# *WMWatcher*: Preventing Workload-Related Misconfigurations in Production Environment

Shulin Zhou, Zhijie Jiang†, Shanshan Li†, Xiaodong Liu,
Zhouyang Jia, Yuanliang Zhang, Jun Ma, Haibo Mi
National University of Defense Technology, Changsha, China
{zhoushulin, jiangzhijie, shanshanli, liuxiaodong, jiazhouyang}@nudt.edu.cn
{zhangyuanliang13, majun}@nudt.edu.cn, haibo_mihb@126.com

2023 30th Asia-Pacific Software Engineering Conference (APSEC) | 979-8-3503-4417-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/APSEC60848.2023.00038

***Abstract*—Among the misconfigurations with increasing prevalence and severity in recent years, workload-related misconfigurations, i.e. misconfigurations under certain workloads with valid configuration values, account for a significant portion. Since the runtime constraints of configuration parameters are influenced by workloads, piror researches could not handle workload-related misconfigurations at present. To solve the situation mentioned above, we conducted an empirical study on how configuration variables interact with other program variables, and summarized five handling type of the interactions happen in branch statements. Based on the study, we proposed *WMWatcher* to help system admins to prevent workload-related misconfigurations in production environment. *WMWatcher* infers the runtime constraints of configuration parameters under certain workload by instrumenting probes in source code and monitoring the corresponding status. The experiments on seven open-source software systems proved that *WMWatcher* could automatically instrument proper probes while bringing only 2.33% extra runtime overhead at most. And the case study demonstrates the effectiveness of *WMWatcher* in preventing workload-related misconfigurations in real-world scenarios.**

***Index Terms*—Misconfiguration, Workload-related, Static analysis, Runtime monitoring**

## I. Introduction

In recent years, misconfigurations have inevitably drawn tremendous attention for their increasing prevalence and severity [1]–[4], leading to service failures and outages in companies and service providers, such as Amazon, Google, and Facebook [5]–[8]. Among the misconfigurations that cause real-world problems, workload-related misconfigurations, i.e. misconfiguration under certain workloads with valid configuration values, account for a significant portion. Yin et al. [4] stated that a large percentage (46.3%∼61.9%) of the misconfigurations in their study have perfectly legal parameters, and Xu et al. [9] found that up to 53.3% of misconfigurations in their study are introduced by incorrectly settings of parameters that need to be set according to the runtime environments and workloads. Fig.1 shows a real-world example of a workload-related misconfiguration of configuration parameter "*thread_stack*" in MySQL. The description of "*thread_stack*" in the documentation shows that the valid range is 128KB∼4GB. Nevertheless, when user set the value to 192KB, which is definitely in the valid range, the server still reports a "*Thread stack overrun*" error when conducting specific SQL queries, which is caused by a misconfiguration of parameter "*thread_stack*".

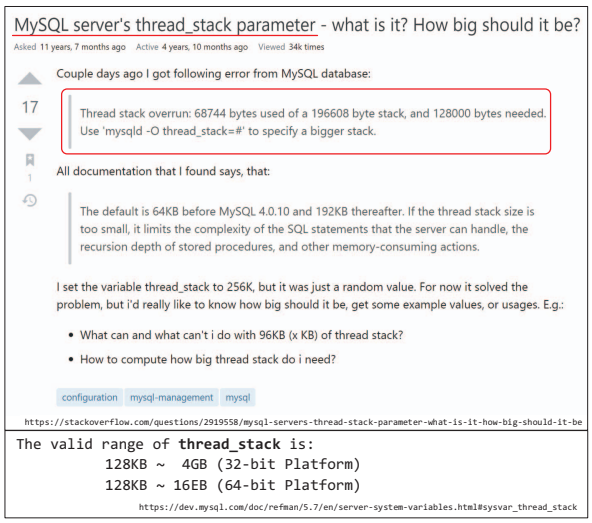



Fig. 1. A motivated example of misconfiguration caused by valid value.

Prior researches [10]–[18] apply various approaches to prevent misconfigurations. They either statically infer configuration constraints from source code [10]–[14] and documentation [15], or simulate the execution of software system to expose misconfigurations early [16]–[18]. However, none of them can prevent workload-related misconfigurations in production. Static methods [10]–[15] could only infer general configuration constraints covering various situations, and simulation-based methods [16]–[18] could only simulate the simple context that can be determined statically [16] or generated by limited test cases [17], [18].

The intrinsic cause of workload-related misconfigurations is that, the status of program variables that interact with *configuration variables* (the program variables that store the value of configuration parameters in source code) is influenced by workloads. Then, the corresponding configuration

†Both Zhijie Jiang and Shanshan Li are the corresponding author.
‡This research was funded by NSFC No.62272473, the Science and Technology Innovation Program of Hunan Province(No.2023RC1001) and NSFC No.62202474.

2640-0715/23/$31.00 ©2023 IEEE
DOI 10.1109/APSEC60848.2023.00038

constraints are subsequently changed when workload changes, making the original valid configuration value invalid anymore. Neither static analysis methods nor simple simulation knows the exact constraints of configuration parameters under various workloads, leading to the incapability of preventing workload-related misconfigurations in production environments. Therefore, in order to figure out how configuration variables interact with other program variables in source code, we carried out an empirical study on four popular open-source software systems. Considering that branch statements often play the major role in deciding the runtime control-flow of software systems, we mainly focused on the behaviors of configuration-related branch statements in our study. To simplify the expression, we hereinafter refer the branch statements involving configuration variables and other program variables to as *branch interactions* of configuration parameters, and refer the subsequent handling code of the branch statement to as *handling logic* of current *branch interaction*.

Based on the empirical study, we summarized five handling types of the handling logic in source code. Derived from their characteristics, we designed and implemented *WMWatcher* (**W**orkload-related **M**isconfiguration **Watcher**), to help system admins to prevent workload-related misconfigurations in production environment. *WMWatcher* consists of two phases, 1) instrumenting probes for branch interactions in source code, and 2) monitoring the corresponding program variables at runtime. Based on the monitoring information, *WMWatcher* could infer the constraints of configuration parameters under current workload, thus helping system admins to prevent misconfigurations when configuration adjustment are needed. We conducted experiments on seven open-source software systems to evaluate *WMWatcher*. *WMWatcher* instruments 214 probes in total, with an average 93.5% precision and 84.7% recall in identifying the handling types of the handling logic. Also, performance testing on benchmarks illustrated that *WMWatcher* brings only 2.33% extra runtime overhead at most. What's more, with several case studies related to real-world scenarios, we demonstrate the effectiveness of *WMWatcher* in preventing workload-related misconfigurations in production environment.

In summary, this paper makes the following contributions:

- We conducted an empirical study on four popular open-source software systems about how configuration variables interact with other program variables in branch statements to influence the control-flow of program, and summarized five handling types of the handling logic after branch interactions.
- We designed and implemented *WMWatcher*, an automated tool to instrument probes for branch interactions in source code, and monitor the corresponding program variables at runtime. *WMWatcher* could infer the constraints of configuration parameters under current workload for system admins, and help to prevent misconfigurations when configuration adjustment are needed.
- We evaluated *WMWatcher* on seven open-source software systems. *WMWatcher* could reach 93.5% precision and 84.7% recall in identifying the handling types. Also, *WMWatcher* brings only 2.33% extra runtime overhead at most. With several typical case studies, we demonstrated the effectiveness of *WMWatcher* in misconfiguration prevention in production environment.

## II. Empirical Study

Since the main cause for workload-related misconfigurations is that the corresponding configuration constraints are changed under different workloads, we took an in-depth look into the source code about how configuration variables interact with other program variables. We mainly focused on the branch statements involving both configuration variables and other program variables, because branch statements decide the control-flow and control the software's behaviors in most scenarios. This section mainly introduces the methodology of our empirical study, and exhibits the results of the study.

### A. Methodology

*1) Data Set:* As shown in Table.I, we chose four well-known software systems to conduct the empirical study. All of these software systems are open-source server systems and widely deployed in the field. Considering the fact that there are quite a few configuration parameters in these software systems, we focused on the configuration parameters in numeric types in our study, since numeric parameters commonly have wide valid ranges, and often cause misconfigurations in production environment [9].

*2) Study Methodology:* There are three steps in our manual analysis. First, we locate the corresponding configuration variables of configuration parameters in source code. Then, we identify the usages of these configuration variables in branch statements, in which they are compared with other program variables rather than constants. Finally, we classify the handling types after the branch statements based on the handling logic and the semantics of current caller function. This procedure was conducted by two researchers with at least four years of experience in software engineering, and a third researcher as judge with eight years of research experience. The overall study took almost two person-months.

**i) Mapping of Configuration Parameter-Variables.** Xu et al. [10] find that software developers often use clean interface to manage the configuration parameter-to-variable mapping information. Then Zhou et al. [19] implemented an automated tool ConfMapper to accomplish the parameter-to-variable mapping task. We conducted the configuration parameter-to-variable mapping based on ConfMapper, with manual check to ensure the correctness.

**ii) Identification and Classification of Branch Interactions.** To identify branch interactions in source code, we search the usages of configuration variables in the following scenarios: 1) A configuration variable is directly compared with other program variables in branch statements, e.g. `IfStmt`, `WhileStmt`; 2) A configuration variable is directly called as an argument of a function, then the corresponding parameter in function implementation is compared with other program

TABLE I
NUMBER OF CONFIGURATION PARAMETERS IN STUDIED SOFTWARE SYSTEMS.

| Software | Number of Config Params | Number of Config Params in Numeric Type |
|---|---|---|
| PGSQL | 285 | 124 |
| Httpd | 702 | 128 |
| Redis | 146 | 52 |
| MySQL | 461† | 95 |
| **Total** | **1594** | **399** |

† Note: We focus on the configuration of two main storage modules, InnoDB and MyISAM, in MySQL.

TABLE II
STATISTICS OF DIFFERENT HANDLING TYPES IN STUDIED SOFTWARE SYSTEMS.

| Software | HL† | ULB | IF | PI | NL |
|---|---|---|---|---|---|
| PGSQL | 8 | 4 | 3 | 7 | 11 |
| Httpd | 34 | 5 | 0 | 1 | 24 |
| Redis | 22 | 6 | 7 | 1 | 20 |
| MySQL | 50 | 12 | 5 | 2 | 38 |
| **Total** | **114** | **27** | **15** | **11** | **93** |

† Note: **HL** represents for *Hard Limit*, **ULB** represents for *Upper/Lower Boundary*, **IF** represents for *Incomplete Functionality*, **PI** represents for *Performance Influence*, and **NL** represents for *Normal Logic*.

variable in branch statements; 3) A configuration variable is directly assigned to a program variable $var_p$, or indirectly assigned to a program variable $var_p$ after the calculation with other constants, then this program variable $var_p$ is used in a similar way to scenario 1 and 2; 4) A configuration variable is directly called as an argument of a function, then the corresponding parameter in function implementation is used similar to scenario 3. During this procedure, two researchers cross-checked their collection of configuration usages. If there was any disagreement, the judge was consulted to reach final consensus.

Based on the identified usages of configuration variables in source code, we continued to classify the handling logic after branch interactions. We followed the same cross-check methodology as the above steps, and additionally, to make the result more consistent, we consulted a third of the classification results in our group's weekly meeting. Finally, we summarized five handling types of the handling logic. The detailed illustration will be presented in Sec.II-B.

### B. Handling Types After Interaction

We classified the handling logic into five handling types, namely *Hard Limit*, *Upper/Lower Boundary*, *Incomplete Functionality*, *Performance Influence*, and *Normal Logic*. We list the statistics of our study in Table.II. The following is the detailed description and examples of these five handling types.

**Hard Limit.** This type means that if the status of program variable doesn't match the requirement of current configuration setting, the program or current functionality will stop or fail. For example, if variable `newTotal` is greater than configuration variable `temp_file_limit` times constant `1024`, the program will log the error and exit.

```
// postgresql-11.8/src/backend/storage/file/fd.c
newTotal += newPos - vfdP->fileSize;
if(newTotal > (uint64) temp_file_limit * (uint64) 1024)
  ereport(ERROR,
    (errcode(ERRCODE_CONFIGURATION_LIMIT_EXCEEDED),
    errmsg("temporary file size exceeds
    temp_file_limit (%dkB)", temp_file_limit)));
```

**Upper/Lower Boundary.** The type set the configuration variable as the boundary of program variables, thereby affecting certain functionality of program. Compared to Hard Limit type, we call this kind of boundary limit as "soft" limit, because if the value of configuration variable "break" the limit, the software does not go wrong. For example, configuration variable `dconf->minex` is applied as the lower boundary of variable `x`. Similar usage is applied to configuration variable `dconf->maxex` as the upper boundary. Based on these two upper/lower boundary, the Httpd server only handles variable `x` between `dconf->minex` and `dconf->maxex`.

```
// httpd-2.4.48/modules/cache/mod_cache.c
x = control.s_maxage ? control.s_maxage_value
          : control.max_age_value;
x = x * MSEC_ONE_SEC;

if (x < dconf->minex) {
  x = dconf->minex;
}
if (x > dconf->maxex) {
  x = dconf->maxex;
}
```

**Incomplete Functionality.** In this type, the configuration variable pre-checks whether a varialbe reach the threshold, then controls whether a functionality should be continued. For example, if variable `okslaves` is not greater than configuration variable `server.cluster_migration_barrier`, it means there is no enough `okslaves` in master node, and the slave migration functionality will not be conducted.

```
// redis-6.2.5/src/cluster.c
void clusterHandleSlaveMigration(int max_slaves) {
  ... ...
  /* Step 2: Don't migrate if my master will not
    be left with at least 'migration-barrier'
    slaves after my migration. */
  if (mymaster == NULL) return;
  for (j = 0; j < mymaster->numslaves; j++)
    if (!nodeFailed(mymaster->slaves[j]) &&
      !nodeTimedOut(mymaster->slaves[j])) okslaves++;
  if (okslaves <=
      server.cluster_migration_barrier) return;
  ... ...
}
```

**Performance Influence.** This type means the different branch result will influence the performance to a degree. Take the following code snippet as example, the configuration variable `thd->variables.max_heap_table_size` is used to control the location where temp tables are created, i.e. choose to store in memory or in disk. When variable `outer_fanout` times `rowsize` is greater than `thd->variables.max_heap_table_size`, the sql_planner in MySQL will will create temp table in disk, thus causing the performance degradation.

```
// mysql-5.7.33/sql/sql_planner.cc
Cost_model_server::enum_tmptable_type tmp_table_type;
if (outer_fanout * rowsize <
    thd->variables.max_heap_table_size)
  tmp_table_type= Cost_model_server::MEMORY_TMPTABLE;
else
  tmp_table_type= Cost_model_server::DISK_TMPTABLE;
```

**Normal Logic.** This type covers the normal business controlled by branch interaction. For example, if variable `dbpos` is less than configuration variable `conf->bufbytes`, the program will do the normal memory copy operation, and the copy size is calculated based on the different value of `len` at runtime.

```
// httpd-2.4.48/modules/generators/mod_cgi.c
if (conf->logname && dbpos < conf->bufbytes) {
  int cursize;
  if ((dbpos + len) > conf->bufbytes) {
    cursize = conf->bufbytes - dbpos;
  }
  else {
    cursize = len;
  }
  memcpy(dbuf + dbpos, data, cursize);
  dbpos += cursize;
}
```

### C. Identification of Handling Types

Based on the manual study, we found that there are common code patterns for *Hard Limit*, *Upper/Lower Boundary*, and *Incomplete Function*.

For *Hard Limit* type, 86.8% (99/114) of the handling logic contains error-handling log statements (78/99), error-related jumping statements (19/99), or error-handling function calls (2/99). We could identify them by checking whether there is any behavior of exiting, aborting, or error handling in the handling logic. For *Upper/Lower Boundary* type, we found that 92.6% (25/27) of them fall into the similar code pattern shown in Fig.**??**(b). So it could be identified by matching the patterns. For *Incomplete Function* type, 73.3% (11/15) of them are directly `return` or `return NULL` after the branch statements, and the rest are usually contain extra handlings like with an audit log statement (not error-handling log statements), release a lock, etc. So we could search the early `return/exit` operations in function implementation to identify them. We will illustrate the detailed implementation in Sec.III.

This paper doesn't cover the identification of *Performance Influence* and *Normal Logic* type. Considering the various functionality and implementation in different software systems, it is difficult to identify these two types automatically. Take the *Performance Influence* type in following code snippet as example. When variable `levels_needed` (line 4) is not less than configuration variable `geqo_threshold`, Postgres daemon will apply `GEQO` mechanism to plan queries (line 5), otherwise it will apply standard join search algorithm (line 7). It is impractical to automated understand the performance's difference between `standard_join_search()` and `geqo()` under certain worload.

```
// postgresql-11.8/src/backend/optimizer/path/allpaths.c
if (join_search_hook)
  return (*join_search_hook) (root, levels_needed,
                  initial_rels);
else if (enable_geqo && levels_needed >= geqo_threshold)
  return geqo(root, levels_needed, initial_rels);
else
  return standard_join_search(root, levels_needed,
                  initial_rels);
```

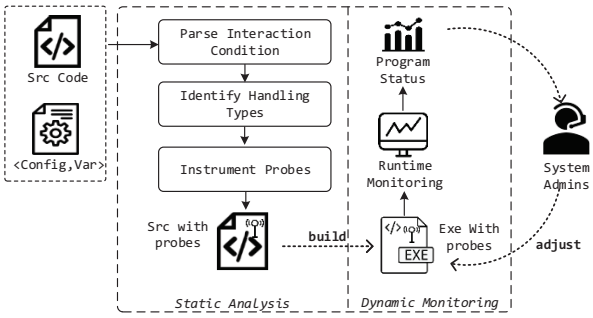



Fig. 2. Workflow of *WMWatcher*.

## III. Design of *WMWatcher*

In this section, we present the details of *WMWatcher*. We first give a schematic overview of *WMWatcher*, and then describe each component individually.

### A. Workflow of WMWatcher

The workflow of *WMWatcher* is illustrated in Fig.2. *WMWatcher* consists of two main parts, static analysis part and dynamic monitoring part. In static analysis part, *WMWatcher* takes configuration parameter-variable pairs as input, and outputs the instrumented source code. In detail, *WMWatcher* locates the branch interaction, and parses the branch condition at first. Then, *WMWatcher* identifies the handling types of this branch interaction. Finally, *WMWatcher* instruments probe into the source code based on the branch condition and handling type. In dynamic monitoring part, *WMWatcher* monitors the runtime status based on the instrumented probes, and processes the data according to their handling types, finally outputs the corresponding configuration constraints to system admins.

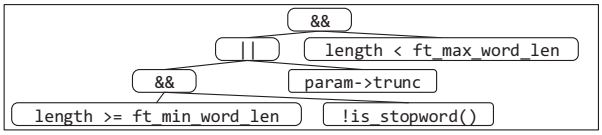
### B. Parsing Interaction Condition

*WMWatcher* first follows the same procedure mentioned in Sec.II-A to locates the usages of configuration variables in source code. Then, *WMWatcher* parses the branch conditions of those usages and exclude the usages that configuration variables compared with other configuration variables or constants. Given a branch condition, *WMWatcher* first parses it into a binary tree, in which all the non-leaf nodes are binary operator $\&\&$ and $||$, and all leaf nodes are basic logical expressions. For example, the branch condition in Fig.3(a) is parsed into the binary tree in Fig.3(b).

Based on the condition binary tree, *WMWatcher* generates a pair of $< pre\_condition, compare\_condition >$ to describe the branch condition related to configuration variables. A $compare\_condition$ is the logical expression that contains configuration variable (or alternative configuration variable found in usage collection). A $pre\_condition$ is

```
// mysql-5.7.33/storage/myisam/ft_parser.c
if (((length >= ft_min_word_len && !is_stopword( (char*)
     word->pos,
     word->len)) || param->trunc) && length <
          ft_max_word_len)
{
    *start=doc;
    param->type= FT_TOKEN_WORD;
    goto ret;
}
```

(a) Code snippet of a branch condition.




(b) Parsed binary tree of the branch condition.

Fig. 3. An example of branch condition and the parsed binary tree.

the AND operation of the logical expressions that should be `True` to ensure the whole branch condition be `True` when *compare_condition* is `True`. Take Fig.3(a) as example, the *compare_condition* of configuration variable `ft_min_word_len` is `length >= ft_min_word_len`, and the corresponding *pre_condition* is `((!is_stopword()) && (length < ft_max_word_len))`.

Next, *WMWatcher* filters out the configuration usages unrelated to program variables, i.e. the corresponding *compare_condition* does not contain any other program variables except configuration variables. Those kinds of usages are well handled by prior researches [10], [11], and doesn't not have different impact on the program under different workloads. After this, *WMWatcher* locates the branch interactions in source code, and generates the corresponding *pre_condition* and *compare_condition*.

### C. Identifying Handling Types

As mentioned in Sec.II-C, *WMWatcher* mainly focuses on identification of *Hard Limit*, *Upper/Lower Boundary*, and *Incomplete Function* types.

**i) Hard Limit.** As prior researches [20]–[24] mentioned, developers often name identifiers and function names with tokens that could represent their semantics. *WMWatcher* utilizes the semantics in those tokens to identify *Hard Limit* type, and mainly focuses on the following three situations: a) There is an error-handling log statements in the handling logic; b) the labels of jumping statements (i.e. `goto` statements or `return` statements) in the handling logic are error-related; c) There are error-handling function calls in the handling logic. If at least one of the above situations occurs in the handling logic, *WMWatcher* treats it as *Hard Limit* handling type.

To identify error-handling log statements, *WMWatcher* refers to the method proposed by SmartLog [25]. Specially, if there is no enough information to identify error-handling log statements by SmartLog's method, *WMWatcher* takes the commonly used log level in log statements into consideration. If a log statement is at a high severe log level, *WMWatcher* treats it as a error-handling log statements even if it cannot be identified by SmartLog's methods. To identify the error-related jumping statements and error-handling function calls, *WMWatcher* mines the words' semantics used in jumping identifiers and the name of function calls. Specifically, *WMWatcher* builds a keywords list $\Psi$ that contains the synonyms of words that describe error, e.g. $error$, $exit$. Then, *WMWatcher* segments the jumpping identifier or name of function call into word set $\Omega$, and match with list $\Psi$. If there is any match between $\Omega$ and $\Psi$, *WMWatcher* identifies the handling type as *Hard Limit* type. For example, the jumping statement `goto write_failed` in MySQL matches with $\Psi$ at word "fail", and function call `ExitPostmaster(1)` in PostgreSQL matches with $\Psi$ at word "exit". Considering that error-handling code is often not too long, *WMWatcher* filters out the handling code that is longer than 5 lines to reduce false positives.

**ii) Upper/Lower Boundary.** *WMWatcher* identifies this type by matching specific code patterns. Similar to the example shown in Fig.**??**(b), *WMWatcher* focuses on the code patterns shown in following code snippet, where `lv` is the program variable, and `cv` is the corresponding configuration variable.

```
if(lv > cv) // upper boundary
    lv = cv;
if(lv < cv) // lower boundary
    lv = cv;

lv = (lv > cv) ? cv : lv; // upper boundary
lv = (lv < cv) ? cv : lv; // lower boundary
```

Especially, functions like `min()/max()` in libraries are often used to do the upper/lower boundary limitation in source code. *WMWatcher* treats them as the same code pattern to identify *Upper/Lower Boundary* type.

**iii) Incomplete Functionality.** To identify *Incomplete Functionality* type, *WMWatcher* mainly focuses on the early `return/exit` in function definition. Specifically, *WMWatcher* first filters the handling logic which is directly "`exit()`", "`return;`" or "`return NULL;`". Then, if there is any statements rather than simple `return` or `exit` in the rest part of the function implementation, *WMWatcher* regards this handling type as Incomplete Functionality type.

### D. Instrumenting Probes

*WMWatcher* instruments probes according to branch interactions' *pre_condition*, *compare_condition*, and handling types. Specifically, *WMWatcher* apply *compare_condition* to acquire the expression between configuration variable and all other program variables, then use *pre_condition* as the precheck of recording statement. The instrumented probe is in the form of 5-tuple $p =< pc, n_c, v_c, v_{vs}, op, type >$, where $pc$ is the *pre_condition*, $n_c$ is the name string of current configuration parameter, $v_c$ is the configuration variable, $v_{vs}$ is the value of expression that compared with $v_c$ in the converted *compare_condition*, $op$ is the operator between $v_c$ and $v_{vs}$, and $type$ is the handling type. Take Fig.**??**(a) as example, the instrumented pseudo probe is shown in the following code snippet.

```
  newTotal += newPos - vfdP->fileSize;
+ n_c = "temp_file_limit";
```

```
+ v_c = temp_file_limit;
+ v_vs = (newTotal/((uint64) 1024));
+ op = "<";
+ type = "HARD_LIMIT";

+ if (true) // pre_condition
+ {
+   record(n_c, v_c, v_vs, op, type);
+ }
- if(newTotal > (uint64)temp_file_limit*(uint64)1024)
+ if( v_c < v_vs ) // compare_condition
    ereport(ERROR,
      (errcode(ERRCODE_CONFIGURATION_LIMIT_EXCEEDED),
        errmsg("temporary file size exceeds
        temp_file_limit (%dkB)", temp_file_limit)));
```

### E. Monitoring Status

After building and deploying the instrumented software system, *WMWatcher* infers the configuration constraints in the dynamic monitoring part in Fig.2. In detail, based on the runtime monitoring information, *WMWatcher* infers the constraints related to *Hard Limit* type as "***hard limit***" constraints, and infers those related to *Upper/Lower Boundary* and *Incomplete Function* types as "***soft limit***" constraints. To infer hard limit constraints in instrumented code snippet in Sec.III-D, *WMWatcher* records the $v_{vs}$ value that mostly close to $v_c$ according to their *op* operator type, and output to system admins. To infer soft limit constraints, *WMWatcher* records all the $v_{vs}$ values that probes output, and generates the corresponding distribution statistics as output. To be noticed, breaking the soft limit constraints would not directly cause the software system or paritial functionality failed, so it is usually treated as a warning for experienced system admins to better understand how program behaves under current workload, and choose suitable configuration values to achieve better performance or other intentions when configuration adjustment are needed.

## IV. Evaluation

In this section, we evaluate *WMWatcher* upon 7 popular open-source software systems. We try to answer three Research Questions as follows:

**RQ1**: Can *WMWatcher* prevent workload-related misconfigurations in real-world scenarios?

**RQ2**: How accurate is *WMWatcher* on instrumenting probes in static analysis part?

**RQ3**: How much extra overhead is brought by *WMWatcher*?

### A. Experiment Setup

To evaluate *WMWatcher* from different aspects, we conducted a series of experiments on seven open-source software systems listed in the first column of Table.III. In case of the bias on *WMWatcher*'s design from our empirical study in Sec.II, we newly added three other software systems to prove the generalization. All the target software systems are representative and widely-used in the field. Also, all of them are server systems, requiring high stability and reliability in production environment. We manually collected the numeric configuration parameters of these seven software systems, and generated the configuration parameter-variable pairs based on ConfMapper [19] with manual check. All the experiments are deployed on a machine with Ubuntu 18.04 operating system, the Intel i7 9700k CPU, and 16 GB memory.

### B. RQ1: Effectiveness of WMWatcher

To demonstrate the effectiveness of *WMWatcher* in preventing workload-related misconfigurations in production environments, we built 9 real-world scenarios related to our target software systems. We take the configuration parameter "*thread_stack*" in MySQL, and "*LimitRequestFields*" in Httpd as example to illustrate how *WMWatcher* helps system admins to prevent misconfigurations.

*1) Parameter "thread_stack" in MySQL:* As illustrated in Sec.I, configuration parameter "*thread_stack*" in MySQL would cause misconfiguration when user set it to 192KB, and users are confused by this abnormal scenario since they set a value in the valid range. The reason for this misconfiguration is that parameter "*thread_stack*" has a hard limit for its min value under this workload, if user changes "*thread_stack*" to a value less than this limit, MySQL server will encounter the misconfiguration.

```
// mysql-5.7.33/sql/sql_parse.cc
if ((stack_used=used_stack(thd->thread_stack,
    (char*) &stack_used)) >=
    (long) (my_thread_stack_size - margin))
{
  char* ebuff= new (std::nothrow)
          char[MYSQL_ERRMSG_SIZE];
  my_snprintf(ebuff, MYSQL_ERRMSG_SIZE,
        ER(ER_STACK_OVERRUN_NEED_MORE), stack_used,
        my_thread_stack_size, margin);
  my_message(ER_STACK_OVERRUN_NEED_MORE,
        ebuff, MYF(ME_FATALERROR));
  delete [] ebuff;
  return 1;
}
```

If *WMWatcher* is applied to MySQL, the value of `stack_used + margin` could be monitored under current workload, then the corresponding constraint, i.e. the minimal value of "*thread_stack*", could be inferred. If users need to decrease the value of "*thread_stack*", they should keep it satisfy the constraint to ensure the normal service of MySQL server.

*2) Parameter "LimitRequestFields" in Httpd:* Configuration parameter "*LimitRequestFields*" is used to control the maximum number of fields in a http request. A http request's field contains rich information, such as media type, charset, encoding, cookie information, and so on. Suppose that there is risk of DDoS (Distributed Denial of Service) attack, server admins would decrease the value of "*LimitRequestFields*" to avoid the worthless consumption of server resources by malformed packets. However, if users use a lot of cookies in their browsers, the number of fields in a http request would exceed the limit of "*LimitRequestFields*", then Httpd server will refuse the request of users, and return the `400` error code to users, as the hard limit shown in following code snippet. If *WMWatcher* is applied to Httpd Server, server admins would know the highest value of fields' number in recent access history, and choose suitable value to ensure both security and availability.

TABLE III
THE ACCURACY OF *WMWatcher* IN IDENTIFYING HANDLING TYPES.

| Software | # of Probes | P† | R† | F1† |
|---|---|---|---|---|
| PGSQL | 12 | 100% | 80.0% | 88.9% |
| Httpd | 36 | 80.6% | 74.4% | 77.3% |
| Redis | 30 | 100% | 85.7% | 92.3% |
| MySQL | 58 | 94.8% | 82.1% | 88.0% |
| Vsftpd | 5 | 100% | 100% | 100% |
| Nginx | 40 | 90.0% | 87.8% | 88.9% |
| Postfix | 33 | 100% | 97.1% | 98.5% |
| **Total** | **214** | **93.5%** | **84.7%** | **88.9%** |

† Note: "P" represents for "Precision", "R" represents for "Recall", and "F1" represents for "F1-Score".

```
// httpd-2.4.48/server/protocol.c
if (r->server->limit_req_fields
  && (++fields_read > r->server->limit_req_fields))
{
  r->status = HTTP_BAD_REQUEST;
  apr_table_setn(r->notes, "error-notes", "The number of"
    "request header fields exceeds this server's limit.");
  ap_log_rerror(APLOG_MARK, APLOG_INFO, 0, r,
    APLOGNO(00563), "Number of request headers exceeds"
    "LimitRequestFields");
  return;
}
```

### C. RQ2: Accuracy of WMWatcher

In static analysis part, *WMWatcher* locates the branch interactions, then instruments probes on the parsed condition and handling type. *WMWatcher* instruments 214 probes in the seven software systems in total.

To answer RQ2, we mainly focus the accuracy of parsing branch interactions' condition and identifying handling types. To verify the correctness in generating *pre_condition* and *compare_condition*, we randomly sampled 200 output cases of *WMWatcher*, and manually checked the correctness based on the context of the branch conditions. Given the circumstances that this procedure handles the fixed code pattern, *WMWatcher* could reach 100% correctness in parsing branch conditions. As for the accuracy of identifying handling types, we manually checked the results and show the statistics in Table.III. On average, *WMWatcher* could reach 93.5% precision and 84.7% recall. Also, the precision and recall on the three newly-added software systems are better than the first four used in our empirical study, thus demonstrating the generalization of *WMWatcher*.

We manually analyzed the false positive (FP) and false negative (FN) cases. The majority of the FPs (71.4%) are mainly introduced by mistakenly identifying configuration variables as other common program variables. As the following code snippet shows, the variable `retained->first_thread_limit` points to the original value of configuration variable "*ThreadLimit*", and *WMWatcher* mistakenly treats it as a non-configuration variable.

```
// httpd-2.4.48/server/mpm/event/event.c
if (!retained->first_thread_limit)
{
  retained->first_thread_limit = thread_limit;
}
else if (thread_limit != retained->first_thread_limit)
{
  /* don't need a startup console version here */
  ap_log_error(APLOG_MARK,APLOG_WARNING,0,s,APLOGNO(00506)
       "changing ThreadLimit to %d from original
       value of %d not allowed during restart",
       thread_limit, retained->first_thread_limit);
  thread_limit = retained->first_thread_limit;
}
```

The rest of the FPs (21.4%) are caused by various reasons, such as mistakenly identifying function calls like `log_mutex_exit_all()` as error-handling-related, or misunderstanding the complex semantics in context.

The FN mainly comes from the following three aspects. First, 58.3% of the cases are not coverd by *WMWatcher*. In the following code example, MySQL uses local variable `connection_accepted` to record the error status, and returns the variable at the end of the function implementation, rather than directly returns an error code.

```
//mysql-5.7.33/conn_handler/connection_handler_manager.cc
bool Connection_handler_manager::valid_connection_count()
{
  bool connection_accepted= true;
  mysql_mutex_lock(&LOCK_connection_count);
  if (connection_count > max_connections)
  {
    connection_accepted= false;
    m_connection_errors_max_connection++;
  }
  mysql_mutex_unlock(&LOCK_connection_count);
  return connection_accepted;
}
```

Second, 27.8% of the cases require further comprehension of the context to identify the handling types. Take following code as example, the `while` statement implements an Upper Boundary checking mechanism between configuration variable `server.slowlog_max_len` and variable `server.slowlog`.

```
// redis-6.2.5/src/slowlog.c
while(listLength(server.slowlog)>server.slowlog_max_len)
  listDelNode(server.slowlog, listLast(server.slowlog));
```

Finally, 13.8% of the cases are caused by the lack of AST information for the corresponding code snippets. In some cases, the AST is not parsed fully so that *WMWatcher* could not do the parsing and identification correctly, which might be caused by the lack of some libraries or the relatively low version of Clang we used when implementing *WMWatcher*.

### D. RQ3: Overhead of WMWatcher

The overhead of *WMWatcher* consists of three aspects. First, *WMWatcher* need to analyze source code and instrument probes in static analysis part. Second, there is extra manual effort to build the instrumented software system. Last, *WMWatcher* will bring extra runtime overhead when monitoring.

To evaluate the overhead of *WMWatcher* in static analysis part, We recorded the time *WMWatcher* used on analyzing the target software systems. The statistics are listed in Table.IV. The analysis time of *WMWatcher* on seven software systems is not greater than eight minutes. Also, considering that it is an in-house analysis procedure, the time effort of *WMWatcher*

TABLE IV
THE ANALYSIS TIME OF *WMWatcher*.

| Software | Analysis time of *WMWatcher* (seconds) |
|---|---|
| PGSQL | 186.4 |
| Httpd | 100.2 |
| Redis | 43.8 |
| MySQL | 434.7 |
| Vsftpd | 1.8 |
| Nginx | 29.3 |
| Postfix | 34.9 |

TABLE V
THE BENCHMARKS AND PERFORMANCE METRICS IN TESTING.

| Software | Benchmark | Performance Metrics |
|---|---|---|
| PGSQL | pgbench | Throughput |
| Httpd | ab | Throughput |
| Redis | redis-benchmark | Throughput |
| MySQL | sysbench | Throughput |
| Vsftpd | CeitInspector [26] | Running time |
| Nginx | ab | Throughput |
| Postfix | smtp-source | Running time |

will not influence the service of target software systems in production environment.

To build the instrumented version of software system, we only need to add the implemented dynamic link library for monitoring into the build system, e.g. `Makefile` file, `CMakeLists.txt` file, or `configure` script. In general, only 1 line extra code is added into the file for each software system in our experiments, and other mature software systems could also enable *WMWatcher* with negligible manual effort.

The runtime overhead of *WMWatcher* is the major concern for system admins. To evaluate the extra overhead brought by *WMWatcher* in dynamic monitoring, we applied commonly-used benchmarks (shown in Table.V) to measure the performance metrics of the software systems. Due to the lack of common benchmark for Vsftpd, we applied the test suites provided by Li et al. [26] to evaluate the performance metrics for Vsftpd.

We measure three versions' performance of the targe software system: 1) the original version without instrumentation; 2) the instrumented version without *WMWatcher*'s monitoring; and 3) the instrumented version with monitoring. To eliminate the variation in performance testing [27], we calculated the average value of 10 times' independent test as the final performance metrics. The statistics is shown in Fig.4. We normalized the metrics as follows: If the metric is "Throughput", we measure the extra overhead by $(-(M_2 - M_1)/M_1)$ ("w/ probes & w/o monitor" in Fig.4) and $(-(M_3 - M_1)/M_1)$ ("w/ probes & w/ monitor" in Fig.4), where $M_1$, $M_2$, $M_3$ represent for the metrics of version 1), 2), and 3) repectively. If the metric is "Running time", we measure the extra overhead by $(M_2 - M_1)/M_1$ and $(M_3 - M_1)/M_1$. In total, there is only 2.33% extra runtime overhead at most brought by *WMWatcher* in Httpd (the grey bar in Fig.4), and the overhead brought by instrumented probes is very low, with a maximum of 2.02% in

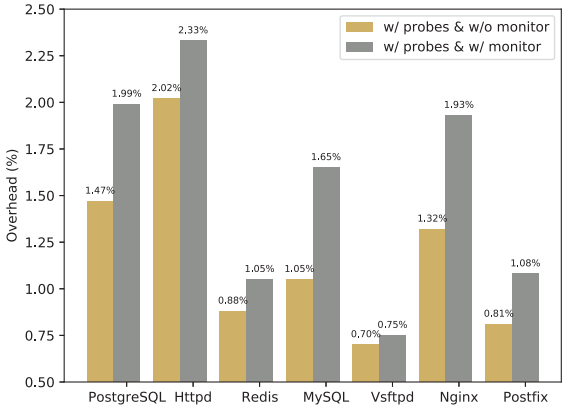


Fig. 4. The extra overhead brought by *WMWatcher* in dynamic monitoring.

Httpd (the tan bar Fig.4). Also, only 0.05%∼0.61% extra overhead is brought by the monitor mechanism of *WMWatcher*, which is negligible for the target software system.

## V. DISCUSSION

**Limitation of *WMWatcher*.** The design of *WMWatcher*, especially the static analysis part, is based on the empirical study about branch interaction in source code. Considering the rich semantics in source code, we could not identify the purpose of every handling logic after a branch statements. So we only consider three handling types that can be identified automatically to ensure accuracy. This thought could also reduce the overhead brought by *WMWatcher* in dynamic monitoring to some degree. On the other hand, *WMWatcher* only considers configuration usages in branch statements at present. There are quite a few situations that configuration parameters directly used in function calls or other mechanisms. For example, some of the memory-related configuration parameters are directly used as arguments in memory allocation functions. As a consequence of the complex semantics and implicit influence of those situations, *WMWatcher* could not handle them at present.

**Effectiveness of overhead evaluation.** In the evaluation of *WMWatcher*'s overhead, we applied the common benchmarks for each software systems. However, those benchmarks could not rest assured that all the instrumented probes will be triggered. It is non-trivial to generate input to trigger specific location of the source code in large-scale software systems. Considering that *WMWatcher* only instruments limited probes, and collects necessary values of the program, the overhead brought by *WMWatcher* will not be high even in extreme situations with $pre\_condition$ to determine whether a probe will be truly triggered.

**Generalization of *WMWatcher*.** *WMWatcher* assumes the availability of the source code. Also, the source code should be parsed by Clang Frontend. Although *WMWatcher* is implemented based on LLVM framework and supports only C/C++ software systems at present, the idea is generically applicable

to other software systems in other programming language, e.g. analyzing Java software with soot framework.

## VI. Related Work

**Misconfiguration prevention.** Quite a few researches focus on learning configuration information from source code [10]–[14], [28], or documentation [15]. Based on the empirical study about configuration usages in source code, Xu et al. [10], [12] extract configuration constraints from source code with predefined code patterns. ConfInLog [13] mines the configuration constraints from log messages based on program analysis and nlp processing. ConfigX [14] detects interaction between configuration parameters and the related code that they affected. Other researches consider mining knowledge related to configuration from the existing data [29]–[32], which require a great amount of training samples. These methods mainly focus on learning the configuration information statically, so they could only infer general configuration constraints covering various situations. *WMWatcher* combines static analysis and dynamic monitoring to infer constraints under certain workload, which is much precise than these methods.

There are methods that apply simulation execution to prevent misconfigurations in production environment [16]–[18]. PCheck [16] generate configuration checking code derived from how program uses the configuration parameters, and validate the system's configuration setting at the initialization phase. Ctest [17], [18] utilizes the existing test cases of software systems to check the configuration setting before deployment. However, PCheck and Ctest could not simulated the production environment. As a complement, *WMWatcher* could infer the runtime configuration constraints.

**Misconfiguration diagnosis.** Many researches try to diagnose the root causes of misconfigurations [33]–[39]. Some methods apply program analysis to build the relationship between program points and configuration parameters [33], [35]–[37], [39], then locate the possible culprit of misconfigurations. Other researches [34], [38] apply a comparison-based method to find out the culprit configuration parameters. When a new misconfiguration happen, they search the database to find the nearest one to guide the diagnosis. Compared the these researches, *WMWatcher* mainly focuses on misconfiguration prevention. Nevertheless, *WMWatcher* could also help to diagnose the misconfiguration with the rich monitoring information.

**Performance Tuning.** Configuration performance tuning for software systems is the hotpoint in current research fields. The main purpose is to find the optimal configuration from the extremely huge configuration space, which could be roughly classified into search-based method, and model-based methods. Search-based methods mainly try to explore the configuration space of software system to find the optimal setting within limited testing times, mainly based on searching algorithms such as control theory [40], Gaussian Process [41], Generative Adversarial Networks [42], Genetic Algorithm [43], and so on. Model-based methods mainly focus on building model between configuration values and software performance metrics [44]–[48]. Compared to these performance tuning researches, the goal of *WMWatcher* is to infer the configuration constraints of software system under certain runtime environments, which is the precondition to ensure the correctness when tuning the configuration, and it could also be used to decrease the configuration space for target software systems when finding the optimal configuration.

## VII. Conclusion

Given the prevalence of misconfigurations in software systems and existing methods' incapability of preventing workload-related misconfigurations, this paper conducted an empirical study about how configuration variables interact with program variable to control the runtime behaviors of software systems. Guided by the study, we designed and implemented *WMWatcher*, an automated tool to instrument probes in source code, and monitor the status of program at runtime. *WMWatcher* could infer the constraints of configuration parameters under certain workload to help system admins to prevent misconfigurations when configuration adjustment are needed. Experiments on seven open-source software systems and several case studies in real-world scenarios demonstrated the feasibility and effectiveness of *WMWatcher*.

## References


[1] G. Amvrosiadis and M. Bhadkamkar, "Getting back up: Understanding how enterprise data backups fail," in *Proceedings of the 2016 USENIX Annual Technical Conference*, 2016, pp. 479–492.

[2] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing? lessons from hundreds of service outages," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 2016, pp. 1–16.

[3] A. Rabkin and R. H. Katz, "How hadoop clusters break," *IEEE Software*, vol. 30, no. 4, pp. 88–94, 2013.

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

[5] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The datacenter as a computer: Designing warehouse-scale machines*. Morgan & Claypool Publishers, 2018.

[6] R. Speed, "That salesforce outage: Global dns downfall started by one engineer trying a quick fix." https://www.theregister.com/2021/05/19/salesforce_root_cause/, 2021.

[7] B. T. Sloss, "An update on sunday's service disruption." https://cloud.google.com/blog/topics/inside-google-cloud/an-update-on-sundays-service-disruption, 2019.

[8] J. Shieber, "Facebook blames a server configuration change for yesterday's outage." https://techcrunch.com/2019/03/14/facebook-blames-a-misconfigured-server-for-yesterdays-outage/, 2019.

[9] 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 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2015, pp. 307–319.

[10] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013, pp. 244–259.

[11] X. Liao, S. Zhou, S. Li, Z. Jia, X. Liu, and H. He, "Do you really know how to configure your software? configuration constraints in source code may help," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 832–846, 2018.

[12] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and discovering software configuration dependencies in cloud and datacenter systems," in *Proceedings of the 2020 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 362–374.

[13] 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 *Proceedings of the IEEE/ACM 29th International Conference on Program Comprehension*, 2021, pp. 94–105.

[14] J. Zhang, R. Piskac, E. Zhai, and T. Xu, "Static detection of silent misconfigurations with deep interaction analysis," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021.

[15] C. Xiang, H. Huang, A. Yoo, Y. Zhou, and S. Pasupathy, "Pracextractor: Extracting configuration good practices from manuals to detect server misconfigurations," in *Proceedings of the 2020 USENIX Annual Technical Conference*, 2020, pp. 265–280.

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

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

[18] 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.

[19] S. Zhou, X. Liu, S. Li, W. Dong, X. Liao, and Y. Xiong, "Confmapper: Automated variable finding for configuration items in source code," in *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security Companion*, 2016, pp. 228–235.

[20] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, and Y. Luo, "Nomen est omen: Exploring and exploiting similarities between argument and parameter names," in *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering*, 2016, pp. 1063–1073.

[21] A. Rice, E. Aftandilian, C. Jaspan, E. Johnston, M. Pradel, and Y. Arroyo-Paredes, "Detecting argument selection defects," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 2017.

[22] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language api descriptions," in *Proceedings of the 34th IEEE/ACM International Conference on Software Engineering*, 2012, pp. 815–825.

[23] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 419–428.

[24] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 111–124, 2015.

[25] Z. Jia, S. Li, X. Liu, X. Liao, and Y. Liu, "Smartlog: Place error log statement by deep understanding of log intention," in *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2018, pp. 61–71.

[26] 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.

[27] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, "Taming performance variability," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, 2018, pp. 409–425.

[28] A. Rabkin and R. Katz, "Static extraction of program configuration options," in *Proceedings of the 33rd IEEE/ACM International Conference on Software Engineering*, 2011, pp. 131–140.

[29] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "Encore: Exploiting system environment and correlation information for misconfiguration detection," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 687–700.

[30] M. Santolucito, E. Zhai, and R. Piskac, "Probabilistic automated language learning for configuration files," in *Proceedings of the 28th International Conference on Computer Aided Verification*. Springer, 2016, pp. 80–87.

[31] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac, "Synthesizing configuration file specifications with association rule learning," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–20, 2017.

[32] S. Mehta, R. Bhagwan, R. Kumar, C. Bansal, C. Maddila, B. Ashok, S. Asthana, C. Bird, and A. Kumar, "Rex: Preventing bugs and misconfiguration in large services using correlated change analysis," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*, 2020, pp. 435–448.

[33] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis." in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010, pp. 1–14.

[34] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, "Context-based online configuration-error detection," in *Proceedings of the 2011 USENIX conference on USENIX Annual Technical Conference*, 2011, pp. 28–28.

[35] A. Rabkin and R. Katz, "Precomputing possible configuration error diagnoses," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2011, pp. 193–202.

[36] S. Zhang and M. D. Ernst, "Automated diagnosis of software configuration errors," in *Proceedings of the IEEE/ACM 35th International Conference on Software Engineering*. IEEE, 2013, pp. 312–321.

[37] ——, "Which configuration option should i change?" in *Proceedings of the IEEE/ACM 36th International Conference on Software Engineering*, 2014, pp. 152–163.

[38] R. Potharaju, J. Chan, L. Hu, C. Nita-Rotaru, M. Wang, L. Zhang, and N. Jain, "Confseer: leveraging customer support knowledge bases for automated misconfiguration detection," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1828–1839, 2015.

[39] M. Sayagh, N. Kerzazi, and B. Adams, "On cross-stack configuration errors," in *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering*. IEEE, 2017, pp. 255–265.

[40] S. Wang, C. Li, H. Hoffmann, S. Lu, W. Sentosa, and A. I. Kistijantoro, "Understanding and auto-adjusting performance-sensitive configurations," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 154–168.

[41] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding faster configurations using flash," *IEEE Transactions on Software Engineering*, vol. 46, no. 7, pp. 794–811, 2020.

[42] L. Bao, X. Liu, F. Wang, and B. Fang, "Actgan: Automatic configuration tuning for software systems with generative adversarial networks," in *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 465–476.

[43] R. Singh, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Optimizing the performance-related configurations of object-relational mapping frameworks using a multi-objective genetic algorithm," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, 2016, pp. 309–320.

[44] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski, "Variability-aware performance prediction: A statistical learning approach," in *Proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering*, 2013, pp. 301–311.

[45] H. Ha and H. Zhang, "Deepperf: Performance prediction for configurable software with deep sparse neural network," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, pp. 1095–1106.

[46] M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Kastner, "White-box analysis over machine learning: Modeling performance of configurable systems," in *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering*, 2021, pp. 1072–1084.

[47] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1009–1024.

[48] L. Bao, X. Liu, Z. Xu, and B. Fang, "Autoconfig: Automatic configuration tuning for distributed message systems," in *Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering*, 2018, pp. 29–40.