ConfFuzz: Parameter-Aware Greybox Fuzzing for Configurable Cloud Systems

Shuai Wang
University of Illinois
Urbana-Champaign, IL, USA
swang516@illinois.edu

Hao Wang
University of California
Berkeley, CA, USA
hwang628@berkeley.edu

Darko Marinov
University of Illinois
Urbana-Champaign, IL, USA
marinov@illinois.edu

Tianyin Xu✉
University of Illinois
Urbana-Champaign, IL, USA
tyxu@illinois.edu

Yongle Zhang
Purdue University
West Lafayette, IN, USA
yonglezh@purdue.edu

Abstract

Cloud systems are highly configurable, with hundreds to thousands of configuration parameters. Configuration-related bugs are major sources of cloud failures, yet most tests only run under default configurations, leaving systems vulnerable to untested configurations.

This paper applies coverage-based greybox fuzzing to configuration testing of cloud systems. We use developer-written unit tests as fuzz drivers: they avoid the cluster-initialization overhead of system tests and thus provide the high test execution speed that greybox fuzzing demands. Since cloud systems expose hundreds to thousands of configuration parameters but any unit test typically exercises only a small subset of them (15.14% on average), we design parameter-aware mutation and fuzzing energy allocation strategies: (1) we restrict mutation to only the configuration parameters exercised by each seed, and (2) we incorporate both code coverage and configuration parameter coverage into fuzzing energy allocation, giving more effort to seeds that explore more code and parameters.

We implement these ideas in ConfFuzz and evaluate it on nine widely used cloud systems. Compared with a greybox fuzzing baseline without parameter-aware mutation and energy allocation, ConfFuzz covers roughly 3.2 times more branches while exploring substantially more configuration parameters. In total, ConfFuzz and its variants have detected 125 bugs, of which 14 have been fixed and 12 more have been confirmed.

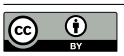
CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Configuration testing, Greybox fuzzing, Cloud systems, Unit tests

ACM Reference Format:
Shuai Wang, Hao Wang, Darko Marinov, Tianyin Xu, and Yongle Zhang. 2026. ConfFuzz: Parameter-Aware Greybox Fuzzing for Configurable Cloud Systems. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837476



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837476

1 Introduction

Modern cloud systems are highly configurable, with hundreds to thousands of configuration parameters spanning diverse types, semantics, and constraints [13, 44, 58, 76, 85]. The enormous configuration space makes it fundamentally difficult to test and verify software correctness [34, 57, 67–69]. As a result, configurations have been a major source of cloud system failures. For example, faulty configurations were reported as the second largest cause of service disruptions in a major Google production service [8]. At Meta, 16% of service-level incidents, including significant outages (e.g., [55]), were introduced by configuration changes [40, 58]. Other studies report comparable trends across cloud ecosystems [22, 37, 47, 53]. Mature systems often have abundant developer-written unit tests that achieve high code coverage [25, 57], but those tests usually exercise only a small set of handpicked configurations, most often the default configuration. Systems therefore remain vulnerable to failures triggered by untested configuration changes.

Recent research on automated configuration testing of cloud systems mainly follows two directions. Configuration testing [15, 57, 66, 67, 75, 78] identifies the parameters exercised by each unit test and generates a limited set of configurations for testing, but it does not use execution feedback to guide subsequent generation. Configuration fuzzing [14, 18, 19, 31] mutates configuration values automatically. For example, ECFuzz [31] targets large cloud systems, but it remains blackbox: it does not use execution feedback and relies on a small number of manually written system tests (2.6 per project on average), which limits both test execution speed and the breadth of code that testing can reach [21, 23, 45, 77].

Greybox fuzzing has recently been applied to configuration testing of command-line applications [30, 61, 63, 90], showing that coverage can further guide configuration fuzzing toward unexplored behavior. However, the configuration spaces of command-line applications are orders of magnitude smaller than the configuration spaces of cloud systems, with each application exposing only 1 to 14 options, 92.6% of which are booleans [90]. Therefore, how to reduce the search space and guide fuzzing to efficiently reveal unexplored system behaviors remains a challenge for greybox configuration fuzzing of cloud systems.

In this paper, we apply coverage-based greybox fuzzing to configuration testing of cloud systems. We address the above challenge by leveraging exercised configuration parameters gathered during execution to guide mutation and fuzzing energy allocation. Specifically, we make the following design decisions:

- To provide the high test execution speed and code coverage that greybox fuzzing demands, we use existing developer-written unit tests as fuzz drivers. Such tests are abundant in mature projects and execute far faster than the system tests used by the prior cloud-system configuration fuzzer [31], which usually spend seconds to minutes initializing clusters, starting services, and distributing workloads. We transform existing unit tests into fuzz drivers by instrumenting the unified configuration APIs (§2.3) that mature cloud systems employ, building on ideas from Ctest [57] and UTopia [26].
- Based on our observation that a unit test typically uses only a small fraction of all parameters (15.14% on average) and mutating one parameter can steer execution to a new path that reads additional parameters, we design parameter-aware mutation which dynamically tracks which configuration parameters are exercised during fuzzing and restricts mutation to only those parameters.
- Different unit tests can also exercise dramatically different numbers of parameters. ConfFuzz therefore introduces parameter-aware fuzzing energy allocation [11], which allocates fuzzing energy based on both code coverage and configuration parameter coverage, giving more effort to seeds that explore more code and more of the configuration space.

We implement these ideas in ConfFuzz and evaluate it on nine large cloud systems with hundreds to thousands of configuration parameters. ConfFuzz advances configuration fuzzing to cloud-system scale by integrating and customizing the three ingredients above: unit tests as fuzz drivers, reduction of the configuration space to the dynamically exercised parameters of each seed, and exercised parameters as feedback for seed selection and energy allocation. In contrast, ECFuzz [31], the prior cloud-system configuration fuzzer, is blackbox with no feedback, and ConfigFuzz [90], the prior greybox configuration fuzzer for command-line options, does not scale to the large configuration space of cloud systems.

We compare ConfFuzz against a greybox baseline that reimplements the core strategy of ConfigFuzz [90] for Java-based cloud systems, because the original ConfigFuzz targets C/C++ programs and command-line options. Our evaluation shows that ConfFuzz finds more bugs than the baseline, and covers roughly 3.2 times more branches while exploring substantially more configuration parameters. We also perform ablation studies of parameter-aware mutation and fuzzing energy allocation, which show that both improve fuzzing efficiency: ConfFuzz covers 13.3% to 14.2% more branches than its two parameter-aware variants. Across all campaigns, ConfFuzz and its variants expose 125 configuration-related bugs, including 14 fixed bugs and 12 additional confirmed bugs.

Summary. This paper makes the following contributions:

- A greybox configuration fuzzing design for cloud systems with unified configuration APIs, instantiated with existing unit tests as fuzz drivers;
- Dynamic parameter-aware fuzzing with per-execution tracking, mutation over exercised parameters, and fuzzing energy allocation guided by code coverage and parameter usage;
- An evaluation of ConfFuzz on nine widely used cloud systems, including ablation studies and 125 detected bugs.

2 Background and Motivation

2.1 Blackbox/Greybox Configuration Fuzzing

Prior work has explored fault injection and fuzzing for configurable systems through both configuration-file and command-line interfaces [14, 18, 19, 27, 30–32, 38, 61, 63, 69, 81, 84, 90]. These techniques include both blackbox and greybox designs. Blackbox fuzzing mutates configurations and detects bugs through coarse outcomes such as crashes and failures. Greybox fuzzing additionally uses execution feedback, such as code coverage, to guide mutations.

ECFuzz [31] is prior work on blackbox configuration fuzzing for large cloud systems. It targets systems with large configuration spaces, but it does not use execution feedback such as code coverage. Its search therefore remains unguided by program behavior. When the target system exposes hundreds to thousands of parameters but any single execution exercises only a small fraction of them, unguided search is expensive: the fuzzer may spend many executions mutating parameters that the current test or fuzz driver never reads, and therefore make little progress toward new paths. The problem is more severe in complex cloud systems, where end-to-end testing often relies on a small number of system tests or validators and may not reach important modules such as recovery and fault-tolerance mechanisms. Prior work has shown that these modules are difficult to reach [21, 23, 45, 77] and prone to configuration-related defects [77].

ConfigFuzz [90] is a greybox configuration fuzzer that uses code coverage to guide fuzzing of command-line options. Follow-up work extends option fuzzing with whitebox analysis [63, 64], LLM-guided option-combination prediction [56, 61], interleaved option-file fuzzing [30], and dynamic configuration scheduling [17]. However, these techniques target a different setting from this paper. In ConfigFuzz's evaluation, for example, each studied program exposes only 1 to 14 options, and 25 of the 27 options across all studied programs are simple booleans. In such a small option space, the lack of parameter-level guidance is less restrictive. Cloud systems present a different challenge: they typically expose hundreds to thousands of parameters with diverse types and semantics. Code coverage alone does not tell which configuration parameters were exercised in the current execution, and therefore does not indicate which part of the configuration space should be mutated next.

Coverage-based greybox fuzzing has been widely studied and shown effective in many testing settings [11, 12, 20, 42, 48, 49, 54]. Greybox fuzzing is therefore a natural starting point for configuration testing of cloud systems. Cloud systems, however, expose a second source of guidance beyond code coverage. Each execution not only reaches some set of code paths but also reveals which configuration parameters were exercised on those paths. Runtime parameter-usage information can therefore focus later mutations on the relevant region of the configuration space.

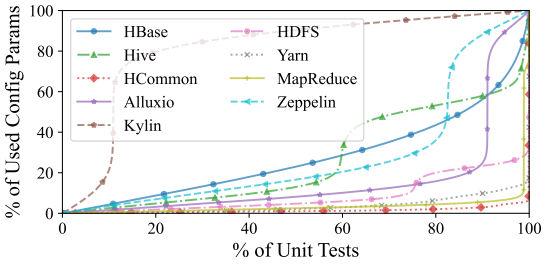


Figure 1: The CDF of the percentage of configuration parameters used by unit tests across nine evaluated projects.

2.2 Unit Tests as Fuzz Drivers

Greybox fuzzing needs fuzz drivers that can be executed repeatedly and at low cost. Existing developer-written unit tests satisfy this requirement well. Ctest [57] showed that many developer-written unit tests exercise configuration parameters and therefore have strong potential for configuration testing. UTopia [26] showed that existing unit tests can be transformed into fuzz drivers by turning them into parameterized unit tests [59]. However, unit tests also sharpen the guidance problem because a unit test typically exercises only a small slice of the full configuration space.

Figure 1 shows the percentage of configuration parameters exercised by unit tests across nine evaluated cloud systems. On average, a unit test exercises only 15.14% of the total parameters. In HCommon, MapReduce, and Yarn, 90% of unit tests exercise less than 10% of the parameters. Directly mutating parameters from the full configuration space is therefore inefficient in unit-test-driven greybox fuzzing. When a fuzzer mutates parameters that the current test does not read, the resulting configuration is likely to produce the same behavior as the parent seed. The narrow exercised-parameter set of a unit test makes this inefficiency substantial.

The exercised-parameter set is also not fixed. Mutating one parameter can steer execution to a new path that exercises additional parameters. A one-time parameter set collected before fuzzing, such as the set observed under the default configuration, is therefore insufficient for iterative greybox fuzzing. The fuzzer must track parameter usage in each fuzzing round and use that information together with code coverage to guide later mutation.

2.3 Configuration APIs

ConfFuzz instruments configuration APIs for dynamic tracking of configuration parameters and instruments configuration constructors for in-situ mutation of configuration values during fuzzing (§3.2.1). Mature software projects, including the nine projects in our evaluation, typically expose unified configuration APIs to manage hundreds to thousands of parameters systematically. Prior work on configuration testing and analysis [9, 15, 31, 36, 38, 51, 52, 57, 67, 77, 81, 87–89] has repeatedly relied on this design, and our approach builds on the same observation.

In modern cloud systems, configurations are commonly represented as dedicated classes that expose a common pair of operations: GET and SET. The GET API takes the form "`<T> get(String parameter)`"; it accepts a configuration-parameter name and returns its value. The SET API takes the form "`void set(String parameter, <T> value)`"; it updates the value of a given parameter with the input `value`. In practice, these raw GET and SET APIs are often wrapped by typed APIs such as `getInt` and `getBoolean` [2, 35].

These simple, unified APIs provide natural points for instrumentation to track the runtime usage of configuration parameters and to directly mutate the in-memory representation of configuration objects. They can also be instrumented efficiently without reasoning about complex parsing logic or diverse configuration file formats such as XML, INI, JSON, and YAML.

Algorithm 1: Coverage-based greybox fuzzing.

Input: program p, set of initial inputs I
Output: a queue of seeds S and failing inputs F

```
1  S ← I
2  F ← ∅
3  totalCoverage ← ∅
4  repeat
5      s ← selectSeed(S)
6      for 1 ≤ i ≤ numCandidates(s, S) do
7          candidate ← mutate(s)
8          coverage, result ← run(p, candidate)
9          if result = Failure then
10             F ← F ∪ {candidate}
11         else if coverage ⊈ totalCoverage then
12             S.append(candidate)
13             totalCoverage ← totalCoverage ∪ coverage
14         if time budget expires then
15             break
16 until time budget expires;
17 return S, F
```

3 ConfFuzz

ConfFuzz is a greybox configuration fuzzer for highly configurable cloud systems that uses developer-written unit tests as fuzz drivers. Unlike prior configuration fuzzers [18, 19, 31, 90], ConfFuzz explicitly captures the interaction between configuration parameters and system code. This design leads to two key decisions:

- ConfFuzz mutates only the configuration parameters exercised by the selected fuzz driver, instead of mutating a random subset from the full parameter space of the target system;
- ConfFuzz uses both code coverage and exercised-parameter information to decide which executions to save as seeds and how much fuzzing energy to allocate to each seed.

Algorithm 1 shows conventional code-coverage-guided greybox fuzzing. The fuzzer maintains a seed queue of interesting inputs, allocates mutation budget with `numCandidates(s, S)`, which scores the selected seed s against the rest of the queue S using code coverage alone (line 6), mutates saved inputs (line 7), and appends a candidate to the seed queue only when executing it expands total code coverage (lines 8–12). ConfFuzz, shown in Algorithm 2, keeps this greybox structure but specializes it to configuration fuzzing with unit tests. The teal statements in Algorithm 2 mark the ConfFuzz-specific extensions.

ConfFuzz extends traditional coverage-based greybox fuzzing in three ways. First, ConfFuzz runs the fuzz driver once under the default configuration D and uses that execution to create the initial seed (lines 1–2), because the default configuration is the natural

Algorithm 2: ConfFuzz: Param-aware greybox configuration fuzzing.

Input: test t, default configuration D
Output: a queue of seeds S (each with configuration, code coverage and exercised parameters) and failing configurations F

```
1  coverage, params, _ ← run(t, D)
2  S ← [(D, coverage, params)]
3  F ← ∅
4  totalCoverage ← coverage
5  totalParams ← params
6  repeat
7      s ← selectSeed(S)
       /* param-aware energy allocation                    */
8      for 1 ≤ i ≤ numCandidates(s, S) do
           /* param-aware configuration mutation           */
9          candidate ← mutateConfig(config(s), params(s))
10         coverage', params', result ← run(t, candidate)
11         if result = Failure then
12             F ← F ∪ {candidate}
13         else if coverage' ⊈ totalCoverage
               ∨ params' ⊈ totalParams then
14             S.append((candidate, coverage', params'))
15             totalCoverage ← totalCoverage ∪ coverage'
16             totalParams ← totalParams ∪ params'
17         if time budget expires then
18             break
19 until time budget expires;
20 return S, F
```

valid starting point for a unit test. Second, each saved seed stores not only a configuration and its code coverage but also the set of parameters exercised in that execution. This runtime parameter-usage information lets ConfFuzz mutate only the parameters that affected the parent execution (line 9). Third, ConfFuzz extends the energy allocation and seed saving logic. The method `numCandidates(s, S)` uses both the code coverage of the selected seed and its exercised parameters to decide how many candidates to generate (line 8), and ConfFuzz saves a seed when it either covers new code or exercises a previously unseen parameter (lines 10–14).

The rest of this section describes three main parts of ConfFuzz:

- Instrumentation support for transforming existing unit tests into configuration fuzz drivers (§3.1);
- Parameter-aware configuration mutation to mutate only exercised parameters during fuzzing executions (§3.2);
- Parameter-aware energy allocation that assigns more fuzzing energy to seeds that cover more code and more exercised configuration parameters (§3.3);

We also describe the failure-analysis support of ConfFuzz in §3.4, which helps users debug the failures found by ConfFuzz.

3.1 Transforming Unit Tests into Fuzz Drivers

ConfFuzz follows the same high-level idea as UTopia [26]: it reuses existing unit tests as fuzz drivers. The transformation differs, however, because the fuzzed input in our setting is not a conventional API argument, as in UTopia, but a configuration object. In configuration testing, a unit test typically creates a configuration object

```
1  public class Configuration {
2    public Configuration() {
3  -   this(loadDefaultConfig);
4  +   this(ConfFuzzGenerator.genConfig());
5    }
6  } /* .../hadoop/conf/Configuration.java */
```

(a) Configuration constructor

```
1  public String get(String name) {
2  +   String confFuzzParam = name;
3      String[] names = handleDeprecation(deprecationContext.get(),
               name);
4      String result = null;
5      for(String n : names) {
6  +      confFuzzParam = n;
7          result = substituteVars(getProperty(n));
8      }
9  +   ConfigTracker.recordGet(confFuzzParam, result);
10     return result;
11 } /* .../hadoop/conf/Configuration.java */
```

(b) Configuration GET API

Figure 2: Instrumentation for Hadoop configuration constructor and GET API.

and later exercises only a small number of parameters through configuration APIs. ConfFuzz therefore needs to connect a mutated configuration object to the test under fuzz. It does so by instrumenting the constructor of the configuration class to load mutated configuration parameter values during fuzzing.

Figure 2a shows an instrumentation example in Hadoop [4]. Whenever a test creates a configuration object, the instrumented constructor invokes the configuration generator of ConfFuzz (line 4 in Figure 2a), obtains a mutated configuration object, and passes it to the test. If the class does not expose such a constructor, ConfFuzz instruments the GET APIs for each configuration parameter and returns the value generated by ConfFuzz when the test invokes the GET API. By connecting the unit test with the fuzzed configuration object, ConfFuzz can transform existing unit tests into fuzz drivers without modifying the tests themselves.

In the GET API of Figure 2b, `handleDeprecation` resolves a requested name to a set of names that includes aliases and deprecated names, and ConfFuzz updates `confFuzzParam` inside the same name-resolution loop that produces `result`. `ConfigTracker` therefore records the *resolved* name whose value the GET API returns, which is exactly the configuration parameter the test observes, rather than an unrelated alias.

Generality and assumptions. ConfFuzz targets systems that read configuration parameters through a uniform GET API such as `get(String)` and typed wrappers like `getInt` and `getBoolean` that retrieve a value by parameter name. This abstraction originates in `java.util.Properties` and is pervasive in mature Java systems and libraries (e.g., Apache Commons Configuration). This same uniformity is used by a broad class of configuration analysis and testing tools [13, 38, 57, 66, 68, 74, 76]. A configuration parameter is a named value read through such an API.

For a unit test to serve as a fuzz driver, it must (1) read at least one parameter through an instrumentable GET API, (2) exhibit configuration-dependent behavior, (3) pass under the default configuration, and (4) be reproducible enough to support debugging. ConfFuzz's effectiveness relies on the abundance of such tests, which prior studies have documented [15, 38, 57, 66, 67, 78].

Not all unit tests are suitable for configuration fuzzing because some tests do not exercise any configuration parameter. ConfFuzz automatically identifies suitable tests by running the full unit test suite once under the default configuration and selecting only the tests that exercise at least one configuration parameter.

3.2 Parameter-Aware Configuration Mutation

Parameter-aware configuration mutation allows ConfFuzz to use unit tests effectively as fuzz drivers. Traditional coverage-based greybox fuzzing (Algorithm 1) mutates a saved input without knowing which parts of that input affected the current execution. As §2.2 showed, a unit test typically exercises only a small subset of configuration parameters. If the fuzzer mutates across the full parameter space, many fuzzing rounds will spend effort on parameters that the current test never exercises. To avoid this inefficiency, ConfFuzz tracks the parameters exercised by each execution and passes that information to `mutateConfig(config, params)` in Algorithm 2 (line 9). As a result, each mutation round changes only the parameters exercised by the parent execution.

3.2.1 Dynamic configuration parameter tracking. ConfFuzz instruments configuration GET API(s) to dynamically track the configuration parameters that a fuzz driver actually exercises in each fuzzing execution. In modern cloud systems, configuration parameters are usually accessed either through a centralized configuration interface with GET APIs or through specialized APIs for individual configuration parameters. ConfFuzz instruments both types of APIs. Figure 2b shows the instrumentation for Hadoop [4]; only three new lines are required. Hadoop also exposes other GET APIs such as `getInt` and `getDouble`, but they are wrappers around the `get` method in Figure 2b. Instrumenting `get` is therefore sufficient.

When a configuration parameter is accessed during fuzzing, ConfFuzz records it in the parameter tracker (`ConfigTracker` in Figure 2b). The high-level instrumentation strategy resembles that of Ctest [57]. The key difference is that Ctest logs the parameters used by a test under the default configuration as a one-time analysis step, whereas ConfFuzz tracks exercised parameters in every fuzzing execution and uses the tracked set to guide later mutation.

3.2.2 Mutation workflow. The dynamically tracked parameter set determines which parameters ConfFuzz may mutate in each fuzzing round. During fuzzing, ConfFuzz saves selected executions as seeds. Each seed stores the configuration used in that execution, the code coverage from that execution, and the set of parameters exercised by the fuzz driver under that configuration (line 14 in Algorithm 2). The saved configuration becomes the base for later mutation, and the stored parameter set determines which parameters may be mutated if that seed is selected.

Starting from default configuration. ConfFuzz starts fuzzing from the default configuration because it is valid and well tested. If a fuzz driver fails even under the default configuration, that failure already indicates a bug, and the driver is not suitable for fuzzing. The blackbox configuration fuzzer ECFuzz [31] also starts from the default configuration. The difference is that ECFuzz mutates from the default configuration in every fuzzing round, whereas ConfFuzz uses the default configuration only to create the initial seed for each fuzz driver.

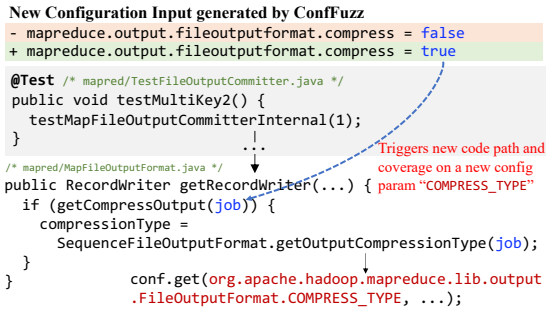


Figure 3: A new configuration parameter `COMPRESS_TYPE` used by the unit test `testMultiKey2`, via a new control flow, when a boolean parameter changes.

Maintaining the parameter set. For each fuzz driver, ConfFuzz first runs the driver under the default configuration and records the parameters exercised by that execution. The default configuration, its coverage, and the recorded parameters form the initial seed. Later, when ConfFuzz selects a seed as the parent for mutation, it starts from the configuration stored in that seed rather than from the default configuration, and mutates only parameters in the parameter set stored with that seed. After executing the mutated configuration, ConfFuzz records the parameters exercised in the new execution. If that execution is later saved as a new seed, ConfFuzz stores the new configuration, the new coverage, and the updated parameter set. The exercised-parameter set can therefore change dynamically during fuzzing, and ConfFuzz always mutates the parameters exercised by the selected parent seed. Figure 3 shows an example in which a test in MapReduce exercises a new parameter after a boolean parameter in the current parameter set changes from `false` to `true`, enabling a new control flow.

Generating values for selected parameters. After ConfFuzz selects the parameters to mutate from the parent seed, it must generate different values for those parameters. All other configuration parameters retain their values from the parent seed. Parameters outside the exercised-parameter set are not mutated either, so their values usually remain at the defaults inherited from the parent. ConfFuzz follows prior configuration fuzzers [31, 90] in generating values for the selected parameters. Specifically, ConfFuzz performs type inference and semantic inference for each configuration parameter. There is a large body of work on type inference for configuration parameters through program analysis [52, 81] and text analysis [10, 31, 32, 50, 86]. We use a simple analysis that infers the type of each parameter from the type of its default value. For `String`-typed parameters, semantic inference uses predefined regular expressions, which are the same patterns used for value generation, in a manner similar to prior work [31, 32, 86, 90]. Despite its simplicity, we find this approach effective in practice.

3.3 Parameter-Aware Energy Allocation

The previous subsection described how ConfFuzz selects and mutates configuration parameters in each fuzzing round. We now describe how ConfFuzz extends seed admission and mutation budgeting in traditional coverage-based greybox fuzzing.

Traditional coverage-based greybox fuzzing saves a candidate as a seed when the candidate expands total code coverage relative

to the current seed queue (lines 8–12 in Algorithm 1). ConfFuzz keeps that rule and adds a second saving condition: ConfFuzz also saves a candidate as seed when its exercised parameter set contains a parameter outside `totalParams` (lines 10–14 in Algorithm 2). New exercised parameters often accompany new code coverage because they are frequently reached through control flow that the current seed queue has not covered yet. The converse does not always hold: a candidate may expand code coverage without exercising any previously unseen parameter. This second saving condition therefore captures marginal novelty in the configuration space directly. A candidate that exposes a previously unseen parameter enters the seed queue immediately.

Traditional coverage-based greybox fuzzing already uses code coverage to allocate more energy to seeds with broader coverage and thus generate more candidates from those seeds. ConfFuzz preserves this coverage-guided budgeting and extends it with parameter information. Specifically, ConfFuzz uses `numCandidates(s, S)` (line 8) for each selected seed. For a seed s, let $coverage(s)$ be the set of code branches covered by s and $params(s)$ be the set of configuration parameters exercised by s; let S be the current seed queue. ConfFuzz defines $w_c(s,S) = \frac{|coverage(s)|}{\max_{s' \in S} |coverage(s')|}$ and $w_p(s,S) = \frac{|params(s)|}{\max_{s' \in S} |params(s')|}$, and computes

$$numCandidates(s,S) = \max\left(1, \left\lceil b \cdot w_c(s,S) \cdot w_p(s,S) \right\rceil\right),$$

where b is the base mutation budget. New candidates are appended to the seed queue S for later selection. Each iteration generates candidates only within the per-seed mutation budget and time budget, so although S grows as new seeds are appended, the algorithm does not overrun its time budget. The scope of Algorithm 2 is a single unit-test fuzz driver, and ConfFuzz applies it independently to each fuzz driver. The coverage term preserves the standard greybox intuition that seeds with broader code coverage deserve more follow-up mutations. The parameter term adds an analogous preference for seeds whose executions span more of the configuration space. Seeds that cover more code and exercise more parameters therefore receive a larger share of the mutation budget, while seeds with smaller energy remain eligible for exploration. This budgeting rule is intentionally different from the saving rule above. The saving rule rewards marginal discoveries in the current round by admitting candidates that expose new branches or new parameters. The budgeting rule then prioritizes saved seeds in later rounds according to the breadth of behavior that those seeds already exercise. Under this design, a candidate that first reveals a new parameter receives additional follow-up fuzzing after it has been saved as a seed and selected for mutation in later rounds.

3.4 Debugging Support

When ConfFuzz detects test failures, users still need to localize the underlying bugs. Fully manual debugging of fuzzing results is both tedious and difficult, and one immediate challenge is identifying the failure-inducing configuration parameters among a large set of mutated parameters. ConfFuzz provides an automatic configuration-minimization tool to pinpoint failure-inducing parameters (§3.4.1). It also filters benign failures and groups failures with the same cause to assist users in processing a large number of failures (§3.4.2).

Algorithm 3: Fault localization algorithm of ConfFuzz

Input: test t, failing configuration C_{fail}, max attempt limit $MAX_{attempts}$, default configuration $C_{default}$
Output: minimal configuration that triggers the failure C_{min}

1 // C_{fail} *stores only the parameters that differ from* $C_{default}$
2 *failure, exception* $\leftarrow$ RUN$(t, C_{default} \uplus C_{fail})$
3 *attempts* $\leftarrow 0$
4 **while** *attempts* $< MAX_{attempts} \wedge$ SIZE$(C_{fail}) > 1$ **do**
5 *attempts* $\leftarrow$ *attempts* + 1
6 $C_{left}, C_{right} \leftarrow$ RANDOMSPLITINTOHALVES(C_{fail})
7 **if** RUN$(t, C_{default} \uplus C_{left})$ = *(failure, exception)* **then**
8 $C_{fail} \leftarrow C_{left}$
9 **else if** RUN$(t, C_{default} \uplus C_{right})$ = *(failure, exception)* **then**
10 $C_{fail} \leftarrow C_{right}$
11 $C_{min} \leftarrow$ REMOVEONEBYONE(t, C_{fail})
12 **return** C_{min}

3.4.1 Localizing failure-inducing configuration parameters. For a given failure, ConfFuzz uses Algorithm 3, similar to delta debugging [83], to localize the configuration parameters that trigger the failure. The idea is to apply randomized binary search to find a minimal set of parameters that can reproduce the failure.

Algorithm 3 repeatedly divides the set of mutated configuration parameters relative to $C_{default}$ into two subsets and tries to reproduce the failure with each subset (lines 7–10). If one subset successfully reproduces the failure, ConfFuzz continues the search with that subset. When the minimum triggering set spans both subsets, neither subset can reproduce the failure. ConfFuzz retries with a new partition until it reaches the maximum attempt limit (lines 4–6). At the end, ConfFuzz falls back to a more methodical one-by-one removal strategy (line 11): it removes one parameter, reruns the test, and adds the parameter back if the failure does not reproduce. If the failure stops reproducing after one parameter is removed, that parameter belongs to the failure-inducing set.

3.4.2 Filtering out benign failures. ConfFuzz filters out benign test failures that are unlikely to indicate bugs based on their symptoms. If the log message or exception type clearly indicates that the failure was caused by configuration, we treat the failure as one that the system already anticipated. ConfFuzz considers a test failure benign when the exception type and message clearly indicate that the system rejected the configuration as intended. More specifically, the failure is benign if the exception is either IllegalArgumentException or IOException with configuration- or parameter-related keywords in the error message, or if it is ComparisonFailure or AssertionError thrown from the system code, and the error message includes the name or value of at least one failure-inducing configuration parameter. These rules are neither sound nor complete, but we find them empirically effective in filtering out the benign test failures. Those benign failures are not false positives in the strict sense, but are better understood as misconfigurations [57], which are not the focus of ConfFuzz.

3.4.3 Grouping failures. Finally, ConfFuzz groups test failures by symptom so that users can inspect one representative failure from each group. Failures in the same group usually share the same root cause but were exposed by different tests. ConfFuzz groups two failures if they: (1) share the same exception type, (2)

Table 1: Cloud system projects used in our evaluation.

Project/Module	# Config Params	# Fuzz Drivers	LOC	SHA
HCommon/hadoop-common	892	998	253K	d37586
HDFS/hadoop-hdfs	2771	1890	362K	d37586
HBase/hbase-server	1348	980	431K	d38552
Alluxio/core	847	158	63K	bc8e1d
Hive/ql	669	1944	560K	6b05d6
MapReduce/client-core	270	232	63K	d37586
Yarn/hadoop-yarn-common	242	341	119K	d37586
Kylin/core-cube	150	105	21K	322ab6
Zeppelin/zeppelin-zengine	146	182	23K	f2b16f

are induced by the same set of parameters (§3.4.1), and (3) have the same stack trace excluding trace lines from external libraries outside the target project. In our evaluation, ConfFuzz reduced hundreds of thousands of failures to a few hundred groups (§5.1).

3.5 Implementation

We implement ConfFuzz in 9,800+ lines of Java code. ConfFuzz uses JQF [48] as the fuzzing backend for greybox coverage-guided fuzzing. We set the base mutation budget b of ConfFuzz to 50 by default, following JQF [48], and expose it as a configurable parameter. Besides JQF, ConfFuzz uses RgxGen [60] to generate randomized parameters that match a given regular expression; generation is deterministic for a given seed to support debugging.

ConfFuzz uses two maps for dynamic parameter tracking: a local map and a global map. The local map tracks the parameter set exercised in the current fuzzing round, while the global map tracks all parameters exercised across all fuzzing rounds. When a candidate execution is saved as a seed, ConfFuzz stores the parameter set recorded in the local map with that seed. ConfFuzz retrieves this stored parameter set when the same seed is selected again as a parent in Algorithm 2. We measure the overhead of dynamic parameter tracking in ConfFuzz by comparing test execution time with and without tracking, and the average overhead is only 3%.

4 Evaluation Setup

This section describes the experimental setup for evaluating ConfFuzz. The evaluation measures both the bug-finding effectiveness of ConfFuzz and the contribution of its main design components.

Evaluated Projects. We evaluate ConfFuzz on nine widely used open-source cloud systems: HBase, HCommon, HDFS, Alluxio, Hive, Kylin, MapReduce, Yarn, and Zeppelin. Table 1 summarizes the selected module for each project, together with the number of configuration parameters, the number of fuzz drivers, the module size, and the evaluated commit. All selected projects are highly configurable: the selected module exposes 2771 parameters in HDFS, 1348 in HBase, and 892 in HCommon; even the smaller systems expose many, such as 150 in Kylin and 146 in Zeppelin. For each project, we select a recent commit that passes continuous integration (CI). Similar to prior work [15, 57, 67], we evaluate one core module per project to keep the cost manageable while still covering the main configuration-API surface of each project.

Evaluated Variants. We evaluate four greybox variants: ConfFuzz, ConfFuzz$_{\text{Base}}$, ConfFuzz$_{\text{Track+Def}}$, and ConfFuzz$_{\text{Track}}$. ConfFuzz$_{\text{Base}}$ serves as the prior-work baseline. It reimplements the greybox strategy of ConfigFuzz [90], the state-of-the-art greybox fuzzer for program options, and adapts the strategy to Java-based cloud systems, because ConfigFuzz targets C/C++ programs and command-line options and cannot be applied directly to our setting. ConfFuzz$_{\text{Base}}$ starts from the default configuration, mutates parameters from the full configuration space, and relies only on code-coverage guidance. ConfFuzz$_{\text{Track+Def}}$ adds parameter-aware configuration mutation (§3.2). ConfFuzz$_{\text{Track}}$ keeps that mutation strategy but does not start from the default configuration, which isolates the effect of default initialization (§3.2.2). ConfFuzz further adds parameter-aware energy allocation (§3.3). Table 2 summarizes the components enabled in each variant. All four variants share the same API instrumentation, unit-test fuzz drivers, configuration-value generator, and failure processing. We do not include ECFuzz [31] as a blackbox baseline because ECFuzz drives system tests and uses unit tests as a *filter*: a generated configuration that fails a unit test is discarded before system testing. ConfFuzz takes the opposite view, using unit tests as fuzz drivers and treating configuration-induced unit-test failures as the bugs to expose. The failures that ConfFuzz targets are therefore, by construction, the ones that ECFuzz filters out. The two also fuzz at different granularities: ECFuzz fuzzes a whole system, whereas ConfFuzz fuzzes each unit-test driver independently, so ConfFuzz can run many more fuzzing executions within the same time budget. Prior work has already shown greybox fuzzing to be more effective than blackbox fuzzing [11, 20, 42, 49, 71, 73, 84, 90]. We also do not compare against configuration-testing techniques [57, 66, 78]: they test a specific configuration or code under a specific configuration rather than search the configuration space, so they are not fuzzers and thus not directly comparable baselines.

Table 2: Evaluated variants and the components enabled.

Variant	Code Coverage Guidance	Start From Default Config	Param-Aware Config Mutation	Param-Aware Energy Allocation
ConfFuzz$_{\text{Base}}$	✓	✓	✗	✗
ConfFuzz$_{\text{Track}}$	✓	✗	✓	✗
ConfFuzz$_{\text{Track+Def}}$	✓	✓	✓	✗
ConfFuzz	✓	✓	✓	✓

Fuzzing Setup. For each project, we start from the original unit test suite and keep only the tests that exercise at least one configuration parameter as fuzz drivers. To keep the cost manageable, we fuzz at most 1,000 tests per project: we use all eligible tests when a project has at most 1,000 such tests, and otherwise randomly sample 1,000. After discarding drivers that fail under the default configuration, this process yields 4,838 fuzz drivers across the nine projects. We fuzz each driver for 30 minutes using the same unit-test-driven fuzzing setup as UTopia [26]. Using unit tests as fuzz drivers gives high test execution speed: across all campaigns, ConfFuzz averages 20.57 executions per second (48.6 ms per execution). For comparison, ECFuzz [31] reports that system testing on the same class of cloud systems (including HCommon, HDFS, HBase, and Alluxio) executes 0.0371 testcases per second on average, about 27 seconds per execution, so ConfFuzz's unit-test drivers are roughly 550 times faster. One run of one variant therefore requires 2,419 VM-hours, and the 20 main campaigns in this study (four variants and five repetitions per variant) require 48,380 VM-hours before reruns caused by VM preemption and subsequent bug triage. We repeat each experiment five times to reduce the effect of fuzzing

randomness. The experiments run on Azure VMs [43] with dual-core CPUs, 8 GB RAM, Ubuntu v20.04.2, and Java v11.0.19. Our conclusions hold within a fixed time budget with unit-test drivers; we do not claim that they extrapolate to arbitrarily long campaigns. The 30-minute budget covers 4,838 drivers across nine systems, four variants, and five repetitions, amounting to 48,380 VM-hours (over 2,000 VM-days), well beyond the usual 24-hour by 10-trial fuzzer evaluations. Within this budget, ConfFuzz finds more bugs and explores more parameters than all other variants.

Accounting for Flakiness and Unexpected Crashes. We count only reproducible failures in the bug statistics. Before fuzzing, we run each candidate fuzz driver twice under the default configuration in the same JVM and discard drivers that fail in either run. During bug analysis, we exclude non-reproducible failures caused by flaky tests, JVM crashes, and VM crashes; we rerun campaigns affected by infrastructure failures.

Instrumentation Efforts. We instrument the configuration APIs of each evaluated project as described in §3.1. Although we are not developers of these projects, the instrumentation is straightforward because each project exposes unified configuration APIs: instrumenting the GET APIs and the configuration-class constructors took about half an hour per project and added 20 lines of code on average, with a minimum of 11 lines for Kylin and a maximum of 37 lines for Hive.

5 Evaluation Results

Our evaluation answers the following questions:

- RQ1: Does ConfFuzz find more bugs than its variants?
- RQ2: Does ConfFuzz explore more of the parameter space?
- RQ3: Does ConfFuzz achieve higher code coverage?
- RQ4: What bugs do ConfFuzz and its variants find?

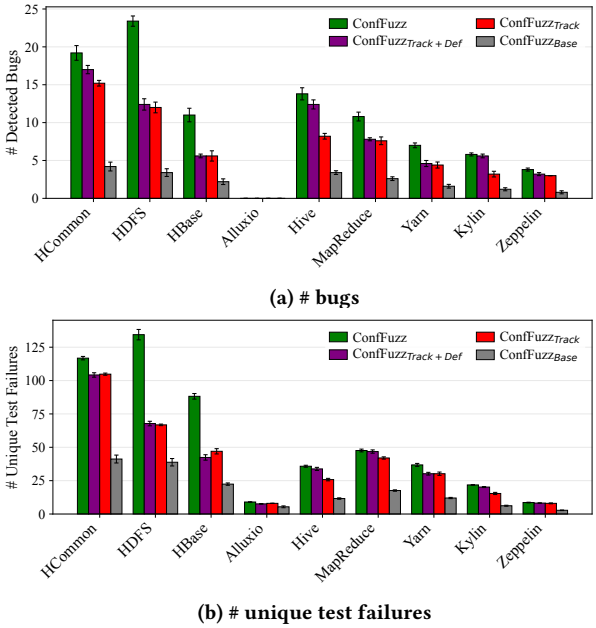
5.1 RQ1: Does ConfFuzz Find More Bugs?

We compare the bugs found by ConfFuzz with those found by its three variants across all nine evaluated projects to assess the overall effectiveness of ConfFuzz and the contribution of each design decision. Figure 4 reports the per-project average number of unique failures and bugs over the five runs described in §4, with standard error of the mean (SEM) to capture run-to-run variation. Table 3 complements the figure with 95% confidence intervals for the means, paired 95% confidence intervals for the differences between ConfFuzz and each variant, paired t-test p-values, and Cohen's d_z effect sizes.

Across the nine evaluated projects, ConfFuzz finds the most bugs in eight and ties only on Alluxio, where no evaluated variant finds a bug. Across all projects, ConfFuzz (leftmost, green, in Figure 4) finds 94.8 bugs on average, compared with 68.6 for ConfFuzz$_{\text{Track+Def}}$ (purple), 59.2 for ConfFuzz$_{\text{Track}}$ (red), and 19.4 for ConfFuzz$_{\text{Base}}$ (rightmost, grey). The largest gains appear on HDFS (23.4 vs. 12.4, 12.0, and 3.4) and HBase (11.0 vs. 5.6, 5.6, and 2.2).

Comparing ConfFuzz$_{\text{Base}}$ with ConfFuzz$_{\text{Track+Def}}$ isolates the effect of parameter-aware mutation (§3.2). Parameter-aware mutation raises the average bug count from 19.4 to 68.6. ConfFuzz$_{\text{Base}}$ mutates parameters from the full configuration space, which is extremely large in the evaluated projects, whereas ConfFuzz$_{\text{Track+Def}}$ mutates only the much smaller exercised-parameter subset, whose parameters are more likely to affect execution. The improvement is particularly large in systems with large configuration spaces such as HDFS and HBase, where randomly choosing an exercised parameter from the full space is unlikely and mutating unexercised parameters often reproduces the same execution path.

Comparing ConfFuzz$_{\text{Track+Def}}$ with ConfFuzz$_{\text{Track}}$ isolates the effect of starting from the default configuration (§3.2.2). The effect is limited: the two variants find a similar number of bugs on average (68.6 vs. 59.2). In HBase, for example, ConfFuzz$_{\text{Track+Def}}$ and ConfFuzz$_{\text{Track}}$ both find 5.6 bugs on average. Unit tests often exercise only a small subset of configuration parameters, so starting



(a) # bugs

(b) # unique test failures

Figure 4: The number of bugs and unique test failures found on all evaluated projects (averaged over five runs).

Table 3: Formal statistical summary for RQ1 across five runs.

Metric	ConfFuzz	Track+Def	Track	Base
# Bugs	94.8 [90.6, 99.0]	68.6 [66.7, 70.5]	59.2 [54.1, 64.4]	19.4 [17.0, 21.8]
# Fail.	499.0 [486.2, 511.8]	361.2 [349.3, 373.1]	348.0 [340.8, 355.2]	158.0 [150.8, 165.2]

(a) Mean totals with 95% confidence intervals.

Metric	Variant	Δ	95% CI	p-value	d_z
# Bugs	Track+Def	+26.2	[22.3, 30.1]	4.7e-05	8.41
	Track	+35.6	[27.8, 43.4]	2.3e-04	5.64
	Base	+75.4	[71.7, 79.1]	5.7e-07	25.42
# Fail.	Track+Def	+137.8	[115.9, 159.7]	6.3e-05	7.81
	Track	+151.0	[143.9, 158.1]	4.9e-07	26.49
	Base	+341.0	[330.0, 352.0]	1.1e-07	38.61

(b) Paired comparisons on total counts. Δ denotes ConfFuzz minus the comparing variant.

from a random configuration over that subset does not necessarily prevent the fuzzer from reaching deep code paths and can sometimes even expose paths that exercise more parameters than the default configuration does (see §5.2). This suggests that starting from the default configuration, although common in recent configuration fuzzing work [29, 31, 90], is not a critical design choice for configuration fuzzing in highly configurable cloud systems as long as parameter-aware mutation is in place to steer fuzzing toward exercised parameters.

Comparing ConfFuzz with ConfFuzz$_{\text{Track+Def}}$ isolates the effect of parameter-aware energy allocation (§3.3). Parameter-aware energy allocation raises the average bug count from 68.6 to 94.8, because it steers fuzzing toward seeds that exercise more parameters and therefore increases the chance of reaching bugs gated by newly exercised parameters; §5.2 confirms that ConfFuzz indeed explores more parameters.

Unique test failures show the same ranking as unique bugs, with ConfFuzz leading on every project and showing the largest margins on HDFS and HBase. ConfFuzz triggers 499.0 unique test failures on average, compared with 361.2 for ConfFuzz$_{\text{Track+Def}}$, 348.0 for ConfFuzz$_{\text{Track}}$, and 158.0 for ConfFuzz$_{\text{Base}}$. HDFS and HBase again show the largest margins: ConfFuzz reaches 134.4 and 88.2 unique failures, while the three variants are at 67.8/66.8/38.8 and 42.4/47.0/22.4, respectively.

Figure 5 reports the union of unique bugs across five runs, where a bug counts for a variant if that variant exposes it in at least one run. We omit ConfFuzz$_{\text{Base}}$ from the Venn diagram because every bug it finds is also found by another variant and it therefore adds nothing to the union. Together, ConfFuzz and its variants expose 125 unique bugs. ConfFuzz finds 118 unique bugs across the nine evaluated systems, whereas ConfFuzz$_{\text{Track+Def}}$ and ConfFuzz$_{\text{Track}}$ find 83 and 76, respectively. ConfFuzz alone contributes 38 bugs that neither tracking variant finds, whereas the two tracking variants contribute only 2 and 1 exclusive bugs. Seven bugs are not found by ConfFuzz but are found by one or both tracking variants, most likely because ConfFuzz prioritizes seeds that exercise more parameters (§3.3) and can thus miss seeds that exercise fewer parameters but still trigger unique bugs. This tradeoff is worthwhile overall, given the much larger exclusive and mean bug counts reported above.

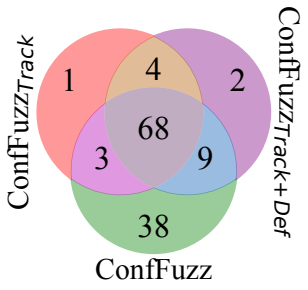


Figure 5: Number of unique bugs found in nine projects.

Formal statistics support the same conclusion. Table 3a shows that ConfFuzz reaches a mean total bug count of 94.8 (95% CI [90.6, 99.0]) and a mean total failure count of 499.0 (95% CI [486.2, 511.8]), and Table 3b shows that ConfFuzz gains 26.2 bugs and 137.8 failures on average over ConfFuzz$_{\text{Track+Def}}$, with larger gains over the other two variants. Every paired comparison is significant under the paired t-test ($p < 0.001$), every 95% confidence interval for the difference excludes zero, and every effect size is large.

5.2 RQ2: Does ConfFuzz Explore More of the Configuration Parameter Space?

We next evaluate whether ConfFuzz explores more of the configuration parameter space, which can explain why it finds more bugs. We compare ConfFuzz with ConfFuzz$_{\text{Track+Def}}$ and ConfFuzz$_{\text{Track}}$, because ConfFuzz$_{\text{Base}}$ does not track exercised parameters and therefore has no corresponding notion of explored parameter space.

Figure 6 shows, for each project, the ratio between the number of exercised configuration parameters during fuzzing and the number exercised by the initial seed, with shaded regions showing run-to-run variation. A ratio of 1.0 means that fuzzing never moves beyond the initial parameter set during the 30-minute campaign. ConfFuzz finishes with the highest ratio in every project, which shows that parameter-aware mutation and parameter-aware energy allocation together help ConfFuzz explore more parameters than the two tracking variants.

The largest gaps appear in projects where ConfFuzz also gains the most bugs: on HDFS it grows the exercised-parameter set to 39.5X the initial size, versus 25.6X and 25.0X for the two tracking variants; the corresponding ratios are 6.0X, 4.7X, and 4.1X on HBase and 2.18X, 1.37X, and 1.41X on HCommon. MapReduce, Hive, and Yarn show smaller but consistent gains. Kylin, Zeppelin, and Alluxio stay close to 1.0 for all variants, indicating that their fuzz drivers reveal few additional parameters beyond the initial seed; even there, ConfFuzz reaches the largest explored parameter count.

ConfFuzz$_{\text{Track+Def}}$ and ConfFuzz$_{\text{Track}}$ show similar growth trends and final ratios in most projects, suggesting that starting from the default configuration has no consistent effect on the explored parameter space. Only ConfFuzz, with parameter-aware energy allocation (§3.3), consistently explores more parameters: once a mutation exposes a new parameter, ConfFuzz records it, saves the corresponding seed, and allocates more effort to expanding that region of the configuration space.

Beyond ratios, we also report absolute exercised-parameter counts. Across all ConfFuzz driver executions, the median count grows from 22 in the initial seed to 35 after 30-minute fuzzing, and the mean count grows from 118.8 to 194.2, an increase of 75.4 exercised parameters on average. In total, 60.1% of drivers exercise at least one new parameter during fuzzing.

5.3 RQ3: Does ConfFuzz Achieve Higher Code Coverage?

Having shown that ConfFuzz explores more of the configuration parameter space (§5.2), we now examine whether this translates into higher code coverage. We measure branch coverage achieved by ConfFuzz and compare it with ConfFuzz$_{\text{Track+Def}}$, ConfFuzz$_{\text{Track}}$, and ConfFuzz$_{\text{Base}}$. Following [26], we report *union* branch coverage: for each project, we union the branches covered across all of the project's fuzz drivers, averaged over five runs. Figure 7 shows the union branch coverage at the end of 30-minute fuzzing, normalizing each project to ConfFuzz$_{\text{Track}}$ = 100% so that projects ranging from about 700 to 27,000 branches are comparable.

ConfFuzz attains the highest union branch coverage on eight of nine projects against ConfFuzz$_{\text{Track}}$ and ConfFuzz$_{\text{Track+Def}}$, and on all nine projects against ConfFuzz$_{\text{Base}}$. Taking the geometric mean of the per-project coverage ratios, ConfFuzz covers about

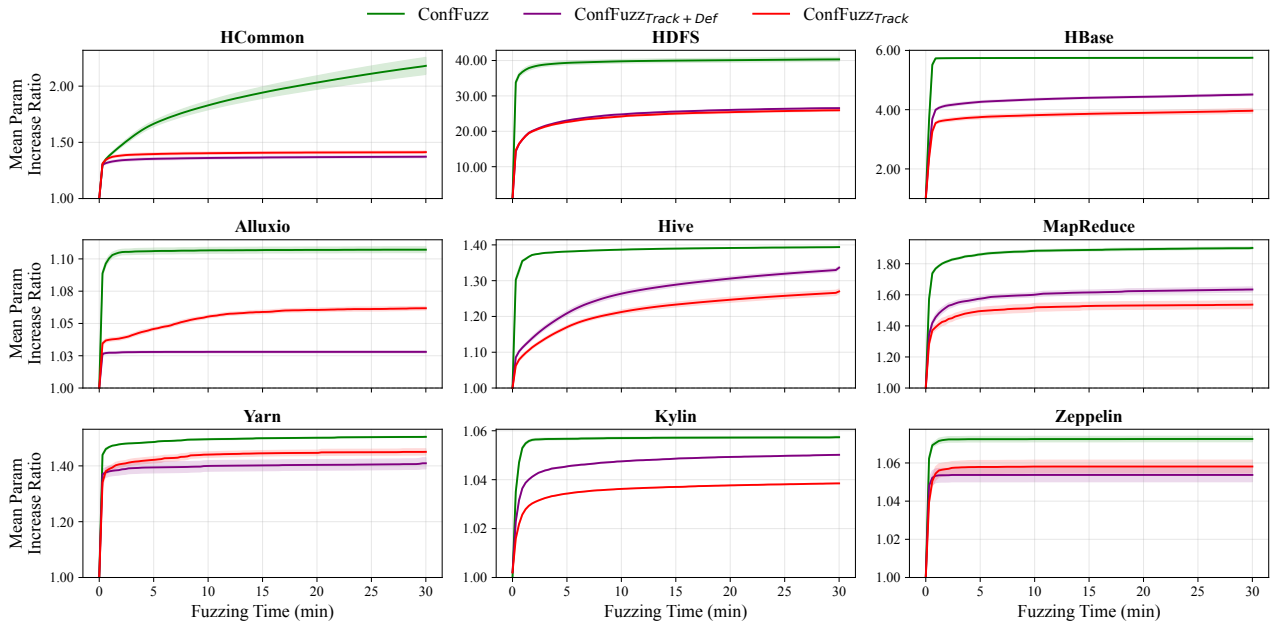


Figure 6: Mean growth ratio of exercised configuration-parameter count per variant and project, averaged across five runs and normalized to the count exercised by the initial seed (every curve starts at 1.0). Shaded regions show variation across five runs.

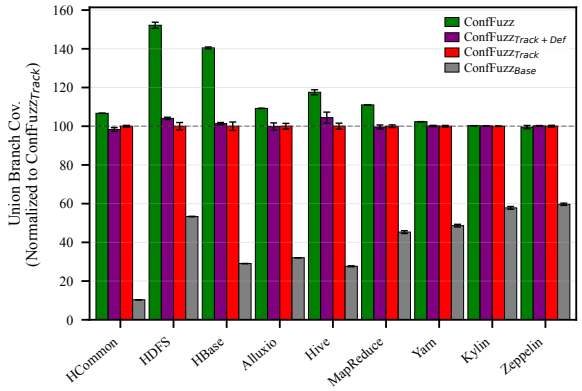


Figure 7: Union branch coverage [26] at 30 minutes, averaged over five runs and normalized to ConfFuzz$_{Track}$.

14.2% more branches than ConfFuzz$_{Track}$, about 13.3% more than ConfFuzz$_{Track+Def}$, and roughly 3.2 times the coverage of ConfFuzz$_{Base}$. The largest gains concentrate on the high-parameter-count projects (HDFS +52.2%, HBase +40.5%, Hive +17.5% over ConfFuzz$_{Track}$), while low-parameter-count projects show small differences (Kylin +0.3%, Yarn +2.3%). The single loss is on Zeppelin, the smallest-parameter-count project, where ConfFuzz is 0.4% below ConfFuzz$_{Track}$ and 0.6% below ConfFuzz$_{Track+Def}$.

The coverage trend across projects from Figure 7 mirrors the parameter-exploration trend from Figure 6: the projects where ConfFuzz achieves the largest parameter exploration are exactly the ones with the largest coverage gains, because exercising more configuration parameters generally exercises more code. On projects with few configuration parameters, the three ConfFuzz variants perform almost identically because there is little parameter space to explore. ConfFuzz$_{Base}$ trails all three variants on every project, since it mutates parameters chosen uniformly at random from the whole parameter set rather than from the exercised set, which shows the benefit of restricting mutation to exercised parameters.

Table 4: Summary of bugs found by ConfFuzz variants.

ID	Bug Type	Count	Percentage
B1	Crashing with no message	41	32.8%
B2	NullPointerException	25	20.0%
B3	Semantic bugs	23	18.4%
B4	Arithmetic bugs	19	15.2%
B5	Bugs in test code	15	12.0%
B6	Test polluter	2	1.6%
Σ		125	100.00%

5.4 RQ4: Summary of New Bugs Found

Table 4 categorizes the bugs found by ConfFuzz and its variants, and Table 5 presents one case study per type. ConfFuzz exposes many crashing bugs that lead to runtime exceptions, as well as semantic bugs that cause inconsistent behavior [5], infinite loops [6], and resource leaks [3]. We define a bug as a defect with an observable failure symptom, and we only report bugs that reproduce on the test with the offending configuration. We have reported 77 of the 125 bugs to the developers. So far, 26 have received developer responses: 14 have been fixed, 12 have been confirmed, and the remaining reports are pending developer response.

Crashing with no message. Here the system crashes while parsing the fuzzed configuration but emits either no message or a cryptic one, leaving users without actionable diagnosis and even sending them in the wrong direction [79–81]. In HDFS-17101 (Table 5), when the value of parameter `fs.permissions.umask-mode` is 612, HDFS fails to perform IO operations without any error message, which makes diagnosis difficult.

NullPointerException. Mutated configurations often enable a feature without initializing all required state, a corner case that

Table 5: Examples of bugs found by ConfFuzz (of different types).

Type	Bug ID	Exception	Fuzzed Parameters (*parameter=value*)	Fuzzed Test
B1	HDFS-17101	AssertionError	fs.permissions.umask-mode=612	TestDecommissioningStatusWithBackoffMonitor #testDecommissionDeadDN
B2	HBASE-28452	NullPointer Exception	hbase.client.default.rpc.codec="org.apache. hadoop.hbase.regionserver.wal.SecureWALCellCodec"	AbstractTestIPC #testLauncher
B3	HDFS-17105	IndexOutOfBounds Exception	dfs.namenode.max.extra.edits. segments.retained=-1974676133	TestNNStorageRetentionManager #testNoLogs
B3	MAPREDUCE-7448	AssertionError	mapreduce.fileoutputcommitter. cleanup.skipped=true	TestFileOutputCommitter #testCommitterWithDuplicatedCommitV1
B4	HBASE-27993	Arithmetic Exception	hbase.regionserver.logroll.multiplier=0.0, hbase.wal.provider=multiwal	TestWALMethods #testGetSplitEditFilesSorted
B5	HADOOP-18814	ClassCast Exception	hadoop.security.authentication="kerberos"	TestRPC #testReaderExceptions
B6	MAPREDUCE-7443	IOException	fs.permissions.umask-mode=243	TestFileOutputCommitter #testRecoveryUpgradeV1V2

default testing can miss. In HBASE-28452 (Table 5), setting HBase's `codec` parameter to `SecureWALCellCodec` throws NullPointerException: the codec class lacks an expected constructor, so the RPC client never initializes its thread-pool executor, yet the `finally` block still calls `shutdown()` on it.

Semantic bugs. ConfFuzz also finds semantic bugs deeply buried in the code and tightly coupled with specific system features. In HDFS-17105 (Table 5), setting `dfs.namenode.max.extra.edits.segments.retained` to a negative value crashes the HDFS NameNode's storage-retention manager with an IndexOutOfBoundsException: the purge loop removes edit logs while the list size exceeds the retention count, so a negative value drains the list past empty and then accesses index 0. In MAPREDUCE-7448 (Table 5), enabling a MapReduce option that skips cleanup of the _temporary folder also prevents FileOutputCommitter from writing the _SUCCESS marker that blocks repeated commits, so later commit attempts run `mergePaths` again, duplicate task output, and corrupt data.

Arithmetic bugs. Arithmetic bugs usually arise from calculations that do not account for unexpected parameter values. In HBASE-27993 (Table 5), setting `hbase.regionserver.logroll.multiplier` to 0.0 causes `logRollSize` to become zero, which triggers a division-by-zero exception and crashes the HBase RegionServer.

Bugs in test code. ConfFuzz also exposes bugs in test code. In HADOOP-18814 (Table 5), the HCommon test casts every caught exception to RemoteException, when `hadoop.security.authentication` is set to `kerberos` and authentication fails with an IOException, the test crashes with ClassCastException.

Test polluter. ConfFuzz finds test polluters [70] that cause configuration-related flaky tests. In MAPREDUCE-7443 (Table 5), the test pollutes the disk because it does not delete a directory created during execution, so after ConfFuzz mutates `fs.permissions.umask-mode` to an umask that revokes write permission, all subsequent runs fail with permission denials.

6 Related Work

Configuration Testing and Configuration Fuzzing. Configuration testing validates software behavior under non-default configurations by running tests in the presence of configuration values and changes [15, 57, 66, 67, 75, 78]. Recent work brought it to cloud systems by linking configuration parameters with developer-written unit tests [57, 78], and improved its practicality through test prioritization and selection [15, 65, 67]. Related work also analyzes configuration parameters and bugs through option extraction, parameter diagnosis, and bug detection [38, 52, 68, 69, 88].

Configuration fuzzing automatically mutates configuration values to expose crashes and semantic failures, and prior work has fuzzed configuration files and command-line interfaces [18, 19, 84]. Recent techniques include ConfigFuzz [90] and POWER [29] for program options and ECFuzz for large-scale cloud systems [31]. Subsequent option-fuzzing work adds whitebox analysis, LLM-guided prediction, option-file interleaving, dynamic scheduling, performance and variability awareness, and configuration-dependency fault injection [1, 7, 14, 17, 30, 56, 61, 63, 64], and benefits from work inferring parameter semantics from code, documentation, or learned patterns [10, 32, 86]. ConfFuzz is also greybox, but targets API-level configuration objects, uses unit tests as fuzz drivers, and adds runtime parameter information as guidance.

Greybox Fuzzing with Richer Guidance. Greybox fuzzing began with code-coverage guidance [11, 20, 42] and later incorporated additional signals when coverage alone did not capture the structure of the search space: custom generators that preserve input structure and validity [24, 46, 48, 49], target distance [12], rare-path information [54], static or dynamic data-flow information [16, 28], domain-specific runtime signals such as memory usage or typestate [62, 72], and strategies learned from prior executions [33, 39, 41, 82]. ConfFuzz follows the same principle: its domain-specific signal is the set of configuration parameters exercised by each execution, which matters because a workload usually reads only a small subset, and that subset can change as mutations open new control flow.

7 Conclusion

This paper presents ConfFuzz, a parameter-aware greybox configuration fuzzer for highly configurable cloud systems. ConfFuzz uses existing unit tests as fuzz drivers, dynamically tracks the configuration parameters exercised in each execution, restricts later mutation to those parameters, and allocates fuzzing energy using both code coverage and parameter coverage. We evaluated ConfFuzz on nine widely used cloud systems with 4,838 fuzz drivers and three variants of its design. The results show that code coverage alone is insufficient here: effective search also requires parameter-aware mutation and energy allocation. Compared with a code-coverage-only greybox baseline that mutates across the full configuration space, ConfFuzz finds substantially more bugs and achieves the highest parameter coverage in all nine systems, exposing 125 configuration-related bugs across all campaigns.

Acknowledgments

This work was partially supported by the National Science Foundation under grant CNS-2145295.

Data Availability Statement

All code and data are at https://doi.org/10.5281/zenodo.19248494.

References

[1] Md Tahmid Ahmed, Abhishek Dev, and Shiyi Wei. 2026. Variability-Aware Fuzzing. In *ICSE*.

[2] Apache Commons. n.d.. Apache Commons Configuration Interface. https://commons.apache.org/proper/commons-configuration/javadocs/v1.10/apidocs/index.html.

[3] Apache Hadoop. n.d.. Bad ipc.client.connection.idle-scan-interval.ms cause resource leaks. https://issues.apache.org/jira/browse/HADOOP-18800.

[4] Apache Hadoop. n.d.. Hadoop Configuration. https://hadoop.apache.org/docs/current/api/org/apache/hadoop/conf/Configuration.html.

[5] Apache Hadoop. n.d.. Inconsistent Behavior for FileOutputCommitter V1. https://issues.apache.org/jira/browse/MAPREDUCE-7448.

[6] Apache Hive. n.d.. Inifite array growth when optimized hashtable size is set to 0. https://issues.apache.org/jira/browse/HIVE-27519.

[7] Haesue Baik, Chenyang Yang, Vasudev Vikram, Pooyan Jamshidi, Rohan Padhye, and Christian Kaestner. 2025. Differential Performance Fuzzing of Configuration Options. In *SBFT*. doi:10.1109/sbft66712.2025.00014

[8] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. doi:10.1007/978-3-031-01761-2

[9] Farnaz Behrang, Myra B. Cohen, and Alessandro Orso. 2015. Users Beware: Preference Inconsistencies Ahead. In *ESEC/FSE*. doi:10.1145/2786805.2786869

[10] Ranjita Bhagwan, Sonu Mehta, Arjun Radhakrishna, and Sahil Garg. 2021. Learning Patterns in Configuration. In *ASE*. doi:10.1109/ase51524.2021.9678525

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *CCS*. doi:10.1145/2976749.2978428

[12] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *CCS*. doi:10.1145/3243734.3243849

[13] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. 2020. Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems. In *ESEC/FSE*. doi:10.1145/3368089.3409727

[14] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Zhen Yan, and Yu Jiang. 2025. CAFault: Enhance Fault Injection Technique in Practical Distributed Systems via Abundant Fault-Dependent Configurations. In *ATC*.

[15] Runxiang Cheng, Lingming Zhang, Darko Marinov, and Tianyin Xu. 2021. Test-Case Prioritization for Configuration Testing. In *ISSTA*. doi:10.1145/3460319.3464810

[16] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In *ASE*. doi:10.1109/ase51524.2021.9678888

[17] Lang Chu, Minhuan Huang, Xiang Li, Yuanping Nie, Wenyu Zhen, and Qian Yan. 2025. OptionFuzz: The Fuzzer with Coverage-Guided Dynamic Configuration Scheduling. In *ICCCS*. doi:10.1109/icccs65393.2025.11069832

[18] Huning Dai, Christian Murphy, and Gail Kaiser. 2010. Configuration Fuzzing for Software Vulnerability Detection. In *ARES*. doi:10.1109/ares.2010.22

[19] Huning Dai, Christian Murphy, and Gail Kaiser. 2010. CONFU: Configuration Fuzzing Testing Framework for Software Vulnerability Detection. In *IJSSE*. doi:10.4018/978-1-4666-1580-9.ch009

[20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *WOOT*.

[21] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. 2011. Fate and Destini: A Framework for Cloud Recovery Testing. In *NSDI*.

[22] 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 *SoCC*. doi:10.1145/2987550.2987583

[23] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. 2008. EIO: Error Handling is Occasionally Correct. In *FAST*.

[24] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *USENIX Sec*.

[25] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code Coverage at Google. In *ESEC/FSE*. doi:10.1145/3338906.3340459

[26] B. Jeong, J. Jang, H. Yi, J. Moon, J. Kim, I. Jeon, T. Kim, W. Shim, and Y. Hwang. 2023. UTopia: Automatic Generation of Fuzz Driver using Unit Tests. In *S&P*. doi:10.1109/sp46215.2023.10179394

[27] Lorenzo Keller, Prasang Upadhyaya, and George Candea. 2008. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In *DSN*. doi:10.1109/dsn.2008.4630084

[28] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. DAFL: Directed Grey-box Fuzzing Guided by Data Dependency. In *USENIX Sec*.

[29] Ahcheong Lee, Irfan Ariq, Yunho Kim, and Moonzoo Kim. 2022. POWER: Program Option-Aware Fuzzer for High Bug Detection Ability. In *ICST*. doi:10.1109/icst53961.2022.00032

[30] Ahcheong Lee, Youngseok Choi, Shin Hong, Yunho Kim, Kyutae Cho, and Moonzoo Kim. 2025. ZigZagFuzz: Interleaved Fuzzing of Program Options and Files. *TOSEM* (2025). doi:10.1145/3697014

[31] 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 *ICSE*. doi:10.1145/3597503.3623315

[32] 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. *TR* (2018). doi:10.1109/tr.2018.2865962

[33] Siqi Li, Yun Lin, Xiaofei Xie, Yuekang Li, Xiaohong Li, Weimin Ge, Yang Liu, and Jin Song Dong. 2021. A First Look at the Effect of Deep Learning in Coverage-guided Fuzzing. In *ASE*. doi:10.1109/ase51524.2021.9678794

[34] Xinyu Lian, Yinfang Chen, Runxiang Cheng, Jie Huang, Parth Thakkar, Minjia Zhang, and Tianyin Xu. 2025. Large Language Models as Configuration Validators. In *ICSE*. doi:10.1109/icse55347.2025.00017

[35] Lightbend. n.d.. Typesafe Configuration Interface. https://www.javadoc.io/doc/com.typesafe/config/1.3.1/com/typesafe/config/Config.html.

[36] Max Lillack, Christian Kästner, and Eric Bodden. 2014. Tracking Load-time Configuration Options. In *ASE*. doi:10.1145/2642937.2643001

[37] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. 2019. What Bugs Cause Production Cloud Incidents?. In *HotOS*. doi:10.1145/3317550.3321438

[38] Sixiang Ma, Fang Zhou, Michael D Bond, and Yang Wang. 2021. Finding Heterogeneous-Unsafe Configuration Parameters in Cloud Systems. In *EuroSys*. doi:10.1145/3447786.3456250

[39] Björn Mathis, Rahul Gopinath, and Andreas Zeller. 2020. Learning Input Tokens for Effective Fuzzing. In *ISSTA*. doi:10.1145/3395363.3397348

[40] Ben Maurer. 2015. Fail at Scale: Reliability in the Face of Rapid Change. *CACM* (2015). doi:10.1145/2814330

[41] Raveendra Kumar Medicherla, Raghavan Komondoor, and Abhik Roychoudhury. 2020. Fitness Guided Vulnerability Detection with Greybox Fuzzing. In *ICSE*. doi:10.1145/3387940.3391457

[42] Michal Zalewski. 2014. AFL. http://lcamtuf.coredump.cx/afl.

[43] Microsoft. n.d.. Azure Linux virtual machines pricing. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/.

[44] Mukelabai Mukelabai, Damir Nešić, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *ASE*. doi:10.1145/3238147.3238201

[45] Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. 2004. Understanding and Dealing with Operator Mistakes in Internet Services. In *OSDI*.

[46] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. 2020. Generating Highly-structured Input Data by Combining Search-based Testing and Grammar-based Fuzzing. In *ASE*. doi:10.1145/3324884.3418930

[47] David Oppenheimer, Archana Ganapathi, and David A. Patterson. 2003. Why Do Internet Services Fail, and What Can Be Done About It?. In *USITS*.

[48] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-Guided Property-Based Testing in Java. In *ISSTA*. doi:10.1145/3293882.3339002

[49] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *ISSTA*. doi:10.1145/3293882.3330576

[50] Rahul Potharaju, Joseph Chan, Luhui Hu, Cristina Nita-Rotaru, Mingshi Wang, Liyuan Zhang, and Navendu Jain. 2015. ConfSeer: Leveraging Customer Support Knowledge Bases for Automated Misconfiguration Detection. In *VLDB*. doi:10.14778/2824032.2824079

[51] Ariel Rabkin and Randy Katz. 2011. Precomputing Possible Configuration Error Diagnosis. In *ASE*. doi:10.1109/ase.2011.6100053

[52] Ariel Rabkin and Randy Katz. 2011. Static Extraction of Program Configuration Options. In *ICSE*. doi:10.1145/1985793.1985812

[53] Ariel Rabkin and Randy Katz. 2013. How Hadoop Clusters Break. *SM* (2013). doi:10.1109/ms.2012.73

[54] Seemanta Saha, Laboni Sarker, Md Shafiuzzaman, Chaofan Shou, Albert Li, Ganesh Sankaran, and Tevfik Bultan. 2023. Rare Path Guided Fuzzing. In *ISSTA*. doi:10.1145/3597926.3598136

[55] Jonathan Shieber. [n. d.]. Facebook blames a server configuration change for yesterday's outage. https://techcrunch.com/2019/03/14/facebook-blames-a-misconfigured-server-for-yesterdays-outage/.

[56] Momoko Shiraishi, Yinzhi Cao, and Takahiro Shinagawa. 2026. PILOT: Command-Line Interface Fuzzing via Path-Guided, Iterative Large Language Model Prompting. In *S&P*. doi:10.1109/sp63933.2026.00211

[57] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing Configuration Changes in Context to Prevent Production Failures. In *OSDI*.

[58] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic Configuration Management at Facebook. In *SOSP*. doi:10.1145/2815400.2815401

[59] Suresh Thummalapenta, Madhuri R. Marri, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2011. Retrofitting unit tests for parameterized unit testing. In *FASE*. doi:10.1007/978-3-642-19811-3_21

[60] Vladislavs Varslavans. n.d.. Regex: generate matching and non-matching strings. https://github.com/curious-odd-man/RgxGen/.

[61] Dawei Wang, Geng Zhou, Li Chen, Dan Li, and Yukai Miao. 2024. ProphetFuzz: Fully Automated Prediction and Fuzzing of High-Risk Option Combinations with Only Documentation via Large Language Model. In *CCS*. doi:10.1145/3658644.3690231

[62] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided Fuzzer for Discovering Use-After-Free Vulnerabilities. In *ICSE*. doi:10.1145/3377811.3380386

[63] Kelin Wang, Mengda Chen, Liang He, Purui Su, Yan Cai, Jiongyi Chen, Bin Zhang, Chao Feng, and Chaojing Tang. 2024. OSmart: Whitebox Program Option Fuzzing. In *CCS*. doi:10.1145/3658644.3690228

[64] Kelin Wang, Mengda Chen, Liang He, Yanhao Wang, Purui Su, Jiongyi Chen, Yan Cai, Bin Zhang, Chao Feng, Chaojing Tang, and Guojun Peng. 2026. OSmartPro: A Large Language Model-Assisted Option Fuzzing Approach. *Cybersecurity* (2026). doi:10.1186/s42400-026-00557-8

[65] Shuai Wang. 2026. *Unified Automated Reliability Testing for Configurable Cloud Systems*. Ph. D. Dissertation. University of Illinois Urbana-Champaign.

[66] Shuai Wang, Xinyu Lian, Qingyu Li, Darko Marinov, and Tianyin Xu. 2024. Ctest4J: A Practical Configuration Testing Framework for Java. In *FSE Companion*. doi:10.1145/3663529.3663799

[67] Shuai Wang, Xinyu Lian, Darko Marinov, and Tianyin Xu. 2023. Test Selection for Unified Regression Testing. In *ICSE*. doi:10.1109/icse48619.2023.00145

[68] Teng Wang, Haochen He, Xiaodong Liu, Shanshan Li, Zhouyang Jia, Yu Jiang, Qing Liao, and Wang Li. 2023. ConfTainter: Static Taint Analysis for Configuration Options. In *ASE*. doi:10.1109/ase56229.2023.00067

[69] Teng Wang, Zhouyang Jia, Shanshan Li, Si Zheng, Yue Yu, Erci Xu, Shaoliang Peng, and Xiangke Liao. 2023. Understanding and Detecting On-The-Fly Configuration Bugs. In *ICSE*. doi:10.1109/icse48619.2023.00062

[70] Anjiang Wei, Pu Yi, Zhengxi Li, Tao Xie, Darko Marinov, and Wing Lam. 2022. Preempting Flaky Tests via Non-Idempotent-Outcome Tests. In *ICSE*. doi:10.1145/3510003.3510170

[71] Moshi Wei, Yuchao Huang, Jinqiu Yang, Junjie Wang, and Song Wang. 2023. CoCoFuzzing: Testing Neural Code Models With Coverage-Guided Fuzzing. In *TR*. doi:10.1109/tr.2022.3208239

[72] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MemLock: Memory Usage Guided Fuzzing. In *ICSE*. doi:10.1145/3377811.3380396

[73] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal Fuzzing with Large Language Models. In *ICSE*. doi:10.1145/3597503.3639121

[74] Chengcheng Xiang, Haochen Huang, Andrew Yoo, Yuanyuan Zhou, and Shankar Pasupathy. 2020. PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations. In *ATC*.

[75] Chengcheng Xiang, Li Zhong, Eric Mugnier, Nathaniel Nguyen, Yuanyuan Zhou, and Tianyin Xu. 2025. Testing Access-Control Configuration Changes for Web Applications. *arXiv* (2025). doi:10.48550/arXiv.2505.12770

[76] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, You Have Given Me Too Many Knobs! Understanding and Dealing with Over-Designed Configuration in System Software. In *ESEC/FSE*. doi:10.1145/2786805.2786852

[77] 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 *OSDI*.

[78] Tianyin Xu and Owolabi Legunsen. 2019. Configuration Testing: Testing Configuration Values as Code and with Code. *arXiv* (2019). doi:10.48550/arXiv.1905.12195

[79] Tianyin Xu, Han Min Naing, Le Lu, and Yuanyuan Zhou. 2017. How Do System Administrators Resolve Access-Denied Issues in the Real World?. In *CHI*. doi:10.1145/3025453.3025999

[80] Tianyin Xu, Vineet Pandey, and Scott Klemmer. 2016. An HCI View of Configuration Problems. *arXiv* (2016). doi:10.48550/arXiv.1601.01747

[81] 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 *SOSP*. doi:10.1145/2517349.2522727

[82] Guixin Ye, Tianmin Hu, Zhanyong Tang, Zhenye Fan, Shin Hwei Tan, Bo Zhang, Wenxiang Qian, and Zheng Wang. 2023. A Generative and Mutational Approach for Synthesizing Bug-Exposing Test Cases to Guide Compiler Fuzzing. In *ESEC/FSE*. doi:10.1145/3611643.3616332

[83] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *ESEC*. doi:10.1145/318774.318946

[84] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2021. Testing Configurations. https://www.fuzzingbook.org/html/ConfigurationFuzzer.html.

[85] Jialu Zhang, Ruzica Piskac, Ennan Zhai, and Tianyin Xu. 2021. Static Detection of Silent Misconfigurations with Deep Interaction Analysis. In *OOPSLA*. doi:10.1145/3485517

[86] Jiaqi Zhang, Lakshmi Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *ASPLOS*. doi:10.1145/2644865.2541983

[87] Sai Zhang and Michael D. Ernst. 2013. Automated Diagnosis of Software Configuration Errors. In *ICSE*. doi:10.1109/icse.2013.6606577

[88] Sai Zhang and Michael D. Ernst. 2014. Which Configuration Option Should I Change?. In *ICSE*. doi:10.1145/2568225.2568251

[89] Sai Zhang and Michael D. Ernst. 2015. Proactive Detection of Inadequate Diagnostic Messages for Software Configuration Errors. In *ISSTA*. doi:10.1145/2771783.2771817

[90] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. 2023. Fuzzing Configurations of Program Options. In *TOSEM*. doi:10.1145/3580597

Received 2026-03-26; accepted 2026-06-18