WingFuzz: Implementing Continuous Fuzzing for DBMSs

Jie Liang, Zhiyong Wu, and Jingzhou Fu, *Tsinghua University;* Yiyuan Bai and Qiang Zhang, *Shuimu Yulin Technology Co., Ltd.;* Yu Jiang, *Tsinghua University*

https://www.usenix.org/conference/atc24/presentation/liang

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

WINGFUZZ: Implementing Continuous Fuzzing for DBMSs

Jie Liang
Tsinghua University

Zhiyong Wu[*]
Tsinghua University

Jingzhou Fu
Tsinghua University

Yiyuan Bai
Shuimu Yulin Technology Co., Ltd.

Qiang Zhang
Shuimu Yulin Technology Co., Ltd.

Yu Jiang[†]
Tsinghua University

Abstract

Database management systems (DBMSs) are critical components within software ecosystems, and their security and stability are paramount. In recent years, fuzzing has emerged as a prominent automated testing technique, effectively identifying vulnerabilities in various DBMSs. Nevertheless, many of these fuzzers require specific adaptation for a DBMS with a particular version. Employing these techniques to test enterprise-level DBMSs continuously poses challenges due to the diverse specifications of DBMSs and the code changes in their rapid version evolution.

In this paper, we present the industry practice of implementing continuous DBMS fuzzing on enterprise-level DBMSs like ClickHouse. We summarize three main obstacles in implementing, namely the diverse SQL grammar in test case generation, the ongoing evolution of codebase in continuous testing, and the disturbance of noises during anomaly analysis. We propose WINGFUZZ, which utilizes specification-based mutator generation, corpus-driven evolving code fuzzing, and noise-resilient anomaly assessment to address them. By working with the engineers in continuous DBMS fuzzing, we have found a total of 236 previously undiscovered bugs in 12 widely-used enterprise-level DBMSs including ClickHouse, DamengDB, and TenDB. Due to its favorable test results, our efforts received recognition and cooperation invitations from some DBMS vendors. For example, ClickHouse's CTO praised: "Which tool did you use to find this test case? We need to integrate it into our CI." and WINGFUZZ has been successfully integrated into its development process.

1 Introduction

Modern software applications rely heavily on database management systems (DBMSs) to efficiently store and manage data [4, 15]. Due to the complexity of DBMSs and the rapid iteration of versions, bugs are inevitably present. Once these bugs are exploited by attackers, it may lead to a series of problems such as information leakage, system crashes, and disruptions to the operation of upper-tier applications. Consequently, the resilience and security of DBMSs have become paramount considerations for organizations and corporations [1, 5, 9].

Both the industry and academia have developed many methods for detecting issues in DBMSs [24, 32, 42, 46]. Among them, fuzzing [33, 35, 63] is an effective way to assess the security of DBMSs [43, 54, 61], revealing lots of vulnerabilities. Its basic process is to generate a stream of malformed inputs and monitor the exception behaviors [28, 29, 31, 59]. To test a DBMS, fuzzers typically model the specific SQL specification [51] for each DBMS to generate structured SQL inputs that are both syntactically and semantically correct. For example, SQLsmith [43] models the grammar of PostgreSQL [39] as an abstract syntax tree (AST) to generate queries, and it has found about 80 bugs in PostgreSQL since 2015 to 2023 [10].

However, employing these techniques to implement continuous fuzzing for DBMSs poses obstacles, especially for iteratively developed enterprise-level database systems. The first obstacle is the *diverse SQL grammar*, posing difficulties in test case generation. Inputs to distinct DBMS typically adhere to unique grammar rules. For example, the syntax for concatenating strings is different in PostgreSQL and MySQL: PostgreSQL uses the '||' operator for string concatenation, whereas MySQL requires the `CONCAT()` function. Consequently, to thoroughly test a DBMS, many fuzzers have to be manually customized to unique grammars. For example, SQUIRREL [61] has two different versions for testing PostgreSQL and MySQL. For enterprise-level DBMSs, automated adaptation solutions are essential to accommodate various DBMSs and rapidly evolving DBMS versions.

The second obstacle is the *ongoing evolution of codebase*, which creates difficulties for continuous testing. To meet the ever-changing demands of users and emerging security challenges, along with rapidly advanced technologies, an enterprise-level DBMS undergoes frequent updates to improve performance, fix vulnerabilities, and introduce innovative functionalities. For example, ClickHouse uses a continu-

[*]Jie Liang and Zhiyong Wu contributed equally to this work.
[†]Yu Jiang is the corresponding author.

ous integration system to automatically verify and merge new commits into its codebase [3]. To adequately test a DBMS, fuzzers must keep pace with these changes in the evolving systems, yet efficiently testing the updated portion in a new version remains challenging.

The third obstacle is the *disturbance of noises*, introducing complexities in anomaly detection and analysis. First, the automatic recovery mechanisms introduce noise into the fuzzers' detection of anomalies, as they typically rely on checking whether the connection is normal. When an anomaly happens, the recovery mechanism takes control of the erroneous thread to restore normal state and data. The connection may not even be broken in the process, resulting in a miss of the anomaly. Moreover, many state-of-the-art fuzzers execute tests on a DBMS consecutively and thus tests have interferences. The noise will come from the execution of the prelude test cases because they will make changes to the system state.

In this paper, we present and analyze these obstacles in implementing continuous fuzzing on enterprise-level DBMSs. We propose WINGFUZZ which implements continuous fuzzing for DBMSs. It utilizes specification-based mutator construction, corpus-driven evolving code fuzzing, and noise-resilient anomaly assessment to address the obstacles. First, the framework generates a unique query parser for each DBMS automatically by adhering to the DBMS grammar specifications. Subsequently, it concurrently executes long-term fuzzing, utilizing generated test cases to initiate commit fuzzing which prioritizes the coverage in the updated commit. Lastly, the framework monitors each thread and isolates the anomaly thread to capture anomalies along with their comprehensive data. The identified anomalies are then de-duplicated and reported to developers for prompt resolution.

By working with the engineers in the fuzzing process, we have uncovered a total of 236 previously undiscovered bugs in 12 DBMSs such as ClickHouse [13], DamengDB [16], PolarDB [38], and TenDB [49]. Among them, 232 bugs are confirmed. Our efforts received recognition from the vendors of these DBMSs. For example, the CTO of ClickHouse praised: "Which tool did you use to find this test case? We need to integrate it into our CI." [14], and the continuous fuzzing of WINGFUZZ has been successfully integrated into the development process of ClickHouse. The developers of MonetDB praised: "They are the best kind we can wish for as bug reports.", and the founder of SQLite praised WINGFUZZ as a "ground-breaking analysis tool" [21]. In summary, we make the following contributions:

1. We summarize three obstacles in implementing continuous DBMS fuzzing: diverse SQL grammar, ongoing codebase evolution, and noise disturbance.

2. We propose and implement a continuous DBMS fuzzing framework called WINGFUZZ to provide corresponding solutions, consisting of specification-based mutator construction, corpus-driven evolving code fuzzing, and noise-resilient anomaly assessment.

3. We deployed WINGFUZZ to continuously fuzz 12 enterprise-level DBMSs like ClickHouse, and identified 236 previously unknown bugs, enhancing their security and earning recognition from the respective vendors.

2 Background

DBMS and SQL. A Database Management System (DBMS) is a software application that allows users to create, store, retrieve, and manage data [4, 15]. DBMSs utilize Structured Query Language (SQL) [8, 51] to define, manipulate, and query the database's data. A SQL input must adhere strictly to syntactic and semantic correctness. Any input that deviates from the specified format and correctness will be rejected.

Continuous Integration System. The continuous integration (CI) [17, 18, 45] is a software development practice that involves integrating every code change into a shared repository, automatically subjecting each modification to testing before committing. This practice guarantees that new code not only smoothly integrates with the existing codebase but also does so without introducing errors.

DBMS Recovery Mechanism. A DBMS can experience failures arising from software issues or hardware malfunctions. It is crucial for the DBMS in a production environment to remain operational even in the face of failure. DBMS recovery techniques [26, 53] play a vital role in recovering data after a system failure and ensuring the maintenance of the database's atomicity and persistence properties.

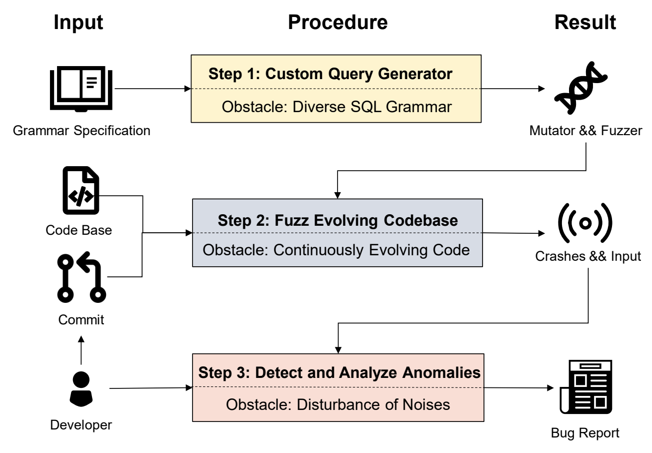
3 Obstacles in Continuous DBMS Fuzzing

While the characteristics of different DBMSs may vary, the basic process for implementing continuous fuzzing on them is similar. Figure 1 shows the three steps of the basic process, namely customize query generator, fuzz evolving codebase, and detect and analyze anomalies.

3.1 Step 1: Customize Query Generator

The first step is to customize the query generator for a unique DBMS. Query generation serves as the heart of DBMS fuzzing, as it determines the function and features that can be covered. A DBMS always possesses many features. To thoroughly test a DBMS, it is important to generate SQL queries covering these features because many potential bugs might exist in the related code. Moreover, it is important to ensure the generated queries exhibit both syntactic and semantic correctness. Any questionable input will be rejected, thus preventing exploration of the deeper logic.

Obstacle 1: diverse SQL grammar. Different DBMSs have unique syntax, semantics, and optimization strategies [2,



Input
Procedure
Result
Grammar Specification
Step 1: Custom Query Generator
Obstacle: Diverse SQL Grammar
Mutator && Fuzzer
Code Base
Step 2: Fuzz Evolving Codebase
Obstacle: Continuously Evolving Code
Crashes && Input
Commit
Developer
Step 3: Detect and Analyze Anomalies
Obstacle: Disturbance of Noises
Bug Report

Figure 1: Three steps and their related obstacles of implementing continuous fuzzing on an enterprise-level DBMS.

6, 55]. The diverse SQL grammar poses difficulties in the generation of SQL test cases. Tailoring the generator to the specifics of each DBMS ensures that the generated queries are compatible with the target system, enhancing the effectiveness of the testing process. This customization is essential for producing queries that are not only syntactically correct but also aligned with the unique language intricacies of the targeted database system to cover deep logic. For example, in Figure 2, the data types used in the column definitions are specific to each database system (e.g., `VARCHAR` in PostgreSQL vs. `String` in ClickHouse). Besides, ClickHouse uses the `ENGINE` clause to define the storage engine [2], whereas PostgreSQL does not employ comparable grammar.

```
-- PostgreSQL SQL Syntax
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    name VARCHAR(50)
);

-- ClickHouse SQL Syntax
CREATE TABLE employees (
    employee_id UInt32,
    first_name String
) ENGINE = MergeTree ORDER BY employee_id;
```

Figure 2: PostgreSQL and ClickHouse have different SQL formats in creating tables, which are shadowed in yellow and green, respectively.

3.2 Step 2: Fuzz Evolving Codebase

The second step is continuous fuzzing on evolving code. The code of an enterprise-level DBMS continuously evolves to meet the ever-changing business requirements, leverage technological advancements, address security concerns, enhance usability, and meet evolving compliance requirements. To ensure stability, security, and efficiency, the committed code changes often undergo rigorous testing and validation.

Obstacle 2: ongoing evolution of the codebase. The ongoing evolution of the codebase creates difficulties for continuous fuzzing. To fuzz evolving changed code, fuzzing needs to be conducted swiftly for a code commit while simultaneously retaining the capability to thoroughly test the entire codebase. And it is challenging to efficiently test the updated portions in a new version. First, when applied to a specific code submission, the challenge lies in isolating and evaluating the security impact of that submission within the broader context of the application. Understanding the exact triggers, inputs, and data paths affected by the new code can be intricate, especially in complex software ecosystems with numerous dependencies.

Moreover, the need for speed in software development adds complexity to this obstacle. In practice, an enterprise-level DBMS always employs a CI system to integrate code commits. It requires rapid security validation to ensure that code submissions do not introduce vulnerabilities. However, the exhaustive nature of traditional fuzzing can hinder the agility of development teams. Performing thorough fuzzing can be time-consuming, given that it often demands significant execution time to explore a broad spectrum of potential inputs.

3.3 Step 3: Detect and Analyze Anomalies

The final step involves the detection and analysis of bugs against interruption, encompassing anomaly recognition, de-duplication of anomalies, and their minimization. Anomalies may occur during the query execution. To detect them, a fuzzer needs to actively monitor the system to identify any deviations from the expected or desired behavior. After that, the anomalies need to be de-duplicated to unique ones. To pinpoint the root causes of anomalies, the inputs that trigger them and the site information of the anomaly (e.g., data, metadata, stack trace) will be saved and analyzed.

Obstacle 3: disturbance of noises. The presence of noise during anomalous events can impact both anomaly detection and analysis. The noise in detection mainly comes from the built-in anomaly recovery mechanism. Fuzzers typically determine anomalies by assessing the availability of connections to the DBMS, but the recovery mechanism will ensure the DBMS maintains its connection. For example, the intricate post-crash procedures in ClickHouse, which include logging queries, dump generation, and error checking, contribute to delays in promptly obtaining crucial crash site information. These processes, while essential for system recovery and ensuring data integrity, hinder the swift identification and resolution of anomalies. In the process, ClickHouse maintains its connections with the fuzzer and can even continue accepting generated queries. Consequently, a fuzzer encounters challenges in determining whether the DBMS has crashed.

Secondly, the recovery mechanism will also bring noises

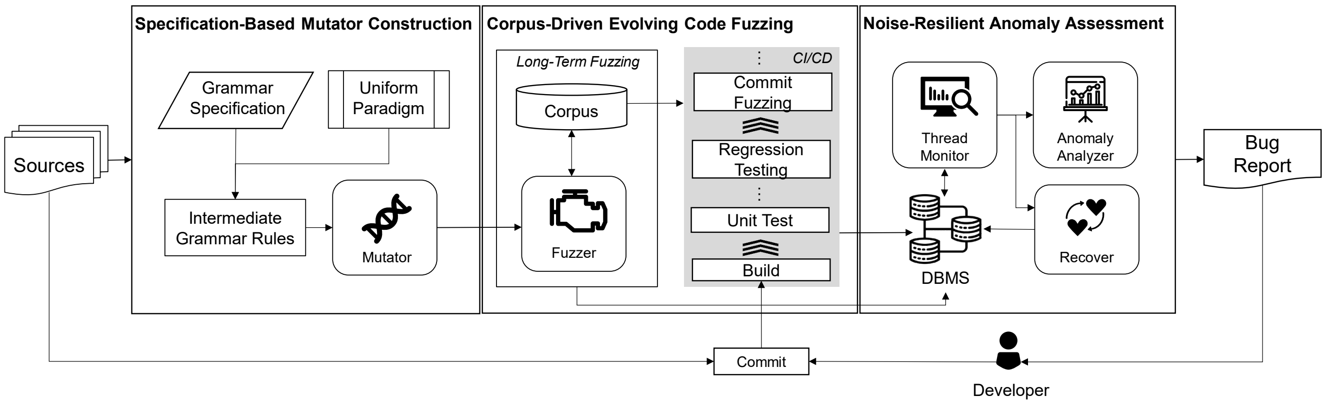


Figure 3: Overview of WINGFUZZ's solutions. (1) To automatically adapt to the diverse grammar, WINGFUZZ derives the query mutator from DBMS grammar specification. (2) To continuously fuzz evolving code, WINGFUZZ utilizes corpus-driven evolving code fuzzing. It continuously fuzzes the latest DBMS version, accumulating corpus to conduct rapid commit fuzzing on code that has undergone version changes. (3) To detect anomalies against noises, WINGFUZZ directly traces each thread to capture exceptional signals. Once an anomaly is captured, it will collect comprehensive information to generate a detailed bug report. After that, it terminates and restarts the entire system, resetting the database for the subsequent fuzzing cycle.

for anomaly analysis. Specifically, the mechanism might interrupt the problematic thread, blending the call stack of the anomaly with the call stacks of multiple threads. This introduces noise to the stack trace used for exception analysis. In addition, the noise from the previous query execution will also influence the analysis. Many state-of-the-art fuzzers execute tests on DBMSs consecutively, leading to interference between tests. Once an anomaly is triggered, it may not only be caused by the current test case but also related to the state change of the previous test cases.

4 Solutions of WINGFUZZ

Figure 3 shows the overview of WINGFUZZ. (1) In Step 1, to overcome the diverse SQL grammar, WINGFUZZ constructs the query mutator from the grammar specification of the DBMS. (2) In Step 2, to address the ongoing evolution codebase, WINGFUZZ continuously fuzzes the latest version, accumulating corpus to perform rapid commit fuzzing on code affected by version changes. (3) In Step 3, to insulate the noises, WINGFUZZ isolates the execution of test cases and directly captures anomalous signals.

4.1 Specification-Based Mutator Construction

WINGFUZZ addresses Obstacle 1 by *automatically deriving the customized mutator from the DBMS grammar specification.* It generates new queries by mutating existing queries while preserving syntax structures. The SQL mutator comprises two components: a SQL transfer and an AST mutator. The SQL transfer converts a SQL query into an AST. Subsequently, the AST mutator alters the structure (e.g., add a "`ORDER BY`" after a `SELECT` statement) or data elements (e.g., replace a table with a `SELECT` clause) of the original AST to generate a new one. Then the new AST is transformed into a SQL query by analyzing the dependencies between objects and popular data. The SQL mutator identifies the structures and data of a query through intermediate grammar rules, which are represented by a uniform paradigm.

Uniform Grammar Paradigm. This paradigm is an extension of the *BNF* paradigm with semantic information. It follows a context-free grammar, incorporating data elements that describe the SQL data model of a DBMS. Data elements are labeled with symbols that distinguish categories (e.g., tables and columns). Figure 4 shows an example of utilizing the paradigm to describe rules for `altertablestmt` recursive reduction. Specifically, `altertablestmt` is a non-terminal symbol, followed by its corresponding reduction rule. In the last row, `name` (which represents the `TABLESPACE` name) is marked with *@DataTName*. Utilizing the uniform grammar paradigm, various grammar rules can be uniformed, allowing for the construction of a customized SQL mutator.

```
altertablestmt:
ALTER TABLE relation_expr alter_table_cmds
| ALTER TABLE IF_P EXISTS relation_expr alter_table_cmds
| ALTER TABLE relation_expr partition_cmd
| ALTER TABLE IF_P EXISTS relation_expr partition_cmd
| ALTER TABLE ALL IN_P  TABLESPACE name
@DataTName@ SET TABLESPACE name @DataTName@ opt_nowait
```

Figure 4: An example of the uniform grammar paradigm.

Customized Mutator Construction. Algorithm 1 shows the procedure to automatically construct the SQL mutator with the SQL grammar specification. Firstly, it extracts tokens and SQL grammar rules (e.g., keywords and specific rules) from the SQL grammar file, then translates them into intermediate grammar rules using the uniform grammar paradigm (Lines 1-3). Then, for each grammar rule in the intermediate grammar rules, we analyze the rule to find the data-related elements that represent the data model (e.g., table, column, and schema) in DBMSs. We will add the specific mutation rules for each data element (e.g., a table object can only be populated with the table name), which are used to mutate the data element of AST (Lines 5-7). Finally, the tokens and rules will be used to generate a lexer for the SQL transfer (Lines 9-10), and the uniform grammar paradigm G' will be used to generate the AST mutator (Line 11).

Algorithm 1: Customized Mutator Construction

```
   Input  : SQL Grammar File: $G$
   Output : SQL Transfer: $T$, SQL Mutator: $M$
1  $Tokens \leftarrow$ extractTokens ($G$);
2  $Rules \leftarrow$ extractRules ($G$) ;
3  $G' \leftarrow$ constructNewParadigm ($Tokens, Rules$);
4  for $R \in Rules$ do
5      if containDataElement ($R$) then
6          addSQLDataMutationModel ($R$);
7      end
8  end
9  $L \leftarrow$ generateLexer($Tokens$) ;
10 $T \leftarrow$ generateTransfer($L, Rules$) ;
11 $M \leftarrow$ generatorMutator($G'$);
```

4.2 Corpus-Driven Evolving Code Fuzzing

WINGFUZZ overcomes Obstacle 2 by *employing the corpus generated through long-term fuzzing to conduct rapid commit fuzzing.* Figure 5 shows the basic process of evolving code fuzzing. In the long-term fuzzing, WINGFUZZ maintains a corpus containing the smallest inputs ever discovered in fuzz testing, which are capable of providing the broadest code coverage. When a commit is submitted to the codebase, it will carry out commit fuzzing. The process begins by extracting specific inputs related to the commit from the existing corpus determined by changed functions. Then a time-limited fuzzing is performed which is guided by coverage related to the new commit code.

Following the successful completion of these tests, the long-term fuzzing test is reinitiated to commence testing the latest code changes following the merged commit. Note that previously covered code can still be tested by this strategy. For long-term fuzzing, previous seeds are retained in the seed pool. If the changed code impacts the code that was previously covered, the effect on these codes can still be tested by the existing seed. The following are details of commit fuzzing.

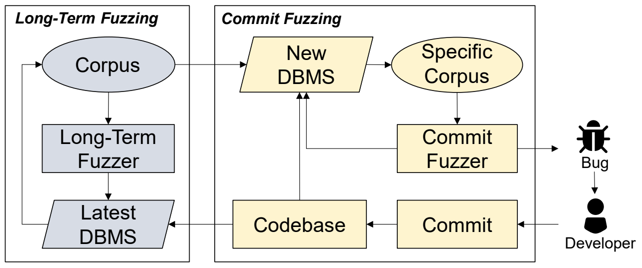


Figure 5: Corpus-driven evolving code fuzzing.

Commit Fuzzing. Commit fuzzing aims to rapidly ascertain whether the submitted commit contains any potential issues or vulnerabilities. Traditional fuzzing procedures often involve testing the entire system, which can be inefficient when trying to focus on a specific area of the codebase. This inefficiency can hinder the quick validation of the security of a commit. To address the problem, we propose Algorithm 2 to focus on fuzzing the commit code.

Algorithm 2: Commit Fuzzing

```
   Input  : Corpus in Long-Term Fuzzing: $C$,
            A map from a test case to its covered funs: $M$
            Codebase: $B$,
            Commit: $commit$
   Output : Specific Corpus: $C'$
1  $F \leftarrow$ getRelvantFuns($commit$);
2  $C'$= initialCorpus();
3  for $c \in C$ do
4      if $M[c] \cap F \neq \varnothing$ then
5          $C' \leftarrow C' \cup c$;
6      end
7  end
8  $D \leftarrow$ instrumentCompileCommit($B, F$);
9  $gbitmap \leftarrow$ initial();
10 repeat
11     $s \leftarrow$ select($C'$);
12     $s' \leftarrow$ ASTMutate($s$);
13     $(sbitmap, scbitmap) \leftarrow$ execute($D, s'$);
14     if canCover($scbitmap$) && hasNew($sbitmap$,
         $gbitmap$) then
15         $C' \leftarrow C' \cup s'$;
16         $M[s'] \leftarrow$ getCoveredFuns($s'$);
17         $gbitmap \leftarrow$ update($gbitmap, sbitmap$);
18     end
19 until $Timeframe\ Expiration$ || $Anomaly\ Triggered$;
```

The algorithm first filters a specific corpus that could cover the commit changes. To speed up corpus filtering, instead of re-executing existing test cases for coverage checking, we utilize static analysis to obtain interesting test cases that cover relevant code for new commits. First, we analyze the commit and get the relevant functions of the committed changes.

Specifically, the smallest functions that either contain the committed changes or invoke the new functions will be deemed relevant functions to the commit (Line 1). After that, we traverse each test case in the corpus and filter the ones that could cover the relevant functions (Lines 3-7). More precisely, we maintain a map from a test case to its covered functions. Any test case with covered functions overlapping commit-relevant functions will be added to the specific corpus.

Following the filtration of the specific corpus, WINGFUZZ initiates the fuzzing process using it as the initial corpus. We compile the codebase with the commit to instrument the DBMS that specially tracks the coverage of the committed changes (Line 8). In addition to utilizing the shared coverage bitmap in traditional fuzzing, we designed a dedicated standalone coverage bitmap specifically for committed changes. The commit fuzzing process is similar to coverage-guided fuzzing, including the process of test case selection (Line 11), mutation (Line 12), and corpus update (Lines 13-18). In the mutation, WINGFUZZ uses the customer mutator introduced in Section 4.1. When updating the corpus, the test case that can both cover the committed code and has new coverage will be added to the corpus. Commit fuzzing is terminated either upon the expiration of the predefined timeframe [25] or the occurrence of an anomaly.

CI Integration. Commercial DBMSs always utilize CI systems to merge code commits. Corpus-driven evolving fuzzing can be seamlessly integrated into CI with the following steps:

(1) Incorporate commit fuzzing as a check step in the integration testing, like following stress testing. The identification of any anomalies during commit fuzzing will result in marking the check as failed. Details about the triggering exception are preserved, and the code merge process fails. Developers are then required to fix the code before proceeding with subsequent commits.

(2) Place long-term fuzzing on dedicated servers to run continuously. When a commit passes all the checks in CI, the long-term fuzzing initiates a fresh testing cycle to assess the updated code with the newly merged commit. During this restart, all previous test cases in the corpus and new test cases found in commit fuzzing will be dry-run to construct a new corpus to start the new cycle. If long-term fuzz testing discovers issues or vulnerabilities, it will trigger an automatic reporting mechanism. The mechanism doesn't halt the testing process; testing continues seamlessly alongside the reporting of any problems encountered. The inputs that trigger anomalies will also be added to the regression testing set.

4.3 Noise-Resilient Anomaly Assessment

WINGFUZZ addresses Obstacle 3 by *directly tracing each thread of a DBMS and dropping databases before executing each test case.* To avoid system noise when an anomaly is triggered, WINGFUZZ directly traces each thread and halts the entire system. Figure 6 shows the overall process. Specifically, WINGFUZZ monitors each thread in real time for anomaly detection. Once an exception signal is received from any thread, it is categorized as a DBMS anomaly. The monitor intercepts the signal and suspends the thread. Subsequently, the bug analyzer will extract the stack trace of the thread along with other comprehensive data, such as the shared bitmap for coverage, and relevant system variables. The detailed information serves as a valuable resource for subsequent analysis and debugging.

After that, the bug analyzer terminates the whole DBMS, bypassing its recovery mechanism including built-in error logging, checking, and other functionalities. Then, WINGFUZZ deletes all databases and restarts the DBMS to provide a clean environment for subsequent tests. The detailed anomaly information can be analyzed online or off-line. We can utilize the stack trace and other information to deduplicate the anomalies and generate concise reports elucidating the potential root causes of anomalies and affected components.

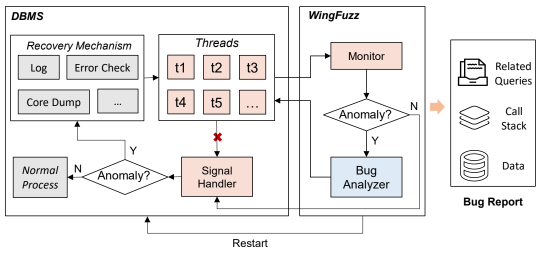


Figure 6: WINGFUZZ directly traces each thread for anomaly identification and halts the whole system for bug analysis.

Moreover, to avoid the noise from previous executions, WINGFUZZ adopts a strategy of isolating the execution of individual test cases. This involves the systematic dropping of all databases before initiating the execution of each test case. The approach eliminates residual data and side effects from previous tests, providing a fresh state for the precise execution of the current test case. Once a test case triggers an anomaly, it is convenient to use the current test case for bug reproduction and analysis.

5 Implementation

Based on the above solutions, we implement them into a continuous DBMS fuzzing framework named WINGFUZZ. It mainly consists of three parts: a query mutator constructor, an evolving code fuzzer, and a noise-resilient bug analyzer.

The query mutator constructor is based on the *bison* and *flex*. Generally, most DBMSs have a grammar description file like *bison* files in its source code, which describes the tokens and grammar rules. For example, the grammar file for PostgreSQL can be downloaded in its Github [7]. For DBMSs that lack the file, we can also extract their grammar description from the official document. The uniform grammar paradigm follows the *BNF* format with some SQL data elements. The mutator constructor first converts the tokens and grammar rules to *bison* and *flex* format following the uniform paradigm with 2034 lines of Python code. It then uses *bison* and *flex* to generate the SQL transfer and AST mutator. The fuzzing module encompasses the basic components for conducting fuzzing, including a wrapped compiler that encapsulates different parameters for the coverage instrument, a test case selector, a corpus updater, and the mutator constructed based on the grammar specification. The support for CI integration is implemented with scripts.

The noise-resilient bug analyzer monitors the exceptional signals of each thread to capture anomalies. The exceptional signals are operating system signals that can indicate various errors that need immediate attention, such as `SIGSEGV`, `SIGILL`, `SIGBUS`, `SIGABRT`, and `SIGFPE`. It could happen in any child process or child thread of the DBMS. To catch these anomaly signals, we start the DBMS as the child process of the monitor via fork() and exec(), and use `ptrace` to monitor the DBMS process: on the one hand, we trace all signals received by the DBMS process. If the signals are one of the anomaly signals, it means a potential bug is triggered, and we call the API of `libunwind-ptrace` to retrieve the call stack from the process. On the other hand, we trace the clone(), fork(), vfork(), and exec() of the process to catch all child processes and threads, monitoring them comprehensively.

6 Evaluation

In this section, we begin by comparing WINGFUZZ with other fuzzers. Subsequently, we demonstrate the contributions of each component in WINGFUZZ by gradually adding them. Moreover, we present the practice of deploying WINGFUZZ on 12 popular or enterprise-level DBMSs like ClickHouse.

6.1 Compared with Existing Fuzzers

To show the effectiveness of WINGFUZZ, we compared it with three state-of-the-art fuzzers, including conventional mutation-based fuzzer SQUIRREL as well as generated-based fuzzer SQLancer and SQLsmith, which are widely used in the industry to test DBMSs.

Tested DBMSs and Metrics. We chose four open-source DBMSs as tested DBMSs, namely PostgreSQL, MySQL, MariaDB, and PolarDB, which are widely used in industry and academic research. They were selected because all the compared fuzzers support testing them. Since other fuzzers do not support fuzzing evolving code, we used the latest version of four DBMSs as the target. We evaluated fuzzers using two metrics, namely branches covered and unique bugs triggered. The unique bugs were identified by comparing the call stack and manual analysis. For a fair comparison, when we finished fuzzing, we collected the queries generated by each fuzzer and dry-ran the queries to uniform the branch coverage.

Basic Setup. We performed all experiments on a machine running 64-bit Ubuntu 20.04 with 128 cores (AMD EPYC 7742 Processor @ 2.25 GHz) and 504 GiB of main memory. All DBMS test instances run in Docker containers, each containing a single DBMS service and a DBMS test tool. The test tools operate with default configurations, such as instrumentation methods and initial seed corpus. We run each DBMS test instance for 24 hours, a commonly used time frame for comparative evaluation. For quantitative comparisons, we run the docker containers for each DBMS experiment with 5 CPU cores and 32 GiB of main memory.

Branch Coverage. Table 1 shows the branches covered by those fuzzers in 24 hours. They show that WINGFUZZ covers a total of 211620, 197059, 132397 more branches than SQLancer, SQLsmith, and SQUIRREL, respectively.

Table 1: Number of branches covered in 24 hours.

DBMS	SQLancer	SQLsmith	SQUIRREL	WINGFUZZ
PostgreSQL	61,420	66,821	63,742	84,954
MySQL	69,182	67,291	77,829	104,281
MariaDB	60,031	69,281	91,932	129,832
PolarDB	56,928	58,729	93,281	140,114
Total	247,561	262,122	326,784	459,181
Increment	211,620	197,059	132,397	–

The main reason contributing to the enhanced coverage is that WINGFUZZ supports more SQL grammar of each DBMS than other fuzzers. Specifically, SQLancer generates queries based on custom pattern rules, limiting its support to the SQL grammar associated with the defined test oracle. Similarly, SQLsmith focuses on the AST model of the DBMS's `SELECT` statement, predominantly generating `SELECT` statements for testing. As a result, it lacks exploration of other DBMS functionalities. Benefiting from feedback guidance and the data dependency graph, SQUIRREL outperforms both SQLancer and SQLsmith in these DBMSs. Nevertheless, it still suffers from the limited support for the SQL grammar of the target DBMSs. For example, SQUIRREL currently supports only a subset of 165 grammar rules for MySQL, whereas the parser of MySQL encompasses 695 grammar rules.

In contrast, WINGFUZZ adopts a specification-based mutator construction approach to maximize support for the entire SQL grammar of the target DBMSs. The process begins by extracting the complete grammar rules and tokens from the specification of the target DBMSs, which are then translated into uniform intermediate grammar rules. Subsequently, WINGFUZZ generates its SQL grammar parser and SQL grammar lexer based on these uniform rules. In addition, WINGFUZZ also introduces the AST model of SQL queries for mutation, which could help maintain the syntax and semantic correctness of the SQL queries during mutation. More comprehensive support for SQL grammar allows WINGFUZZ to effectively trigger more complex logic within the target DBMS. For instance, WINGFUZZ accommodates all 695 SQL grammar rules of MySQL, ultimately leading to the discovery of 26452 more branches than SQUIRREL on MySQL.

Unique Bugs. Besides branch coverage, WINGFUZZ also triggered more bugs than other fuzzers in 24 hours. Table 2 shows the number of bugs found by each tool. Specifically, WINGFUZZ finds a total of 25, 24, 21 more unique bugs than SQLancer, SQLsmith, and SQUIRREL, respectively.

Table 2: Number of reproduced bugs triggered in 24 hours.

DBMS	SQLancer	SQLsmith	SQUIRREL	WINGFUZZ
PostgreSQL	0	1	1	2
MySQL	1	1	1	7
MariaDB	0	0	2	7
PolarDB	1	1	2	11
Total	2	3	6	27
Increment	25	24	21	–

The enhancement in bug triggering primarily stems from two factors. Firstly, there is an improvement in coverage. The increased branch coverage implies that WINGFUZZ can encompass unique functionalities within the target DBMS, where potential bugs might be located. Secondly, WINGFUZZ utilizes a noise-resilient anomaly assessment. Unlike other fuzzers that may ignore anomalies due to the noise brought by recovery mechanisms, WINGFUZZ directly monitors the exceptional signals from each thread of the target DBMS. Moreover, during the bug reproduction process, we observed that many bugs from other fuzzers cannot be triggered using their recorded inputs, due to the lingering execution state of previously executed test cases. In contrast, WINGFUZZ resets the database for each test case, ensuring that most of the anomalies detected by WINGFUZZ can be reproduced.

6.2 Contributions of Each Component

To evaluate the effectiveness of each component in WINGFUZZ, we implement WingFuzz^{raw}, WingFuzz^{g}, and WingFuzz^{g+e}. WingFuzz^{raw} uses the raw mutation algorithm in AFL [59] without grammar adaptation. WingFuzz^{g} enables specification-based mutator construction. Based on that, WingFuzz^{g+e} also enables evolve-code fuzzing. For a fair comparison, we first run commit fuzzing for 30 minutes and then enable normal fuzzing. WingFuzz^{all} (i.e., WINGFUZZ) enables all three components.

Table 3: Number of branches covered in 24 hours.

DBMS	WingFuzz^{raw}	WingFuzz^{g}	WingFuzz^{g+e}	WingFuzz^{all}
PostgreSQL	12,938	84,023	87,384	84,954
MySQL	11,082	103,940	105,402	104,281
MariaDB	12,393	130,294	131,394	129,832
PolarDB	20,112	139,739	141,593	140,114
Total	56,525	457,996	465,773	459,181

Table 4: Number of reproduced bugs triggered in 24 hours.

DBMS	WingFuzz^{raw}	WingFuzz^{g}	WingFuzz^{g+e}	WingFuzz^{all}
PostgreSQL	0	0	0	2
MySQL	0	2	2	7
MariaDB	0	2	2	7
PolarDB	0	1	4	11
Total	0	5	8	27

Table 3 and Table 4 demonstrate the number of branches covered and reproduced bugs found by these versions in 24 hours. With grammar-based mutation, WingFuzz^{g} finds 401471 more branches and 5 more bugs than WingFuzz^{raw}. The improvement contributes to the improved syntax and semantic correctness provided by grammar adaption. Combined with evolving code fuzzing, WingFuzz^{g+e} finds 7777 more branches and 3 more bugs than WingFuzz^{g}. This improvement is due to commit fuzzing effectively guiding the fuzzing process to cover changed code regions more promptly. Additionally, when noise-resilient anomaly assessment is also enabled, WingFuzz^{all} finds 6592 fewer branches because of the extra overhead. But it finds 19 more reproduced bugs than WingFuzz^{g+e}. The improvement in reproduced bugs is because the third component isolates the test case, thereby facilitating bug reproduction.

6.3 Practice of Deploying WINGFUZZ

In this section, we first illustrate the deployment process of WINGFUZZ, then present the bug detection results for 12 DBMSs we deployed along with some case studies.

6.3.1 Process of Deployment

The deployment process adheres to the procedure outlined in Section 3, encompassing three key steps: query generator customization, evolving codebase fuzzing, and anomaly detection and analysis. Let's use ClickHouse as an example to illustrate the deployment process.

Step 1: Customize query generator. In contrast to traditional row-based databases that store data in rows, ClickHouse organizes data by columns. Consequently, the SQL grammar of ClickHouse exhibits notable differences from that of traditional DBMS due to its optimization for columnar data storage. Nevertheless, within ClickHouse's source code, there exists a grammar file, which delineates the SQL grammar rules in the `bison` format. Utilizing this grammar file, WINGFUZZ initially extracts both tokens and context-independent grammar rules. Subsequently, the tokens are integrated into the `sqllex.l` file to generate the lexer. The grammar rules are reorganized with the IR structure and then incorporated into `parser-mutate.y` file to generate the grammar parser and AST mutator. The resulting lexer, parser, and AST mutator collectively contribute to SQL mutation.

Step 2: Fuzz evolving codebase. We first conducted long-term testing on the latest version of ClickHouse on a dedicated server. This process involved constant monitoring of ClickHouse's latest commits using scripts provided by WINGFUZZ. Throughout this procedure, a total of 10 bugs were identified. The reporting of these bugs garnered the attention of ClickHouse developers, including their CTO [14]. The developers promptly acknowledged and addressed these problems [50]. With the invitation from ClickHouse, we collaborated with their engineers to integrate WINGFUZZ into their development process. As described in Section 4.2, we integrated WINGFUZZ into the internal development workflow, continuously conducting long-term independent tests on the latest code. At the time of the paper writing, we have identified 31 bugs in ClickHouse. Two-thirds of them were found after integration, demonstrating the effectiveness of corpus-driven evolving code fuzzing. Apart from ClickHouse, we are collaborating with engineers of DBMSs such as DamengDB, YashanDB, and TenDB to deploy WINGFUZZ.

Step 3: Detect and analyze anomalies. ClickHouse has a robust error recovery mechanism. When a system anomaly occurs, ClickHouse logs error information, generates core dump files, checks errors, performs data recovery, and subsequently restarts the DBMS. This mechanism ensures smooth running, but the noise it introduces hinders the fuzzer's ability to capture and analyze exceptions. Furthermore, this mechanism incurs a considerable time cost, substantially impacting the fuzzing efficiency. Following the solutions in Section 4.3, WINGFUZZ directly monitors each thread. Upon triggering an anomaly in any thread, WINGFUZZ assumes control of the system. It extracts anomaly information like call stack from the thread. Following this, WINGFUZZ terminates all threads, clears the database, and restarts ClickHouse. In the anomaly analysis phase, WINGFUZZ de-duplicate anomalies based on coverage and call stack. Ultimately, anomaly reporting is accomplished by combining stack information with the input that triggered the exception.

6.3.2 DBMS Vulnerability Results

As Table 5 shows, we utilize WINGFUZZ to test 12 DBMSs (e.g., ClickHouse, DamengDB, and MariaDB), and WINGFUZZ reported a total of 236 bugs. We have reported all identified bugs to the corresponding DBMS vendors and have received positive responses from them. Among them, 232 bugs have been confirmed. Out of the identified bugs, about four-fifths stem from buffer overflow, segmentation violations, and use-after-free vulnerabilities. An attacker may use them to execute arbitrary code to control the system or gain special privileges, which could cause significant damage. Additionally, WINGFUZZ also found vulnerabilities such as null pointer dereferences, undefined behaviors, and assertion failures. These vulnerabilities can also inflict substantial damage on database services that must operate continuously.

Table 5: Number of anomalies reported by WINGFUZZ.

DBMS	Tested Versions	Reported	Confirmed
ClickHouse [13]	23.7.1-23.7.4	31	31
DamengDB [16]	8.1.1.87	60	60
MariaDB [34]	10.8-11.1	10	10
MonetDB [36]	11.46.0-11.48.0	12	12
MySQL [37]	8.0.32-8.1.0	11	9
PostgreSQL [39]	14.1-14.2	3	2
PolarDB [38]	2.0-2.3	21	20
SQLite [47]	3.37.3-3.39.0	17	17
TDengine [48]	2.4.0-2.6.0	24	24
TenDB [49]	3.3.1-3.3.2	18	18
VastBase-G100 [52]	2.2-build-11	23	23
YashanDB [57]	23.1.1.100	6	6
Total	-	236	232

More importantly, our efforts have garnered expressions of gratitude from numerous DBMS vendors. The acknowledgment and appreciation received from these vendors are represented in Figure 7, highlighting the widespread recognition and positive reception of our endeavors within the industry. This validation from DBMS vendors serves as a testament to the impact and value of our work.

"Wingtecher Lab's ground-breaking analysis tools that found a way to get one of those assert() statements to fail."[21]

Founder of SQLite

"Thank you very much for your reports. They are the best kind we can wish for as bug reports!"

COO of MonetDB

"We are very happy with the findings and we want more!"[50]

"Which tool did you use to find this test case? We need to integrate it into our CI."[14]

CTO of ClickHouse

Figure 7: Some affirmations from various DBMS vendors.

6.3.3 Case Studies

Case Study 1: *A segmentation violation in ClickHouse caused by ORDER BY tuple of window functions.* The bug is hidden in six C++ files in the *srcInterpreters* directory of ClickHouse's source code, which invokes the `ORDER BY` statements and `tuple` data types. Note that this bug is related to the unique grammar of ClickHouse.

```
CREATE TABLE v0 ( v1 Nullable ( Int32 ) , col2
String , col3 Int32 , col4 Int32 ) ENGINE =
MergeTree () ORDER BY tuple () ;
SELECT * FROM v0 ORDER BY tuple ( count ( * ) OVER
( ) , v1 , v1 , v0 . v1 ) ;
```

Figure 8: The case that triggered the segmentation violation.

Root Cause Analysis. Figure 8 shows the test case that triggered the bug. First, it creates a table *v*0 including 4 columns with the MergeTree engine. The data in table *v*0 are ordered according to the values with the `ORDER BY tuple()` subclause. Then, it queries the data from table *v*0 with `SELECT` statements and `ORDER BY` statements. Finally, the server crashed when it used `ORDER BY` to order the tuple of `count(*)` window functions with the `SELECT` statement. The bug is invoked by the combination of ClickHouse's `ORDER BY tuple()` and window functions, which is unique in ClickHouse. With the specification-based mutator construction, WingFuzz generates cases with unique grammars and detects this bug. In addition, when this anomaly occurs, system noise in ClickHouse can affect the identification and extraction of the bug's information. Due to WingFuzz's isolation of the execution, we can extract the triggering test case and related call stack for reproducing and analyzing.

Case Study 2: *A use-after-free in MariaDB caused by updated functionality of* `IN` *expression.* The bug was hidden deeply in the class `Item_func_in` of `item_cmpfunc.cc`. It poses significant damage as it can be exploited by an attacker to gain access to sensitive information belonging to other users. Note that the bug was introduced while upgrading the version to enhance the functionality of the `IN` expression.

```
CREATE TEMPORARY TABLE v0 ( v1 TINYBLOB , v2
TINYINT  , v3  BINARY GENERATED ALWAYS AS (  v1 IN
( FALSE , CURRENT_USER ( ) IS NULL  , 34 )  ) ) ;
ALTER TABLE v0 ADD COLUMN v0 MEDIUMINT;
INSERT IGNORE INTO v0 VALUES ( 'x' , 'x' , 'x',  1 ) ;
SELECT * FROM v0;
```

Figure 9: The test case that triggered the use-after-free.

Root Cause Analysis. The class `Item_func_in` is designed for `IN` expression, and the `in_vector *array` is a main member of it to store the temporary data of expression. In this case, the `array` was freed by mistake, which caused a use-after-free bug. Figure 9 shows the test case that triggered that bug. First, it creates a table *v*0 and associates attribute *v*1 with *v*3 with the `GENERATED` clause and `IN` expression. Meanwhile, the `array` in `Item_func_in` would be used to store the temporary data in the associated column. Then it executes the `ALTER` statement and the `INSERT` statement to change the structure of the table and insert the record into the table *v*0. However, the `cleanup()` function is called by mistake and the `array` is freed, which may still be used when operating the table *v*0. Finally, a use-after-free happens when MariaDB executes the `SELECT` statement to query data from *v*0.

The bug was introduced during the upgrade process of MariaDB's version, specifically in the enhancement of the `IN` expression functionality. However, the commit successfully passed through all checks, including unit testing and regression testing. WingFuzz promptly identified the bug through evolving code fuzzing when MariaDB released the new version. Its evolving code fuzzing enables the detection of such bugs triggered by version upgrades.

Case Study 3: *A buffer overflow in MonetDB caused by complex comparison expressions in queries.* When using the complex nested comparison expression with the `WITH` clause on a table in MonetDB, the optimizer of MonetDB triggers a buffer overflow while optimizing these comparisons.

```
CREATE TABLE v0 ( v1 SMALLINT ) ;
UPDATE v0 SET v1 = v1 <= ( WITH v0 ( v1 ) AS ( SELECT
( CASE WHEN 59 THEN ( 0 * ( ( 'x' < v1 = 255 > v1 -
v1 ) ) ) END ) ) SELECT v1 > 16 OR v1 > 2147483647
AND v1 >= 27 AS v4 FROM v0 ORDER BY v1 > v1 % v1 %
( v1 ) NULLS LAST ) OR v1 > -1 ;
```

Figure 10: The test case that triggered the buffer overflow.

Root Cause Analysis. Figure 10 shows the test case that triggered the bug. To optimize the complex expression in the `UPDATE` statement, MonetDB tries to rewrite the comparison by reconstructing the AST nodes. However, when meeting the nested comparisons, a wrong type of AST node was bound to the expression, which caused the buffer overflow in further processing. The bug is invoked by a complicated `UPDATE` statement. The issue was discovered because WingFuzz performs grammar adaptations to produce statements with nested comparisons, which MonetDB does not handle well.

Case Study 4: *An undefined behavior (integer overflow) in MonetDB when calling SQL function levenshtein().* When passing two large strings to the function `levenshtein()`, a piece of code in MonetDB that calculates the array length triggers an integer overflow. If execution continued, this led to array out-of-bounds access and then the DBMS crashed.

```
CREATE TABLE v0 ( v1 CHAR ( 100 ) );
INSERT INTO v0 VALUES ( 222 ) , ( 10 ) , ( 3 ) ,
( 947 ) ,…, ( NULL ) ,…, ( 34 ) ;
INSERT INTO v0 ( v1 ) SELECT group_concat ( 'table
tn3 row 99' ) FROM v0 , v0 AS tri , v0 AS OMW WHERE
10 LIMIT 4 ;
SELECT levenshtein ( v1 , v1 , 16 , 10 , 561 ) , v1 ,
v1 FROM v0 ;
```

Figure 11: The test case that triggered the undefined behavior.

Root Cause Analysis. Figure 11 shows the test case that triggered the bug. MonetDB uniquely supports the function `levenshtein()` to calculate the Damerau–Levenshtein distance between two strings. When the lengths of the two strings are m and n, MonetDB needs to allocate an array of length $m * n$ to perform the algorithm. However, the variable used to calculate the array length was stored as a 32-bit integer, which led to an integer overflow when the lengths of the two strings were large. MonetDB fixed the bug by changing the data type from int to long. The bug is invoked by a unique function supported by MonetDB. Because of grammar adaptation, WingFuzz could test the related grammar and cover the function. Due to the noise-resilient anomaly assessment, WingFuzz captured this undefined behavior and recorded the related information.

Case Study 5: *An assertion failure in MariaDB when inserting data into tables with spatial index.* When creating an InnoDB table with a SPATIAL index and inserting multiple rows of data, MariaDB threw an assertion failure '!cursor->index->is_committed()' and raise SIGABRT.

```
CREATE TABLE t1(f1 SERIAL, f2 LINESTRING NOT NULL
DEFAULT LineFromText('LINESTRING(1 1,2 2,3 3)'),
SPATIAL INDEX(f2))ENGINE=InnoDB;
INSERT INTO t1(f1) VALUES(0), (1), (2);
```

Figure 12: The test case that triggered the assertion failure.

Root Cause Analysis. Figure 12 shows the test case that triggered the bug. When an InnoDB table contains any index, the InnoDB engine will try bulk insertion when inserting multiple rows. The bulk insertion depends on the primary key which is automatically constructed by the index. However, the SPATIAL index is a special index that never constructs the primary key. Thus, when the bulk insertion was looking for the primary key in the SPATIAL index, it triggered the assertion failure. The bug is invoked by a combination of SPATIAL index and INSERT statement. It is triggered by the richer grammar brought with grammar adaptation. Due to the noise-resilient anomaly assessment, WingFuzz captured the signal thrown by the assertion failure to report it.

7 Lessons Learned

Adapt various DBMS features to improve semantic correctness and complexity of test cases. In practice, we find that many bugs are related to the specific dialects or features of the DBMSs. They are found because WingFuzz tries to adapt the grammar and features as much as possible. Even if new features are added in a new version, WingFuzz can also update its mutator to reflect changes in the grammar files. Handling these diverse grammar helps WingFuzz to improve the syntax and semantic correctness. Specifically, WingFuzz employs AST-based dependency analysis. It first identifies data nodes in the AST. Then it queries the database metadata to record the dependent data elements and maps them onto a data dependency table. During mutation, it selects the appropriate data elements from the data dependency table to populate the nodes to ensure semantic correctness.

Compared to WingFuzz^{raw} without grammar adaption, WingFuzz generates about 10x more semantic-correct test cases. Moreover, the queries generated by WingFuzz contain about 11 SQL statements on average, with one statement including 19 clauses. To further enhance the complexity required to expose bugs with more complex conditions, we can integrate other methods like statement sequence generation [27] into WingFuzz.

Integrate fuzzing into the DBMS development and accumulate corpus to accelerate testing. Incorporating fuzzing into the DBMS development helps to identify problems early and minimize damage. It is important to ensure that every code change undergoes automated fuzzing before deployment. The most significant outcome of DBMS fuzzing is the corpus of SQL inputs. This corpus serves as a valuable resource for building and maintaining a diverse set of test cases, enabling the fuzzer to continually evolve and adapt to changing codebases. Based on the corpus, WingFuzz utilizes commit fuzzing to accelerate the verification of updated code. We compare WingFuzz against WingFuzz^{-} that disables the commit fuzzing. With commit fuzzing, WingFuzz took 3.5 hours to reach the number of branches that WingFuzz^{-} in 24 hours, which accelerates the process by 6X respectively.

Isolate test case execution, and directly monitor the DBMS underlying status based on specific test oracles to identify vulnerabilities. Isolating test case execution by dropping databases may bring extra overhead, but it improves the functionality to reproduce bugs. For example, without dropping databases, WingFuzz executed 3104 more test cases and detected 1 more crashes in 24 hours on PolarDB. But only 4 crashes can be reproduced directly. The remaining bugs require additional human effort to analyze. In contrast, with dropping enabled, WingFuzz identified 11 bugs, all of which could be reproduced. Moreover, in a large system, indirect acquisition of anomaly information may be subject to various internal factors. Therefore, testing tools must be capable of directly intervening in the system's execution.

Our work focuses on the challenges in deploying continuous fuzzing for industrial DBMS, but there are also other difficulties in DBMS fuzzing. For example, designing test oracles for performance bugs is challenging because it's difficult to find a ground truth to measure the response time. Additionally, testing distributed DBMSs presents unique challenges. These systems introduce complexities related to synchronization, data consistency, and fault tolerance across multiple nodes due to their inherent nature, which are not widespread in monolithic database systems.

8 Related Work

8.1 Continuous Fuzzing

Continuous fuzzing is a security testing methodology that involves continuously and automatically testing software applications for vulnerabilities using fuzz testing techniques. Google's OSS-Fuzz [44], for instance, continuously tests more than 600 open-source projects with fuzzing and has found tens of thousands of bugs. Recently, continuous fuzzing has been integrated into the development pipeline in practice [23]. Klooster et al. [25] focus on optimizing fuzzing for CI/CD by examining ways to reduce unnecessary fuzzing efforts and developing prioritization strategies for allocating resources to fuzzing campaigns. CIDFuzz [60] is a fuzzing tool that addresses the common issue in CI of frequent code changes that existing methods may not effectively test. AFLGo [11] and Hawkeye [12] implement directed greybox fuzzing, which guides fuzzing to test specific code parts and can be used for patch testing. A study by Zhu and Böhme [62] analyzes bug reports from OSSFuzz [44] and reveals that 77% of bugs are due to recent code modifications. Therefore they present AFLChurn, which implements regression greybox fuzzing and prioritizes fuzzing efforts on code that has been changed more frequently or recently. Yoo et al. [58] focus on enhancing the configurability of continuous fuzzing at the unit level, specifically in the context of SAP HANA.

Similar to them, WINGFUZZ considers the issue in the complexity of the CI system and focuses on fuzzing committed changes. Differently, WINGFUZZ also preserves the long-term fuzzing to continuously test the latest version and provides the generated corpus as a valuable source for rapid commit testing. Additionally, WINGFUZZ considers the important DBMS characteristics, utilizing customized mutators and conducting noise-resilient anomaly assessments to ensure a thorough and accurate bug finding.

8.2 Generation Based DBMS Fuzzing

Generation-based fuzzers generate SQL test cases following the pre-defined generating rules. For example, SQLsmith [43] models SQL specifications into an AST model and generates amounts of queries. It checks bugs by monitoring if they cause disconnections to the server. SQLancer [40–42] designs three test oracles to detect logic errors and generates queries following the oracles. For example, its NOREC [40] oracle requires constructing a SQL query with `WHERE` and `JOIN` clauses. Then it translates the query to an equivalent one by moving the condition in the `WHERE` clause after the `SELECT`. MOZI [30] utilize configuration-based equivalent transformation to find logic and performance bugs. APOLLO [24] utilizes SQLsmith as the generator to construct SQL queries and perform regression testing between different DBMS versions to detect performance anomalies.

Different from them, WINGFUZZ is a mutation-based fuzzing framework. It leverages coverage to guide the query generation process. In contrast to the mentioned fuzzers, WINGFUZZ automatically adapts the grammar of a new DBMS by constructing a customized mutator from the DBMS's grammar file.

8.3 Mutation Based DBMS Fuzzing

Mutation-based fuzzers generate new queries based on the existing test cases guided by coverage. AFL [59], libFuzzer [31], and HONGGFUZZ [22] are used in OSS-Fuzz to test in-memory database system like SQLite. To produce more meaningful test cases, these fuzzers always collect SQL keywords into a dictionary to help the mutation. Nevertheless, their generated SQL test cases still have lots of syntax and semantic errors. The following works import SQL grammar modeling into the mutation process. SQUIRREL [61] designs an intermediate representation (IR) to represent the AST. It mutates queries based on the IR to analyze the dependencies between objects and keep their semantic correctness. RATEL [54] improves the coverage feedback precision by utilizing bijective block mapping. LEGO [27] finds type-affinities from existing test cases and utilizes affinities to synthesize cases with more SQL type sequences. UNICORN [56] designs a hybrid input synthesis to generate queries with time-series elements for time-series DBMSs. GRIFFIN [20] presents a grammar-free mutation strategy that reshuffles statements from existing queries with metadata-guided semantic fixing. To generate initial seeds for mutation, SEDAR [19] transfers test cases from other popular DBMSs with LLMs.

WINGFUZZ also utilizes mutation-based methods to generate queries. Different from these fuzzers which manually develop a mutator for each DBMS, WINGFUZZ customizes the mutator by deriving it from the grammar files. Moreover, WINGFUZZ isolates the execution environment of each test case, effectively minimizing noise interference between them. WINGFUZZ also directly captures exceptional signals from the target system, enhancing its ability to identify anomalies and gather relevant information for further analysis.

9 Conclusion

In this paper, we present the practice of implementing and deploying continuous fuzzing on enterprise-level DBMSs. In contrast to utilizing fuzzing on function libraries or utility programs, the process is more challenging. The diverse input grammar, ongoing evolving code, and system noises in a DBMS introduce difficulties in automatically adapting to diverse grammars, continuously fuzzing evolving code, and capturing and analyzing implicit anomalies. We analyze these obstacles and propose WINGFUZZ which utilizes specification-based mutator construction, corpus-driven evolving code fuzzing, and noise-resilient anomaly assessment to address them. We implement WINGFUZZ to test 12 enterprise-level DBMSs and found a total of 236 previously undiscovered bugs. Because of positive testing outcomes, we received praise from these vendors and WINGFUZZ has been incorporated into the development processes of ClickHouse.

Acknowledgements

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. 62302256, 92167101, 62021002), and China Postdoctoral Science Foundation (No. 2023M731953).

References

[1] Bugs found in database management systems. https://www.manuelrigger.at/dbms-bugs. Accessed: June 7, 2024.

[2] Clickhouse syntax. https://clickhouse.com/docs/en/sql-reference/syntax. Accessed: June 7, 2024.

[3] Continuous integration checks. https://clickhouse.com/docs/en/development/continuous-integration. Accessed: June 7, 2024.

[4] databases. https://en.wikipedia.org/wiki/Database. Accessed: June 7, 2024.

[5] Google chrome impacted by new magellan 2.0 vulnerabilities. https://www.zdnet.com/article/google-chrome-impacted-by-new-magellan-2-0-vulnerabilities/. Accessed: June 7, 2024.

[6] Postgresql 14beta3 documentation. https://www.postgresql.org/files/documentation/pdf/14/postgresql-14-A4.pdf. Accessed: June 7, 2024.

[7] Postgresql' grammar description file. https://github.com/postgres/postgres/blob/master/src/backend/parser/gram.y. Accessed: June 7, 2024.

[8] Sql commands. https://www.postgresql.org/docs/13/sql-commands.html. Accessed: June 7, 2024.

[9] Sqlite patches use-after-free bug that left apps open to code execution, denial-of-service exploits. https://portswigger.net/daily-swig/sqlite-patches-use-after-free-bug-that-left-apps-open-to-code-execution-denial-of-service-exploits. Accessed: June 7, 2024.

[10] Sqlsmith score list. https://github.com/anse1/sqlsmith/wiki#score-list. Accessed: June 7, 2024.

[11] Böhme, M., Pham, V.-T., Nguyen, M.-D., and Roychoudhury, A. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)* (2017).

[12] Chen, H., Xue, Y., Li, Y., Chen, B., Xie, X., Wu, X., and Liu, Y. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security* (2018), pp. 2095–2108.

[13] Query billions of rows in milliseconds. https://clickhouse.com/, 1 2024. Accessed: June 7, 2024.

[14] Clickhouse server 23.7.1.857 terminated by sigabrt through create table and select stmts. https://github.com/ClickHouse/ClickHouse/issues/52049, 7 2023. Accessed: June 7, 2024.

[15] Coronel, C., and Morris, S. *Database systems: design, implementation and management*. Cengage learning, 2019.

[16] Damengdb. https://en.dameng.com/, 1 2024. Accessed: June 7, 2024.

[17] Duvall, P. M., Matyas, S., and Glover, A. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[18] Fowler, M., and Foemmel, M. Continuous integration, 2006.

[19] Fu, J., Liang, J., Wu, Z., and Jiang, Y. Sedar: Obtaining high-quality seeds for DBMS fuzzing via cross-dbms SQL transfer. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024* (2024), ACM, pp. 146:1–146:12.

[20] Fu, J., Liang, J., Wu, Z., Wang, M., and Jiang, Y. Griffin: Grammar-free dbms fuzzing. In *Conference on Automated Software Engineering (ASE'22)* (2022).

[21] Hipp, R. Potential database corruption in sqlite versions 3.35.0 through 3.37.1. https://sqlite.org/forum/forumpost/ac381d64d804407e?raw, 1 2022. Accessed: June 7, 2024.

[22] Security oriented fuzzer with powerful analysis options. https://github.com/google/honggfuzz. Accessed: June 7, 2024.

[23] Jonathan Metzman, G. O. S. S. T. Clusterfuzzlite: Continuous fuzzing for all. https://github.com/google/clusterfuzzlite, 11 2021. Accessed: June 7, 2024.

[24] Jung, J., Hu, H., Arulraj, J., Kim, T., and Kang, W. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems (to appear). In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)* (Tokyo, Japan, Aug. 2020).

[25] Klooster, T., Turkmen, F., Broenink, G., Hove, R. T., and Böhme, M. Continuous fuzzing: A study of the effectiveness and scalability of fuzzing in ci/cd pipelines. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)* (2023), pp. 25–32.

[26] Kumar, V., and Son, S. H. *Database recovery*, vol. 12. Springer Science & Business Media, 2012.

[27] Liang, J., Chen, Y., Wu, Z., Fu, J., Wang, M., Jiang, Y., Huang, X., Chen, T., Wang, J., and Li, J. Sequence-oriented dbms fuzzing. In *2023 IEEE International Conference on Data Engineering (ICDE)*, IEEE.

[28] Liang, J., Wang, M., Chen, Y., Jiang, Y., and Zhang, R. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (Los Alamitos, CA, USA, mar 2018), IEEE Computer Society, pp. 562–566.

[29] Liang, J., Wang, M., Zhou, C., Wu, Z., Jiang, Y., Liu, J., Liu, Z., and Sun, J. Pata: Fuzzing with path aware taint analysis. In *2022 2022 IEEE Symposium on Security and Privacy (SP)(SP). IEEE Computer Society, Los Alamitos, CA, USA* (2022), pp. 154–170.

[30] Liang, J., Wu, Z., Fu, J., Wang, M., Sun, C., and Jiang, Y. Mozi: Discovering DBMS bugs via configuration-based equivalent transformation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024* (2024), ACM, pp. 135:1–135:12.

[31] Libfuzzer. https://www.llvm.org/docs/LibFuzzer.html. Accessed: June 7, 2024.

[32] Liu, X., Zhou, Q., Arulraj, J., and Orso, A. Automatic detection of performance bugs in database systems using equivalent queries. In *Proceedings of the 44th International Conference on Software Engineering* (2022), pp. 225–236.

[33] Manès, V. J., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., and Woo, M. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering 47*, 11 (2019), 2312–2331.

[34] Mariadb. https://mariadb.org/, 1 2024. Accessed: June 7, 2024.

[35] Miller, B. P., Fredriksen, L., and So, B. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM 33*, 12 (Dec. 1990).

[36] The database system to speed up your analytical jobs. https://www.monetdb.org/, 1 2024. Accessed: June 7, 2024.

[37] Mysql. https://www.mysql.com/, 1 2024. Accessed: June 7, 2024.

[38] Polardb. https://www.alibabacloud.com/product/polardb, 1 2024. Accessed: June 7, 2024.

[39] Postgresql. https://www.postgresql.org/, 1 2024. Accessed: June 7, 2024.

[40] Rigger, M., and Su, Z. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020), pp. 1140–1152.

[41] Rigger, M., and Su, Z. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages 4*, OOPSLA (2020), 1–30.

[42] Rigger, M., and Su, Z. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation OSDI 20)* (2020), pp. 667–682.

[43] Seltenreich, A., Tang, B., and Mullender, S. Sqlsmith: a random sql query generator.

[44] Serebryany, K. OSS-Fuzz - google's continuous fuzzing service for open source software. USENIX Association.

[45] Shahin, M., Babar, M. A., and Zhu, L. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE access 5* (2017), 3909–3943.

[46] Slutz, D. R. Massive stochastic testing of SQL. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, New York, USA*, Morgan Kaufmann, pp. 618–622.

[47] Sqlite home page. https://www.sqlite.org/index.html, 1 2024. Accessed: June 7, 2024.

[48] Embrace industry 4.0 with tdengine — the next generation data historian. https://tdengine.com/, 1 2024. Accessed: June 7, 2024.

[49] Tendb cluster. https://github.com/Tencent/TenDBCluster-TenDB, 1 2024. Accessed: June 7, 2024.

[50] Tran, S. Fuzzing collaboration with wingfuzz- part 1. https://clickhouse.com/blog/fuzzing-wingfuzz, 8 2023. Accessed: June 7, 2024.

[51] Van Der Lans, R. F. *The SQL standard: a complete guide reference*. Prentice Hall International (UK) Ltd., 1989.

[52] Vastbase. https://www.vastdata.com.cn/, 1 2024. Accessed: June 7, 2024.

[53] Verhofstad, J. S. Recovery techniques for database systems. *ACM Computing Surveys (CSUR) 10*, 2 (1978), 167–195.

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

[55] Widenius, M., Axmark, D., and Arno, K. *MySQL reference manual: documentation from the source*. " O'Reilly Media, Inc.", 2002.

[56] Wu, Z., Liang, J., Wang, M., Zhou, C., and Jiang, Y. Unicorn: Detect runtime errors in time-series databases with hybrid input synthesis. In *Symposium on Software Testing and Analysis (ISSTA'22)* (2022).

[57] Yashandb. https://www.yashandb.com/, 1 2024. Accessed: June 7, 2024.

[58] Yoo, H., Hong, J., Bader, L., Hwang, D. W., and Hong, S. Improving configurability of unit-level continuous fuzzing: An industrial case study with sap hana. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2021), pp. 1101–1105.

[59] Zalewski, M. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: June 7, 2024.

[60] Zhang, J., Cui, Z., Chen, X., Yang, H., Zheng, L., and Liu, J. Cidfuzz: Fuzz testing for continuous integration. *IET Software 17*, 3 (apr 2023), 301–315.

[61] Zhong, R., Chen, Y., Hu, H., Zhang, H., Lee, W., and Wu, D. Squirrel: Testing database management systems with language validity and coverage feedback. In *The ACM Conference on Computer and Communications Security (CCS), 2020* (2020).

[62] Zhu, X., and Böhme, M. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2021), CCS '21, Association for Computing Machinery, p. 2169–2182.

[63] Zhu, X., Wen, S., Camtepe, S., and Xiang, Y. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR) 54*, 11s (2022), 1–36.