ECFuzz: Effective Configuration Fuzzing for Large-Scale Systems

Junqiang Li
University of Electronic Science and Technology of China
Chengdu, China
lijunqiang@std.uestc.edu.cn

Senyi Li
University of Electronic Science and Technology of China
Chengdu, China
lisy@std.uestc.edu.cn

Keyao Li
University of Electronic Science and Technology of China
Chengdu, China
likeyao@std.uestc.edu.cn

Falin Luo
University of Electronic Science and Technology of China
Chengdu, China
luofalin@std.uestc.edu.cn

Hongfang Yu*
University of Electronic Science and Technology of China
Chengdu, China
yuhf@uestc.edu.cn

Shanshan Li
National University of Defense Technology
Hunan, China
shanshanli@nudt.edu.cn

Xiang Li
National Key Laboratory of Science and Technology on Information System Security
Beijing, China
ideal_work@163.com

ABSTRACT

A large-scale system contains a huge configuration space because of its large number of configuration parameters. This leads to a combination explosion among configuration parameters when exploring the configuration space. Existing configuration testing techniques first use fuzzing to generate different configuration parameters, and then directly inject them into the program under test to find configuration-induced bugs. However, they do not fully consider the complexity of large-scale systems, resulting in low testing effectiveness. In this paper, we propose ECFuzz, an effective configuration fuzzer for large-scale systems. Our core approach consists of (i) Multi-dimensional configuration generation strategy. ECFuzz first designs different mutation strategies according to different dependencies and selects multiple configuration parameters from the candidate configuration parameters to effectively generate configuration parameters; (ii) Unit-testing-oriented configuration validation strategy. ECFuzz introduces unit testing into configuration testing techniques to filter out configuration parameters that are unlikely to yield errors before executing system testing, and effectively validate generated configuration parameters. We have conducted extensive experiments in real-world large-scale systems including HCommon, HDFS, HBase, ZooKeeper and Alluxio. Our evaluation shows that ECFuzz is effective in finding configuration-induced crash bugs. Compared with the state-of-the-art configuration testing tools including ConfTest, ConfErr and ConfDiagDetector, ECFuzz finds 60.3–67 more unexpected failures when the same 1000 testcases are injected into the system with an increase of 1.87x–2.63x. Moreover, ECFuzz has exposed 14 previously unknown bugs, and 5 of them have been confirmed.

KEYWORDS

Configuration, Large-Scale Systems, Testing, Fuzzing

ACM Reference Format:
Junqiang Li, Senyi Li, Keyao Li, Falin Luo, Hongfang Yu, Shanshan Li, and Xiang Li. 2024. ECFuzz: Effective Configuration Fuzzing for Large-Scale Systems. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3623315

1 INTRODUCTION

Configuration is another program input that directly controls program behavior like input data. Configuration usually has two manifestations: the command-line option and configuration file, as shown in Figure 1. Among them, each configuration-related option is called a configuration parameter. Figure 1(a) shows the configuration parameters when using the greybox fuzzer AFL to fuzz the program Xmllint. Each configuration parameter represents a part of the program's functionality. For example, "-i in" represents the path of the initial seed fed to the program under test (PUT) Xmllint. Figure 1(b) shows some configuration parameters of the default configuration file in the large-scale system software HBase, which uses the XML file format to configure different functions. Each configuration parameter is contained within a *<property>* element and

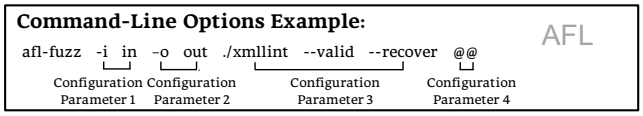
*Hongfang Yu is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

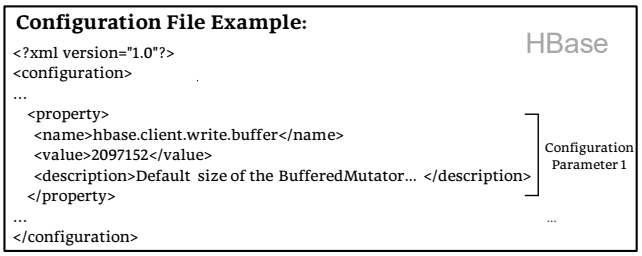
ICSE '24, April 14–20, 2024, Lisbon, Portugal

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3623315



(a) The command-line option



(b) The configuration file

Figure 1: An example to show configurations.

includes three attributes: <*name*>, <*value*> and <*description*>, representing the name, value and description of the configuration parameter, respectively.

To test whether different configuration parameters can cause large-scale system crashes, a straightforward approach is to generate all possible values for all configuration parameters in the large-scale system. However, on the one hand, there are usually hundreds of configuration parameters in large-scale systems (e.g., HDFS v2.9.2 contains 431 configuration parameters [8]); On the other hand, most configuration parameters have flexible constraints (e.g., the configuration parameter $hbase.client.write.buffer$ in HBase represents the size of the write buffer. The valid value is a positive integer and its constraint is $[0, INT\_MAX]$). Therefore, this leads to the problem of combinatorial explosion among configuration parameters when exploring the configuration input space. **To improve the effectiveness of configuration testing, the key is to select a small set of representative configuration parameters as testcases [33]**. Two factors need to be considered to achieve this goal: 1) Configuration parameter generation: generate high-quality configuration parameters as testcases to avoid being filtered out in advance during the startup stage of the PUT and to explore more code during the program running stage; 2) Configuration parameter validation: an effective testcase validation method can quickly validate whether the generated testcases help find errors and avoid wasting a lot of time in testcases that are unlikely to yield errors during system testing.

However, existing configuration testing techniques do not fully consider the complexity of large-scale systems, resulting in low testing effectiveness. There are the following two challenges:

1. **How to effectively generate configuration parameters?** Existing configuration testing techniques generally use fuzzing to generate configuration parameters. Some work [15, 29, 36, 40] uses dictionary-based or grammar-based fuzzing techniques to generate configuration parameters that satisfy grammatical validity; other work [2, 14, 17–19, 32, 38] uses constraint-based fuzzing techniques to generate configuration parameters that violate configuration constraints. However, in most work, each fuzzing campaign only focuses on generating a single configuration parameter independently. In the large-scale system, there are usually thousands of configuration parameters, and each configuration parameter is associated with a part of the program code. If only a single configuration parameter is considered in each fuzzing campaign, it means that only limited code related to the configuration parameter can be fuzzed each time. Moreover, in the large-scale system, in addition to the constraints of a single configuration parameter, there are also dependencies between different configuration parameters. For example, configuration parameter *B* can only take effect when configuration parameter *A* takes effect. Ignoring dependencies will result in invalid configuration parameters.

2. **How to effectively validate the generated configuration parameters?** Existing configuration testing techniques often directly inject the generated configuration parameters into the PUT to execute system testing and then monitor the status of the PUT to validate whether the configuration parameters can trigger bugs. In small-scale programs, the time cost of the method is acceptable because of the fast execution speed of system testing. However, in a large-scale system, the time cost of executing a system testing is very high. Once a configuration parameter that is unlikely to yield errors has been generated, it often takes a lot of time to test it systematically. This waste of resources is especially serious in the large-scale system. Therefore, if the effect of the generated configuration parameter can be validated quickly in advance, configuration parameters that are unlikely to yield errors will be filtered out before executing system testing.

In this paper, we propose ECFuzz, an effective configuration fuzzer for large-scale systems. Specifically, ECFuzz addresses the above challenges as follows.

Multi-dimensional configuration generation strategy. To effectively generate configuration parameters in large-scale systems, we propose a multi-dimensional configuration generation strategy. First, ECFuzz collects the dependencies among the configuration parameters in the large-scale system and designs different mutation strategies according to different dependencies. Then, ECFuzz selects multiple configuration parameters from the candidate configuration parameters and uses the designed mutation strategy in turn for each. Finally, ECFuzz intelligently adjusts the number of configuration parameters according to the testing results.

Unit-testing-oriented configuration validation strategy. Mature large-scale systems have relatively complete unit tests to check whether the program unit can correctly implement the functions, performance, interfaces and constraints in the detailed design specification. Previous work [25] has shown that configuration parameters in a large-scale system can be associated with its unit tests and establish an internal relationship between configuration and code. Also, unit testing is usually executed faster than system testing. Therefore, ECFuzz introduces unit testing into configuration testing techniques to execute effective configuration validation. The key idea is to run different unit tests corresponding to the generated configuration parameters before executing system testing. If a certain configuration parameter causes the unit tests to fail, ECFuzz will inject the configuration parameter into the system testing to validate whether it can trigger system-level bugs. Otherwise, ECFuzz considers this configuration parameter unlikely to find bugs and filters it out before executing system testing.

In summary, the main contributions of this paper are as follows:

- We propose a multi-dimensional configuration generation strategy, which uses dependency analysis to choose a reasonable set of parameters to fuzz within a configuration.
- We propose a unit-testing-oriented configuration validation strategy, which uses unit tests as a proxy to system tests to do an initial testcase reduction to weed out parameter configurations unlikely to yield errors.
- We implement a prototype of ECFuzz and evaluate it on five widely used large-scale systems, including HCommon, HDFS, HBase, ZooKeeper and Alluxio. Our evaluation shows that ECFuzz is effective in finding configuration-induced crash bugs. Specifically, ECFuzz has exposed 14 previously unknown configuration-induced bugs and 5 of them have been confirmed. To promote the development of configuration testing, we publicly release the effective configuration fuzzer ECFuzz[1] as an open-source software.

2 BACKGROUND

2.1 Configuration Testing

Figure 2 shows the workflow of configuration testing techniques. During the process of configuration testing, the program under test (PUT) receives two kinds of input: configuration input and test input. Then, configuration validation is performed on this configuration input. The system monitor monitors the status of the PUT to determine whether a bug is found.

Configuration testing mainly focuses on how to generate more effective configuration inputs as testcases and validate these testcases. It is divided into two steps: 1) Generate configuration parameters; 2) Validate generated configuration parameters.

For the former, existing configuration testing techniques use fuzzing to generate representative configuration parameters. Because configuration usually has a highly structured data structure, some work (e.g., ConfigFuzz [40], TOFU [29], POWER [15]) uses dictionary-based or grammar-based fuzzing techniques to generate valid configuration parameters that conform to the configuration grammar format. Another type of work is configuration error injection testing (CEIT) [2, 14, 17–19, 32, 38], which uses constraint-based fuzzing to analyze the constraint relationship of each configuration parameter in advance and generate configuration parameters that violate these constraints (that is, misconfigurations) to test the PUT reaction under these misconfigurations.

For the latter, existing configuration testing techniques directly inject the generated configuration parameters into the PUT for system testing and monitor the status of the PUT to validate whether these configuration parameters can find bugs.

However, the above techniques do not fully consider the complexity of large-scale systems, leading to ineffective testing.

2.2 The Complexity of Large-Scale Systems

Compared with traditional small-scale programs, the complexity in large-scale systems is more obvious, mainly reflected in two aspects: the relational complexity of configuration parameters and the time complexity of system testing.

[1]https://github.com/ecfuzz/ECFuzz

Table 1: Description of dependency relationships.

Type	Description	Example
Control Dependency	The second parameter only works if the value of the first parameter is *True*.	hbase.data.umask.enable hbase.data.umask
Value Relationship Dependency	The value of one parameter is constrained by another parameter.	alluxio.master.worker.threads.max alluxio.master.worker.threads.min
Overwrite Dependency	The second parameter overwrites the first parameter.	dfs.client.failover.connection.retries ipc.client.connect.max.retries
Default Value Dependency	If the value of the first parameter is not available, the value of the second parameter will be used as its default value.	hbase.client.pause hbase.client.pause.cqtbe
Behavioral Dependency	The first parameter works with the second parameter together on some behavior of the system.	fs.ftp.host fs.ftp.host.port

Table 2: Comparison of test execution speeds of different scale programs (Unit: *Execs/Sec*).

Small-Scale Programs			Large-Scale Systems		
Program	Format Type	System Testing	Program	System Testing	Unit Testing
Mjs	JavaScript	2167	HCommon	0.039	2.89
Jasper	JPEG	4833	HDFS	0.023	0.39
Swftophp	SWF	2197	HBase	0.015	0.36
Xmllint	XML	2543	ZooKeeper	0.10	0.66
Readelf	ELF	1787	Alluxio	0.0085	6.08
Average	-	2705.4	-	0.0371	2.0760

The relational complexity of configuration parameters. The configuration parameters of large-scale systems have complex relationships, including two types: constraint relationships and dependency relationships. Among them, constraint relationships refer to the conditions that a single configuration parameter should meet, e.g., when the administrator tries to rewrite a certain configuration parameter in the configuration file, he should follow the rules specific to this parameter, such as the file path or boolean option [19]. At present, there have been many works [7, 17, 19] summarizing constraints of configuration parameters. Dependency relationships refer to conditions that should be satisfied between different configuration parameters. cDEP [8] summarizes five typical dependencies in large-scale systems, as shown in Table 1.

However, existing configuration testing techniques do not take these dependencies into account, and often spend a lot of time generating invalid testcases. For example, there is a control dependency between the parameter *hbase.data.umask.enable* and the parameter *hbase.data.umask* in program HBase, i.e., the latter parameter takes effect only when the previous parameter is *true*. Therefore, when the parameter *hbase.data.umask* is tested, the corresponding dependency parameter *hbase.data.umask.enable* needs to be set to *true*. To generate configuration parameters more effectively, we need to design corresponding mutation strategies for different dependency relationships.

The time complexity of system testing. Due to the extensive scale of large-scale systems, the time cost for system testing is high. To show the speed gap between different scale programs, we selected five representative programs, respectively.

The results are shown in Table 2. Among them, for small-scale programs, the system testing tool we use is AFL [35]. For the large-scale system, the system testing tool is CeitInspector [18], and the unit testing tool is ctest [25]. It is worth noting that CeitInspector itself does not support testing the above large-scale systems, and we have expanded it. We find that the time cost of system testing in large-scale systems is very expensive, and its average speed is several orders of magnitude slower than small-scale programs. The

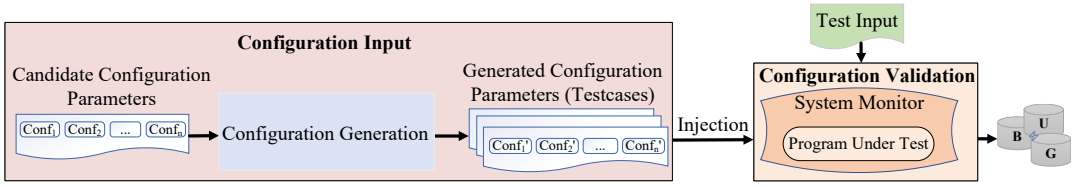


Figure 2: Workflow of configuration testing.

average number of testcases executed per second in small-scale programs is 2705.4. While in system testing for large-scale systems, the execution speed is 0.0371. In other words, the time spent by system testing for large-scale systems executing one testcase can make small-scale programs execute more than 72K testcases. Therefore, once a configuration parameter that is unlikely to yield errors has been generated, lots of time for system testing will be wasted. Fortunately, the speed of unit testing is acceptable compared to system testing. We can use the faster unit testing results as a guide to filter out some configuration parameters that are unlikely to yield errors in advance before executing system testing.

3 ECFUZZ

3.1 Overview

The basic workflow and main components of ECFuzz are shown in Figure 3. Like the process of configuration testing techniques, ECFuzz is mainly divided into two parts: *generate configuration parameters* and *validate generated configuration parameters*.

Generate configuration parameters: ECFuzz takes the default configuration file and dependency table as initial inputs, and then uses configuration generation to generate new configuration parameters as testcases. In order to show the process of ECFuzz more clearly, we first explain some keywords, as shown below.

- **Default configuration file.** The default configuration file is usually provided by the official and contains the default values of most configuration parameters. Figure 1(b) shows some configuration parameters of the default configuration file in the large-scale system HBase, which uses the XML file format to configure different program functions.
- **Dependency relationships table**. The dependency relationships table includes dependencies between different configuration parameters. ECFuzz uses the dependency relationships table analyzed by cDEP (as shown in Table 1) and designs the corresponding mutation strategy for each dependency relationship.
- **Seed**. In this paper, a seed refers to a collection of configuration parameters to be tested during each iteration. That is, in this iteration, ECFuzz considers these configuration parameters as interesting. ECFuzz mutates the configuration parameters in the seed to get a new testcase.
- **Testcases**. Unlike traditional fuzzing techniques, in configuration testing, the testcases represent a collection of configuration parameters. The generated testcases will be injected into the PUT to test the PUT reaction under these configuration parameters.

Configuration generation includes seed generation (§3.2) and smart mutation (§3.3). The default configuration files of large-scale systems often have hundreds of configuration parameters, and it is not possible to test all of them in one fuzzing campaign. Therefore, ECFuzz first uses seed generation to select the configuration parameters that need to be fuzzed in this fuzzing campaign as seeds and then uses smart mutation to mutate these configuration parameters to obtain testcases. Specifically, consider that the default configuration file $Conf$ contains n configuration parameters, i.e., $Conf$=<$Conf_1, Conf_2, ..., Conf_n$>. The purpose of seed generation is to select k ($0 < k \leq n$) configuration parameters from $Conf$ as a seed, i.e., seed=<$Conf_1, Conf_2, ..., Conf_k$>. And the purpose of smart mutation is to mutate these configuration parameters to obtain testcases, i.e., testcases=<$Conf_1', Conf_2', ..., Conf_k'$>.

Validate generated configuration parameters: After generating testcases, ECFuzz automatically obtains the corresponding unit tests through the mapping relationship between configuration parameters and unit tests. Specifically, consider that k configuration parameters correspond to M unit tests in a fuzzing campaign, i.e., <$Unit\_Test\_1, Unit\_Test\_2, ..., Unit\_Test\_M$>. In configuration validation (§3.4), ECFuzz executes the corresponding M unit tests and monitors their running results in real time. Once a failure occurs, ECFuzz terminates the unit tests and sets the result of unit tests to fail. If all unit tests are finished and no failure occurs, the result of unit tests is set to success.

The testcase that fails to pass the unit tests is put into the system testing. Meanwhile, to avoid missing the case that the unit tests run successfully but the result of actual system testing still contains exceptions, the testcase is also put into the system testing with a certain probability when the unit tests run successfully. If the system testing still fails, a potential error has been found.

To better demonstrate the design details of ECFuzz, these three components will be described in detail later in this section.

3.2 Seed Generation

In configuration fuzzing, a seed is a set of configuration parameters. It indicates which configuration parameters are worth spending energy to mutate in the fuzzing campaign. We randomly select several configuration parameters from the default configuration file, and then search related configuration parameters according to the dependency relationships table, as shown in Algorithm 1.

Dependency relationships table. §3.1 describes the types of dependencies in the dependency relationships table analyzed by cDEP[2]. Here, we describe the dependency relationships table itself T in the form of a list of tuples, i.e., T=[($Type$), ($Conf_A$), ($Conf_B$)]. Among them, $Type$ represents the types of dependencies, and $Conf_A$ and $Conf_B$ represent two configuration parameters satisfying this relationship. For example, if there is a control dependency between

[2]https://github.com/xlab-uiuc/cdep-fse-ae/blob/master/cDep_result/intra.csv

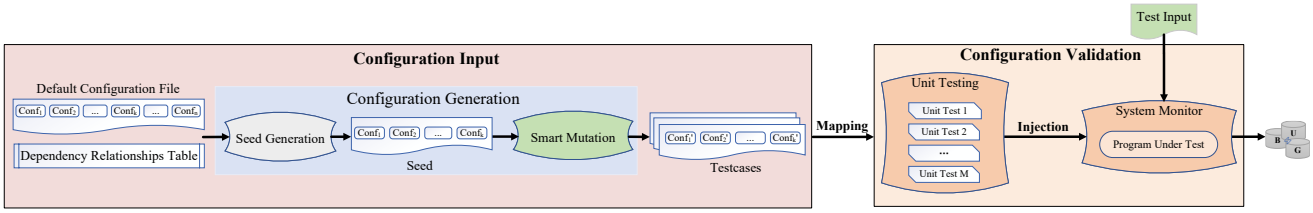


Figure 3: Overview of ECFuzz.

Algorithm 1: Seed Generation

input : The default configuration file C, dependency relationships table *Dependency*
output : A new generated seed S

```
1  k ← randint(3, 6);
   /* Random selection                                  */
2  C_s ← randomChoice(C, k);
   /* search related configuration parameters           */
3  for c in C_s do
4      for cr in Dependency[c] do
           /* avoid repeated adding                     */
5          if not cr in C_s then
6              add(C_s, cr);
7          end
8      end
9  end
10 S ← newSeed(C_s);
```

Algorithm 2: Smart Mutation

input : Dependency relationships table *Dependency*, constraint relationships table *Constraint*, a seed that need to be mutated *Seed*, the number of mutations N
output : A new generated testcase *Testcase*

```
1  Testcase ← ∅;
2  for ConfItem in 1..len(Seed) do
3      add ConfItem to Testcase;
4  end
5  for t in 1..N do
6      index ← randomChoiceIndex(len(Seed));
7      if Seed[index] ∈ Dependency then
8          newValue ← GenValue(Dependency, Seed[index]);
9      else
10         newValue ← GenValue(Constraint, Seed[index]);
11     end
12     Testcase[index] ← newValue;
13 end
```

parameters *hbase.data.umask.enable* and *hbase.data.umask*, a row in the dependency table T can be expressed as: [(*Control Dependency*), (*hbase.data.umask.enable*), (*hbase.data.umask*)].

Random selection. The random selection strategy randomly selects k configuration parameters from the default configuration file (Line 1–Line 2). Among them, the value of k is related to the strategy of smart mutation. We analyze it in Section 3.3.

Search related configuration parameters. To ensure the quality of the seeds, we need to add the corresponding dependency configuration parameters according to their dependencies after selecting the configuration parameters (Line 3–Line 9). Specifically, ECFuzz traverses randomly selected configuration parameters C_s (Line 3). If a related configuration parameter *cr* is found in the dependency relationships table (Line 4) and it is the first time it appears in the C_s (Line 5), the configuration parameter *cr* is added to the C_s (Line 6). Finally, we select the above configuration parameters to construct a new seed (Line 10).

3.3 Smart Mutation

The smart mutation algorithm fully considers the dependencies and constraints corresponding to the configuration parameters in seeds, as shown in Algorithm 2.

During the smart mutation, ECFuzz first initializes the testcase (Line 1–Line 4). Then, the smart mutation begins according to the number of mutations (Line 5). Before each mutation starts, the index of one of the configuration parameters is randomly selected as the mutation object (Line 6). Subsequently, ECFuzz determines whether the configuration parameter has the corresponding dependencies (Line 7). If exists, the *newValue* is obtained by mutation according to the dependencies (Line 8). Otherwise, the *newValue* is obtained by mutation according to the constraints (Line 10). Finally, the *newValue* overrides the initial value corresponding to the testcase (Line 12). In the above process, generating *newValue* based on dependencies and constraints is the key point of the entire algorithm.

3.3.1 Generate a new configuration parameter value according to dependencies. ECFuzz uses the dependency relationships table analyzed by cDEP [8], as shown in Table 1. cDEP supports 5 types of dependencies, including control dependency, value relationship dependency, overwrite dependency, default value dependency and behavioral dependency. In practice, we find that the overwrite dependency appears only 3 times in the table, thus ECFuzz does not consider the overwrite dependency. According to the characteristics of different dependency relationships, we design the corresponding mutation methods.

Given two configuration parameters $Conf_A$ and $Conf_B$, we use different mutation methods for the four supported dependencies.

Control dependency: The second parameter only works if the value of the first parameter is *True*.

Code Patterns. The following code snippet shows a typical example of control dependency, in which *intervals* (storing the value of

the second parameter *rpc.metrics.percentiles.intervals*) only takes effect when *rpcQuantileEnable* (storing the value of the first parameter *rpc.metrics.quantile.enable*) is *true*.

```
if ( rpcQuantileEnable ) {
  rpcQueueTimeMillisQuantiles = new MutableQuantiles[ intervals ];
  for (int i = 0; i < intervals; i++) { ... }
}
```

Mutation method. In the dependency relationships table, which of $Conf_A$ or $Conf_B$ is the first parameter is not given. Therefore, ECFuzz determines it by judging whether the type of $Conf_A$ and $Conf_B$ is *bool*. If neither of them is *bool*, ECFuzz skips this record. Otherwise, if the configuration parameter of the current mutation is the first parameter, ECFuzz will mutate according to its constraints; if it is the second parameter, ECFuzz will first set the value of the first parameter to *True*, and then mutate the configuration parameter according to its constraints.

Value relationship dependency: The value of one parameter p is constrained by another parameter q. There are three kinds of such constraints: (i) Numeric. For example, the value of p is greater than the value of q. (ii) Logical. For example, the value of p is *true* or $false$ depending on the value of q. (iii) Set. For example, the value of $p \subseteq$ the value of q. The unknown bug Alluxio-17509 in §4.4 shows a typical example of set value relationship dependency.

Code Patterns. The following code snippet shows a typical example of numeric value relationship dependency, in which the value of *alluxio.master.worker.threads.min* is less than the value of another parameter *alluxio.master.worker.threads.max*.

```
mMinThreads=conf.getInt( "alluxio.master.worker.threads.min" );
mMaxThreads=conf.getInt( "alluxio.master.worker.threads.max" );
Preconditions.checkArgument (
      mMaxThreads >= mMinThreads, ...);
```

Mutation method. In this case, ECFuzz mutates $Conf_A$ and $Conf_B$ according to their constraints at the same time.

Default value dependency: If the value of the first parameter is not available, the value of the second parameter will be used as its default value.

Code Patterns. The following code snippet shows a typical example of default value dependency, in which *editDirs* (storing the value of the first parameter $dfs.namenode.edits.dir$) is empty (i.e., not set), the value of the second parameter $dfs.namenode.name.dir$ is returned.

```
public static List <URI> getNamespaceEditsDirs(...) {
    if ( editsDirs .isEmpty()) {  //dfs.namenode.edits.dir
      return getStorageDirs(conf, "dfs.namenode.name.dir" );
    }
}
```

Mutation method. To avoid the invalid mutation caused by only mutating the second parameter when the first parameter is available, ECFuzz directly mutates one of the configuration parameters according to its constraints and then sets the other configuration parameter to the same value.

Behavioral dependency: The first parameter works with the second parameter together on some behavior of the system.

Code Patterns. The following code snippet shows a typical example of behavioral dependency, in which both parameters $fs.ftp.host$ and $fs.ftp.host.port$ determine the connected *socket*.

```
String host = conf.get( "fs.ftp.host" );
int port = conf.getInt( "fs.ftp.host.port" );
client.connect(host, port);
```

Mutation method. In this case, ECFuzz mutates $Conf_A$ and $Conf_B$ according to their constraints at the same time.

3.3.2 Generate a new configuration parameter value according to constraints. To obtain the constraints of configuration parameters, we first carefully classify the types of configuration parameters, and then design corresponding matching methods according to the characteristics of different types [19, 25]. According to the types and characteristics of configuration parameters, we use regular matching to infer each type. For example, to determine whether a configuration parameter is a type *IpPortAddr*, the regular expression we designed is $\hat{}\backslash d\{1,3\}\backslash.\backslash d\{1,3\}\backslash.\backslash d\{1,3\}\backslash.\backslash d\{1,3\}:\ \backslash d+\$$. Finally, ECFuzz extracts the corresponding configuration constraints according to the configuration parameter types and randomly generates a value from the value range of extracted constraints.

3.3.3 Discussion. For the smart mutation, the number of mutations N significantly affects its performance. If N is small, it is difficult to find bugs caused by multiple configuration parameters. In special cases, such as N is 1, the smart mutation algorithm degenerates into the single mutation algorithm. If N is large, it will greatly increase the time cost and difficulty of mutation. Bugs caused by a large number of configuration parameters together are relatively rare. In HCommon, HDFS, HBase, ZooKeeper and Alluxio, a configuration change modifies 6 parameters on average according to [8]. Furthermore, large N will lead to redundancy in the testing result. More effort is required to pinpoint which of the configuration parameter is the root cause of the bug. Therefore, we find that the range of 3–6 is more appropriate when using stacking mutations.

To address the above challenges, we design a smart mutation strategy. Specifically, we first fuzz the PUT using the more efficient single mutation algorithm. When a new unexpected failure of the PUT cannot be found for some time, we automatically switch to the stacking mutations algorithm to improve the quality of configuration parameters in the testcase.

3.4 Configuration Validation

In configuration validation, ECFuzz builds the running environment for the program under test (PUT), including the *unit testing stage* and *system testing stage*, as shown in Algorithm 3.

3.4.1 Unit Testing Stage. The unit testing stage quickly tests the bug-triggering potential of testcases by searching for and executing unit tests associated with the testcases. The unit testing stage consists of the following three parts:

Unit testing search. According to the mapping relationship of ctest [25], ECFuzz searches out the parts only related to the configuration parameters in the testcases from a large number of unit tests provided by the PUT as the candidate unit test set.

Unit testing reduction. Numerous configuration parameters in a testcase will result in a huge candidate unit test set obtained by the unit testing search. This may make it run longer than full system testing. Therefore, ECFuzz uses sampling and time filtering algorithms to compress the size of the candidate unit test set.

Algorithm 3: Configuration Validation

```
   input : A testcase T, unit testing mapping U_M, probability
           of compulsorily executing system testing P_ces
   output: Crashing testcase T_c
1  T_c ← ∅;
2  ▷ Unit Testing Stage:
   // unit testing search
3  T_ut ← search(U_M, T);
   // unit testing reduction
4  reduction(T_ut);
   // unit testing execution
5  S_u ← runTest(T_ut);
   /* skip system testing with a probability when
      all unit tests succeed                        */
6  if isSuccessful(S_u) then
7      if random(0, 1) > P_ces then
8          return;
9      end
10 end
11 ▷ System Testing Stage:
   // system environment construction
12 T_sys ← constructSystemTest(T);
   // configuration file replacement
13 Conf ← generateConfigurationFile(T);
   // system testing execution
14 S_s ← runTest(T_sys, Conf);
15 if containUnexpectedFailures(S_s) then
16     add T to T_c;
```

Specifically, the sampling algorithm samples a part of the candidate unit test set according to a certain proportion to replace the original set. The time filtering algorithm detects the execution time of each unit test and filters out unit tests that take too long according to the prior knowledge from [9].

Unit testing execution. After obtaining the reduced set of candidate unit tests, ECFuzz constructs the running environment required to execute the unit tests. The result of the unit test stage is successful only if the testcase passes all unit tests. It is worth noting that ECFuzz does not need to run all the unit tests every time. Once the testcase fails a unit test, the unit testing stage is terminated and the unit testing result is set to failure. Moreover, the result of the unit testing stage is returned to the system testing stage regardless of whether the unit tests pass or not, but the testcase is saved for later analysis only if the unit tests fail.

3.4.2 System Testing Stage. After running the unit tests, ECFuzz enters the system testing stage where it further tests whether the testcase can trigger the bug in the real system environment. The system testing stage consists of the following three parts:

System environment construction. ECFuzz constructs the corresponding system compilation environment, running environment and test input according to the type of PUT.

Configuration file replacement. Before starting the system testing, ECFuzz converts the configuration parameter information in the testcase into a complete configuration file that meets the input requirements of the large-scale systems. The new configuration file is used to replace the default configuration file as the new input for the PUT.

System testing execution. ECFuzz creates processes to start system tests and constantly monitors the status of testing execution. Specifically, system testing monitors the PUT from three dimensions according to the running phase of the program: startup exception, runtime exception and shutdown exception. 1) Startup exception: The program cannot be fully started in the early stage. 2) Runtime exception: Program crashes, hangs or throws exceptions. 3) Shutdown exception: The program cannot be closed normally.

We argue that it is a normal reaction if the system has a startup exception after injecting some configuration parameters because it means that the system can check these configuration parameters immediately before the configurations are put online for production [31]. Therefore, we regard the runtime exception and shutdown exception as *unexpected failures*. Once a testcase triggers unexpected failures, we consider that a potential error has been found and record it for subsequent reproduction and analysis (Line 15–Line 16). It is worth noting that the runtime exception includes various types of exceptions, such as *DeadlineExceededException*. Each type of exception can represent a unique runtime exception found. Therefore, we count the types of exceptions as one of the effectiveness metrics.

4 EVALUATION

Implementation. ECFuzz is implemented with 5902 lines of Python code and 1701 lines of Java code. The Python code is used to implement the entire fuzzing process, including seed generation, smart mutation and configuration validation. The Java code is used to construct test inputs and monitor the internal state of the program under test (PUT).

In configuration generation, ECFuzz takes the default configuration file and dependency relationships table as initial inputs. Among them, ECFuzz obtains the default configuration file from the official website of the PUT and uses the dependency relationships table in cDEP [8]. And then ECFuzz designs different mutation strategies according to the characteristics of different dependency relationships.

In configuration validation, ECFuzz obtains the corresponding unit tests through the mapping relationship between configuration parameters and unit tests. In our implementation, we adopt the mapping relationship in ctest [25]. In the system testing stage, we construct corresponding test suites for each PUT as test inputs, including the process startup testing, functional completeness testing and process shutdown testing. Moreover, in order to comprehensively monitor the PUT at the system level, we have designed two levels of monitoring solutions, including the internal monitor and external monitor.

- **Internal monitor** refers to monitoring the internal state of the PUT under test inputs after injecting testcases. If an exception is found in the PUT, including startup exception, runtime exception and shutdown exception, the exception is thrown by the internal monitor.
- **External monitor** obtains the exceptions thrown from the internal monitor and parses them to count these exceptions. Moreover, during system testing, it monitors the memory

Table 3: Details of large-scale systems under test.

Program	Description	LoC	#Param.
HCommon (2.8.5)	Hadoop core lib/runtime	253K	272
HDFS (2.8.5)	Distributed file system	587K	297
HBase (2.2.2)	Distributed database	856K	205
ZooKeeper (3.5.6)	Distributed coordination	150K	17
Alluxio (2.1.0)	In-memory storage	365K	360

1. LoC indicates total lines of codes.
2. #Param. indicates total number of configuration parameters we used.

and CPU of the operating system and logs of PUTs. If an exception is a runtime exception or shutdown exception, the external monitor treats it as an unexpected failure.

Experimental setup. To evaluate the performance of our fuzzing system for large-scale systems, we choose five mature and widely-used open-source systems: HCommon, HDFS, HBase, ZooKeeper and Alluxio as PUTs. The details are shown in Table 3. These software systems are widely studied and have been adopted by previous excellent work [25].

To ensure fairness, all the experiments run on a KVM virtual machine configured with 4-core 4G memory and the OS of 64-bit Ubuntu 16.04 LTS.

When evaluating ECFuzz, we set the following default values. For the number of mutations N in Algorithm 2, we randomly choose an integer from 3-6. For the probability of compulsorily executing system testing P_{ces} in Algorithm 3, we set it to 0.1. For unit testing reduction in the unit testing stage, we set the sampling algorithm to randomly sample 20% of all unit tests and set the time filtering algorithm to run no more than 1s for a single unit test and no more than 15s for all unit tests in total. We also conducted each experiment for 12 hours and repeated each experiment 5 times to determine the statistical significance of the results.

Metrics. We mainly target the following effectiveness metrics for evaluation:

- **Quality of testcases:** the average unexpected failures found per injected system testcase.
- **Types of exceptions:** the total types of exceptions found in the runtime exception.
- **Bug counting:** the total unknown bugs that ECFuzz could find in large-scale systems.

Research questions. We answer four research questions:

RQ1: How does ECFuzz's multi-dimensional configuration generation strategy compared to the single mutation?

RQ2: How effective is ECFuzz's unit-testing-oriented configuration validation strategy in improving the quality of testcases?

RQ3: How is the quality of testcases and types of exceptions by ECFuzz compared to the state-of-the-art configuration testing techniques?

RQ4: How effective is ECFuzz in exposing unknown bugs compared to the state-of-the-art configuration testing techniques?

4.1 RQ1: Multi-Dimensional Configuration Generation Strategy Evaluation

To effectively generate configuration parameters, ECFuzz designs a multi-dimensional configuration generation strategy including seed generation and smart mutation. To show the effectiveness of the multi-dimensional configuration generation strategy, we mainly compare it with the strategy based on single mutation. The single mutation refers to selecting only a single configuration parameter from candidate configuration parameters without considering dependencies during each fuzzing campaign. We denote this single mutation strategy as *ECFuzz-S*. Table 5 and Table 4 show the results of the quality of testcases and types of exceptions for different strategies, respectively.

Table 4: Types of exceptions for multi-dimensional configuration generation strategy evaluation.

Program	ECFuzz-S	ECFuzz	ECFuzz vs. ECFuzz-S	Unique Exceptions
HCommon	1	2	1	-
HDFS	1	2	1	SafeModeException
HBase	2	4	2	TableNotDisabledException NoNodeException
ZooKeeper	1	2	1	SessionExpiredException NodeExistsException
Alluxio	2	6	4	ResourceExhaustedException IllegalStateException InvalidPathException RuntimeException Exception NativeIoException
Total	7	16	9 (**128.6%**↑)	*Union*

1. Unique exceptions indicate unique types of exceptions only found by ECFuzz. Note that the data is the average value of all five results. Therefore, we use the Equation $U_1 - U_2$ to count unique exceptions, where U_1 and U_2 represent the union of types of exceptions that ECFuzz and ECFuzz-S find in all five results, respectively.
2. *Union* indicates a union of unique exceptions, including SafeModeException, TableNotDisabledException, NoNodeException, SessionExpiredException, NodeExistsException, ResourceExhaustedException, IllegalStateException, InvalidPathException, RuntimeException, Exception and NativeIoException.

Answer to RQ1. For the quality of testcases, we find that ECFuzz has a significant improvement compared to ECFuzz-S. Specifically, the total quality of testcases of ECFuzz-S and ECFuzz are 20.6‰ and 92.5‰, respectively. This means that ECFuzz finds 71.9 more unexpected failures when the same 1000 testcases are injected into the system. For types of exceptions, ECFuzz finds multiple unique exceptions that ECFuzz-S cannot find in all five programs. Overall, ECFuzz finds 9 more types of exceptions with an increase of 128.6% compared to ECFuzz-S. There are two reasons why ECFuzz has achieved the above good results: 1) ECFuzz fully considers the configuration dependencies in large-scale systems and designs corresponding mutation strategies for each dependency, improving the ability to find unexpected failures; 2) ECFuzz selects multiple configuration parameters from candidate configuration parameters in each fuzzing campaign and intelligently adjusts them based on the test results, thus ECFuzz can explore more program codes in the same single fuzzing campaign.

4.2 RQ2: Unit-Testing-Oriented Configuration Validation Strategy Evaluation

To effectively validate the generated configuration parameters, ECFuzz designs a unit-testing-oriented configuration validation strategy. To show its effectiveness, we compare it with the strategy that injects directly into system testing without using unit testing. We denote this strategy as *ECFuzz-W*. Table 5 shows the results of the quality of testcases for different strategies.

Answer to RQ2. We find that the total quality of testcases of ECFuzz-W and ECFuzz are 38.4‰ and 92.5‰, respectively. This means that ECFuzz finds 54.1 more unexpected failures when the

Table 5: Quality of testcases for different configuration testing tools.

Program	Testcases	Exceptions			Results	
		Startup Exception	Runtime Exception	Shutdown Exception	Unexpected Failures	Quality of Testcases
ECFuzz-S						
HCommon	680	29	5	0	5	7.4‰
HDFS	161	33	2	0	2	12.4‰
HBase	321	164	20	2	22	68.5‰
ZooKeeper	2162	2127	35	0	35	16.2‰
Alluxio	80	10	6	0	6	75.0‰
Total	3404	2363	68	2	70	20.6‰
ECFuzz-W						
HCommon	945	96	12	0	12	12.7‰
HDFS	1006	257	12	0	12	11.9‰
HBase	3553	3513	39	1	40	11.3‰
ZooKeeper	4716	4267	318	0	318	67.4‰
Alluxio	370	97	25	0	25	67.6‰
Total	10590	8230	406	1	407	38.4‰
ConfTest						
HCommon	972	146	4	0	4	4.1‰
HDFS	1073	424	3	0	3	2.8‰
HBase	3508	3490	17	1	18	5.1‰
ZooKeeper	4908	4609	221	0	221	45.0‰
Alluxio	380	178	41	0	41	107.9‰
Total	10841	8847	286	1	287	26.5‰
ConfErr						
HCommon	931	45	2	0	2	2.1‰
HDFS	961	130	0	0	0	0‰
HBase	3519	3502	16	1	17	4.8‰
ZooKeeper	4894	4588	226	0	226	46.2‰
Alluxio	238	86	24	0	24	100.8‰
Total	10543	8351	268	1	269	25.5‰
ConfDiagDetector						
HCommon	946	83	4	0	4	4.2‰
HDFS	1019	282	2	0	2	2.0‰
HBase	3395	3371	22	2	24	7.1‰
ZooKeeper	4832	4461	269	0	269	55.7‰
Alluxio	375	157	41	0	41	109.3‰
Total	10567	8354	338	2	340	32.2‰
ECFuzz						
HCommon	697	80	15	0	15	21.5‰
HDFS	272	127	10	0	10	36.8‰
HBase	328	299	27	2	29	88.4‰
ZooKeeper	969	794	146	0	146	150.7‰
Alluxio	79	28	17	0	17	215.2‰
Total	2345	1328	215	2	217	92.5‰

same 1000 testcases are injected into the system. ECFuzz-W directly injects the generated testcases into the PUT to execute system tests, generating more testcases. However, most testcases are filtered by the PUT's configuration parsing code, which leads to startup exceptions. Compared to ECFuzz-W, ECFuzz uses a unit-testing-oriented configuration validation strategy to filter out testcases that are unlikely to yield errors in advance before executing system testing, reducing the costly system testing overhead. Therefore, ECFuzz can significantly improve the quality of testcases.

4.3 RQ3: Quality of Testcases and Types of Exceptions Evaluation

To show the effectiveness of ECFuzz, we compare it with the three state-of-the-art configuration testing tools, including ConfTest, ConfErr and ConfDiagDetector.

- **ConfTest [19]** studied 8 mature open-source and commercial software and summarized a fine-grained classification of option types. Based on this classification, it could extract syntactic and semantic constraints of each type to generate misconfiguration.
- **ConfErr [14]** used human error models rooted in psychology and linguistics to generate realistic configuration errors. The configuration errors generated by ConfErr include spelling errors, structural errors and semantic errors.
- **ConfDiagDetector [38]** inferred the likely data type of each configuration option from an existing, well-formed example configuration. Then, it mutated the existing configuration by repeatedly replacing the value of each option with random values or values from a predefined pool.

Table 5 and Figure 4 show the results of the quality of testcases and types of exceptions for different tools, respectively.

Answer to RQ3. For the quality of testcases, we find that ECFuzz achieves significant improvements compared to other state-of-the-art tools. Specifically, the total quality of testcases of ConfTest, ConfErr, ConfDiagDetector and ECFuzz are 26.5‰, 25.5‰,

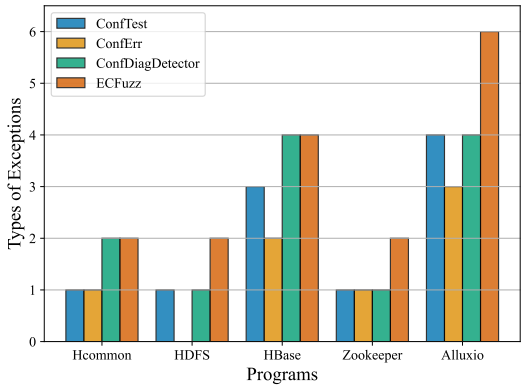


Figure 4: Types of exceptions for comparison with the state-of-the-art.

32.2‰ and 92.5‰, respectively. This means that ECFuzz finds 60.3–67 more unexpected failures when the same 1000 testcases are injected into the system. Moreover, the quality of testcases for ECFuzz-W is 38.4‰, which is also better than other state-of-the-art tools. As with other tools, ECFuzz-W directly injects the generated testcases into the PUT to execute system tests. The difference is that ECFuzz-W uses a multi-dimensional configuration generation strategy to generate high-quality testcases. For types of exceptions, we find that ECFuzz outperforms other state-of-the-art tools in most programs. Specifically, ECFuzz finds 1, 1 and 2 more types of exceptions on the HDFS, Zookeeper and Alluxio programs, respectively. For the HCommon and HBase programs, ECFuzz also maintains at least the same effectiveness as other state-of-the-art tools. Overall, ECFuzz achieves good results in both quality of testcases and types of exceptions, which proves the effectiveness of the multi-dimensional configuration generation strategy and the unit-testing-oriented configuration validation strategy in ECFuzz.

4.4 RQ4: Unknown Bugs Evaluation

ECFuzz is an effective configuration fuzzer for large-scale systems and has found many previously unknown configuration-induced bugs. The detailed information is shown in Table 6.

Answer to RQ4. ECFuzz finds the most number of unknown bugs compared to other state-of-the-art tools. Specifically, ECFuzz has detected 14 previously unknown configuration-induced bugs and 5 of them have been confirmed.

Take the bug Alluxio-17509 as an example, as shown in Figure 5. There is a set value relationship dependency between the configuration parameter *alluxio.work.dir* (indicating the working directory of *alluxio*) and the parameter *alluxio.master.journal.folder* (indicating the directory where the logs of the master node are stored). Specifically, the default value of the latter configuration parameter uses the value of the previous configuration parameter as a part. In this case, ECFuzz mutates both of the above configuration parameters according to their constraints at the same time.

When the configuration parameter *aluxio.work.dir* is mutated to / and the parameter *alluxio.master.journal.folder* is mutated to ${*alluxio.work.dir*}/*journal** (*journal** represents any mutation to *journal*), the latter configuration parameter has the value

Configuration Change:

```
- alluxio.work.dir=${alluxio.home}
- alluxio.master.journal.folder=${alluxio.work.dir}/journal
+ alluxio.work.dir=/
+ alluxio.master.journal.folder=${alluxio.work.dir}/journal*
```

Configuration Read:

```
public static URI getJournalLocation() {
    String journalDirectory =
    ServerConfiguration.get(PropertyKey.MASTER_JOURNAL_FOLDER);
    ...
    try {
         return new URI(journalDirectory);
     }
    ...
} /* alluxio/master/journal/JournalUtils.java */
```

Root Cause:

```
public void format() throws IOException {
    File journalPath = mConf.getPath();
    if (journalPath.isDirectory()) {
      org.apache.commons.io.FileUtils.cleanDirectory(mConf.getPath());
    }
    ...
} /* alluxio/master/journal/raft/RaftJournalConfiguration.java */
```

Clean all files

Figure 5: Bug Alluxio-17509 exposed by ECFuzz.

*//journal** . During the configuration read, the program i) first reads this value and saves it to the variable *journalDirectory*; ii) then determines whether the value ends with a separator, if not, a separator is added to the end and the value becomes *//journal*/*; iii) finally constructs a new *URI* based on this value as the return value. Specifically, the program calls the method *getpath* of class *URI*, and the value *//journal*/* is resolved to / as the return value. In Linux operating systems, / represents the root directory. Therefore, when the program runs the log directory *format*, it deletes all the files in the root directory, which leads to serious consequences.

5 RELATED WORK

The configuration complexity of large-scale systems and the high frequency of configuration changes make configuration-induced error one of the main causes of failures and performance problems today [1, 5, 12, 13, 21, 22, 26, 34].

Existing configuration testing techniques generally use fuzzing to generate configuration parameters. According to different fuzzing techniques, we divide configuration testing techniques into two categories: grammar-based and constraint-based configuration testing techniques.

Grammar-based configuration testing uses dictionary-based or grammar-based fuzzing techniques to generate configuration parameters that satisfy grammatical validity. ConfigFuzz [40], TOFU [29], POWER [15] and Testing Configurations in the Fuzzing Book [36] are typical of this. The configuration parameters generated by this method can effectively pass some branches and improve the code coverage. However, these works are only applicable to small-scale configurable programs, such as the program Xmllint. In large-scale systems, configuration parameters have many other constraints besides grammar, such as semantics.

Table 6: Unknown bugs for comparison with the state-of-the-art.

ISSUE-ID	Description	Component(s)	Status	ECFuzz-S	ECFuzz-W	ConfTest	ConfErr	ConfDiagDetector	ECFuzz
HBase-25886	Function Exception	client	Pending	✓	✓	✓	✓	✓	✓
HBase-26114	Startup Exception	master	Confirmed	✓	✓	✗	✓	✓	✓
HBase-27016	Shutdown Exception	master	Pending	✓	✓	✓	✓	✓	✓
HBase-27341	Function Exception	logging	Pending	✓	✓	✓	✓	✓	✓
HDFS-16408	Function Exception	namenode	Confirmed	✗	✗	✓	✗	✗	✓
HDFS-16653	Function Exception	dfsadmin	Confirmed	✗	✓	✗	✗	✗	✓
HDFS-16697	Function Exception	namenode	Confirmed	✓	✓	✗	✗	✓	✓
HDFS-16721	Unclear Description	dfs-client	Fixing	✗	✗	✗	✗	✗	✓
Hadoop-18216	Blocked Request	io	Confirmed	✓	✓	✓	✗	✓	✓
Hadoop-18362	Unclear Description	test	Fixing	✗	✗	✗	✗	✗	✓
Alluxio-17057	Function Exception	logging	Fixing	✗	✗	✓	✓	✓	✓
Alluxio-17062	Function Exception	logging	Fixing	✓	✓	✓	✓	✗	✓
Alluxio-17064	Function Exception	logging	Pending	✓	✓	✓	✓	✓	✓
Alluxio-17509	Function Exception	master	Pending	✗	✓	✗	✗	✗	✓
Total	-	-	-	8/14	10/14	8/14	7/14	8/14	14/14

1. Hadoop in this table indicates HCommon.

In contrast to grammar-based configuration testing, *constraint-based configuration testing* uses constraint-based fuzzing techniques to generate configuration parameters that violate configuration constraints. Among them, constraints include both grammar and semantic constraints. This approach uses the error configuration space instead of the complete configuration space to improve the effectiveness of configuration testing. Configuration error injection testing (CEIT) [2, 14, 17–19, 32, 38] is typical of this, which tests the system's reaction to misconfigurations and verifies the system's resilience. The key for CEIT is to accurately extract configuration constraints. Some work [7, 23, 24, 27, 28, 30, 37] uses machine learning and text analysis techniques to infer configuration constraints from the configuration text. And other work [3, 4, 6, 8, 10, 11, 16, 20, 23, 31, 32, 39] uses program analysis techniques, such as data flow analysis (DFA) to infer configuration constraints from the PUT code. At present, CEIT has been widely used in testing the configurations of large-scale software systems.

However, in large-scale systems, only considering the constraint relationships of a single configuration parameter is not enough. cDEP [8] finds that dependency relationships exist in different software configurations in large-scale systems and uses static analysis to automatically discover multiple custom configuration dependencies from the bytecode. This motivates us to develop an effective mutation strategy to cover different dependency relationships during fuzzing. More seriously, the time cost of system testing in large-scale systems is very expensive, and its average speed is several orders of magnitude slower than small-scale programs. In configuration validation, the above configuration testing techniques directly inject the generated configuration parameters into the PUT for system testing. Once a configuration parameter that is unlikely to yield errors has been generated, lots of time for system testing will be wasted. Therefore, this paper aims to improve the effectiveness of configuration testing from configuration parameter generation and validation.

6 CONCLUSION

In this paper, we propose ECFuzz, which effectively tests configurations in large-scale systems by improving the process of configuration generation and configuration validation. In configuration generation, ECFuzz considers dependencies of configuration parameters in large-scale systems and designs mutation strategies for each dependency. In configuration validation, ECFuzz uses unit testing to filter out some testcases that are unlikely to yield errors before running system testing. We have conducted extensive experiments on real-world large-scale systems including HCommon, HDFS, HBase, ZooKeeper and Alluxio. The results show that ECFuzz achieves significant improvements and has exposed 14 previously unknown bugs and 5 of them have been confirmed.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their insightful comments. This research was supported by the Key Research and Development Projects of Sichuan Province (No. 2022ZDZX0006) and the Science and Technology Innovation Program of Hunan Province (No. 2023RC1001).

REFERENCES

[1] George Amvrosiadis and Medha Bhadkamkar. 2016. Getting Back up: Understanding How Enterprise Data Backups Fail. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Denver, CO, USA) *(USENIX ATC '16)*. USENIX Association, USA, 479–492.

[2] F. A. Arshad, R. J. Krause, and S. Bagchi. 2013. Characterizing configuration problems in Java EE application servers: An empirical study with GlassFish and JBoss. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, Los Alamitos, CA, USA, 198–207. https://doi.org/10.1109/ISSRE.2013.6698919

[3] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. 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* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 307–320.

[4] Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 237–250.

[5] Luiz Andr Barroso, Jimmy Clidaras, and Urs Hlzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines* (2nd ed.). Morgan & Claypool Publishers.

[6] Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Adithya Abraham Philip. 2018. Orca: Differential Bug Localization in Large-Scale Services. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 493–509.

[7] R. Bhagwan, S. Mehta, A. Radhakrishna, and S. Garg. 2021. Learning Patterns in Configuration. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 817–828. https://doi.org/10.1109/ASE51524.2021.9678525

[8] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. 2020. Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 362–374. https://doi.org/10.1145/3368089.3409727

[9] Runxiang Cheng, Lingming Zhang, Darko Marinov, and Tianyin Xu. 2021. Test-Case Prioritization for Configuration Testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 452–465. https://doi.org/10.1145/3460319.3464810

[10] Zhen Dong, Artur Andrzejak, and Kun Shao. 2015. Practical and Accurate Pinpointing of Configuration Errors Using Static Analysis. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME) (ICSME '15)*. IEEE Computer Society, USA, 171–180. https://doi.org/10.1109/ICSM.2015.7332463

[11] Zhen Dong, M. Ghanavati, and A. Andrzejak. 2013. Automated diagnosis of software misconfigurations based on static analysis. In *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE Computer Society, Los Alamitos, CA, USA, 162–168. https://doi.org/10.1109/ISSREW.2013.6688897

[12] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2670979.2670986

[13] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/2987550.2987583

[14] L. Keller, P. Upadhyaya, and G. Candea. 2008. ConfErr: A tool for assessing resilience to human configuration errors. In *2008 IEEE International Conference on Dependable Systems & Networks With FTCS and DCC (DSN)*. IEEE Computer Society, Los Alamitos, CA, USA, 157–166. https://doi.org/10.1109/DSN.2008.4630084

[15] Ahcheong Lee, Irfan Ariq, Yunho Kim, and Moonzoo Kim. 2022. POWER: Program Option-Aware Fuzzer for High Bug Detection Ability. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 220–231. https://doi.org/10.1109/ICST53961.2022.00032

[16] Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. 2020. Statically Inferring Performance Properties of Software Configurations. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 10, 16 pages. https://doi.org/10.1145/3342195.3387520

[17] Shanshan Li, Wang Li, Xiangke Liao, Shaoliang Peng, Shulin Zhou, Zhouyang Jia, and Teng Wang. 2018. Confvd: System reactions analysis and evaluation through misconfiguration injection. *IEEE Transactions on Reliability* 67, 4 (2018), 1393–1405.

[18] Wang Li, Zhouyang Jia, Shanshan Li, Yuanliang Zhang, Teng Wang, Erci Xu, Ji Wang, and Xiangke Liao. 2021. Challenges and Opportunities: An in-Depth Empirical Study on Configuration Error Injection Testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 478–490. https://doi.org/10.1145/3460319.3464799

[19] Wang Li, Shanshan Li, Xiangke Liao, Xiangyang Xu, Shulin Zhou, and Zhouyang Jia. 2017. ConfTest: Generating Comprehensive Misconfiguration for System Reaction Ability Evaluation. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering* (Karlskrona, Sweden) *(EASE'17)*. Association for Computing Machinery, New York, NY, USA, 88–97. https://doi.org/10.1145/3084226.3084244

[20] Max Lillack, Christian Kästner, and Eric Bodden. 2018. Tracking Load-Time Configuration Options. *IEEE Trans. Softw. Eng.* 44, 12 (dec 2018), 1269–1291. https://doi.org/10.1109/TSE.2017.2756048

[21] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. 2019. What Bugs Cause Production Cloud Incidents?. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 155–162. https://doi.org/10.1145/3317550.3321438

[22] Ben Maurer. 2015. Fail at Scale: Reliability in the Face of Rapid Change. *Queue* 13, 8 (sep 2015), 30–46. https://doi.org/10.1145/2838344.2839461

[23] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Maddila, B. Ashok, Sumit Asthana, Christian Bird, and Aditya Kumar. 2020. Rex: Preventing Bugs and Misconfiguration in Large Services Using Correlated Change Analysis. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'20)*. USENIX Association, USA, 435–448.

[24] Mark Santolucito, Ennan Zhai, Rahul Dhodapkar, Aaron Shim, and Ruzica Piskac. 2017. Synthesizing Configuration File Specifications with Association Rule Learning. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 64 (oct 2017), 20 pages. https://doi.org/10.1145/3133888

[25] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing Configuration Changes in Context to Prevent Production Failures. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 42, 17 pages.

[26] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 328–343. https://doi.org/10.1145/2815400.2815401

[27] Ozan Tuncer, Anthony Byrne, Nilton Bila, Sastry Duri, Canturk Isci, and Ayse K. Coskun. 2020. ConfEx: A Framework for Automating Text-based Software Configuration Analysis in the Cloud. https://doi.org/10.48550/ARXIV.2008.08656

[28] Fei Wang, Zhengjian Zhao, Zhichao Wang, Minchao Ma, and Junjie Lu. 2022. Intelligent Software Service Configuration Technology Based on Association Mining. *Journal of Physics: Conference Series* 2185, 1 (jan 2022), 012024. https://doi.org/10.1088/1742-6596/2185/1/012024

[29] Zi Wang, Ben Liblit, and Thomas Reps. 2020. TOFU: Target-Oriented FUzzer. *arXiv e-prints* (2020), arXiv–2004.

[30] Chengcheng Xiang, Haochen Huang, Andrew Yoo, Yuanyuan Zhou, and Shankar Pasupathy. 2020. PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 18, 16 pages.

[31] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 619–634.

[32] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do Not Blame Users for Misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 244–259. https://doi.org/10.1145/2517349.2522727

[33] Tianyin Xu and Yuanyuan Zhou. 2015. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Comput. Surv.* 47, 4, Article 70 (jul 2015), 41 pages. https://doi.org/10.1145/2791577

[34] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 249–265. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

[35] Michał Zalewski. 2016. *American fuzzy lop.* Google. Retrieved October 28, 2022 from https://lcamtuf.coredump.cx/afl

[36] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The fuzzing book.

[37] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. *SIGPLAN Not.* 49, 4 (feb 2014), 687–700. https://doi.org/10.1145/2644865.2541983

[38] Sai Zhang and Michael D. Ernst. 2015. Proactive Detection of Inadequate Diagnostic Messages for Software Configuration Errors. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/2771783.2771817

[39] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. 2019. The Inflection Point Hypothesis: A Principled Debugging Approach for Locating the Root Cause of a Failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 131–146. https://doi.org/10.1145/3341301.3359650

[40] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. 2023. Fuzzing Configurations of Program Options. *ACM Trans. Softw. Eng. Methodol.* (feb 2023). https://doi.org/10.1145/3580597