MissConf: LLM-Enhanced Reproduction of Configuration-Triggered Bugs

Ying Fu*
NUDT, China
fuying@nudt.edu.cn

Teng Wang*
NUDT, China
wangteng13@nudt.edu.cn

Shanshan Li†
NUDT, China
shanshanli@nudt.edu.cn

Jinyan Ding
NUDT, China
dingjinyan20@nudt.edu.cn

Shulin Zhou
NUDT, China
zhoushulin@nudt.edu.cn

Zhouyang Jia
NUDT, China
jiazhouyang@nudt.edu.cn

Wang Li
NUDT, China
liwang2015@nudt.edu.cn

Yu Jiang†
Tsinghua University, China
jiangyu198964@126.com

Xiangke Liao
NUDT, China
xkliao@nudt.edu.cn

ABSTRACT

Bug reproduction stands as a pivotal phase in software development, but the absence of configuration information emerges as the main obstacle to effective bug reproduction. Since configuration options generally control critical branches of the software, many bugs can only be triggered under specific configuration settings. We refer to these bugs as configuration-triggered bugs or CTBugs for short. The reproduction of CTBugs consumes considerable time and manual efforts due to the challenges in deducing the missing configuration options within the vast search space of configurations. This complexity contributes to a form of technical debt in software development.

To address these challenges, we first conducted an empirical study on 120 CTBugs from 4 widely used systems to understand the root causes and factors influencing the reproduction of CTBugs. Based on our study, we designed and implemented MissConf, the first LLM-enhanced automated tool for CTBug reproduction. MissConf first leverages the LLM to infer whether crucial configuration options are missing in the bug report. Once a suspect CTBug is found, MissConf employs configuration taint analysis and dynamic monitoring methods to filter suspicious configuration options set. Furthermore, it adopts a heuristic strategy for identifying crucial configuration options and their corresponding values. We evaluated MissConf on 5 real-world software systems. The experimental results demonstrate that MissConf successfully infers the 84% (41/49) of the CTBugs and reproduces the 65% (32/49) CTBugs. In the reproduction phase, MissConf eliminates up to 76% of irrelevant configurations, offering significant time savings for developers.

*Both authors contributed equally to this research.
†Shanshan Li and Yu Jiang are the corresponding authors.

KEYWORDS

Bug reproduction, Software configuration, Software maintenance

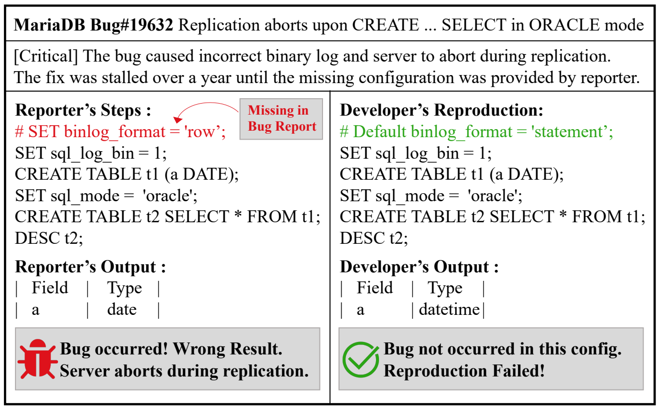
ACM Reference Format:
Ying Fu*, Teng Wang*, Shanshan Li†, Jinyan Ding, Shulin Zhou, Zhouyang Jia, Wang Li, Yu Jiang†, and Xiangke Liao. 2024. MissConf: LLM-Enhanced Reproduction of Configuration-Triggered Bugs. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3639478.3647635

1 INTRODUCTION

Bug reproduction is a crucial phase in software development, playing a pivotal role in identifying and resolving issues while ensuring the overall robustness of the software system. Once developers receive a bug report, the initial step in debugging is to reproduce the bug by executing the provided steps in the bug report. However, previous works found that bug reports from popular security forums have an extremely low success rate of reproduction(4.5%-43.8%). The main reason is the lack of information, where more than 87% reports did not include detailed information on software configurations and environments [32].

Modern software systems are highly configurable [41], providing users with a large number of configuration options. By setting the values of configuration options, software systems can be customized to achieve specific functionality or performance goals without modifying their source code [21, 33]. For example, MariaDB has more than one thousand configuration options [12], which are distributed across various modules such as storage engines, optimizers, logging systems, among others. These configuration options often manifest within conditional branch statements, dictating the specific program path to be executed.

Numerous bug reports [4–6, 8, 34] show that configuration options with non-default but legal values are crucial factors in bug reproduction. In other words, these bugs can only be triggered within specific program paths that are controlled by particular configuration options' settings. In this paper, we refer to these bugs as *Configuration Triggered Bugs*, or CTBugs for short. And we refer to those configuration options with particular values as *crucial configuration options* of CTBug. Generally, the bug reporters try to



MariaDB Bug#19632 Replication aborts upon CREATE ... SELECT in ORACLE mode

[Critical] The bug caused incorrect binary log and server to abort during replication.
The fix was stalled over a year until the missing configuration was provided by reporter.

Reporter's Steps :

```
# SET binlog_format = 'row';
SET sql_log_bin = 1;
CREATE TABLE t1 (a DATE);
SET sql_mode = 'oracle';
CREATE TABLE t2 SELECT * FROM t1;
DESC t2;
```

Reporter's Output :

```
| Field | Type |
| a     | date |
```

Bug occurred! Wrong Result. Server aborts during replication.

Developer's Reproduction:

```
# Default binlog_format = 'statement';
SET sql_log_bin = 1;
CREATE TABLE t1 (a DATE);
SET sql_mode = 'oracle';
CREATE TABLE t2 SELECT * FROM t1;
DESC t2;
```

Developer's Output :

```
| Field | Type     |
| a     | datetime |
```

Bug not occurred in this config. Reproduction Failed!

Figure 1: A real-world example of CTBug reproduction issue. *The bug fix process was stalled for more than a year due to the lack of crucial configuration information* [6].

list the configuration options that they consider important for bug reproduction in bug reports, which we refer to as *reported configuration options*. However, the reported option set lacks some crucial ones with non-default values, making the reproduction of CTBugs particularly challenging.

Figure 1 illustrates a CTBug in MariaDB [6], the reproduction and fix of which was stalled for over a year due to the absence of crucial configuration information. The bug was labeled as critical, which caused the server to record the incorrect binary log and abort during replication. As shown in Figure 1, the bug is triggered only when the option *binlog_format* is set to '*row*', whose default value is *statement*. In the row binlog format, the server adopts different logging strategies for data manipulation language (DML) [48] statements and data definition language (DDL) [47] statements, thereby triggering this bug. There are many CTBugs that remain unreproducible due to the inability to pinpoint the missing configuration options. Developers often cannot identify which crucial configuration options are missing based on the content of bug reports, resulting in these issues being postponed or not given sufficient attention. This leads to the persistent existence of hard-to-reproduce bugs, constantly putting the systems at risk of potential attacks [49].

Reproducing CTBugs faces substantial challenges. Firstly, discerning whether the inability to reproduce the CTBug is a result of missing configuration options poses a challenge. Nonetheless, this is crucial to avoid investing excessive time in configuring combinations that do not yield meaningful results. Secondly, identifying the specific configuration option that triggers the bug requires considerable effort and expertise from developers. This is particularly challenging given the huge number of software configurations available. Thirdly, selecting the appropriate option values is also a key factor in triggering CTBugs. Due to the wide range of possible values for configuration options, making the efficient sampling of suitable values a challenging task. These challenges result in a significant time and resource requirement in reproducing CTBugs, consequently contributing to a form of "technical debt" [29] in software development.

There has been much research on bug reproducing. Some works utilize in-house methods to collect a variety of failure information for replicating the environment and inputs necessary to reproduce bugs, including function call sequences [25, 26], crash stack [31, 35, 46], user interface events [23, 24], and runtime logs [51]. However, it is hard for those works to reproduce CTBugs, especially when the only accessible information stems directly from the reports themselves. Many other works [16, 22, 27, 44, 52–54] focus on converting the steps described in the bug report into test cases. For example, ReCDroid+ [54] uses natural language processing (NLP) and deep learning to synthesize event sequences. LIBRO [27] uses Large Language Models (LLMs) to automate test generation from general bug reports. However, these works assume that bug reports contain complete environment and configuration information. Consequently, they cannot address the issue of missing crucial configuration options, which usually hinders the successful reproduction of CTBugs.

To address these challenges, we first conducted an in-depth empirical study of 120 CTBugs across 4 widely-used software systems. We found that **execution path resulting from the reproduction steps intersects with the branch statements influenced by the missing crucial configuration options**. Therefore, the potential missing options could be found in the suspicious option set, which is encountered at branch statements during the execution. Moreover, the configuration options specified by users in bug reports can assist in identifying the missing crucial configurations. If some reported configuration options are not present in the program's execution path, a crucial option is probably missing.

Based on the study, we propose MissConf, an LLM-enhanced automated CTBug reproducing tool. MissConf takes the hard-to-reproduce bug and its report as inputs, and outputs whether the bug report lacks crucial configuration options and the values of the options. MissConf takes two main steps in this process. 1) MissConf first infers whether crucial configuration options are missing in the bug report. MissConf utilizes Large Language Models (LLMs) to comprehend the semantics of bug reports and execution details of reported configuration options to infer potential configuration omissions. 2) If found, MissConf employs taint analysis and dynamic monitoring methods to filter suspicious option sets. Then, it adopts a heuristic strategy to identify the crucial configuration options and corresponding values.

We evaluate the effectiveness of MissConf in inferring missing configuration options in CTBug reports, and reproducing CTBugs. We reproduced 49 known CTBugs which are not included in the study from 5 popular open-source projects. MissConf can eliminates up to 76% of irrelevant configurations. On average, MissConf correctly infers the 84% (41/49) of the CTBugs and reproduces the 65% (32/49) CTBugs.

To summarize, this paper makes three major contributions.

- We conducted the comprehensive empirical study of 120 CTBugs across 4 open-source systems, and summarized the key factors influencing CTBug reproduction.
- We designed and implemented MissConf, the first LLM-enhanced automated tool for CTBugs reproduction, offering improvement solutions for this technical debt issue. MissConf utilizes LLMs to identify CTBugs, and employs taint

Table 1: Software systems and CTBugs used in the empirical study. ***The fifth column shows the number of hard-to-reproduce CTBugs in each system, which makes developers encounter difficulties in reproducing these bugs due to the lack of crucial configuration options.***

Project	Description	# Options	# CTBugs	# Hard-to-Rep. CTBugs
MySQL	SQL DBMS	1814	68	15
MariaDB	SQL DBMS	1253	17	5
Redis	NoSQL DBMS	195	17	9
Squid	Proxy Cache	336	18	8

analysis and dynamic monitoring to detect the missing crucial configuration options. The source code of MissConf can be found at:
https://github.com/MISSCONF-2024/MissConf

- We evaluated MissConf on 5 widely-used projects. MissConf can correctly infer the 84% (41/49) of the CTBugs and reproduce the 65% (32/49) CTBugs. In the reproduction phase, MissConf eliminates up to 76% of irrelevant configurations, offering significant time savings for developers.

The remainder of the paper is organized as follows. We first conduct an in-depth empirical study of CTBugs in Section 2. Based on this, we describe the design of MissConf in Section 3. Experimental settings and evaluation results are in Section 4. Discussions and threats to validity are presented in Section 5 and Section 6. Section 7 gives an overview of the relevant literature, and Section 8 concludes.

2 UNDERSTANDING CTBUGS

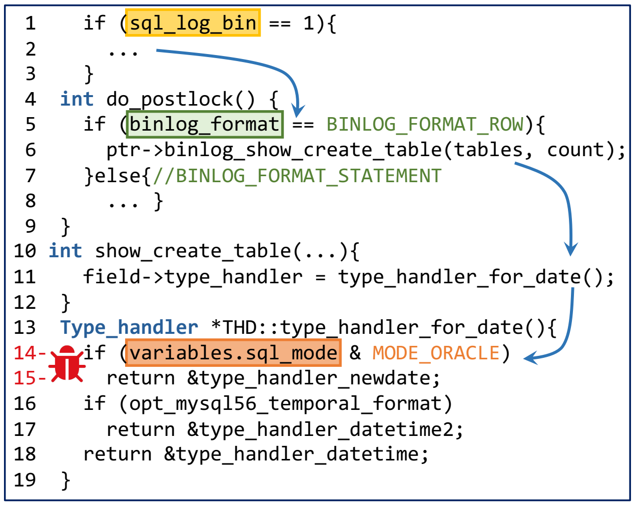
In this section, we will first describe the methodology of our empirical study on CTBugs, then introduce our findings including how configuration influences CTBug execution, and the factors affecting CTBug reproduction.

2.1 Study Methodology

The study methodology includes the criteria to choose studied subjects, and the methods to collect and analyze the bugs.

Studied Subjects. Table 1 shows the 4 software systems used in our empirical study. We selected these software systems based on the following criteria: a) These software projects span a diverse array of domains, including database management, web server, etc. b) They are highly configurable and well-tested (e.g., MariaDB has more than 1,200 configuration options). c) They are open-source and well-maintained by the community, which allows us to delve into the characteristics and triggering conditions of the bug, based on the developers' discussions.

CTBug Collection. We collected CTBugs from three sources: bug tracking systems (e.g., GitHub Issue Tracker [11], JIRA[3], Bugzilla[9]), mailing lists, and fix commits. First, We conducted searches on these sources using heuristic keywords (e.g., *assertion*



```
1    if (sql_log_bin == 1){
2      ...
3    }
4  int do_postlock() {
5    if (binlog_format == BINLOG_FORMAT_ROW){
6      ptr->binlog_show_create_table(tables, count);
7    }else{//BINLOG_FORMAT_STATEMENT
8      ... }
9  }
10 int show_create_table(...){
11   field->type_handler = type_handler_for_date();
12 }
13 Type_handler *THD::type_handler_for_date(){
14-  if (variables.sql_mode & MODE_ORACLE)
15-    return &type_handler_newdate;
16   if (opt_mysql56_temporal_format)
17     return &type_handler_datetime2;
18   return &type_handler_datetime;
19 }
```

Figure 2: A motivating example of how configurations influence CTBug execution. ***The bug has three crucial configuration options: sql_log_bin, binlog_format and sql_mode.***

fail, *segment fault*, *crash*) to collect bugs. Subsequently, we applied additional keywords related to configuration options (e.g., *configuration*, *option*, *set*) to filter out potential CTBugs. This process identified 572 candidates.

Validation and Analysis. After collecting the dataset, we identified the potential CTBug by analyzing whether the test cases in the bug reports contained configuration option set statements or configuration file information. We then tried to reproduce each potential CTBug to confirm these issues were indeed triggered by special configurations. It would be identified as a CTBug if it can be reproduced following the steps in the bug report, but became non-reproducible when certain mentioned configuration options are removed. Each case was inspected by two evaluators with 7 years of software development experience. In cases of divergence, a third senior evaluator was consulted to facilitate additional discussions until a consensus was reached. This validation and analysis process spanned two months. In the end, we accumulated a total of 120 CTBugs from the 4 projects. Among these bugs, 30% are hard-to-reproduce due to the lack of crucial configuration options in the bug report description. We conducted further analysis on each CTBug to explore the root causes and factors influencing the reproduction of CTBugs.

2.2 How Configurations Influence CTBug Execution

To understand how the crucial configuration options affect CTBugs, we analyzed the bug patch and manually tracked the execution of each bug, which enabled us to figure out the influence of configurations at the source code level.

Figure 2 shows the root cause of an example CTbug, MariaDB Bug#19632 [6, 7]. The occurrence of this bug is governed by three crucial configuration options. Specifically, the bug manifests when *sql_log_bin* is set to '*1*' (line 1), *binlog_format* is set to '*row*' (line 5) and *sql_mode* is set to with '*oracle*' (line 14). Through the propagation of data and control flow, these crucial configuration options influence the conditional variables present in branch instructions, which finally dictate whether the faulty code (line 15) could be triggered. Given that the code where the bug manifests might be considerably distant from its crucial configuration options, it is challenging for developers to pinpoint the missing configuration options through static analysis.

However, the process of executing reproduction steps and tracking the execution trace provided significant insights. We observed that, **the program execution path resulting from the reproduction steps intersects with the branch statements influenced by the missing crucial configuration options**. This inspires us to record the set of configurations encountered at branch statements during the execution of reproduction steps, thereby identifying the crucial configuration options of the CTBugs.

Moreover, the bug reporters often specify critical configuration options for bug reproduction in the report, even though these configurations may be incomplete. We refer to these options as *reported configuration options.* We discovered that if a particular critical option is missing, the program execution path might deviate from the path where the bug manifests. And some other critical options specified by reporters may not appear on this altered path. For instance, in Figure 2, if *binlog_format* is omitted in the report, the program execution would not pass through the branch statement controlled by *sql_mode* (line 14). This observation also informs our approach to detect configuration omission in reports: **After following the reproduction steps, if the reported configuration options are not found in the program's execution path, it is likely that a crucial configuration option is missing**.

Finding 1: Configurations propagate through data and control flows, influencing the control flow of the software system, and consequently impacting the execution path of CTBug.

2.3 Factors Affecting CTBug Reproduction

In order to guide and facilitate CTBug reproduction, we conduct a comprehensive study of factors affecting the triggering conditions of CTBug in this section. In specific, we study the following three sub-questions:

- What are the common types of crucial configuration options in CTBugs?
- How many missing crucial configuration options are there in hard-to-reproduce CTBug reports?
- What is the timing of setting crucial configuration options for CTBugs?

Configuration Types. We undertook a detailed analysis of the software's configuration documentation to better understand the configuration classifications of CTBugs. These configurations are mainly categorized into three fundamental types: Numeric, Enumeration, and String [30]. Software manuals usually offer extensive details on numeric configurations, delineating their valid ranges, and listing all potential valid values for enumeration configurations.

Table 2: Proportion of each type of configuration option.

Software	Enumeration	Numeric	String	Sum
MySQL	54 (80%)	11(16%)	3(4%)	68
MariaDB	10 (60%)	7 (40%)	0 (0%)	17
Redis	7 (41%)	7(41%)	3 (18%)	17
Squid	10 (56%)	6(33%)	2 (11%)	18
Total	81 (68%)	31 (26%)	8 (6%)	120

The breakdown of each configuration type is presented in Table 2. We observed that the numeric and enumeration configurations make up the majority of CTBugs, while the string type configurations account for only a minor portion, representing merely 6.7%. This inspires us to prioritize enumeration and numerical options, when detecting missing crucial configuration options.

Finding 2: Most of the configuration types involved in CTBug are enumeration and numerical types, only a few bugs (6.7%) are triggered by string configurations.

Number of Missing crucial configuration options. The number significantly impacts the reproduction of CTBugs: the more missing crucial configuration options, the larger the search space. Therefore, we initially studied the number of missing crucial configuration options in all hard-to-reproduce CTBugs. Our survey results revealed that 91.9% (34/37) of hard-to-reproduce CTBug reports had only one missing crucial configuration. This suggests that when searching for the missing crucial configuration options, we can prioritize cases with just one, thus accelerating the process.

Finding 3: Most (91.9%) of hard-to-reproduce CTBugs have only one missing crucial configuration option.

Timing of Configuration Setting. Configurations typically take effect either during system startup or through on-the-fly update while the system is running [43]. Therefore, the timing of configuration setting would also affect the reproduction of CTBug. We studied all the hard-to-reproduce CTBugs, and discovered that in 95% of these CTBug reports, the missing crucial configuration options are set before the initiation of test case executions. Essentially, in the majority of cases, the reporters do not miss documenting on-the-fly configuration updates while the system is running. In this regard, when reproducing CTBugs, focusing on setting the missing crucial configuration options before the reproduction steps can effectively reduce the search space.

Finding 4: In the case of most (95%) hard-to-reproduce CTBugs, the missing crucial configuration options are set before the execution of test cases.

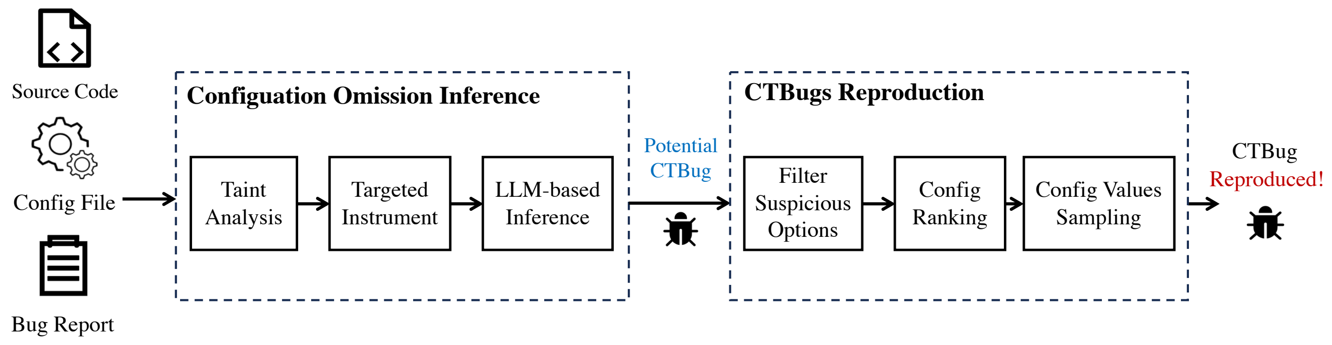


Figure 3: Overview of MissConf. MissConf mainly contains two parts: *Configuration Omission Inference* and *CTBug Reproduction.* Given a hard-to-produce bug report, MissConf first employs LLMs to infer potential configuration omissions based on program execution information. Upon obtaining inference results, MissConf conducts bug reproduction.

3 REPRODUCING THE CTBUGS

In this section, we describe the design of MissConf, the first automated tool to infer missing configuration options and reproduce CTBugs. We first introduce the overview of MissConf. After that, we introduce two main components of MissConf, *i.e.*, missing configuration detection and CTBug reproduction.

3.1 Overview of MissConf

Figure 3 shows the overview of MissConf, which requires three inputs: source code of the target software, the configuration file, and the bug report. MissConf consists of two main parts: firstly, identifying whether crucial configuration options are missing in the bug report, and secondly, pinpointing the missing configuration options to reproduce the bug.

Configuration omission inference. When MissConf encounters a bug report that cannot be immediately reproduced, it begins with a preliminary assessment to determine if crucial configuration options are missing in the report. Based on the insights presented in Section 2.2, a bug report will be considered to have missing crucial configuration options *if some reported configuration options specified by users are not evident in the program's execution path after following the reproduction steps.*

To achieve this, MissConf employs taint analysis and targeted instrumentation techniques to identify which reported configuration options appear in the program's execution path. However, inferring missing options remains challenging, as users might specify irrelevant options in the bug report. To address this challenge, MissConf utilizes Large Language Models (LLMs) to comprehend the semantics of the report description and the reproduction steps. It then reasons about the relationship between the reported configuration options and the bug report, and infers potential configuration omissions.

CTBugs reproduction. Drawing on the insights from Section 2.2, MissConf utilizes the results of instrumented software execution to identify potential crucial configuration options, whose affected branch statements intersect with the program execution path. Then, it searches possible values for these configuration options and executes the reproduction steps to ascertain if the bug is reproduced. A major challenge in this task is the huge search space. To address this challenge, MissConf leverages the findings in Section 2.3 and introduce heuristic strategies to guide configuration prioritization and value sampling.

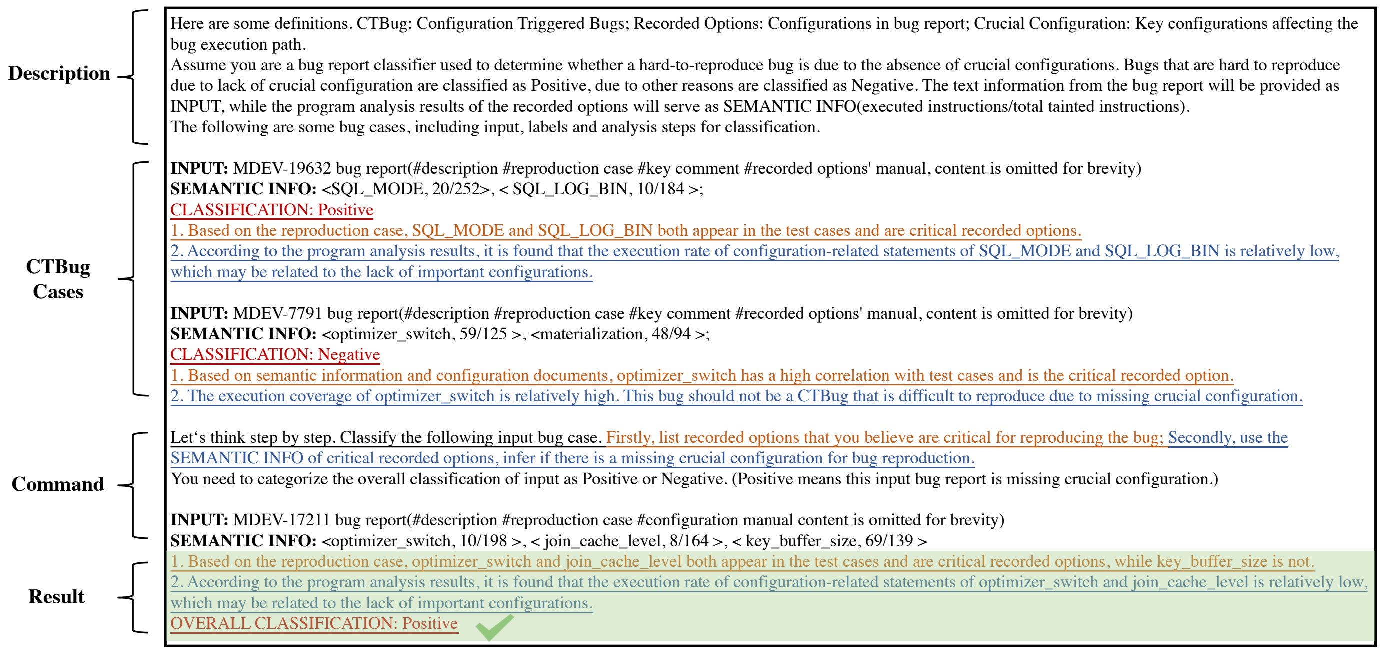
3.2 Configurations Omission Inference

Software systems have a large number of configuration options with a wide range of possible values. Blindly attributing the inability to reproduce a bug to missing configuration options can consume vast amounts of time and resources, considering the expansive configuration space. Therefore, it is essential to conduct a preliminary inference on bug reports to identify whether non-reproducibility stems from missing crucial configuration options. MissConf first leverages taint analysis and targeted instrumentation methods to pinpoint reported configuration options present in the program's execution path. Then, MissConf utilizes Large Language Models to infer potential configuration omissions.

3.2.1 Taint Analysis and Instrumentation.

Configuration Analysis. Taint analysis tracks some selected data of interest as entry points and propagates them along program execution paths according to a customized policy. Recent work [42] has investigated the propagation policy of configuration options and implementing ConfTainter as a taint analysis infrastructure for configurations. Therefore, we implement a taint analysis prototype on top of ConfTainter [42] to track the propagation of configurations. In this phase, the entry points are the original variables of the reported configuration options. The outcome of the sub-component is a collection of tainted LLVM instructions in Intermediate Representation (IR), featuring tainted variables as instruction operands.

Targeted Instrumentation. MissConf performs instrumentation at the IR level and employs a byte map on shared memory to log whether tainted instructions are executed. Specifically, MissConf iterates through the tainted instructions set for each reported option, inserting several IR instructions before each tainted instruction to manipulate the byte map. Subsequently, we initialize the reported option to the values specified in the bug report and execute the CTBug's reproduction steps within the instrumented program. We then record the tainted instructions traversed for each reported option during this execution.


Description
CTBug Cases
Command
Result

Here are some definitions. CTBug: Configuration Triggered Bugs; Recorded Options: Configurations in bug report; Crucial Configuration: Key configurations affecting the bug execution path.
Assume you are a bug report classifier used to determine whether a hard-to-reproduce bug is due to the absence of crucial configurations. Bugs that are hard to reproduce due to lack of crucial configuration are classified as Positive, due to other reasons are classified as Negative. The text information from the bug report will be provided as INPUT, while the program analysis results of the recorded options will serve as SEMANTIC INFO(executed instructions/total tainted instructions).
The following are some bug cases, including input, labels and analysis steps for classification.

INPUT: MDEV-19632 bug report(#description #reproduction case #key comment #recorded options' manual, content is omitted for brevity)
SEMANTIC INFO: <SQL_MODE, 20/252>, < SQL_LOG_BIN, 10/184 >;
CLASSIFICATION: Positive
1. Based on the reproduction case, SQL_MODE and SQL_LOG_BIN both appear in the test cases and are critical recorded options.
2. According to the program analysis results, it is found that the execution rate of configuration-related statements of SQL_MODE and SQL_LOG_BIN is relatively low, which may be related to the lack of important configurations.

INPUT: MDEV-7791 bug report(#description #reproduction case #key comment #recorded options' manual, content is omitted for brevity)
SEMANTIC INFO: <optimizer_switch, 59/125 >, <materialization, 48/94 >;
CLASSIFICATION: Negative
1. Based on semantic information and configuration documents, optimizer_switch has a high correlation with test cases and is the critical recorded option.
2. The execution coverage of optimizer_switch is relatively high. This bug should not be a CTBug that is difficult to reproduce due to missing crucial configuration.

Let's think step by step. Classify the following input bug case. Firstly, list recorded options that you believe are critical for reproducing the bug; Secondly, use the SEMANTIC INFO of critical recorded options, infer if there is a missing crucial configuration for bug reproduction.
You need to categorize the overall classification of input as Positive or Negative. (Positive means this input bug report is missing crucial configuration.)

INPUT: MDEV-17211 bug report(#description #reproduction case #configuration manual content is omitted for brevity)
SEMANTIC INFO: <optimizer_switch, 10/198 >, < join_cache_level, 8/164 >, < key_buffer_size, 69/139 >
1. Based on the reproduction case, optimizer_switch and join_cache_level both appear in the test cases and are critical recorded options, while key_buffer_size is not.
2. According to the program analysis results, it is found that the execution rate of configuration-related statements of optimizer_switch and join_cache_level is relatively low, which may be related to the lack of important configurations.
OVERALL CLASSIFICATION: Positive ✓

Figure 4: Example prompt of inferring missing configurations. MissConf first uses the bug description, reproduction steps, and configuration manual to deduce whether the reported configuration options are crucial for bug reproduction. Subsequently, MissConf uses the information of reported configuration options in program execution to infer if there is a missing crucial option. The LLM will give an overall classification conclusion. The above case utilizes two-shot learning for LLM, but in reality, MissConf employs four-shot learning.

3.2.2 *LLM-Based bug inferencer.*

Large language models (LLMs) are models that are trained using self-teaching algorithms on large unlabeled corpora. LLMs deliver substantial enhancements in their performance across natural language processing (NLP) tasks. MissConf uses the text processing and inferential learning capabilities of LLM to discern the relationship between the reported configuration options and the bug report. Then, MissConf can utilize LLM to further infer whether crucial configuration options are missing in the bug report, using the runtime information of the reported configuration options.

Prompt Engineering [39, 45] is the process of devising the most effective queries for specific tasks in LLMs. Previous research [15, 28, 55] indicates that embedding question-answer examples within a prompt enhances the performance of LLMs. The prompt construction of MissConf is primarily informed by the Chain-of-Thought(COT) method [45], which is widely acknowledged as the state-of-the-art prompting technique. MissConf proposes the approach of *progressive-COT* in this phase. Specifically, MissConf first uses the bug description, reproduction steps, and configuration manual to deduce whether the reported configuration options are crucial for bug reproduction. Subsequently, using the information of reported configuration options in program execution, MissConf infers if there is a missing crucial option for bug reproduction.

The prompt is structured as follows: MissConf employs four-shot prompt learning, including two positive and two negative input examples. If sufficient bug report examples are lacking, MissConf can still leverage zero-shot learning, given that LLMs have been demonstrated to possess capabilities in both zero-shot and few-shot learning scenarios.

Figure 4 illustrates examples of MissConf's prompts. When working with a test example, the LLM will adhere to the demonstration format, performing step-by-step reasoning according to the prompt requirements. First, it will list the crucial configuration options in the reported option set, then determine whether a crucial configuration is missing in the report based on the program analysis results, and finally give the final classification conclusion.

3.3 CTBug Reproduction

After MissConf infers that crucial configuration options are missing in the bug report, it attempts to search for the missing ones and their values. Based on Finding 3&4, MissConf can adjust just one configuration option and then execute the reproduction steps to reproduce the bug. This process, however, is non-trivial. Firstly, a software system might have hundreds of configuration options, making it impractical and time-consuming to experiment with each option individually. MissConf should filter out the suspicious option set to reduce the time for bug reproduction. Secondly, the order in which suspicious options are tried can also influence the efficiency of bug reproduction. MissConf should prioritize suspicious options to swiftly identify the missing ones. Thirdly, for each suspicious option, MissConf will sample potential values that could trigger the bug, which may have a vast array of possible values.

Filtering Out Suspicious Options. In mature software projects, the number of configuration options can be huge, making exhaustive enumeration of potential crucial configuration options for unreproducible CTBugs impractical. MissConf addresses this challenge by filtering out the suspicious options that might directly influence the CTBug's reproduction. As detailed in Section 2.2, an effective strategy to identify these suspicious options is by monitoring and recording the set of options that affect the control-flow execution path of the bug.

To achieve this, MissConf employs similar taint analysis and targeted instrumentation techniques detailed in Section 3.2.1. Based on finding 2, MissConf can focus solely on configuration options of enumeration and numerical types. The entry points for taint analysis in this phase are the original variables of these two configuration types. Then, MissConf instruments before the tainted branch instructions of these configuration options. Similarly, we set the reported configuration options based on the values provided in the bug report and run the CTBug's reproduction steps within the new instrumented software. We subsequently record the option traversed during this execution and label them as suspicious options.

Ranking Suspicious Options. While MissConf filters out suspicious options to reduce the configuration search space, reproducing bugs remains a time-intensive task. To mitigate this, MissConf employs heuristic strategies to prioritize suspicious options. The rationale behind this prioritization is derived from the insights gained from our study in Section B on how configurations influence CTBug execution. We discovered that reported configuration options are often crucial for reproducing bugs. The fewer reported configuration options that appear in a program's execution path, the more likely it is that the path diverges from where the bug occurs. Therefore, *when the execution path after a suspicious option contains fewer reported configuration options, there's a higher likelihood that the current suspicious option needs to be mutated.* Conversely, the more reported configuration options there are, the lower the priority for mutating the current suspicious option.

To achieve this target, MissConf instruments before the tainted branch instructions of suspicious options, capturing the timestamps when the program passes through these branches. Similarly, for the reported configuration options, MissConf instruments before their tainted instructions to record the timestamps of program execution through each reported option. Algorithm 1 illustrates the primary procedure for ranking suspicious configuration options. For any two suspicious options, MissConf first compares the number of reported configuration options that follow each option within the execution path (calculated by the *CNum* function). The option with a smaller number is given higher priority (Line 4-5). If the numbers are the same, we then compare their initial timestamps within the execution path. The option with an earlier timestamp is assigned a higher priority (Line 6-7).

Sampling Configuration Values. After identifying and ranking the suspicious configurations, MissConf proceeds to sample potential configuration values that might trigger the bug. Depending on the types of configurations, MissConf employs different strategies for value sampling.

Algorithm 1: Ranking suspicious configuration options

```
Input: SArray is the array of suspicious options.
Output: RArray is the array of ranked options.
        Configuration options with higher priority are
        positioned earlier in the array.
1 RArray ← SArray;
2 for i=0; i<RArray.length(); i++ do
3     for j=i+1; j<RArray.length(); j++ do
4         if RArray[i].CNum() > RArray[j].CNum() then
5             Swap(RArray[i], RArray[j]);
6         else if RArray[i].CNum()==RArray[j].CNum() ∧
            RArray[i].time > RArray[j].time then
7             Swap(RArray[i], RArray[j]);
8 return RArray
```

- *Enumerated Configurations.* The user manual typically outlines all permissible values for configurations of the enumerated type. For instance, MySQL provides 3 valid values for the option 'binlog_format'. Therefore, MissConf samples all the valid values mentioned in the manual for such configurations.
- *Numeric Configurations.* The user manual also specifies the valid value ranges for numeric configurations. However, the spectrum of permissible values for numerical configurations is broader than enumerated ones. For example, the option 'bulk_insert_buffer_size' has a valid range of $0 \sim 2^{32}-1$ bytes. Given the challenge of determining an appropriate sampling density, MissConf adopts the strategy of exponential sampling [43]. Test values for these numerical configurations commence at the minimal allowable value and escalate exponentially until they reach the upper limit. For the option 'bulk_insert_buffer_size' as an illustration, the test values would span $\{0, 1, 2, 4, 8, \dots, 2^{32} - 1\}$.

4 EVALUATION

We evaluated MissConf in terms of its ability to infer missing configurations and its effectiveness in reproducing CTBugs. Our evaluation aims to answer the following research questions:

RQ1: How effective is MissConf in inferring configuration omission? This question examines the precision and recall in distinguishing CTBug reports from other bug reports.

RQ2: How effective is MissConf in reproducing CTBugs? This question examines the recall of MissConf by calculating the percentage of bugs that can be reproduced among all CTBugs.

RQ3: How does MissConf perform in narrowing down the configuration search space? This question evaluates the efficiency by calculating the percentage of suspicious options filtered from all the options.

RQ4: How does MissConf perform in ranking suspicious configurations? This question evaluates the efficiency by determining the relative position of the missing crucial option in the sorted suspicious option set.

Target Software Systems. We chose MySQL and Squid as the target software as we found 23 hard-to-reproduce CTBugs in them during the empirical study. To avoid over-fitting, we also chose SQLite[19], Nginx[36] and Httpd[1], which were not included in the study. We repeated the CTBug collection and validation methods on the five systems, and found 26 CTBugs that were not included in the study in Section 2. Please note that not all collected CTBugs are hard-to-reproduce bugs, which only make up approximately 30% of the total. Therefore, we removed one crucial configuration option from the CTBug report and retained all other descriptions to simulate hard-to-reproduce CTBugs. We evaluated MissConf on all the 49 bugs.

Evaluation Setup. We perform the evaluation on a machine with 8 cores (Intel Core i7-9700K CPU @3.6GHz), 32 GiB RAM, and Ubuntu 20.04 as the operating system.

4.1 Effectiveness of inferring configuration omission

As mentioned in Section 3.2, the first step of MissConf is to perform an initial assessment of bug reports to determine if a hard-to-produce bug is caused by missing crucial configuration options. MissConf utilizes taint analysis and targeted instrumentation techniques to gather the runtime information of record options, and then utilize LLM to further infer whether crucial configuration options are missing based on bug report text and runtime information.

To evaluate the MissConf's ability to infer configuration omission, we first build a new dataset. The dataset includes not only CTBugs that are hard to reproduce due to missing crucial configuration options, but also bugs that are hard to reproduce for other reasons. As shown in Table 3, we have constructed a balanced dataset where the "CTBug report" column represents positive samples, and the "Other report" column represents negative samples.

We conducted experiments on ChatGPT 3.5 [10] using the prompts described in Section 3.2.2. These experiments encompassed three different settings: 0-shot, 2-shot, and 4-shot. The 0-shot setting means no prior knowledge was provided for the LLM. The 2-shot setting means one positive sample and one negative sample were given. The 4-shot setting means two positive samples and two negative samples were provided.

Table 3 demonstrates MissConf's capability to identify potential CTBugs. It can be observed that when using the 4-shot prompt, the average precision reaches 73%, and the recall reaches 84%. Furthermore, through a comparative analysis, it becomes evident that the LLM few shot learning is indeed necessary, as the recall of 4-shot has improved 53% compared to the 0-shot setting.

At the same time, we also acknowledge that there are cases where misclassification occurs. However, the primary objective of this step is to enhance efficiency and save time in the bug reproduction process. When we skip the initial screening of bug reports and directly proceed with subsequent reproduction efforts, we would need to perform taint analysis on all configuration options of the software (often numbering in the hundreds). Reproducing a bug under these conditions typically takes around 8 hours. If the attempted bug reproduction is not hampered by the absence of critical configurations, a significant amount of time would be wasted. In contrast, the initial screening step involves taint analysis of only a small number of reported configuration options and usually takes just a few minutes. Therefore, we consider the trade-off between bug reproduction efficiency and precision in this step to be acceptable.

Table 3: The precision and recall of inferring configuration omission.

Project	CTBug report	Other report	Setting	Precision	Recall
MySQL	20	20	0-shot	33%	5/20 (25%)
			2-shot	67%	14/20 (70%)
			4-shot	74%	17/20 (85%)
Squid	9	6	0-shot	50%	3/9 (33%)
			2-shot	86%	6/9 (67%)
			4-shot	80%	8/9 (89%)
SQLite	10	10	0-shot	44%	4/10 (40%)
			2-shot	55%	6/10 (60%)
			4-shot	73%	8/10 (80%)
Ngnix	4	4	0-shot	50%	1/4 (25%)
			2-shot	75%	3/4 (75%)
			4-shot	80%	4/4 (100%)
Httpd	6	6	0-shot	40%	2/6 (33%)
			2-shot	75%	3/6 (50%)
			4-shot	57%	4/6 (67%)
Total	49	46	0-shot	41%	15/49 (31%)
			2-shot	68%	32/49 (65%)
			4-shot	73%	41/49 (84%)

Answer to RQ1: This result indicates MissConf can effectively infer the configuration omission with the precision of 73% and recall of 84% (41/49).

4.2 Effectiveness of reproducing CTBugs

As shown in Table 4, the experimental dataset comprises 49 CTBugs from 5 software systems. Out of these, 41 CTBugs were correctly inferred, and MissConf successfully reproduced 32 CTBugs. So the overall bug reproduction success rate of MissConf is 65%. Specifically, when a potential CTBug report is correctly inferred, the bug reproduction success rate is 78%.

One significant reason for MissConf's reproduction failures is the incorrect inference during the initial screening, with a detailed analysis provided in Section 4.1. In cases where correct inference was made, there were 9 cases of MissConf's reproduction failures, primarily attributed to three main reasons.

Table 4: Effectiveness of reproducing CTBugs.

Project	Known CTBugs	Correctly Inferred Bugs	Reproduced Bugs
MySQL	20	17	14
Squid	9	8	6
SQLite	10	8	6
Nginx	4	4	3
Httpd	6	4	3
Total	49	41	32

First, the missing crucial configurations were unable to conduct taint analysis (4 cases). For example, in MySQL Bug #91980, the missing crucial configuration option is *autocommit*, which corresponds to a specific bit in program variable *option_bits*. To perform operations like assignment and retrieval on *autocommit*, it needs to use bitwise operations on *option_bits*. Currently, MissConf's taint analysis has not achieved the precision required for bit-sensitive analysis. Second, some bug reproduction requires specific workloads (2 cases). For example, reproducing a specific CTBug may require building a cluster environment. Third, the CTBug reproduction failed due to the requirement of on-the-fly changes (3 cases).

Answer to RQ2: This result indicates MissConf can effectively reproduce the CTBugs with the overall bug reproduction success rate of 65% (32/49).

4.3 Effectiveness of narrowing down the configuration search space

As mentioned in Section 3.3, MissConf utilizes taint analysis and targeted instrumentation techniques to filter out the suspicious options that could directly impact the CTBug's reproduction. We evaluate the effectiveness of narrowing down the configuration search space in this section. Each subfigure in Figure 5 represents MissConf's ability to filter suspicious configuration options for the corresponding software on the horizontal axis. The vertical axis in each subfigure represents the percentage of filtered suspicious configuration options compared to the total number of software configurations. A smaller ratio indicates that we have successfully filtered out more irrelevant configuration options. We utilize box plots to summarize and present the distribution of the data. It can be observed that, following the filtering process by MissConf, on average, we only need to traverse the 24% of the total configuration set to reproduce a CTBug effectively. In the best-case scenario, MySQL CTBug reproduction can be achieved by examining only 11% of the total configuration set. Considering that MySQL has thousands of configurations available for taint analysis, MissConf still achieves impressive results, demonstrating the effectiveness of taint analysis and targeted instrumentation techniques. For SQLite, MissConf can eliminate approximately 40% of irrelevant configurations. However, since SQLite has a limited number of configuration options available for taint analysis, only 16 in total, the overall number of configurations for subsequent reproduction is small, and it will not impact the subsequent reproduction phase.

Answer to RQ3: This result indicates MissConf can effectively narrow down the 76% configuration search space on average.

4.4 Effectiveness of ranking suspicious configurations

During the CTBug reproduction phase, MissConf employs heuristic strategies aimed at prioritizing suspicious options. One of our key contributions is the design of Algorithm 1, a sophisticated method for sorting elements within the suspicious configuration set, ensuring a streamlined and efficient prioritization process. To comprehensively evaluate the performance of the ranking algorithm, we conducted an extensive series of comparative experiments, directly comparing Algorithm 1 with a randomized ranking strategy. Figure 6 illustrates the effectiveness of MissConf in ranking suspicious configuration options compared to the random strategy. After employing Algorithm 1 as ranking strategy, the missing crucial option is typically located within the first 30.39% of the suspicious option set. When employing the random ranking strategy, MissConf on average needed to traverse the top 60.37% of the relevant configuration set before locating the missing crucial configuration options, which was 29.98% less efficient compared to using Algorithm 1.

Answer to RQ4: This result indicates the ranking suspicious configurations algorithm of MissConf improves the sorting of missing crucial configuration by 29.98% compared to a random strategy.

5 DISCUSSION

Impact of Taint Analysis. Taint analysis on large-scale systems may introduce enormous overhead. To achieve a lightweight configuration taint analysis, MissConf only tracks the propagation of

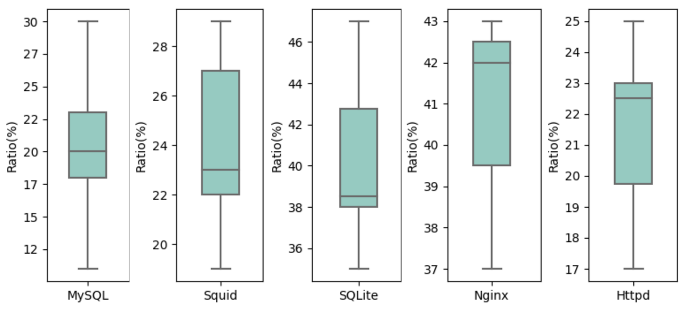


Figure 5: Ratio of filtered suspicious configuration options compared to the total number of software configurations.

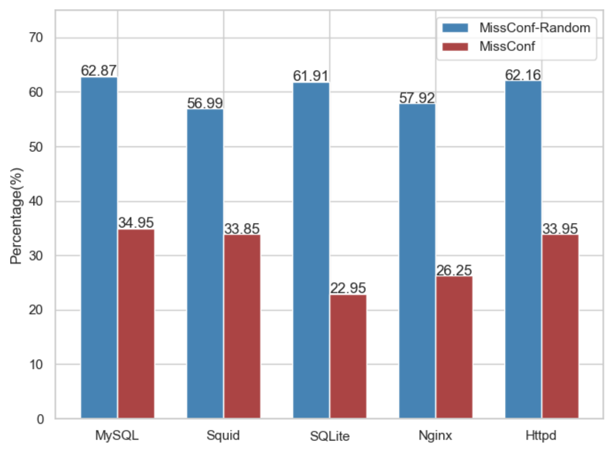


Figure 6: Effectiveness of configuration ranking algorithm.

variables that are tainted by configuration options through assignment operations and prunes call graph by excluding basic blocks that are not successors of tainted branches. Despite making these compromises on precision, the time required for taint analysis on a single option can be limited to a minute level. Further, taint analysis is deployed offline and needs to be run only one time for each different version of the program.

Noise of Irrelevant Options. The size of the obtained suspicious configuration option set determines the effectiveness of bug reproduction. Each time a test case is executed, MissConf monitors whether the instrumented branches are executed. This information reflects the option associated with that particular test case. However, in server-client systems like MySQL [2], the server runs in the background for extended periods, potentially continuously triggering the execution of instrumented branches, which can introduce noise into the data. To address this, MissConf conducts several dry runs, collecting the range of noise data before the tool's formal detection phase. This allows for precise identification of the relevant options during the actual detection process.

Bug Reproduction. CTBug occurs only when the specific option value is set. MissConf sets possible candidate values for *Enumerated* and *Numerical* options. However, some options are string-type variables and the values are associated with the workload. MissConf can't handle string-type options well. So we will explore how to detect string-type options and set their values to satisfy different workloads. Besides, some CTBugs require on-the-fly configuration modifications, and MissConf changes configurations before executing test cases to avoid the explosion of attempting changes at all possible times. Recent work by Wang et al. aims to detect on-the-fly bugs [43]. We will refer to the work and address the issue of reproducing on-the-fly CTBugs in our future work.

6 THREATS TO VALIDITY

Our study may suffer threats to the external and internal validity.

External Threats. The selection of software projects could potentially impact our empirical study. To mitigate this concern, all the projects included in our study are mature and widely adopted, each with a development and maintenance history spanning at least 15 years. Many CTBugs have already been identified and resolved by developers. However, the bug reporting process in mature software has been continuously optimized, and recent bug reports often contain full reproduction information. We believe that in newer and less mature software applications, there may be a higher occurrence of real cases where CTBugs cannot be reproduced.

Internal Threats. Another threat lies in the localization and manual validation process of CTBugs. We acquire bug reports through web scraping, automatically filter them using keywords, and manually confirm the relevance of these reports to CTBugs. We then manually analyze CTBug reports and reproduce the CTBugs, which could potentially introduce errors due to human mistakes. To enhance the accuracy of the validation process, three authors collaboratively examined all reported inconsistencies and reached a consensus on each of them.

7 RELATED WORKS

Bug Reproduction. Bug reproduction is a pivotal process in software development and quality assurance. By replicating reported bugs in a controlled environment, developers can analyze the root causes, leading to quicker and more accurate fixes. Significant research efforts have been dedicated to the field of bug reproduction. Some studies [23–26, 31, 35, 46, 51] employ in-house methods to gather a diverse range of failure data, encompassing function call sequences, crash stack, runtime logs, etc., thereby enabling the replication of the specific environment and inputs required for bug reproduction. BUGREDUX [26] aims to generate executions that replicate observed field failures by utilizing function execution data collected from real-world scenarios. CrashDroid [46] automates the process of reproducing a bug by translating the crash call stack into expressive steps. Some other works [16, 22, 27, 44, 52, 54] translate the procedural steps provided in bug reports into test cases. For instance, LIBRO [27] employs Large Language Models (LLMs) to automate the generation of tests based on bug reports. Furthermore, Hercules [37] employs symbolic execution techniques to generate test cases capable of triggering the reported software crashes. Nonetheless, these reproduction efforts operate under the assumption that the environment and configuration details supplied in the bug report are complete. As a result, they are unable to tackle the problem of absent crucial configurations, a common obstacle that often impedes the successful reproduction of CTBugs.

Missing Information Detection Many research efforts are dedicated to identifying and addressing missing information crucial for bug reproduction. Some studies utilize various methodologies, including machine learning, keyword matching, and heuristic strategies, to locate omitted essential components within bug reports, such as Expected Behavior (EB) and Steps to Reproduce (S2R) details in bug descriptions [14, 17, 20, 56]. While there exist other studies[13, 14, 38, 40, 50] focused on the detection of missing source code snippets or stack traces. RoBin [18] aims to figure out missing compiled-time configurations to reproduce bugs by analyzing the binary code. These approaches collectively contribute to addressing the challenge of incomplete information in bug reports, thereby enhancing the overall effectiveness of bug reproduction processes. However, restoring the environmental information required for bug reproduction is a fundamental and challenging task. The above works are also based on the assumption that the configuration environment information provided in the bug report is complete and cannot detect missing crucial configurations.

8 CONCLUSIONS

The absence of configuration information is one of the key reasons that make bugs challenging to reproduce. This paper focuses on addressing the issue of reproducing CTBugs. We first conducted an empirical study on 120 CTBugs from 4 open-source projects and summarized the characteristics that influence the CTBugs' occurrence and the reproducing factors of CTBugs. Based on these study findings, we designed and implemented MissConf, which represents the first LLM-enhanced automated solution for CTBug reproduction. MissConf first leverages LLMs to identify whether non-reproducibility stems from missing reported configuration options. Once the potential CTBug is identified, MissConf attempts to search for the missing configurations and their values by employing the heuristic reproduction strategy. The experimental results show that MissConf effectively infers the configuration omission and reproduces the CTBugs with an overall success rate of 65% (32/49). MissConf significantly expedites the process of reproducing CTbugs and consequently greatly saves the developer's time.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments. This research was funded by NSFC No. 62272473, the Science and Technology Innovation Program of Hunan Province (No. 2023RC1001) and NSFC No. 62202474.

REFERENCES

[1] 1995. Apache HTTP Server. https://httpd.apache.org/.
[2] 1995. MySQL. https://www.mysql.com/.
[3] 2002. Jira software. https://www.atlassian.com/software/jira.
[4] 2017. Assertion has_error == thd->get_stmt_da()->is_error()failed. https://bugs.mysql.com/bug.php?id=88273.
[5] 2018. Bug 91975: InnoDB Assertion failure. https://bugs.mysql.com/bug.php?id=91975.
[6] 2019. Replication aborts in ORACLE mode. https://jira.mariadb.org//browse/MDEV-19632.
[7] 2020. MDEV-19632 MariDB 10.5 Patch. https://github.com/MariaDB/server/commit/dd0485fcd795cf82f5e7675312c1755deca04f4f.
[8] 2021. Assertion failed in lock_rec_move. https://jira.mariadb.org//browse/MDEV-25010.
[9] 2023. Bugzilla keyword descriptions. https://bugzilla.mozilla.org/describekeywords.cgi/.
[10] 2023. ChatGPT. https://chat.openai.com/.
[11] 2023. GitHub. https://github.com/.
[12] 2023. Server System Variables. https://mariadb.com/kb/en/server-system-variables/.
[13] Alberto Bacchelli, Anthony Cleve, Michele Lanza, and Andrea Mocci. 2011. Extracting structured data from natural language documents with island parsing. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 476–479.
[14] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. 2008. Extracting structural information from bug reports. In *Proceedings of the 2008 international working conference on Mining software repositories*. 27–30.
[15] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[16] Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. 2019. Assessing the quality of the steps to reproduce in bug reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 86–96.
[17] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 396–407.
[18] Ligeng Chen, Jian Guo, Zhongling He, Dongliang Mu, and Bing Mao. 2021. Robin: Facilitating the reproduction of configuration-related vulnerability. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 91–98.
[19] D. Richard Hipp. 2000. SQLite. https://www.sqlite.org/.
[20] Steven Davies and Marc Roper. 2014. What's in a bug report?. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement(ESEM '14)*. Association for Computing Machinery.
[21] Raphael Pereira de Oliveira, Paulo Anselmo da Mota Silveira Neto, Qi Hong Chen, Eduardo Santana de Almeida, and Iftekhar Ahmed. 2022. Different, Really! A comparison of Highly-Configurable Systems and Single Systems. *Information and Software Technology* 152 (2022), 107035.
[22] Mattia Fazzini, Kevin Moran, Carlos Bernal-Cardenas, Tyler Wendland, Alessandro Orso, and Denys Poshyvanyk. 2022. Enhancing mobile app bug reporting via real-time understanding of reproduction steps. *IEEE Transactions on Software Engineering* 49, 3 (2022), 1246–1272.
[23] Steffen Herbold, Jens Grabowski, Stephan Waack, and Uwe Bünting. 2011. Improved bug reporting and reproduction through non-intrusive gui usage monitoring and automated replaying. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 232–241.
[24] David M Hilbert and David F Redmiles. 2000. Extracting usability information from user interface events. *ACM Computing Surveys (CSUR)* 32, 4 (2000), 384–421.
[25] Yan Hu, Jun Yan, and Kim-Kwang Raymond Choo. 2016. PEDAL: a dynamic analysis tool for efficient concurrency bug reproduction in big data environment. *Cluster Computing* 19 (2016), 153–166.
[26] Wei Jin and Alessandro Orso. 2012. Bugredux: Reproducing field failures for in-house debugging. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 474–484.
[27] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.
[28] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.
[29] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *Ieee software* 29, 6 (2012), 18–21.
[30] Wang Li, Zhouyang Jia, Shanshan Li, Yuanliang Zhang, Teng Wang, Erci Xu, Ji Wang, and Xiangke Liao. 2021. Challenges and opportunities: an in-depth empirical study on configuration error injection testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 478–490.
[31] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. *Acm Sigplan Notices* 40, 6 (2005), 15–26.
[32] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*. 919–936.
[33] CKJDSA Stefan Mühlbauer, Florian Sattler, and N Siegmund. 2023. Analyzing the impact of workloads on modeling the performance of configurable software systems. In *Proceedings of the International Conference on Software Engineering (ICSE), IEEE.*
[34] MySQL. 2018. int change_master(): Assertion inited failed. https://bugs.mysql.com/bug.php?id=92073.
[35] Mathieu Nayrolles, Abdelwahab Hamou-Lhadj, Sofiene Tahar, and Alf Larsson. 2015. JCHARMING: A bug reproduction approach using crash traces and directed model checking. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 101–110.
[36] Nginx, Inc. 2004. Nginx. https://nginx.org/.
[37] Van-Thuan Pham, Wei Boon Ng, Konstantin Rubinov, and Abhik Roychoudhury. 2015. Hercules: Reproducing crashes in real-world application binaries. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 891–901.
[38] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. 2015. Stormed: Stack overflow ready made data. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 474–477.
[39] Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–7.

[40] Peter C Rigby and Martin P Robillard. 2013. Discovering essential code elements in informal documentation. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 832–841.

[41] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 284–294.

[42] Teng Wang, Haochen He, Xiaodong Liu, Shanshan Li, Zhouyang Jia, Yu Jiang, Qing Liao, and Wang Li. [n. d.]. ConfTainter: Static Taint Analysis For Configuration Options. ([n. d.]).

[43] Teng Wang, Zhouyang Jia, Shanshan Li, Si Zheng, Yue Yu, Erci Xu, Shaoliang Peng, and Xiangke Liao. 2023. Understanding and detecting on-the-fly configuration bugs. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*.

[44] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. 2010. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*. 155–166.

[45] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[46] Martin White, Mario Linares-Vásquez, Peter Johnson, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Generating reproducible and replayable bug reports from android application crashes. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 48–59.

[47] Wikipedia. 2023. Data definition language. https://en.wikipedia.org/wiki/Data_definition_language.

[48] Wikipedia. 2023. Data manipulation language. https://en.wikipedia.org/wiki/Data_manipulation_language.

[49] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 818–834.

[50] Deheng Ye, Zhenchang Xing, Chee Yong Foo, Jing Li, and Nachiket Kapre. 2016. Learning to extract api mentions from informal natural language discussions. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 389–399.

[51] Tingting Yu, Tarannum S Zaman, and Chao Wang. 2017. DESCRY: reproducing system-level concurrency failures. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 694–704.

[52] Cristian Zamfir and George Candea. 2010. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*. 321–334.

[53] Yu Zhao, Kye Miller, Tingting Yu, Wei Zheng, and Minchao Pu. 2019. Automatically extracting bug reproducing steps from android bug reports. In *Reuse in the Big Data Era: 18th International Conference on Software and Systems Reuse, ICSR 2019, Cincinnati, OH, USA, June 26–28, 2019, Proceedings 18*. Springer, 100–111.

[54] Yu Zhao, Ting Su, Yang Liu, Wei Zheng, Xiaoxue Wu, Ramakanth Kavuluru, William GJ Halfond, and Tingting Yu. 2022. Recdroid+: Automated end-to-end crash reproduction from bug reports for android apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–33.

[55] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910* (2022).

[56] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. 2010. What makes a good bug report? *IEEE Transactions on Software Engineering(TSE)* 36, 5 (2010), 618–643.