Understanding and Detecting On-the-Fly Configuration Bugs

Teng Wang[†‡], Zhouyang Jia[†‡], Shanshan Li[†*], Si Zheng[†], Yue Yu[†], Erci Xu[†], Shaoliang Peng[§], Xiangke Liao[†]
[†]National University of Defense Technology, Changsha, China
[§]Hunan University, Changsha, China
{wangteng13, jiazhouyang, shanshanli, yuyue, xuerci, xkliao} @nudt.edu.cn, si.zheng1009@gmail.com, slpeng@hnu.edu.cn

***Abstract*—Software systems introduce an increasing number of configuration options to provide flexibility, and support updating the options on the fly to provide persistent services. This mechanism, however, may affect the system reliability, leading to unexpected results like software crashes or functional errors. In this paper, we refer to the bugs caused by on-the-fly configuration updates as on-the-fly configuration bugs, or OCBugs for short.**

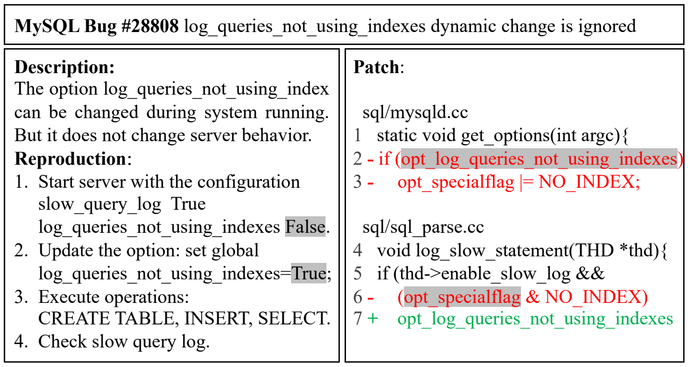
In this paper, we conducted the first in-depth study on 75 real-world OCBugs from 5 widely used systems to understand the symptoms, root causes, and triggering conditions of OCBugs. Based on our study, we designed and implemented PARACHUTE, an automated testing framework to detect OCBugs. Our key insight is that the value of one configuration option, either loaded at the startup phase or updated on the fly, should have the same effects on the target program. PARACHUTE generates tests for on-the-fly configuration updates by mutating the existing tests and conducts differential analysis to identify OCBugs. We evaluated PARACHUTE on 7 real-world software systems. The results show that PARACHUTE detected 75% (42/56) of the known OCBugs, and reported 13 unknown bugs, 11 of which have been confirmed or fixed by developers until the time of writing.

***Index Terms*—on-the-fly configuration updates, bug detection, metamorphic testing**

I. INTRODUCTION

Software systems introduce an increasing number of configuration options to provide flexibility [1]–[3]. Users can set option values through modifying configuration files. After that, software systems load the files during their startup phases. This procedure, however, is still limited since the users have to restart the software system once changing an option value. The requirement of restarting is impractical for software systems that provide persistent services, e.g., database servers and web servers. To solve this problem, modern software systems support updating configuration options at runtime. For example, MySQL-8.0 has 981 configuration options, of which about 63% support runtime updating [4]. We refer to these systems as runtime configurable systems.

The runtime configurable systems create more flexibility, but on-the-fly configuration updates may affect the system reliability at the same time. Many bug reports [5]–[14] show that, dynamically updating configuration options may lead to unexpected results like software crashes or functional errors, even if the new option values are valid. In this paper, we refer to the bugs caused by on-the-fly configuration updates as on-the-fly configuration bugs, or OCBugs for short.



MySQL Bug #28808 log_queries_not_using_indexes dynamic change is ignored

Description:
The option log_queries_not_using_index can be changed during system running. But it does not change server behavior.

Reproduction:

1. Start server with the configuration
   slow_query_log True
   log_queries_not_using_indexes False.
2. Update the option: set global
   log_queries_not_using_indexes=True;
3. Execute operations:
   CREATE TABLE, INSERT, SELECT.
4. Check slow query log.

Patch:

```
  sql/mysqld.cc
1   static void get_options(int argc){
2 - if (opt_log_queries_not_using_indexes)
3 -    opt_specialflag |= NO_INDEX;

  sql/sql_parse.cc
4   void log_slow_statement(THD *thd){
5   if (thd->enable_slow_log &&
6 -    (opt_specialflag & NO_INDEX)
7 +    opt_log_queries_not_using_indexes
```

Fig. 1: A real-world example of on-the-fly configuration bugs. *The dynamic change of MySQL option does not take effects, since MySQL uses an stale value of the option.*

Figure 1 illustrates a real-world OCBug [5] related to the configuration option `log_queries_not_using_indexes` in MySQL, including the error symptom, the reproduction steps, the root cause, and the fix patch. This option is used to retrieve the queries that do not use indexes for row lookups. Administrators use this option to diagnose performance problems of SQL queries. As shown in Figure 1, the user changed the option value from False to True, but the system did not record related queries. The root cause is that MySQL used the stale option value rather than the updated one. Specially, MySQL used variables `opt_log_queries_not_using_indexes` and `opt_specialflag` to save the option value (Line 2-3), but only updated the former one when receiving the updating command. MySQL missed changing variable `opt_specialflag` before using it (Line 6). The patch is to remove the stale variable `opt_specialflag`, and use `opt_log_queries_not_using_indexes` instead.

There has been much research on addressing problems involving configuration-related bugs [15]–[22]. These works reuse official tests and oracles to detect configuration-related bugs and defects. For example, Ctest [22] reuses official tests and production configurations to detect configuration-induced failures. SPEX [16] injects configuration errors into the system under test, and evaluates software reliability regarding misconfigurations. The official test cases, however, are not designed specifically for on-the-fly configuration updates.

[‡] Teng Wang and Zhouyang Jia are co-first authors.
[*] Shanshan Li is the corresponding author.

Therefore, it is hard for those works to detect OCBugs. Many other works [23]–[30] use the Fuzzing technique to expose bugs. This technique requires test oracles (e.g., crashes or memory sanitizers) to determine if a test input passes or not. The OCBugs, however, may or may not lead to obvious symptoms like crashes or bad memory usage. For example, MySQL-28808 [5] in Figure 1 results in functional errors, and requires specific oracles to detect. The most related work for detecting OCBugs is Staccato [31], which checks if values of configuration-related variables are changed after dynamic configuration updates. If not, Staccato reports a bug. This is a conservative method, and may cause many false negatives, since the variables do not necessarily change to correct values. More details will be discussed at the end of Section II-C.

In this paper, we conducted the first in-depth study on OCBugs based on 75 real-world bugs from 5 popular software systems. We studied the symptoms, root causes, and triggering conditions of OCBugs. The major findings include: 1) More than half (59%) of OCBugs have no easy-to-observe symptoms like crashes or memory leaks, meaning an ideal fuzzing tool can handle up to 41% cases. This result inspires us to design specific oracles for OCBugs. 2) The root causes arise from two aspects according to the lifecycle of configuration-related variables, including variables that initially read configuration values, as well as variables that are control/data dependent on those variables. The result shows that nearly half (45%) OCBugs fail to assign the variables with updated values (referred as *propagation bugs*), while about another half (55%) improperly use the updated variables (referred as *usage bugs*). Propagation bugs can be detected by analyzing dependencies among program variables, while usage bugs can not, due to program-specific usage scenarios. Instead, they can only be detected by examining external behaviors of the program.

Guided by the findings, we propose PARACHUTE, an automated testing framework to detect OCBugs. The key insight of PARACHUTE is that the value of one configuration option, either loaded at the startup phase or updated on the fly, should have the same effects on the target program. This insight serves as a novel oracle for testing OCBugs. Based on the root cause study, the effects in this oracle can be further divided into *internal effects* and *external effects*: a) internal effects are value changes of variables related to the option; b) external effects are behaviors that can be observed outside the program. Internal and external effects are complementary to each other, and used to detect propagation and usage bugs, respectively. Both types of effects are necessary, since propagation bugs do not always lead to observable behaviors (i.e., external effects), while usage bugs are usually not caused by wrong configuration-related variables (i.e., internal effects).

PARACHUTE leverages the idea of metamorphic testing [32] to detect OCBugs using the above two types of effects. In general, PARACHUTE conducts two executions in each test. Given an option value, the first execution loads the value at the startup phase, while the second execution updates the option to that value at runtime. Then, PARACHUTE determines if both the internal and external effects are the same between these two executions. There are two challenges in this process. First, the testing space is huge. To address this challenge, we conduct a comprehensive study towards the triggering conditions of OCBugs in Section II-D, and get three conclusions to guide the design of test-case generation. Second, the effects may not happen immediately after an option is dynamically updated. Runtime configurable systems generally allow existing sessions to adopt the updated values after they complete the currently-executing transactions and commands [33]–[36]. This is a common practice, but PARACHUTE may believe the updated options do not take effect. To avoid false positives, we propose a three-stage metamorphic testing approach by conducting two additional executions in each test (i.e., four executions in total). PARACHUTE compares the results of three pairs of executions to confirm if there is an OCBug.

TABLE I: Studied software systems and their descriptions.

Project	**Description**	**LOC**	**# Option**	**# ROption**[†]
MySQL	SQL database	3714K	981	622
PostgreSQL	SQL database	1869K	344	272
Redis	NoSQL database	181K	149	126
Nginx	Web server	144K	664	664
Squid	Web server	309K	342	340

[†] ROption is short for Runtime Configurable Option.

Parachute also have assumptions on tests, target systems and bugs: a) The test suite should not contain flaky tests or test steps causing indeterminate results. b) The target systems should support dynamically updating of configuration options and their source code should be available. c) Parachute is currently unable to detect OCBugs caused by multiple updates.

We evaluated the effectiveness of PARACHUTE in detecting both known and unknown OCBugs. First, we reproduced 38 known OCBugs from the real-world bugs in our empirical study. To avoid over-fitting, we also reproduced 18 known OCBugs from MariaDB and Httpd, which are not included in the study. The evaluation shows that PARACHUTE can successfully detect 42 bugs (75%), while Staccato [31] detects 15 out of the 56 OCBugs. Moreover, PARACHUTE detected 13 unknown OCBugs from 5 software systems, and 11 of them have already been confirmed or fixed by developers.

To summarize, this paper makes three major contributions.

- We conducted the first in-depth study on real-world OCBugs from 5 open-source software systems to help understand the characteristics and root causes of OCBugs.
- We designed and implemented an automated testing framework, PARACHUTE. It can generate tests for on-the-fly configuration updates by mutating the existing tests and conduct differential analysis to identify OCBugs. All data and source code can be found in the repository:
  https://github.com/wangteng13/Parachute
- We evaluated PARACHUTE on 7 software systems. PARACHUTE detected 75% (42/56) of the known OCBugs, and 13 unknown bugs from 5 software systems. Until the time of writing, 11 of the unknown bugs have been confirmed or fixed by developers.

TABLE II: Symptoms of on-the-fly configuration bugs.

Project	Crash	Hang	Memory Leak	Functional Error	Sum
MySQL	5	0	0	16	21
PostgreSQL	7	0	2	5	14
Redis	6	3	2	16	27
Nginx	4	0	0	3	7
Squid	2	0	0	4	6
Total	24	3	4	44	75

II. Understanding OCBugs

We conduct an empirical study on OCBugs to guide the design of Parachute. In this section, we will first describe the study methodology, then introduce our findings including the symptoms, root causes and triggering conditions of real-world OCBugs.

A. Study Methodology

The study methodology includes the criteria to choose study targets, the method to collect OCBugs, as well as how to validate and analyze the collected data.

Studied Subjects. Table I describes 5 software systems used in our study. We chose these projects because: a) they cover different domains, including database and web server; b) they are widely used and studied by the existing works [15], [16], [37]–[39]; c) they are highly-configurable and expose many runtime configurable options; d) they are open-source and well maintained by the community. These criteria ensure the impacts of studied bugs, and allow us to not only obtain the buggy and fixed code versions, but also collect related details of the bugs, such as root causes and reproduction methods.

Data Collection. We collected real-world OCBugs from tracking systems, mailing lists, and fixed commits of the studied projects. In order to locate OCBugs, we used the following two types of keywords to search for related issues and commits: a) keywords related to description of configuration updating, e.g., *reconfig*, *resize* and *update*; b) keywords related to the command to update options dynamically, e.g., *Config SET* for Redis, *nginx -s reload* for Nginx.

Validation and Analysis. We manually validate each potential OCBugs by inspecting each issue description and related code patches. Each case was inspected by two inspectors. When they diverged, a third inspector was consulted for additional discussions until consensus was reached. They spent two months validating and analyzing the bugs. We filter out the issues where configuration options are not updated on-the-fly during software running. For example, users change a configuration file and restart the software. In the case that we are not sure whether a bug is caused by configuration updating or a special value of the related option, we try to reproduce the bug to validate whether the value itself would cause the bug. Eventually, we collected 75 OCBugs from five selected projects. We further analyzed each OCBug to answer the following three research questions:

- **RQ1**: What are the common symptoms of OCBugs?
- **RQ2**: What are the root causes of OCBugs?
- **RQ3**: What are the triggering conditions of OCBugs?

TABLE III: Root causes of on-the-fly configuration bugs.

Propagation Bugs	**34 (45%)**
Fail to consider loading the updated values	7
Load wrong updated values	16
Miss to propagate to other variables	11
Usage Bugs	**41 (55%)**
Fail to consider handling updated values	8
Improperly handle updated values	27
Bad update timing that causes data race	6

B. Symptoms of OCBugs

We study the symptoms of OCBugs to understand how the bugs affect software systems. The results are shown in Table II, OCBugs could cause the systems to crash, hang, memory leak, and functional error.

Crash and Hang. About one third (27/75=36%) of OCBugs lead to system crashes or hangs. For example, in Redis #4545 [6], when Redis is working on AOF rewrite operations, and users close the AOF mode by dynamically turning off the option `appendonly` at the same time, Redis would infinitely repeat the AOF rewrite operations. The detailed root causes will be described in Section II-C2.

Memory Leak. Other OCBugs (4/75=5%) may cause catastrophic memory leak or resource abuse. For example, in PostgreSQL #16160 [7], option `ssl_ca_file` is used to specify the SSL certificate authority file. When users update an unexsiting path for the option and reload PostgreSQL, the system will suffer from memory leaks or even OOM errors, since PostgreSQL did not free the failed file object during reloading configurations.

Functional Error. Most (44/75=59%) of OCBugs result in functional errors, including unexpected behaviors and wrong results. For example, the option in Figure 1 did not take effect after the update. Another example is that MySQL calculated a wrong increment value in MySQL #65225 [8]. Functional errors have no easy-to-observe characteristics to identify, this is different from system crashes and hangs.

> **Finding 1**: About two-fifths (41%) of OCBugs have obvious symptoms like crashes or memory leaks, while most (59%) of OCBugs result in functional errors that have no easy-to-observe characteristics.

This finding implies that most OCBugs are hard to detect by the existing testing technology like Fuzzing, which typically requires easy-to-observe symptoms as test oracals. This means that existing fuzzing tools can detect up to 41% OCBugs. During the study, we found users frequently compare the effects of an option either loaded at the startup phase or updated on the fly, and report bugs [5], [9], [11] if not consistent. Inspired by these bug reports, we propose a more effective test oracle — *The value of one configuration option, either loaded at the startup phase or updated on the fly, should have the same effects on the target program.*

```
1   void set_config_option(const char *name,...){
2 + /* If value == NULL then we reset some value to
3 +  * its default (removed from configuration file).*/
4 +      else if (source == PGC_S_DEFAULT)
5 +           newval = conf->boot_val;
```

(a) Fail to consider loading the updated values.

```
1   // parse options from config file
2   if (!strcmp(argv, "client-output-buffer-limit")){
3       hard = memtoll (argv[2],NULL);
4
5   // parse options when updating
6   set_special_field("client-output-buffer-limit") {
7 -     hard = strtoll (v[j+1],NULL);
8 +     hard = memtoll (v[j+1],NULL);
```

(b) Load wrong updated values.

```
1   // Initialize variables when system starting
2   static void mainInitialize(void){
3       useragentlog = logfileOpen(useragent_log);
4   }
5
6   // Update variables when system reconfiguring
7   static void mainReconfigureFinish(void *) {
8 +     useragentlog = logfileOpen(useragent_log);
```

(c) Miss to propagate to other variables.

```
1   // Called server.hz times per second
2   int serverCron(...){
3       // Trigger an AOF rewrite if needed
4       if (server.rdb_child_pid == -1 &&
5 +         server.aof_fd != AOF_ON &&
6           server.aof_current_size > server.rewrite_min){
7               rewriteAppendOnlyFileBackground();
```

(d) Fail to consider handling updated values.

```
1   if (got_SIGHUP){
2     if (strcmp((Log_directory, currentLogDir) != 0){
3         currentLogDir = pstrdup(Log_directory);
4 +       //Create new directory if not present
5 +       mkdir(Log_directory);
6         logfile_rotate(Log_directory, Log_filename);
```

(e) Improperly handle updated values.

```
1 -  if (!buf_pool_is_obsolete(withdraw_clock)
2 -      && optimistic_latch_leaves(
3 -      cursor->modify_clock,...) {
4 + if (m_block != NULL) {
5 +      rw_lock_s_lock(latch);
6 +      ...
7 +      rw_lock_s_unlock(latch);
```

(f) Bad update timing that causes data race.

Fig. 2: Examples of root causes. *Each example illustrates one type of OCBugs listed in Table III.*

C. Root Causes of OCBugs

We study the root causes of OCBugs by manually analyzing the patches and comments of each OCBug. The overall finding is that the root causes can be clearly classified into two categories: a) incorrect propagations of configuration-related variables (referred as *propagation bugs*); and b) incorrect usages of the variables (referred as *usage bugs*). Configuration-related variables include the variables that read and store the original value of the involved configuration option, as well as variables that are control/data dependent on the original variables. All these variables should be well defined during the propagation phase before using. This classification is straightforward since every variable should be first defined and then used. The results are shown in Table III.

1) Propagation Bugs: Nearly half (34/75=45%) of OCBugs happened during propagating configuration-related variables, meaning failed to assign the variables with updated values. In specific, the propagation process may have three error scenarios: a) the programs do not load the on-the-fly updated values at all; b) the programs try to load the values, but get wrong values since the parsing methods are incorrect; c) the programs correctly load the values, but errors occur because of missing to propagate the values to other configuration-related variables after configuration updates. The following paragraphs will present OCBug examples for each error scenario.

Fail to consider loading the updated values. The updated options are sometimes not loaded by the system. For example, in PostgreSQL #3589 [10], the user removed one option in *postgres.conf* to use its default value, then reloaded configuration file at runtime. The configuration-related variable, however, retained the old value, rather than its default value. In Figure 2(a), developers fixed the bug by changing options to their default values if options were removed from configuration files.

Load wrong updated values. The programs may get wrong values when parsing dynamically updated options. Taking the bug [9] in Figure 2(b) as an example, Redis uses *memtoll()* to parse the option `client-output-buffer-limit` during system startup, but uses *strtoll()* to parse the same option when reconfiguring. One option value might be parsed into different values when using these two methods, e.g., *memtoll("64mb")* returns 67108864, while *strtoll("64mb")* returns 64. The fix is to use *memtoll()* instead of *strtoll()* when reconfiguring.

Miss to propagate to other variables. The variable that holds the original option value may frequently propagate to other variables through data-flow or control-flow dependencies. Figure 2(c) shows an example [11] caused during data-flow propagation. Squid uses the option `useragent_log` to initialize the logfile. The user tried to disable the option and at runtime, but the stale logfile continued to collect logs. This is because the variable `useragentlog`, propagated by data flow in line 3, is not updated. The patch is to update the variable when receiving an updating command (line 8).

Besides the above case, some options may propagate through control-flow paths. Taking Figure 1 as an example, the variable `opt_log_queries_not_using_indexes` holds the original option value, while the variable `opt_specialflag` is controlled by the option value. When the option is true, `opt_specialflag` would be initialized (line 2-3). MySQL, however, does not update `opt_specialflag` when users turn on the option at runtime. The patch is to remove the stale variable `opt_specialflag`.

2) Usage Bugs: Besides the above cases, about another half (41/75=55%) of OCBugs are about using configuration-related variables. The essence of the bugs lies in wrong program logic that uses the variables, while the variables themselves are correctly updated. Our study shows that these cases can be further classified into three types. First, the programs may not use the dynamically updated variables at all, although the variables have been well-defined by assigning or propagating the latest values. Second, the programs have considered using the updated values, but the values trigger bugs since the handling code is faulty. Third, the handling of new option values itself is correct, but the updating timing may trigger data race. We will present OCBug examples for each type in the following paragraphs.

Fail to consider handling updated values. The programs may miss to handle the situation of configuration updates in special program paths. Taking Redis #4545 [6] as an example, of which the symptoms have been described in Section II-B. As shown in Figure 2(d), Redis evaluates whether AOF rewrite is completed every few milliseconds (line 6), and continues to rewrite if not. The developers miss to handle the situation of `appendonly` updating from 'yes' to 'no', when the AOF rewrite has not been completed. It caused Redis to infinitely repeat the AOF rewrite operations (line 7). The fix is to add handling of the updated value (line 5).

Improperly handle updated values. After on-the-fly configuration updates, the programs try to handle and use the updated options, but the handling code may be faulty. For example, in Figure 2(e), when users dynamically update the option `Log_directory` (line 2), PostgreSQL would force log rotation to ensure writing logfiles in the right place (line 6). PostgreSQL, however, does not create a new directory when the option is updated to a nonexistent path. This will cause functional errors in the PostgreSQL logger [40]. The patch is to create a new directory (line 5).

Bad update timing that causes data race. Users can update the options at anytime during program execution. This mechanism will potentially cause data race. For example, in MySQL, if the option `innodb_buffer_pool_size` is reduced, MySQL would resize the buffer pool, and free unused buffer blocks. In Figure 2(f), MySQL #100630 [13] occured if MySQL shrinked the buffer pool just between *buf_pool_is_obsolete()* and *optimistic_latch_leaves()*. The former is to check if the buffer pool is resized, while the latter is to access the buffer. This bug causes buffer overflow, and the patch is to add locks for buffer blocks (line 5-7).

> **Finding 2**: Nearly half (45%) of OCBugs happened during propagating configuration-related variables, while the other half (55%) of OCBugs are about using those variables.

This finding implies that OCBugs can be basically divided into two main types. The first type can be detected by analyzing the states of configuration-related variables inside the target program, since there exist control or data dependencies among the variables. The second type, however, is hard to be detected by program analysis. Instead, they can only be detected by examining external behaviors of the program. Taking the OCBug in Figure 2(d) as an example, it is hard to recognize that the code snippet misses the check in line 5. In this regard, we extend the test oracle described at the end of Section II-B — *The effects should be divided into internal and external effects. Internal effects are value changes of variables related to configuration options, while external effects are behaviors that can be observed outside the program.*

Internal and external effects serve as two complementary oracles to detect propagation and usage bugs, respectively. Propagation bugs do not always lead to observable behaviors (external effects), but program variables may have inconsistent states (internal effects). Usage bugs are usually not caused by wrong configuration-related variables (internal effects), but frequently lead to observable behaviors (external effects). Thus, both types of effects are necessary in detecting OCBugs. Staccato [31] first collects configuration-related program variables, then checks if their values are changed after dynamic configuration updates. If not, Staccato reports a bug. This approach will miss the cases having external effects, since the configuration-related variables have changed as expected. It may also miss some cases having internal effects, which load wrong updated values. In both cases, Staccato cannot report bugs. As a result, Staccato can detect up to (7+11)/75=24% OCBugs in ideal.

D. Triggering Conditions of OCBugs

In order to guide and facilitate automated test-case generation for testing OCBugs, we conduct a comprehensive study towards the triggering conditions of OCBugs in this section. In specific, we break RQ3 into following three sub-questions:

- **RQ3.1**: What option values are able to trigger OCBugs?
- **RQ3.2**: How many updating times can trigger OCBugs?
- **RQ3.3**: What dependencies are required by the OCBugs?

1) Option values: An option value may be either valid or invalid, where an invalid value means breaking the constrains of the option. We first investigate if option values triggering OCBugs should be valid or not. To achieve this, we manually collect option constraints from documents and source code. The results show that both valid values (66/75=88%) and invalid ones (9/75=12%) can trigger OCBugs. This inspires us to generate both valid and invalid option values when testing OCBugs.

For invalid values, we only need to generate one value that breaks the option constraints. For valid values, however, the generating policies may be different according to the option types. It is easy to generate values for Boolean or enumerable options, since we can simply enumerate all possible values. As for numeric options, we need to study the characteristics of the specific values that trigger OCBugs. The results show that, among 33 OCBugs that are related to numeric options with valid values, most (22/33=67%) of them are insensitive to option values. It means an arbitrary different numeric value is enough to trigger a bug. Meanwhile, one third (11/33=33%) of OCBugs can be triggered by changing the values drastically, e.g., exponentially increasing or decreasing the values. For example, MySQL #100630 [13] is triggered by changing the buffer pool size from 2G to 128M. In this regard, when testing numeric options, we can always exponentially increase or decrease their values.

> **Finding 3.1**: Both valid and invalid option values should be taken into consideration when testing OCBugs. The option values should be drastically changed when testing numeric options.

2) Updating times: An OCBug may require multiple updating operations to be triggered. It will be exponentially explosive to test all combinations of multiple updating operations. To help this situation, we study the times of updating operations required to trigger OCBugs. To achieve this, we checked the bug description and commit messages of all OCBugs. The results show that the vast majority (71/75=95%) of OCBugs require one time of on-the-fly update on one option to trigger the bugs. In very limited cases (4/75=5%), multiple updating operations are needed, i.e., updating one option multiple times or even updating multiple options. For example, in Redis #8030 [41], the bug is triggered by updating the option `appendonly` from ‘yes’ to ‘no’, then back to ‘yes’.

> **Finding 3.2**: Most (95%) of OCBugs can be triggered by dynamically updating one option once. It means performing one updating operation in one test execution is enough for exposing the vast majority of OCBugs.

3) Option dependencies: One configuration option usually depends on other options to take effects, no matter the option is loaded at the startup phase or updated on the fly. The dependency problem may also lead to an exponential explosion similar to the above paragraph. Therefore, we study option dependencies required to trigger OCBugs. Please note that these dependencies are different from the cases of updating multiple options above. Here the dependencies mean options that should be set during the startup phase. To achieve this, we record the options set by users during the startup phase, and replace their values with default ones. If an OCBug can be no longer triggered, it means there is a dependency.

The results show that in most (55/75=73%) OCBugs, the target options do not depend on any other option, while more than one fourth (20/75=27%) of OCBugs have dependencies. In this regard, we further investigate the source code and documentation related to the 20 OCBugs. Among these, the updated options of 20% (15/75) OCBugs are data/control dependent on other options in source code. As shown in Figure 1, triggering MySQL #28808 [5] needs to turn on `slow_query_log` to enable the updated option `opt_log_queries_not_using_indexes`. Besides, the dependencies of 7% (5/75) OCBugs are hard to be obtained from source code. For example, triggering MySQL #5394 [42] first needs to turn on `query_cache_type`, then updates `max_sort_length`. The buggy code snippet, however, does not consider using the updated option. As a result, updating `max_sort_length` does not take effect. In this case, it is hard to obtain the dependency between `query_cache_type` and `max_sort_length` from the source code, which does not appear at all.

> **Finding 3.3**: In most (73%) of OCBugs, the target options do not depend on any other option. Dependencies of 20% OCBugs can be obtained by program analysis. However, the other 7% may need combinatorial interaction testing on options.

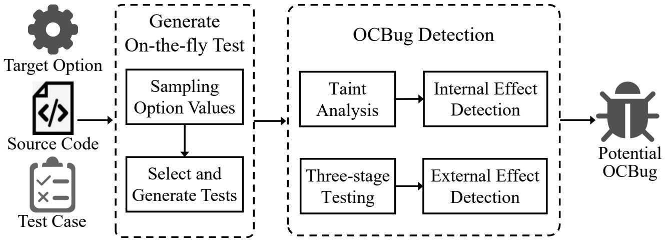


Fig. 3: Overview of PARACHUTE

III. DETECTING OCBUGS

In this section, we describe the design of PARACHUTE, an automated testing framework in detecting OCBugs. We first introduce the overview of PARACHUTE, as well as its technical challenges. After that, we introduce two main components of PARACHUTE, i.e., test-case generations and OCBug detections. Suggested by Finding 2, the detection component is supposed to handle two situations: test cases that cause either internal effects or external effects.

A. Overview of the OCBug Testing Framework

Figure 3 shows the overview of PARACHUTE, which requires three inputs: source code of Software Under Test (SUT), target configuration options of SUT, and the official test suite of SUT. The PARACHUTE framework contains two major tasks: generating on-the-fly tests and detecting OCBugs.

Generating on-the-fly tests. PARACHUTE first generates test cases of on-the-fly configuration updating for the target options. To achieve this, PARACHUTE leverages and mutates the existing test suite. The main challenge of this task is the huge testing space. For each target option, PARACHUTE needs to mutate all test cases of the test suite. For each test case, PARACHUTE further needs to generate a large number of mutations, since one option may have different values, updating times, dependencies and so on. To address this challenge, we conduct a comprehensive study towards the triggering conditions of OCBugs in Section II-D, and get three conclusions to guide the design of test-case generation.

Detecting OCBugs. PARACHUTE then leverages metamorphic testing to detect OCBugs. In specific, PARACHUTE tests the target program twice, using configuration options loaded since system startup or updated on-the-fly, separately. After that, PARACHUTE detects OCBugs based on the following oracles according to Finding 2:

- **Oracle I** (Internal Effects): The values of program variables related to configuration options should be the same, no matter the options are loaded since system startup or updated on-the-fly.
- **Oracle II** (External Effects): The outputs of the system under test should be the same, no matter configuration options of the system are loaded since system startup or updated on-the-fly.

For Oracle I, the main challenge is to determine the involved variables. The challenge of Oracle II is that the effects may not happen immediately after an option is dynamically updated.

Instead, programs usually finish the current workload using old option values, and apply new values later. In this case, the programs have no OCBugs, but PARACHUTE may believe the updated options do not take effect and report false positives. To solve this problem, we propose a three-stage metamorphic testing approach.

B. Generating On-the-fly Tests

Mature software projects usually have official test suite, which is rarely designed for the situation of on-the-fly option updating. Therefore, PARACHUTE mutates the existing test cases to trigger OCBugs. This process, however, is non-trivial. First, a project may have thousands of test cases and hundreds of options. It is time-consuming to perform all test cases for each option. PARACHUTE should filter out the test cases that are not related to the target option. Second, in each selected test case, PARACHUTE will insert a command to update an option, which may have a large number of possible values. PARACHUTE has to determine the values that should be tested. Third, PARACHUTE needs to generate new test cases based on the selected test cases and values. Each new test case contains two executions, since PARACHUTE uses metamorphic testing.

Selecting existing test cases. As running all the test cases for all options may be time-consuming, we need to pre-select a subset of test cases for each target option, and filter out the most majority of cases that are not related to the option. To achieve this, PARACHUTE first integrates *ConfMapper* [43] to find the original variables used to load options, then instruments the usage of those variables by using Clang [44]. After that, PARACHUTE runs all test cases for one time, and obtains the option set that can be triggered by each test case. Finally, PARACHUTE filters out the test cases that cannot trigger the target option.

Determining option values. According to Finding 3.1 in Section II-D, we need to test both valid and invalid values for a target option. To achieve this, PARACHUTE first collects the constraints of the option by applying the existing tools, SPEX [16] and CeitInspector [15]. SPEX uses pattern-based program analysis method to obtain constraints and dependencies of options from source code. CeitInspector defines syntax and semantic constraints for different types of options. On the one hand, PARACHUTE uses the constraint violation rules defined in CeitInspector [15] to generate invalid values of the target option. In specific, for Boolean, enumerable or numeric options, PARACHUTE generates invalid values beyond the value set or valid range (e.g. MIN-1, MAX+1). For options of other types,PARACHUTE generates invalid values by violating their syntax (e.g., an invalid ip address).

On the other hand, PARACHUTE samples values satisfying the configuration constraints. For each Boolean and enumerable option, PARACHUTE chooses all its possible values. As for numeric options, it is hard to test all values. Guided by Finding 3.1, PARACHUTE samples values changed exponentially for a given sampling number. For example, the valid range of option `binlog_cache_size` is [2^{12}, 2^{32}]. PARACHUTE will sample {2^{12}, 2^{16}, 2^{20}, 2^{24}, 2^{28}, 2^{32}}, if users want to sample six values. For options of other types, PARACHUTE generates valid values by satisfying their syntax (e.g., a valid ip address).

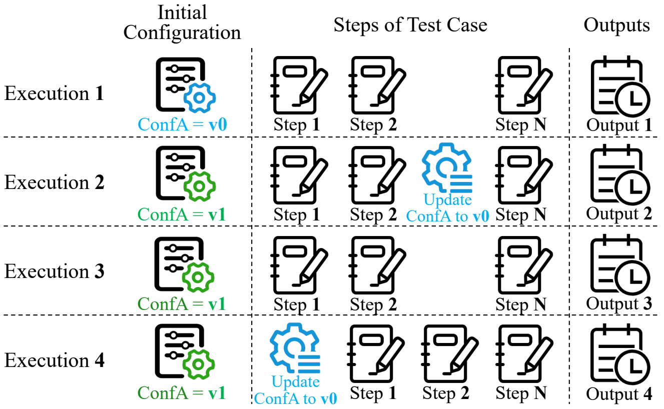


Fig. 4: Examples of metamorphic test executions

Generating new test cases. This process involves two tasks. First, for each pair of the selected values, PARACHUTE needs to generate two executions as one new test case. As shown in the first two executions of Figure 4, Execution 1 assigns the option *ConfA* to *v0* at startup, while Execution 2 uses *v1* at startup but updates the value back to *v0* at a random place during runtime. Please note that, PARACHUTE only needs to insert one updating command according to Finding 3.2. After the update, the program is supposed to have the same behaviors in two executions since *ConfA* has the same value *v0*. PARACHUTE regards this constraint as a metamorphic relation to detect OCBugs.

Second, the updated option *ConfA* may depend on other options to become effective. According to Finding 3.3, besides the 73% OCBugs that do not depend on any other option, dependencies of 20% OCBugs can be obtained by program analysis. In this regard, PARACHUTE integrates *SPEX* [16], an existing tool that can obtain option dependencies automatically. PARACHUTE would satisfy control and value dependencies for the target option before running each new test case. While for the other 7% OCBugs that can only be detected by combination testing, PARACHUTE also provides an exhaustive testing mode with a given time budget provided by users.

C. Detecting OCBugs

With the new test cases available, PARACHUTE can detect OCBugs by using two oracles as mentioned in Finding 2, i.e., comparing both internal and external effects between two executions of each new test case.

1) Detecting OCBugs using Internal Effects: The internal effects are used to detect propagation bugs. When updating an option, the internal effects are value changes of its corresponding variables, including the variable that reads and stores the original value of the option, as well as variables that are control/data dependent on the original variable. Therefore, PARACHUTE needs to collect the option-related variables. To achieve this, PARACHUTE firstly conducts taint analysis to find

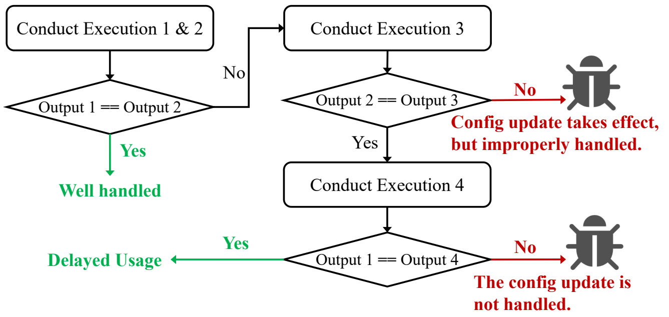


Fig. 5: Flowchart for detecting OCBugs using externel effects

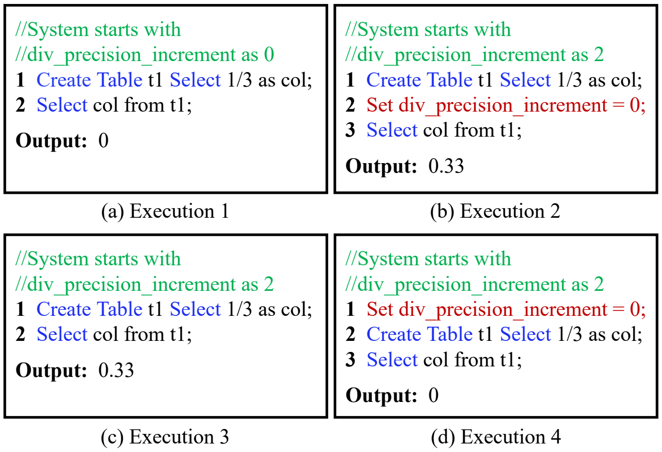
the configuration-related variables, then instruments the source program.

The taint analysis starts from the variable, which first reads and stores the option value. PARACHUTE uses *ConfMapper* [43] to find the original variable of each option, then propagates the taints along data-flow paths. The data-flow analysis is inter-procedural, field-sensitive, and supports pointer analysis. Besides, PARACHUTE also supports control-flow taint analysis. For example, in line 2-3 of Figure 1, the analysis will taint `opt_specialflag`, which is control depended on the option variable `opt_log_queries_not_using_indexes`, and changed to different values in different branches.

Then, PARACHUTE instruments the source program to record the values of tainted variables. One option may taint many program variables, and lead to significant overhead after instrumentation. To remedy this situation, we investigated the propagation bugs again, and found the overwhelming bugs (32/34=94%) were triggered by global variables storing incorrect or stale values. The other two cases will cause crashes when loading new values. The crash cases have obvious symptoms, and do not need to check internal effects. Therefore, PARACHUTE only records global configuration-related variables. For example, both `opt_specialflag` and `opt_log_queries_not_using_indexes` in Figure 1 are global variables. The taint analysis is implemented using LLVM [45], while the instrumentation is based on Clang [44].

2) Detecting OCBugs using External Effects: The external effects are used to detect usage bugs. When updating an option, its external effects are program behaviors that can be observed outside the program. PARACHUTE records outputs, crashes, and hangs as external effects during testing. The challenge here is that the effects of option updating may not happen immediately. Runtime configurable systems generally allow existing sessions to adopt the updated values after they complete the currently-executing transactions and commands [33]–[36]. PARACHUTE needs to avoid false positives caused by the delayed usage of new values.

To achieve this, we propose a three-stage metamorphic testing approach to address this challenge. PARACHUTE conducts two additional executions in each test, i.e., totally four executions in each test as shown in Figure 4. To confirm if there is an OCBug, PARACHUTE compares the results of three pairs of executions, including Execution 1 and 2, Execution 2 and 3, Execution 1 and 4. The workflow is illustrated in Figure 5.



```
//System starts with
//div_precision_increment as 0
Create Table t1 Select 1/3 as col;
Select col from t1;

Output: 0
```

(a) Execution 1

```
//System starts with
//div_precision_increment as 2
Create Table t1 Select 1/3 as col;
Set div_precision_increment = 0;
Select col from t1;

Output: 0.33
```

(b) Execution 2

```
//System starts with
//div_precision_increment as 2
Create Table t1 Select 1/3 as col;
Select col from t1;

Output: 0.33
```

(c) Execution 3

```
//System starts with
//div_precision_increment as 2
Set div_precision_increment = 0;
Create Table t1 Select 1/3 as col;
Select col from t1;

Output: 0
```

(d) Execution 4

Fig. 6: A MySQL example of using three-stage testing

First Stage: PARACHUTE compares external effects between the first two executions. If the effects are the same, it means the program successfully handles the update. If the effects are different, there are three cases: a) the program is using old values for the current transaction, while new values do not take effects so far; b) the program improperly handles new values; c) the program does not handle new values at all. The first case is a common practice, while the last two cases are OCBugs.

Second Stage: PARACHUTE adds an execution in the test case, as shown in Figure 4 Execution 3, which deletes the updating command. PARACHUTE compares the effects between Execution 2 and 3. If the effects are different, it indicates the new value has already taken effects, meaning the program improperly handles the new value. Thus, PARACHUTE reports an OCBug. If the effects are the same, there still are two possibilities: delayed usage or no handling at all.

Third Stage: PARACHUTE adds another execution in the test case, as shown in Figure 4 Execution 4, which places the updating command at the beginning. PARACHUTE compares the effects between Execution 1 and 4. If the effects are the same, it indicates the program successfully handles the updating, since there is no working transaction before the updating command. Otherwise, it means the program does not handle the new value at all. Thus, PARACHUTE reports an OCBug.

Figure 6 shows a real-world example of using the three-stage metamorphic testing approach in MySQL. The option `div_precision_increment` indicates the number of decimal places for answers of division operations. PARACHUTE 1) runs Execution 1 and 2, and finds the outputs are different; 2) runs Execution 2 and 3, and finds the outputs are the same; 3) runs Execution 1 and 4, and finds the outputs are also the same. Thus, PARACHUTE knows the case is caused by delayed usage of the new option value, and does not report any bug.

TABLE IV: The effectiveness of detecting known OCBugs.

OCBug Type	Reproduced OCBugs	Detected by PARACHUTE	Detected by Staccato
Propagation Bugs	**33**	**31**	**15**
Fail to consider loading	8	8	8
Load wrong updated values	15	15	0
Miss to propagate	10	8	7
Usage Bugs	**23**	**11**	**0**
Fail to consider handling	7	4	0
Improperly handle	16	7	0
Total	**56**	**42**	**15**

TABLE V: New OCBugs detected by PARACHUTE.

Bug ID	Version(s)	Status	Type[†]	O.I	O.II[‡]	Staccato
MySQL #105933	v5.7-latest	Confirmed	Type-2		✓	
MySQL #105957	v5.7-latest	Confirmed	Type-2		✓	
MySQL #105964	v5.7-latest	Confirmed	Type-2		✓	
MySQL #105978	v5.7-latest	Confirmed	Type-1	✓	✓	✓
MySQL #106675	v5.7	Confirmed	Type-2		✓	
MySQL #106676	v5.7	Confirmed	Type-2		✓	
MySQL #106684	v5.7	Confirmed	Type-2		✓	
Redis #10119	v6.2-v7.0	Fixed	Type-1	✓		
Squid #5224	v5.0-latest	Pending	Type-1	✓		✓
Squid #5225	v5.0-latest	Pending	Type-1	✓		✓
Postgres #17538	v14.2-latest	Confirmed	Type-1	✓	✓	✓
MariaDB #29076	v10.3-latest	Fixing	Type-2		✓	
MariaDB #29077	v10.3-latest	Fixing	Type-2		✓	

[†] Type-1 is short for propagation bugs; Type-2 is short for usage bugs.
[‡] O.I & O.II is short for Oracle I and Oracle II.

IV. EVALUATION

To evaluate PARACHUTE, we consider the following three research questions:

- **RQ1:** How effective is PARACHUTE in detecting known OCBugs? This question examines the recall of PARACHUTE by calculating the percentage of bugs that can be detected among all known bugs.
- **RQ2:** How effective is PARACHUTE in detecting unknown OCBugs? This question evaluates the precision of PARACHUTE by calculating the percentage of true positives among all reported bugs.
- **RQ3:** Can PARACHUTE outperform the state-of-the-art tool for detecting configuration update bugs? This question compares PARACHUTE with Staccato, the most related work for detecting OCBugs.

A. Effectiveness of Detecting Known OCBugs

We evaluate the effectiveness of PARACHUTE in detecting known OCBugs. As OCBugs with obvious symptoms (e.g., crash, hand, memory leak) can be detected by existing techniques like fuzzing. We mainly evaluate PARACHUTE in detecting OCBugs leading to functional errors. We successfully reproduced 38 out of the 44 functional-error OCBugs in Table II. To avoid over-fitting, we repeated the bug collection methods on MariaDB and Httpd, and reproduced 18 bugs that were not included in the empirical study. Totally, we evaluate PARACHUTE on 56 OCBugs from 7 systems. For each bug, PARACHUTE tests the buggy software version with its official test suite for 20 hours. Each target system runs on a virtual machine hosted on the cloud.

PARACHUTE successfully detected 75% (42/56) of the existing bugs. The results are shown in Table IV. PARACHUTE can detect most (31/33=94%) of propagation bugs, and nearly half (11/23=48%) of usage bugs. PARACHUTE failed to detect 14 bugs due to three reasons. First, the triggering conditions for the bugs are not satisfied (6 cases). As the testing space is huge, PARACHUTE uses heuristic strategies to generate test cases, and thus inevitably misses some corner cases: a) The option requires a special value. For example, Nginx #796 [46] requires updating the option to different paths of the same file, e.g., from absolute path to relative path, or vice versa. b) The option has to be updated multiple times. c) It is hard to obtain the dependency of the option from source code. Second, the taint analysis fails to get configuration-related variables due to complicated pointer and alias analysis (2 cases). Third, some bugs require special environments and operations (6 cases). For example, MariaDB #23988 [47] occurs in a cluster of three nodes. But the official test suite does not satisfy the requirement.

Answer to RQ1: This result indicates PARACHUTE can effectively detect the existing OCBugs with the recall of 75% (42/56).

B. Effectiveness of Detecting Unknown OCBugs

We also apply PARACHUTE on the latest version of SUT to evaluate whether PARACHUTE can detect unknown OCBugs. We evaluate PARACHUTE on the 7 software systems, including MariaDB, Httpd and the systems listed in Table I. For each system, we randomly test 100 options, and conduct PARACHUTE to test each option for 20 hours on three virtual machines in parallel. This experiment takes 21 virtual machines for a month.

PARACHUTE reported 13 true positives and 2 false positives according to our manual analysis. We report the 13 OCBugs to developers, and 11 of the bugs have been confirmed or fixed by developers, as shown in Table V. The 13 OCBugs come from 5 systems, including MySQL, Redis, Squid, PostgreSQL and MariaDB. Among these new bugs, there are 5 propagation bugs and 8 usage bugs. Besides, PARACHUTE also reported 2 false positives, which were caused by indeterminate results of some test cases. For example, in MySQL, the operation *Explain Select* is used to predict the statement execution plan, and returns the number of rows MySQL plans to examine for the query [48]. The number is an estimate and not always exact, which misled the analysis of PARACHUTE.

We found the unknown OCBugs could cause the systems functional errors or performance degradation. For example, in MariaDB #29076, updating the option `time_zone` between two same *Select* operations, would make *Query Cache* incorrectly identify them as different queries. The bug is caused by improperly handling the updated value. It would also cause serious performance degradation in extreme cases; MariaDB repetitively performs time-consuming queries and stores redundant results, rather than returning the result from the cache directly. The bug quickly caught the attention of developers after we reported it.

Answer to RQ2: This result indicates PARACHUTE can effectively detect unknown OCBugs in popular, real-world software systems with the precision of 87% (13/15).

C. Comparison with the State-of-the-art Tool

We compare PARACHUTE with Staccato [31], the state-of-the-art tool for detecting configuration update bugs. Staccato first collects configuration-related program variables, then checks if their values are changed after dynamic configuration updates. We evaluate the effectiveness of Staccato in detecting the same known OCBugs in IV-A, and unknown bugs found by PARACHUTE. Staccato is designed for Java and PARACHUTE is for C/C++, so we evaluate the theoretical upper bound of Staccato in detecting these bugs. As Staccato did not publish reproduction steps for its detected bugs, it is hard to evaluate PARACHUTE on the Java programs evaluated by Staccato.

The evaluation shows that Staccato can detect 27% (15/56) of the reproduced OCBugs, as shown in Table IV. Staccato successfully detected 45% (15/33) of propagation bugs. The reason is that Staccato can only detect whether the option value is updated, but it is hard for Staccato to detect the correctness of the updated values. As a result, Staccato missed all of the bugs caused by *Loading wrong updated values* (as mentioned in Section II-C1). Moreover, Staccato can detect most (7/10) of OCBugs arising from *Missing to propagate*, but failed to detect 3 OCBugs caused by control-flow propagation (e.g., MySQL-28808 in Figure 1). As for usage bugs, the configuration-related variables are correctly updated. Staccato can not detect this type of OCBugs. Moreover, Staccato can detect 4 of the 13 unknown bugs reported by PARACHUTE, as shown in Table V. All the four bugs are caused by *Missing to propagate*. Among both known and unknown bugs, PARACHUTE can detect 2.9× OCBugs compared with Staccato, i.e., (42+13)/(15+4).

Answer to RQ3: This result indicates PARACHUTE can detect more OCBugs (2.9×) compared with the state-of-the-art tool.

V. DISCUSSION

Quality of test suite. PARACHUTE leverages and mutates existing test suite, instead of generating completely new test cases. If the existing tests do not provide proper test environments and operations to trigger the bugs, PARACHUTE will lose the opportunity to detect and identify them. Many *usage bugs* usually require complicated environments and test steps. To this end, PARACHUTE provides interfaces to accept user-provided test suite, to specifically test some functions and scenarios. On the other hand, fuzzing [23]–[30], [49], [50] is popular automated testing technique that generates diverse tests and improves the code coverage. Our future work will lie in combining PARACHUTE with fuzzing techniques to generate high-quality test cases to detect OCBugs.

Generating on-the-fly tests. Since the configuration space is huge, PARACHUTE uses heuristic strategies to sample option values. However, some bugs require special option values. In this regard, PARACHUTE also provides interfaces to accept user-provided options and their dependencies, as users usually have knowledge of the constraints of the options. Besides, PARACHUTE mutates test cases by inserting one updating command at a random place of the existing test, considering the overhead of testing. The placement may lead to both false positives and false negatives. Parachute applies a three-stage comparison approach to avoid the false positives. However, the randomly placed update events may not trigger some bugs, thus lead to false negatives. Our future work is to investigate more efficient methods to mutate tests by integrating program analysis, including multiple updates and intelligent places.

Representativeness of studied software. The findings of our research may only apply to database and web server systems. We attempt to study a wide variety of popular open-source configurable systems. And we selected 5 widely-used systems, which cover different domains, including database and web server. Another criteria to select study targets is that the systems expose many runtime configurable options. It makes us abandon some popular software (e.g., HDFS has only 16 (out of 583) runtime configurable options [51]. Most options can only be updated after restarting the system). In addition, software from other domains could have different characteristics. We will explore the characteristics of OCBugs from more categories of software in the future work.

Execution order of three-stage testing. The execution order in Figure 5 is designed for reducing false positives caused by delay usage of the option, and diagnosing the root causes of the bugs. The order of execution can be adjusted, but these three stages are necessary. For example, comparing Execution 1&4 first can find some bugs early. However, updating options right after execution starts cannot trigger all OCBugs. Parachute still needs Execution 2 and 3 to detect these bugs (e.g., the bug in Figure 2d). In addition, comparing Execution 1&4 cannot reveal the root causes (i.e., improperly handled or not handled) of the early-detected bugs.

VI. RELATED WORK

Detecting configuration-related bugs. Many OCBugs are non-crashed, leading to various forms of functional errors, which requires specific oracles to detect. Popular automated testing techniques, such as Fuzzing [23]–[30], [49], [50], could not effectively detect such functional bugs due to the lack of test oracles. However, fuzzing method could generate various tests to help PARACHUTE detect OCBugs.

Some works [22], [31], [37], [38], [52]–[55] focus on detecting configuration-related functional defects or performance defects in source code. Ctest [22], [52] connects production system configurations to software tests to detect configuration-induced failures. Ctest simply reuses official tests and oracles, which cannot detect OCBugs effectively. CP-Detector [38] suggests performance properties for configuration options to detect Configuration-handling Performance Bugs. The most related work is Staccato [31], which is designed to find bugs for dynamic configuration updates. Staccato identifies a bug if programs use the stale value of configuration-related variables after dynamic configuration updates. Our study shows that Staccato misses all of usage bugs, and the cases which load wrong updated values. In this paper, based on our in-depth research, we conduct metamorphic testing and check both the internal and external effects of configuration updates. Thus, Parachute can detect all types of OCBugs.

Configuration error injection testing. Some works [15]–[21] focus on evaluating software reliability and diagnosability regarding configuration errors. These works inject configuration errors into the system under test (SUT), and then evaluate the SUT reactions. ConfErr [19], ConfInject [21], ConfTest [20], and ConfDiagDetector [17] use predefined mutation rules to generate types of configuration errors. SPEX [16], ConfVD [18] and CeitInspector [15] generate configuration errors by violating the specifications of options. However, all these works directly leverage the official test suite, which are not designed specifically for on-the-fly configuration updates. Therefore, these works are hard to detect OCBugs.

Metamorphic testing. Some works [56]–[61] use metamorphic testing [32] to detect logical bugs. Adamsen et al. [56] use specific metamorphic relations to enhance existing test suites for Android. SetDroid [57] uses setting-wise metamorphic fuzzing for finding system setting defects in Android applications. The work [58] uses metamorphic model-based testing with equivalence of queries to test DAT systems. Our work also leverages the idea of metamorphic testing to detect OCBugs in runtime configurable systems. Based on the root cause study, we proposed two oracles to identify OCBugs.

VII. Conclusion

Many modern software systems support updating configuration options on the fly without restarting the system. This mechanism can improve the system flexibility and provide persistent services at the same time. However, on-the-fly updating configuration may also introduce OCBugs. In this paper, we studied the symptoms, root causes, and triggering conditions of OCBugs, and proposed Parachute to detect OCBugs guided on the study. Parachute leverages the idea of metamorphic testing to detect OCBugs using two types of effects as oracles. The evaluation results show that Parachute can effectively detect both known and unknown bugs, and outperforms the state-of-the-art tool.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments. We also thank Professor Tingting Yu for helpful suggestions, and Yuanliang Zhang, Xiangbing Huang for paper proofreading and experiments. This research was funded by NSFC No. 61872373, No. 62272473, No. 62202474 and No. U19A2067.

References

[1] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, You Have given Me Too Many Knobs!: Understanding and Dealing with over-Designed Configuration in System Software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, pp. 307–319.

[2] Y. Zhang, H. He, O. Legunsen, S. Li, W. Dong, and T. Xu, "An evolutionary study of configuration design and implementation in cloud systems," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 188–200.

[3] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 159–172.

[4] "MySQL 8.0 Reference Manual. Server Option, System Variable, and Status Variable Reference." https://dev.mysql.com/doc/refman/8.0/en/server-option-variable-reference.html, 2022.

[5] "MySQL Bug #28808. log_queries_not_using_indexes variable dynamic change is ignored." https://bugs.mysql.com/bug.php?id=28808, 2007.

[6] "Redis Bug #4545. dead loop AOF rewrite when config set appendonly no." https://github.com/redis/redis/issues/4545, 2017.

[7] "PostgreSQL Bug #16160. Minor memory leak in case of starting postgres server with SSL encryption." https://www.postgresql.org/message-id/16160-18367e56e9a28264%40postgresql.org, 2019.

[8] "MySQL Bug #65225. InnoDB miscalculates auto-increment after changing auto_increment_increment." https://bugs.mysql.com/bug.php?id=65225, 2012.

[9] "Redis Bug #4904. Use memtoll() in CONFIG SET client-output-buffer-limit." https://github.com/redis/redis/pull/4904/, 2018.

[10] "PostgreSQL Bug #3589. postgresql reload doesn't reflect log_statement." https://www.postgresql.org/message-id/200708300302.l7U32sP9005096%40wwwmaster.postgresql.org, 2007.

[11] "Squid Bug #579. useragent log disable." https://bugs.squid-cache.org/show_bug.cgi?id=579, 2005.

[12] "Nginx Bug #945. when setting master_process off, nginx segmentation fault when sent mutiple HUP singals." https://trac.nginx.org/nginx/ticket/945, 2018.

[13] "MySQL Bug #100630. buf_pool_is_obsolete is not thread safe." https://bugs.mysql.com/bug.php?id=100630, 2020.

[14] "Redis Bug #5025. Fix config_set_numerical_field() integer overflow." https://github.com/redis/redis/pull/5020, 2020.

[15] W. Li, Z. Jia, S. Li, Y. Zhang, T. Wang, E. Xu, J. Wang, and X. Liao, "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*, 2021, pp. 478–490.

[16] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 244–259.

[17] S. Zhang and M. D. Ernst, "Proactive detection of inadequate diagnostic messages for software configuration errors," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 12–23.

[18] S. Li, W. Li, X. Liao, S. Peng, S. Zhou, Z. Jia, and T. Wang, "Confvd: System reactions analysis and evaluation through misconfiguration injection," *IEEE Transactions on Reliability*, vol. 67, no. 4, pp. 1393–1405, 2018.

[19] L. Keller, P. Upadhyaya, and G. Candea, "Conferr: A tool for assessing resilience to human configuration errors," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008, pp. 157–166.

[20] W. Li, S. Li, X. Liao, X. Xu, S. Zhou, and Z. Jia, "Conftest: Generating comprehensive misconfiguration for system reaction ability evaluation," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, 2017, pp. 88–97.

[21] F. A. Arshad, R. J. Krause, and S. Bagchi, "Characterizing configuration problems in java ee application servers: An empirical study with glassfish and jboss," in *IEEE International Symposium on Software Reliability Engineering*, 2014.

[22] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu, "Testing configuration changes in context to prevent production failures," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 735–751.

[23] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui, "Healer: Relation learning guided kernel fuzzing," 2021.

[24] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu, "Squirrel: Testing database management systems with language validity and coverage feedback," 2020.

[25] M. Wang, J. Liang, C. Zhou, Y. Jiang, R. Wang, C. Sun, and J. Sun, "RIFF: reduced instruction footprint for coverage-guided fuzzing," in *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, 2021, pp. 147–159.

[26] M. Wang, Z. Wu, X. Xu, J. Liang, C. Zhou, H. Zhang, and Y. Jiang, "Industry practice of coverage-guided enterprise-level dbms fuzzing," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 328–337.

[27] C. Zhou, M. Wang, J. Liang, Z. Liu, and Y. Jiang, "Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 858–870.

[28] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 754–768.

[29] A. Fioraldi, D. C. D'Elia, and D. Balzarotti, "The use of likely invariants as feedback for fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2829–2846.

[30] "American Fuzzy Lop." https://lcamtuf.coredump.cx/afl/, 2022.

[31] J. Toman and D. Grossman, "Staccato: A bug finder for dynamic configuration updates," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[32] S. C. C. Tsong Y. Chen and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," in *Technical Report. HKUST-CS9801, HongKong University of Science and Technology.*, 1998.

[33] "PostgreSQL. Setting Parameters." https://www.postgresql.org/docs/14/config-setting.html, 2022.

[34] "Nginx. Changing Configuration." http://nginx.org/en/docs/control.html, 2022.

[35] "MySQL. Dynamic System Variables." https://dev.mysql.com/doc/refman/8.0/en/dynamic-system-variables.html, 2022.

[36] "Redis. CONFIG SET parameter value." https://redis.io/commands/config-set/, 2022.

[37] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early detection of configuration errors to reduce failure damage," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 619–634.

[38] H. He, Z. Jia, S. Li, E. Xu, T. Yu, Y. Yu, W. Ji, and X. Liao, "Cp-detector: using configuration-related performance properties to expose performance bugs," in *ASE '20: 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020.

[39] S. Zhou, X. Liu, S. Li, Z. Jia, Y. Zhang, T. Wang, W. Li, and X. Liao, "Confinlog: Leveraging software logs to infer configuration constraints," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 94–105.

[40] "PostgreSQL Bug. Log_collector doesn't respond to reloads." https://www.postgresql.org/message-id/4F99E37E.30904

[41] "Redis Bug #8030. AOF: recover from last write error after turn on appendonly again." https://github.com/redis/redis/pull/8030, 2020.

[42] "MySQL Bug #5394. Max_sort_length does not invalidate queries in the query cache." https://bugs.mysql.com/bug.php?id=5394, 2004.

[43] S. Zhou, X. Liu, S. Li, W. Dong, and X. Yun, "Confmapper: Automated variable finding for configuration items in source code," in *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2016.

[44] "Clang: a C language family frontend for LLVM." https://clang.llvm.org/, 2022.

[45] "LLVM Programmers Manual." https://llvm.org/docs/ProgrammersManual.html, 2022.

[46] "Nginx Bug #796. nginx.pid is removed during reload if pid path is changed in nginx.conf but points to the same file through a symlink." https://trac.nginx.org/nginx/ticket/796, 2022.

[47] "MariaDB Bug #23988. SST failed: No route to host after set global wsrep_node_name on donor." https://jira.mariadb.org/browse/MDEV-23988, 2020.

[48] "MySQL 5.7 Reference Manual. EXPLAIN Output Format." https://dev.mysql.com/doc/refman/5.7/en/explain-output.html, 2022.

[49] Q. Zhang, J. Wang, M. A. Gulzar, R. Padhye, and M. Kim, "Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 722–733.

[50] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[51] "Apache Hadoop 3.3.2 - HDFS Architecture." https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, 2022.

[52] R. Cheng, L. Zhang, D. Marinov, and T. Xu, "Test-case prioritization for configuration testing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 452–465.

[53] F. Behrang, M. B. Cohen, and A. Orso, "Users beware: preference inconsistencies ahead," in *Joint Meeting on Foundations of Software Engineering*, 2015, pp. 295–306.

[54] H. Huang, M. Wen, L. Wei, Y. Liu, and S.-C. Cheung, "Characterizing and detecting configuration compatibility issues in android apps," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 517–528.

[55] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 562–571.

[56] C. Q. Adamsen, G. Mezzetti, and A. Mller, "Systematic execution of android test suites in adverse conditions," in *the 2015 International Symposium*, 2015.

[57] J. Sun, T. Su, J. Li, Z. Dong, and Z. Su, "Understanding and finding system setting-related defects in android apps," in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[58] M. Lindvall, D. Ganesan, R. Ardal, and R. E. Wiegand, "Metamorphic model-based testing applied on nasa dat – an experience report," in *International Conference on Software Engineering*, 2015.

[59] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Corts, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[60] Z. Q. Zhou, L. Sun, T. Y. Chen, and D. Towey, "Metamorphic relations for enhancing system understanding and use," *IEEE Transactions on Software Engineering*, vol. 46, no. 10, pp. 1120–1154, 2020.

[61] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 132–142.