DynSQL：a Software Testing Method Based on Stateful Fuzzing

1 Background and Motivation

With the rapid development of information technology, databases have been widely used in various industries. As we all know, there are many DBMSs on the market, such as MySQL, MariaDB, etc. There are also many vulnerabilities within them; if not properly guarded against, they may lead to data leakage, tampering, and other issues. Therefore, we urgently need a software testing tool to accurately uncover the vulnerabilities present in DBMSs.

Existing DBMS fuzzers are limited in generating complex and valid SQL queries because they cannot accurately capture the changes in DBMS state caused by the generated statements. Instead, they either generate only one complex statement in each query to avoid analyzing state changes, or they combine multiple relatively simple statements where the state changes can be easily inferred.

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

Currently, there are two commonly used fuzzers. SQLsmith is unable to establish dependencies between multiple statements, so it only generates one statement in each query. SQUIRREL tends to generate simple statements in queries, which also leads to it producing over 50% of invalid queries. Both have their corresponding drawbacks, hence proposing a practical solution——DynSQL to address these limitations is necessary and important for DBMS fuzz testing.

2 Goal of the Technology

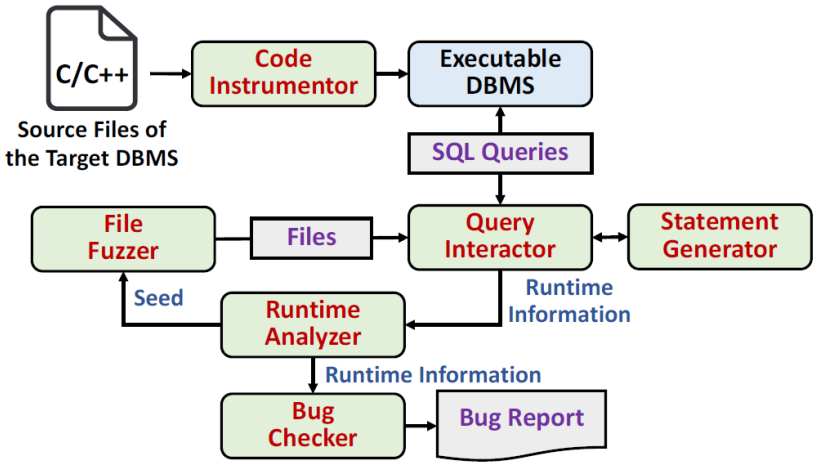
DynSQL is a novel Stateful Fuzzing method designed for database management systems (DBMS), with the primary purpose of effectively testing DBMS and discovering deep bugs. Specifically, DynSQL aims to address the limitations of existing DBMS fuzzers in generating complex and valid SQL queries. These limitations mainly stem from the fact that current tools heavily rely on predefined grammar models and fixed knowledge about DBMS, but fail to capture DBMS-specific state information. The method mainly achieves its purpose through Dynamic Query Interaction, utilization of state information, Error Feedback, an automated framework, practical effectiveness verification, and performance comparison.

Overall, the introduction of DynSQL is to enhance the security of DBMS in a more intelligent and automated manner. By generating complex and valid SQL queries that can trigger deep logic, it aims to discover and fix security vulnerabilities that could be exploited by attackers.

3 Framework of the Technology

3.1 The overall process of DynSQL

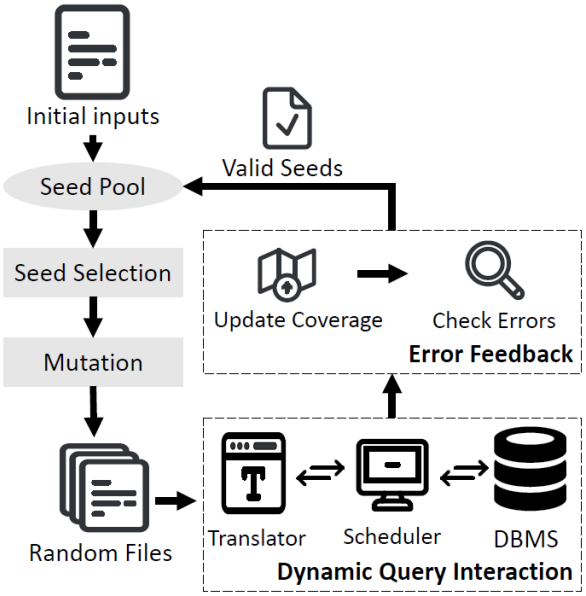
As shown in the figure, this is the main process framework of DynSQL. It can be seen that all six parts play their respective roles.



First, the Code Instrumentor compiles the input source files to generate an executable DBMS; the File Fuzzer performs traditional file fuzzing (such as AFL) to generate files based on given seeds; the files provided by the Fuzzer are transferred to the Dynamic Query Interactor, where the Query Interactor conducts interactive queries; the Statement Generator uses an internal Abstract Syntax Tree (AST) model to generate syntactically correct SQL statements according to the given database schema; the Runtime Analyzer analyzes the runtime information, identifies effective seeds based on error feedback, and passes the seeds to the Fuzzer.

3.2 The main process of Stateful DBMS Fuzzing

The following diagram illustrates the main process of our Stateful Fuzzing Testing. It involves core steps such as Dynamic Query Interaction and Error Feedback, as well as processes like the conveyance of Valid Seeds and the sifting of the Seed Pool.



First, the Initial Inputs are sent to our Seed Pool, where seeds that meet the requirements are selected from the pool, mutated, and corresponding random files are generated.

Then we proceed to Dynamic Query Interaction, the specifics of which will be discussed in Section 3.3.

Following that is Error Feedback, which helps us filter out some invalid test cases to enhance the validity of our queries, with details to be provided in Section 3.4.

We also need to return the effective seeds back to the Seed Pool for use in the next round.

3.3 The main process of Dynamic Query Interaction

This is the key step——the algorithmic process of Dynamic Query Interaction, which is mainly composed of three parts: the Scheduler Function, the Translator Function, and the CheckStatus Function.

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;
```

First, we input the file and the corresponding DBMS, and output the generated queries, code coverage, and status.

The Scheduler Function begins by obtaining the size of the file, initializing the DBMS to its initial state, and initializing a series of values, then enters a loop: it uses the Translator Function to generate SQL statements, adds the generated statements to the query list, executes the SQL statements and modifies the coverage, and uses the CheckStatus Function to check the

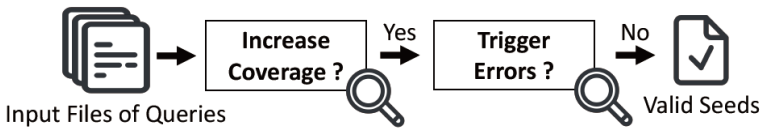
status.

The primary function of the Translator Function is to output SQL statements. It extracts the unread part from the file as a tmp file and then generates the corresponding SQL statement stmt based on the required schema.

The CheckStatus Function is mainly used to check the query status. If the DBMS crashes, it returns true. If the status is an error, it further checks whether it is a syntax or semantic error. If it is not, it returns true. If it is, it returns false.

3.4 The main process of Error Feedback

After undergoing the previous step of Dynamic Query Interaction, why do we still need the subsequent Error Feedback? This is because if we put invalid queries into the Seed Pool, the generated queries are likely to trigger the same syntax or semantic errors as the seed queries, without improving code coverage. Therefore, we need an algorithm to filter out invalid queries to enhance our code coverage. The specific process is shown in the figure below.



First, we input the Files of Queries and check for an increase in Code Coverage. if there is an increase, it indicates that the query has helped us discover new areas of defects, if not, it is discarded immediately. Then, Trigger Errors helps us check whether any errors have been triggered; if no errors have been triggered, it can continue to be used and placed as a valid seed in the Seed Pool for later use.

Error Feedback helps us to eliminate queries that do not increase coverage at the first step, and then judge whether there are any Triggered Errors, selecting those queries that have not triggered errors to be stored in

the pool, ensuring that all queries in our pool are valid.

4 Experimental Result

The experiment mainly verified the following four aspects: Can DynSQL find bugs in real-world DBMSs by generating complex and valid queries? How about the security impact of the bugs found by DynSQL?How do dynamic query interaction and error feedback contribute to DynSQL in DBMS fuzzing? Can DynSQL outperform other state-of-the-art DBMS fuzzers?

I will introduce the experimental results in this section.

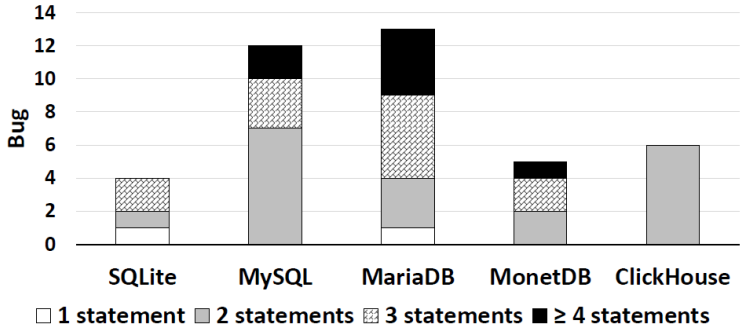
4.1 Can complex and valid queries be generated to discover bugs?

Through the experiment, we obtained the following results.

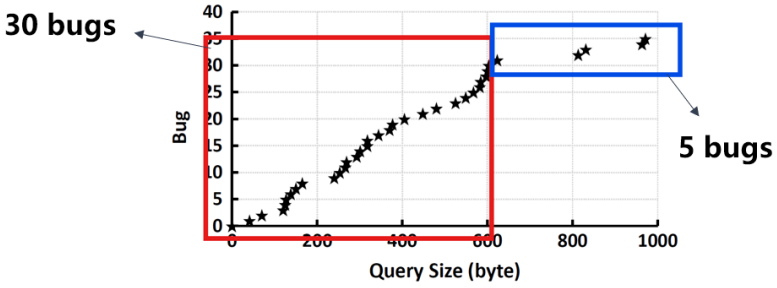
DBMS	Bug			Validity	
	Found	Confirmed	Fixed	Statement	Query
SQLite	4	3	3	279K/286K	24K/30K
MySQL	12	12	6	91K/96K	9.4K/13K
MariaDB	13	13	6	170K/175K	17K/21K
PostgreSQL	0	0	0	154K/160K	14K/18K
MonetDB	5	5	5	70K/72K	7.0K/8.6K
ClickHouse	6	5	1	74K/77K	7.5K/10K
Total	40	38	21	838K/866K	79K/101K

We can observe that DynSQL has discovered a multitude of bugs in DBMS such as MySQL and MariaDB, with the majority (31/40) being caused by memory issues. In terms of validity, DynSQL achieves a 78% valid query rate (79K/101K) in SQL queries, and a 97% valid statement rate (838K/866K) in SQL statements. The above results demonstrate that DynSQL is capable of generating valid and complex SQL statements to detect bugs within DBMS.

From the perspective of the number of statements in queries that trigger DBMS bugs, the majority of bugs are triggered by queries containing 2 to 3, or even 4 SQL statements, which was almost impossible for the previous two fuzzers to accomplish.



In terms of the size of queries that trigger bugs, we found that most queries are concentrated within the 0-600 byte range, and there are even queries exceeding 1000 bytes in size, which also demonstrates the excellence of DynSQL.



4.2 The Security Impact of Discovered Bugs

The following table lists the types of bugs discovered by DynSQL.

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

From this, we can see that Null-pointer dereferences account for 45% of the total number of bugs, and there are many other types of bugs, such as Use-after-free, Integer overflow, and so on.

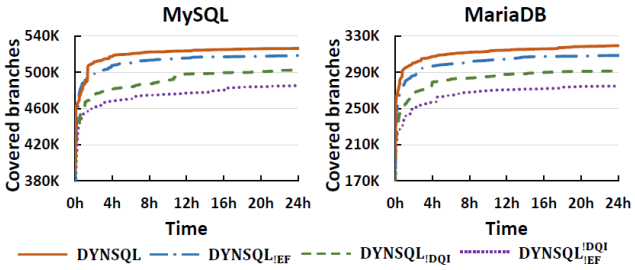
Through the mining of vulnerabilities, we have identified many high-risk vulnerabilities, including a series of CVE vulnerabilities, such as: the integer overflow bug (CVE-2021-46667) in MariaDB, and the use-after-free bug (CVE-2021-46669) in MariaDB, and so on.

4.3 Sensitivity Analysis

To illustrate the role of Dynamic Query Interaction and Error Feedback in DynSQL, we conducted tests by disabling these two techniques respectively. The experimental results are shown below.

DBMS	$\text{DynSQL}_{\neg IEF}^{\neg DQI}$				$\text{DynSQL}_{\neg DQI}$				$\text{DynSQL}_{\neg IEF}$				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

From complete disablement to complete enablement, the number of bugs discovered gradually increases. The data in between also shows that Dynamic Query Interaction plays a key role, with significant improvements in the efficiency of statements and queries, and a gradual increase in code coverage.



Through another experiment (as shown in the figure above), we found that Dynamic Query Interaction and Error Feedback can help us cover code branches more quickly.

Combining the results of the two experiments, we can see that Error Feedback can help the fuzzer generate more Valid Queries to detect more errors. However, Error Feedback cannot increase the complexity of queries; Dynamic Query Interaction uses DBMS state information to improve both the complexity and the validity of queries.

Both have their respective functions and can help us find more bugs.

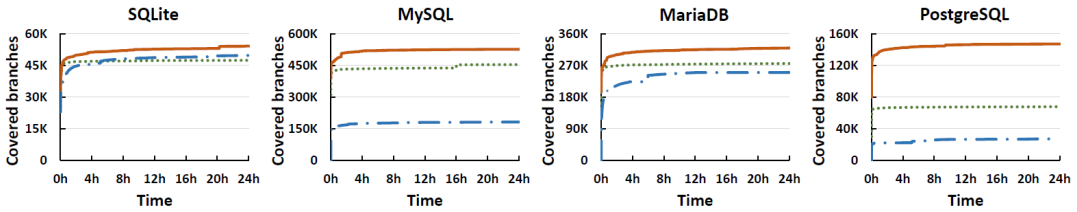
4.4 Comparison to Existing DBMS Fuzzers

Here, we compare DynSQL with two mainstream processors, SQLsmith

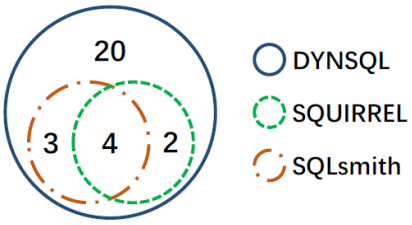
and SQUIRREL, aiming to discover its strengths. The following table shows the comparison results.

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

It can be seen that our DynSQL has discovered more bugs compared to the former two.



In terms of code coverage, DynSQL has its unique advantages compared to the others. DynSQL generates more complex statements than SQUIRREL, and longer statements than SQLsmith. Therefore, the final conclusion is: DynSQL covers the broadest range of code branches!



From the figure above, we can see that SQUIRREL and SQLsmith each have their own strengths, but also shortcomings. For example, SQLsmith can only generate one statement per query, and SQUIRREL is unable to generate logically complex statements, which leads to omissions in the query results of both methods.

However, our DynSQL not only found the 9 bugs discovered by the other two processors but also found another 20 bugs. This is because these additional 20 bugs require more than three complex statements, which is almost impossible for SQUIRREL and SQLsmith to accomplish.

5 Reflection and Perception

This Dynamic Query Interaction Software Testing Technique has helped major DBMS vendors uncover many vulnerabilities, as shown below, and these vulnerabilities have all been assigned CVE IDs.

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

Certainly, existing DBMS fuzzing tools have limitations in generating complex and valid queries. DynSQL addresses this issue through Dynamic Query Interaction and Error Feedback, but this approach may require substantial computational resources and time to generate and process complex SQL queries. Therefore, to address these shortcomings, we propose reducing the generation of invalid queries, detecting other types of vulnerabilities (such as logical errors and performance issues), and automatically extracting AST rules.

Through tools like DynSQL, I have come to realize that the security testing of database management systems is a complex and critical task. It allows for more in-depth testing of DBMS, uncovering and fixing potential security vulnerabilities, thereby enhancing the overall system security.