Understanding and Detecting SQL Function Bugs

Using Simple Boundary Arguments to Trigger Hundreds of DBMS Bugs

Jingzhou Fu
KLISS, BNRist, School of Software,
Tsinghua University, China

Jie Liang*
KLISS, BNRist, School of Software,
Tsinghua University, China

Zhiyong Wu
KLISS, BNRist, School of Software,
Tsinghua University, China

Yanyang Zhao
KLISS, BNRist, School of Software,
Tsinghua University, China

Shanshan Li
National University of Defense
Technology, China

Yu Jiang*
KLISS, BNRist, School of Software,
Tsinghua University, China

Abstract

Built-in SQL functions are crucial in Database Management Systems (DBMSs), supporting various operations and computations across multiple data types. They are essential for querying, data transformation, and aggregation. Despite their importance, the bugs in SQL functions have caused widespread problems in the real world, from system failures to arbitrary code execution. However, the understanding of the bug characteristics is limited. More importantly, conventional function testing methods struggle to generate semantically correct SQL test cases, while DBMS testing efforts are hard to measure built-in SQL functions.

This paper presents a comprehensive study of 318 built-in SQL function bugs, shedding light on their characteristics and root causes. Our investigation reveals that 87.4% of these bugs were caused by improper handling of boundary values of arguments. The boundary values of arguments come from three sources: literal values, type castings, and nested functions. By studying the bugs from three sources, we summarized 10 SQL patterns of bug-inducing queries. Moreover, we designed Soft, a testing tool based on the patterns to test seven widely used DBMSs, including PostgreSQL, MySQL, and ClickHouse. Soft discovered and confirmed 132 previously unknown SQL function bugs. The DBMS vendors took these bugs seriously and fixed 97 bugs in three days. For example, the CTO of ClickHouse commented on one bug: "*We must fix it immediately or get rid of this function.*"

*Jie Liang and Yu Jiang are the corresponding authors.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Keywords: DBMS, SQL Function, Vulnerability Detection

ACM Reference Format:
Jingzhou Fu, Jie Liang, Zhiyong Wu, Yanyang Zhao, Shanshan Li, and Yu Jiang. 2025. Understanding and Detecting SQL Function Bugs: Using Simple Boundary Arguments to Trigger Hundreds of DBMS Bugs. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3689031.3696064

1 Introduction

Built-in SQL functions [23, 44] are predefined functions in SQL that perform specific operations. They are crucial in Database Management Systems (DBMSs), serving as the backbone for efficient data handling and manipulation. These functions cater to different types of data for computations and operations, providing a robust toolkit for managing the complexity of database interactions. For example, Listing 1 shows the `toDecimalString` function, a typical built-in SQL function in ClickHouse. It converts numerical values into string representations with a specified number of digits [31].

Built-in SQL functions are crucial and widely used for the smooth operation and flexibility of DBMSs. Despite their significance, the implementation of built-in SQL functions can sometimes be flawed. In this paper, we regard *built-in SQL function bugs* as those memory issues within the built-in SQL functions that can lead to system crashes, incorrect operations, or unauthorized data manipulations when interacting with a database. For example, the function `toDecimalString` in Listing 1 has an implementation error that can lead to a null pointer dereference [21] when crafted values are passed as arguments of SQL function expressions.

Listing 1. The built-in SQL function `toDecimalString` contains a null pointer dereference in ClickHouse found by our tool, which can cause a denial of service.

```
-- Null pointer dereference in toDecimalString
SELECT toDecimalString('110'::Decimal256(45), *);
```

Built-in SQL function bugs have the following particular attributes that render them notably troublesome:

(1) *Widespread influence.* Built-in SQL functions are integral to database operations since the computation and manipulation of various data types heavily depend on specific built-in SQL functions. For example, PostgreSQL supports 56 built-in SQL functions designed to perform operations and calculations for string data types [23]. *As a consequence, a bug in a commonly used SQL function may be activated repeatedly across numerous SQL statements.* This situation underscores their widespread influence, as each invocation of these flawed functions can propagate errors or vulnerabilities through multiple layers of a DBMS's operations. Besides, other parts of a database or application may also rely heavily on specific built-in SQL functions. A bug in such a function can cascade, causing interoperability issues and leading to broader system failures.

(2) *High severity.* Built-in SQL function bugs pose significant and severe threats to DBMSs, potentially leading to denials of service, unauthorized access to sensitive memory areas, and even arbitrary code execution, which can compromise the security and integrity of the entire DBMS. As an illustrative example, since 2004, PostgreSQL has reported 121 Common Vulnerabilities and Exposures (CVEs) [25], with 31 of these vulnerabilities directly attributed to built-in SQL functions. These 31 vulnerabilities have an average CVSS 2.0 score of 6.09, which is notably higher than the overall average score of 5.54 for all 121 PostgreSQL CVEs. Moreover, built-in SQL function bugs are easy to exploit. For instance, some applications allow searches via user-entered regular expressions, allowing attackers to submit crafted inputs to trigger the potential vulnerabilities in the backend DBMS's regular expression functions.

(3) *Complex and difficult to expose.* The complexity and difficulty in exposing bugs within built-in SQL functions stem from their sophisticated designs, which incorporate intricate logic to manage a diverse array of data types and execute a multitude of operations. The intricate logic and the nuanced handling of data types can obscure bugs, making them difficult to identify and isolate. Such bugs might remain dormant, only surfacing under specific conditions that are not easily replicated or anticipated during standard testing procedures. Furthermore, the challenge is compounded by the difficulty in exposing these bugs, as they can often require detailed and exhaustive testing scenarios that encompass a wide range of use cases and data inputs.

Although built-in SQL function bugs are serious for DBMSs, the understanding of their characteristics is limited. More importantly, there is a lack of testing methods for these SQL functions. Conventional testing approaches for programming languages can test functions in libraries or programs. However, they are not suitable for generating SQL function test cases. For example, traditional unit test generations like Randoop and EvoSuite [17, 46] will build specific unit test cases for library functions. These works typically synthesize test cases containing rich sequences of function calls. In contrast, SQL function expressions are in the form of nested expression structures with specifically formatted function arguments. They significantly differ from the sequence form of test cases generated by these works. On the other hand, existing DBMS testing works like SQLsmith [53] and Squirrel [63] also struggle to test built-in SQL functions. They mainly generate complex SQL syntax structures or sequences of SQL operations to trigger more behaviors in DBMS components. The generation of function expressions is not primarily considered and designed in these works, resulting in missing SQL function bugs of DBMSs.

To better understand and detect SQL function bugs, we studied 318 built-in SQL function bugs from three widely used DBMSs, including PostgreSQL, MySQL, and MariaDB. We carefully analyzed the report and source code of each bug to identify the characteristics of their occurrence stages, function types, root causes, and triggering conditions. We found that about 70.0% of the SQL function bugs occurred at the execution stage. About 40.9% of the SQL function bugs are located in string functions and aggregate functions. We also found that 87.5% of bug-inducing statements contain no more than two function expressions.

More importantly, we found that 87.4% of these SQL function bugs are caused by poor handling of boundary values of function arguments, which could be seen as the main root cause for the built-in SQL function bugs. The boundary values of arguments originate from three sources: boundary literal values, boundary results of type castings, and boundary return values of nested functions. Furthermore, by studying bugs from these three sources, we summarized 10 *boundary-value-generation patterns* of bug-inducing queries.

Based on these findings, we implemented Soft, a testing tool that generates test cases containing built-in SQL function expressions, with arguments produced based on these patterns. To evaluate the performance of Soft, we apply Soft to test the SQL functions in seven widely-used DBMSs: PostgreSQL, MySQL, MariaDB, ClickHouse, MonetDB, DuckDB, and Virtuoso. Compared to existing DBMS testing tools, Soft triggers 984, 1567, and 181 more functions of these DBMSs, and covers 433.93%, 98.70%, and 19.86% more code branches of the SQL function components of DBMSs in 24 hours than Squirrel, SQLancer, and SQLsmith, respectively. Moreover, Soft detected 132 previously unknown bugs in these DBMSs, which have garnered the attention of the DBMS developers. For example, ClickHouse's CTO quickly commented on the SQL function bug shown in Listing 1 when we reported it [21]: "**We must fix it immediately or get rid of this function.**" The CTO then promptly submitted a pull request to remove the flawed function. Within two days, ClickHouse developers fixed the bug and resubmitted the SQL function's code.

In summary, this paper has the following contributions:

1. We analyzed 318 SQL function bugs and summarized the main root cause of SQL function bugs, namely the poor handling of boundary values of arguments.
2. We summarized 10 boundary-value-generation patterns to detect SQL function bugs. Based on that, we proposed Soft to detect SQL function bugs in DBMSs by constructing boundary arguments.
3. Soft discovered 132 previously unknown bugs in real-world DBMSs. All of them were confirmed by developers, and 97 were fixed.

2 Background

DBMSs and SQL functions. A *Database Management System (DBMS)* is software that enables users to define, create, maintain, and control access to databases [59]. *Structured Query Language (SQL)* is the standard language for interacting with DBMS [55]. It is used for creating, modifying, and querying relational databases. *SQL functions* are constructs available in SQL that perform specific operations or return values. They carry out calculations, data modifications, formatting, and various other tasks directly within SQL queries. SQL functions can be broadly categorized into two types:

(1) Built-in Functions [23, 44]: These are predefined functions included within SQL. They are standardized across different platforms, although there might be slight variations in their implementations or the range of available functions.

(2) User-Defined Functions (UDFs) [28, 43]: These are functions created by users to perform operations that are not covered by the built-in SQL functions. UDFs allow for the customization of logic to meet specific data processing needs. *If not specified otherwise, the subsequent occurrences of "SQL functions" in this paper refer to "built-in SQL functions".*

Unlike programming language functions, SQL functions do not have complex function call dependencies and sequences. For example, Listing 2 shows the XML updating operations in SQL functions compared to those in JavaScript library functions. JavaScript employs a function call sequence with a series of classes, objects, and functions to operate the XML object. In contrast, SQL functions do not have arguments like pointers, references, or class objects in programming languages. Instead, SQL functions rely on various formats of argument values to represent the data and operations of SQL functions for flexibility and convenience.

SQL Function Processing Steps. The processing of a SQL function expression in a DBMS typically involves three steps. (1) Parsing arguments: the arguments of a function expression are parsed by DBMSs as literals, column names, or aliases. (2) Calculating and casting argument values: the values of these arguments are calculated by DBMSs, and these values are cast to the corresponding parameter types of the function. (3) Executing the function: the corresponding core implementation of the SQL function computes return values with these given values of parameters. For example,

Listing 2. Comparison of the XML updating operation in SQL function and JavaScript library function.

```
/* Updating XML with JavaScript Function */
xmlDoc = new DOMParser().parseFromString("<a><c></c
     ></a>", "text/xml");
cTag = xmlDoc.getElementsByTagName('c')[0];
bTag = xmlDoc.createElement('b');
cTag.parentNode.replaceChild(bTag, cTag);
newXmlStr = new XMLSerializer().serializeToString(
     xmlDoc);

/* Updating XML with SQL Function in MySQL */
SELECT UpdateXML('<a><c></c><a>', '/a/c[1]', '<c><b
     ></b></c>');
```

for the XML operation shown in Listing 2, the DBMS first parses the arguments as string literals. Then, the strings `'<a><c></c></a>'` and `'<c><b></b></c>'` are converted to the internal XML type of DBMS, while `'/a/c[1]'` is converted to the XPATH type. Finally, the DBMS performs operations on the internal XML and XPATH types to update the structure of the XML value. These steps of processing SQL function expressions are pre-implemented by developers using programming languages.

SQL Function Bugs. When DBMS processes SQL function expressions, memory errors or semantic issues may occur due to flawed implementations. In this paper, *SQL function bugs* refer to the flaws in implementations of processing SQL functions that can cause memory errors, such as null pointer dereferences and buffer overflows. These bugs can be triggered by SQL statements with crafted SQL function expressions, leading to system failure, memory disclosure, and even arbitrary execution.

However, there is currently a lack of testing approaches aimed at detecting SQL function bugs. Traditional library function testing methods for programming languages are also ineffective for SQL functions due to the different characteristics of SQL functions. For instance, these testing methods mainly focus on establishing dependencies between functions and constructing valid function call sequences, which is unsuitable for SQL functions.

In comparison, SQL function testing is more similar to domain testing since the behavior of SQL functions mostly depends on the input arguments passed into them. Domain testing aims to partition the input set into multiple domains and select test cases on the boundaries of these domains to detect domain errors. A domain error means that the input enters an unexpected program path during execution. Similarly, SQL functions accept a range of arguments in specific formats. They handle correct input as intended while returning error messages for incorrect or invalid input. In some cases, if the passed arguments of an SQL function are on a boundary condition, the DBMS may incorrectly handle these edge cases, leading to unexpected execution paths and potential bugs. For instance, if an SQL function incorrectly treats an invalid XML string as a valid one, it could result in subsequent illegal operations and cause the DBMS to crash.

Therefore, inspired by generating boundary test data in domain testing, the testing of SQL functions may also need to focus on their boundary conditions and construct corresponding SQL test cases. However, due to the diverse types of input arguments of SQL functions, it is challenging to identify and partition their domains and boundary conditions directly. To address this, we performed a study of existing SQL function bugs to understand what inputs are more likely to trigger these boundaries and bugs of SQL functions.

3 Methodology

We conducted the study of SQL function bugs on three popular DBMSs: PostgreSQL, MySQL, and MariaDB. We mainly collected SQL function bugs from their bug trackers, including PostgreSQL Bug Report Mailing List [24], MySQL Bug System [12], and MariaDB's JIRA [6]. We also found SQL function bugs from the CVE list [25]. Table 3 presents the number of collected bugs.

To collect SQL function bugs from these bug trackers, we first searched the bugs whose descriptions contain the words "crash" and "signal" to list possible vulnerabilities and found 14,111 bug reports from the bug trackers. The bugs tagged as "duplicated" or "not a bug" are excluded by setting the filters of MySQL's and MariaDB's bug tracker. For PostgreSQL, since its bug tracker does not provide the filter of bug tags, we performed this exclusion in a later step. Then, we analyzed the detailed pages of each bug report to parse the SQL statements present on these pages with SQL parsers. These SQL statements are usually the Proof of Concept (PoC) for reproducing the bugs, and 658 of them contain SQL function expressions. Finally, we manually checked the bug report and the PoC of each bug. If the crash happened after executing a `SELECT` statement containing a function expression, we classified it as an SQL function bug and identified the root cause according to its crash backtrace and bug patch. We also removed duplicated PostgreSQL bugs here based on developers' replies to these bug reports. In this way, we collected 39, 10, and 269 SQL function bugs from the bug trackers of PostgreSQL, MySQL, and MariaDB, respectively.

Threats to Validity. Similar to other characteristic studies, our results have the following limitations that should be taken into account.

Invisibility of high-severity vulnerabilities. The details of high-severity vulnerabilities, such as those causing data disclosures and remote code executions, are generally not disclosed by security teams. For example, high-severity security bugs of MySQL and PostgreSQL are requested to be reported directly to their security team via email, and their security mailing list is not public. Consequently, our study misses some of the most serious vulnerabilities, leading to a lack of corresponding statistical information in our results. However, similar to the crash bugs studied in this paper, these vulnerabilities are mainly caused by memory issues (e.g., buffer overflow). Hence, our study on crash bugs can still provide valuable insights for detecting such vulnerabilities.

Table 1. The numbers of collected built-in SQL function bugs in selected DBMSs.

DBMSs	PostgreSQL	MySQL	MariaDB	Total
Studied Bugs	39	10	269	318

Omission of non-crash Bugs. We utilized the keywords "crash" and "signal" to search for bugs, implying that we did not consider functionality bugs, such as incorrect computation results. This is because a thorough bug search will generate numerous false positives. For instance, many bug reports use "`SELECT COUNT(*)`" to indicate incorrect result set sizes for `SELECT` queries, highlighting logic bugs. However, such bugs are typically caused by the wrong result sets of queries, not the flaws in the `COUNT` function itself. They should not be classified as SQL function bugs.

Representativeness of selected DBMSs. We selected PostgreSQL, MySQL, and MariaDB as the targets of our study, all of which are popular relational DBMSs using SQL as their query languages. We did not include other types of DBMSs in our study, such as graph or time-series DBMSs. This is because they often utilize their own query syntax rather than SQL grammar, which would heavily complicate the analysis. However, our study results could still provide meaningful guidance for these DBMSs due to the similarities in their support for function features.

4 General Findings of SQL Function Bugs

In this section, we perform an in-depth analysis of 318 built-in SQL function bugs. We analyzed these SQL bugs to identify their common characteristics, focusing on three key aspects:

1. The stage of DBMS processing when the crash occurs;
2. The types of SQL functions that triggered the crash;
3. The prerequisite SQL statements that the bug-inducing statement depends on.

4.1 Occurrence Stages

DBMSs usually process SQL queries in three stages: parsing, optimization, and execution. First, the DBMS parser converts the SQL statement sent by users into internal Abstract Syntax Trees (ASTs). Next, the optimizer analyzes the ASTs and tries to generate an optimized query plan. Finally, the executor executes the query plan and returns the results to users, ending the SQL statement process. SQL function bugs within statements can occur during all three stages of processing and potentially cause a crash. Analyzing the specific stage where the bug occurs helps to pinpoint its root cause.

To identify the stage of DBMS processing when crashes occur, we extracted the crash backtrace information from bug reports. There are 230 bug reports containing identifiable backtrace information out of the 318 collected bugs. We examined the symbol names within these backtraces to classify the stage where the DBMS crashes occurred. Regarding the distribution of SQL function bug occurrence stages, we present the following finding:

Finding 1. *Among the studied bugs with identifiable backtraces, about 70.0% (161/230) of them occurred at the execution stage, about 19.6% (45/230) occurred at the optimization stage, and about 10.4% (24/230) occurred at the parsing stage.*

Crashes at the execution stage are mainly caused by crafted arguments passing to the SQL functions. These arguments are successfully parsed into the internal data types of DBMS, but flaws in the computation for the SQL functions cause the DBMS to crash when handling specific arguments. In contrast, crashes at the parsing and optimization stages are typically triggered by complex SQL nested structures within or surrounding the function expressions.

4.2 Types of SQL Functions

DBMSs generally support various types of SQL functions for operating on values of different data types, such as aggregate functions and string functions. For example, the string functions in most DBMSs provide operations on strings like concatenation, replacement, and substring extraction. Different SQL functions offer distinct computations in DBMSs.

To better understand the types of computations leading to SQL function bugs, we conducted a statistical analysis of the functions encountered in PoCs. We parsed the SQL statements within these PoCs and extracted the function expressions contained in each statement. We then classified these functions into different function types following the official documentations of MySQL [44] and PostgreSQL [23], with the results shown in Figure 1. "# of occurrences" for a function type means the total number of occurrences of this type's functions, and "# of unique SQL functions" presents how many different functions of this type have occurred at least once. Notably, the total occurrence count exceeds the number of SQL function bugs, as one PoC can contain multiple function expressions.

Finding 2. *Among the studied bugs, the SQL function expressions have a total occurrence of 508 in the PoCs of these bugs. The most common types of bug-inducing SQL functions are string functions (117/508, about 23.0%) and aggregate functions (91/508, about 17.9%). Over 40% of the bugs were caused by these two types of SQL functions.*

String functions are the most common functions in DBMSs. DBMSs support abundant string functions for various string operations, such as search, replacement, regular expressions, and hashing. The larger number of string functions compared to other functions potentially increases the likelihood of encountering bugs. As shown in Figure 1, there are 57 distinct string functions causing SQL function bugs, which is significantly more than other types of functions.

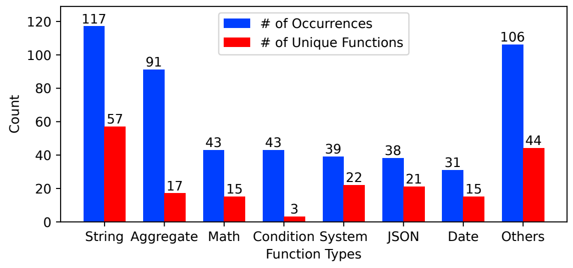

Figure 1. The count of occurrences and the number of unique SQL functions in the bug-inducing statements of studied bugs for each SQL function type.

The aggregate functions are a special type of the SQL functions, which involve more complex computations than non-aggregate functions. Basic functions only need to compute the return value from a few fixed types of parameters. In contrast, aggregate functions operate on all elements of one or more columns at the same time, requiring support for various data types and values. Moreover, aggregate functions are often utilized with keywords like `GROUP BY`, `HAVING`, and `DISTINCT`, involving the grouping, filtering, and sorting operations. This complexity makes aggregate functions more likely to contain flaws.

Table 2. The count of occurrences of function expressions within the bug-inducing statements in PoCs.

Occurrences of Function Expressions	1	2	3	4	≥ 5
Number of Bug-inducing Statements	191	87	23	11	6

Beyond function types, we also analyzed the number of SQL function expressions within each bug-inducing statement. As displayed in Table 2, about 60.1% of bug-inducing statements contain only one SQL function expression inside, about 27.4% contain two SQL function expressions, and the rest include more than two function expressions.

Finding 3. *A majority (278/318, about 87.5%) of the studied bugs contain no more than two function expressions in their bug-inducing statements.*

4.3 Prerequisite SQL Statements

The execution of real-world SQL queries often relies on preexisting data. The data are usually prepared by prerequisite statements, such as table creation and data insertion. Missing these statements can lead to semantic errors or empty result sets. Similarly, Some bug-inducing statements may depend on prerequisite statements to cause a crash. We examined the SQL statements and their dependencies among the PoCs of collected SQL function bugs.

Finding 4. *About 47.5% (151/318) of the studied bugs rely on both table creation and data insertion, about 41.5% (132/318) can crash the DBMS without relying on any table, and about 11.0% (35/318) depend on specific tables without data inserted.*

For the bugs relying on both table creation and data insertion, their PoCs utilize prerequisite `CREATE TABLE` and `INSERT` statements to prepare specific data types and values. These data are then passed to function expressions in the bug-inducing statements via the `FROM` clause, leading to crashes. For the bugs that do not depend on any table, literal expressions are used to construct the crafted data types and values. By comparison, the PoCs depending on empty tables usually contain complex `CREATE TABLE` statements. These table-creation statements include complex column definitions, such as particular column data types and constraints.

5 Root Causes of SQL Function Bugs

We manually reviewed each bug report and further analyzed the root causes of these SQL function bugs. We found that 87.4% (278/318) of these SQL function bugs are caused by improper handling of boundary values of arguments, which could be seen as the main root cause for the built-in SQL function bugs. Boundary values refer to the values at the edges between the acceptable and unacceptable inputs. In the context of SQL functions, the boundary values are those at the edges of expected formats of SQL function arguments, including structures, ranges, lengths, and nested depths. We identified that these boundary values of SQL functions come from three types of expressions: the boundary literal values, the boundary results of type castings, and the boundary return values from nested functions.

It is understandable that boundary values are more likely to cause SQL function bugs. SQL functions accept a variety of formats for the function arguments, such as dates, IP addresses, JSON, and XML. They allow users to write SQL function expressions more conveniently, but require the DBMSs to handle SQL function arguments more cautiously. The implementation of SQL functions must carefully check whether the arguments match the expected formats, properly convert them into built-in types, and accurately perform the operations. Therefore, the various formats of arguments bring complex boundary conditions when handling SQL function expressions, which leads to potential SQL function bugs triggered by boundary values.

5.1 Boundary Literal Values

Boundary literal values refer to extreme and specific numerical values used in testing scenarios to evaluate how a system or function behaves at the boundaries of its input ranges. These values are directly specified as constants or literals within the test cases and are chosen to represent the minimum, maximum, or critical edge cases of the input data. Some SQL functions mishandle certain boundary values from arguments. About 29.5% (94/318) of the studied bugs are caused by the boundary literal values as arguments. These bugs are caused by the flawed implementation of the SQL functions. The relevant code of the implementation lacks sufficient robustness when encountering boundary values, resulting in the occurrence of bugs.

Listing 3. Bugs caused by boundary literal values.

```
-- PostgreSQL CVE-2016-0773:
-- regex matching with char integer overflow
SELECT 'a' ~ '\x7fffffff';

-- MDEV-23415:
-- formatting operation setting with 50 digits
-- Format: FORMAT(number, decimal_places, locale)
SELECT FORMAT('0', 50, 'de_DE');
```

Bug Samples. Listing 3 shows two bugs caused by the boundary literal values.

CVE-2016-0773: an integer overflow in PostgreSQL when handling regex matching [27]. The character value in the regex pattern (`\x7fffffff`) is converted to an int32 variable `b` during processing. The overflow occurs at a C language statement "`for (c = a; c <= b; c++)`" in the implementation of PostgreSQL when handling the regex matching function. This overflow leads to infinite loops and finally causes denials of service. The developers fixed the bug by restricting the character value in regex expressions to always be less than `\x7ffffffe` to avoid the overflow. On the updated version of PostgreSQL, if the regex expression contains `\x7fffffff`, it will throw an error to the client: "ERROR: invalid regular expression: invalid escape sequence".

MDEV-23415: a heap buffer overflow in MariaDB when processing the `FORMAT` *function* [9]. The second argument "50" represents the expected digits of the fractional part in the formatted result. However, the MariaDB library function "String::set_real()" for the formatting operation will automatically return the value in scientific notation when the number of digits exceeds 31. The returned string of scientific notation (such as "1e-32") is shorter than the expected digits, resulting in a heap buffer overflow when writing the formatting results into this string.

Root Causes. The improper handling of boundary literal values in SQL function implementations can cause unexpected behaviors during processing. For example, values that are too large or too small in numeric arguments can cause overflows for integer variables or out-of-range indexes for arrays when DBMS lacks corresponding checks. Developers usually fix these bugs by adding code to verify that these argument values are within the valid range.

It also illustrates one huge difference between the SQL functions in DBMS and the library functions in programming languages. A library can opt not to handle such arguments because illegal argument values are typically considered misuse by developers rather than bugs of the library itself. In contrast, DBMS must handle these arguments carefully since they may originate from external users. Non-handling or mishandling could leave the system vulnerable to attacks from malicious actors.

5.2 Boundary Type Castings

Type casting involves converting a value from one data type to another. Sometimes, function arguments are incorrectly converted to the internal data types of DBMSs, leading to SQL function bugs. When DBMS handles function expressions, function arguments are first converted into internal data types of the DBMS. For instance, a 60-digit decimal in MariaDB function expressions, which is too large to represent by a `double` variable, will be converted into the internal data type within the MariaDB decimal library [22]. However, such type conversions are not always correctly implemented, such as inadequately handling excessively large values or NULL values. About 23.3% (74/318) of studied bugs are caused by boundary arguments constructed by type casting.

Listing 4. Bugs caused by boundary type castings.

```
-- MDEV-8407: numerics with lots of digits
SELECT COLUMN_JSON(COLUMN_CREATE('x',
123456789012345678901234567890123456789012346789));

-- MDEV-11030: casting NULL to a number
SELECT * FROM (
  SELECT IFNULL(CONVERT(NULL, UNSIGNED), NULL)) sq;
```

Bug Samples. Listing 4 displays two bugs caused by the boundary result of type castings.

MDEV-8407: a heap buffer overflow in MariaDB when converting a large number from MariaDB's decimal type to the string type [11]. In this sample, a 48-digit decimal number is first converted to MariaDB's decimal data type when processing the `COLUMN_CREATE` function expression. Subsequently, when handling the `COLUMN_JSON` function, the decimal value will be converted to a string type by the "decimal2string" library function in MariaDB's decimal library. However, the library function contains an error when converting decimals larger than 40 digits. It incorrectly calculates the length of the resulting string, leading to insufficient space allocation for the string and finally triggering a heap buffer overflow.

MDEV-11030: a heap buffer overflow when converting a null value to the integer type in MariaDB [7]. When a value is converted to MariaDB's integer type, MariaDB stores the value into an int64 variable and calculates the digits of the value. However, when the original value is a `NULL` value, MariaDB stores it as the integer `0` but incorrectly calculates the number of its digits as zero ('`0`' occupies one digit rather than zero). The incorrect number of digits results in insufficient space allocation in subsequent string contexts, leading to a heap buffer overflow.

Root Causes. The implementation of DBMS's type casting may contain flaws that are difficult to detect directly. These flaws do not immediately cause the DBMS to crash but result in broken internal data type instances, which are hard to observe. However, the subsequent processing of SQL functions depends on the arguments of these internal data types. These broken internal instances can cause function computation errors and even DBMS crashes. Such SQL function bugs caused by boundary type castings are actually the bugs of the type systems of DBMSs rather than the implementations of SQL functions.

5.3 Boundary Results of Nested Functions

Apart from boundary literal values and type castings, another major cause of SQL function bugs is the arguments derived from the return values of nested functions. About 34.6% (110/318) of the studied bugs are caused by boundary arguments originating from the return values of nested function expressions. These nested functions can construct boundary arguments with specific data types and values, such as special binary streams or extremely long strings in specific formats. Such boundary arguments may also trigger errors in SQL function implementations.

Listing 5. Bugs from boundary values by nested expressions.

```
-- PostgreSQL CVE-2015-5289:
-- a large string constructed by nested structures
SELECT REPEAT('[', 1000)::json;

-- MDEV-14596: interval operation on the row type
-- Format: INTERVAL(N, N1, N2, N3, ...)
SELECT INTERVAL(ROW(1,1),ROW(1,2));
```

Bug Samples. Listing 5 presents two SQL function bugs caused by boundary results of nested functions.

CVE-2015-5289: a stack overflow in PostgreSQL when converting a long string to the JSON format [26]. The bug is triggered by using the repeat function to repeat the character '`[`' 1000 times. in JSON, "`[...]`" represents an array, and such JSON array expressions are recursively processed in PostgreSQL. Each '`[`' character represents the beginning of a new JSON array, which makes PostgreSQL recursively call the "parse_array" library function during parsing. When there are too many '`[`' characters, it keeps calling the library function recursively and overflows the stack in the final due to excessive recursion depth. PostgreSQL fixed this bug by adding recursion depth checks in JSON functions.

MDEV-14596: a segmentation violation in MariaDB triggered by illegal comparison operations on row types [8]. The `INTERVAL` function in MariaDB takes multiple arguments. It compares the 1st argument (`N`) with each subsequent argument (`N1`, `N2`, `N3`, ...), and returns the index of the first argument larger than N. Therefore, `INTERVAL` relies on comparison operations, so its arguments must be comparable objects. In the MDEV-14596 case, two values with the `ROW` type, `ROW(1,1)` and `ROW(1,2)`, are passed as arguments to the `INTERVAL` function. However, `ROW` types do not support comparison operations, and MariaDB did not validate this within the `INTERVAL` function. This leads to a segmentation violation when attempting comparisons on `ROW` types.

Root Causes. These SQL function bugs are still caused by the boundary values or data types of the arguments. The implementations of the SQL functions do not correctly handle these crafted arguments. However, these boundary arguments are not from literal values or type castings, but from the return values of the nested functions. Moreover, these arguments are difficult to represent by literal values or type castings due to their extreme lengths or special data types. Their lengths may even exceed the maximum length allowed for SQL statements by the DBMS.

5.4 Other Root Causes

Apart from the boundary arguments of SQL function expressions, we found other root causes of SQL function bugs, including DBMS configurations, specific table definitions, and complex syntax structures. However, these root causes are less frequent in the studied SQL function bugs. They are harder to trigger since they depend on the unique and complex features of different DBMSs, such as the incorrect settings in their configuration files.

Database System Configurations. Specific configurations within the database systems may also cause errors. For instance, the incorrect or incompatible settings of buffer sizes, improper time zones, and character sets may cause some SQL functions of DBMSs to work improperly, leading to potential errors and failures. In our study, we found that 8 bugs are related to the configurations.

Specific Table Definitions. The specific definitions and structures of tables, such as storage engines, indexes, and column constraints, can alter the handling of `SELECT` statements by DBMSs. For example, the indexes on tables can affect the processing of `WHERE` clauses; if the implementation of indexes contains flaws, it may cause errors when DBMSs process certain function expressions in these clauses with indexing enabled. In our study, we observed that 24 bugs are associated with specific table definitions.

Complex Syntax Structures. Errors can arise from the complexity of SQL syntax structures used in queries and procedures. Complex SQL queries may introduce opportunities for errors, such as crashes during parsing function expressions with complex clauses and nested subqueries contained. We found that there are 8 bugs associated with complex syntax structures in our studied bugs.

6 Boundary Value Generation Patterns

Our study shows that the main root causes of SQL function bugs are the flawed handling of boundary values of arguments, including boundary literal values, boundary type castings, and boundary return values of nested functions. However, Existing testing frameworks cannot effectively construct the boundary values of SQL function arguments and detect SQL function bugs.

Conventional function testing approaches for programming languages generate literal values for arguments, but they are not suitable for generating SQL function test cases with boundary values. For example, traditional test methods like EvoSuite [17, 46] mainly focus on the generation of function call sequences for function testing. They only generate random values for the arguments of function calls, which need a hard time producing values that satisfy the boundary value of SQL function arguments.

Existing DBMS testing tools also struggle to generate test cases that include built-in SQL functions. Even when they do, it remains challenging to produce arguments with boundary values in these expressions. First, most of them generate literal values randomly, without considering the boundary values of SQL function expressions. Second, they focus on constructing syntactically and semantically correct statements but often ignore invalid type castings. Third, most of them generate complex SQL structures with various keywords and clauses, while neglecting nested function expressions.

To identify SQL function bugs caused by boundary values, we summarized 10 boundary-value-generation patterns of bug-inducing SQL queries based on the studied bugs, which can guide the detection of SQL function bugs.

Patterns of Boundary Literal Values. We found about 10.0% (32/318) of bugs result from extreme integer values or decimal values, 6.6% (21/318) arise from empty strings or the NULL value, and 12.9% (41/318) are caused by crafted string literals in certain formats (e.g. JSON and DATE).

Pattern 1.1. $bound \rightarrow$ ±0.999...99, ±999...99, `‘’`, `NULL`, `*`
Pattern 1.2. $f(c) \rightarrow f(bound)$
Pattern 1.3. $f(c) \rightarrow f(c$`[:i]` $+$ 99...999 $+\ c$`[i+1:]`$)$
Pattern 1.4. $f(c) \rightarrow f(c$`[:i]` $+\ c$`[i]`...c`[i]` $+\ c$`[i+1:]`$)$

Consequently, we established rules for generating boundary literal values. In SQL, there are several common types of literals: integers, decimals, strings, NULL, and the asterisk '*' (e.g., "`SELECT * FROM t`"). We construct the boundary values of these literal types by Pattern 1.1. Particularly for integer and decimal values, we enumerate values with different digit lengths. It should be noted that merely attempting extremely large values (e.g., 99...999 with 100 digits) is insufficient, as they might be rejected during the parsing stage due to excessive length. Moreover, different databases have varying maximum allowable decimal digits. Hence, enumerating values with different digit lengths is a more suitable approach to generating boundary literal values.

Once these common boundary literal values are generated, we can apply Pattern 1.2 to utilize them as function arguments in SQL function expressions, thereby generating corresponding SQL statements. For instance, assuming a DBMS supports a function named `‘f’` that requires a single argument, we utilize these boundary values as arguments to generate a series of new SQL statements following Pattern 1.2, such as `SELECT f(NULL)`, `SELECT f(*)`, `SELECT f(‘’)`, and `SELECT f(-0.99999)`.

Table 3. Literal examples generated by Patterns 1.3 and 1.4.

Data Type	Original Value	Generated Value by Pattern 1.3 & 1.4	
json	`'{"a":10}'`	`'{"a":1999}',`	`'{"a":10}}}'`
ip	`'{0.0.0.0}'`	`'999.0.0.0',`	`'0...0.0.0'`
hex	`'0x7f'`	`'0x999f',`	`'0x7fff'`
regex	`'(.){10}'`	`'(.){1999}',`	`'(((.){10}'`

Furthermore, certain functions may require arguments with specific formats of strings. The potential boundary values for such arguments can be roughly constructed by appending repeated characters via Patterns 1.3 and 1.4. For example, a function expression like `JSON('{"key": 0}')` can be utilized by this pattern to produce expressions like `JSON('{"key": 999990}')` and `JSON('{{{{{"key": 0}')`. The two patterns help to construct boundary values that satisfy certain argument formats.

Patterns of Boundary Type Castings. In the bugs caused by boundary type castings that we studied, crashes usually occur when processing SQL function expressions with these faulty instances. Some of the bug-inducing SQL statements can adhere to Pattern 2.1, which represents SQL function expressions with explicit type casting. SQL function arguments are first converted to other data types via the `CAST` operation, triggering various DBMS behaviors related to different internal data types. If there are errors in the type casting implementation, they may cause bugs when the DBMS processes SQL functions with these cast arguments.

Pattern 2.1. $f(c) \rightarrow f(\texttt{CAST}(c \texttt{ AS } type))$
Pattern 2.2. $f(c) \rightarrow f(\texttt{SELECT } c \texttt{ UNION SELECT } type())$
Pattern 2.3. $f(c), f_2(c_2) \rightarrow f(c_2)$

In contrast, Patterns 2.2 and 2.3 help detect errors of implicit type casting. Pattern 2.2 constructs the implicit type casting via the `UNION` operation. In SQL, the results of the same column require the same data type. Hence, performing a `UNION` operation between two `SELECT` statements can implicitly convert one type into another. Pattern 2.3 directly passes the arguments of one function expression to another function to create implicit type casting. This is because arguments for different functions may follow different formats and data types. Passing the arguments of other functions can make the formats and data types mismatch, resulting in implicit type casting. Pattern 1.2 for boundary literal values may also lead to implicit type casting since it can generate argument values of multiple different data types.

Patterns of Nested Functions. The structure of nested functions can be described using Pattern 3.1, Pattern 3.2, and Pattern 3.3. Pattern 3.1 is used to construct arguments of extreme lengths by repeating a prefix of the original argument via the `REPEAT` function. The repetition count is from the boundary literal values from Pattern 1.1. This pattern can also create deep recursive structures for arguments of specific data types, such as ARRAY, JSON, and XML.

Pattern 3.1. $f(c) \rightarrow f(\texttt{REPEAT}(c\texttt{[:i]}, bound))$
Pattern 3.2. $f(c),\ f_2(c_2) \rightarrow f(f_2(c))$
Pattern 3.3. $f(c),\ f_2(c_2) \rightarrow f(f_2(c_2))$

Patterns 3.2 and 3.3 employ other existing function expressions as function arguments. Pattern 3.2 retains the original arguments of the function but wraps it with another function. In contrast, Pattern 3.3 directly replaces the argument with the return value of another function expression. Both patterns have the potential to construct boundary argument values or data types through nested functions.

7 Pattern-Based Bug Detection

Guided by the boundary-value-generation patterns of SQL function bugs, we designed Soft that generates SQL statements based on these boundary argument patterns. We utilized Soft to test seven widely used DBMSs, and it discovered a total of 132 previously unknown SQL function bugs. All of these bugs were confirmed by DBMS developers, and 97 of them have been fixed.

7.1 Implementation

Soft is implemented in Python. It detects SQL function bugs in the following three steps:

Function Expression Collection. Soft initially acquires initial function expressions by scanning the documentation and regression test suite of the DBMS. First, Soft extracts all SQL function names from the documentation of the DBMS. Then, Soft collects the SQL function expressions from the SQL queries in the test suite of DBMS according to these function names. Specifically, since SQL function expressions typically follow the format `func(...)`, we scan all pairs of parentheses in SQL queries, and if the previous token of the pair is a function name, this function name and the pair of parentheses form a SQL function expression.

Pattern-Based Generation. Based on the boundary-value-generation patterns, Soft first generates a series of boundary literal values using Pattern 1.1, which are relied on by Patterns 1.2 and 3.1. Then, for Patterns 1.2, 1.3, 1.4, 2.1, 2.2, and 3.1, Soft enumerates the arguments of these function expressions, generates new arguments by applying the patterns, and replaces the original arguments to create new expressions. For Patterns 2.3, 3.2, and 3.3, Soft performs double enumeration on two function expressions $f(c)$ and $f_2(c_2)$, and generates new arguments and expressions based on the patterns. According to Finding 3, most function bugs can be triggered with no more than two nested functions. Therefore, when the number of functions in an expression is greater than two, Soft no longer generates new expressions for it.

SQL Function Bug Detection. After new function expressions are generated, Soft substitutes them for the original function expressions within the SQL statements. It then executes these statements in DBMSs via their Python clients. If the DBMS crashes during the execution, it indicates that

Table 4. Soft discovered 132 new vulnerabilities within two weeks (PostgreSQL: 1, MySQL: 16, MariaDB: 24, ClickHouse: 6, MonetDB: 19, DuckDB: 21, Virtuoso: 45). [**NPD**: Null Pointer Dereference, **SEGV**: Segmentation Violation, **UAF**: Use-after-Free, **BOF**: Buffer Overflow [Heap(H), Global(G)], **AF**: Assertion Failure, **SO**: Stack Overflow, **DBZ**: Divide-by-Zero]

DBMS	Function Type	Bug Type	Boundary-Value-Generation Patterns	Status
PostgreSQL	aggregate (1)	HBOF(1)	**P2.3**(1)	1 Confirmed, 1 Fixed
MySQL	aggregate (6)	NPD(4), SEGV(1), GBOF(1)	**P1.3**(1), **P3.3**(4), **P2.1**(1)	6 Confirmed
MySQL	date (1)	SEGV(1)	**P3.3**(1)	1 Confirmed
MySQL	spatial (1)	UAF(1)	**P3.3**(1)	1 Confirmed
MySQL	string (2)	HBOF(2)	**P3.2**(1), **P3.3**(1)	2 Confirmed
MySQL	system (5)	NPD(4), HBOF(1)	**P3.2**(1), **P3.3**(4)	5 Confirmed, 1 Fixed
MySQL	xml (1)	UAF(1)	**P3.2**(1)	1 Confirmed
MariaDB	aggregate (4)	NPD(1), SEGV(2), SO(1)	**P1.2**(3), **P2.2**(1)	4 Confirmed
MariaDB	condition (1)	NPD(1)	**P2.2**(1)	1 Confirmed
MariaDB	date (3)	NPD(2), GBOF(1)	**P1.2**(1), **P2.3**(1), **P3.3**(1)	3 Confirmed
MariaDB	json (6)	NPD(2), SEGV(1), AF(1), GBOF(2)	**P1.4**(2), **P2.3**(1), **P3.1**(2), **P3.3**(1)	6 Confirmed
MariaDB	sequence (1)	NPD(1)	**P3.3**(1)	1 Confirmed
MariaDB	spatial (5)	NPD(3), SEGV(1), SO(1)	**P3.2**(1), **P3.3**(4)	5 Confirmed, 3 Fixed
MariaDB	string (4)	NPD(2), HBOF(1), SO(1)	**P1.2**(2), **P3.1**(1), **P3.3**(1)	4 Confirmed, 1 Fixed
ClickHouse	aggregate (1)	NPD(1)	**P1.2**(1)	1 Confirmed & Fixed
ClickHouse	array (1)	NPD(1)	**P2.3**(1)	1 Confirmed & Fixed
ClickHouse	date (1)	NPD(1)	**P1.2**(1)	1 Confirmed & Fixed
ClickHouse	string (3)	NPD(1), SEGV(2)	**P1.2**(1), **P2.3**(1), **P3.1**(1)	3 Confirmed & Fixed
MonetDB	aggregate (7)	NPD(6), SEGV(1)	**P1.2**(1), **P2.1**(1), **P2.2**(2), **P2.3**(2), **P3.3**(1)	7 Confirmed & Fixed
MonetDB	condition (3)	NPD(2), SEGV(1)	**P2.2**(1), **P3.2**(1), **P3.3**(1)	3 Confirmed & Fixed
MonetDB	math (1)	NPD(1)	**P2.2**(1)	1 Confirmed & Fixed
MonetDB	string (6)	NPD(5), HBOF(1)	**P1.2**(1), **P1.3**(1), **P1.4**(1), **P2.3**(3)	6 Confirmed & Fixed
MonetDB	system (2)	SEGV(1), DBZ(1)	**P1.2**(1), **P2.3**(1)	2 Confirmed & Fixed
DuckDB	array (9)	AF(5), HBOF(3), SO(1)	**P1.2**(7), **P1.4**(1), **P2.2**(1)	9 Confirmed & Fixed
DuckDB	date (1)	SO(1)	**P3.1**(1)	1 Confirmed & Fixed
DuckDB	map (3)	AF(1), HBOF(2)	**P1.2**(2), **P2.1**(1)	3 Confirmed & Fixed
DuckDB	json (1)	AF(1)	**P1.2**(1)	1 Confirmed & Fixed
DuckDB	math (2)	AF(1), HBOF(1)	**P1.2**(1), **P2.1**(1)	2 Confirmed & Fixed
DuckDB	string (4)	AF(2), SEGV(2)	**P1.2**(1), **P1.3**(1), **P3.1**(1), **P3.3**(1)	4 Confirmed & Fixed
DuckDB	system (1)	AF(1)	**P2.1**(1)	1 Confirmed & Fixed
Virtuoso	aggregate (5)	NPD(4), SEGV(1)	**P1.2**(1), **P3.2**(1), **P3.3**(3)	5 Confirmed & Fixed
Virtuoso	casting (2)	AF(2)	**P1.2**(2)	2 Confirmed & Fixed
Virtuoso	condition (3)	NPD(2), SEGV(1)	**P3.3**(3)	3 Confirmed & Fixed
Virtuoso	math (5)	NPD(3), SEGV(1), DBZ(1)	**P1.2**(2), **P2.1**(1), **P2.2**(1), **P2.3**(1)	5 Confirmed & Fixed
Virtuoso	spatial (2)	NPD(1), SEGV(1)	**P1.2**(1), **P2.1**(1)	2 Confirmed & Fixed
Virtuoso	string (10)	NPD(2), SEGV(6), SO(1), UAF(1)	**P1.2**(5), **P2.3**(1), **P3.1**(3), **P3.2**(1)	10 Confirmed & Fixed
Virtuoso	xml (3)	NPD(3)	**P1.2**(3)	3 Confirmed & Fixed
Virtuoso	system (15)	NPD(8), SEGV(6), HBOF(1)	**P1.2**(11), **P3.1**(3), **P3.3**(1)	15 Confirmed & Fixed
Total	–	**132 Bugs**	**P1.x**(56), **P2.x**(28), **P3.x**(48)	132 Confirmed, 97 Fixed

an SQL function bug is encountered, and Soft logs the corresponding SQL statements for bug reporting.

7.2 Testing Setup

The experiments were conducted on a machine running 64-bit Ubuntu 20.04 with an AMD EPYC 7742 Processor @ 2.25 GHz, 128 cores, and 504 GiB of main memory. We use Soft to test the latest version of 7 open-source DBMSs, namely PostgreSQL v16.1 [48], MySQL v8.3.0 [45], MariaDB v11.3.2 [5], ClickHouse v23.6.2.18 [30], MonetDB v11.47.11 [40], DuckDB v0.10.1 [16], and Virtuoso v7.2.12 [42], which are widely used in industry. All DBMSs were tested using docker containers that were downloaded directly from their websites, each with 5 CPU cores and 50 GiB of RAM.

7.3 Detected DBMS Vulnerabilities

Soft successfully detected 132 vulnerabilities in two weeks. We also used the latest versions of Squirrel [63], SQLsmith [53], and SQLancer in PQS mode [51] with their default configurations to test these DBMSs, but they did not find any SQL function bugs. Table 4 shows Soft discovered 1, 16, 24, 6, 19, 21, and 45 bugs in PostgreSQL, MySQL, MariaDB, ClickHouse, MonetDB, DuckDB, and Virtuoso, respectively. It shows that the SQL function bugs found by Soft cover various types with different boundary-value-generation patterns. **Specifically, Soft detected 56, 28, and 48 SQL function bugs in patterns of literal values, type castings, and nested functions, respectively.** In particular, the bugs also cover different kinds of memory errors, including 61 null pointer dereferences, 29 segmentation violations, 12 heap buffer overflows, 4 global buffer overflows, 3 use-after-free, 7 stack overflows, 2 divide-by-zero, which can lead to

"We must fix it immediately or get rid of this function."
— CTO of ClickHouse

"These issues are security relevant and in some cases, we hide such issues from public access until the issue is closed."
— MariaDB Security Team

"Thank you for reporting a security related bug in PostgreSQL. However, since posts from this form are immediately public, we ask that you instead email your report to the security team."
— Moderator of pgsql-bugs Mailing List

Figure 2. Feedback from DBMS developers.

serious vulnerabilities in DBMSs. Soft also detected 14 assertion failures, which means the 10 patterns could help Soft to explore the unexpected states of the SQL functions. However, Soft triggered 7 false positives due to generating arguments exceeding memory limits like "`REPEAT('a', 9999999999)`", which made the DBMS terminate the SQL queries forcibly.

We also observed the few discovered bugs in PostgreSQL and fixes in MySQL. For PostgreSQL, it is because PostgreSQL's strict type system and rigorous argument checks reduce the number of boundary value-related bugs, though this means that PostgreSQL users need to write more expressions for explicit casting and formatting. For MySQL, the reason for few fixes is that the vendor only labels a bug as fixed until a new version of MySQL is released, which often takes several months after the bug report.

Feedback from DBMS Developers. Moreover, we have actively reported all 132 SQL function bugs to the corresponding DBMS vendors and received their confirmation feedback. At the time of the paper writing, 132 SQL function bugs have been confirmed, and 97 SQL function bugs have been fixed. More importantly, the detected bugs have garnered the attention of the DBMS developers, as illustrated in Figure 2. For example, the CTO of ClickHouse noticed our bug report and asked the developers to fix it immediately [21]. When we were reporting bugs to MariaDB's JIRA, the developers quickly hid these bugs [10] due to security reasons as shown in Figure 2. Our PostgreSQL bug was asked to report to PostgreSQL Security Team directly since it is a security-related bug [25].

7.4 Bugs of Each Pattern

To demonstrate the effectiveness of ten boundary patterns in detecting SQL function bugs, we analyze the bugs and present the following case studies.

Bugs Related to Boundary Literal Values. In our experiments, we found about 56 SQL function bugs related to the patterns of boundary literal values. The bugs of these patterns are usually fixed by adding additional checking code for the argument values in the SQL function implementation. To ensure system stability, we recommend that developers meticulously handle each argument, thoroughly examining their potential impact on the system's functionality.

Listing 6. A global buffer overflow in MySQL.

```
-- Case 1. global buffer overflow in MySQL
SELECT AVG(1.2999999999999999999999999999999999999
999999999999999999999999999999999999);
```

Case 1: A global buffer overflow in MySQL. Listing 6 shows a test case that triggers a global buffer overflow in MySQL. It is discovered by using patterns of boundary literal values. In this case, the bug is triggered by the `AVG` function when processing an excessively long literal value. The literal value 1.2999...999 exceeds the expected precision and causes a global buffer overflow in MySQL. This overflow occurs because MySQL cannot handle such an extended floating-point value, leading to memory corruption.

Listing 7. A segmentation violation in Virtuoso.

```
-- Case 2. segmentation violation in Virtuoso
SELECT CONTAINS( x , x , * );
```

Case 2: Segmentation violation in Virtuoso. Listing 7 presents a test case that triggers a segmentation violation in Virtuoso. It is found by using patterns of boundary literal values. In this instance, the `CONTAINS` function in Virtuoso is called with three arguments: '`x, x, *`'. The improper handling of these arguments, particularly the asterisk '*', leads to a segmentation violation in Virtuoso. It is because Virtuoso does not check for asterisk arguments in this SQL function's implementation, which causes illegal memory access in the subsequent processing of the SQL function.

Bugs Related to Boundary Type Castings. In our experiments, we found about 28 SQL function bugs related to the patterns of boundary type castings. The bugs within these patterns stem from both deficiencies in the type system and insufficient checks. To enhance the system reliability, we suggest that developers pay more attention to the design of the type system, for instance, considering whether certain member variables of internal data types are allowed to be null pointer values. Additionally, we recommend developers implement more thorough checks during both explicit and implicit type conversions.

Listing 8. A heap buffer overflow in PostgreSQL.

```
-- Case 3. heap buffer overflow in PostgreSQL
SELECT JSONB_OBJECT_AGG(DISTINCT 'a', 'abc');
```

Case 3: A heap buffer overflow in PostgreSQL (CVE-2023-5868) [25]. Listing 8 shows a test case that triggers a heap buffer overflow in PostgreSQL. It is discovered by using patterns of boundary type castings. In this case, the bug is triggered by the `JSONB_OBJECT_AGG` function in PostgreSQL when passing literal string values as arguments. The function attempts to aggregate JSON objects with the keys '`a`' and '`abc`', leading to a heap buffer overflow. This overflow happens because PostgreSQL incorrectly identifies the `unknown`-type arguments within aggregate functions as strings. Consequently, PostgreSQL improperly interprets the argument as being terminated with a '`\0`' character, resulting in memory disclosure when reading the `unknown`-type value as a string.

Listing 9. A stack overflow in DuckDB.

```
-- Case 4. stack overflow in DuckDB
SELECT REPEAT('[{"a":', 100000) UNION (SELECT [ ]);
```

Case 4: A stack overflow in DuckDB. Listing 9 presents a test case that triggers a stack overflow in DuckDB. It is found by using patterns of boundary type castings. This instance involves the REPEAT function in DuckDB, where the function is instructed to repeat the string '[{"a":' 100,000 times. This extensive repetition, combined with a UNION operation involving an empty array, causes a stack overflow. The stack overflow occurs due to excessive recursive calls and deep stack usage beyond the system's capacity to handle, leading to a crash in DuckDB. This reveals a critical flaw in the function's handling of large or deeply nested input, exposing vulnerabilities in the system's robustness and stability.

Bugs Related to Boundary Results of Nested Functions. In our experiments, we found about 48 SQL function bugs related to the patterns of boundary results of nested functions. These bugs are fixed by additionally checking the data types and values returned from other SQL functions. We recommend that developers ensure that the handling of arguments in each SQL function adequately addresses the possible return values of other functions, especially those with extreme lengths or special internal data types.

Listing 10. A global buffer overflow in MariaDB.

```sql
-- Case 5. global buffer overflow in MariaDB
SELECT JSON_LENGTH(REPEAT('[1,', 100), '$[2][1]');
```

Case 5: A global buffer overflow in MariaDB. Listing 10 presents a test case that triggers a global buffer overflow in MariaDB. It is discovered by using patterns of boundary results of nested functions. In this case, the bug is triggered by the JSON_LENGTH function in MariaDB. The function is called with a JSON path '$[2][1]' on a string generated by repeating '[1,' 100 times. This extensive repetition leads to an excessively large JSON string, causing a global buffer overflow. The overflow occurs because the system fails to properly allocate and manage memory for the input with such large nested array expressions, resulting in memory corruption and potential system crashes.

Listing 11. A segmentation violation in MariaDB.

```sql
-- Case 6. Segmentation violation in MariaDB
SELECT ST_ASTEXT(
         BOUNDARY(INET6_ATON('255.255.255.255')));
```

Case 6: Segmentation violation in MariaDB. Listing 11 shows a test case that triggers a segmentation violation in MariaDB. It is found by using patterns of boundary results of nested functions. This instance involves a segmentation violation in MariaDB triggered by a combination of the ST_ASTEXT, BOUNDARY, and INET6_ATON functions. The function INET6_ATON converts the IP address '255.255.255.255' to its binary form, which is then passed to the BOUNDARY function and subsequently to ST_ASTEXT. This sequence of operations causes a segmentation violation, likely due to improper handling or invalid memory access during the type casting and boundary calculation processes, leading to a crash in MariaDB. This reveals vulnerabilities in the system's handling of complex function chains and type castings.

7.5 Comparison with Other Testing Works

To demonstrate the effectiveness of our methods, we compared Soft against three state-of-the-art DBMS testing tools, namely Squirrel, SQLancer, and SQLsmith, which are widely used in the industry. Among the DBMSs we tested, Squirrel supports PostgreSQL, MySQL, and MariaDB; SQLsmith supports PostgreSQL and MonetDB; while SQLancer supports PostgreSQL, MySQL, MariaDB, and ClickHouse.

We evaluated these DBMS testing tools based on the number of covered SQL functions and code branches of built-in SQL function components in 24 hours. We also countered the unique SQL function bugs discovered by these tools in 24 hours. For a fair comparison, when we finished DBMS testing, we collected the queries generated by each DBMS tool and reran all the queries to uniform the branch coverage.

Table 5. Number of triggered built-in SQL functions by generated SQL statements in 24 hours.

DBMS	Squirrel	SQLancer	SQLsmith	Soft
PostgreSQL	29	123	417	456
MySQL	23	35	–	323
MariaDB	22	20	–	279
ClickHouse	–	24	–	711
MonetDB	–	–	29	171
Total	74	202	446	2,956
Increment*	984	1,567	181	–

* Increments are calculated only for commonly supported DBMSs.

Covered Functions and Related Code Branches. Tables 5 and 6 show the number of SQL functions and code branches of these SQL function modules covered by those DBMS testing tools over 24-hour testing. The results indicate that Soft outperformed other DBMS testing tools. Specifically, Soft covered 984, 1567, and 181 more SQL functions, as well as 433.93%, 98.70%, and 19.86% more branches in built-in SQL function components than Squirrel, SQLancer, and SQLsmith, respectively. The main reason for the improvement of Soft is the boundary-value-generation patterns utilized. These generation patterns enable Soft to construct boundary argument values and function expressions for a wide range of SQL functions. These boundary arguments help trigger the deep logic when DBMS processes function expressions, leading to higher branch coverage. In contrast, other tools lack a universal method for generating function expressions and their boundary arguments. For example, SQLancer requires writing function models in Java code to support the generation of a new function, and it only supports generating random values for SQL function arguments. The complexity of adaptation and the lack of boundary argument generation limit its ability on SQL function testing.

Table 6. Number of covered code branches of DBMSs' built-in SQL function modules in 24 hours.

DBMS	Squirrel	SQLancer	SQLsmith	Soft
PostgreSQL	2,106	6,106	11,768	13,334
MySQL	1,105	1,927	–	6,914
MariaDB	1,758	1,732	–	6,283
ClickHouse	–	26,655	–	45,836
MonetDB	–	–	551	1,431
Total	4,969	36,420	12,319	73,798
Increment*	21,562	35,947	2,446	–

* Increments are calculated only for commonly supported DBMSs.

Triggered SQL Function Bugs. Squirrel, SQLancer, and SQLsmith did not find any SQL function bugs in 24 hours. In contrast, Soft triggered 22 unique SQL function bugs within 24 hours, including 1, 5, 6, 3, and 7 bugs in PostgreSQL, MySQL, MariaDB, ClickHouse, and MonetDB, respectively. Unlike Squirrel, SQLancer, and SQLsmith, our tool Soft specifically targets boundary values of SQL function arguments. By focusing on boundary literal values, type castings, and nested function return values, Soft is able to identify and test edge cases that are often the root cause of bugs. Besides, Soft utilizes a pattern-based generation technique derived from our study of 318 SQL function bugs. These patterns provide a structured methodology for creating test cases that are more likely to uncover hidden bugs.

8 Discussion

UDF Bugs. This paper only exploits vulnerabilities in the built-in SQL functions. However, user-defined functions of DBMS can also be problematic to a large extent. In contrast to built-in SQL functions, the user-defined functions are typically written with PL/SQL statements, so the DBMS's handling of user-defined functions focuses on parsing and executing these statements. Therefore, the testing of DBMS user-defined functions can be summarized in the testing of the database's PL/SQL system. We will focus on this part in the future to analyze and test the user-defined functions of DBMSs as well.

Correctness Bugs in SQL Functions. In this paper, we mainly focus on memory safety problems in built-in SQL functions. However, it is still possible for a SQL function to have correctness problems (or logic problems), where functions return incorrect results. To address these issues, we can extend the existing testing frameworks to construct test guidelines that include function expressions with equivalent semantics and perform differential testing. Equivalent semantic function expressions can be constructed by replacing intermediate clauses in existing nested structures, e.g., intermediate result replacement, where clause splitting (e.g., TLP [50]) and transformation (e.g., NoREC [49]).

Reality of Found Bugs. In this paper, we found that the core cause of the built-in function problem is the poor handling of boundary values, so we test the built-in function by generating literals, type conversions, and nested structures to get different boundary values. In the real world, software systems often need to process a wide variety of input data, including those under various boundary conditions. These boundary conditions might be triggered due to user input, interactions with external systems, or internal data processing errors. Moreover, once these bugs occur, they can lead to application crashes and serious security risks. Among the 132 bugs we reported, all were confirmed by developers, and 97 were fixed within three days.

Extending Existing DBMS Testing Works with Soft. While Soft focuses on detecting SQL function bugs, its boundary-value-generation patterns can also help discover other DBMS bugs. For example, DBMS have some data-sensitive operations, such as indexing, sorting, and filtering, which are performed by the `CREATE INDEX`, `ORDER BY`, and `WHERE` clauses, respectively. We can test these DBMS operations more thoroughly with Soft by generating boundary values for various data types. Furthermore, we can integrate Soft into existing grammar-based DBMS testing frameworks to enhance their bug detection ability. For instance, grammar-based works can first construct syntactically correct structures of different SQL statements, and then Soft fills in the custom values of each clause of these SQL statements. In this way, Soft can help existing works to trigger more boundary behaviors in scenarios other than SQL functions.

9 Related Work

DBMS Testing. The efforts in DBMS testing can be broadly categorized into three main areas: testing for crash bugs, identifying correctness (or logic) bugs, and evaluating performance issues. Crash bug testing [18–20, 33, 36, 53, 57, 63] focuses on identifying conditions under which the DBMS might unexpectedly stop working or crash. These works typically concentrate on generating valid SQL statements and then monitoring whether the DBMS under test crashes. For example, SQLsmith [53] is a generation-based fuzzer to generate SQL queries for testing. Squirrel [63] proposes an IR-based mutation method to generate queries with syntactical and semantical correctness. Lego [33] uses type-affinity analysis to generate queries with more SQL type sequences to explore the SQL state space. Griffin [19] proposes a grammar-free fuzzing method for query generation.

DBMS correctness testing [15, 49–51, 54] aims to verify that the DBMS accurately executes queries. The state-of-the-art works focus on constructing effective logic testing oracles. The mainstream approach involves adopting metamorphic testing and differential testing. For example, PQS [51] detects whether the pivot row exists in the preset query results. NoREC [49] detects inconsistencies between query results before and after optimization. Mozi [37] utilizes equivalent configuration transformation to test whether the result matches.

DBMS performance testing [32, 39, 60] assesses the performance under different workloads and conditions. Apollo [32] utilizes performance regression testing to find errors in the evolution of code during development. Amoeba [39] generates semantically equivalent queries and compares their response times to identify performance issues.

Soft aims to find crash bugs in built-in SQL functions. Unlike traditional DBMS testing works, Soft specifically focuses on built-in SQL function bugs, which are mainly caused by poor handling of boundary values. Consequently, it aims to generate queries with nested SQL functions or type conversions to trigger bugs. In addition, it generates only one SQL statement with functions per case, and its complexity is reflected in the SQL functions. Traditional testing work in a test case generally has multiple statements, and complexity is reflected in the SQL clauses.

Testing of Function Libraries. Traditional tests for function libraries include unit tests and fuzzing. Unit testing [17, 46, 52] involves writing and executing tests for individual functions or routines within the library to verify that each performs as expected in isolation. Tools like Evosuite and Randoop [17, 46] can generate unit tests for Java libraries with rich function sequences and provide assertions for correctness. These methods allow developers to check the correctness of each function, including edge cases and error-handling paths. Fuzzing [2, 35, 38, 47, 56, 62] involves automatically generating a wide range of inputs to test the function library's robustness and error-handling capabilities. libFuzzer [38] provides a standardized fuzzing interface, LLVMFuzzerTestOneInput, which is used to call the function and construct a fuzzing driver for testing the function. AFL [62] is one of the most popular fuzzers and can use the interface to test functions. Many fuzzers improve test coverage through various techniques by combining it with static analysis, taint analysis [2, 34], or symbolic execution [47, 61].

Unlike them, Soft is concerned with testing built-in SQL functions. It specializes in generating test cases that satisfy the SQL syntax requirements and boosts the complexity of built-in SQL functions as much as possible to trigger boundary values for testing. Nevertheless, Soft's idea of triggering boundary arguments can also be applied to traditional approaches to testing functions.

Domain Testing. Domain testing is a software testing technique that generates a set of test cases to detect potential errors on boundary conditions. It partitions input space into multiple path domains and computes the boundaries of these domains. It then detects the incorrect path domain (i.e., domain errors, such as incorrect branch predicate expressions) by generating test inputs on the boundaries. White et al. [58] first proposed domain testing and designed a test data selection strategy, and Clarke et al. [4] further introduced two alternative domain testing strategies to improve on the error bound. Nevertheless, these domain testing strategies are white-box testing methods depending on the analysis of control flows and execution paths of programs. It is difficult to apply these techniques to large systems like DBMSs.

In contrast, Soft tests SQL functions by constructing the boundary conditions studied from existing SQL function bugs. We found that the bug-inducing boundary conditions mainly come from the boundary literals, boundary type castings, and boundary results from nested functions. We summarized them into ten patterns to trigger the boundary conditions of SQL functions and detect bugs.

Robustness Testing. Robustness testing is essential in ensuring the security and reliability of a system. It usually involves introducing unexpected events to the system, such as fault injections or invalid inputs [1, 3, 13, 14, 29, 41]. AndroFIT [14] designed a vast set of faults of different components in Android OS and implemented these faults in automated fault injection tools to detect their reliability issues. Postmonkey [1] sends invalid, delaying, and random messages to inject faults in distributed embedded systems with low computation power. ASTAA [29] tests autonomy systems by generating invalid values into the system message fields to construct the test cases in XML formats. Since different systems or components can have unique architectures, the design of robustness testing approaches for a specific system relies on certain characteristics of the system itself.

Soft can be considered as robustness testing targeted at the SQL function components of DBMSs. We studied some characteristics of SQL function bugs and summarized 10 boundary-value-generation patterns of these bugs. Guided by these bug-inducing patterns, we designed the frameworks of Soft to discover SQL function bugs, which can help developers improve the robustness of DBMSs.

10 Conclusion

In this paper, we have presented an in-depth analysis of 318 built-in SQL function bugs across three DBMSs. Our investigation reveals that a significant majority (87.4%) of these bugs are attributed to improper handling of boundary values in function arguments. These boundary values originate from boundary literal values, type castings, and nested function return values. To address these issues, we developed Soft, a tool that leverages the patterns identified in our study to generate SQL test cases that effectively target these boundary conditions. By employing Soft across seven widely used DBMSs, we discovered 132 previously unknown bugs, all of which have been confirmed.

Acknowledgments

We thank the shepherd and reviewers for their valuable comments. This research is partly sponsored by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. 62302256, 92167101, 62021002), and Chinese Postdoctoral Science Foundation (2023M731953).

References

[1] Khaled Alnawasreh, Patrizio Pelliccione, Zhenxiao Hao, Mårten Rånge, and Antonia Bertolino. 2017. Online robustness testing of distributed embedded systems: An industrial approach. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 133–142.

[2] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (S&P)*.

[3] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Ming Gu, Qing Liao, and Yu Jiang. 2024. Chronos: Finding Timeout Bugs in Practical Distributed Systems by Deep-Priority Fuzzing with Transient Delay. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1939–1955.

[4] Lori A. Clarke, Johnette Hassell, and Debra J. Richardson. 1982. A close look at domain testing. *IEEE Transactions on Software Engineering* 4 (1982), 380–390.

[5] MariaDB Corporation. 2024. MariaDB. https://mariadb.org/. Accessed: September 23, 2024.

[6] MariaDB Corporation. 2024. MariaDB's JIRA. https://jira.mariadb.org. Accessed: September 23, 2024.

[7] MariaDB Corporation. 2024. MDEV-11030. https://jira.mariadb.org/browse/MDEV-11030. Accessed: September 23, 2024.

[8] MariaDB Corporation. 2024. MDEV-14596. https://jira.mariadb.org/browse/MDEV-14596. Accessed: September 23, 2024.

[9] MariaDB Corporation. 2024. MDEV-23415. https://jira.mariadb.org/browse/MDEV-23415. Accessed: September 23, 2024.

[10] MariaDB Corporation. 2024. MDEV-32315. https://jira.mariadb.org/browse/MDEV-32315. Accessed: September 23, 2024.

[11] MariaDB Corporation. 2024. MDEV-8407. https://jira.mariadb.org/browse/MDEV-8407. Accessed: September 23, 2024.

[12] Oracle Corporation. 2024. MySQL Bug System. https://bugs.mysql.com/. Accessed: September 23, 2024.

[13] Domenico Cotroneo, Domenico Di Leo, Francesco Fucci, and Roberto Natella. 2013. Sabrine: State-based robustness testing of operating systems. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 125–135.

[14] Domenico Cotroneo, Antonio Ken Iannillo, Roberto Natella, and Stefano Rosiello. 2019. Dependability assessment of the Android OS through fault injection. *IEEE Transactions on Reliability* 70, 1 (2019), 346–361.

[15] Wenqian Deng, Jie Liang, Zhiyong Wu, Jigzhou Fu, Mingzhe Wang, and Yu Jiang. 2025. CONI: Detecting Database Connector Bugs via State-Aware Test Case Generation. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. 1–12.

[16] DuckDB 2024. DuckDB Website. https://duckdb.org/. Accessed: September 23, 2024.

[17] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[18] Jingzhou Fu, Jie Liang, Zhiyong Wu, and Yu Jiang. 2024. Sedar: Obtaining High-Quality Seeds for DBMS Fuzzing via Cross-DBMS SQL Transfer. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[19] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-free DBMS fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[20] Ying Fu, Zhiyong Wu, Yuanliang Zhang, Jie Liang, Jingzhou Fu, Yu Jiang, Shanshan Li, and Xiangke Liao. 2025. THANOS: DBMS Bug Detection via Storage Engine Rotation Based Differential Testing. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. 1–12.

[21] GitHub. 2024. A Bug of DecimalString Function. https://github.com/ClickHouse/ClickHouse/issues/52407. Accessed: September 23, 2024.

[22] GitHub. 2024. Decimal in MariaDB. https://mariadb.com/kb/en/decimal/. Accessed: September 23, 2024.

[23] The PostgreSQL Global Development Group. 2024. Chapter 9. Functions and Operators. https://www.postgresql.org/docs/current/functions.html. Accessed: September 23, 2024.

[24] The PostgreSQL Global Development Group. 2024. PostgreSQL Bug Report Mailing List. https://www.postgresql.org/list/pgsql-bugs/. Accessed: September 23, 2024.

[25] The PostgreSQL Global Development Group. 2024. PostgreSQL Security Information. https://www.postgresql.org/support/security/. Accessed: September 23, 2024.

[26] The PostgreSQL Global Development Group. 2024. Unchecked JSON Input Can Crash the Server. https://www.postgresql.org/support/security/CVE-2015-5289/. Accessed: September 23, 2024.

[27] The PostgreSQL Global Development Group. 2024. Unchecked Regex Can Crash the Server. https://www.postgresql.org/support/security/CVE-2016-0773/. Accessed: September 23, 2024.

[28] The PostgreSQL Global Development Group. 2024. User-Defined Functions. https://www.postgresql.org/docs/current/xfunc.html. Accessed: September 23, 2024.

[29] Casidhe Hutchison, Milda Zizyte, Patrick E Lanigan, David Guttendorf, Michael Wagner, Claire Le Goues, and Philip Koopman. 2018. Robustness testing of autonomy software. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. 276–285.

[30] ClickHouse Inc. 2024. ClickHouse Website. https://clickhouse.com/. Accessed: September 23, 2024.

[31] ClickHouse Inc. 2024. Type Conversion Functions. https://clickhouse.com/docs/en/sql-reference/functions/type-conversion-functions. Accessed: September 23, 2024.

[32] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2020. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*. Tokyo, Japan.

[33] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. 2023. Sequence-oriented DBMS fuzzing. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*.

[34] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. Pata: Fuzzing with path aware taint analysis. In *2022 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1–17.

[35] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Jianzhong Liu, and Yu Jiang. 2024. Dodrio: Parallelizing Taint Analysis Based Fuzzing via Redundancy-Free Scheduling. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 244–254.

[36] Jie Liang, Zhiyong Wu, Jingzhou Fu, Yiyuan Bai, Qiang Zhang, and Yu Jiang. 2024. WingFuzz: Implementing Continuous Fuzzing for DBMSs. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 479–492.

[37] Jie Liang, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Chengnian Sun, and Yu Jiang. 2024. Mozi: Discovering DBMS Bugs via Configuration-Based Equivalent Transformation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*.

[38] LibFuzzer 2024. LibFuzzer. https://www.llvm.org/docs/LibFuzzer.html. Accessed: September 23, 2024.

[39] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic detection of performance bugs in database systems using equivalent queries. In *Proceedings of the 44th International Conference on Software Engineering*. 225–236.

[40] MonetDB 2024. The Database System to speed up your Analytical Jobs. https://www.monetdb.org/. Accessed: September 23, 2024.

[41] Roberto Natella, Stefan Winter, Domenico Cotroneo, and Neeraj Suri. 2018. Analyzing the effects of bugs on software interfaces. *IEEE*

Transactions on Software Engineering 46, 3 (2018), 280–301.

[42] Openlink. 2024. Virtuoso Open-Source Edition. https://vos.openlinksw.com/owiki/wiki/VOS. Accessed: September 23, 2024.

[43] Oracle. 2024. Chapter 6 Adding Functions to MySQL. https://dev.mysql.com/doc/extending-mysql/8.3/en/adding-functions.html. Accessed: September 23, 2024.

[44] Oracle. 2024. Functions and Operators. https://dev.mysql.com/doc/refman/8.0/en/functions.html. Accessed: September 23, 2024.

[45] Oracle 2024. MySQL. https://www.mysql.com/. Accessed: September 23, 2024.

[46] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[47] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *USENIX Security Symposium*.

[48] PostgreSQL 2024. PostgreSQL. https://www.postgresql.org/. Accessed: September 23, 2024.

[49] Manuel Rigger and Zhendong Su. 2020. 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*. 1140–1152.

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

[51] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation OSDI 20)*. 667–682.

[52] Per Runeson. 2006. A survey of unit testing practices. *IEEE software* 23, 4 (2006), 22–29.

[53] Andreas Seltenreich, Bo Tang, and Sjoerd Mullender. 2018. SQLsmith: a random SQL query generator. https://github.com/anse1/sqlsmith

[54] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, New York, USA*. Morgan Kaufmann, 618–622.

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

[56] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Jingzhou Fu, Zhuo Su, Qing Liao, Bin Gu, Bodong Wu, and Yu Jiang. 2024. Data Coverage for Guided Fuzzing. In *33rd USENIX Security Symposium (USENIX Security 24)*. 2511–2526.

[57] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 328–337. https://doi.org/10.1109/ICSE-SEIP52600.2021.00042

[58] Lee J White and Edward I Cohen. 1980. A domain strategy for computer program testing. *IEEE transactions on software engineering* 3 (1980), 247–257.

[59] Wikipedia 2024. databases. https://en.wikipedia.org/wiki/Database. Accessed: September 23, 2024.

[60] Zhiyong Wu, Jie Liang, Jingzhou Fu, Mingzhe Wang, and Yu Jiang. 2025. PUPPY: Finding Performance Degradation Bugs in DBMSs via Limited-Optimization Plan Construction. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. 1–12.

[61] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium.*

[62] Michał Zalewski. 2017. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: September 23, 2024.

[63] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: Testing Database Management Systems with Language Validity and Coverage Feedback. In *The ACM Conference on Computer and Communications Security (CCS), 2020.*