's exceptional ability to make concrete plans has maximized the productivity of each of our meetings, from which I benefited tremendously. Tianyin's high standards has immensely improved the quality of this thesis. His unique vision forms the basis of this work, and his passion towards research keeps motivating me. I sincerely appreciate having the opportunity to work with all of them.

Thanks to Professor Darko Marinov for providing me funding for a semester, as well as invaluable comments and constructive feedbacks on the early stage of this work.

I sincerely thank Professor Michael Walfish for leading me into the intriguing world of systems research. His class stimulated my interest in computer systems, and motivated me to enter graduate school.

I would also like to thank Lawrence Ying and Jeff Day for being fantastic hosts and providing me with challenging tasks during the two summers I spent at Google. The industry experience perfectly complements my study at school.

I am incredibly fortunate to know a group of amazing friends: Bingzhe, Rui, Minshu, Tong, Du, Shichu, and Kuan-Yen, to name a few. My life in Urbana-Champaign for the past two years would have been unimaginably harder without them. I have learnt so much from all of them, and I am deeply grateful for all the good times we share together.

I want to thank Mandy for always being there for me. I am extremely lucky to have her in my life, and I look forward to our journey ahead.

Lastly, I want to thank my parents, and their parents, for their unconditional love and support.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 BACKGROUND AND RELATED WORK 3
- 2.1 Software Testing 3
- 2.2 Limitation of State of the Art 3
- 2.3 Configuration Testing 5

CHAPTER 3 THE HADOOP CONFIGURATION INTERFACE 6
- 3.1 Configuration Files 6
- 3.2 Configuration APIs 7
- 3.3 Configuration Usage in Test Code 8

CHAPTER 4 METHODOLOGY AND INFRASTRUCTURE 10
- 4.1 Test Selection and Mapping 10
- 4.2 Value Generation and Injection 13
- 4.3 Run Tests and Interpret Results 14

CHAPTER 5 RESULTS AND FINDINGS 16
- 5.1 Effectiveness 16
- 5.2 False Negatives 18
- 5.3 False Positives 27
- 5.4 Test Rewrites 32

CHAPTER 6 FUTURE WORK 35
- 6.1 Addressing Limitations 35
- 6.2 Towards A Standardized Practice 35

CHAPTER 7 CONCLUSION 37

REFERENCES 38

CHAPTER 1: INTRODUCTION

On March 14, 2019, Facebook, WhatsApp, and Instagram all went down for around a day, and the root cause of this incident was later addressed on Facebook's offcial Twitter account as “a server configuration change” [1]. On June 2, 2019, Google Cloud experienced a major outage that impacted multiple services globally for around four hours, which they later published a detailed postmortem [2] revealing that the outage is caused by “two normally-benign misconfigurations, and a specific software bug”. On July 11, 2019, Twitter was down for around an hour due to “an internal configuration change” [3]. And the list goes on. For years, misconfiguration has been among the dominant causes of system failures in production [4, 5], resulting in wide spread outages impacting millions of users.

The increasing complexity of software also increases the difficulty in detecting misconfigurations. Although software source code is often heavily tested before releases, configuration files are not. What makes things worse is that configurations are typically updated independently from the software code that uses them. A typical software deployment process is that developers write code which takes in configurable values, test their code with the configurable values fixed, and push their code to production. After this, they update configuration values through a separate process or specialized tools whenever they desire a behavior change controlled by the updated configurations [6, 7], which is an extremely error-prone process.

The question then arises: if software source code can be tested, why can't configuration values be tested? Since software tests are already exercising configuration values as constants, a natural first step attempt would be to try reuse these existing tests for configuration testing [8]. In an effort to identify the opportunities and challenges to reuse exiting source code tests for checking configuration correctness, this thesis presents a thorough case study of Hadoop, a mature and widely-used open-source software project. The thesis makes the following contributions:

- Develop a semi-automated infrastructure for doing test reuse analysis;
- Quantify the effectivness of reusing existing tests in the context of Hadoop Common and HDFS modules;
- Categorize different cases why some tests cannot be effectively reused;
- In-depth analysis with detailed examples about those non-reusable tests;
- Identify opportunities to rewrite some tests so that they could be effectively reused.

The rest of this thesis is structured as follows. Chapter 2 discusses traditional software testing as an inspiration for configuration testing, as well as state-of-the-art research on checking configuration correctness and their limitations. Chapter 3 gives a brief overview of Hadoop configuration interface, providing necessary context to interpret results in later chapters. Chapter 4 explains the infrastructure being developed for doing the study. Chapter 5 presents the quantitative results on reuse effectiveness and does in-depth analysis on the ineffective cases. The future work is discussed in Chapter 6, and Chapter 7 concludes the thesis.

CHAPTER 2: BACKGROUND AND RELATED WORK

This chapter first discusses how traditional software testing helps catch software bugs but not software misconfigurations; then summarizes state-of-the-art research achievements on dealing with misconfiguration and their limitation; and lastly describes configuration testing, taking pages from software testing practices.

2.1 SOFTWARE TESTING

Software testing is arguably one of the most important software development practices, and has been tightly integrated into nowadays software development pipelines in both large tech giants and small startups [9, 10, 11]. By writing tests that setup various use cases, reach as much source code paths as possible, and exercise different aspect of the software product, developers rely on testing to break their code in ways that they had not thought of, and in early development stages before code breaks in production. Testing has shown to be useful in revealing software bugs that are often notoriously hard to spot only by asking developers to reason about the code logic.

Based on how much code is covered by the test, software tests could be divided into different levels. Unit tests normally only check the correctness of individual methods; integration tests normally check if multiple methods work correctly together; system tests treat the software system as a black box and test its end-to-end behavior. Other than these, there are specialized tests that not only check the functional correctness of the code, but also check non-functional aspects such as performance and security constraints.

2.2 LIMITATION OF STATE OF THE ART

Despite all its usefulness, traditional software testing is geared towards testing software behavior based on certain constant values in code, and cannot effectively deal with configurable values. Software configurations are dynamic and heterogenous by definition, and thus could be a source of bug which easily slip through the software testing guard.

Thus, developers and researchers over the years have come up with numerous ways of combating misconfigurations that worked relatively well, but not without limitations. I present these limitations below.

2.2.1 Static Validation

Lots of efforts has been put into building better static validation techniques [12, 13, 14, 15, 16], which take in a set of predefined specifications, and check if configuration values obey those specifications. The limitation with this approach is two fold:

Codifying a complete rule set is hard. Even with new programming language support and automatic rule inference or specifications deriving [5, 13, 14, 15], coming up with an accurate and complete rule set is an inherently hard process. New languages could help formalize how rules are expressed, but developers expressing those rules may not have a complete understanding of the software, and automatically derived specifications generally make trade-offs between accuracy and scalability.

Cannot capture undesired code behavior induced by misconfiguration. Static validation, even with comprehensively compiled rule sets, only checks the correctness of configurations alone without interacting with the software, and thus fundamentally limits its ability to ensure the configured value works correctly with the software. A typical example is when a configuration specifies a file path whose content will be used by source code to construct some object or state: static validation could only check if the file name is valid or if such file exists, yet it has no way of checking if the the content of that file actually helps constructing that object or state correctly.

2.2.2 Learning Based Method

Researchers also investigate using machine learning techniques to detect misconfigurations [14, 17, 18], which takes various forms ranging from training a classifier that differentiates correct configuration values from incorrect ones, to combining NLP techniques for analyzing crash logs or technical knowledge base associating misconfigurations with their root causes as well as solutions.

The classification based approach does not consider that outliers may not be misconfigurations, and vice versa [19]. The configuration values deviating from its regular range could come from experienced developers trying to do hand-optimization for system performance, while default values, which would certainly fall into the regular range, could well be misconfigurations when they need to be set based on runtime environments but are incorrectly kept unchanged.

The combined machine learning and NLP approach for mining information from documents inherently requires a large and high-quality dataset to apply the learning algorithms on. However, such dataset, if exists all at, might not always be available to developers.

2.2.3 Canary Test

The idea of canary test is to run the software in a realistic enough environment, which only a small group of users have access, and observe any abnormal behavior of the software. There are different practices for doing canary, some build fully-fledged dedicated canary analysis services [20], some just use it loosely as a deployment strategy to follow, but all aiming at finding bugs in the early stage of software releases.

While canaries, and system tests (Section 2.1) for this matter, are useful for revealing bugs that only appear when different components of the software are put together in a production-like environment, it is hard to reason if these tests could actually exercise the change of configuration values. Prior study has shown that configurations may only be used under special conditions, which could be hard for the canary or system test to trigger.

2.3 CONFIGURATION TESTING

Configuration testing [8] tests configuration values just like traditional software testing for software source code described in Section 2.1. Upon a configuration change, there should be tests that exercise on the configuration values about to be pushed to production. The tests pass if software components related to this configuration change still behaves as expected, and fail if the configuration change leads to undesired behavior of the software. Configuration testing provides developers with clear indication on whether their configuration changes are legal with respect to the software being configured, complementing the aforementioned existing techniques.

CHAPTER 3: THE HADOOP CONFIGURATION INTERFACE

This Chapter describes the configuration interface of the Hadoop project, including the configuration file structure and the APIs (application program interfaces) for getting and setting configuration values. It establishes the background and context to interpret our methodology and results described in the latter chapters.

Note that the configuration ecosystem of Hadoop is representative and resembles a large number of system software projects, such as Spark, HBase, Alluxio, and OpenStack [5, 21]. Therefore, our methodology and infrastructure (described in Chapter 4) can be easily extended to other software projects.

3.1 CONFIGURATION FILES

Hadoop adopts key-value style configuration files in XML format. Each Hadoop component maintains a specific configuration file (e.g., `core-site.xml` for Hadoop Common and `hdfs-site.xml` for HDFS). A configuration file consists of a set of configuration parameters. Each configuration parameter is represented by a name and a value, identified by `<name>` and `<value>` in the configuration file, respectively.

Figure 3.1 shows a snippet of the configuration file (`core-site.xml`) from the Hadoop `Common` module.

```
<configuration>
    <property>
        <name>
            net.topology.script.number.args
        </name>
        <value>
            100
        </value>
    </property>

    <property>
        ...
    </property>
    ...
</configuration>
```

Figure 3.1: A snippet from Hadoop `core-site.xml`.

As noted in the Hadoop official document[1], each configuration file is managed by an

[1]http://hadoop.apache.org/docs/r2.8.5/api/org/apache/hadoop/conf/Configuration.html

abstracted `resource`, identified by the name or of that file. There is a precedence for loading different resources, allowing a customized value overwriting the default one for the same key. More specifically, each of the Hadoop configuration class as specified in Table 3.1 has a `static` block allowing developers to specify what are the resources they want to load in order. For example, in `Configuration.java` inside Hadoop Common, we have the loading sequence shown in figure 3.2, with the latter loaded resource having higher precedence to overwrite the previous loaded ones. Notably, each parameter inside the configuration file could also use an optional field `<final>` (default to `false`) indicating that no subsequent overwrites of this value is allowed.

In our study (see Section 4), we added an additional resource to ease the injection of new configuration values (Section 4.2.2). We create our own XML files: `ctest-core-default.xml` for Hadoop Common and `ctest-hdfs-default.xml` for HDFS and set them with the highest loading precedence.

```
static{
  // Irrelevant part omitted.
  addDefaultResource("core-default.xml");
  addDefaultResource("core-site.xml");

  // Loading our own resource for injecting values.
  addDefaultResource("ctest-core-default.xml");
}
```

Figure 3.2: Configuration loading sequence for Hadoop Common

3.2 CONFIGURATION APIS

Hadoop has a set of unified configuration APIs for getting and setting configuration values, referred to as the *getter* and *setter* APIs. The APIs are developed based on Apache Commons Configuration. All configuration parameters are stored in a `Configuration` object as a key-value pair (see Section 3.1). The `Configuration` object has two types of interfaces, i.e., *getter methods* and *setter methods*. The basic form of the interfaces are described as follows:

- `String get(String name, String defaultValue)`
  Gets the value of `name`, or return `defaultValue` if no such property exists.

- `void set(String name, String value)`
  Sets `value` for `name`. Subsequent `get()` will return this value being set.

There are a range of variants of these two APIs, such as `getInt()`, `getRaw()`, `setBoolean()`, `setEnum()`, which are specialized for different value types and purposes.

Figure 3.3 shows an example of how getter/setter methods are used in the Hadoop code. The configuration parameter is named `net.topology.script.number.args` with the default value being `100`. Line 6 is a getter method, which returns the default value in the configuration file for the parameter, or returns the `DFAULT_ARG_COUNT` if there is not a default value defined. Line 9 is a setter method, which sets the value for this parameter to be 10, and all subsequent getter method calls to this parameter should return 10.

```
1   public static final String SCRIPT_ARG_COUNT_KEY =
        "net.topology.script.number.args";
2   public static final int DEFAULT_ARG_COUNT = 100;
3   Configuration conf = new Configuration();
4
5   // Getter example.
6   maxArgs = conf.getInt(SCRIPT_ARG_COUNT_KEY, DEFAULT_ARG_COUNT);
7
8   // Setter example.
9   conf.setInt(SCRIPT_ARG_COUNT_KEY, 10);
```

Figure 3.3: Simplified code snippet for getter and setter methods

Note that different components in Hadoop maintain different variants of the `Configuration` class by inheriting `Configuration`. We summarize these subclasses in Table 3.1. On the other hand, we find that all these subclasses of the `Configuration` class is simple wrappers and use the same set of getter and setter methods. Therefore, we only need to instrument a small number of getter/setter methods in the base `Configuration` class in our study, as described in Section 4.1.1.

Project Name	**Configuration Class**
hadoop-common-project	org.apache.hadoop.conf.Configuration
hadoop-hdfs-project	org.apache.hadoop.hdfs.HdfsConfiguration
hadoop-yarn-project	org.apache.hadoop.yarn.conf.YarnConfiguration
hadoop-mapreduce-project	org.apache.hadoop.mapred.JobConf

Table 3.1: Configuration classes in different Hadoop modules

3.3 CONFIGURATION USAGE IN TEST CODE

The study presented in this thesis focuses primarily on test code. Therefore, it is necessary to examine how test code uses configuration values, and the APIs used in the test code. We find that test code uses the same configuration APIs described in Section 3.2 as the main programs. Therefore, the study methodology and infrastructure relies on the getter/setter APIs. Note that a test could set certain values for one or more configuration parameters during the test setup, which constructs necessary context for the test to run.

Further note that in some cases, the configuration APIs are overused as a singleton hash map (which is a bad programming practice). Fortunately, it is straightforward to filter out such cases, as the keys in those cases are not configuration parameter names but other strings. We simply ignore cases like this, and only focus our analysis on the configuration parameters listed in `*-default.xml`.

CHAPTER 4: METHODOLOGY AND INFRASTRUCTURE

This chapter describes the methodology of our study and the semi-automated infrastructure we set up to conduct the study. Recall that our goal is to study the feasibility and effectiveness of reusing existing software tests as configuration tests. Here, *software tests* are referring to as existing tests that are designed for testing software implementations (which typically hardcodes configuration values in the test code, as discussed in Section 2.1), while *configuration tests* are referring to the new type of tests for the purpose of testing configuration values.

The main idea is to replace the hardcoded or default configuration values used by existing software tests with representative values, including both correct and erroneous values (see Section 4.2). Then, we evaluate whether all tests pass on a correct values and any test fails on an incorrect value. To achieve this, we have built the following components:

- Associating configuration parameters with software tests that use them;
- Generating and injecting configuration values;
- Running tests and analyzing test results.

We will discuss the detailed test results and findings in Section 5.

4.1 TEST SELECTION AND MAPPING

Our first step is to identify existing software tests that use configuration parameters during the test run. These tests can potentially be used as configuration tests, as they exercise the configuration values. Since Hadoop has a unified configuration interface (Section 3.2), we instrument the configuration getter and setter methods by adding log statements to record the configuration parameters retrieved during the execution of the test run. Hadoop implements all the tests in the `JUnit` testing framework, and uses Maven Surefire plugin to run the tests. After each test run, all the log messages will be recorded in Maven surefire reports at the granularity of a test *class*. These runtime logs enable us to associate configuration parameters with the software tests that exercise the values of the parameter. In other words, for the configuration parameter, these tests are potential candidates that can be reused as configuration tests.

4.1.1 Instrumenting Configuration Interfaces

We add log printing statements in configuration getter/setter methods to capture the events that retrieve or mutate configuration values during the test run. Note that we only need to instrument the following four methods instead of all the getter or setter methods, because the other methods are merely wrappers around the following ones, and all tests must go through these methods to get or set configuration values.

- `public String get(String name)`
- `public String get(String name, String defaultValue)`
- `public String getRaw(String name)`
- `public void set(String name, String value, String source)`

After the instumentation, `get(String name)`, `get(String name, String defaultValue)` and `getRaw(String name)` will log `"[CTEST][GET-PARAM] <name>"` where `<name>` is the key name of configuration parameter. So we could record which configuration parameter was got during tests running. Similarly, `set(String name, String value, String source)` will log `"[CTEST][SET-PARAM-VALUE] <name> <value>"` where we could record which configuration parameter was set and which value it was set to.

4.1.2 Test Listener

Since the Surefire report is at a test class level instead of test method level, to obtain a full list of test methods, we make use of `RunListener`. It is an event listener for `JUnit`, which allows implementing customized functions upon occurrence of various predefined event. Hadoop has a `TimedOutTestsListener` that extends `RunListener`. All we need is to add two override methods `testStarted` and `testFinished` that runs before and after each test method run to log the test class name and method name. Then we can get a complete list of test methods from the information `TimedOutTestsListener` logged out while doing a full module test run. In our case, a full module test run simply means running `mvn test` inside the module we want to analyze. We show in figure 4.1 the two override function we add inside `TimedOutTestsListener`.

It is worth noting that this is not the only way to get the complete list of all test method. One could modify Hadoop's `pom.xml` to log out all the test method, or use static analysis to extract the list. We take the dynamic event listener approach because it is easy to implement, and this does not mess up Hadoop's `pom.xml`.

```java
@Override
public void testStarted(Description description) {
    System.out.println("[CTEST-LISTENER][METHOD-START] " +
        description.getClassName() + "#" + description.getMethodName());
}

@Override
public void testFinished(Description description) {
    System.out.println("[CTEST-LISTENER][METHOD-FINISH] " +
        description.getClassName() + "#" + description.getMethodName());
}
```

Figure 4.1: Test listener to log each test method name

4.1.3 Mapping Configuration Parameters to Tests

Lastly, we take the complete list of test methods for a module as well as the instrumented configuration interface, run each test method one-by-one using `mvn surefire:test -DTest=<TEST_METHOD>`. After each test method run, we process the Maven surefire report, parse out the configuration keys and values logged in this report. These parameters are associated with the test method that just finished running, denoting as

$$t_1 \rightarrow \{p_1, p_2, p_3, ...\} \tag{4.1}$$

After all tests finish, we map each configuration parameter to a set of tests that uses it, in the form of

$$\{p_1 \rightarrow \{t_{11}, t_{12}, ..., t_{1i}\}, p_2 \rightarrow \{t_{21}, t_{22}, ..., t_{2j}\}, ...\}. \tag{4.2}$$

For the purpose of finer grained analysis, we separate all tests associate with a parameter into two categories: *getter-only* test, which only gets the value for this parameter and potentially other parameters without any modification to the configuration file throughout the test lifetime; and *getter-setter* test, which not only gets the value for this parameter, but also sets the value of some other parameters. To further clarify, If there is a test that sets the value of p_1, and later gets the value of p_1, the test is not included in the associated test suite for p_1.

This way, we map two test suites for each configuration parameter, and dump the mapping for all parameters to disk.

4.2 VALUE GENERATION AND INJECTION

The second step is to generate representative configuration values, including both correct and erroneous values, and to run the tests using these values instead of the hardcoded values in the original test code. The purpose is to evaluate whether the tests can be directly reused for configuration testing after parameterizing the hardcoded values. If it cannot be reused, we need to identify the root cause.

4.2.1 Generating Representative Values

For each parameter, we study its specifications using the combined knowledge from the parameter description and online examples. Based on the specification, we then choose 3 good (correct) values, and 3 bad (incorrect) values for this parameter. If any of the associated tests fail for any of the incorrect values we generate, we count that parameter as being able to be effectively tested through reusing existing tests, and it follows that those tests able to single out that incorrect value is reusable for configuration testing purpose.

The subtlety here is how to choose the correct and incorrect values for a parameter, and here are the general guidelines we followed:

- If the parameter is of **numeric type (int, float, long)**:
  Correct values could be:
  a. default number as specified in *-default.xml;
  b. a reasonably large number (maybe MAXINT but does not always make sense);
  c. 0 if specification allows, otherwise 1;
  Incorrect values could be:
  a. a random string;
  b. a very small number (often use -10 but does not always make sense);
  c. 0 if specification does not allow, otherwise use a float not allowed by the specification (often use -0.5 but does not always make sense).
- If the parameter is of **string** type, this is much harder and should be dealt with on a case by case basis. A general rule of thumb is to add more special characters when choose incorrect values.

The above guidelines are only meant to serve as suggestions, although they do cover a lot of cases for numeric types. String types are indeed harder and require more manual work. We tried to make the best decision possible when choosing these values, but admittedly, this is still a overall error-prone process, as the correct value we choose might actually be incorrect and the incorrect value we choose might actually be correct due to our misunderstanding of the specification. Having some way of automatically generating high-quality correct and incorrect values could be an attractive research project, but is not the goal of this thesis.

4.2.2 Injecting New Configuration Values

We build a tool to inject values we choose for each parameter before we run the tests. For Hadoop Common, it always uses `org.apache.hadoop.conf.Configuration` class to load, get and set the configuration parameter values. It loads values from `core-default.xml` and `core-site.xml` by default, and the values loaded from the later will overwrite the former one. We create a customized configuration file (`ctest-core-default.xml`) and make `Configuration` load this file with the highest precedence, so that the values from the customized XML can overwrite the values from the two default XML files (Section 3.1). Every time before we run associated tests for a parameter, we write the key-value pair of that parameter in the correct format (Section 3.1) into `ctest-core-default.xml` so we can make sure the tests will be running with the values we choose.

For HDFS, both `Configuration` and `HdfsConfiguration` are used to load configuration values. `HdfsConfiguration` loads `hdfs-default.xml` and `hdfs-site.xml` in a similar way as `Configuration` does. We also create a `ctest-hdfs-default.xml`, which is the last resource to load, and write the injected values into this file.

One caveat is that some tests from HDFS only use `Configuration` to load the configuration values, which means only injecting into `ctest-hdfs-default.xml` will not correctly return the value injected. To fix this, we make an observation that all tests need either `Configuration` or `HdfsConfiguration` to load configuration values, and decide to inject values into both `ctest-core-default.xml` and `ctest-hdfs-default.xml`.

4.3 RUN TESTS AND INTERPRET RESULTS

We build a fully automated test runner in about 500 lines of Python code. For each parameter, we first inject one of the chosen values for that parameter (Section 4.2.2), then run all the tests associate with this parameter using Maven Surefire plugins's command line utilities. Finally, we collect Maven surefire logs and record the test results (either PASS or FAIL) for each test run.

Notably, we use a few options to improve test running accuracy and efficiency. First, the test is running one by one to rule out potential flakiness caused by order-dependent flaky tests. Second, given running test one by one is extremely time consuming. we use the `mvn surefire:test` option for each test run to reduce runtime. This option skips the build stage that would have incur addtional time (about 10 seconds) for every test before it starts running. The option can reduce runtime by at least 50% on average. Third, for each run of associated test set of a parameter on an injected incorrect value, we will abort the run immediately when one test fails. The motivation here is also to reduce runtime. Since the purpose is to determine whether a test set is effective in determining the correctness of a value, and the failing of any test is already a indication of this test set being effective, there is no need to continue running other tests in the set for this particular value.

Test efficiency optimizations. We observe that for some parameters with a large number (more than 2000) of tests associated, running all tests takes way too long and does not scale. We found this to be true for more than half of the HDFS configuration parameters. Therefore, reducing the number of tests while preserving the testing effectiveness is desired.

We find that the test redundancy in HDFS is mostly attributed to a special class – `org.apache.hadoop.hdfs.MiniDFSCluster`. Owing to the distributed nature of HDFS, a large percentage of its tests are using this `MiniDFSCluster` to setup a cluster, before continuing the actual test steps. Instantiating this class results in invoking getter function on 75% of the total HDFS configuration parameters. That is to say, 75% of HDFS parameters has a large number of associated tests only because these tests instantiated `MiniDFSCluster` during setup, which is a similar process for different tests. Based on this information, we use an aggressive test reduction approach: for a set of tests $\{t_{i1}, ..., t_{ij}\}$ associated with a parameter p_i, if all tests in this set only use p_i in setting up the `MiniDFSCluster` and nowhere else, only one test in this should be kept in the final mapping for p_i.

Adopting this reduction plan, we are able to reduce the number of tests for some parameters from around 2500 to around 300, immensely shortening the time takes to run the process described at the start of this section.

CHAPTER 5: RESULTS AND FINDINGS

This chapter presents the results and findings of our study on 127 configuration parameters from Hadoop Common and 90 configuration parameters from HDFS (randomly selected), with the methodology described in Section 4.

5.1 EFFECTIVENESS

We first show the effectiveness of *directly* reusing existing software tests for configuration testing. The analysis is at the granularity of each configuration parameter. Recall that a configuration parameter p corresponds to a set of tests $T = \{t_1, t_2, ..., t_n\}$, referred to as the test suite for the parameter. We generate six configuration values, three correct and three incorrect, for each test to run against.

We define the following categories regarding reusing existing software tests for a given parameter.

- **Effective:** For each correct value, all tests in the test suite T of the parameter pass. For each erroneous value, at least one test in the test suite T fails.
- **Useful:** Among all tests in the test suite T, there exists at least one test that fails upon at least one erroneous value while passes upon all the correct value.
- **No Useful Test:** The parameter does not have any tests, or the test suite is not useful (defined above), i.e., none of the tests can capture one of the erroneous values while does not fail on the correct values.

We show the distribution of each category above in Figure 5.1 for all the parameters we studied in both Hadoop Common and HDFS. And we show in Figure 5.1 the percentage of parameters, among all available parameters, have at least one test associated. To summarize:

Hadoop Common has 270 parameters in total, among which 228 parameters are each associated with at least one test. Among the 228 parameters, we randomly selected and studied 127 parameters, and 53 of them are effective, yielding an effective rate of 41.7%. There are in total 17 parameters with "useful" tests, yielding a useful rate of 13.4%.

HDFS has 296 parameters in total, among which 278 parameters are each associated with at least one test. And among the 278 parameters, we randomly selected and studied 90 parameters, and 44 of them are effective, yielding an effective rate of 48.8%. There are in total 13 parameters with “useful” tests, yielding a useful rate of 14.4%.

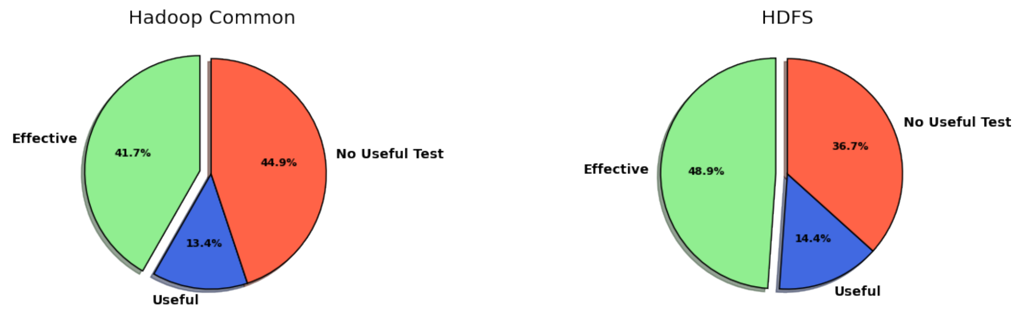

Figure 5.1: Parameter category distribution among studied ones

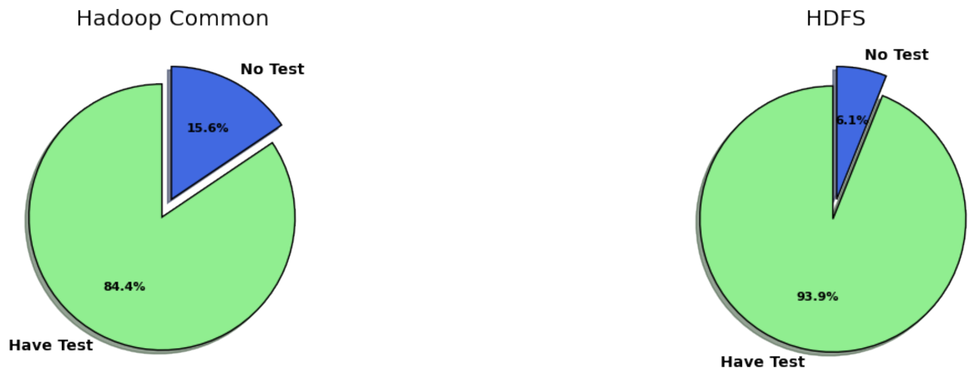

Figure 5.2: Parameters have at least one test associated

Notice that our counts on “useful” category are a bit off, since we only count parameters exhibiting false negative behavior on two or less of the three incorrect values we choose, effectively counting all parameters that exhibit false positive behavior as “not useful”. In reality, since the false positive behavior is only due to a subset of all associated tests, these parameters could still have tests that capture one or more incorrect values while not exhibiting false positive behavior. Essentially, we are under-counting instead of over-counting the number of parameters having useful tests. Accurate counting of the “useful” category requires false negative and false positive analysis on a test level instead of the current parameter level, and we leave this as future work.

For those parameters not categorized as effective, their test suite has either false positive(s) or false negative(s), or both, defined as follows:

- **False Negative (FN):** One or more of the erroneous value passes all the tests in the test suite T. Note that as long as one test can catch the error, we treat the test suite as effective (we do not require every test in T can capture all the errors as different tests can exercise different aspects of the values).

- **False Positive (FP):** One or more of the correct values fail at least one test in the test suite T. Similar as false negatives, false positives are counted at the level of the entire test suite instead of the individual test in the test suite.

For the rest of this chapter, we give in-depth analysis of both False Negatives (Section 5.2) and False Positives (Section 5.3), providing detailed examples from Hadoop Common and HDFS. Table 5.1 give a summarize of the overall distribution of false negatives and false positives.

Module	**Total Studied**	**FN**	**FP**	**Both FN&FP**
Common	127	59	19	4
HDFS	90	19	29	2

Table 5.1: General statistics for Hadoop Common and HDFS

5.2 FALSE NEGATIVES

This section explains the root causes of false negatives by providing detailed explanation and concrete code snippets. Note that false negatives are at the level of a test suite for a configuration parameter instead of an individual test. One parameter's test suite could fall in multiple categories, i.e., different tests in the test suite fail for different reasons. This is relatively rare but by all means possible.

The code patterns contributing to false negatives can be categorized as follows:

A. **Shallow Tests Only**
Test performs simple or no operation on parameters.

B. **Fallback or Auto-correction**
Code catches and handles Execption by resetting the Exception-causing value.

C. **Incomplete Validation**.
Test includes some, but not all, validation logic.

D. **Not Reaching Validation**.
Test does not reach the validation logic contained in program.

E. **Swallowed Exception**.

Exception handler log error without aborting execution.

F. **Non-functional Symptoms**.

Consequence of having the parameter misconfigured is non-functional.

We show in table 5.2 the distribution of different false negative categories for Hadoop Common and HDFS among the parameters we studied, in the form of number of parameters in each category.

Module	**Total**	**A**	**B**	**C**	**D**	**E**	**F**
Common	59	20	7	4	8	6	14
HDFS	19	0	2	1	5	4	7

Table 5.2: False negative statistics for Hadoop Common and HDFS

5.2.1 Shallow Tests Only

We observe that there are a couple tests that *gets* a list of configuration parameters, but perform only simple or no operations on these parameters (e.g., looping through and printing the values), instead of actually using them in ways that corresponds to the original intention of having them as configurations. We call these tests *shallow test*. Configuration parameters only having these tests will certainly exhibit false negative behavior, as the erroneous values do not fail the tests. For example, `net.topology.impl` is one of the configuration parameters only having these shallow tests.

Figure 5.3 shows such a shallow test named `testGetChangedProperties`. This test is checking if changes of a configuration file could be reflected. The `getChangedProperties` method loops through getting each key value pair in the configuration for finding new changes, and this is why lots of parameters are associated with this test. Since this test does nothing but looping through these parameters, it cannot check if the value a parameter is incorrect, and thus result in the false negative behavior. There are 8 shallow tests in Hadoop Common and 3 in HDFS, all listed in Table 5.2.1 (test name ignoring the `org.apache.hadoop` prefix). Among the 127 Hadoop Common parameters we studied, 20 of them only have such tests, and among 90 HDFS parameters we studied, none of them only have such tests.

```java
@Before
public void setUp () {
  conf1 = new Configuration();
  conf2 = new Configuration();

  // set some test properties
  conf1.set(PROP2, VAL1);

  // different value as conf1
  conf2.set(PROP2, VAL2);
}

@Test
public void testGetChangedProperties() {
  Collection<ReconfigurationUtil.PropertyChange> changes =
    ReconfigurationUtil.getChangedProperties(conf2, conf1);

  assertTrue("expected 1 changed properties but got " + changes.size(),
             changes.size() == 1);
}

public static Collection<PropertyChange>
  getChangedProperties(Configuration newConf, Configuration oldConf) {
    Map<String, PropertyChange> changes = new HashMap<String, PropertyChange>();

    // iterate over old configuration
    for (Map.Entry<String, String> oldEntry: oldConf) {
      String prop = oldEntry.getKey();
      String oldVal = oldEntry.getValue();
      String newVal = newConf.getRaw(prop);

      if (newVal == null || !newVal.equals(oldVal)) {
        changes.put(prop, new PropertyChange(prop, newVal, oldVal));
      }
    }
    return changes.values();
  }
```

Figure 5.3: Simplified code snippet for a shallow test in Hadoop Common

5.2.2 Fallback or Auto-correction

We find that Hadoop Common and HDFS have fallback and auto-correction logic for a small number of configuration parameters. When the configuration value is erroneous, the program would either (1) catch and handle the exception and then use the default value instead of the erroneous configuration value, or (2) change the value to the upper/lower bound value if the error is an out-of-bound error. In these two cases, the test will pass upon erroneous values (because the value is corrected) and lead to false nega-

Hadoop Common
conf.TestCommonConfigurationFields#testCompareConfigurationClassAgainstXml
conf.TestCommonConfigurationFields#testCompareXmlAgainstConfigurationClass
conf.TestCommonConfigurationFields#testXmlAgainstDefaultValuesInConfigurationClass
conf.TestConfiguration#testDumpConfiguration
conf.TestConfServlet#testWriteJson
conf.TestReconfiguration#testConfIsUnsetAsync
conf.TestReconfiguration#testConfIsUpdatedOnSuccessAsync
conf.TestReconfiguration#testGetChangedProperties
HDFS
tools.TestHdfsConfigFields#testCompareXmlAgainstConfigurationClass
tools.TestHdfsConfigFields#testCompareConfigurationClassAgainstXml,
tools.TestHdfsConfigFields#testXmlAgainstDefaultValuesInConfigurationClass

Table 5.3: The lists of shallow tests in Hadoop Common and HDFS.

tive behavior. For example, `net.topology.table.file.name` in Hadoop Common has a test `org.apache.hadoop.net.TestTableMapping#testNoFile` which calls into the logic, as shown in Figure 5.4.

In this example, the code is expected to load a mapping specified by a file with the configuration value as its path. If the filename defined by the parameter is not set or the file does not exist, the code falls back to the auto-correction logic using the predefined `NetworkTopology.DEFAULT_RACK` and an empty map instead of crashing. Thus, a misconfigured mapping file path that does not exist will not cause the associated test to fail.

Fallback logic are auto-correction are considered good practice towards resilience to configuration errors, as long as clear log messages are provided to inform users about the value changes [22, 5, 23, 24]. With the fallback/auto-correction in place, one could argue that the erroneous values are acceptable as it will be corrected anyway. On the other hand, we believe that it is important to report the behavior as false negatives as it is known the fallback/auto-correction could potentially mask errors (which could lead to unexpected, more severe consequences).

5.2.3 Incomplete Validation

A common case is that a given test includes certain validation logic for the target configuration values, but not all. Therefore, some erroneous values can be exposed during testing, while the others (often more tricky errors) cannot.

For example, as the Hadoop project uses specialized getter APIs (such as `getInt()` and

```java
public synchronized List<String> resolve(List<String> names) {
  if (map == null) {
    map = load();
    if (map == null) {
      LOG.warn("Failed to read topology table. " +
        NetworkTopology.DEFAULT_RACK + " will be used for all nodes.");
      map = new HashMap<String, String>();
    }
  }
  // Finish resolve...
}

private Map<String, String> load() {
  Map<String, String> loadMap = new HashMap<String, String>();

  String filename = getConf().get(NET_TOPOLOGY_TABLE_MAPPING_FILE_KEY, null);
  if (StringUtils.isBlank(filename)) {
    LOG.warn(NET_TOPOLOGY_TABLE_MAPPING_FILE_KEY + " not configured. ");
    return null;
  }

  try (BufferedReader reader =
        new BufferedReader(new InputStreamReader(
            new FileInputStream(filename), StandardCharsets.UTF_8))) {
      // Handle file reading omitted.
  } catch (Exception e) {
    LOG.warn(filename + " cannot be read.", e);
    return null;
  }
  return loadMap;
}
```

Figure 5.4: Simplified code snippet for fallback logic

`getClass()`), the use of these specialized APIs is effective in catching type errors directly. However, not all the errors are type errors.

Consider the parameter `io.map.index.interval`. It is intended to be used for keeping two files (a data file and an index file) in sync. Namely, for every `io.map.index.interval` records written in the data file, an entry is written in the index file. Intuitively, a value 0 should be a incorrect value for this configuration parameter, since it makes no sense to say "for every 0 records."

This parameter has 21 tests associated, but all use the parameter in a same way as shown in Figure 5.5. For any type errors, the tests will fail because the getter APIs will catch those errors. However, for the value 0, all the tests will pass. As shown in Figure 5.5, when the configuration value is used for constructing a branch condition, when a `writer` performs the `append()` operation. The consequence of using the value 0 is that the index file will be updated for each data file written, which could be unexpected to the user.

```java
@Test
public void testMidKey() throws Exception {
  MapFile.Writer writer = null;
  try {
    // conf comes from the test class setup.
    writer = new MapFile.Writer(conf, ...);
    writer.append(new IntWritable(1), new IntWritable(1));
    writer.close();
    // Reader and relevant assertion omitted.
  }
}
public synchronized void append(WritableComparable key, Writable val)
    throws IOException {

    long pos = data.getLength();
    // indexInterval take value from the configuration parameter.
    if (size >= lastIndexKeyCount + indexInterval && pos > lastIndexPos) {
      // Update position and index file omitted.
    }

    // Update data file omitted.
  }
```

Figure 5.5: Simplified code snippet for partial validation

5.2.4 Not Reaching Validation

In some other cases, the program contains comprehensive validation code to check the correctness of the configuration value; however, none of the tests associated with that parameter reaches the validation logic, resulting in false negatives when the values are erroneous.

Figure 5.6 shows such an example. The configuration parameter `hadoop.http.authentication.type` has a well defined validation logic as shown in Figure 5.6. However, none of its associated tests actually exercises this validation logic, verified by adding a log in the logic, running all associated tests, and observing no test outputing that log line.

In Figure 5.6, the authentication type is read from configuration file and put into `authHandlerName`. It is used in `getAuthenticationHandlerClassName()` to construct the `authHandlerClassName`, and later used in `initializeAuthHandler()` to load a class. If we have a random string that cannot be loaded as a class, this part of the code will throw an exception. Since none of the test code actually uses such logic, this parameter exhibits false negative behavior.

```java
public void init(FilterConfig filterConfig) throws ServletException {
  // Irrelevant part of init() omitted.
  // config construction omitted.
  String authHandlerName = config.getProperty(AUTH_TYPE, null);
  String authHandlerClassName;
  if (authHandlerName == null) {
    throw new ServletException("Authentication type must be specified: " +
        PseudoAuthenticationHandler.TYPE + "|" +
        KerberosAuthenticationHandler.TYPE + "|<class>");
  }
  authHandlerClassName = AuthenticationHandlerUtil
          .getAuthenticationHandlerClassName(authHandlerName);

  initializeAuthHandler(authHandlerClassName, filterConfig);
}

protected void initializeAuthHandler(String authHandlerClassName, FilterConfig
    filterConfig)
  throws ServletException {
  try {
    Class<?> klass = Thread.currentThread()
      .getContextClassLoader().loadClass(authHandlerClassName);
    authHandler = (AuthenticationHandler) klass.newInstance();
    authHandler.init(config);
  } catch (ClassNotFoundException | InstantiationException |
      IllegalAccessException ex) {
    throw new ServletException(ex);
  }
}
```

Figure 5.6: Simplified code snippet for unreached validation

5.2.5 Exception Swallowed

In this category, the erroneous configuration value triggers an exception; and the exception is caught by the program or the test code, but the exception handler "swallowed" the exception (e.g., printing out a log message without terminating or aborting the execution). This is a normal pattern in production systems, as often times developers do not want the system to crash upon having certain errors that they considered non-fatal.

Figure 5.7 shows an example. The configuration parameter is `fs.trash.checkpoint.interval` and the test is `org.apache.hadoop.fs.TestFsShellCopy#testDirectCopy`. The parameter is expected to be a numeric type; however setting its value to be a random string do not fail the test. This is because when the `IllegalArgumentException` is caught, the system only prints out the error. The shell command `-rm` does fail, but `testDirectCopy` still passes, as it does not depend on the file being removed correctly. So the parameter still exhibits false negative behavior, despite being caught by the `try-catch` logic.

```java
// Omit code and assertions irrelevant to this parameter.

@Test
public void testDirectCopy() throws Exception {
  FsShell shell = new FsShell(conf);
  shell.run(new String[] { "-rm", target_File.toString() });
}

// Inside FsShell.java, executed when we call shell.run()
public int run(String argv[]) throws Exception {
  Command instance = null;
  try {
    instance = commandFactory.getInstance(cmd);
    try {
      exitCode = instance.run(Arrays.copyOfRange(argv, 1, argv.length));
    } finally {
      scope.close();
    }
  } catch (IllegalArgumentException e) {
    // Only print out the error!
    displayError(cmd, e.getLocalizedMessage());
    printUsage(System.err);
  }
  return exitCode;
}

// Inside TrashPolicyDefault.java when the parameter is read.
public void initialize(Configuration conf, FileSystem fs) {
  this.emptierInterval = (long)(conf.getFloat(
    FS_TRASH_CHECKPOINT_INTERVAL_KEY, FS_TRASH_CHECKPOINT_INTERVAL_DEFAULT)
    * MSECS_PER_MINUTE);
}
```

Figure 5.7: Simplified code snippet for exception caught but swallowed

5.2.6 Non-functional Symptoms

Sometimes, the consequence of having a misconfiguration is non-functional. The code will use that misconfigured value as it is, yet resulting in security or performance issues.

As an example, parameter `dfs.namenode.audit.log.debug.cmdlist` specifies the list of HDFS NameNode command that are written to NameNode audit log only if the log level is `debug`. Figure 5.8 shows how source code uses this parameter.

If we set the value of this parameter to any random string that is not a valid NameNode command, the consequence would be that `!debugCmdSet.contains(cmd)` always evalutes to `true`, and all the commands are logged even if the log level is `info`. This could hurt performance as logging often involves file IO and is slow, but such misconfigured value does not hurt the correct functioning of the system, and does not result in test failure as well.

```java
  // Irrelevant code omitted.
  public void initialize(Configuration conf) {
    debugCmdSet.addAll(Arrays.asList(conf.getTrimmedStrings(
        DFSConfigKeys.DFS_NAMENODE_AUDIT_LOG_DEBUG_CMDLIST)));
  }

  public void logAuditEvent(...) {
    if (auditLog.isDebugEnabled() ||
    (auditLog.isInfoEnabled() && !debugCmdSet.contains(cmd))) {
      // Logic for logging the command.
  }
  }
```

Figure 5.8: Simplified code snippet for non-functional symptoms

5.2.7 A Note on False Errors

One of the key challenge of the study is that we do not have the complete specifications of the configuration parameters [25]. The configuration value generation is done manually and does not guarantee soundness (see Section 4.2). As a result, it is possible that the value that we believe to be erroneous is actually valid, and have a semantic meaning that is not mentioned anywhere in its specification.

```java
public static DataTransferThrottler getThrottler(Configuration conf) {
  long transferBandwidth =
    conf.getLong(DFSConfigKeys.DFS_IMAGE_TRANSFER_RATE_KEY,
                 DFSConfigKeys.DFS_IMAGE_TRANSFER_RATE_DEFAULT);
  DataTransferThrottler throttler = null;
  if (transferBandwidth > 0) {
    throttler = new DataTransferThrottler(transferBandwidth);
  }
  return throttler;
}

// Usage of throttler. Irrelevant code omitted.
private static void copyFileToStream(..., DataTransferThrottler throttler,
  ...) throws IOException {
  if (throttler != null) {
    throttler.throttle(num, canceler);
  }
}
```

Figure 5.9: Simplified code snippet for false error

Figure 5.9 shows an example. We use a negative integer as an erroneous value of the configuration parameter, `dfs.image.transfer.bandwidthPerSec`. A negative value is expected to be incorrect because the bandwidth cannot be negative. However, tests associated with this parameter uses the value in a special way—only creating a throttler if this parameter is greater than 0. So a negative value that we believe to be incorrect for this parameter actually has a meaning, which is never do throttle.

5.3 FALSE POSITIVES

The code patterns contributing to false positives can be categorized as follows:

A. **Test Timeout**.
   Configuration value slow down test run, resulting in timeout.

B. **Hardcoded Test Code**.
   Test setup assuming certain default configuration value.

C. **Unreliable Code**.
   Code handle technically correct but less ideal configuration values poorly.

D. **Code Bugs**.
   Configuration value trigger dormant code bug.

We show in table 5.4 the distribution of different false positive categories for Hadoop Common and HDFS among the parameters we studied, in the form of number of parameters in that category.

Module	**Total**	**A**	**B**	**C**	**D**	**B&C**	**A&C**
Common	19	3	8	8	0	0	0
HDFS	29	0	14	10	1	3	1

Table 5.4: False positive statistics for Hadoop Common and HDFS

5.3.1 Test Timeout

Some tests has a test-level timeout constraint that would result in test failure if not satisfied. The timeout is set by developers based on their knowledge of the max possible (or allowable) duration for the test to run. However, such test-level timeout is independent from configurations related to this test. Changing the configuration value–even to a valid one–could potentially slow down the test run, and result in failing the test due to timeout.

For example, the configuration parameter `ha.zookeeper.session-timeout.ms` has a default value of 10,000, and setting it to 500,000 fails the test `ha.TestZKFailoverController#testNoZK`. The test is shown in Figure 5.10.

```java
@Rule
public Timeout testTimeout = new Timeout(3 * 60 * 1000);

@Test
public void testNoZK() throws Exception {
  stopServer();
  DummyHAService svc = cluster.getService(1);
  assertEquals(ZKFailoverController.ERR_CODE_NO_ZK,
      runFC(svc));
}
```

Figure 5.10: Test snippet for timeout characteristic change

The test is expecting the correct error code when Zookeeper is not running, and has a `@Rule` restricting the execution time. Setting `ha.zookeeper.session-timeout.ms` to 500,000 results in retry connecting for 500,000 ms before reporting connection error, which is longer than the timeout 180,000 for this test class, thus failing the test and resulting in false positive behavior.

5.3.2 Hardcoded Test Code

When writing tests, developers often use hardcoded values for simplicity of test setup. These hardcoded values are normally picked with the assumption of using default configuration values. Once the default configuration is change to anything else, these tests will no longer pass. Thus, even a correct configuration value would result in test failure (i.e., false positives).

In some cases, the test has an assertion that directly expects a default configuration value. An example is with parameter `net.topology.table.file.name`, which we analyzed in Section 5.2.2 for its false negative behavior. Interestingly, the test method, `org.apache.hadoop.net.TestTableMapping#testNoFile`, is the only test associated with this parameter, and happen to result in false positive behavior as well. The test is shown in Figure 5.11

The test is expecting default value for `net.topology.table.file.name`, which is empty (`null`), and asserts that the rack being set is equal to the default value. If, however, the value is not empty, the rack will no longer being set to default value, but to what is specified in the mapping file. Thus, even when we set the parameter to a correct file path that points to a correct mapping file, this test will still fail, as that is not the same as the the `DEFAULT_RACK`.

In other cases, the test has an assert that expects some objects or states whose construction relies on the default configuration value. For example, parameter `io.map.index.skip` is used

```java
@Test
public void testNoFile() {
  TableMapping mapping = new TableMapping();

  Configuration conf = new Configuration();
  mapping.setConf(conf);

  List<String> names = new ArrayList<String>();
  names.add(hostName1);
  names.add(hostName2);

  // resolve() implementation omitted. It returns NetworkTopology.DEFAULT_RACK
  // if net.topology.table.file.name is not set.
  List<String> result = mapping.resolve(names);
  assertEquals(names.size(), result.size());
  assertEquals(NetworkTopology.DEFAULT_RACK, result.get(0));
  assertEquals(NetworkTopology.DEFAULT_RACK, result.get(1));
}
```

Figure 5.11: Test snippet for relying on hardcoded configuration value directly

to specify the number of index entries to skip between each entry in an index file, and has a test `org.apache.hadoop.io.TestMapFile#testMidKey` as shown in Figure 5.12. We looked at this test before in Section 5.2.3, but now we looked at it for a different purpose.

```java
@Test
public void testMidKey() throws Exception {
  Path dirName = new Path(TEST_DIR, "testMidKey.mapfile");
  FileSystem fs = FileSystem.getLocal(conf);
  Path qualifiedDirName = fs.makeQualified(dirName);

  MapFile.Writer writer = null;
  MapFile.Reader reader = null;
  try {
    writer = new MapFile.Writer(conf, fs, qualifiedDirName.toString(),
      IntWritable.class, IntWritable.class);
    writer.append(new IntWritable(1), new IntWritable(1));
    writer.close();

    reader = new MapFile.Reader(qualifiedDirName, conf);
    assertEquals(new IntWritable(1), reader.midKey());
  } finally {
    IOUtils.cleanup(null, writer, reader);
  }
}
```

Figure 5.12: Test snippet for relying on hardcoded configuration value indirectly

The parameter, `io.map.index.skip`, represents the number of index entries to skip between each entry and is default to 0. This test writes an online file and tries to verify that the `midkey` is indeed the only line written. However, if we set the value of `io.map.index.skip` to anything higher than 0, the `midkey` will be `null` as it will be skipped, and the assertion will fail. The specification of this parameter definitely allows a positive value, but the test is written assuming the default value 0, and thus resulting in a false positive behavior.

5.3.3 Unreliable Code

Some of the values we choose are technically correct but not ideal and thus unlikely to be used in production. We analyzed one example in Section 5.3.1. Sometimes however, the consequence of having these less ideal values could be more severe than just changing the test timeout characteristics – sometimes the software will misbehave, potentially crash or hang indefinitely. This is different from having a source code bug, because this exposes reliability issues instead of correctness issues.

```java
private long bytesPerPeriod;
public synchronized void setBandwidth(long bytesPerSecond) {
  bytesPerPeriod = bytesPerSecond*period/1000;
}

public synchronized void throttle(long numOfBytes, Canceler canceler) {
  while (curReserve <= 0) {
    long now = monotonicNow();
    long curPeriodEnd = curPeriodStart + period;

    if ( now < curPeriodEnd ) {
      try {
        wait( curPeriodEnd - now );
      } catch (InterruptedException e) {
        // Handle interrupt omitted.
      }
    } else if ( now < (curPeriodStart + periodExtension)) {
      curPeriodStart = curPeriodEnd;
      curReserve += bytesPerPeriod;
    } else {
      // Discard the prev period omitted.
    }
  }
}
```

Figure 5.13: Simplified code snippet for unreliable implementation

Figure 5.13 shows such an example. The parameter `dfs.image.transfer.bandwidthPerSec` has a logic in code where result of an integer division is rounded to 0, causing no byte getting sent. The consequence is that the test timeout would be reached. As shown in Figure 5.13, bytesPerSecond takes the configured value from dfs.image.transfer.bandwidthPerSec and is used to calculate bytesPerPeriod, which is then used to increment curReserve for exiting the while loop. However, if bytesPerSecond is set to 1 and period takes the default 500, bytesPerPeriod will be rounded to 0, resulting in curReserve never gets incremented, and the while loop hang indefinitely.

While setting throttling bandwidth to 1 bytes per second is abnormal, it is still technically a correct value and we argue that the system should not just hang on this input. The test TestTransferFsImage#testImageUploadTimeout which calls the above logic has a timeout value of 100000, and thus hanging on throttling logic, causing the test to fail and resulting in false positive behavior.

5.3.4 Code Bugs

Sometimes, the reason for having test fail on a correct configuration is simple: there is a bug in source code. Developers did not handle some cases that are tied to a configurable value correctly. As an example, the value of parameter dfs.namenode.audit.loggers should be any implementation of org.apache.hadoop.hdfs.server.namenode.AuditLogger, one of which is org.apache.hadoop.hdfs.server.namenode.top.TopAuditLogger. However setting to TopAuditLogger fails because of the logic described in Figure 5.14 during logger construction.

```java
private List<AuditLogger> initAuditLoggers(Configuration conf) {
  Collection<String> alClasses =
      conf.getTrimmedStringCollection(DFS_NAMENODE_AUDIT_LOGGERS_KEY);
  for (String className : alClasses) {
    try {
      AuditLogger logger;
      if (DFS_NAMENODE_DEFAULT_AUDIT_LOGGER_NAME.equals(className)) {
        logger = new DefaultAuditLogger();
      } else {
        logger = (AuditLogger) Class.forName(className).newInstance();
      }
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
}
```

Figure 5.14: Simplified code snippet for code bugs

The existing implementation for TopAuditLogger does not have a default constructor, thus failing the newInstance() call and the test, resulting in false positive behavior.

5.4 TEST REWRITES

Now that we understand all the test patterns preventing directly reuse existing tests for configuration testing, we make a first-step attempt to rewrite some of the tests, aiming to make them reusable for capturing a misconfiguration while not fail on a correct configuration.

We find that scenarios in Section 5.2.5 and Section 5.2.4 are easier to rewrite. As an example, recall that parameter `fs.trash.checkpoint.interval` mentioned in Section 5.2.5 passes all associated tests on any value even a random string. This is due to the exception handler swallow the exception by only printing out the error. Besides, while the shell operation

```
shell.run(new String[] { "-rm", target_File.toString() });
```

does not finish successfully, the correctness of this test doesn't depend on this shell operation's successful finish, and thus the test doesn't fail. Our goal is thus to make the test fail when the shell operation fail, that is, when this parameter is not set correctly. The approach we take is to check the return status of the shell operation, and assert failure if it is an error (non-zero) status. We do this by modifying the shell execution line to:

```
assertEquals("Shell operation fail.", shell.run(
    new String[] { "-rm", target_File.toString() }), 0);
```

Now this test fails if the parameter is set to a string. However, since there's fall back logic for this parameter when it's smaller than zero, as shown in Figure 5.15, this test then falls into the fallback or auto-correction category (Section 5.2.2), which is harder to rewrite, and one could even argue that further rewriting is not necessary given the presence of an auto-correction logic.

```
// In TrashPolicyDefault#Emptier.
// deletionInterval take the value of parameter fs.trash.interval.
if (emptierInterval > deletionInterval || emptierInterval <= 0) {
  LOG.info("The configured checkpoint interval is " +
           (emptierInterval / MSECS_PER_MINUTE) + " minutes." +
           " Using an interval of " +
           (deletionInterval / MSECS_PER_MINUTE) +
           " minutes that is used for deletion instead");
  this.emptierInterval = deletionInterval;
}
```

Figure 5.15: Code snippet for auto-correction logic

We show another example where we transfer a parameter from "no useful test" category to "useful" category by utilizing the existing but non-reaching validation logic. Recall the parameter `hadoop.http.authentication.type` we looked at in Section 5.2.4, which could take values from `"simple"`, `"kerberos"`, `"ldap"`, `"multi-scheme"`, or a valid classname. The parameter belongs to "no useful test" category if we just directly reuse the associated test, since its test code is using mock (through Java `Mockito` and does not reach any actual valiation logic, passing on any invalid value. A bit more digging reveals that in `TestAuthenticationFilter.java`, there are lots of tests related to the authentication parameters, including this one. A general pattern of these tests is that they use `Mockito` to return some hard-coded value on getting a configuration parameter or calling a function, and then test some operation related to these authentication configuration parameter. For example, `testFallbackToRandomSecretProvider()` has the following code:

```
// AuthenticationFilter.AUTH_TYPE take the value of
    hadoop.http.authentication.type.
FilterConfig config = Mockito.mock(FilterConfig.class);
Mockito.when(config.getInitParameter(AuthenticationFilter.AUTH_TYPE))
   .thenReturn("simple");
```

Figure 5.16: Useful logic in testFallbackToRandomSecretProvider()

A straightforward idea would be to change the `.thenReturn("simple")` to return whatever is in the configuration file, and call the `init()` method (see Section 5.2.4) method, fail the test run if an error is thrown. We take this approach and write a test modeled after the `testFallbackToRandomSecretProvider()` method (about 40 lines), with the core logic shown in Figure 5.17.

However this is not a complete solution: this works for detecting incorrect values, which make the parameter has useful tests, but introduces False Positives. The current rewrite passes `"simple"` and fail on all bogus class names, which is good. But, `"kerberos"`, `"ldap"`, and `"multi-scheme"` also fail, which is an false positive behavior. The reason they fail is because certain setups are not done properly for those authentication types. We acknowledge that setting up the necessary context for these three type to pass test might be non-trivial, and could require hard coding other configuration values.

We see from the above examples that rewritting test to make them effective could take more than one step, and seems to be a manual process in its nature. We call for developer support here, as we belive that developers of the software could easily do such rewrite with these additional validation logic we mined through associating parameters with tests.

```java
  @Test
public void testHTTPAuthenticationType() {
  try {
    FilterConfig config = Mockito.mock(FilterConfig.class);
    Configuration conf = new Configuration();
    Mockito.when(config.getInitParameter(AuthenticationFilter.AUTH_TYPE))
      .thenReturn(conf.get("hadoop.http.authentication.type"));

    // Some other Mockito setup code omitted.

    filter.init(config);
    Assert.assertTrue(true);
  } catch (ServletException ex) {
    LOG.error("hadoop.http.authentication.type is misconfigured!");
    Assert.fail(ex.toString());
  } catch (Exception ex) {
    LOG.error("Something else mysteriously failed!");
    Assert.fail(ex.toString());
  } finally {
    filter.destroy();
  }
}
```

Figure 5.17: Simplified rewritten test snippet for non-reaching validation

CHAPTER 6: FUTURE WORK

6.1 ADDRESSING LIMITATIONS

Our methodology described in chapter 4 has a couple limitations that we would like to address and refine in the near future:

Not handling flaky tests. Flaky test has significant impact on our test running results. Since when a test fail on a presumably correct value, we do not know if it is due to the test being flaky, or the parameter actually exhibits false positive behavior. Currently we rely on a manually maintained blacklist to filter out flaky tests, which reduces the efficiency of our process. Flaky tests has its own line of research [26, 27] and is not the focus of the thesis, but we could use the existing research results to refine our infrastructure.

Not considering dependencies. The methodology we adopted runs associated tests for each configuration parameter in isolation, but in reality multiple configuration parameters could have dependencies. While our current infrastructure does a decent job reporting test results for each parameters without considering dependencies, our manual code inspection reveals a handful of cases where parameter dependencies result in false errors (Section 5.2.7). Dependency in configuration parameters is another line of research [25] as well, and we plan to take it into consideration when we refine our infrastructure.

Potentially biased value generation. Our value generation process (Section 4.2) is manual and error-prone. Ideally the process of generating correct and incorrect values should be automated, or at least more disciplined. We plan to do type inferencing on parameters, divide them into finer-grained groups, and have representative correct and incorrect values per group. Another option is to use values collected from container or VM images [28].

6.2 TOWARDS A STANDARDIZED PRACTICE

We believe configuration testing should be a standardized testing practice just like source code tests, and achieving this vision needs support from the software development community. To incentivize developers to provide support, extensions should be developed to existing unit test framework that allow developers to write unit tests with testing configurations in mind, and make it easy for them to report information such as whether a test could be reused, how to best reuse it, etc. Taking a step forward, we envision developing a new framework for configuration testing, which should allow developers or system administrators to write new tests from scratch specifically for configuration.

CHAPTER 7: CONCLUSION

To conclude, this thesis presents a case study of the Hadoop project, evaluating opportunities and challenges in reusing software source code test for configuration testing. We explain the methodology adopted, and show quantitative analysis on the effectiveness of directly reusing existing tests, with yields an effective rate of 41.7% and 48.8%, for Hadoop Common and HDFS module, respectively. We then give in-depth analysis on why some of the tests cannot be effectively reused, and explore possibilities to make these tests reusable by manually rewriting them.

We believe configuration testing is a missing piece in nowadays software development practices. We hope that the thesis could serve as a first attempt to test configuration with source code, and inspire future research in this topic. Further, we hope developers could view examples in this thesis as an indication of the benefit of as well as the efforts needed from them to write software tests with testing configurations in mind, which is by no means prohibitive.

REFERENCES

[1] "Facebook returns after its worst outage ever," https://www.cnn.com/2019/11/28/tech/instagram-facebook-outage-thanksgiving/index.html.

[2] "Google Cloud Networking Incident #19009," https://status.cloud.google.com/incident/cloud-networking/19009.

[3] "The latest: Twitter blames internal configuration for outage," https://apnews.com/a276c518a59a4cfea702d4d07d11ae2c.

[4] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "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.

[5] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do Not Blame Users for Misconfigurations," in *Proceedings of the 24th ACM Symposium on Operating System Principles (SOSP'13)*, November 2013.

[6] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic Configuration Management at Facebook," in *Proceedings of the 25th ACM Symposium on Operating System Principles (SOSP'15)*, October 2015.

[7] A. Sherman, P. Lisiecki, A. Berkheimer, and J. Wein, "ACMS: Akamai Configuration Management System," in *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)*, Boston, MA, May 2005.

[8] T. Xu and O. Legunsen, "Configuration Testing: Testing Configuration Values as Code and with Code," *arXiv:1905.12195*, July 2019.

[9] V. Garousi and J. Zhi, "A Survey of Software Testing Practices in Canada," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1354–1376, 2013.

[10] J. A. Whittaker, J. Arbon, and J. Carollo, *How Google Tests Software*. Addison-Wesley, 2012.

[11] D. Martin, J. Rooksby, M. Rouncefield, and I. Sommerville, "'Good' Organisational Reasons for 'Bad' Software Testing: An Ethnographic Study of Testing in a Small Software Company," in *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, Minneapolis, MN, 2007.

[12] S. Baset, S. Suneja, N. Bila, O. Tuncer, and C. Isci, "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.

[13] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "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.

[14] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac, "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.

[15] M. Santolucito, E. Zhai, and R. Piskac, "Probabilistic Automated Language Learning for Configuration Files," in *Proceedings of the 28th International Conference on Computer Aided Verification (CAV'16)*, July 2016.

[16] O. Tuncer, N. Bila, S. Duri, C. Isci, and A. K. Coskun, "ConfEx: Towards Automating Software Configuration Analytics in the Cloud," in *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2018.

[17] J. Mickens, M. Szummer, and D. Narayanan, "Snitch: Interactive Decision Trees for Troubleshooting Misconfigurations," in *Proceedings of the 2nd USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SYSML'07)*, April 2007.

[18] R. Potharaju, J. Chan, L. Hu, C. Nita-Rotaru, M. Wang, L. Zhang, and N. Jain, "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.

[19] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, You Have Given Me Too Many Knobs! Understanding and Dealing with Over-Designed Configuration in System Software," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*, August 2015.

[20] Š. Davidovič and B. Beyer, "Canary Analysis Service," *Communications of the ACM*, vol. 61, no. 5, pp. 54–62, May 2018.

[21] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "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.

[22] T. Xu, H. M. Naing, L. Lu, and Y. Zhou, "How Do System Administrators Resolve Access-Denied Issues in the Real World?" in *Proceedings of the 35th Annual CHI Conference on Human Factors in Computing Systems (CHI'17)*, Denver, CO, May 2017.

[23] S. Zhang and M. D. Ernst, "Proactive Detection of Inadequate Diagnostic Messages for Software Configuration Errors," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15)*, Baltimore, MD, July 2015.

[24] T. Xu, V. Pandey, and S. Klemmer, "An HCI View of Configuration Problems," *arXiv:1601.01747*, January 2016.

[25] T. Xu and Y. Zhou, "Systems Approaches to Tackling Configuration Errors: A Survey," *ACM Computing Surveys*, vol. 47, no. 4, July 2015.

[26] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically Detecting Flaky Tests," in *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, Gothenburg, Sweden, June 2018.

[27] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An Empirical Analysis of Flaky Tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*, Hong Kong, November 2014.

[28] T. Xu and D. Marinov, "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.