Understanding and Detecting On-the-Fly Configuration Bugs

***Abstract*—Software systems introduce an increasing number of configuration options to provide flexibility. In order to improve the flexibility and provide persistent services, modern software systems support updating configuration options on the fly without restarting the system. However, on-the-fly updating configuration options also affects 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 open-source software 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 can generate tests for on-the-fly configuration updates with existing tests and conduct differential analysis to identify OCBugs. We evaluated PARACHUTE on 7 software systems. The results show that PARACHUTE detected 75% (42/56) of the known OCBugs, and reported 13 unknown bugs from 5 software. Until the time of writing, 11 of the unknown bugs have been confirmed or fixed by developers.

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 vaule. The requirement of restarting is impractical for software systems providing 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 may affect the system reliability at the same time. Many bug reports [5]–[14] show that, on-the-fly 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.

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 update the former one when receiving the updating command. MySQL missed to change 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.

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.*

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. Therefore, those works are hard 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 in 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 study the symptoms, root causes, and triggering conditions of OCBugs. The major findings include: 1) More than half (64%) of OCBugs have no easy-to-observe symptoms like crashes, meaning an ideal fuzzing tool can handle up to 36% cases. This result inspires us to design specific oracles for OCBugs. 2) The root causes of OCBugs arise from two aspects: *incorrect propagations* of configuration-related variables (45%), or *incorrect usages* of the variables (55%). The former cases can be detected by analyzing internal variables of the program, while the latter cases are hard to be detected by program analysis due to program-specific usage scenarios. Instead, they can only be detected by examining external behaviors of the program. More details of these two kinds of bugs will be in Section II-C.

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. Based on the root cause study, these effects 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. An internal effect does not necessarily lead to observable behaviors, which may also requires specific inputs. Meanwhile, an external effect is not always caused by wrong configuration-related variables, which can cause *incorrect propagations* bugs, but not *incorrect usages* bugs.

Parachute leverages the idea of metamorphic testing [32] to detect OCBugs using the above two types of effects. In general, Parachute tests the program with two executions. 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 [40]–[43]. 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.

We evaluate the effectiveness of Parachute in detecting both known and unknown OCBugs. First, we reproduced 38 known OCBugs from the real-world OCBugs 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] detected 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.

TABLE I: Studied software systems and their descriptions.

Project	Desc.	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	342

[†] ROption is short for Runtime Configurable Option.

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 with existing tests and conduct differential analysis to identify OCBugs. All data and source code can be found in the repository:
  https://github.com/ocbug/ocbug
- 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.

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 way 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], [33]–[35]; c) they are highly-configurable and expose many runtime configurable options; d) they are open-source and well maintained by community. These criteria ensure the impacts of studied bugs, and allows us to not only obtain the buggy and fixed code versions, but also collect related details of the bugs, such as root causes and reproduce methods.

Data Collection. We collected real-world OCBugs from tracking systems, mailing lists, and fix commits of the studied

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

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

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

Incorrect propagations of configuration-related variables	**34**
Fail to consider loading updated values	7
Load wrong updated values	16
Miss to propagate to other variables	11
Incorrect usages of configuration-related variables	**41**
Fail to consider handling updated values	8
Improperly handle updated values	27
Bad update timing that causes data race	6

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 manully validate each potential OCBugs by inspecting each issue description and related code patches. Each case is inspected by two inspectors. When they diverged, a third inspector was consulted for additional discussion until consensus was reached. It 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 would 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 analyze 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?

B. Symptoms of OCBugs

We study 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, functional error and resource misuse.

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.

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 updating. 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.

Resource Abuse. Other OCBugs (4/75=5%) may cause catastrophic resource abuse. For example, in PostgreSQL-16160 [7], option `ssl_ca_file` is used to specifie 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.

> **Finding 1**: About one third (36%) of OCBugs have obvious symptoms like crash or hang, while most (64%) of OCBugs result in functional errors that have no easy-to-observe characteristics.

This finding implies that most OCBugs are hard to be detected by the existing testing technology like Fuzzing, which typically requires easy-to-observe symptoms as test oracals. It means an ideal fuzzing tool can detected up to 36% 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.*

C. Root Causes of OCBugs

We study 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; and b) *incorrect usages* of the variables. Configuration-related variables include the variable that reads and stores the original value of the involved configuration option, as well as variables that are control/data dependent on the original variable. All these variables should be well defined druing 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) Incorrect propagations of configuration-related variables: Nearly half (34/75=45%) of OCBugs happened during propagating configuration-related variables. In specific, the propagation process may occur 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.

```
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 (argv[2],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) Incorrectly 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.*

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, remained the old value, rather than its default value. In Figure 2(a), developers fixed the bug by changing options to their default values when 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 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 turning on the option at runtime. The patch is to remove the stale variable `opt_specialflag`.

2) Incorrect usages of configuration-related variables: Besides the above cases, the other half (41/75=55%) of OCBugs are about using configuration-related variables. 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 continue 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 [36]. The patch is to create a new directory.

Bad update timing that causes data race. Users can update options at anytime during program executions. 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 in 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.*

On one hand, an internal effect does not necessarily lead to observable behaviors, which may also require other triggering conditions. On the other hand, an external effect is not always caused by wrong configuration-related variables. Instead, it may be caused by other program logic that use the variables. 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 internel effects, which load wrong updated values. In both cases, Staccato cannot report bugs. As a result, Staccato can detected 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 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. It means we need 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 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 if testing 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 all bug descriptions 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 [37], the bug is triggered by first 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 exponential explosion similar to the above paragraph. Therefore, we study option dependencies required to trigger OCBugs. Please note that these dependencies are different to 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 most (55/75=73%) of OCBugs 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 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. For example, 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 [38] 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`, which does not appear at all.

> **Finding 3.3**: Most (73%) of OCBugs do not depend on any other option. Dependencies of 20% OCBugs can be obtained by program analysis. The other 7% can only be detected by exhaustive testing of option combinations.

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 spesific, Parachute tests the target program twice, using configuration options loaded since system startup or updated on-the-fly, separately. After

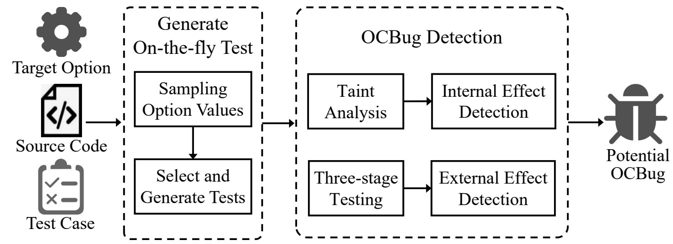

Fig. 3: Overview of Parachute

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, its 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 on 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 exections, 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* [39] to find the variables used to load options, then instruments the uses of those variables by using Clang [45]. 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 can not 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 constraints of the option by applying the existing tools [15], [16]. On one hand, PARACHUTE uses the constraint violation rules defined in [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 in [15], 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).

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*. Thus, PARACHUTE can leverage the above 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 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 *incorrect propagations* 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

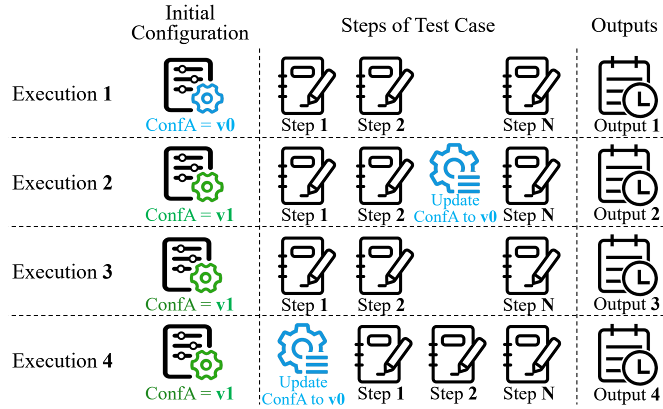

Fig. 4: Examples of metamorphic test executions

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 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* [39] 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`.

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 *incorrect propagations* bugs again, and find the overwhelming bugs (32/34=94%) are 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 [44], while the instrumentation is based on Clang [45].

2) Detecting OCBugs using External Effects: The external effects are used to detect *incorrect usages* 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 updateing 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 [40]–[43]. 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

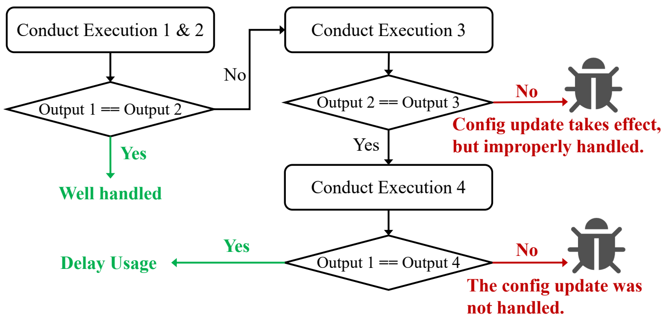

Fig. 5: Flowchart for detecting OCBugs with externel effects

address this challenge. The workflow is illustrated in Figure 5.

- **First Stage:** PARACHUTE compares external effects between the first two executions. If the effects are the same, it means the program successfully handle the updating. 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. Then, 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: delay 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 of the test. Then, PARACHUTE compares the effects between Execution 1 and 4. If the effects are the same, it indicates the program successfully handle the updating, since there is no working transaction before the updating command in Execution 4. 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 delay usage of the new option value, and does not report any bug.

IV. Evaluation

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

RQ1: How effective is PARACHUTE in detecting the known OCBugs?

```
//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

RQ2: How effective is PARACHUTE in detecting unknown OCBugs?

RQ3: How does PARACHUTE compare with the state-of-the-art configuration bug detection tool?

A. *Effectiveness of Detecting Known OCBugs*

We evaluate the effectiveness of PARACHUTE in detecting known OCBugs. As PARACHUTE is primarily designed to detect functional OCBugs, we tried our best effort to reproduce all the 44 OCBugs studied in Section II, whose symptoms are functional errors. We successfully reproduced 38 bugs. To avoid over-fitting, we followed the bug collection steps on MariaDB and Httpd, and successfully reproduced 18 bugs that are not included in the empirical study. We evaluate PARACHUTE on these 56 OCBugs.

We run PARACHUTE on a 64-bit Ubuntu 18.04 machine (8 cores, Intel Core i7-9700K, and 32GB RAM). To detect a known OCBug, we conductt PARACHUTE with the official test suite for 20 hours, on the buggy version of the software.

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 *Incorrect propagations* bugs, and nearly half (11/23=48%) of *Incorrect usages* bugs. PARACHUTE failed to detected 14 bugs due to the following reasons: 1) The triggering conditions for the bugs are not met (6 cases). As the testing space is huge, PARACHUTE uses heuristic strategies to generate tests for configuration updates. Some corner cases are missed: a) updating the target option to special values (e.g. Triggering Nginx-796 [46] needs to update the option to a new file path, but pointing to the same file); b) updating the target option more than one time; c) the dependency for the target option are hard to obtain from source code by existing works. 2) The taint analysis is not sound (2 cases). PARACHUTE is limited by complicated pointer and alias analysis. 3) Some bugs require proper test scenarios and operations (6 cases). For example, MariaDB-23988 [47] occured in a cluster of three nodes. But the official test suites do not satisfy the required scenarios and operations.

TABLE IV: The effectiveness of detecting existing OCBugs.

OCBug Type	Reproduced OCBugs	Detected by PARACHUTE	Detected by Staccato
Fail to consider loading.	8	8	8
Load wrong updated values.	15	15	0
Miss to propagate.	10	8	7
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‡	Oracle*I*	Oracle*II*	Staccato
MySQL-1	v5.7-latest	Confirmed	Type-2		✓	
MySQL-2	v5.7-latest	Confirmed	Type-2		✓	
MySQL-3	v5.7-latest	Confirmed	Type-2		✓	
MySQL-4	v5.7-latest	Confirmed	Type-1	✓	✓	✓
MySQL-5	v5.7	Confirmed	Type-2		✓	
MySQL-6	v5.7	Confirmed	Type-2		✓	
MySQL-7	v5.7	Confirmed	Type-2		✓	
Redis-1	v6.2-v7.0	Fixed	Type-1	✓		
Squid-1	v5.0-latest	Pending	Type-1	✓		✓
Squid-2	v5.0-latest	Pending	Type-1	✓		✓
Postgres-1	v14.2-latest	Confirmed	Type-1	✓	✓	✓
MariaDB-1	v10.3-latest	Fixing	Type-2		✓	
MariaDB-2	v10.3-latest	Fixing	Type-2		✓	

† The Bug ID is hidden for double-blind review.
‡ Type-1 is short for Incorrect propagations of configuration-related variables; Type-2 is short for Incorrect usages of configuration-related variables.

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

B. Effectiveness of Detecting Unknown OCBugs

We also apply PARACHUTE on the recent released version of the target systems to evaluate if PARACHUTE can detect unknown OCBugs. We evaluate PARACHUTE on the 7 software systems, including MariaDB, Httpd and the systems listed in Table I. Because each software has hundreds of runtime configurable options, we randomly select 100 options from each system for testing. We condutct PARACHUTE to test each option for 20 hours.

PARACHUTE reported 13 true positives and 3 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, shown in Table V. The 13 OCBugs come from 5 systems, including MySQL, Redis, Squid, PostgreSQL and MariaDB. Among these new bugs, 5 cases are *Incorrect propagations* bugs, and 8 cases are *Incorrect usages* bugs.

We find the unknown OCBugs would cause the systems functional errors or performance degradation. For example in MySQL, updating option `time_zone` between two same *Select* operations, would make MySQL *Query Cache* invalid to identify the same queries. Specially, changing `time_zone` had no effect on the result of the *Select* operations. However, *Query Cache* incorrectly identified them as different queries, due to improperly handling the updated value. It would also cause serious performance degradation in extreme cases; MySQL does repetitive query operations and stores redundant results, rather than returning the result from cache directly.

Meanwhile, PARACHUTE also reported 3 false positives in the target systems. 1) One false positive comes from Oracle I. PostgreSQL uses option `statement_timeout` to control the maximum time of executing any statement. PARACHUTE found the option was propagated to a global variable, but had not been changed after configuration update. However, we communicated with the developer and confirmed that it was not a bug. It is designed to updated after the current statement is executed. 2) Two false positives come from Oracle II, which are caused by inexact results of some operations in the tests. 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]. However, the number is an estimate and not always exact, which misled the external effect analysis of PARACHUTE.

Answer to RQ2: This result indicates PARACHUTE can effectively detect unknown OCBugs in popular, real-world software systems with limited false positives (3/16=19%).

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

We compare PARACHUTE with Staccato [31], one of the state-of-the-art technique for detecting configuration-related 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 the unknown bugs which were found by PARACHUTE. Because Staccato is a bug detection tool for java programs and PARACHUTE for C/C++ programs, we evaluate the theoretical upper bound of Staccato in detecting these bugs. As Staccato did not publish the reproduction steps for its detected bugs, we do not evaluate PARACHUTE on the same java programs that stacatto was evaluated on. We do not compare PARACHUTE with Fuzzing, because Fuzzing could not effectively detect functional bugs, due to the lack of special test oracles. However, Fuzzing techniques could generate various tests to help PARACHUTE detect OCBugs in the future work.

The evaluation shows that Staccato can detect 27% (15/56) of the reproduced OCBugs, shown in Table IV. On one hand, Staccato can detect less than half (15/34=44%) of *Incorrect propagations* bugs. We analyze and find that Staccato can only detect whether the option value is updated, but misses the ability to detect the correctness of the updated values. So, Staccato missed all of the bugs caused by *Loading wrong updated values*. Moreover, Staccato can detect most (7/10) of OCBugs arising from *Missing to propagate*, while also also failed to detect 3 OCBugs caused by control-flow propagation. (e.g. MySQL-28808 in Figure 1). Traditional taint analysis usually ignores this type of propagation. On the other hand, in *Incorrect usages* bugs, the configuration-related variables are correctly updated. So, Staccato has no ability (0%) to detect this type of OCBugs. However, PARACHUTE can detect all types of OCBugs.

Moreover, we find Staccato can only detect four of the 13 unknown bugs, reported by PARACHUTE, as shown in Table V. The four bugs were both caused by *Missing to propagate*. While, the other 9 bugs are caused by *Loading wrong updated values* and *Incorrect usages of configuration-related variables*. Staccato has limited ability to detect these types of OCBugs.

Answer to RQ3: This result indicates PARACHUTE can detect more types of OCBugs than the state-of-the-art technique, Staccato.

D. Discussion

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

Reproducing bugs. We tried our best to reproduce the known OCBugs, but failed to reproduce some due to the following reasons: 1) The bugs need special environment and workload to trigger. For instance, MariaDB-18699 [51] requires distributed cluster and complicated workload. 2) A few bugs need special system status and scenarios when updating the option. For instance, Redis-8030 [37] was triggered in the situation where Redis AOF write errored due to disk error.

V. THREATS TO VALIDITY

A main threat to validity is likely insufficient representativeness of configurable software used in our study. We attempt to study a wide variety of popular open-source configurable systems; the 5 studied systems cover a variety of domains, including database systems and web server. Another criterion for our selection of studied systems is that the system exposes many options which are runtime configurable. It makes us abandon some popular software (e.g. HDFS has only 16 (out of 583) runtime configurable options [52]. Most options can only be updated after restarting the system). The findings of our research may only apply to database and web server systems. Software, from other domains or closed-source, could have different characteristics.

Another main threat is likely incompleteness of keywords to collect OCBugs. To alleviate the threat, we use two types of keywords to search for the issues and commits, related to OCBugs. The final main threat is likely incorrectness of manual inspection. To minimize the effect, each bug is inspected by two inspectors. If the two inspectors diverged, a third inspector was consulted for additional discussion until consensus was reached.

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], [33], [34], [53]–[56] focus on detecting configuration-related functional defects or performance defects in software codes. Ctest [22], [53] 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 [34] suggests performance properties for configuration options to detect Configuration-handling Performance Bugs. The most related work for detecting OCBugs is Staccato [31], which is designed to find bugs for dynamic configuration updates. Staccato collects configuration-related program variables, then checks whether their values are changed after dynamic configuration updates. Our study shows that Staccato misses all of Incorrect usage bugs, and the cases which load wrong updated values. In this paper, based on our in-depth research, we conduct metamorphic testing by mutating existing tests to identify OCBugs. PARACHUTE check both the internal and external effects of configuration updates. So, PARACHUTE can detect all types of OCBugs and report diagnosis information to help fix the bugs.

Configuration Error Injection Testing. Some works [15]–[21] focus on evaluating software reliability and diagnosability regarding configuration errors. Configuration error injection testing is to inject configuration errors into the system under test (SUT), and then evaluate the SUT reaction under system test suites. 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 configuration options, including semantic type, data range and dependencies. 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 [57]–[62] use metamorphic testing [32] to detect logical bugs. Adamsen et al. [57] use specific metamorphic relations to enhance existing test suites for Android. SetDroid [58] uses setting-wise metamorphic fuzzing for finding system setting defects in Android applications. The work [59] 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 of OCBugs, we proposed two oracles to identify OCBugs.

VII. Conclusion

Many modern software support updating configuration options on the fly without restarting the system, in order to improve the flexibility of configuration and provide persistent services. However, on-the-fly updating configuration also affects the system reliability, resulting in software crashes and functional errors. We refer to the bugs caused by on-the-fly configuration updates as OCBugs. In this paper, we conducted the first in-depth study on real-world OCBugs from 5 open-source software systems. 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 can generate tests for on-the-fly configuration updates with existing tests and conduct differential analysis to identify OCBugs. PARACHUTE can detecte 75% (42/56) of the known OCBugs and 13 unknown bugs. Until the time of writing, 11 of the unknown bugs have been confirmed or fixed by developers.

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] 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.

[34] 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.

[35] 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.

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

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

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

[39] 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.

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

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

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

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

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

[45] "Clang: a C language family frontend for LLVM." https://clang.llvm.org/, 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] "MariaDB Bug #18699. Galera: Rolling upgrade: Upgraded node is stopped on commit if wsrep_trx_fragment_size ¿ 0." https://jira.mariadb.org/browse/MDEV-18699, 2019.

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

[53] 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.

[54] 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.

[55] 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.

[56] 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.

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

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.