DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation

Zu-Ming Jiang, *ETH Zurich;* Jia-Ju Bai, *Tsinghua University;* Zhendong Su, *ETH Zurich*

https://www.usenix.org/conference/usenixsecurity23/presentation/jiang-zu-ming

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation

Zu-Ming Jiang
ETH Zurich

Jia-Ju Bai
Tsinghua University

Zhendong Su
ETH Zurich

Abstract

Database management systems (DBMSs) are essential parts of modern software. To ensure the security of DBMSs, recent approaches apply fuzzing to testing DBMSs by automatically generating SQL queries. However, existing DBMS fuzzers are limited in generating complex and valid queries, as they heavily rely on their predefined grammar models and fixed knowledge about DBMSs, but do not capture DBMS-specific state information. As a result, these approaches miss many deep bugs in DBMSs.

In this paper, we propose a novel stateful fuzzing approach to effectively test DBMSs and find deep bugs. Our basic idea is that after DBMSs process each SQL statement, there is useful state information that can be dynamically collected to facilitate later query generation. Based on this idea, our approach performs dynamic query interaction to incrementally generate complex and valid SQL queries, using the captured state information. To further improve the validity of generated queries, our approach uses the error status of query processing to filter out invalid test cases. We implement our approach as a fully automatic fuzzing framework, DynSQL. DynSQL is evaluated on 6 widely-used DBMSs (including SQLite, MySQL, MariaDB, PostgreSQL, MonetDB, and ClickHouse) and finds 40 unique bugs. Among these bugs, 38 have been confirmed, 21 have been fixed, and 19 have been assigned with CVE IDs. In our evaluation, DynSQL outperforms other state-of-the-art DBMS fuzzers, achieving 41% higher code coverage and finding many bugs missed by other fuzzers.

1 Introduction

Database management systems (DBMSs) play essential roles in modern data-intensive applications [13, 22, 44], providing fundamental functionalities of data storage and management. Due to the large code size and complex logic of DBMSs, bugs are inevitably introduced during development and maintenance. By exploiting DBMS bugs, attackers can introduce malicious threats of paralyzing the system [1, 5, 7] or even hacking secret data [14, 46].

Fuzzing is a promising technique for bug detection [2, 4, 15, 17, 26, 30, 47], and it is applied to testing DBMSs [18, 43, 45, 48] by generating SQL (Structured Query Language) queries that contain a series of SQL statements [48]. Specifically, some fuzzers [18, 43] utilize well-defined rules to randomly generate SQL queries, feed these queries to target DBMSs, and check whether bugs are triggered. To improve bug detection in DBMSs, several approaches [45, 48] further involve feedback mechanisms. After the execution of each test case, they collect the runtime information (e.g. code coverage) of target DBMSs and check whether interesting behavior (e.g. covering new branches) occurs. If so, the test case will be stored as a seed for later test-case generation.

However, existing DBMS fuzzers are still limited in generating complex and valid queries to find deep bugs in DBMSs. In general, a complex query contains multiple SQL statements that involve various SQL features (e.g. multi-level nested subqueries), while a valid query satisfies the dependencies among its statements (e.g., a subsequent statement references the elements defined in an earlier statement) and guarantees both syntactic and semantic correctness. Existing DBMS fuzzers always make a trade-off between complexity and validity of generated queries. For example, SQLsmith [43] generates only one statement in each query, avoiding the analysis of dependencies among statements, which sacrifices complexity for validity; SQUIRREL [48] uses an intermediate representation (IR) model to infer dependencies and generate queries that contain multiple statements, but it produces over 50% invalid queries and tends to generate simple statements.

The contradiction between query complexity and query validity occurs because existing DBMS fuzzers heavily rely on their predefined grammar models and fixed knowledge about DBMSs, but do not capture runtime state information. They either neglect state changes (e.g. SQLsmith [43]), or statically infer the corresponding states (e.g. SQUIRREL [48]) but suffer from soundness and completeness issues. Without accurate state information, these fuzzers tend to build incorrect dependencies among statements or misuse SQL features, causing many invalid queries to be generated. To generate valid test

Table 1: Conceptual comparison of DBMS fuzzers

Features	SQLsmith	SQUIRREL	DynSQL
Stateful Fuzzing	None	Partial	Full
Query Generation	Static	Static	Dynamic
Program Feedback	None	Code Cov	Code Cov+Error
Query Validity	High	Middle	High
Statement Number	One	Multiple	Multiple
Statement Complexity	High	Low	High

cases, these fuzzers have to limit the complexity of generated queries to tolerate their inaccurate state information.

In fact, DBMSs process each query statement by statement, and the states of manipulated databases change after each statement is executed. In the interval of statement processing, DBMS-specific state information, including the latest database schema and status of statement processing, is available. However, existing DBMS fuzzers fail to capture such information, as their query generation is finished before query execution. To solve this problem, we propose a novel stateful fuzzing approach to perform *dynamic query interaction* that merges query generation and query execution. This approach feeds each generated statement to the target DBMS and then dynamically interacts with the DBMS to collect the latest state information after the statement is executed. The collected state information is used to guide the generation of subsequent statements. Benefiting from dynamic query interaction, our fuzzing approach transforms the complicated process of query generation into several simple and independent processes of statement generation, and thus can generate complex and valid SQL queries effectively.

In addition, to further improve the validity of the generated queries, our fuzzing approach uses *error feedback* to guide test-case generation with code coverage. It collects the information on query execution and observes whether the generated queries pass syntactic and semantic checks of the target DBMS. If the generated queries trigger any syntactic or semantic error, these queries are identified to be invalid and are discarded directly. By using error feedback, our fuzzing approach guarantees that all selected seeds are valid, which is useful to generate valid test cases in subsequent mutations.

We implement our approach as a stateful DBMS fuzzing framework, DynSQL. Table 1 concludes the conceptual differences between DynSQL and two state-of-the-art DBMS fuzzers, according to their design and our evaluation results. Both DynSQL and SQUIRREL are aware of the state changes caused by generated statements. However, SQUIRREL tends to infer incorrect state information when it processes the semantics of complex statements. DynSQL addresses these problems by dynamically capturing the latest state information for query generation. Besides the feedback of code coverage used by SQUIRREL, DynSQL also uses error information to improve the validity of generated queries. Benefiting from these improvements, DynSQL can effectively generate valid queries containing multiple complex statements.

Overall, we make the following technical contributions:

- We propose a novel stateful fuzzing approach to address the limitations of existing DBMS fuzzers. Our approach performs dynamic query interaction that merges query generation and query execution to effectively generate complex and valid SQL queries. In addition, it uses error feedback to improve the validity of the generated queries.
- Based on our approach, we implement DynSQL, a practical DBMS fuzzing framework that automatically detects deep bugs in DBMSs, by generating complex and valid queries.
- We evaluate DynSQL on 6 widely-used DBMSs, including SQLite, MySQL, MariaDB, PostgreSQL, MonetDB, and ClickHouse. DynSQL finds 40 unique bugs among which 38 have been confirmed, 21 have been fixed, and 19 have been assigned CVE IDs. We compare DynSQL to state-of-the-art DBMS fuzzers, including SQLsmith and SQUIRREL. Owing to its effectiveness in generating complex and valid SQL queries, DynSQL achieves 41% higher code coverage and finds many bugs missed by other fuzzers.

2 Background and Motivation

In this section, we first introduce how DBMSs process SQL queries in brief, then illustrate the difficulty of generating complex and valid queries, and finally reveal the limitations of existing DBMS fuzzers.

SQL processing in DBMS. SQL queries are designed to perform the communication between users and DBMSs. To manage data, users often integrate multiple SQL statements (e.g., `SELECT` statement) into a query, and then send the query to the DBMS, which manipulates databases. After receiving a query, the DBMS first decomposes it into several statements and then processes the statements in sequence.

DBMSs generally process each statement in four phases: parsing, optimization, evaluation, and execution [38]. In the parsing phase, DBMSs first check the syntactic correctness of the statement according to their predefined grammar rules and then check the semantic correctness according to the current database schema. If any syntactic or semantic check fails, the statement is discarded directly, and the whole query processing may be terminated. In the later phases, DBMSs optimize the low-level expression of the statement and generate several possible execution plans. Then, DBMSs evaluate the cost of each execution plan and finally execute the most efficient plan. After the statement is executed, DBMSs update the states including database schema and the execution status, and then DBMSs process the following statement in the query.

Query generation. On the one hand, DBMS fuzzers should guarantee both syntactic and semantic correctness of the generated queries, so that these queries can pass validation checks without being discarded in earlier stages. However, doing so is difficult, as DBMS fuzzers not only need to obey specific SQL features and grammars, but also have to analyze possible

```
CREATE TABLE t1 (f1 INTEGER);
CREATE VIEW v1 AS
   SELECT subq_0.c4 AS c2, subq_0.c4 AS c4
   FROM (
      SELECT ref_0.f1 AS c4
      FROM t1 AS ref_0
      WHERE (SELECT 1)
   ) AS subq_0
   ORDER BY c2, c4 DESC;
WITH cte_0 AS (
   SELECT subq_0.c4 as c6
   FROM (
      SELECT 11 AS c4
      FROM v1 AS ref_0
   ) AS subq_0
   CROSS JOIN v1 AS ref_2)
SELECT 1;
```

Figure 1: A malicious query that crashes the MariaDB server.

DBMS state changes caused by the generated statements so as to precisely build statement dependencies and correctly reference attributes. On the other hand, to explore infrequent states, DBMS fuzzers should generate complex SQL queries to trigger deep logic of optimization, evaluation, and execution in DBMSs. However, increasing query complexity significantly increases the difficulty of guaranteeing query validity. Thus, it is important but very challenging to generate both complex and valid SQL queries to detect deep bugs in DBMSs.

Figure 1 shows a malicious query that crashes the MariaDB server and enables denial-of-service (DoS) attacks. This vulnerability affects a wide range of versions (10.2-10.5) of the MariaDB server and has existed for over 5 years until DynSQL found it. The query triggering this bug has been simplified by us and developers, and it is considered as the minimal test case. However, this query is still complex in structures and semantics. It contains three SQL statements. The first one is a simple `CREATE TABLE` statement that creates a table `t1`. The second statement is a `CREATE VIEW` statement that involves three-level nested sub `SELECT` statements to create a view `v1`. The sub statement at the first level uses a sub `SELECT` statement in its `FROM` clause. The sub statement at the second level references the created table `t1` and uses a sub `SELECT` statement again in its `WHERE` clause. The last statement in the query is a simple `SELECT` statement with a complex `COMMON TABLE EXPRESSION` (`CTE`). The `CTE` uses a two-level sub `SELECT` statement. The sub statement at the first level uses a sub `SELECT` statement in its `FROM` clause, and it is joined with the created view `v1` that is referenced again in the sub `SELECT` statement at the second level.

In fact, generating this query is difficult in DBMS fuzzing. First, this query contains multiple statements that cause DBMS state changes. Fuzzers need to capture such changes so that the subsequent statements can correctly reference the elements built by the earlier statements. Second, these statements utilize various SQL features, such as sub `SELECT` statement, `CTE`, `CROSS JOIN`, etc, which make it hard for fuzzers to accurately infer possible state changes.

Limitations of existing DBMS fuzzers. Existing DBMS fuzzers are limited in generating complex and valid SQL queries (e.g. the malicious query in Figure 1), because they cannot accurately capture the DBMS state changes caused by generated statements. Instead, they either generate only one complex statement in each query to avoid the analysis of state changes [18, 43], or combine multiple relatively simple statements where the state changes can be easily inferred [48].

SQLsmith [43], a popular grammar-based DBMS fuzzer, can generate complex SQL statements using its well-defined abstract syntax tree (AST) rules; but it is stateless without considering state changes caused by its generated statements, as a result of which it cannot build the dependencies among multiple statements and thus generates only one statement in each query. Using an intermediate representation (IR) to maintain query structures, SQUIRREL [48] is aware of state changes caused by its generated statements. It considers various SQL features and maintains the scopes and lives of multiple variables in statements, causing its IR mechanism to be complicated and error-prone when inferring state changes caused by complex statements. To mitigate this impact, SQUIRREL tends to generate simple statements in queries. Even so, SQUIRREL still generates over 50% invalid queries [48].

Without accurate DBMS state information, existing fuzzers are limited in generating complex and valid queries to extensively test DBMSs, causing many deep bugs to still exist. Therefore, proposing a practical solution to address these limitations is necessary and important for DBMS fuzzing.

3 Stateful DBMS Fuzzing

DBMSs process each statement of queries in sequence. After each statement is executed, the content of manipulated databases and the status of DBMSs are dynamically changed. In the interval between two statement execution, DBMSs record their latest state information, which accurately reflects their real-time situations, including the latest database schema and the status of statement processing. The state information is valuable for guiding query generation. However, such information is available only after each statement is executed, so existing DBMS fuzzers cannot capture it, because they perform static query generation before query execution.

To solve this problem, we propose a novel stateful fuzzing approach, which captures accurate state information by dynamically interacting with target DBMSs, not statically inferring state changes. Figure 2 shows the overview of our approach. Its core is *dynamic query interaction*, which merges query generation and query execution into an interactive process. During the interaction, our approach continuously captures the latest DBMS states to incrementally generate complex and valid queries where dependencies among statements are satisfied with correct data references. To further improve the validity of generated queries, our approach utilizes *error feedback* to filter out invalid test cases in the seed pool.

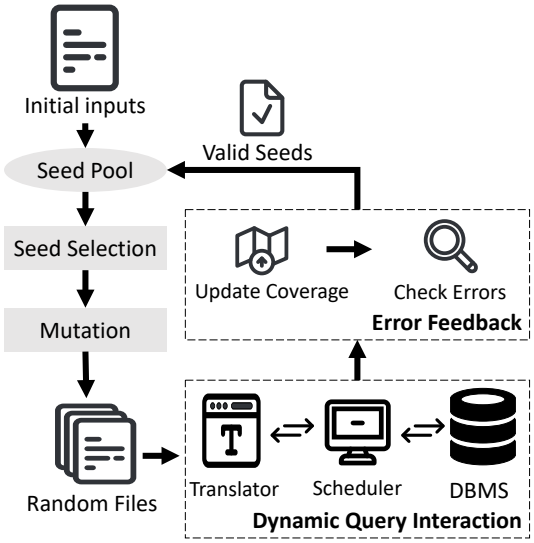

Figure 2: Overview of stateful DBMS fuzzing.

Algorithm 1: Dynamic Query Interaction

```
   input  : file, DBMS
   output : query, cov, status
1  Function Scheduler(file, DBMS):
2      file_size ← GetFileSize(file);
3      DBMS ← INITIAL_STATE;
4      rb ← 0; query ← []; cov ← {};
5      for rb < file_size do
6          schema ← QueryDBMS(DBMS);
7          stmt, rb ← Translator(schema, file, rb);
8          query ← [query, stmt];
9          status, cov ← ExeStmt(stmt, DBMS);
10         if CheckStatus(status, query) then
11             break;
12     return query, cov, status;
13 Function Translator(schema, file, rb):
14     tmp_file ← file[rb, GetFileSize(file) - 1];
15     StmtGenerator.RandomSource(tmp_file);
16     stmt, tmp_rb ← StmtGenerator.Gen(schema);
17     return stmt, rb + tmp_rb;
18 Function CheckStatus(status, query):
19     if status == CRASH then
20         ReportCrash(query);
21         return TRUE;
22     if status ∈ ERROR then
23         if status ∉ SynErr and status ∉ SemErr then
24             ReportAbnormalError(query);
25         return TRUE;
26     return FALSE;
```

3.1 Dynamic Query Interaction

Overview. Before generating each statement, dynamic query interaction first queries the target DBMS to capture state information, including database schema and status of statement processing. Then, this technique uses such information to generate a statement and feeds it to the DBMS. After the statement is executed, this technique interacts with the DBMS again for the latest state information and uses it to generate subsequent statements. In this way, dynamic query interaction can accurately capture the state changes caused by the executed statements, and thus it can effectively generate complex and valid queries.

Dynamic query interaction mainly consists of two parts, `Scheduler` and `Translator`. `Scheduler` is used to interact with the target DBMS to capture the latest DBMS states, transfer the database schema to `Translator`, and manage the whole interaction process. `Translator` is used to translate an input file into an SQL statement based on the received database schema. The workflow of dynamic query interaction is described in Algorithm 1. Given an input file and the target DBMS, dynamic query interaction outputs the generated query, the code coverage of the target DBMS, and the status of query processing. The following discusses the details of dynamic query interaction.

Scheduler. First, `Scheduler` initializes the used variables, including *file_size* of the input file, the target DBMS, the read bytes *rb*, the *query*, and the coverage *cov*. Then, `Scheduler` enters a loop, which will end if all bytes in the input file *file* have been read. In this loop, `Scheduler` first queries the target DBMS to obtain its latest database schema (e.g. the attributes of tables, columns, views, indexes) and then sends the queried *schema*, the *file*, and read bytes *rb* to `Translator`, which will return a generated statement *stmt* and the updated *rb*. `Scheduler` stores *stmt* to the end of the *query* and feeds *stmt* to the target DBMS. After the DBMS processes the statement, `Scheduler` collects the covered branches into *cov*, and checks the *status* of statement processing. If the *status* indicates that a crash or an error is triggered, `Scheduler` exits the loop. When the loop ends, `Scheduler` returns the generated *query*, the code coverage *cov* of the DBMS execution, and the final *status* of the query processing.

Translator. Receiving the parameters from `Scheduler`, `Translator` first extracts the fresh part of *file*, which has not been read yet, into *tmp_file*. `Translator` uses an internal SQL statement generator `StmtGenerator` to generate a statement according to the provided *schema* and *tmp_file*. Finally, `Translator` returns the generated statement *stmt* and updates the number of bytes that have been read by `StmtGenerator`.

The internal `StmtGenerator` deploys AST models to generate SQL statements. Typically, AST-based tools [39, 41, 43] generate random SQL statements according to their random seed (e.g. system clock), which makes the generation inefficient and difficult to guide [30, 32]. In contrast to these tools, `StmtGenerator` uses *tmp_file* as its random seed, which means that when `StmtGenerator` traverses its AST tree to generate SQL statements, it decides which paths should be chosen according to the value read from *tmp_file*. Specifically, it makes a decision by calculating the result of `v` mod `n`, where `v` is the value read from the input file and `n` is the number of available choices. In this way, `StmtGenerator` deterministically generates SQL statements according to the provided input file *file* and the current database schema *schema*.

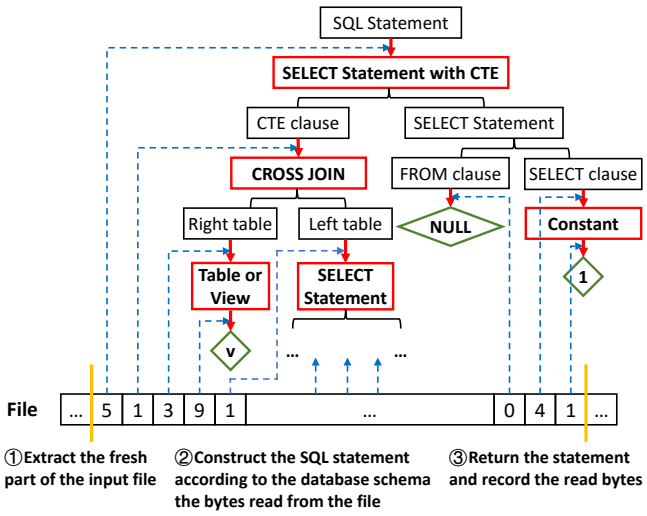

Figure 3: Generating the third SQL statement in Figure 1.

Figure 3 shows the generation process of the third statement in Figure 1. After receiving the fresh part of the input file from `Translator`, `StmtGenerator` starts traversing its AST model to generate a SQL statement. When it needs to determine the type of the statement, it reads a byte from the file and gets value 5, which indicates that it should generate a `SELECT` statement with `CTE`. Then, `StmtGenerator` reads the file again and gets value 1, which indicates that it should use `CROSS JOIN` to construct `CTE`. It needs to further determine the right table and the left table used in `CROSS JOIN`, and decides to use existing tables or views for the right table after getting the value 3 from the file. As there are two available candidates (i.e. table `t1` and view `v1`), `StmtGenerator` reads the file and gets value 9. According to the calculated result of 9 mod 2, it uses the second candidate (i.e. view `v1`). The subsequent generation process follows a similar procedure.

Status checking. `Scheduler` checks the execution status after each statement is fed to the target DBMS. Specifically, `Scheduler` checks whether any crash of the DBMS or its manipulated databases is triggered. If so, it reports the crash with the query, including the crash-triggering statement and the earlier statements generated in the previous interaction. Similar to SQLancer [37], `Scheduler` also checks whether the target DBMS reports any error. If so, it performs a further check and reports an abnormal error if the error is not a syntactic or semantic error. For example, "Subquery result missing" in MonetDB is an abnormal error that indicates the DBMS has lost the data of calculated results. These checks enable our dynamic query interaction to report suspicious bugs that make the target DBMS alert but do not cause it to crash directly. Note that if any crash or any error is triggered, `Scheduler` will terminate the interaction loop because the target DBMS has entered a problematic state.

Example. Figure 4 illustrates how our dynamic query interaction generates the malicious query in Figure 1 and detects the vulnerability. When the test begins, `Scheduler` first queries

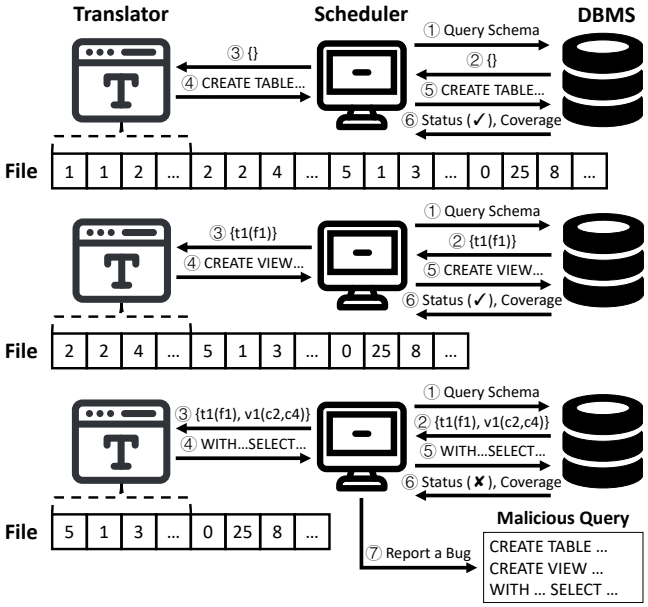

Figure 4: Dynamic query interaction for the query in Figure 1.

the target DBMS (i.e. MariaDB) to obtain its database schema. The schema is empty as no statement has been processed yet. `Scheduler` sends the empty schema to `Translator`. Then, `Translator` traverses its AST model and generates a `CREATE TABLE` statement according to the read value from the file. `Scheduler` feeds the statement to the DBMS and collects the code coverage and the status of statement processing. As there is no crash or error, `Scheduler` enters the next round of interaction. In the second round, `Scheduler` queries the DBMS for the latest schema again and then sends the schema to `Translator`. As the `CREATE TABLE` statement has been executed, the schema contains a created table `t1`. `Translator` first removes the part of the file that has been read and then uses the processed file to generate a new statement, according to the updated schema. It reads some bytes from the file and generates a `CREATE VIEW` statement, which references the table `t1`. The generated statement is fed to the DBMS again and is processed normally, so `Scheduler` enters the third round. In the third round, `Scheduler` queries the DBMS and gets the latest schema that is extended with a view `v1` containing two columns `c2` and `c4`. This schema is sent to `Translator`, and `Translator` accordingly generates a `SELECT` statement with `CTE` where the view `v1` in the schema is referenced. When `Scheduler` feeds the generated statement to the DBMS, a crash is triggered, so `Scheduler` terminates the interaction and reports a bug with the bug-triggering query that contains the 3 generated statements.

3.2 Error Feedback

The input file of dynamic query interaction controls the process of query generation. Proper files can guide the approach to generate complex and valid SQL queries, while useless

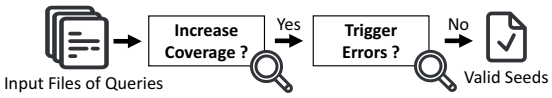

Figure 5: Workflow of error feedback.

files can result in repeated and trivial queries. Thus, our stateful DBMS fuzzing needs to produce effective input files. Coverage-guided fuzzers [2, 6, 15, 26, 30] for general files seem useful to achieve this goal, with the program feedback of code coverage. In DBMS fuzzing, when an invalid query triggers a new syntactic or semantic error, code coverage is indeed increased. In this case, existing coverage-guided fuzzers save this invalid query into the seed pool, and use it as a seed to perform mutation for generating other similar queries. However, these generated queries are very likely to trigger the same syntactic or semantic error with the seed query, without increasing code coverage.

To tackle this problem and further improve the validity of generated queries, we propose *error feedback* to filter out the input files of invalid queries in the seed pool. Figure 5 shows the workflow of error feedback. For each input file of SQL query, our approach checks whether the query increases the code coverage during execution. If the query causes the DBMS to cover new branches at runtime, our fuzzing approach merges these branches into the global coverage. For each input file of SQL query that increases the coverage, our approach further checks whether the query makes the target DBMS execute abnormally. If the DBMS reports any error, the input file will be discarded for seed mutation. In this way, our fuzzing approach guarantees that all the identified seeds can produce valid SQL queries when they are used as input files in dynamic query interaction. Using these valid seeds, our seed mutation achieves a high possibility of generating valid queries during fuzzing.

4 Framework and Implementation

Based on our stateful DBMS fuzzing approach, we develop a new fuzzing framework, DynSQL, to detect deep bugs in DBMSs by generating complex and valid queries. DynSQL uses Clang [8] to compile and instrument target DBMSs for collecting coverage information. Figure 6 shows the architecture of DynSQL, which consists of six modules:

- ***Code instrumentor.*** It compiles and instruments the code of the target DBMS, and generates an executable program that receives and processes SQL queries.
- ***Query interactor.*** It receives input files from the file fuzzer and performs dynamic query interaction to generate complex and valid queries. It also collects necessary runtime information of the target DBMS for dynamic analysis.
- ***Statement generator.*** It uses an internal AST model to generate syntactically correct SQL statements that only reference the data claimed in the given database schema.

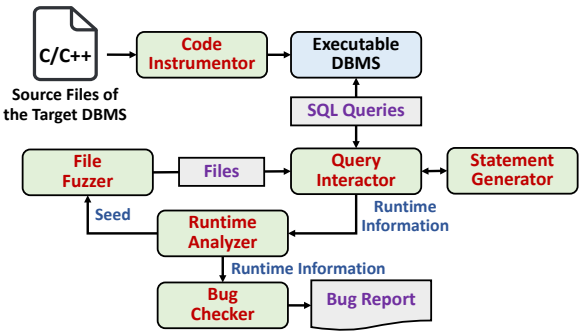

Figure 6: Overall architecture of DynSQL.

- ***Runtime analyzer.*** It analyzes the collected runtime information, identifies seeds according to error feedback, and selects a seed for the next round of fuzzing.
- ***File fuzzer.*** It performs conventional file fuzzing to generate files based on the given seeds. We implement this module by mainly referring to AFL [2].
- ***Bug checker.*** It detects bugs based on the collected runtime information and generates corresponding bug reports.

The following discusses the important details of DynSQL.

DBMS supporting. DBMSs often provide interfaces for external programs to operate and query their databases. These interfaces are used by DynSQL to set up the testing process. Specifically, DynSQL needs interfaces to start the DBMS, connect to the DBMS, stop the DBMS, send SQL statements, get results of statement processing, and query database schema. DBMSs often have well-defined interfaces for these operations, so it is convenient for users to adopt DynSQL in different DBMSs. In our experience, it took one of our authors less than one hour to make DynSQL support a DBMS.

SQL statement generation. We implement our AST-based statement generator referring to SQLsmith [43], and additionally support some SQL features, such as `GROUP BY`, `UNION`, etc. Because many DBMSs use their own SQL dialects and the common core of their SQL features is small [37, 39], it is difficult to use one grammar template to test all DBMSs effectively. To address this problem, we fix the general parts of the supported SQL features according to SQL standard [40] and make other parts optional. When testing a specific DBMS, we enable the optional SQL features supported by this DBMS according to its official documents.

Bug detection. DynSQL uses ASan [3] as its default checker to detect critical memory bugs. In addition, DynSQL analyzes the abnormal errors collected in dynamic query interaction (Section 3.1) to detect bugs that lead DBMSs to report strange error messages.

Query minimization. To reproduce and locate DBMS bugs more conveniently, we perform minimization for each generated query that triggers a new bug. Our minimization process mainly refers to APOLLO [18] and C-Reduce [33]. In some cases, developers help us further minimize the bug-triggering queries with their professional knowledge.

Table 2: Basic information of the target DBMSs

DBMS	Mode	Version	LOC
SQLite	Serverless	v3.33.0	165K
MySQL	Client/Server	v8.0.22	3.25M
MariaDB	Client/Server	v10.5.9	3.45M
PostgreSQL	Client/Server	v13.2	1.05M
MonetDB	Client/Server	Oct2020_17	307K
ClickHouse	Client/Server	v21.5.6.6	640K

5 Evaluation

To understand the effectiveness of DynSQL, we evaluate it on real-world and production-level DBMSs. Specifically, our evaluation aims to answer the following questions:

- ***Q1*** Can DynSQL find bugs in real-world DBMSs by generating complex and valid queries? (Section 5.2)
- ***Q2*** How about the security impact of the bugs found by DynSQL? (Section 5.3)
- ***Q3*** How do dynamic query interaction and error feedback contribute to DynSQL in DBMS fuzzing? (Section 5.4)
- ***Q4*** Can DynSQL outperform other state-of-the-art DBMS fuzzers? (Section 5.5)

5.1 Experimental Setup

We evaluate DynSQL on 6 open-source and widely-used DBMSs of the latest versions as of our evaluation, including SQLite [42], MySQL [29], MariaDB [27], PostgreSQL [31], MonetDB [28], and ClickHouse [9]. We choose these DBMSs because they are widely used according to DB-Engines Ranking [12] and extensively tested [18, 24, 41, 43, 48]. The basic information of these DBMSs is listed in Table 2 (The lines of the source code are counted by CLOC [10]). We run the evaluation on a regular PC with eight Intel processors and 16GB physical memory, and the OS used is Ubuntu 18.04.

5.2 Runtime Testing

Following the evaluation setup of SQUIRREL [48] and the recommendations of Klees et al. [21], we use DynSQL to fuzz each target DBMS five times and calculate the average to get sound results. We use 24 hours as the fuzzing timeout because we observe that the branch coverage and the found bugs of 6 target DBMSs converge and hardly change after 24 hours, which is consistent with SQUIRREL.

Table 3 shows the results of runtime testing. The columns "Found", "Confirmed" and "Fixed" show the number of bugs that are found by DynSQL, confirmed and fixed by developers, respectively. The columns "Statement" and "Query" show the numbers of SQL statements and queries, respectively, which are valid and generated (valid/generated).

Generated queries and statements. DynSQL generates 101K SQL queries, and 79K of them are valid. The percentage of valid queries is 78%. These generated queries contain 866K SQL statements, and 838K of them are valid. The percentage of valid statements is 97%. The average number of statements contained by each valid query is 8.6. These results indicate that DynSQL can effectively generate valid queries that contain multiple statements. We investigate the invalid statements, and find that they fail to pass validation checks as they use complex expressions, whose results dissatisfy the constraints of their data type or the integrity constraints of the databases.

Table 3: Detailed results of DBMS fuzzing

DBMS	Bug			Validity	
	Found	Confirmed	Fixed	Statement	Query
SQLite	4	3	3	279K/286K	24K/30K
MySQL	12	12	6	91K/96K	9.4K/13K
MariaDB	13	13	6	170K/175K	17K/21K
PostgreSQL	0	0	0	154K/160K	14K/18K
MonetDB	5	5	5	70K/72K	7.0K/8.6K
ClickHouse	6	5	1	74K/77K	7.5K/10K
Total	40	38	21	838K/866K	79K/101K

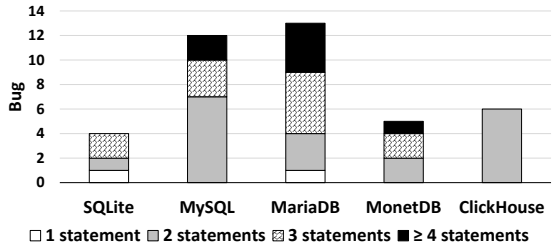

Figure 7: Number of SQL statements that trigger DBMS bugs.

Found bugs. DynSQL finds 40 unique bugs, including 4 in SQLite, 12 in MySQL, 13 in MariaDB, 5 in MonetDB, and 6 in ClickHouse. Among these bugs, 31 are memory bugs, and 9 are semantic bugs that cause DBMSs to report strange errors. The details of these bugs are discussed in Section 5.3. We reported these bugs to related developers. Among them, 38 bugs have been confirmed, 21 bugs have been fixed, and 19 have been assigned CVE IDs. For the 17 unfixed bugs (e.g. two heap-buffer-overflow bugs in MySQL), the developers have not figured out exact root causes due to the complex logic of DBMSs, or they have not built proper fixing patches that do not degrade DBMS performance. For the 2 unconfirmed bugs, we are still waiting for the response from developers.

Statements in bug-triggering queries. We analyze the number of statements in the queries triggering DBMS bugs. Note that all the analyzed queries have been simplified. The results in Figure 7 indicate that only 2 bugs can be triggered by using 1 statement. These 2 bugs are triggered by a `SELECT` statement and a `CREATE TABLE` statement, respectively. 19 bugs can be triggered using queries with 2 statements. These queries all use a `CREATE TABLE` statement and a `SELECT` statement. For the 19 remaining bugs, the bug-triggering queries contain at least 3 statements. These queries often use different kinds of statements with various SQL features. Listings 1-6 of Appendix A show some examples of these queries.

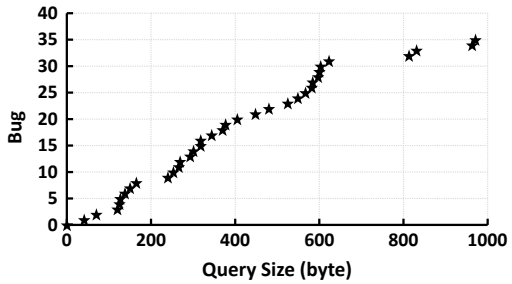

Figure 8: Bytes of SQL queries that trigger DBMS bugs.

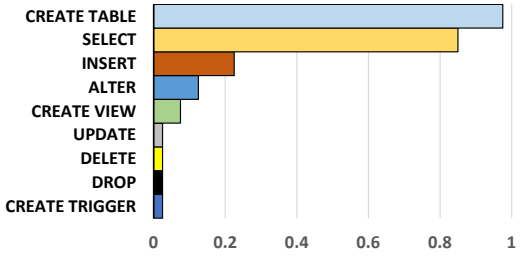

Figure 9: Percentage of queries including specific statements that trigger DBMS bugs.

```
--- a/sql/sql_lex.cc
+++ b/sql/sql_lex.cc
@@ -2998,7 +2998,7  @@ -3006,7 +3006,8
bool st_select_lex::setup_ref_array(...) {
  ...
-  const uint n_elems= (n_sum_items +
+  const size_t n_elems= (n_sum_items +
              n_child_sum_items +
              item_list.elements +
              select_n_reserved +
              select_n_having_items +
              select_n_where_fields +
              order_group_num +
              hidden_bit_fields +
-             fields_in_window_functions) * 5;
+             fields_in_window_functions) * 5ULL;
+  DBUG_ASSERT(n_elems % 5 == 0);
  ...
  // Overflowed n_elems is very small
  Item **array= static_cast<Item**>(
              arena->alloc(sizeof(Item*) * n_elems));
  if (likely(array != NULL))
    // the array will be referenced later
    ref_pointer_array= Ref_ptr_array(array, n_elems);
  return array == NULL;
}
```

Figure 10: Integer overflow in MariaDB.

Size of bug-triggering queries. Figure 8 shows the size of the bug-triggering queries. Among the 40 found bugs, 35 bugs are triggered by queries whose sizes are smaller than 1000 bytes. As query size increases from 0 to 600 bytes, the number of triggered bugs increases almost linearly. When query size increases up to 600 bytes, 30 bugs are found. The queries in Listing 1-4 of Appendix A are the examples. When query size increases from 600 to 1000 bytes, only 5 additional bugs (e.g. Listing 6 of Appendix A) are found. For the 5 remaining bugs (e.g. Listing 5 of Appendix A), their bug-triggering queries are very complex and hard to simplify. The biggest query is over 300K bytes, triggering an integer overflow in MariaDB.

Validity of bug-triggering queries. We tried to check the validity of all 40 bug-triggering queries, but they caused DBMSs to abnormally abort due to ASan alerts (for the 31 memory bugs) or strange errors (for the 9 semantic bugs), and thus we cannot clearly perform validity checking. Therefore, we focus on the queries triggering the 21 fixed bugs. We first apply the developers' patches to fix corresponding bugs and then check whether these queries work normally. We find that all these queries are valid, without syntactic or semantic errors. Indeed, many of these bugs are related to the deep logic of query processing, and thus they are never triggered by invalid queries discarded by earlier validation checks.

Statement distribution of bug-triggering queries. Figure 9 shows the statement distribution of the 40 bug-triggering queries. The `CREATE TABLE` and `SELECT` statements are the most common statements in these queries. In most cases, the `CREATE TABLE` statement is used to create a basic table for the subsequent statements to access and manipulate, and thus most generated queries contain this statement. The majority (32/34) of `SELECT` statements are used as the last statements in the queries, which trigger 80% of the found bugs. The remaining bugs are triggered by `CREATE TABLE` (10%), `ALTER` (2.5%), `DROP` (2.5%), `UPDATE` (2.5%) and `DELETE` (2.5%) statements, respectively. The `INSERT`, `ALTER` and `CREATE VIEW` statements are often used as intermediate statements that cause DBMSs into specific states tending to trigger bugs.

5.3 Security Impact

We classify the 40 found bugs by their security impact, and show the results in Table 4. 18 bugs found by DynSQL are null-pointer dereferences that can be exploited to perform denial-of-service (DoS) attacks by repeatedly crashing DBMSs. DynSQL finds 7 critical memory bugs, including 2 use-after-free bugs, 2 stack-buffer-overflow bugs, 2 heap-buffer-overflow bugs, and 1 integer-overflow bug. These 7 bugs can cause severe security problems like privilege escalation and information leaks. DynSQL also finds 6 assertion failures, indicating target DBMSs reach unexpected states. By analyzing abnormal error reports, DynSQL additionally finds 9 semantic bugs. Among these 40 bugs, 19 have been assigned with CVE IDs. The details of these CVEs are shown in Table 8 of Appendix B. To better understand the security impact of bugs found by DynSQL, we explain three confirmed bugs:

Case study 1: integer overflow in MariaDB. This bug is identified as a critical vulnerability and was assigned CVE-2021-46667. It allows attackers to write and read arbitrary data in the memory space. By exploiting this bug, attackers can overwrite the data of other users, escalate their privileges and even perform remote code execution (RCE). The related code of this bug is shown in Figure 10. The MariaDB server calculates the number of items in a `SELECT` statement and then stores the result in an unsigned integer `n_elems`. After that,

Table 4: Types of the found bugs

Bug type	SQLite	MySQL	MariaDB	PostgreSQL	MonetDB	ClickHouse	Total
Null-pointer dereference	0	8	9	0	1	0	18
Use-after-free	0	0	2	0	0	0	2
Stack buffer overflow	1	1	0	0	0	0	2
Heap buffer overflow	0	2	0	0	0	0	2
Integer overflow	0	0	1	0	0	0	1
Assertion failure	1	0	1	0	3	1	6
Abnormal error	2	1	0	0	2	5	9

FILE: MariaDB/sql/sql_class.cc

```
void Item_change_list::rollback_item_tree_changes() {
  ...
  I_List_iterator<Item_change_record> it(change_list);
  Item_change_record *change;
  while ((change= it++)) {
    ....
    // change has been freed
    *change->place= change->old_value;
  }
  ...
}
```

Figure 11: Use after free in MariaDB.

the server uses `n_elems` as a parameter to allocate a memory `array` that is later used to store and fetch items in the `SELECT` statement. However, if the processed `SELECT` statement has specific complex structures, the calculated result of its items may be greater than the maximum value (i.e. $2^{32}-1$ in Linux 64 bit) of `n_elems`. As a result, an integer overflow happens in `n_elems` that can become very small. In this case, the server allocates a memory area of `array` that is much smaller than needed. When the server stores or fetches items in `array` with big indexes, a heap buffer overflow will be further triggered, and attackers can exploit it to write or read arbitrary data in the memory space. This bug is actually hard to find. DynSQL uses a `SELECT` statement larger than 300K bytes to trigger the integer overflow of `n_elems`. To fix this bug, the developers use `size_t` to define `n_elems` in order to increase its maximum value ($2^{64}-1$ in Linux 64 bit). The developers also insert an assertion to prevent the integer overflow.

Case study 2: use after free in MariaDB. This bug is caused by misusing rollback mechanism and was assigned CVE-2021-46669. When processing a statement, the MariaDB server stores each change caused by this statement into a list and deletes the changes if the statement is processed successfully. At subsequent stages, the server checks the change list. If the list is not empty, indicating a failure of statement execution, the server rolls back the changes. When processing a specific query generated by DynSQL, the server frees the content of changes but forgets to delete these changes in the list, which triggers the rollback mechanism, causing the freed changes in the list to be dereferenced. Figure 11 shows the code where the bug is triggered. This bug can be exploited to overwrite the data in arbitrary addresses with the freed data.

Case study 3: missing subquery result in MonetDB. This bug is found due to capturing abnormal errors collected in dynamic query interaction. The bug-triggering query contains a `CREATE TABLE` statement and a `SELECT` statement with a `CTE`. When the bug is triggered, the MonetDB server reports an error message that indicates "Subquery result missing". Literally, this error is not a syntactic or semantic error caused by invalid queries. We report it as a bug that affects the availability of supported SQL features. The MonetDB developers confirmed that this bug was caused by incorrect optimization and fixed it by modifying optimization-related code.

5.4 Sensitivity Analysis

To understand the contribution of dynamic query interaction and error feedback, we perform sensitivity analysis by disabling these techniques. Specifically, we design DynSQL$_{!DQI}$, DynSQL$_{!EF}$ and DynSQL$^{!DQI}_{!EF}$. In DynSQL$_{!DQI}$, we disable only dynamic query interaction; in DynSQL$_{!EF}$, we disable only error feedback; in DynSQL$^{!DQI}_{!EF}$, we disable both dynamic query interaction and error feedback. In the cases of disabling dynamic query interaction, because our statement generator relies on database schema to work, we provide the schema only when the first table is created, and do not update the schema in subsequent statement generation. We evaluate DynSQL$_{!DQI}$, DynSQL$_{!EF}$ and DynSQL$^{!DQI}_{!EF}$ on the six DBMSs in Table 2. Similar to Section 5.2, we use each fuzzer to test each DBMS five times and set the time limit of each fuzzing to 24 hours. Table 5 shows the average results. In Table 5, the columns "Statement" and "Query" show the number of SQL statements and queries, respectively, which are valid and generated (valid/generated).

Validity of queries and statements. The percentages of valid statements and queries generated by DynSQL$^{!DQI}_{!EF}$ are only 62% and 36%, respectively. By enabling error feedback (i.e. DynSQL$_{!DQI}$), these percentages are increased to 71% and 55%, respectively. It indicates that error feedback can improve the validity of generated statements and queries by filtering out invalid seeds in the fuzzing process. By enabling dynamic query interaction (i.e. DynSQL$_{!EF}$), the percentages of valid statements and queries are dramatically increased to 95% and 68%, respectively, because dynamic query interaction involves state information (i.e. the latest database schema and status of statement processing) to boost query generation. By enabling both dynamic query interaction and error feedback (i.e. DynSQL), the percentages of valid statements and queries are increased to 97% and 78%, respectively.

Table 5: Results of sensitivity analysis

DBMS	DynSQL$^{!DQI}_{!EF}$				DynSQL$_{!DQI}$				DynSQL$_{!EF}$				DynSQL			
	Statement	Query	Coverage	Bug	Statement	Query	Coverage	Bug	Statement	Query	Coverage	Bug	Statement	Query	Coverage	Bug
SQLite	175K/307K	11K /30K	49K	1	208K/305K	16K/35K	51K	1	285K/293K	21K/29K	53K	3	279K/286K	24K/30K	54K	4
MySQL	65K/100K	2.8K/12K	485K	4	65K/96K	7.5K/13K	502K	6	89K/94K	9.2K/13K	518K	10	91K/96K	9.4K/13K	526K	12
MariaDB	123K/177K	7.9K/18K	275K	3	136K/176K	12K/22K	291K	6	165K/174K	13K/21K	309K	9	170K/175K	17K/21K	319K	13
PostgreSQL	103K/160K	6.4K/17K	125K	0	123K/162K	11K/18K	132K	0	144K/157K	13K/18K	141K	0	154K/160K	14K/18K	147K	0
MonetDB	41K/80K	3.2K/6.9K	126K	2	53K/78K	7.1K/11K	137K	2	66K/70K	4.9K/6.6K	145K	5	70K/72K	7.0K/8.6K	149K	5
ClickHouse	53K/83K	1.9K/7.8K	435K	3	55K/82K	6.3K/11K	458K	4	72K/76K	8.3K/12K	466K	6	74K/77K	7.5K/10K	476K	6
Total	560K/907K	33K/92K	1495K	13	641K/899K	61K/110K	1571K	17	821K/864K	69K/101K	1632K	33	838K/866K	79K/101K	1671K	40

Table 6: Time-usage percentage of each stage in DynSQL

DBMS	Schema Querying	Query Generation	Query Execution
SQLite	1.17%	3.16%	95.67%
MySQL	0.15%	0.44%	99.41%
MariaDB	1.29%	4.67%	94.04%
PostgreSQL	1.20%	10.04%	88.76%
MonetDB	0.13%	2.20%	97.67%
ClickHouse	0.19%	1.42%	98.39%
Average	0.69%	3.66%	95.65%

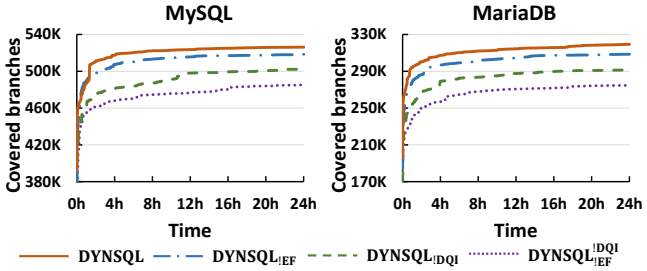

Figure 12: Covered branches of MySQL and MariaDB.

From Table 5, we also observe that DynSQL generates slightly fewer valid queries in MonetDB and ClickHouse, compared to DynSQL$_{!DQI}$ and DynSQL$^{!DQI}_{!EF}$, for two reasons. First, after enabling dynamic query interaction or error feedback, the fuzzer generates more valid statements in total, but each valid query contains more valid statements, and thus the number of valid queries may decrease. Second, enabling dynamic query interaction or error feedback can also slightly slow down query generation, due to monitoring DBMS states or checking query results.

Runtime overhead. The overhead of error feedback is small because it only adds some *if* checks in the process of seed identification. Compared to DynSQL$^{!DQI}_{!EF}$, the number of statements generated by DynSQL$_{!DQI}$ decreases by less than 1%. Dynamic query interaction may introduce overhead as it needs to query database schema from target DBMSs. However, this overhead is often small. On the one hand, most DBMSs provide efficient methods for these kinds of queries. On the other hand, our fuzzer often generates complex queries that are time-consuming for DBMSs, so the runtime overhead introduced by querying the latest database schema is very small in contrast. Compared to DynSQL$^{!DQI}_{!EF}$, the number of statements generated by DynSQL$_{!EF}$ only decreases by 5%.

To further validate the overhead, we record the time usage of schema querying, query generation, and querying execution, respectively. The average results are shown in Table 6. For each test case, the time usage of database schema querying and query generation is on average less than 5%, while more than 95% of the time is used for query execution. Note that when testing PostgreSQL, DynSQL spends more time (10%) on query generation, because DynSQL uses more complex generation logic to satisfy the statement grammars of PostgreSQL. The results indicate that the performance bottleneck is query execution, and the overhead introduced by dynamic querying interaction is comparatively small.

Code coverage. On average, DynSQL$_{!DQI}$, DynSQL$_{!EF}$ and DynSQL cover 5%, 10% and 13% more code branches than DynSQL$^{!DQI}_{!EF}$, respectively. These results indicate that dynamic query interaction and error feedback can help fuzzers cover more code branches. Figure 12 shows the growth of covered branches for MySQL and MariaDB during fuzzing. Four fuzzers cover new code branches quickly in earlier tests and then cover fewer and fewer branches in later tests. In almost the whole fuzzing process, DynSQL covers more branches than the other three substitution fuzzers.

Bug detection. DynSQL$_{!DQI}$ additionally finds 4 bugs missed by DynSQL$^{!DQI}_{!EF}$ by enabling error feedback. Indeed, error feedback can help the fuzzer generate more valid queries to detect more bugs. However, error feedback cannot improve query complexity, so DynSQL$_{!DQI}$ and DynSQL$^{!DQI}_{!EF}$ find only the bugs triggering within two statements and miss the bugs triggering by over two statements (e.g. the bug in Figure 1). In comparison, dynamic query interaction leverages DBMS state information to improve both query complexity and query validity, and thus DynSQL$_{!EF}$ finds 20 bugs missed by DynSQL$^{!DQI}_{!EF}$. By further enabling error feedback in DynSQL$_{!EF}$, DynSQL additionally finds 7 bugs.

5.5 Comparison to Existing DBMS Fuzzers

We compare DynSQL to two state-of-the-art DBMS fuzzers, SQLsmith [43] and SQUIRREL [48]. SQLancer [35–37] is also a well-known DBMS testing tool, but it mainly focuses on test oracles, which require test cases with specific patterns to

Table 7: Results of comparison

DBMS	SQLsmith			SQUIRREL			DynSQL		
	Statement	Query	Bug	Statement	Query	Bug	Statement	Query	Bug
SQLite	265K/267K	265K/267K	1	31M/45M	2.4M/9.8M	1	279K/286K	24K/30K	4
MySQL	100K/102K	100K/102K	3	506K/854K	17K/171K	3	91K/96K	9.4K/13K	12
MariaDB	148K/152K	148K/152K	3	245K/392K	425/78K	2	170K/175K	17K/21K	13
PostgreSQL	192K/197K	192K/197K	0	8M/10M	35K/560K	0	154K/160K	14K/18K	0
Total	705K/718K	705K/718K	7	40M/56M	2.5M/11M	6	695K/716K	64K/82K	29

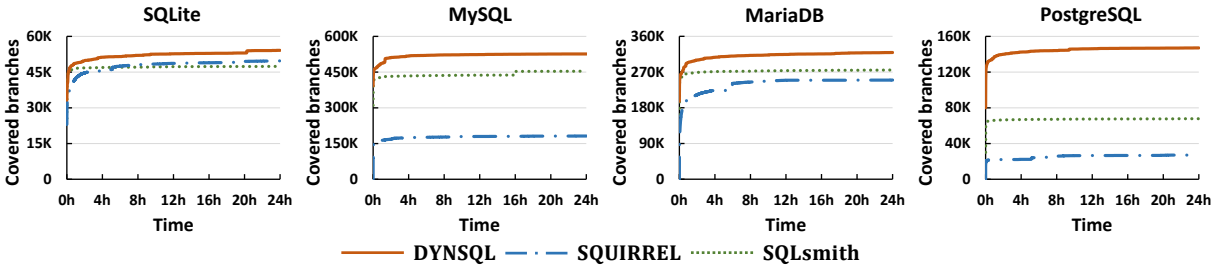

Figure 13: Code coverage of DynSQL and the other DBMS fuzzers.

find logic bugs in DBMSs, while DynSQL aims at generating complex and valid queries to detect common bugs, especially memory bugs. Considering DynSQL and SQLancer are designed for different research problems, we do not conduct the comparison experiment to SQLancer.

We exploit DynSQL, SQUIRREL, and SQLsmith to test SQLite, MySQL, MariaDB, and PostgreSQL, because SQUIRREL supports only these DBMSs and applying it to other DBMSs requires significant modification of the SQUIRREL implementation [45]. Besides, SQLsmith supports only SQLite, PostgreSQL, and MonetDB originally. To perform better comparisons, we extend SQLsmith to support MySQL and MariaDB via little modification of its code. For each test case, we randomly generate a database with tables for SQLsmith because it requires an available database to begin its test. We use each fuzzer to test each DBMS five times, and the time limit is 24 hours. Table 7 shows the comparison results. The columns "Statements" and "Query" show the number of SQL statements and queries, respectively, which are valid and generated (valid/generated).

Generated queries and statements. Because SQLsmith generates each query with only one statement, the number of its generated statements is equal to the number of generated queries. According to Table 7, the percentages of valid statements and valid queries are both 98%. However, it cannot generate queries containing multiple statements. Using its IR model, SQUIRREL can generate queries with multiple statements, whose average number of statements in each query is 5.1. However, SQUIRREL generates many invalid statements and queries, as its percentages of valid statements and valid queries are only 71% and 23%, respectively. In contrast, the percentages of valid statements and queries generated by DynSQL are up to 97% and 78%, respectively, and the average number of statements contained by each generated

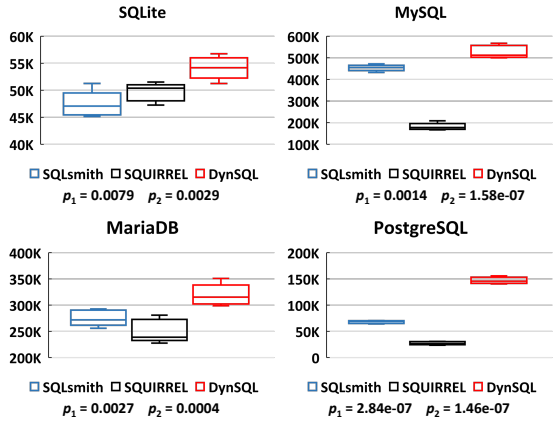

Figure 14: Box plots of code-coverage comparison.

query is 8.7. These results indicate that DynSQL can generate more valid queries containing multiple statements. In addition, we observe that SQUIRREL generates more statements than DynSQL and SQLsmith in the given testing time, because SQUIRREL typically generates simple statements that can be quickly executed by DBMSs.

Code coverage. As shown in Figure 13, DynSQL covers, on average, 41% and 166% more code branches than SQLsmith and SQUIRREL, respectively. Figure 14 shows the box plots of code-coverage comparison, where p_1 and p_2 are the p values of SQLsmith vs. DynSQL and SQUIRREL vs. DynSQL, respectively. Both p_1 and p_2 are less than 0.05, indicating that DynSQL covers significantly more code branches than SQLsmith and SQUIRREL. Indeed, though SQUIRREL generates more statements, DynSQL and SQLsmith can generate more complex statements than SQUIRREL, to cover deeper logic of DBMS code. Compared to SQLsmith, DynSQL further generates queries with multiple statements, to cover many more code branches.

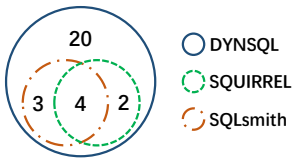

Figure 15: Relation of bugs detected by three DBMS fuzzers.

Bug detection. We describe the relation of found bugs in Figure 15. SQLsmith misses 2 bugs found by SQUIRREL. Indeed, these two bugs can only be triggered by at least three statements, but SQLsmith can generate only one statement in each query. SQUIRREL also misses 3 bugs found by SQLsmith. Indeed, these bugs can only be triggered by statements with complex structure and references, which SQUIRREL fails to generate. DynSQL finds all the bugs found by SQLsmith and SQUIRREL, and additionally finds 20 bugs. Most of the 20 bugs require bug-triggering queries to contain at least three complex statements, which are difficult for SQLsmith and SQUIRREL to generate.

Query complexity. We first minimize the 29 bug-triggering queries generated by each fuzzer and then count their query size, statement number, and SQL keyword number. We also analyze their data dependency by checking the number of data that is defined in earlier statements and then referenced in subsequent statements. Note that we only consider bug-triggering queries because the raw queries generated by each fuzzer typically contain redundant components (e.g. useless SQL clauses and statements), and accurate minimization is difficult to perform on such queries if they do not cause distinct behavior (e.g. triggering bugs). Figure 16 shows the results. SQUIRREL can generate queries with multiple statements. However, the generated queries are small, with fewer keywords and less referenced data, which indicates that these queries use relatively simple statements. In contrast, SQLsmith can generate bigger queries that have more keywords and more referenced data, but it is limited in generating multiple statements for each query. Although we provide it with a `CREATE TABLE` statement for initialization, SQLsmith still cannot generate queries with at least three statements. All the bug-triggering queries of SQUIRREL and SQLsmith can be generated by DynSQL, and DynSQL can also generate queries with multiple and complex statements. Figure 1 shows a query that can only be generated by DynSQL. Some other examples are shown in Listing 1-6 of Appendix A.

6 Limitations and Future Work

In this section, we discuss the limitations and future works of DynSQL according to its current implementation.

Invalid queries. As demonstrated in Section 5.2, DynSQL still generates 3% invalid statements and 22% invalid queries, which are mainly caused by constraint violations. Because DynSQL randomly generates expressions in statements, the

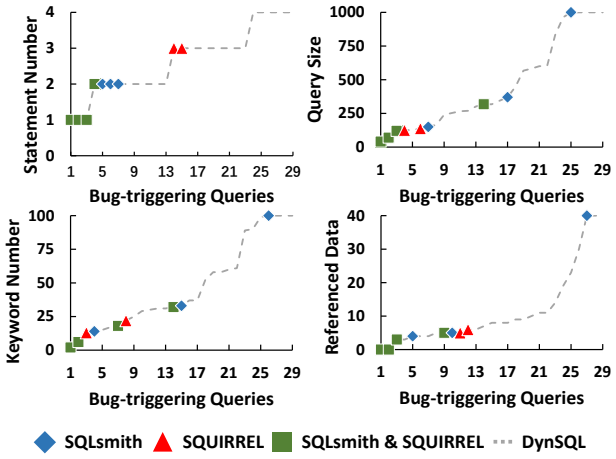

Figure 16: Comparison of query complexity for the 29 bugs found by DynSQL.

calculated values of these expressions probably violate the constraints of their data types. Moreover, some tables created by the earlier statements use specific clauses (e.g., `UNIQUE`, `NOT NULL`, and `CHECK`) to claim integrity constraints that limit the valid range of data. When subsequent DML (Data Manipulation Language) statements (e.g. `INSERT` and `UPDATE`) update these tables, if the updated data violates their integrity constraints, semantic errors will occur.

In our experience, it is difficult to eliminate these kinds of semantic errors. On the one hand, some generated expressions are very complicated, and their calculated values are difficult to accurately obtain. On the other hand, some constraints (e.g. constraints claimed in `CHECK` clause) are implicit, and thus it is difficult to extract the valid range of specific data. To mitigate this problem, we plan to exploit constraint solvers (e.g. SAT solvers) when generating expressions.

Other kinds of bugs. DynSQL mainly detects memory bugs with existing sanitizers like ASan, and it can additionally find semantic bugs leading to strange error messages. However, many semantic bugs silently affect the execution of DBMSs, and cause no obvious problems. For example, logic bugs do not cause any memory problem or strange error message, but make DBMSs return incorrect results. Performance bugs do not crash DBMSs directly, but often slow down its execution. Detecting these bugs is challenging because there is no general test oracle for checking whether the bugs have been triggered. Several existing DBMS testing works [35–37] use special oracles to detect such bugs. But these oracles require fixed patterns of test cases, which limits their scalability. In the future, we will improve the detection of semantic bugs in DynSQL by referring to these approaches.

AST rules construction. DynSQL uses an AST-based generator to produce SQL statements in each query. At present, we construct the AST rules based on our domain knowledge. Specifically, we construct general AST rules according to SQL-92 standard [40] and additionally write specific rules

for each supported DBMS according to its official documents. Users can enable just our general AST rules to test a new DBMS. However, to thoroughly test the code about DBMS-specific usages, users may need to write additional AST rules to enable their unique SQL features. To reduce such manual effort, inspired by existing work [11, 16], we plan to adopt machine learning techniques to automatically extract AST rules from valid queries.

7 Related Work

7.1 DBMS Testing

Some approaches are proposed to check the reliability and security of DBMSs. They either discover specific kinds of bugs [18, 23, 25, 34–37], or detect common bugs using general techniques [19, 20, 39].

DBMS testing for specific bugs. SQLancer [41] is designed to detect logic bugs in DBMSs, and it integrates several novel approaches [35–37]. PQS [37] can generate queries that require the target DBMS to return a result set where a specific row should be included. If the DBMS fails to fetch the row, it indicates a logic bug has been triggered. NoREC [35] is a metamorphic testing approach. It translates a query that can be optimized by the target DBMS to a query that cannot be optimized effectively. NoREC identifies a bug when these two queries make the DBMS return different results. To detect performance bugs, AMOEBA [25] transfers the given queries to semantically equivalent queries and then checks whether these queries result in performance differences.

To trigger specific kinds of DBMS bugs, these approaches generate test cases with specific patterns, limiting their extensibility to detect other kinds of bugs. In contrast to these approaches, DynSQL is designed to detect common bugs in DBMSs and does not limit the patterns of test cases.

DBMS testing for common bugs. RAGS [39] uses differential testing to detect bugs in DBMSs. It randomly generates SQL statements and then feeds them to several DBMSs whose databases are the same. RAGS reports bugs if the execution status or returned results of these DBMSs are different. However, this approach is limited because the common part of supported SQL features in different DBMSs is small [37, 39]. Some approaches [19, 20] convert SQL statement generation into SAT problems and generate valid statements to test DBMSs by solving syntactic and semantic constraints of the SQL language. However, these approaches cannot infer state changes caused by generated statements and thus can generate only one statement in each query. In contrast, DynSQL can generate queries with multiple statements by capturing the latest DBMS states before each statement generation. In addition, these approaches perform random test-case generation, or exhaustively enumerate all possible test cases. Comparatively, DynSQL uses coverage and error feedback to evolutionarily generate effective and valid test cases.

7.2 Fuzzing

General-purpose fuzzing. Fuzzing has been proven to be a promising technique for bug detection [2, 6, 15, 17, 26, 30, 47]. AFL [2] is one of the most famous fuzzers for general programs. It uses code coverage as feedback to boost its test-case generation and integrates various mutation strategies and engineering techniques to improve its efficiency. To cover more branches, Angora [6] first uses taint analysis to track specific input bytes that influence the control flows, and then it leverages gradient descent to quickly search for the suitable values of these bytes that satisfy the path constraints. To find more bugs, QSYM [47] adopts symbolic execution in fuzzing, and further uses dynamic binary translation to integrate symbolic emulation into the native execution, which dramatically reduces the overhead of symbolic execution.

However, existing work [48] proves that general fuzzers cannot effectively test DBMSs. These fuzzers fail to involve any SQL knowledge and AST rules, and thus generate many invalid queries violating syntactic or semantic checks.

DBMS fuzzing. To test DBMSs more effectively, several approaches [18, 43, 45, 48] combine fuzzing with grammar-based generation techniques. SQLsmith [43], a state-of-the-art DBMS fuzzer, uses its embedded AST rules to randomly generate SQL queries. However, each query generated by SQLsmith contains only one statement, because it is not aware of the state changes caused by generated statements. To generate queries with multiple statements, SQUIRREL [48] uses a new intermediate representation (IR) to model SQL queries and statically infers DBMS state changes caused by the generated statements. However, its static inference is inaccurate without runtime information, as a result of which it still generates over 50% invalid queries in the evaluation.

These fuzzers are limited in generating complex and valid queries, as they fail to consider state changes or infer accurate state information. In contrast, DynSQL performs dynamic query interaction to capture accurate state information, including the latest database schema and the status of statement processing. In this way, DynSQL can effectively generate complex and valid queries to detect deep bugs in DBMSs.

8 Conclusion

In this paper, we develop a practical DBMS fuzzing framework, named DynSQL, which can effectively generate complex and valid SQL queries to detect deep bugs in DBMSs. DynSQL integrates a novel technique, dynamic query interaction, to capture accurate DBMS state information and boost query generation. In addition, DynSQL uses error feedback to further improve the validity of generated queries. We have evaluated DynSQL on 6 widely-used DBMSs, and it finds 40 unique bugs. We also compare DynSQL to state-of-the-art DBMS fuzzers, and the results indicate that DynSQL finds more bugs in DBMSs with higher code coverage.

Acknowledgments

We thank our anonymous reviewers for their helpful and constructive feedback on earlier versions of the paper. We also thank the DBMS developers for triaging and fixing our reported bugs. This work was supported in part by the National Natural Science Foundation of China under Project 62002195. Jia-Ju Bai is the corresponding author.

References

[1] B. Acohido. Small banks and credit union attack set for tuesday, 2013. https://www.usatoday.com/story/cybertruth/2013/05/06/ddos-denial-of-service-small-business-cybersecurity-privacy/2139349/.

[2] American fuzzy lop. https://github.com/google/AFL.

[3] ASan: address sanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizer.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 23rd International Conference on Computer and Communications Security (CCS)*, pages 1032–1043, 2016.

[5] E. V. Buskirk. Facebook confirms denial-of-service attack, 2009. https://www.wired.com/2009/08/facebook-apparently-attacked-in-addition-to-twitter/.

[6] Peng Chen and Hao Chen. Angora: efficient fuzzing by principled search. In *Proceedings of the 2018 Symposium on Security and Privacy (S&P)*, pages 711–725, 2018.

[7] C. Cimpanu. Google chrome impacted by new magellan 2.0 vulnerabilities, 2019. https://www.zdnet.com/article/google-chrome-impacted-by-new-magellan-2-0-vulnerabilities/.

[8] Clang: a llvm-based compiler for C/C++ programs. https://clang.llvm.org/.

[9] ClickHouse. https://clickhouse.com/.

[10] CLOC: count lines of code. https://cloc.sourceforge.net/.

[11] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th International Symposium on Software Testing and Analysis (ISSTA)*, pages 95–105, 2018.

[12] DB-Engines Ranking. https://db-engines.com/en/ranking.

[13] UM Fayyad. Data science revealed: A data-driven glimpse into the burgeoning new field, 2011. https://embed.cs.utah.edu/creduce/.

[14] L. Franceschi-Bicchierai. Hacker tries to sell 427 milllion stolen myspace passwords for $2,800, 2016. https://www.vice.com/en/article/pgkk8v/427-million-myspace-passwords-emails-data-breach.

[15] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: data flow sensitive fuzzing. In *Proceedings of the 29th USENIX Security Symposium*, pages 2577–2594, 2020.

[16] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&Fuzz: machine learning for input fuzzing. In *Proceedings of the 32nd International Conference on Automated Software Engineering (ASE)*, pages 50–59, 2017.

[17] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Fuzzing error handling code using context-sensitive software fault injection. In *Proceedings of the 29th USENIX Security Symposium*, pages 2595–2612, 2020.

[18] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. APOLLO: automatic detection and diagnosis of performance regressions in database systems. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, 2020.

[19] Shadi Abdul Khalek, Bassem Elkarablieh, Yai O. Laleye, and Sarfraz Khurshid. Query-aware test generation using a relational constraint solver. In *Proceedings of the 23rd International Conference on Automated Software Engineering (ASE)*, pages 238–247, 2008.

[20] Shadi Abdul Khalek and Sarfraz Khurshid. Automated SQL query generation for systematic testing of database engines. In *Proceedings of the 2010 International Conference on Automated Software Engineering (ASE)*, pages 329–332, 2010.

[21] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 International Conference on Computer and Communications Security (CCS)*, pages 2123–2138, 2018.

[22] Doug Laney et al. 3d data management: Controlling data volume, velocity and variety. *META group research note*, 6(70):1, 2001.

[23] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. QTune: a query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment*, 12(12):2118–2130, 2019.

[24] libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[25] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. Automated performance bug detection in database systems. *arXiv preprint arXiv:2105.10016*, 2021.

[26] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: optimized mutation scheduling for fuzzers. In *Proceedings of the 28th USENIX Security Symposium*, pages 1949–1966, 2019.

[27] MariaDB. https://www.mariadb.org/.

[28] MonetDB. https://www.monetdb.org/.

[29] MySQL. https://www.mysql.com/.

[30] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th International Symposium on Software Testing and Analysis (ISSTA)*, pages 329–340, 2019.

[31] PostgreSQL. https://www.postgresql.org/.

[32] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. Quickly generating diverse valid test inputs with reinforcement learning. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, pages 1410–1421, 2020.

[33] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 2012 International Conference on Programming Language Design and Implementation*, pages 335–346, 2012.

[34] Kim-Thomas Rehmann, Changyun Seo, Dongwon Hwang, Binh Than Truong, Alexander Boehm, and Dong Hun Lee. Performance monitoring in sap hana's continuous integration process. *ACM SIGMETRICS Performance Evaluation Review*, 43(4):43–52, 2016.

[35] Manuel Rigger and Zhendong Su. 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 (ESE/FSE)*, pages 1140–1152, 2020.

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

[37] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 667–682, 2020.

[38] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020.

[39] Donald R. Slutz. Massive stochastic testing of SQL. In *Proceedings of 24rd International Conference on Very Large Data Bases (VLDB)*, pages 618–622, 1998.

[40] Database Language SQL, 1992. http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt.

[41] SQLancer. https://github.com/sqlancer/sqlancer.

[42] SQLite. https://www.sqlite.org/index.html.

[43] SQLsmith. https://github.com/anse1/sqlsmith.

[44] Michael Stonebraker, Sam Madden, and Pradeep Dubey. Intel" big data" science and technology center vision and execution plan. *ACM SIGMOD Record*, 42(1):44–49, 2013.

[45] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. Industry practice of coverage-guided enterprise-level DBMS fuzzing. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE SEIP)*, pages 328–337, 2021.

[46] Yahoo says all three billion accounts hacked in 2013 data theft, 2017. https://www.reuters.com/article/us-yahoo-cyber/yahoo-says-all-three-billion-accounts-hacked-in-2013-data-theft-idUSKCN1C82O1.

[47] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium*, pages 745–761, 2018.

[48] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. SQUIRREL: testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 International Conference on Computer and Communications Security (CCS)*, pages 955–970, 2020.

A Examples of Bug-Triggering Queries

Listing 1-6 show 6 generated queries that trigger 6 bugs, respectively. Note that many bugs found by DynSQL have not been fixed yet, and related developers (e.g. developers in MySQL) hope that we do not publish the malicious queries triggering unfixed bugs, in order to protect their customers. Considering their concerns, we choose only the fixed bugs and show their corresponding queries. These 6 selected bugs are missed by both SQUIRREL and SQLsmith, and the queries have been simplified by us and related developers.

```
CREATE TABLE t1 (i1 INT);
INSERT INTO t1 VALUES (1),(2),(3);
CREATE VIEW v1 AS
   SELECT t1.i1
   FROM (
      t1 a JOIN t1 ON (
         t1.i1 = (
            SELECT t1.i1
            FROM t1 b)));
SELECT 1
FROM (
   SELECT count (SELECT i1 FROM v1)
   FROM v1
   ) dt1;
```

Listing 1: Generated query that crashes MariaDB 5.5-10.5

```
CREATE TABLE t1 (i1 INT PRIMARY KEY);
INSERT INTO t1 VALUES (62),(66);
CREATE TABLE t2 (i1 INT);
SELECT 1
FROM t1
WHERE t1.i1 = (
      SELECT t1.i1
      FROM t2
   UNION
      SELECT dt1.i1
      FROM (t1 AS dt1)
      WINDOW w1 AS (PARTITION BY t1.i1)
      LIMIT 1
   );
```

Listing 2: Generated query that crashes MariaDB 10.2-10.5

```
CREATE TABLE t1 (a INT NOT NULL PRIMARY KEY);
INSERT INTO t1 VALUES (0),(4),(31);
CREATE TABLE t2 (i INT);
DELETE FROM t1
WHERE t1.a = (
      SELECT t1.a
      FROM t2
   UNION
      SELECT DISTINCT 52
      FROM t2 r
      WHERE t1.a = t1.a
   );
```

Listing 3: Generated query that crashes MariaDB 10.2-10.5

```
CREATE TABLE t1 (id int);
CREATE VIEW v1 AS
   SELECT
      b AS a,
      b AS b
   FROM (
      SELECT id AS b
      FROM t1
      ) AS dt
   ORDER BY a,b;
WITH cte AS (
   SELECT dt.b
   FROM (
         (SELECT 11 AS b FROM v1) dt
      JOIN v1
      ON 1)
   )
SELECT 5 ;
```

Listing 4: Generated query that crashes MariaDB 10.2-10.5

```
CREATE TABLE t1 (a1 TEXT) engine=myisam;
SELECT c1 FROM (
   SELECT DISTINCT
      t1.a1 AS c1,
      t1.a1 AS c2,
      t1.a1 AS c3,
      ...
      t1.a1 AS c2591,
      t1.a1 AS c2592
   FROM t1
   ) dt;
```

Listing 5: Generated query that crashes MariaDB 10.2-10.5

```
CREATE TABLE t1 (
   t1_c0 INTEGER,
   t1_c1 TEXT,
   t1_c2 INTEGER,
   PRIMARY KEY (t1_c0));
ALTER TABLE t1 RENAME COLUMN t1_c2 TO t1_c3;
WITH cte_0 AS (
   SELECT
      ref_0.t1_c1 AS c0
   FROM
      t1 AS ref_0
   GROUP BY ref_0.t1_c1)
SELECT
   ref_2.c0 AS c0
FROM (
   (t1 AS ref_1 INNER JOIN cte_0 AS ref_2
      ON ((
         SELECT c0
         FROM cte_0
         ORDER BY c0
         LIMIT 1 OFFSET 5
      ) IS NULL))
   INNER JOIN (
      SELECT
         ref_3.c0 AS c0
      FROM
         cte_0 AS ref_3
      WHERE EXISTS (
         SELECT
            (SELECT max(t1_c0) FROM t1) AS c0
         FROM
            cte_0 AS ref_4
         ) AND 0<>0
      ORDER BY c0 ASC LIMIT 86
      ) AS subq_0
   ON ((ref_1.t1_c3 >= ref_1.t1_c3)
       AND (ref_1.t1_c3 <= ref_1.t1_c0)));
```

Listing 6: Generated query that crashes MySQL 8.0

B CVE details

We applied for CVE IDs of the bugs found by DynSQL, and 19 were assigned, including 7 for MySQL bugs and 12 for MariaDB bugs. Among them, CVE-2021-46667 and CVE-2021-46669 are considered to have high security impact. Table 8 shows the details of the 19 assigned CVEs. The column "File location" shows the file name and line number where the bug is triggered, and the column "Exploitation" shows the possible attack by exploiting the bug.

Table 8: The details of 19 assgined CVEs

DBMS	Bug type	File location	Exploitation	CVE ID
MySQL	Null-pointer dereference	MySQL/sql/item_subselect.cc:799	Denial-of-service	CVE-2021-2357
MySQL	Null-pointer dereference	MySQL/sql/sql_optimizer.cc:8881	Denial-of-service	CVE-2021-2425
MySQL	Null-pointer dereference	MySQL/storage/innobase/dict/dict0dd.cc:4184	Denial-of-service	CVE-2021-2426
MySQL	Null-pointer dereference	MySQL/strings/ctype-utf8.cc:5603	Denial-of-service	CVE-2021-2427
MySQL	Null-pointer dereference	MySQL/sql/item_subselect.cc:660	Denial-of-service	CVE-2021-35628
MySQL	Null-pointer dereference	MySQL/sql/sql_derived.cc:182	Denial-of-service	CVE-2021-35635
MySQL	Heap-buffer overflow	MySQL/sql/sql_optimizer.cc:4231	Data leakage	CVE-2022-21438
MariaDB	Null-pointer dereference	MariaDB/sql/sql_select.cc:25122	Denial-of-service	CVE-2021-46657
MariaDB	Null-pointer dereference	MariaDB/sql/field_conv.cc:204	Denial-of-service	CVE-2021-46658
MariaDB	Null-pointer dereference	MariaDB/sql/sql_lex.cc:2502	Denial-of-service	CVE-2021-46659
MariaDB	Null-pointer dereference	MariaDB/sql/sql_base.cc:6013	Denial-of-service	CVE-2021-46661
MariaDB	Use-after-free	MariaDB/sql/item.cc:7956	Data leakage	CVE-2021-46662
MariaDB	Null-pointer dereference	MariaDB/storage/maria/ha_maria.cc:2656	Denial-of-service	CVE-2021-46663
MariaDB	Assertion failure	MariaDB/sql/sql_select.cc:18576	Denial-of-service	CVE-2021-46664
MariaDB	Null-pointer dereference	MariaDB/sql/sql_select.cc:18647	Denial-of-service	CVE-2021-46665
MariaDB	Null-pointer dereference	MariaDB/sql/item.cc:3333	Denial-of-service	CVE-2021-46666
MariaDB	Integer overflow	MariaDB/sql/sql_lex.cc:3521	Remote code execution	CVE-2021-46667
MariaDB	Null-pointer dereference	MariaDB/storage/maria/ha_maria.cc:2782	Denial-of-service	CVE-2021-46668
MariaDB	Use-after-free	MariaDB/sql/sql_class.cc: 2914	Privilege escalation	CVE-2021-46669