Sequence-Oriented DBMS Fuzzing

Jie Liang*, Yaoguang Chen†, Zhiyong Wu*, Jingzhou Fu*, Mingzhe Wang*‡, Yu Jiang*✉
Xiangdong Huang*, Ting Chen§, Jiashui Wang†¶, Jiajia Li†
* KLISS, BNRist, School of Software, Tsinghua University, China
† Ant Group, China
‡ ShuiMuYuLin Ltd, China
§ UEST-Center for Cybersecurity, China
¶ Zhejiang University, China

***Abstract*—The SQL specification consists of hundreds of statement types, which leads to difficulties in DBMS fuzzing: state-of-the-art works generally reuse the statements of predefined types; the limited types cannot cover the full input space and test the corresponding logic consequently. In this paper, we propose LEGO, a fuzzer to generate SQL sequences with abundant types to improve DBMS fuzzing coverage. The key idea of sequence generation is *type-affinity*, which indicates the meaningful occurrence of SQL type pairs (e.g., `INSERT` and `SELECT`). During each fuzzing iteration, LEGO first proactively explores SQL statements of different types and analyzes affinities with coverage feedback. Next, when a new affinity is discovered, LEGO synthesizes new SQL sequences containing the types progressively.**

We evaluate LEGO on PostgreSQL, MySQL, MariaDB, and Comdb2 against SQLancer, SQLsmith, and SQUIRREL. The sequence-oriented fuzzing helps LEGO outperform other fuzzers on branch coverage by 44%–198%. More importantly, in the continuous fuzzing, LEGO has discovered 102 new vulnerabilities confirmed by the corresponding vendors, including 6 bugs in PostgreSQL, 21 bugs in MySQL, 42 bugs in MariaDB, and 33 bugs in Comdb2. Among them, 22 CVEs have been assigned due to their severe security influences.

***Index Terms*—DBMS fuzzing, SQL Type Sequence**

I. INTRODUCTION

Database management systems (DBMSs) are crucial for modern data-intensive systems [49]. Serving as the intermediary between the user and the database, a DBMS offers the solution to optimize and manage the storage and retrieval of data [6, 23, 27, 30]. Security vulnerabilities, especially memory bugs such as buffer overflow are particularly dangerous for DBMS because they might allow attackers to steal information, tamper data, crash systems, and bring heavy losses [4, 10, 32, 35, 51, 54]. Existing works have focused on logic, performance, and memory bugs in DBMSs. To test logic and performance bugs, many representative schemes utilize differential testing [16, 35, 39]. Recently, many fuzzing methods are applied to detect memory bugs of DBMSs, focusing on generating valid SQL queries [37, 46, 53, 54].

The abundance of **SQL Type Sequences** contained by generated test cases is crucial for fuzzing a DBMS. As Figure 1 shows, a test case (surrounded by the grey box) is an input for a DBMS, and it always consists of a sequence of SQL statements [24, 40, 50] (surrounded by the orange box). Generally, SQL statements have hundreds of types. Type is the category divided by the same functionalities, like `SELECT` and `INSERT`. SQL Type Sequence is a sequence of the types for each SQL statement in a test case (surrounded by the red box). SQL Type Sequence implicitly describes the semantic characteristics of a test case, and its abundance is important to cover the functionalities of a target DBMS. First, specific DBMS logic must be triggered by a specific type of statement. For example, testing `SELECT` statements alone cannot exercise the logic of `INSERT` statements. Second, specific DBMS logic must be triggered by statements of a specific order. For example, `CREATE` a table then `INSERT` into it makes sense, while `INSERT` first and `CREATE` the table next renders the first statement useless. In short, the abundance of SQL Type Sequences is determined by both the combination and permutation of statement types.

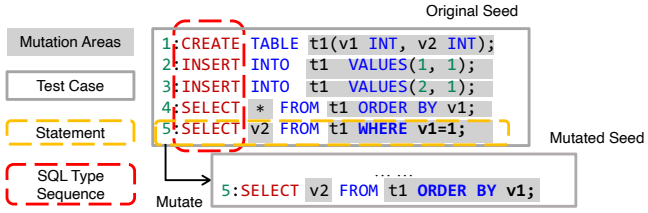

Fig. 1. Breakdown of inputs generation in mutation-based DBMS fuzzing. Executions are performed at test case level, where a test case consists of a sequence of *statements*. To generate mutated seeds from an original seed, fuzzers mutate the *inner* structure given a predefined SQL *type*.

However, existing fuzzing works have put hard work into generating valid test cases since SQL is a highly-structured language, while neglecting to enrich **SQL Type Sequences**. In general, fuzzers could be divided into generation-based [35, 37] and mutation-based [15, 51, 54]. Generation-based fuzzers generate test cases according to custom rules. Consequently, the number of states and relationships between sequences are limited by the rules. For example, SQLsmith mainly generates `SELECT` statements, so it will only explore limited state space and the relationships are limited to the same type.

Mutation-based fuzzers select an test case (also called a seed) from an input pool, generate many new inputs with mutation, and save the input back to the pool if its execu-

✉Yu Jiang is the corresponding author.

tion triggers new code regions. To ensure the correctness of generated seeds, existing works focus on the inner structure of individual statements. For example, Figure 1 colorizes the available mutation areas in each statement with a grey background. Most fuzzers generally first select one individual statement and perform mutating operations on the mutation areas to generate a new test case. For example, SQUIRREL first selects the 5th statement, and then it changes the inner structure from "WHERE v1=1" to "ORDER BY v1". Note that the type of the mutated statement is still "SELECT", and both the original and mutated seed have the same SQL Type Sequence. Therefore, the abundance of SQL Type Sequence does not increase. As a result, it is hard for existing fuzzers to explore the full input space and cover the corresponding logic in the target DBMS. Consequently, generating abundant SQL Type Sequences can be a promising method to improve the coverage and overall effectiveness of DBMS fuzzing.

However, generating abundant SQL Type Sequences can be challenging. First is the state explosion. A DBMS generally has hundreds of SQL statement types [31], even though given a maximum sequence length, the total number of all the possible sequences is numerous. For example, suppose a DBMS has 100 statement types, even though limiting the sequence to only 5 statements, possible unique sequences will reach 10 billion. Second, many statement sequences make little sense because the types contained may not be related to each other. For example, if one statement creates a trigger while another changes the permissions to access data, the affinity between them is low. Thus the sequence composed by them has less contribution to the testing effectiveness. Third, the test cases with generated SQL Type Sequences may not be suitable for fuzzing. Suppose a test case is very long and has many repeated subsequences, a fuzzer might get stuck in handling it although it contains various types.

In this paper, we propose LEGO, which improves the effectiveness of DBMS fuzzing in finding memory-safety bugs by increasing the abundance of SQL Type Sequences. To tackle the above challenges, LEGO adopts a progressive approach. The key concept is *type-affinity*, which describes the pattern of composing two SQL statements. Specifically, it is a chronological relation between two adjacent SQL statement types contained in a test case. First, LEGO employs proactive affinity analysis guided by DBMS implementation code coverage feedback to explore type-affinities at the beginning of each fuzzing iteration. It picks an existing test case from the corpus, changes each statement to another type, and determines its significance by analyzing the coverage. If the change results in new code coverage, then an affinity will be recorded. Next, LEGO exploits the affinity by synthesizing new sequences it induces to further increase coverage. When a new affinity is discovered, LEGO permutes all SQL Type Sequences containing the affinity with a limited length. The sequences are then instantiated to executable test cases. With sequence-enriched test cases progressively synthesized from affinities discovered by proactive exploration, LEGO continuously explores the state space of target DBMSs.

We evaluate LEGO on the latest version of PostgreSQL, MySQL, MariaDB, and Comdb2 against SQLancer, SQLsmith, and SQUIRREL. The sequence-oriented fuzzing helps LEGO cover 198%, 44%, and 120% more branches than SQLancer, SQLsmith, and SQUIRREL on average, respectively. More importantly, in the continuous fuzzing (i.e., constantly running fuzzing without stopping it until the code is modified [29]), LEGO finds 102 new vulnerabilities while others find 11 of them in total. The vulnerabilities include 6 bugs in PostgreSQL, 21 bugs in MySQL, 42 bugs in MariaDB, and 33 bugs in Comdb2. Among them, 22 bugs are confirmed as CVEs in the U.S. National Vulnerability Database.

II. SQL TYPE SEQUENCE

Basic concepts. *Database Management Systems* (DBMSs) refer to the software used to manage the storage and retrieval data in databases [49]. *Structured Query Language* (SQL) is a domain-specific language used to interact with a DBMS to mange the data within it [50]. *SQL statements* are the smallest execution unit fed into a DBMS. *Test cases* or *queries* are used as input and they are the objects of the basic operations (such as mutation) for DBMS fuzzers [54], also known as seeds. A test case always consists of a sequence of SQL statements.

SQL statements types. In general, SQL statements have hundreds of types [31]. Specifically, a statement type defines one certain kind of specific operation on a certain type of object. For example, `CREATE TABLE` and `CREATE VIEW` are two types. Statement types could be roughly divided into four categories: Data Definition Language (DDL, e.g., `CREATE TABLE`), Data Query Language (DQL, e.g., `SELECT`), Data Manipulation Language (DML, e.g., `INSERT`), Data Control Language (DCL, e.g., `GRANT`) [40, 50]. Besides them, some other types of statements are used to deal with the transaction within the database (e.g., `COMMIT`).

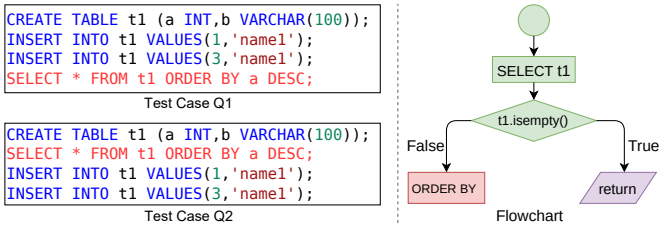

Fig. 2. An example to illustrate the importance of importing SQL Type Sequence. The left is two test cases and the right is the flowchart to make the plan for the `SELECT` statement. The two test cases on the left contain the same statement combinations. However, the same `SELECT` statements will execute different branches due to the different orders in SQL Type Sequence.

Definition. *SQL Type Sequence* is a sequence of the types for each *SQL statement* in a *test case*, which abstracts the execution order of *SQL statements* inside a test case by type. Take Figure 2 as an example, the SQL Type Sequences of the test case Q1 is "`CREATE TABLE`→ `INSERT`→ `INSERT`→ **`SELECT`**", while the SQL Type Sequences of the test case Q2 is "`CREATE TABLE`→ **`SELECT`**→ `INSERT`→ `INSERT`". The two test cases have the same SQL statements, but the different SQL Type Sequences.

Importance of abundant SQL Type Sequences. SQL Type Sequences implicitly characterize the semantics of test cases. The abundance of SQL Type Sequences is important for adequate DBMS fuzzing. First, the richness of types contained in sequences is a prerequisite for covering the various functions of a DBMS. More importantly, some code logic must be reached by executing some specific sequences. Specifically, even for the exact same SQL statements, various permutations will constitute different sequences, and each of which may cover completely different code regions. For example, the two test cases Q1 and Q2 in Figure 2 have the same combinations of four statements (i.e., 1 `CREATE TABLE`, 1 `SELECT`, and 2 `INSERT`). However, Q1 gets sorted data but Q2 obtains empty results because of the different execution orders. Specifically, Q1 fetches the data after inserting them, while Q2 queries the data before they are prepared.

Finally, abundant SQL Type Sequences with statement-level structure and data mutation can further facilitate fuzzing. Most existing DBMS fuzzers are skilled in mutating structure and data in a single SQL statement, not sequences. Abundant type sequences can increase the breadth of the state space explored by fuzzing, and fine mutations on top of that can further increase the depth of exploration. In summary, promoting the abundance of SQL Type Sequences in the generated test cases facilitates the exploration of the state space of fuzzers, thus enabling the discovery of more potential bugs. However, generating abundant type sequences is fraught with challenges.

Challenge in generating abundant SQL Type Sequences. Arbitrarily permuting or combining types and listing all possibilities are two straightforward ways to generate abundant SQL Type Sequences. However, neither of them is practical because of the following challenges:

C1: The full state space of SQL Type Sequences is enormous. Arbitrarily permuting can only explore limited space, while listing all possibilities will suffer from state explosion. To deal with different business scenarios, enterprise DBMSs usually define lots of statement types for different functionalities. For example, PostgreSQL's manual [31] describes 188 types of SQL statements. Assuming that a test case contains 20 statements on average, the number of all possible sequences is 3×10^{45}. In our experiment, SQUIRREL executes 10–60 test cases per second, and it would take more than 10^{35} years to execute all the possible SQL Type Sequences.

C2: Many sequences of SQL types are meaningless. Consequently, listing all sequences or arbitrarily permuting types suffer from meaningless SQL Type Sequences. In other words, generating them does not contribute to the abundance. The meaningless is reflected in two aspects: ① Many statement sequences may easily bring semantic errors. For instance, if "`SELECT * FROM t2`" is executed before "`CREATE TABLE t2 (v0 int)`", their coverage is low due to semantic errors. ② Many statement types are not closely related, and forming them into a sequence does not cover new logic. For example, one statement creates a table, while another changes the permissions to access specific data. There may be no relationship between them.

C3: The test cases with generated SQL Type Sequences may not be suitable for fuzzing. Fuzzing is computationally intensive and requires a large number of executions with generated test cases to find bugs. Consequently, fuzzers prefer seeds with high coverage that can be run quickly to perform mutations. Even though a test case has abundant SQL Type Sequences, it may still be hard to assist fuzzing. For example, in our experiments, we find a seed fed to SQUIRREL that contains 945 SQL statements. It repeatedly calls hundreds of `INSERT` statements to store data. These statements have very similar behaviors, contributing little to coverage but increasing the workload to parse and execute. As a result, SQUIRREL hung for 23 minutes while executing this seed.

Status of existing fuzzers. Existing fuzzers mainly focus on generating syntactically and semantically correct seeds, while neglecting to generate abundant SQL Type Sequences. Generation-based fuzzing and mutation-based fuzzing are two main types. Generation-based fuzzers (e.g., SQLsmith and SQLancer) generate seeds based on custom rules. To meet challenges C1 and C2, a compromise solution is to manually add a large number of rules for generating sequences. However, the solution can be labor-intensive, while the abundance remains limited. For challenge C3, it is possible to simplify the rules to improve execution speed. But the simplification is likely to decrease coverage. Mutation-based fuzzers (e.g., SQUIRREL and RATEL) mutate seeds by changing existing seeds, but most of them only change the structure or data in individual statements. Therefore, the sequence and the relationship the test case contained will not be changed. For challenges C1 and C2, the key technical difficulty is generating meaningful and abundant sequences from the large state space. However, existing fuzzers lack approaches to address that. For challenge C3, existing approaches include selecting more times of seeds with fast execution or trimming seeds guided by coverage. However, they may still produce large seeds that tend to make them stuck.

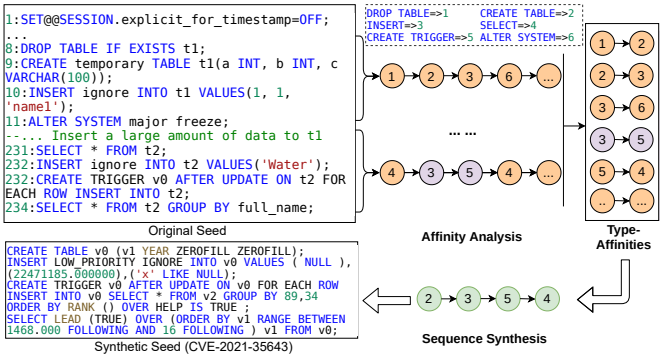

Fig. 3. An example to show the process of affinity analysis and sequence synthesis. For a seed that hits new branches, LEGO first analyzes type-affinities in original SQL Type Sequence. With the analyzed affinities, LEGO synthesizes more SQL Type Sequences and instantiates them into test cases.

Basic Idea of LEGO. LEGO explores the SQL type space to proactively analyze the affinities between types. It then exploits these affinities to generate high-quality test cases,

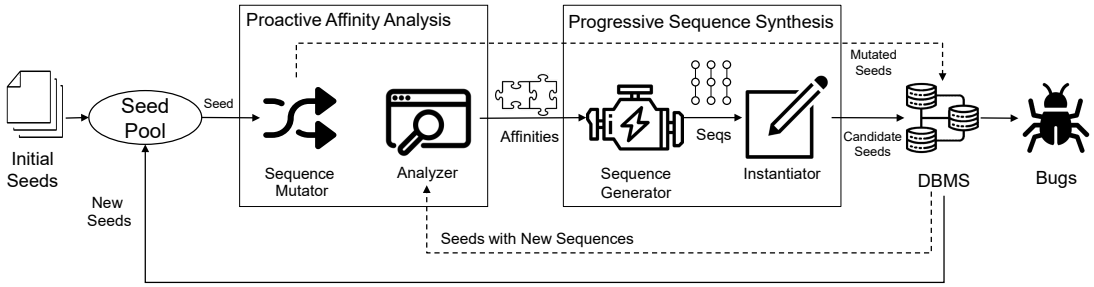

Fig. 4. Design of LEGO. 1) In Step 1, LEGO explores the SQL type space to proactively analyze the affinities. It picks an existing seed from the seed pool, performs sequence-oriented mutation to change each statement's type. If the mutated seed covers new branches in the target DBMS, then the type-affinity caused by type changes will be recorded. 2) In Step 2, LEGO exploits type-affinities to synthesize test cases progressively. When a new affinity is discovered, LEGO permutes all SQL Type Sequences containing the type-affinity. The sequence is then instantiated to executable test cases to feed into the target DBMS. With sequence-enriched seeds progressively synthesized from affinities discovered by proactive exploration, LEGO continuously explores the state space of the target DBMS and finds its bugs.

which contribute to the abundance of SQL Type Sequence and are suitable for fuzzing.

To address challenges C1 and C2, LEGO introduces **type-affinity** to abstract the problem of synthesizing meaningful SQL Type Sequences. Specifically, type-affinity is a chronological relation between the types of adjacent SQL statements. LEGO uses type-affinity to determine what type of statements should be concatenated after an existing one, thus enabling the exploration of a wide range of state spaces while ensuring that meaningful sequences are generated. LEGO first explores new type-affinities by performing sequence-oriented mutations on existing test cases. To ensure that affinities are meaningful, LEGO performs affinity analysis only on seeds that cover new branches in the target DBMS. For challenge C3, to be suitable for fuzzing and to generate abundant sequences, we limit the maximum sequence length while using sequence synthesis to accommodate various sequence lengths. Although some bugs that can only be triggered by long repeated SQL sequences. However, handling them may degrade the performance of fuzzer or even cause fuzzer to be stuck.

Figure 3 shows the process of LEGO to produce a test case that crashes MySQL server. The bug has been assigned CVE-2021-35643 by Oracle [28] because of the severe consequences: it could be used to break down the MySQL servers directly via the network. To generate the test case, a specified SQL Type Sequence should first be synthesized. For simplicity, we use ① to ⑥ to represent the type "`DROP TABLE`", "`CREATE TABLE`", "`INSERT`", "`SELECT`", "`CREATE TRIGGER`", and "`ALTER SYSTEM`", respectively. Note the original test case is rather long and the figure only shows a part of that. LEGO first extracts type-affinities in its SQL Type Sequence. For example, from line 8 to 11, the sub sequence is " ①→ ②→ ③→ ⑥", and from line 231 to 234, the sub sequence is " ④→ ③→ ⑤→ ④". From the SQL Type Sequence, LEGO finds a new type-affinity, namely "an `INSERT` statement could be followed by a `CREATE TRIGGER` statement" (i.e., ③→ ⑤ in the figure).

With the new type-affinity, LEGO synthesizes new sequences containing it and instantiates the sequences into test cases. The synthesized test cases are usually short in sequence length, simple in SQL structures but abundant in SQL types, and some of them may cover new coverage or directly trigger new crashes. In Figure 3, we also show a synthesized test case that crashes the MySQL server. The sequence of it is " ②→ ③→ ⑤→ ④". We can find that the new seed is much shorter while having a different sequence. Seeds like it can be executed at a high speed and help fuzzer to explore new state-space quickly. Based on the specified sequence, LEGO finally triggers the crash.

III. DESIGN OF LEGO

Figure 4 illustrates the overall sequence-oriented fuzzing process of LEGO, which mainly contains two steps in each iteration: 1) *proactive affinity analysis* and 2) *progressive sequence synthesis*. The following text presents the details.

A. Proactive Affinity Analysis

As aforementioned, abundant sequences in generated test cases can improve the effectiveness of DBMS fuzzing. However, arbitrarily permuting or combining various types to form SQL Type Sequences are difficult to work since the statements may not be closely related to each other. In addition, the space of all possible sequences is huge. In contrast, LEGO proactively produces new sequences by employing sequence-oriented mutation. Sequences that cover new code regions of mutated seeds will be considered meaningful and the type-affinities in them will be extracted.

1) Type-Affinity: In reality, the sequence of statement types in a test case always appear with a certain pattern. For example, the type sequence "`CREATE TABLE`→ `INSERT`→ `SELECT`" is a common pattern to create, update, and query data. To better describe the pattern and abstract the problem, we define statement **type-affinity**, which is a chronological relation between adjacent statements. Specifically, if one statement follows close to another statement, we consider the types of the two statements to have a chronological relation. We use the partially ordered tuple $(type_1, type_2)$ to represent this relation as a type-affinity, which means that $type_1$ could be followed by $type_2$. For example, when a `INSERT` statement follows close to a `CREATE TABLE` statement in a real seed, then type `INSERT` has the chronological relation to type

CREATE TABLE. Then the type-affinity (CREATE TABLE, INSERT) is built. Based on type-affinities, LEGO continually selects the next statement type to synthesize new sequences and instantiates them to test cases that have rich semantics.

2) Proactive Sequence-Oriented Mutation: To analyze type-affinities, LEGO explores the sequence state space by sequence-oriented mutation to produce SQL Type Sequences different from the current test case. LEGO follows the theories behind coverage-guided fuzzing to detect meaningful sequences. Coverage-guided fuzzers gradually explore the program's state space with coverage feedback heuristically. LEGO reuses this principle to find meaningful sequences inside the huge state space of SQL Type Sequences. Specifically, when a mutated SQL Type Sequence finds new branches, the combination is regarded as meaningful. Consequently, we record the type-affinities resulting from the change for further sequence synthesis. On the contrary, when a generated sequence does not cover new branches, it is not good for expanding coverage and will be discarded.

Algorithm 1: Sequence-Oriented Mutation

```
Input : Input seed: Q,
        Type-affinities: T,
        Target DBMS: D
for statement s ∈ Q do
    Q1 = substitute(s, Q);
    if hitNewBranch(Q1, D) then
        analyzeAffinities(Q1, T);
    end
    Q2 = insertAfter(s, Q);
    if hitNewBranch(Q2, D) then
        analyzeAffinities(Q2, T);
    end
    Q3 = delete(s, Q);
    if hitNewBranch(Q3, D) then
        analyzeAffinities(Q3, T);
    end
end
```

Algorithm 1 illustrates the process of sequence-oriented mutation. When a seed needs to be mutated, we mutate each of its statements in turn by substitution, insertion, and deletion. ① *Substitution*: The mutation changes the current statement with another statement. It first randomly selects a different statement type and then instantiates it into a statement to replace the current one. To fix semantic errors, it will analyze the dependencies following the methods of SQUIRREL and refill in SQL data. The change in type results in a change of the SQL Type Sequences. If the mutated seed hits new branches, it will be preserved and its type-affinities will be analyzed. ② *Insertion*: This mutation adds a random SQL statement after the current statement. Like the substitution, it selects a random statement type, instantiates it to a SQL statement, and fixes semantic errors. The mutated seed covering new branches will be retained and its type-affinities will be recorded. ③ *Deletion*: The mutation removes the current statement to compose a new test case. The test case will be validated and refilled in concrete data. LEGO will also analyze the affinities brought by the deletion from the mutated seed that finds new branches.

We take Figure 5 to illustrate the process of sequence-oriented mutation. The original seed has 5 statements. Its SQL Type Sequence is "CREATE TABLE→ INSERT→ INSERT→ UPDATE→ SELECT". When the 4th statement is being mutated, suppose we substitute it with a DELETE statement. The change results in a new SQL Type Sequence, and two new type-affinities: "INSERT→ DELETE" and "DELETE→ SELECT" are also generated. Second, we insert after it with a DELETE statement. The mutation helps us find two new type-affinities: "UPDATE→ DELETE" and "DELETE→ SELECT". Finally, we delete the 4th statement. One new type-affinity "INSERT→ SELECT" is created.

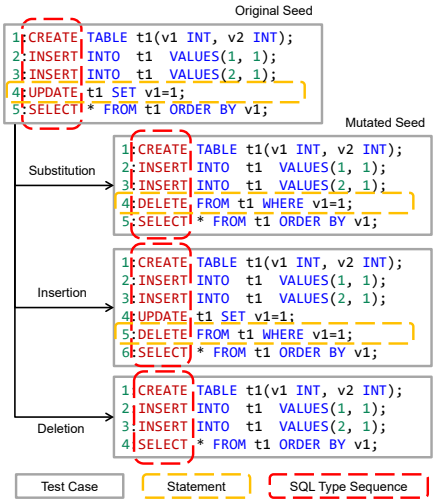

Fig. 5. LEGO employs sequence-oriented mutation to compose new SQL Type Sequences. Mutations include substitutions, insertions, and deletions.

3) Type-Affinity Analysis: As mentioned earlier, type-affinities abstract the principle of combining different SQL statements. Analyzing affinities helps us list new permutations of SQL statement types, so that abundant and meaningful sequences can be composed. Sequence-oriented mutation proactively explores the affinities by producing test cases with different SQL Type Sequences. To extract the type-affinities of test cases, the prerequisite is identifying the SQL types of the statements. This is challenging because a DBMS may have many or even its own unique types. For example, PostgreSQL defines 188 SQL statement types, 1,025 keywords, and 349 subclause types (e.g., expression and "WITH" clause). We use an AST model to accurately identify statement types. The model is built from DBMS's grammar specification to support identifying all statement types and other structures.

Algorithm 2 illustrates the process of affinity analysis. It parses the test case and saves the result into type-affinity T: a Map whose key is the statement type and value is a set of the next statement types that could follow closely to the key. The algorithm parses each SQL statement in order. It identifies the type of the current SQL statement. With the recorded type of the last statement, it learns the type-affinity: $lastType \rightarrow currentType$. Specifically, it first parses each

Algorithm 2: Type-Affinity Analysis

```
   Input  : Test case: Q,
            Type-affinity map (type→Set⟨type⟩): T
1  lastType = NULL;
2  for statement s in Q do
3      currentType = parse(s);
4      if lastType != NULL then
5          if lastType == currentType then
6              continue;
7          end
8          if lastType ∉ T then
9              affinity = Set();
10             T[lastType] = affinity;
11         end
12         add(T[lastType], currentType);
13     end
14     lastType = currentType;
15 end
```

SQL statement in the test case to get its type (line 3). If $currentType$ is the same as $lastType$, it will ignore it, since composing only one type does not contribute much to the abundance of SQL Type Sequences (lines 5-7). If the type of the last statement ($lastType$) does not recorded, then a set is created and T adds this set as the value for $lastType$. Next, the algorithm adds the $currentType$ into the set of $lastType$ in T (lines 8-12). After parsing one statement, the algorithm will update $lastType$ (line 14) and try the next SQL statement. After processing all statements, LEGO analyzes all possible type-affinities appearing in the test case.

B. Progressive Sequence Synthesis

The analyzed type-affinities supply the possibility to synthesize abundant and meaningful sequences of SQL statements. Beginning from specific starting statement types (e.g., `CREATE TABLE`), LEGO progressively synthesizes all possible SQL Type Sequences shorter than a specified length in accordance with type-affinities. Take Figure 6 (a) as an example, assuming that the root node represents a starting type and each path from the root to other nodes represents a SQL Type Sequence. With analyzed affinities, LEGO tries to produce all SQL Type Sequences. Then LEGO instantiates sequences into test cases with non-repetitive structures to explore the state space of the target DBMS.

As new type-affinities continue to be discovered, the challenge is progressively synthesizing SQL Type Sequences with the newly found affinities. In other words, LEGO tries to generate all possible type sequences not longer than the given length. When a new affinity is found, for effectiveness, we want to only synthesize all new type sequences containing it. As Figure 6 shows, when a new affinity "④→ ⑥" is found, we do not want to re-synthesize all the SQL Type Sequences, but generate the sequences only containing this affinity, which is marked as the red arrows in Figure 6 (b).

To achieve the progressive synthesis, a data structure called *Prefix Sequence* is designed to record all the generated sequences of specified lengths which end with a certain SQL

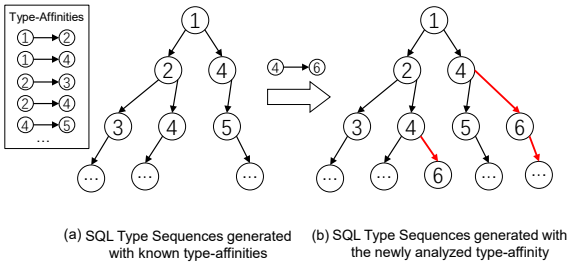

Fig. 6. When one new affinity is found, LEGO only synthesizes all new type sequences containing it.

statement type. Specifically, suppose we use a vector S to store all sequences that have been generated that are not longer than length LEN. *Prefix Sequence* is a `Map` whose keys are a pair (τ, λ) and whose values are a vector of the indexes in S that corresponding sequence ends with the type τ and has the length λ. When LEGO finds a new type-affinity $t_1 \rightarrow t_2$, it will find all possible prefix sequences ending with t_1 that are shorter than LEN according to *Prefix Sequence*, and then it will synthesize all new sequences that are not longer than LEN containing this new type-affinity.

Algorithm 3: Progressive Sequence Synthesis

```
   Input  : Length of the target sequence: LEN,
            New type-affinity: t1 → t2,
            Prefix sequence: PS,
            Type-affinity map: T,
            Vector of generate sequences: S
1  for level ← 1 to LEN − 1 do
2      prefixSeqIndex = PS([t1, level]);
3      if isEmpty(prefixSeqIndex) then
4          continue;
5      end
6      for seqIndex ∈ prefixSeqIndex do
7          seq = clone(S[seqIndex]);
8          pushBack(seq, t2);
9          pushBack(S, clone(seq));
10         addRecord(PS, t2, level + 1,length(S)-1);
11         listseq(level + 1, t2, seq);
12     end
13 end
14 Function listSeq(level, nodeType, seq):
15     if level ≥ LEN then
16         return;
17     end
18     for nextType ∈ T[nodeType] do
19         pushBack(seq, nextType);
20         listSeq(level + 1, nextType, seq);
21         pushBack (S, clone(seq));
22         addRecord(PS, nextType,
             level + 1,length(S)-1);
23         pop(seq);
24     end
25 End Function
```

Algorithm 3 shows the overall process of synthesizing new sequences when a new type-affinity $t_1 \rightarrow t_2$ is found. The possible sequences containing the affinity will be saved in vector S. For each allowed length, the algorithm checks for the

existence of an already generated sequence with that length, ending with t_1 in the new type-affinity. If not, the process will try the next length (lines 2-5). Or else, it will first add t_2 after each found prefix sequence seq and push back the clone of seq to S (lines 6-9). The index of the output sequence is also recorded into PS (line 10). Then it will synthesize all sequences based on seq. Note that because $t_1 \rightarrow t_2$ is a new type-affinity, all sequences generated based on that are new sequences. Then the algorithm calls the recursive function $listSeq$ to list all possible sequences with the specified length (line 11). Function $listSeq$ (lines 14-25) tries to compose all possible next types based on the current type and all type-affinities. Each synthesized sequence will be recorded. When the sequence length reaches the specified limit, it will return to try the next combination. For example, suppose the length of target sequence is 2, current sequence is "`CREATE TABLE`", type-affinity is "`CREATE TABLE`$\rightarrow$ [`INSERT, SELECT`]", then we get all possible sequences of length 2: "`CREATE TABLE, INSERT`" and "`CREATE TABLE, SELECT`".

After synthesizing all the sequences, LEGO instantiates each sequence into an executable test case. The challenge is instantiating valid test cases from type affinities. Specifically, we need to instantiate corresponding SQL statements just from a type. It is difficult to ensure syntactic and semantic correctness. LEGO addresses the issue by utilizing AST (abstract syntax tree) [48] as an intermediate representation between test cases and types. In instantiation, the dependencies between statements are also analyzed and maintained. The instantiation includes three steps. First, AST synthesis. When finding a new seed, LEGO parses each of its statements to extract AST structures and saves them into the global library. In instantiation, for each entry in the SQL Type Sequences, LEGO randomly selects a type-matched structure from the library to build the AST. Second, statement concatenation. LEGO translates the AST of each entry into SQL statements and concatenates them into a candidate SQL test case. Finally, validation. The candidate test case is re-translated to an AST. The dependencies between different data are analyzed, and the AST will be filled with concrete values that satisfy all dependencies. After that, the AST is translated to an executable test case and fed to the target DBMS. Because of the randomness in selecting structures, one SQL Type Sequence will be instantiated multiple times to increase the diversity for combinations of AST structures and type sequences.

For example, with the sequence "`PRAGMA`$\rightarrow$ `CREATE TABLE`$\rightarrow$ `INSERT`", LEGO first constructs the SQL statement skeleton and instantiates some SQL data with random values, like "PRAGMA foreign_keys=on; CREATE TABLE v0(x INT PRIMARY KEY, y INT REFERENCE); INSERT INTO v2(v1) VALUES(100);". This test case contains semantic errors because TABLE v2 is not exist. Then, LEGO builds the data dependency graph of the SQL statement and fixes it with the correct data. Finally, the instantiated test case is "PRAGMA foreign_keys=ON; CREATE TABLE v0(x INT PRIMARY KEY, y INT REFERENCE); INSERT INTO v0(x) VALUES(100);".

IV. IMPLEMENTATION

LEGO is implemented based on AFL++ [13]. The affinity analyzer and the sequence synthesizer are two main components. The affinity analyzer implements Algorithm 1 and 2 to analyze and record the new type-affinities. The synthesizer implements Algorithm 3 to synthesize new SQL Type Sequences. LEGO integrates the two components as a custom mutator of AFL++. Besides, we also implement the syntax-preserving mutations as conventional mutation methods.

The affinity analyzer and the sequence synthesizer are supported by the AST parser. LEGO reuses the IR (intermediate representation) defined by SQUIRREL and implements AST parser based on Bison 3.3.2 [7] and Flex 2.6.4 [14]. Since the specifications of the latest DBMSs like PostgreSQL contain lots of dialects or unique features, we use a lot of rules in LEGO's AST parser to make sure it supports these features. Specifically, for the parser of MariaDB and MySQL, We use 748 definitions of tokens for Flex, 852 declarations and 2855 rules for Bison. And for the parser of PostgreSQL and Comdb2, the numbers are 494, 695, and 3179, as well as 201, 205, and 455, respectively. Besides, we also write logic to translate these newly added rules to IRs and transform IRs back. To better adapt to DBMSs, we write their fuzzing drivers using AFL++'s persistent mode.

V. EVALUATION

We evaluated LEGO in terms of its ability to discover new vulnerabilities, as well as its efficiency in exploring the state space of the target DBMSs. Our evaluation aims at answering the following research questions:

- **RQ1:** Can LEGO discover new vulnerabilities?
- **RQ2:** Can LEGO perform better than other state-of-the-art DBMS fuzzers?
- **RQ3:** How effective are sequence-oriented algorithms?

A. Evaluation Setup

Tested DBMSs and compared fuzzers. To evaluate the generality and efficiency of LEGO, we used the latest version of four open-source DBMSs for evaluation, namely PostgreSQL, MySQL, MariaDB, and Comdb2, which are widely used in industry and academic research. PostgreSQL [26, 30] is a object-relational DBMS with over 30 years of active development. MySQL [27, 47] is one of the most popular open-source DBMSs. MariaDB [5, 23] is a community-developed fork of MySQL. Comdb2 [11, 36] clusters RDBMS built on optimistic concurrency control techniques. To encompass as many state-of-the-art DBMS fuzzers as possible, we compared LEGO to popular fuzzer SQUIRREL and SQLancer from the academy and SQLsmith from the industry.

Basic setup. We performed all experiments on a machine running 64-bit Ubuntu 20.04 with 128 cores (AMD EPYC 7742 Processor @ 2.25 GHz) and 488 GiB of main memory. All the DBMSs were instrumented with AddressSanitizer (ASAN) [38]. For each fuzzer, we used their default configurations, such as instrumentation methods and seed corpus. We

TABLE I
LEGO DISCOVERED 102 NEW VULNERABILITIES (POSTGRESQL: 6, MYSQL: 21, MARIADB: 42, COMDB2: 33) WHILE OTHERS FOUND 11 IN TOTAL. [**UAF**: USE-AFTER-FREE, **BOF**: BUFFER OVERFLOW [HEAP (H), STACK (S)], **AF**: ASSERTION FAILURE, **SEGV**: SEGMENTATION VIOLATION, **UAP**: USE-AFTER-POISON, **NPD**: NULL POINTER DEREFERENCE, **UB**: UNDEFINED BEHAVIOR]

DBMS	Component	Bug Type and Number	Identifier
PostgreSQL	Optimizer	BOF(1), AF(1), SEGV(2)	BUG #17097, BUG #110303, BUG #17152, BUG #17151
PostgreSQL	Parser	AF(1)	BUG #17094
PostgreSQL	DML	AF(1)	BUG #17067
MySQL	Optimizer	BOF(3), SBOF(1), NPD(4), HBOF(1), UAF(1), AF(2)	CVE-2021-2357, CVE-2021-2055, CVE-2021-2230, CVE-2021-2169, CVE-2021-2444
MySQL	DML	SBOF(1), SEGV(2)	CVE-2021-35645
MySQL	Auth	SBOF(1), SEGV(2)	CVE-2021-35643
MySQL	Storage	SEGV(1), AF(2)	CVE-2021-35641
MariaDB	Optimizer	NPD(2), BOF(1), UAP(3), SEGV(2), AF(1)	CVE-2022-27376, CVE-2022-27379, CVE-2022-27380, MDEV-26403, MDEV-26432, MDEV-26418, MDEV-26416, MDEV-26419, MDEV-26430
MariaDB	DML	BOF(1), UAP(1), AF(1), SEGV(1)	CVE-2022-27377, CVE-2022-27378, MDEV-26120, MDEV-25994
MariaDB	Parser	BOF(1), UAF(2), SEGV(1)	CVE-2022-27383, MDEV-26355, MDEV-26313, MDEV-26410
MariaDB	Storage	SEGV(7), UAP(2), UAF(2), BOF(2)	CVE-2022-27385, CVE-2022-27386, MDEV-26404, MDEV-26408,MDEV-26412, MDEV-26421, MDEV-26434, MDEV-26436, MDEV-26420, MDEV-26408, MDEV-26431, MDEV-26432, MDEV-26433
MariaDB	Item	AF(4), SEGV(3), UAP(2), UAF(1)	MDEV-26405, MDEV-26407, MDEV-26411, MDEV-26414, MDEV-26438, MDEV-26428, MDEV-26417, MDEV-26434, MDEV-26437, MDEV-26427
MariaDB	Lock	SEGV(2)	MDEV-26425, MDEV-26424
Comdb2	Bdb	UB(6)	CVE-2020-26746
Comdb2	Berkdb	BOF(1), UB(7)	CVE-2020-26745
Comdb2	Csc2	BOF(1)	CVE-2020-26744
Comdb2	Db	UB(4), UAF(1), SEGV(3)	CVE-2020-26743
Comdb2	Mem	BOF(1), HBOF(1), SEGV(1)	CVE-2020-26741, CVE-2020-26742
Comdb2	Sqlite	UB(5), SEGV(2)	–
Total		**102 bugs, 22 CVEs**	

tried to run as many tests as possible to make a comprehensive comparison between LEGO with other fuzzers. However, we encountered some compatibility issues. Specifically, since SQLsmith does not officially support the syntax of MySQL, MariaDB, and Comdb2 [41], we only compared LEGO against SQLsmith on PostgreSQL. We ran each DBMS with one fuzzer for 24 hours, which is a widely used time setup. Each fuzzer instance was run separately in a docker with one CPU core. To distinguish bugs, we first got them from unique crashes by comparing the call stack. To improve accuracy, we also further analyzed the bugs manually.

B. DBMS Vulnerability Detection

1) Overall Results: The four tested DBMSs are widely used by users and well tested by engineers, making it difficult to find new bugs. Nevertheless, LEGO managed to detect 102 vulnerabilities in continuous fuzzing, while others found only 11 of them in total. Specifically, SQLancer and SQLsmith did not find any bugs. SQUIRREL found 3 bugs in MySQL and 8 bugs in MariaDB, respectively. Table I shows LEGO discovered 6, 21, 42, and 33 bugs in PostgreSQL, MySQL, MariaDB, and Comdb2, respectively. Some of these vulnerabilities can be exploited in just a few steps and have serious repercussions. Specifically, among 102 vulnerabilities, there are 61 vulnerabilities (17 buffer overflows, 7 use after frees, 29 segmentation violations, and 8 use-after-poisons) that are very dangerous. They could be powerful attack primitives which lead to arbitrary code execution. Specifically, they can be exploited to attack the DBMS server through the network to control the system or elevate privileges. In the other words, these bugs could cause a high availability impact on the DBMS server by an attacker. There are also 6 null pointer dereferences, 13 assertion failures, and 22 undefined behaviors, which indicate the internal errors of DBMSs and might lead to denial-of-service by crashing the DBMS or other unexpected damages.

We have actively reported all the bugs to the corresponding DBMS vendors and received their confirmation feedback. At the time of the paper writing, 22 bugs have been confirmed as CVEs in the U.S. National Vulnerability Database. Among them, according to CVSS score [12], 8 CVEs are flagged as high-risk, 8 CVEs are flagged as medium-risk, and 6 CVEs are reserved at the request of the vendor (unpublished and have no score) due to their high severity and complexity. The results demonstrate that sequence-oriented fuzzing could help LEGO to explore the unexpected states, which may lead to serious vulnerabilities in target DBMSs. We analyzed the bugs and found that many of them were related to the unexpected SQL Type Sequence. Following is a case study to show the effectiveness of LEGO for detecting vulnerabilities in PostgreSQL which has existed for about 2 years prior to the writing of the paper.

2) Case Study: In this section, we introduce and analyze a SEGV in PostgreSQL. The bug lies in PostgreSQL's optimizer component. It happens when the optimizer makes the plan for the query of a clause. Specifically, the bug is triggered by the unexpected sequence which composes the `NOTIFY` statement and the `WITH` clause.

The mechanism to trigger the bug. Figure 7 shows a test case that can trigger the bug. First, it executes the `CREATE TABLE` statement to create a table $v0$ in PostgreSQL. Then it creates an instead rule on the `INSERT` operation for table

```
1: CREATE TABLE v0( v4 INT, v3 INT UNIQUE, v2 INT , v1 INT UNIQUE ) ;
2: CREATE OR REPLACE RULE v1 AS ON INSERT TO v0 DO INSTEAD NOTIFY COMPRESSION;
3: COPY ( SELECT 32 EXCEPT SELECT v3 + 16 FROM v0 ) TO STDOUT CSV HEADER ;
4: WITH v2 AS (INSERT INTO v0 VALUES (0)) DELETE FROM v0 WHERE v3 = - - - 48;
```

Fig. 7. A test case that can trigger a SEGV in PostgreSQL, which composes `NOTIFY` and `WITH` into a sequence.

$v0$, which executes an `NOTIFY` instead of inserting values. After executing the `COPY` operation to transfer data, it uses an `WITH` clause to create a temporary view, which updates the data of $v0$ (i.e., inserts one value and deletes some other records). However, the SQL Type Sequence "`CREATE RULE`→ `NOTIFY`→ `COPY`→ `WITH`" constructs a logic that is not considered in PostgreSQL.

Crash Code in Optmizer Component :

```
void replace_empty_jointree(Query *parse){
...
// parse->jointree is null causing the crash
if (parse->jointree->fromlist != NIL)return;  ...
parse->jointree->fromlist = list_make1(rtr); }
```

Root Cause Code in Rewrite Component :

```
static List * RewriteQuery(Query *parsetree, List *rewrite_events)
{ ...
// Recursively process any insert/update/delete statements in WITH
foreach(lc1, parsetree->cteList){
CommonTableExpr *cte = lfirst_node(CommonTableExpr, lc1);
Query *ctequery = castNode(Query, cte->ctequery);
List *newstuff;
newstuff = RewriteQuery(ctequery, rewrite_events);
// Handle single-statement DO INSTEAD rules
if (list_length(newstuff) == 1){
// Push the single Query back into the CTE node
ctequery = linitial_node(Query, newstuff);
// Add code to handle the case of NOTIFY statement to fix the bug
...
Assert(!ctequery->canSetTag);
cte->ctequery = (Node *) ctequery;
}
```

Fig. 8. The crash code of the SEGV and the fixing methods. The root cause of the bug is that the rewrite component ignores the situation when a `NOTIFY` statement is followed by a `WITH` statement in a test case.

Figure 8 shows the relevant crash code to this bug, as well as the fixing methods. PostgreSQL crashes in its optimizer component's `replace_empty_jointree` function when the backend process makes plans for the `WITH` statement. The root cause of this bug is in PostgreSQL's rewrite component, which ignores the situation when a `NOTIFY` statement is followed by a `WITH` statement. Specifically, since the `NOTIFY` statement is associated with the `INSERT` operation for table $v0$, PostgreSQL will invoke the rewrite rule to replace the insert operation "`INSERT INTO v0 VALUES (0)`" in the `WITH` statement with a `NOTIFY` statement. In the code, PostgreSQL calls the `RewriteQuery` function to process insert statements in WITH clauses. However, `NOTIFY` commands are not supported to replace the `INSERT` statement in a `WITH` clause. In other words, it misses the case where a DML [40] statement is rewritten by a `NOTIFY` statement. As a result, PostgreSQL gets an NULL `jointree` which causes the SEGV in `replace_empty_jointree` in planning later.

The developers of PostgreSQL responded that they currently lack the support to rewrite `INSERT/ UPDATE/ DELETE` statement with `NOTIFY` in a `WITH` clause. And they missed the checks of the case. They fixed this issue and added new test cases which have the SQL Type Sequence "`CREATE RULE`→ `NOTIFY`→ `COPY`→ `WITH`" to do regression test.

The reason for detecting the bug only by LEGO. The bug is invoked by an unexpected SQL Type Sequence, which is rarely used by testers, and as a result, the bug hides for a long time. With the analyzed type-affinities, LEGO synthesizes abundant SQL Type Sequences containing the expected sequence. Based on the type sequence of corresponding synthesized seeds, LEGO mutates them into more seeds effectively and finally synthesizes the specific test case to trigger this bug. Other fuzzer are hard to compose the specific SQL Type Sequence thus they cannot find the bug. Specifically, SQLsmith mainly generates `SELECT` SQL statements, which would ignore the bugs composed of different types of SQL statements. SQLancer generates test cases based on custom pattern rules mainly for `SELECT` statements, while only a limited number of SQL Type Sequences can be generated. SQUIRREL generates test cases mainly by changing the structure or data in one individual statement, so it is hard to generate new sequences of SQL types beyond the sequence contained in the existing seeds. Consequently, the bugs which have new SQL Type Sequences will be missed by them.

C. Comparison with Other DBMS Fuzzers

We evaluated fuzzers using two metrics, namely branches covered and bugs triggered. The two metrics are used as the standard in fuzzing evaluation [8, 17, 44], and have been widely used in fuzzing works [35, 42, 54]. To evaluate LEGO, we compared it against SQLancer, SQLsmith, and SQUIRREL. For a fair comparison, when we finished fuzzing, we collected the seeds generated by each fuzzer and rerun the input seeds to uniform the branch coverage. In addition, the bugs were distinguished and identified by comparing the call stack and manual analysis.

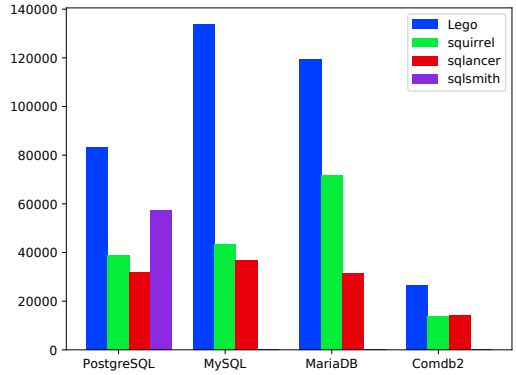

Fig. 9. Number of branches covered by LEGO, SQUIRREL, SQLancer, and SQLsmith on 4 DBMSs in 24 hours.

Coverage. Figure 9 demonstrates the branches covered by those fuzzers over 24-hour fuzzing. It shows that LEGO performed better. Specifically, LEGO covered 198%, 44%, and 120% more branches than SQLancer, SQLsmith, and SQUIRREL on average, respectively. The fundamental reason for improving coverage is that DBMSs are rich in states, and the states are sensitive to the type and execution order of SQL statements. The same set of SQL statements can

trigger different code regions when executed with different orders. Since LEGO focuses on generating abundant SQL Type Sequences, thus it is sensitive to the order of execution of statements and can cover more code regions as well as unlock bugs hidden in them. SQLancer and SQLsmith are two state-of-the-art DBMS fuzzers that generate test cases from rules. However, they could only generate limited SQL Type Sequences. Specifically, SQLancer continuously generates test cases for fuzzing based on custom pattern rules, while only a limited number of SQL Type Sequences can be generated. Furthermore, SQLsmith mainly generates `SELECT` statements for PostgreSQL to ensure semantic correctness.

SQUIRREL is an advanced mutation-based DBMS fuzzer. It uses coverage feedback to guide it in exploring the code regions of the target DBMS. Due to the coverage feedback and its efforts to improve syntactic and semantic correctness, SQUIRREL performs better than SQLancer in these DBMSs. However, it still suffers from the lack of the abundance of SQL Type Sequences. Starting from the initial seeds, SQUIRREL mainly modifies the structure and data inside single SQL statements. Therefore the seeds it produces is hard to change the SQL Type Sequences of the initial seeds. Because of the limitation in sequences' abundance, SQUIRREL may be not able to trigger some functionalities of target DBMSs.

TABLE II
NUMBER OF TYPE-AFFINITIES GENERATED BY DIFFERENT FUZZERS

DBMS	SQLancer	SQUIRREL	LEGO
PostgreSQL	474	34	2101
MySQL	50	21	643
MariaDB	119	28	734
Comdb2	127	36	229
Total	770	119	3707
Increment	2937	3588	–

In contrast, LEGO is designed to increase the abundance of SQL Type Sequences. Type affinity describes the pattern of composing sequences, which reflects the abundance of SQL Type Sequences. Table II shows the type-affinities contained by seeds generated by different fuzzers in 24 hours. SQLsmith is excluded because it contains only one statement per test case. The table shows that LEGO found more type-affinities than other fuzzers, which allowed LEGO to produce more meaningful sequences to increase the abundance. Specifically, it proactively explores the type sequence space through sequence-oriented mutations and analyzes type-affinities in mutated seeds that have new coverage. It then uses these type-affinities to synthesize meaningful sequences. The increase in sequence abundance assists LEGO to trigger more logic in the target DBMSs. Consequently, LEGO had better coverage than others in the target DBMS.

Bugs. Table III shows the number of bugs found by each fuzzer. It shows that LEGO found 52, 52, and 41 more bugs than SQLancer, SQLsmith, and SQUIRREL, respectively. SQLancer focuses on detecting logic bugs in DBMSs, but the bug-finding process is limited by its predefined rules.

TABLE III
NUMBER OF BUGS TRIGGERED IN 24 HOURS

DBMS	SQLancer	SQLsmith	SQUIRREL	LEGO
PostgreSQL	0	0	0	2
MySQL	0	–	3	11
MariaDB	0	–	8	32
Comdb2	0	–	0	7
Total	0	0	11	52
Increment	52	52	41	–

Consequently, it did not trigger bugs in the latest versions of these DBMSs. SQLsmith generates only limited types of statements, especially the `SELECT` type. It greatly ensures syntax correctness, however, the abundance of SQL Type Sequences is also limited. The evaluation results show it did not find any bugs in the latest version of PostgreSQL. Based on the improved coverage, LEGO explores more state space of the target DBMSs. Exploring more states increases the likelihood of finding bugs. Besides, as the case study shows, many of the triggered bugs have unexpected SQL Type Sequences. LEGO proactively analyzes type-affinities of statements from meaningful test cases. Based on type-affinities, LEGO progressively synthesizes abundant SQL Type Sequences. Moreover, LEGO lays the foundation for conventional mutations to use these sequences to mutate and find bugs. Thus, LEGO found more bugs than SQLancer, SQLsmith, and SQUIRREL.

D. Effectiveness of Sequence-Oriented Algorithms in LEGO

To measure the effectiveness of the sequence-oriented fuzzing algorithm and exclude other differences such as the extension in AST parser, we implement LEGO- for comparison, which disables the sequence-oriented algorithms including proactively affinity analysis and progressive sequence synthesis. Note that the affinity analysis provides the fundamental elements for sequence synthesis, the tightly-coupled nature requires us to disable them altogether. We compare LEGO- against LEGO on PostgreSQL, MySQL, MariaDB, and Comdb2 for 24 hours.

Table IV shows the eventual number of type-affinities found and branches covered by LEGO and LEGO- on four DBMSs. First, LEGO *is able to find more type-affinities than* LEGO-. Specifically, Table IV shows that LEGO found 337, 48, 119, and 29 more type-affinities when compared to LEGO- on PostgreSQL, MySQL, MariaDB, and Comdb2, respectively. When disabling sequence-oriented algorithms, the conventional mutation methods in LEGO- are limited to changing individual statements of a test case. Differently, LEGO proactively explores type-affinities to increase the abundance of SQL Type Sequences. Consequently, LEGO found more type-affinities than LEGO-. Second, *with more type-affinities,* LEGO *can cover more branches in four DBMSs*. Specifically, LEGO covered 20%, 15%, 25%, and 7% more branches when compared to LEGO- on PostgreSQL, MySQL, MariaDB, and Comdb2, respectively. More type-affinities help LEGO synthesize more meaningful SQL Type Sequences. With the increase

TABLE IV
NUMBER OF TYPE-AFFINITIES FOUND AND BRANCHES COVERED BY LEGO- AND LEGO

DBMS		Type-Affinities			Branches		
Name	Types	LEGO-	LEGO	Increment	LEGO-	LEGO	Improvement
PostgreSQL	188	1764	2101	337↑	69301	83149	20%↑
MySQL	158	595	643	48↑	116713	133840	15%↑
MariaDB	160	615	734	119↑	95570	119238	25%↑
Comdb2	24	200	229	29↑	24625	26269	7%↑

in sequence abundance, more functions in target DBMSs could be triggered. Thus LEGO covered more branches than LEGO-.

Moreover, the table also illustrates the correlation among the number of statement types, increments in type-affinity finding, and improvements in branch coverage: *when a DBMS has more statement types,* LEGO *tends to make more type-affinity increments while making more branch coverage improvements*. Specifically, the first two columns show the statement types of PostgreSQL, MySQL, MariaDB, and Comdb2. The 5th column shows increments in type-affinity finding made by LEGO, and the 8th column shows the improvement in branch coverage. From the 2nd column and the 5th column, we can find that when one DBMS has more statement types, LEGO could have more increments in type-affinity finding. The increment in type-affinities results in the increment of SQL Type Sequences. Correspondingly, LEGO could also make more improvements in the number of branches covered. For example, LEGO increased the number of type-affinities on PostgreSQL by 337 when compared to LEGO-, and it improved the branch coverage by 20%. LEGO's improvements to Comdb2 were smaller than those to other DBMSs. It might be due to the fewer types of Comdb2 than other DBMSs. Specifically, there are only 24 types in Comdb2, which restricts the upper limit of the increment in type-affinities. In the experiment, LEGO had a 29 increment when compared to LEGO-. As a result, LEGO correspondingly made a 7% improvement in the number of branches covered for Comdb2.

VI. DISCUSSION

Redundant type-affinities. In proactively type-affinity analysis, changes besides SQL types may trigger new coverage and cause redundant type-affinities. For example, new code regions might be found by changes of data or new combinations between non-adjacent SQL statements. This issue has a limited impact on LEGO. First, LEGO progressively synthesizes sequences containing new type-affinities based on the existing sequences. Thus redundant type-affinities would not cost too many resources. Second, if redundant type-affinities are caused by the combination between non-adjacent SQL statements, the affinity could help LEGO to synthesize sequences containing that. Specifically, because relations are transitive, considering adjacent statements could also cover non-adjacent cases. For example, if A→B and B→C is learned, LEGO also knows the implication of A→B→C. In other words, relations of non-adjacent relations (A→B→C) can be learned from adjacent statements (A→B and B→C). Nevertheless, we plan to refine type-affinities in the future, such as importing the model of non-adjacent combinations between types.

Semantic abundance of the synthesized test cases. LEGO analyzes the type-affinities from mutated seeds which cover new branches. In this process, extracting only the type information loses some semantic information of the test case. LEGO addresses the problem in two steps. First, besides types, it parses the AST structures and stores them in a library. Second, during the synthesis process, it randomly combines the SQL Type Sequences and type-matched structures to reconstruct the semantic information. Nevertheless, some semantic information will still be lost. In the future, we plan to learn semantic information (e.g., dependencies between statements) automatically, and use it to guide sequence synthesis.

Limiting sequence length may miss some bugs. There are indeed some bugs that can only be triggered by long repeated sequences. However, handling them may degrade fuzzing performance or even stall the fuzzer. For example, we found SQUIRREL hung for 23 minutes for a test case, whose length is 945 and has hundreds of repeated `INSERT` statements. Therefore, limiting the length is a practical way to ensure the fuzzers work normally. We conducted an additional experiment on MariaDB for fuzzing 24 hours with different lengths. The results show that LEGO finds 30, 35, and 27 bugs when setting the length to 3, 5, and 8, respectively. It illustrates that cutting length will miss some bugs, while increasing length will also miss bugs due to performance degradation. In the future, to detect bugs triggered by long sequences, we plan to split long sequences into several equivalent short sequences.

Adaptability of LEGO. The approach of LEGO is general for most DBMSs. To adapt to a new DBMS, LEGO should learn the SQL type information specific to the target DBMS. This can be achieved by providing the original grammar specification (e.g. BNF or bison/yacc file) to LEGO. LEGO will automatically derive SQL type information from the grammar specification and reuse the existing type-affinity infrastructure. Nevertheless, the number of statement types would influence the performance of LEGO. For example, LEGO has fewer improvements to LEGO- on Comdb2 than other DBMSs. To mitigate the problem, besides types, we will also further increase the diversity of synthesized sequences to improve the process. For example, we could combine different inner structures across statements.

Feasibility of extending existing fuzzers with LEGO. Directly adapting LEGO's solution to existing fuzzers by simple incremental changes is difficult. One possible solution

is to use the type-affinities found by LEGO. For generation-based fuzzers, we can add rules transformed from LEGO's type-affinity. And for mutation-based fuzzers, we can add mutation operators under the guidance of LEGO's type-affinity. However, without LEGO's results, existing work needs to re-implement LEGO's logic to increase abundance effectively.

VII. RELATED WORK

In this section, we will focus on some tasks related to DBMS fuzzing and highlight how they differ from LEGO.

Finding logic and performance bugs in DBMSs. Schemes aimed at logical bugs focus on the correctness of DBMSs. Logic bugs would not crash the system but may cause a DBMS to return unexpected results, such as leaking extra rows. RAGS [39] uses differential testing, namely detecting logic bugs by running the same query on different DBMSs and checking the result consistency. SQLancer [35] synthesizes queries to fetch a random row from existing tables in the target DBMS. If the DBMS fails to fetch that, then the DBMS might have a bug. Its following works [34, 33] also apply similar strategies by building functionally equivalent queries. Schemes aimed at performance bugs focus on the actual execution of DBMSs. Performance bugs can slow down an entire DBMS system or even bring it to a halt. APOLLO [16] generates queries to test two versions of the same DBMS. If the execution times for two versions are significantly different, then a performance bug is found. These fuzzers focus on finding differences between versions to locate bugs, rather than increasing the abundance of SQL Type Sequences.

LEGO differs from these works by aiming to find memory bugs by generating test cases containing abundant SQL Type Sequences. Compared to logic and performance bugs, memory bugs happen more frequently. We did a cursory survey of the number of different type of bugs reported on MariaDB since 2009, and we found that the number of memory bugs is larger than the other two types combined. More importantly, memory bugs are even more damaging. Because they allow an attacker to leak or corrupt memory, the attacker can execute remote code or even gain control of the whole system.

Generation-based DBMS fuzzing. Generation-based fuzzers [25, 35, 37, 43] have been used to test DBMSs for decades. They always generate enormous test cases based on custom rules, but the rules in turn also limit the SQL Type Sequences they can generate. Because generating a fully valid test case proves to be an NP-complete problem [21], generation-based fuzzers generally enhance semantic correctness while ensuring syntactic correctness. Some works treat generation as the process to satisfy constraints [1, 22] and use SAT solvers to generate potential queries [2]. SQLsmith [37] is one of the state-of-the-art generation-based DBMS fuzzers. It continuously generates syntactically correct SQL statements. But it only generates limited types of SQL statements (e.g., most are `SELECT` for PostgreSQL) to ensure the database is unchanged, which causes a restricted number of sequences.

LEGO is different from these works. Unlike fixed statement generation, It utilizes mutation-based methods to enrich sequences beyond predefined rules. Therefore, LEGO runs automatically and can progressively synthesizes SQL Type Sequences. Based on the coverage feedback to analyze type-affinities, LEGO could increase the abundance of SQL Type Sequences and trigger various behaviors of the target DBMSs.

Mutation-based DBMS fuzzing. Mutation-based fuzzers have been widely used to test software and find many bugs [3, 9, 18, 19, 20, 45, 46, 51, 52, 53, 54]. They generally leverage coverage feedback to test the target programs. Traditional mutation-based fuzzers (e.g., AFL [53]) could easily adapt to testing DBMS libraries like SQLite [6]. However, the mutated seeds generated by random mutations can hardly pass the syntax checks of DBMSs. Some works adopt advanced program analysis techniques like taint analysis [3, 20] and symbolic execution [19, 52]. However, it is still hard for them to generate test cases that are both correct in syntax and semantics. Recently, SQUIRREL [54] was proposed to generate valid test cases by syntax-preserving and semantics-guided mutation. It designs an intermediate representation to maintain the structure information of test cases. RATEL [46] further adapts mutation-based fuzzing into several enterprise-level DBMSs. It improves the feedback precision, enhances the robustness of input generation, and performs an online investigation on the root cause of bugs. UNICORN [51] combines syntax-preserved mutation and time-series guided mutation to generate time-series queries. GRIFFIN [15] uses metadata graph to mutate test cases in a grammar-free way.

LEGO differs from these works by enhancing mutation with the sequence synthesis. Unlike random mutations, LEGO proactively analyzes the type-affinities from test cases. The type-affinities are used to synthesize meaningful SQL Type Sequences, which implicitly contain the semantic information of test cases. Therefore, LEGO can continually cover new branches and find previously-unknown bugs.

VIII. CONCLUSION

This paper presents LEGO, a fuzzer that automatically analyzes type-affinities to increase the abundance of SQL Type Sequences. It first analyzes the type-affinities from any two adjacent statement types that appear in test cases generated by proactively sequence-oriented mutation. Then it synthesizes new sequences based on the analyzed affinities. LEGO outperforms three state-of-the-art fuzzers on four popular DBMSs, namely PostgreSQL, MySQL, MariaDB, and Comdb2. More importantly, LEGO finds 102 new vulnerabilities. Among them, 22 bugs are confirmed as CVEs due to their severe security influences. Our future work will focus on refining the type-affinity analysis to improve efficiency.

IX. ACKNOWLEDGEMENTS

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000, No2021QY0604) and NSFC Program (No. 62022046, 92167101, U1911401, 62021002, 62192730).

References

[1] Shadi Abdul Khalek and Sarfraz Khurshid. 2010. Automated SQL query generation for systematic testing of database engines. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 329–332.

[2] alloy 2022. Documentation Alloy 6. https://alloytools.org/documentation.html. Accessed: November 29, 2022.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*.

[4] Adam Bannister. 2021. SQLite patches use-after-free bug that left apps open to code execution, denial-of-service exploits. https://portswigger.net/daily-swig/sqlite-patches-use-after-free-bug-that-left-apps-open-to-code-execution-denial-of-service-exploits. Accessed: November 29, 2022.

[5] Daniel Bartholomew. 2014. *MariaDB cookbook*. Packt Publishing Ltd.

[6] ST Bhosale, Miss Tejaswini Patil, and Miss Pooja Patil. 2015. Sqlite: Light database system. *Int. J. Comput. Sci. Mob. Comput* 44, 4 (2015), 882–885.

[7] Bison 2022. Bison. https://www.gnu.org/software/bison/. Accessed: November 29, 2022.

[8] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *44th IEEE/ACM International Conference on Software Engineering, ser. ICSE*, Vol. 22.

[9] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *USENIX Security Symposium*.

[10] Catalin Cimpanu. 2019. Google Chrome impacted by new Magellan 2.0 vulnerabilities. https://www.zdnet.com/article/google-chrome-impacted-by-new-magellan-2-0-vulnerabilities. Accessed: November 29, 2022.

[11] comdb2 2022. Comdb2 GitHub. https://github.com/bloomberg/comdb2. Accessed: November 29, 2022.

[12] CVSS 2022. Common Vulnerability Scoring System version 3.1: User Guide. https://www.first.org/cvss/user-guide. Accessed: November 29, 2022.

[13] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*.

[14] Flex 2022. Flex, the fast lexical analyzer generator. https://github.com/westes/flex. Accessed: November 29, 2022.

[15] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-Free DBMS Fuzzing. In *Conference on Automated Software Engineering (ASE'22)*.

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

[17] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*.

[18] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 2018. PAFL: Extend Fuzzing Optimizations of Single Mode to Industrial Parallel Mode.

[19] Jie Liang, Yu Jiang, Mingzhe Wang, Xun Jiao, Yuanliang Chen, Houbing Song, and Kim-Kwang Raymond Choo. 2019. Deepfuzzer: Accelerated deep greybox fuzzing. *IEEE Transactions on Dependable and Secure Computing* (2019).

[20] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. PATA: Fuzzing with Path Aware Taint Analysis. In *2022 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, Los Alamitos, CA, USA*. 154–170.

[21] Eric Lo, Carsten Binnig, Donald Kossmann, M Tamer Özsu, and Wing-Kai Hon. 2010. A framework for testing DBMS features. *The VLDB Journal* 19, 2 (2010), 203–230.

[22] Michaël Marcozzi, Wim Vanhoof, and Jean-Luc Hainaut. 2012. Test input generation for database programs using relational constraints. In *Proceedings of the Fifth International Workshop on Testing Database Systems*. 1–6.

[23] MariaDB 2022. MariaDB. https://mariadb.org/. Accessed: November 29, 2022.

[24] MariaDB 2022. SQL Statements and Structure. https://mariadb.com/kb/en/sql-statements-structure. Accessed: November 29, 2022.

[25] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. 2008. Generating targeted queries for database testing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 499–510.

[26] Bruce Momjian. 2001. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York.

[27] MySQL. 2022. MySQL. https://www.mysql.com/. Accessed: November 29, 2022.

[28] Oracle 2022. Oracle Official Website. https://www.oracle.com/cn/index.html. Accessed: November 29, 2022.

[29] Yevgeny Pats. [n. d.]. Why (Continuous) Fuzzing. https://about.gitlab.com/blog/2020/12/10/why-continuous-fuzzing/. Accessed: November 29, 2022.

[30] PostgreSQL 2022. PostgreSQL. https://www.postgresql.org/. Accessed: November 29,

2022.

[31] PostgreSQL SQL Commands 2022. SQL Commands. https://www.postgresql.org/docs/13/sql-commands.html. Accessed: November 29, 2022.

[32] Manuel Rigger. 2022. Bugs found in Database Management Systems. https://www.manuelrigger.at/dbms-bugs. Accessed: November 29, 2022.

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

[34] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 211:1–211:30. https://doi.org/10.1145/3428279

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

[36] Alex Scotti, Mark Hannum, Michael Ponomarenko, Dorin Hogea, Akshat Sikarwar, Mohit Khullar, Adi Zaimi, James Leddy, Fabio Angius, Rivers Zhang, and Lingzhi Deng. 2016. Comdb2: Bloomberg's Highly Available Relational Database System. *Proc. VLDB Endow.* 9, 13 (2016), 1377–1388. https://doi.org/10.14778/3007263.3007275

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

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

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

[40] SQL 2021. SQL — DDL, DQL, DML, DCL and TCL Commands. Retrieved April 1, 2023 from https://www.geeksforgeeks.org/sql-ddl-dql-dml-dcl-tcl-commands/

[41] SQLsmith 2022. SQLsmith Description. https://github.com/anse1/sqlsmith#description. Accessed: November 29, 2022.

[42] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 344–358.

[43] Jiajie Wang, Puhan Zhang, Lei Zhang, Haowen Zhu, and Xiaojun Ye. 2013. A model-based fuzzing approach for DBMS. In *2013 8th International Conference on Communications and Networking in China (CHINACOM)*. IEEE, 426–431.

[44] Mingzhe Wang, Jie Liang, Chijin Zhou, Yuanliang Chen, Zhiyong Wu, and Yu Jiang. 2021. Industrial Oriented Evaluation of Fuzzing Techniques. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 306–317.

[45] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. 2021. {RIFF}: Reduced Instruction Footprint for {Coverage-Guided} Fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 147–159.

[46] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 328–337.

[47] Michael Widenius, David Axmark, and Kaj Arno. 2002. *MySQL reference manual: documentation from the source*. " O'Reilly Media, Inc.".

[48] wikipedia. 2022. Abstract syntax tree. https://en.wikipedia.org/wiki/Abstract_syntax_tree. Accessed: November 29, 2022.

[49] wikipedia 2022. *Databases*. Retrieved April 1, 2023 from https://en.wikipedia.org/wiki/Database

[50] wikipedia. 2022. SQL. https://en.wikipedia.org/wiki/SQL. Accessed: November 29, 2022.

[51] Zhiyong Wu, Jie Liang, Mingzhe Wang, Chijin Zhou, and Yu Jiang. 2022. Unicorn: Detect Runtime Errors in Time-Series Databases With Hybrid Input Synthesis. In *Symposium on Software Testing and Analysis (ISSTA'22)*.

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

[53] Michał Zalewski. 2022. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: November 29, 2022.

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