Griffin: Grammar-Free DBMS Fuzzing

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

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

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

Mingzhe Wang
ShuiMuYuLin Ltd
Tsinghua University, China

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

ABSTRACT

Fuzzing is a promising approach to DBMS testing. One crucial component in DBMS fuzzing is grammar: since DBMSs enforce strict validation on inputs, a grammar improves fuzzing efficiency by generating syntactically- and semantically-correct SQL statements. However, due to the vast differences in the complex grammar of various DBMSs, it is painstaking to adapt these fuzzers to them. Considering that lots of DBMSs are not yet well tested, there is an urgent need for an effective DBMS fuzzing approach that is free from grammar dependencies.

In this paper, we propose Griffin, a grammar-free mutation based DBMS fuzzer. Rather than relying on grammar, Griffin summarizes the DBMS's state into *metadata graph*, a lightweight data structure which improves mutation correctness in fuzzing. Specifically, it first tracks the metadata of the statements in built-in SQL test cases as they are executed, and constructs the metadata graph to describe the dependencies between metadata and statements iteratively. Based on the graphs, it reshuffles statements and employs metadata-guided substitution to correct semantic errors. We evaluate Griffin on MariaDB, SQLite, PostgreSQL, and DuckDB. Griffin covers 73.43%-274.70%, 80.47%-312.89%, 43.80%-199.11% more branches, and finds 27, 27, and 22 more bugs in 12 hours than SQLancer, SQLsmith, and Squirrel, respectively. In total, Griffin finds 55 previously unknown bugs with 13 CVEs assigned.

KEYWORDS

DBMS Fuzzing, Grammar-Free, Vulnerability Detection

ACM Reference Format:
Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-Free DBMS Fuzzing. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3560431

*Jie Liang and Yu Jiang are the corresponding authors.

1 INTRODUCTION

Database management systems (DBMSs) are widely used for data storage in software [43]. To meet the different software needs, hundreds of DBMSs with different features and functionalities have been developed [40]. However, because DBMSs are big systems, they usually contain vulnerabilities, which may lead to denial-of-service (DoS), remote code execution, data breach, etc. These vulnerabilities are harmful to the security of DBMS and stored data.

Fuzzing is an automated technique to discover vulnerabilities [30]. It has been applied to test DBMSs in recent years and found a large number of vulnerabilities. Generally, fuzzers generate a number of SQL statements to feed into DBMSs. And abnormal behaviors (e.g., crashes) will be recorded to identify bugs. Unlike the practice of fuzzing C libraries like `libpng`, generating high-quality inputs for DBMSs can be challenging due to complicated checks on syntax and semantics. Specifically, a DBMS will first check the syntactic correctness of a SQL statement. Incorrect statements will be rejected outright. Next, the DBMS will check the semantics of the statement for correctness (e.g., do not use elements that do not exist). Again, statements with errors will be dropped. Therefore, to adequately test the DBMSs, generating syntactically and semantically correct SQL statements is a prerequisite.

To generate syntactically- and semantically-correct statements, a general idea is to *model the grammar of SQL* as AST (abstract syntax tree) in input generation.l Many popular works are generation-based [32, 33]. For example, SQLancer [32] extracts the basic information about the database such as database objects and files and then utilize AST to generate valid `INSERT` or `SELECT` statements. Other works apply mutation-based fuzzing on DBMSs to generate various types of statements and import coverage feedback [39, 51]. For example, Squirrel [51] deeply customizes different syntax parsers according to the grammar of the target DBMS. With the parser, it mitigates errors in syntax and semantics of SQL statements mutated from existing ones. Ratel [39] further extends it by building dictionaries for unsupported grammar.

However, existing efforts heavily rely on the grammar of the target DBMSs to generate valid SQL statements to ensure efficiency. Considering a new DBMS or a different version that differs in grammar, it is labor-intensive to adapt conventional fuzzers to them. For example, to support most of MariaDB's grammar, Squirrel would need to add 9,050 lines of code [14]. Due to the high requirement on manual adaptations, the state-of-the-art DBMS fuzzers nowadays cover only a few of the most popular ones, such as PostgreSQL, MariaDB, and SQLite. However, there are 391 different DBMSs on the market currently [19], and most of them use different grammars. Although is an urgent need for other DBMSs to improve their security as well, adapting these advanced fuzzers to them would take too long and require too many human resources to be economically feasible. Consequently, there is a great need for a DBMS fuzzing approach that is free of grammar dependencies to be general to different DBMSs.

The main challenge of grammar-free DBMS fuzzing is to appropriately maintain syntactic and semantic correctness of newly generated queries. For example, when mutating the statement "SELECT Name, Department FROM Student", the random mutation will easily cause syntax errors because it might destruct SQL keywords. More importantly, to ensure semantic correctness, the variable "Name" should be mutated and replaced with another type-matched variable that follows the constraint (i.e., also a column name and exists in the table). If another name is used arbitrarily, such as "Price", it may not make any sense, or it may not be a column name, or even if it is a column name, it may not exist in the table "Student". However, without grammar, it is very difficult to identify variables like column names and obtain dependencies of SQL statements to ensure semantic correctness.

To address the challenge, we propose Griffin, a DBMS fuzzer that generates valid SQL statements in a grammar-free way. Rather than relying on grammar, Griffin summarizes the DBMS's state into a *metadata graph*, a lightweight data structure which helps improving mutation correctness in fuzzing. Specifically, it first tracks the metadata (e.g. related database objects and files) of the statements in built-in SQL test cases as they are executed. Based on the metadata, Griffin constructs the metadata graph to describe the dependencies between metadata and statements iteratively. With the metadata graph, Griffin reshuffles statements from existing test cases and then reconstructs the metadata graph by substituting metadata nodes to correct semantic errors. Compared to other conventional DBMS fuzzers which require lots of labor costs to generate a large amount of SQL grammar adaptation code, Griffin are more conveniently adapted to various DBMSs because it is free from the dependencies of grammar.

To demonstrate the robustness of Griffin, we directly apply Griffin to three well-tested DBMSs MariaDB, SQLite, and PostgreSQL, as well as one newly developed DBMS DuckDB *without any specific adaptations*. Griffin covers 73.43%-274.70%, 80.47%-312.89%, 43.80%-199.11% more branches, and finds 27, 27, and 22 more bugs in 12 hours than SQLancer, SQLsmith, and Squirrel, respectively. More importantly, Griffin finds 24, 16, 3, and 12 unknown bugs in MariaDB, SQLite, PostgreSQL, and DuckDB, respectively. Among them, 13 bugs are confirmed as CVEs in the National Vulnerability Database. The results show that without grammar, Griffin is still effective in generating valid SQL statements and finding vulnerabilities.

In summary, we have the following contributions:

(1) We found that existing DBMS fuzzers rely on grammar. The intensive-labor limits their application.
(2) We propose a grammar-free DBMS fuzzing that generates valid SQL statements by analyzing and constructing metadata graphs to guide the mutation.
(3) We implement the approach in Griffin. In evaluation, Griffin found 55 previous-unknown bugs. Among them, 13 bugs are confirmed as CVEs.

2 BACKGROUND AND MOTIVATION

This section contains basic knowledge about DBMSs such as SQL grammar and metadata, along with the challenge of grammar-free DBMS fuzzing and our basic idea.

DBMS. A database management system (DBMS) is software that stores and accesses data based on a certain database model [41]. For example, one of the most popular database models is the relational model, and DBMSs based on this model are often referred to as relational database management systems (RDBMS) [45]. There are about 391 popular DBMS products [19], and they differ significantly in terms of supported features, such as operating system support, data types, triggers, storage engines, etc.

Table 1: the `CREATE TRIGGER` statement in different DBMSs. MariaDB uses the if-else statement for the conditional statements, and SQLite uses the when clause and update statement to do the same, while PostgreSQL must declare an additional function to create a trigger. Although `CREATE TRIGGER` is a common feature in most relational DBMSs, there are still lots of differences between the grammar of different DBMSs.

DBMS	CREATE TRIGGER statement
DuckDB	Not supported

MariaDB	`CREATE TRIGGER t AFTER UPDATE ON account`
	`FOR EACH ROW`
	`WHEN NEW.amount < 0 BEGIN`
	` UPDATE account SET amount = 0`
	` WHERE rowid = OLD.rowid;`
	`END`

MariaDB	`CREATE TRIGGER t BEFORE UPDATE ON account`
	`FOR EACH ROW`
	`BEGIN`
	` IF NEW.amount < 0 THEN`
	` SET NEW.amount = 0;`
	` END IF;`
	`END`

PostgreSQL	`CREATE FUNCTION trigger_f()`
	`RETURN TRIGGER LANGUAGE PLPGSQL`
	`AS $$`
	`...`
	`$$`
	`CREATE TRIGGER t BEFORE UPDATE ON account`
	`FOR EACH ROW`
	`EXECUTE PROCEDURE trigger_f();`

SQL grammar. Structured query language (SQL) is a domain-specific language used to manage data held in a DBMS [46]. *SQL grammar* describes the syntax and semantic features of the language. *SQL statements* are the carriers for the SQL grammar, and they are the smallest execution unit fed into the DBMSs. There are always differences in grammar from one DBMS to another one. For example, even though there are 156 DBMSs that support the relational model [20], they still have grammatical differences in the same functionality. As Table 1 shows, SQLite, MariaDB, and PostgreSQL all support the "create trigger" statement. However, the grammar is very different. To create a trigger in some conditions,

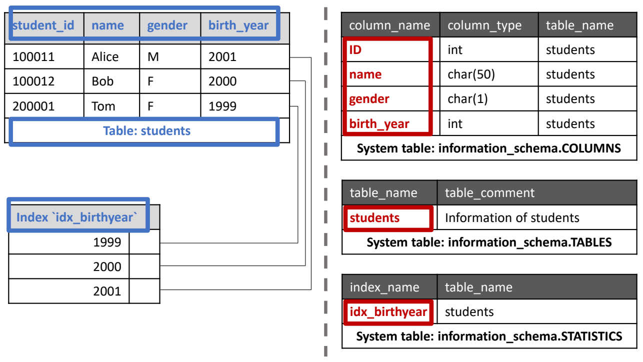


Figure 1: The metadata of a table named "students". It describes the information about the table (i.e., names and comments), columns (i.e., names, data types, corresponding table names), and statistics (i.e., indexes for birth_year).

SQLite uses the "when" clause and "update" statement, MariaDB uses the "if-else" for conditional statements, while PostgreSQL declares an additional function.

Metadata. Metadata in DBMS is data that characterizes the actual data [42, 44]. Simply, it could be defined as data about the data [28]. Specifically, it holds the information about the schema (e.g., tables names and data types) or other related information of the actual data, such as storage (e.g., table size) and data elements (e.g., columns, attributes). For example, Figure 1 shows the metadata of a table named "students". It describes the information about the table (i.e., names and comments), columns (i.e., names, data types, corresponding table names), and statistics (i.e., indexes for birth_year). Metadata plays a crucial role in a DBMS. It can be considered as an index for accessing actual data. Consequently, DBMSs always utilize metadata to check the semantic correctness of SQL statements.

DBMS fuzzing. DBMS fuzzers generate a number of SQL statements and feed them to target DBMSs for execution. Unlike normal programs, DBMSs always have strict syntax and semantic checks on SQL statements. The wrong ones will be discarded directly. To generate syntactically and semantically correct SQL statements, DBMS fuzzers always model the grammar of SQL (e.g., model as an AST) in statement generation.

DBMS fuzzers could be categorized as generation-based and mutation-based. Both of them rely on grammar in statement generation. Generation-based fuzzers generate SQL statements based on a specified model derived from grammar. They need to build a specific grammar generation model for each DBMS. Mutation-based fuzzers modify current test cases to generate new ones. They typically build a specific SQL parser and AST mutator for each DBMS to support the syntax-correct query mutation.

Dependency on grammar limits the adaptability of DBMS fuzzers. The dependency on grammar limits the adaptability of DBMS fuzzers. Specifically, no matter what generation model, SQL parser, or AST mutator, they are all strongly dependent on the DBMS grammar, and adaptation to a new DBMS requires an amount of extra code, which is time-consuming and labor-intensive. Table 2 shows the lines of extra code of SQLsmith, SQLancer, and Squirrel to adapt SQLite, PostgreSQL, MariaDB, and DuckDB. It demonstrates that all three fuzzers require a lot of additional code to adapt to different DBMSs due to the dependency on grammar. For example, SQLancer uses over 8000 lines of code to build the SQL generation model for PostgreSQL. Although SQLsmith requires about 300 lines of code to adapt a new DBMS, it only partially supports the grammar because SQLsmith can only generate SELECT statements for PostgreSQL.

Table 2: The lines of code of SQLsmith, SQLancer, and Squirrel to adapt for SQLite, PostgreSQL, MariaDB, and DuckDB. The numbers of lines of code are calculated from the official repositories of these fuzzers. Note that due to the difficulties in adapting the grammar, SQLsmith does not support MariaDB and Squirrel does not support DuckDB.

LoC	SQLsmith	SQLancer	Squirrel
SQLite	1375	10020	5248
PostgreSQL	1466	8012	7515
MariaDB	-	2085	9050
DuckDB	1325	2597	-

More importantly, there are 391 different DBMSs on the market, most of which use different grammar. Current state-of-the-art DBMS fuzzers only support the most popular DBMSs, like PostgreSQL and SQLite. If we want to adapt the advanced fuzzers to these different DBMSs, it will take too long and require too many human resources to be economically feasible. Therefore, to reduce the adaptation cost and apply fuzzing to more DBMSs, a fuzzer, we urgently need grammar-free DBMS fuzzing.

Challenges in grammar-free DBMS fuzzing. The main challenge of fuzzing DBMSs with grammar-free fuzzing is to generate SQL statements while maintaining syntax and semantic correctness. Without grammar, it is very difficult for a generation-based fuzzer to build a generation model. Thus we mainly focus on mutation-based fuzzing. Without a grammar parser, fuzzer cannot distinguish between tokens in statements. Consequently, random mutations tend to corrupt the delicate structure of SQL statements, making it possible to produce syntax errors. Even if the token in the statement that can be mutated happens to be selected, like the table name in "SELECT", the improper mutation can cause semantic errors. For example, when mutating the statement "Select Name From Student", the variable "Student" might be mutated. But when it is mutated to an arbitrary table name that does not exist in the DBMS, it will cause semantic errors.

The basic idea of Griffin: make the best use of metadata in DBMSs. To address the challenge (i.e., maintain syntax and semantic correctness), rather than relying on grammar, the basic idea of Griffin is to summarize the DBMS's states into *metadata graph*, a lightweight data structure that improves mutation effectiveness of fuzzing. First, Griffin reshuffles SQL statements from *existing valid SQL test cases* to preserve the syntax correctness of the newly generated SQL test cases. Then, Griffin mutates the variables of each SQL statement based on metadata information of DBMS to ensure semantic correctness. As mentioned earlier, the metadata

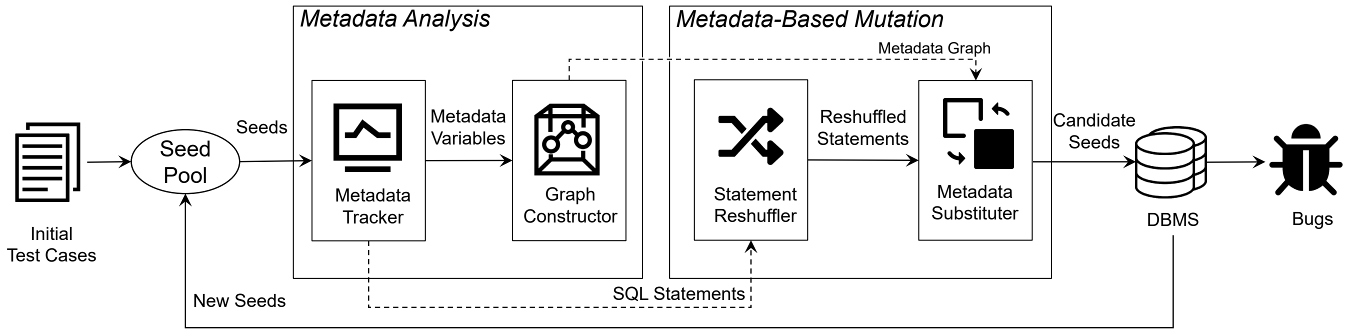


Figure 2: Design of Griffin. Griffin tries to generate SQL test cases in a grammar-free way. It has the following steps: (1) Metadata analysis. Griffin tracks the metadata of the statements as they are executed. Based on the metadata, Griffin constructs the metadata graph to describe the dependencies of metadata and statements. (2) Metadata-based Mutation. Griffin first reshuffles statements from existing test cases and then reconstructs the metadata graph by substituting the metadata node to correct semantic errors. With metadata-guided mutation, Griffin continuously generates semantic correct SQL test cases to find bugs in DBMSs. More importantly, Griffin can be conveniently adapted to various databases because it is free from the dependencies of grammar.

describes the states of a DBMS, i.e., information about database objects such as tables, views, triggers, and indexes. Therefore, based on the information of metadata in DBMS, we could identify the variables (e.g., table name or column name) in SQL statements and substitutes the incorrect variables to ensure semantic correctness. Benefit from the valid SQL statement generation, Griffin is effective to trigger various behaviors of DBMSs. More importantly, Griffin is free from the specific grammar of DBMSs, thus it can easily be adapted to various DBMSs.

3 DESIGN

Figure 2 shows the overall design of Griffin. It first analyzes the metadata for each statement to build metadata graphs iteratively. Based on the graphs, it reshuffles statements and employs metadata-guided substitution to correct semantic errors.

3.1 Metadata Analysis

When Griffin gets a new test case, it analyzes the metadata used for each statement and the metadata for the whole DBMS. Specifically, it first tracks the metadata of each statement as they are executed. Based on the metadata, Griffin constructs the metadata graph to describe the dependencies of metadata and statements.

3.1.1 Metadata tracking. We use the **metadata graph** as a generic description of metadata. It is a directed graph in the form (V_m, V_s, E), where V_m, V_s represent two types of nodes, and E represents edges. Specifically, V_m notates the set of all metadata nodes, which represent database objects that can be found by a metadata query, such as tables, columns, or triggers. V_s notates the set of all statement nodes. A statement node represents a SQL statement. Correspondingly, the graph has three types of edges. (1) The edge from the metadata node `m` to the statement node `s` means that the statement in `s` is semantically correct only if the database object in `m` exists in the current database. (2) The edge from statement node `s` to metadata node `m` means the database object in `m` will be created in the current database if `s` is executed successfully. (3) The edge from metadata node `m1` to metadata node `m2` means the database object `m2` is contained by `m1`, such as a table contained by a schema or a column contained by a table.

```
SELECT name, type, parentname
FROM (SELECT table_name as name, "TABLE" as type,
             null as parentname
      FROM information_schema.tables
      UNION
      SELECT column_name, column_type, table_name
      FROM information_schema.columns
      UNION
      SELECT trigger_name, "TRIGGER", null
      FROM information_schema.triggers
      UNION
      SELECT index_name, "INDEX", null
      FROM information_schema.statistics) AS t;
```

Figure 3: A query to get metadata for DBMSs supporting the information schema feature. Each row of the query result corresponds with a metadata node, namely `m`. For the columns of the query result, `name`, `type`, and `parentname` represent the attribute name, type, and parent object of `m`, respectively.

Particularly, each statement node `s` has the attribute `s.text`, which means the plain text of the statement. Each metadata node `m` has the attributes `m.name` and `m.type`. `m.name` represents the object name like table name, and `m.type` refers to the object type like "TABLE". For example, after running the statement "`CREATE TABLE book (id INTEGER)`", the DBMS will have a new table (i.e., named "book") in the schema (i.e., named "`main`"), and the table has an integer column named "id". The metadata can be represented by two nodes, namely `id`, `book`, where `id.name` = `'id'`, `id.type` = `'TABLE'`, `book.name` = `'name'`, and `book.type` = `'INTEGER'`.

To track metadata and obtain current metadata nodes, including their names and types, and the containment relationships between database objects, we can execute specific queries. Specifically, for DBMSs supporting the ANSI-standard information schema, we can execute the queries shown in Figure 3. For others, their specialized metadata queries could be used, such as the queries on the sqlite_master table in SQLite.

3.1.2 Metadata graph constructing. To build the metadata graph, we should determine the statement nodes, metadata nodes, and edges between them to be added to the graph. Suppose we use S_i to represent the i-th SQL statements, We use the following rules to add edges in the metadata graph according to the dependencies between the statement S_i and the DBMS metadata:

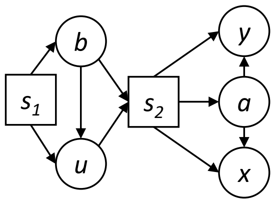
(1) If a database object `m` is used by the statement S_i, we need to add an edge from `m` to S_i into E. It must satisfy two requirements: ① database object `m` is created before executing the statement S_i. We can find out which metadata are created before the statement S_i through the aforementioned metadata tracking. ② database object `m` appears in the statement S_i in some form. Since the statement S_i can be considered as plain text following the SQL grammar as S_i`.text`, we can check whether the m`.name` field appears in S_i`.text`.
(2) If a database object m is created by the statement S_i, we need to add an edge from S_i to `m`. It can also be determined by metadata tracking.
(3) If a database object `m2` is contained by another database object `m1`, such as a column contained by a table or a table contained by a schema, we need to add an edge from `m1` to `m2`. Similarly, containment relationships can be calculated by metadata tracking.

Algorithm 1: Build the Metadata Graph with Statements

```
Input  : s: the statement node.
         G: the metadata graph.
begin
    $G^* \leftarrow G$;
    Add node $s$ into $G^*$;
    foreach metadata node $m$ in $G$ do
        if IsSubstring($m.name, s.text$) then
            Add edge $(m, s)$ into $G^*$;
    end
    foreach $m \in$ MetadataCreatedBy($s$) do
        Add node $m$ into $G^*$;
        Add edge $(s, m)$ into $G^*$;
        if ParentObject($m$) is not null then
            Add edge (ParentObject($m$), $m$) into $G^*$;
        end
    end
    $G \leftarrow G^*$;
end
```

Algorithm 1 illustrates the process of building a metadata graph based on the information collected by metadata tracking. It first copies the current metadata graph G to G^*. Then it adds the statement node `s` to G^*. For each metadata node `m` of G, it adds an edge from the metadata node `m` to the statement node `s` if the name of the node is a substring of the statement's plain text. For each metadata node created by `s` (for example, the statement "CREATE table `t` (a int)" will create the node for the database object "`t`" and the column "`a`"), it will add the metadata node `m` into G^* and add an edge from the statement node `s` to the metadata node `m`. If metadata node `m` has a parent object, it will also add an edge to m from the node representing the parent object. Finally, it will replace G with G^*.

For example, considering we have an existing test case shown in Figure 4, it does the following steps with two SQL statements: ① Creates a table named "b", ② Creates a table named "a" referencing table "b". After running the first statement, a database object table named "b" and another column object "u" are generated in the database, so the metadata nodes `b`, `u`, and the edge `(b, u)` will be added to the metadata graph. In addition, considering that the first statement adds `b` and `u`, we add the statement node `s1` to the metadata graph and add the edges `(s1, b)`, `(s1, u)` to E. Similarly, the second statement adds database objects named 'a', 'x', 'y', with edges `(s2, a)`, `(s2, x)`, and `(s2, y)` added into E.



Stmt node	s.text

s1	`CREATE TABLE b(`
	`  u INT PRIMARY KEY);`

s2	`CREATE TABLE a(`
	`  x INT PRIMARY KEY,`
	`  y INT REFERENCES`
	`b(u));`

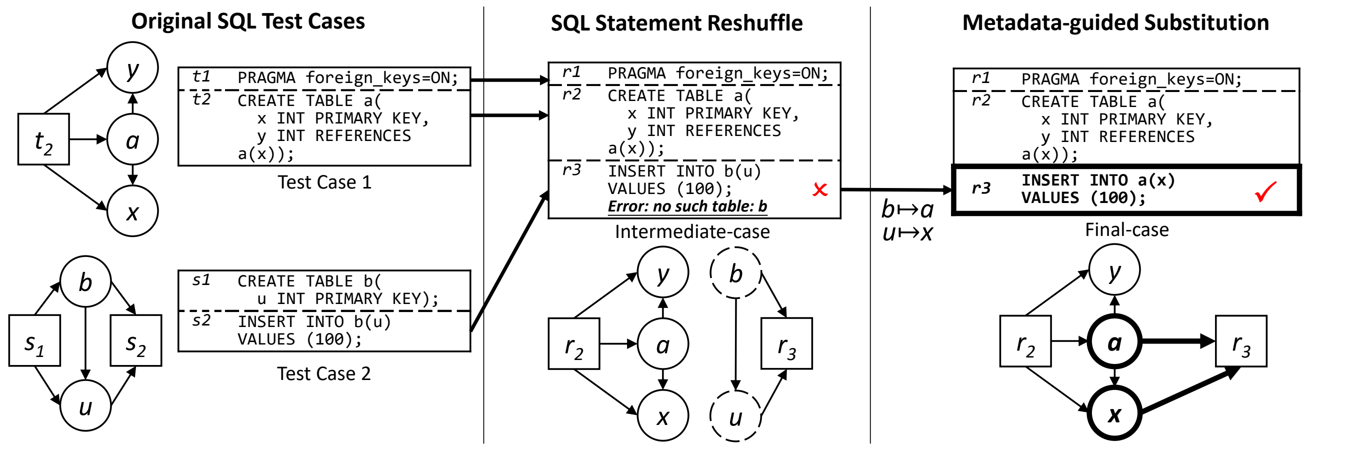
Metadata node	m.type
b	TABLE
u	INT
a	TABLE
x	INT
y	INT

Figure 4: The example of metadata analysis for building the metadata graph. There are two kinds of nodes: statement nodes with a square shape and metadata nodes with a circular shape. The graph shows the relation between the nodes. Specifically, with the "`s1`" statement executed, it creates two metadata nodes "b" and "u". Similarly, the statement node "`s2`" creates three metadata nodes "a", "y", and "x". Note that nodes "x" and "y" are linked with node "a" because "x" and "y" are attributes of table "a" according to the "`s2`" statement, as shown in "`s.text`".

3.2 Metadata-Based Mutation

Metadata-based mutation generates semantically-correct new test cases by the following steps: first, reshuffle the SQL statements from original SQL test cases; second, correct semantic errors with metadata-guided substitutions. Figure 5 gives an example of the metadata-based variation process. Griffin first reshuffles the SQL statements of the original SQL test cases and generates a new test case named intermediate-case, as shown in the middle part. The intermediate-case in the middle part contains some semantic errors and an incorrect metadata graph. Then, Griffin corrects the semantic errors to generate the final test case by reconstructing its metadata graph with metadata-guided substitution.

3.2.1 SQL statement reshuffle. The reshuffle operation aims to create a new test case from existing test cases. First, it randomly selects several existing test cases as the source for mutation. It then



```
t1  PRAGMA foreign_keys=ON;
t2  CREATE TABLE a(
      x INT PRIMARY KEY,
      y INT REFERENCES
    a(x));
```

```
s1  CREATE TABLE b(
      u INT PRIMARY KEY);
s2  INSERT INTO b(u)
    VALUES (100);
```

```
r1  PRAGMA foreign_keys=ON;
r2  CREATE TABLE a(
      x INT PRIMARY KEY,
      y INT REFERENCES
    a(x));
r3  INSERT INTO b(u)
    VALUES (100);
    Error: no such table: b
```

```
r1  PRAGMA foreign_keys=ON;
r2  CREATE TABLE a(
      x INT PRIMARY KEY,
      y INT REFERENCES
    a(x));
r3  INSERT INTO a(x)
    VALUES (100);
```

Figure 5: An example of metadata-based mutation. First, Griffin reshuffles the SQL statements of original test cases to generate the intermediate-case. However, the intermediate-case contains semantic errors because it tries to insert a value into a non-existent table "b". Second, Griffin uses metadata-guided substitution to correct semantic errors by reconstructing the metadata graph of intermediate-case. It substitutes "b" and "u" with "a" and "x" in the metadata graph to correct the error in the "s3" statements and generate the final-case.

Algorithm 2: Find the substitutable metadata

```
Input  : $G$: metadata graph of current DBMS.
         $G_0$:metadata graph of the test case.
         $s_0$: the statement node in $G_0$ to substitute.
Output : Possible substitution for predecessors of $s_0$, a
         map
begin
    match ⟵ map();
    pre ⟵ metaPredecessors($s_0, G_0$);
    foreach $m_0 \in$ pre, $m \in$ MetadataNodes($G$) do
        if IsSubstitutable($m_0, G_0, m, G$) then
            match[$m_0$].insert($m$);
        end
    end
    return match;
end
Function IsSubstitutable($m_0, G_0, m, G$):
    if $m_0.type \neq m.type$ then return false;
    foreach $n_0 \in$ metaSuccessors($m_0, G_0$) do
        matchable ⟵ false;
        foreach $n \in$ metaSuccessors($m, G$) do
            if IsSubstitutable($n_0, G_0, n, G$) then
                matchable ⟵ true;
            end
        end
        if matchable is false then return false;
    end
    return true;
```

does the following operations repeatedly. (1) Randomly select an SQL test case. (2) Add the current statement of the test case to the candidate case. (3) Delete the current statement of the test case and move to the next statement. These operations are performed continuously until no statement exists in all original test cases. Finally, the reshuffle operation will remove some statements from the new test case. Otherwise, since the new test case is as large as the two test cases combined, the generated test cases will get larger and larger, which is harmful to the performance of fuzzing. To reduce the size of the new test case, we remove each statement in it with a certain probability. For example, as Figure 5 shows, the reshuffle operation combines the first statement of Test Case 1, the second statement of Test Case 1, and the second statement of Test Case 2 into intermediate-case.

Reshuffling can generate new test cases with different combinations and sequences of statements. However, it might break the semantic correctness. For example, in Figure 5, the statement "r3" will throw a semantic error because it tries to insert a value into a non-existent table "b". To maintain semantic correctness, we need to substitute the metadata nodes in these statements by type and relational predicates in the next subsection.

3.2.2 Metadata-guided substitution. Metadata-guided substitution aims to correct semantic errors in reshuffling by rebuilding the metadata graph of test cases. For example, as aforementioned, in Figure 5, the reshuffled SQL test case contains a semantic error. Therefore, the metadata graph of the new SQL test cases is unconnected. Specifically, the nodes r2 and r3 in the middle part are unconnected, because table b is not created in advance. Griffin corrects the semantic errors by reconstructing the metadata graph with metadata-guided substitution.

Incorrect metadata nodes locating. Griffin first locates the incorrect metadata nodes in the old metadata graph G_0, and substitutes the metadata nodes by following two predicates:

(1) The type of metadata should match.
(2) The relation of metadata should match.

Specifically, for the first predicate, the type of each metadata node should keep the same before and after substitutions. For example, for the statement "`ALTER TABLE t ...`", if we substitute "`t`" with "`a`", then "`a`" should also represent a table in the current DBMS. Otherwise, it will encounter a semantic error like "no such table 'a"'. For the second predicate, the relation between metadata nodes should keep the same before and after substitutions. For example, for the statement "`SELECT * FROM student.name`", "`name`" is a column of the table "`student`". If we substitute "`student`" and "`name`" with "`x`" and "`y`", then "`y`" should also represent a column in the table "`x`". Otherwise, it will encounter semantic errors such as "There is no 'y' column in the 'x' table".

Substitutions calculating. Griffin then calculates all possible substitutions that satisfy the two predicates for each metadata node in G_0 with Algorithm 2. For each statement node `s0` in the metadata graph, it first computes all the predecessor metadata nodes of the statement node (Lines 2-3). Then, for each metadata node of statement node `s0`, Griffin records all matchable metadata nodes into a map by determining whether the metadata substitutable with the function `isSubstitutable` (Lines 4-8). To determine whether the metadata node `m` in the metadata graph is matchable for node `m0`. Griffin first checks the type of two metadata nodes. If `m0` and `m` are not type-matchable, they are not matchable. For type-matchable metadata nodes `m0` and `m`, if they have matchable successor metadata nodes, `m0` and `m` are matchable (Lines 11-21).

For example, the metadata graph of the intermediate-case contains errors because of the no-existing nodes `b` and `u` as Figure 5 shows. Griffin substitutes them with some of the nodes existing: `a`, `x`, or `y`, through Algorithm 2. It finds that there are two possible substitutions: ① `b` to `a` and `u` to `x`, ② `b` to `a` and `u` to `y`. The case named `final-case` in Figure 5 shows the result of the first substitution for node `r2`. The edges `(b, r3)` and `(u, r3)` are replaced by edges `(a, r3)` and `(x, r3)`. Consequently, the string "b" and "u" in the text of statement `r3` are replaced by "a" and "x".

4 IMPLEMENTATION

We implement Griffin with 9,532 lines of code in C++. We build Griffin on top of AFL++ 4.00c [12] and implement the custom mutator feature with our metadata analysis and metadata-based mutation. The metadata analyzer and metadata mutator are two main components of Griffin. The metadata analyzer implements Algorithm 1 to build the metadata graph and update the graph while fuzzing. The metadata mutator is used to reshuffle the SQL statements from original seeds and implements Algorithm 2 to find the matchable metadata for metadata-guided substitution.

The effort to adapt Griffin to a new DBMS is to implement a client. First, the client should send SQL queries to the target DBMS for execution. The client can be implemented with the Open Database Connectivity (ODBC) driver, which is common for different DBMSs. Second, we should customize the way of querying metadata from the target DBMS in the client. Basically, for DBMSs that support ANSI-standard [2], we can get the metadata with `SELECT` statements. For other DBMSs, we need to use their specific APIs to implement this part for querying metadata.

5 EVALUATION

We evaluated Griffin in terms of its ability to find new bugs, as well as the efficiency of grammar-free fuzzing on real-world DBMSs. Our evaluation aims to answer the following questions:

(1) What's the performance on bug detection of Griffin in real-world DBMSs?
(2) How about the coverage and bug findings when comparing Griffin with other DBMS fuzzers?
(3) What are the contributions of metadata-based mutation to fuzzing coverage?

Test DBMSs. To demonstrate the bug founding ability of Griffin cross various DBMSs, we directly apply Griffin to three well-tested DBMSs MariaDB, SQLite, and PostgreSQL, as well as one newly developed DBMS DuckDB without any specific adaptations. The chosen versions of these DBMSs are SQLite 3.37.0, DuckDB 0.3.2, MariaDB 10.7.1, and PostgreSQL 14.1.

DBMS fuzzers. To evaluate the effectiveness of Griffin, we choose SQLsmith, SQLancer, and Squirrel, which are widely used in industry and academia, for performance comparison. SQLancer and SQLsmith are generated-based DBMS fuzzers. They can generate SQL queries continuously for fuzzing without initial seeds. Griffin and Squirrel are mutate-based DBMS fuzzers, which need some SQL queries as the initial seeds for fuzzing. We collected open-source test cases from the official regression test suite of each DBMS and extracted SQL statements from them as the initial seeds of Squirrel and Griffin. Note that due to Squirrel does not support the grammar of DuckDB and SQLsmith does not support the grammar of MariaDB, we skipped the evaluation of Squirrel on DuckDB and SQLsmith on MariaDB.

Basic setup. The evaluation was performed 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. When compiling and running, all the DBMSs enabled the AddressSanitizer(ASAN) [34] for bug finding and the LLVM source-based code coverage [37] to collect code coverage. For the real-world bug finding, Griffin runs on SQLite, DuckDB, MariaDB, and PostgreSQL for one week. For quantitative comparisons, we run all fuzzers on each DBMS for 12 hours with 2 CPUs and 20G memory.

5.1 DBMS Bug Detection

Griffin has successfully detected 55 previously unknown bugs in four well-tested DBMSs within one week. Table 3 shows the statistics of bugs detected by Griffin in each DBMS including 16 in SQLite, 12 in DuckDB, 24 in MariaDB, and 3 in PostgreSQL, respectively. It shows that Griffin with AddressSanitizer deployed can detect various bugs including 4 undefined behaviors, 15 assertion failures, 3 NULL pointer dereferences, 21 segmentation violations, 1 buffer-overflow, 5 use-after-poison, and 6 heap-use-after-free. We have actively reported all the bugs to the corresponding DBMS vendors, all of these bugs have been confirmed and 50 of them have been fixed. In particular, 13 bugs are assigned with unique CVE IDs in the NVD [31].

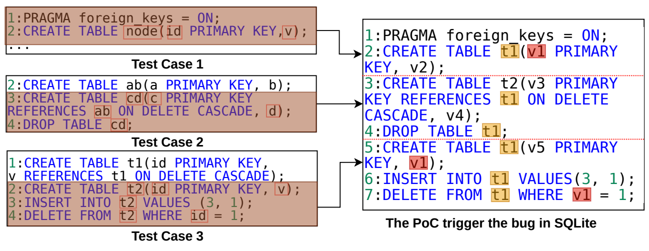
We also analyze the proof of concepts (PoCs) that trigger these bugs in DBMSs. The result shows that 33 of them cannot be generated by other DBMS fuzzers directly. For SQLancer and SQLsmith,

Table 3: Statistics of previously unknown bugs detected by GRIFFIN, 55 confirmed with 13 CVEs assigned.

Project(fixed/detected)	Component	Bug Type
SQLite (16/16)	insert	Segmentation Violation (1)
	btree	Segmentation Violation (1), Assertion Failure (2)
	pager	Assertion Failure (2)
	optimization	Assertion Failure (1)
	trigger	Assertion Failure (2)
	parse	Assertion Failure (1)
	window	Segmentation Violation (1)
	journal	Assertion Failure (1)
	prepare	Assertion Failure (1)
	bad assert	Assertion Failure (3)
DuckDB (11/12)	common	Segmentation Violation (1), Use-After-Free (2), Null Pointer Dereference (1)
	execution	Null Pointer Dereference (1)
	function	Undefined Behavior (2)
	sqlite api	Null Pointer Dereference (1)
	optimizer	Undefined Behavior (1)
	planner	Assertion Failure (1)
	parallel	Undefined Behavior (1)
	storage	Use-After-Free (1)
MariaDB (20/24)	virtual columns	Use-After-Poison (4), Segmentation Violation (7), Buffer-Overflow (1)
	optimizer	Use-After-Free (1), Segmentation Violation (4)
	update	Use-After-Free (1), Segmentation Violation (1), Use-After-Poison (1)
	parser	Use-After-Free (1)
	handler	Segmentation Violation (1)
	item_timefunc	Segmentation Violation (1)
	storage engine	Assertion Failure (1)
PostgreSQL (3/3)	optimizer	Segmentation Violation (3)
Total	25 components	50 fixed, 55 confirmed with 13 CVEs

they cannot generate these SQL test cases due to the lack of grammar rules support. For SQUIRREL, although the initial seeds contain some specific SQL grammar, it skips the mutation of those seeds because they cannot be recognized by the inner parser of SQUIRREL, which does not support the specific grammar. In contrast, GRIFFIN is designed to be independent of the grammar, and it can make use of these SQL test cases in initial seeds with metadata mutation to find those bugs. In addition, after identifying and fixing these bugs, the vendors of these DBMSs also add these SQL test cases that trigger these bugs to the regression test case set of these DBMSs.

Case study. We introduce and analyze a Assertion Failure found by GRIFFIN in SQLite, which has lied 13 years since the foreign key constraints were first introduced in SQLite version 3.6.19 in 2009. It happens when the SQLite uses an invalid virtual trigger on foreign key constraints.



```
1:PRAGMA foreign_keys = ON;
2:CREATE TABLE node(id PRIMARY KEY,v);
...
```

```
2:CREATE TABLE ab(a PRIMARY KEY, b);
3:CREATE TABLE cd(c PRIMARY KEY
REFERENCES ab ON DELETE CASCADE, d);
4:DROP TABLE cd;
```

```
1:CREATE TABLE t1(id PRIMARY KEY,
v REFERENCES t1 ON DELETE CASCADE);
2:CREATE TABLE t2(id PRIMARY KEY, v);
3:INSERT INTO t2 VALUES (3, 1);
4:DELETE FROM t2 WHERE id = 1;
```

```
1:PRAGMA foreign_keys = ON;
2:CREATE TABLE t1(v1 PRIMARY
KEY, v2);
3:CREATE TABLE t2(v3 PRIMARY
KEY REFERENCES t1 ON DELETE
CASCADE, v4);
4:DROP TABLE t1;
5:CREATE TABLE t1(v5 PRIMARY
KEY, v1);
6:INSERT INTO t1 VALUES(3, 1);
7:DELETE FROM t1 WHERE v1 = 1;
```

Figure 6: The PoC and the generation process that triggers the bug hidden in SQLite since 2009. GRIFFIN generates the PoC with the three test cases on the left by metadata analysis and metadata-based mutation. GRIFFIN first reshuffles the SQL statements of Test Case 1, Test Case 2, and Test Case 3 to generate new SQL test cases. With the metadata analysis, GRIFFIN eliminates semantic errors by reconstructing the metadata graph.

The mechanism to trigger the bug. The right part of Figure 6 shows the SQL query that triggers a Assertion Failure in SQLite. First, it enables the foreign key feature of SQLite with the PRAGMA statement. Then, it creates two tables named "t1" and "t2". Note that table "t2" contains an ON DELETE CASCADE constraint on the primary key "c", which references the primary key "a" of the table "t1". The constraints ON DELETE CASCADE will generate a virtual trigger in SQLite to perform a cascade delete automatically. This virtual trigger is cached in the schema definition of the table "t2". After executing the statement DROP TABLE t1, the definition of table "t1" and the virtual trigger on "t1" should have been removed. However, the virtual trigger on "t1" is not being cleared due to the wrong implementation of constraints on foreign keys. As a result, with the new definition of the new table "t1", the cached old virtual trigger becomes invalid, causing a Assertion Failure when a delete operation is performed on the table "t1".

The developers of SQLite responded that this bug is a real, long-standing bug for 12 years. It arises when the foreign key constraint is first developed. The developers have written a number of related test cases to ensure the correctness of the foreign key constraints, but the bug is still missed by them. Consequently, the PoC we reported that triggered the bug has been added to the regression test case set of SQLite.

The reason for detecting the bug by GRIFFIN. The PoC to trigger this bug invokes three tables, three kinds of operations on the table, and two kinds of keys on the table, which contain abundant combination and semantic information. It is difficult to manually list all combinations of operations and make sure the semantics of test cases are correct. Therefore, the unit test case sets of SQLite do not find the bug. In addition, SQLancer, SQLsmith, and SQUIRREL have not found this bug, although they have been fuzzing SQLite for a few years. The main reason is the limitation of the SQL grammar rules. SQLsmith can only generate SELECT statements, while SQLancer can not generate DELETE statements, because their generation models are based on the limited SQL grammar. Similarly, SQUIRREL can not generate the PoC in the right part of Figure 6 to trigger this bug because it does not support parsing and generating the ON DELETE CASCADE grammar yet.

In contrast, GRIFFIN detected the bug in three days when fuzzing with the unit test case set as the initial seeds. With the grammar-free design, GRIFFIN can mutate all kinds of SQL statements of various DBMSs. Figure 6 shows the detailed process of generating the PoC to trigger this bug. Test Cases 1, Test Case 2, and Test Case 3 are the SQL queries before mutation. GRIFFIN first builds a metadata graph by locating all the table and column names in the SQL statement through metadata queries. Then, it reshuffled the SQL statements of original inputs and generated new SQL queries by metadata-guided substitution. For example, the table names in Test Cases 1 and Test Case 3 are all replaced with "v0". Finally, GRIFFIN detected the bug that was missed by other DBMS fuzzers.

5.2 Comparison with Existing Fuzzers

To evaluate the performance, we perform fuzzing with GRIFFIN, SQLsmith, SQLancer, and SQUIRREL on each test DBMS for 12 hours. We use the number of branches and the number of unique crashes as the basic metrics of the experiments. For a fair comparison, when we finished the fuzzing experiments, we collected the SQL queries generated by each fuzzer and dry-ran the queries to unify the branch coverage.

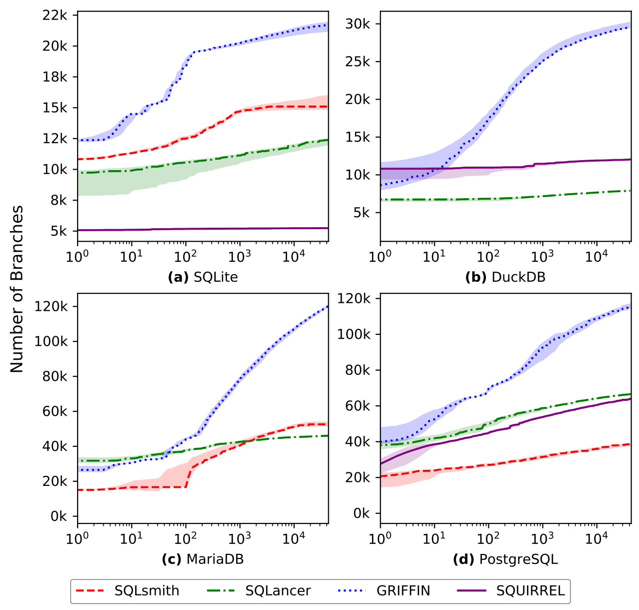


Figure 7: The growing trend of the number of branches covered by different DBMS fuzzers in 12 hours. Displayed are the medians and the 99% confidence intervals of 15 runs. Note that Squirrel does not support DuckDB and SQLsmith does not support MariaDB.

Coverage. Figure 7 shows the number of branches covered by each DBMS fuzzer in 12-hour experiments. From the figure, we can see that Griffin outperformed other fuzzers in terms of branch coverage. Specifically, Griffin covers 73.43%-274.70%, 80.47%-312.89%, 43.80%-199.11% more branches than SQLancer, SQLsmith, and Squirrel after fuzzing 12 hours, respectively.

The main reason for the improvement can be explained by the grammar-free mutation of Griffin. The two generation-based fuzzers, SQLancer and SQLsmith, can only generate SQL statements based on their predefined models, which cover only a portion of the entire SQL grammar of the DBMS. For example, SQLsmith can only generate SELECT statements for PostgreSQL because of limited predefined AST models. Therefore, SQL queries that conform to the SQL grammar of the DBMS but are not built in the predefined model cannot be generated by SQLsmith and SQLancer. For Squirrel, the mutated-based DBMS fuzzer, it can only apply mutation on the SQL statements which can be parsed by the inner SQL parser. However, due to the parser of Squirrel only supporting part of the SQL grammar, the mutable SQL queries are limited. In contrast, Griffin can mutate any statement in the input, as long as it can be executed by the target DBMS, in a syntactically and semantically correct way since Griffin is designed to be grammar-free. In other words, Griffin can generate a large number of new inputs with various grammar by mutating existing inputs. Consequently, Griffin can achieve higher branch coverage compared to other DBMS fuzzers.

Unique crashes. Table 4 shows the number of unique crashes found by each DBMS fuzzer after fuzzing for 12 hours. From the table, we can see that Griffin finds more unique crashes than other state-of-the-art DBMS fuzzers. Specifically, Griffin found 27, 27, and 22 more unique crashes than SQLsmith, SQLancer, and Squirrel in 12 hours, respectively.

Table 4: Number of unique crashes triggered in 12 hours of each DBMS fuzzer on SQLite , DuckDB , MariaDB , and PostgreSQL . Shown are the medians of 15 repeated times.

	SQLsmith	SQLancer	Squirrel	Griffin
SQLite	0	0	0	1
DuckDB	0	0	-	9
MariaDB	-	0	5	15
PostgreSQL	0	0	0	2

Based on the improved coverage and mutated abundant kinds of SQL statements, Griffin can trigger more various behaviors of DBMS. As the case study shows, Griffin can generate SQL queries that other DBMS fuzzers can not generate based on limited SQL grammar. In addition, based on the metadata mutation, Griffin can reshuffle new SQL statements and generate new semantically-correct test cases with metadata-guided substitution. As a result, Griffin triggered the behaviors that other DBMS fuzzers can not, and found more new crashes than other DBMS fuzzers.

5.3 Effectiveness of Metadata-Based Mutation

To understand the contribution of the metadata mutation in Griffin, we implemented Griffin- which disables metadata mutation. We use the *semantic correctness ratio* (i.e., the proportion of the semantically correct SQL queries collected during fuzzing to all generated SQL queries) to measure the effectiveness of metadata mutation in improving semantic correctness. Note that metadata mutation is used to mutate variables in statements and to ensure semantic correctness by type checking and relation checking.

Table 5 shows the semantic correct ratio by Griffin and Griffin- on four DBMS while fuzzing for 12 hours. From the table, we can see that Griffin generated more semantic-correct SQL queries and covers more branches than Griffin-. Specifically, compared with Griffin-, Griffin generated 92.44%, 124.93%, 33.42%, and 149.68% more semantic-correct SQL queries, and covered 24.58%, 57.18%, 44.45%, and 107.02% more branches on PostgreSQL, MariaDB, SQLite, DuckDB, respectively. The result is reasonable because we design the metadata mutation to improve the semantic correctness. Without metadata mutation, Griffin- could only mutate new SQL queries by reshuffling the SQL statements from the initial seeds. As a result, most of the inputs generated by Griffin- contain semantic errors that can not pass the semantic checks of the DBMS and do not explore more state-space of DBMS.

From the table, we can also notice that Griffin only achieves 33.42% improvement in the ratio of semantic-correct SQL queries in SQLite. This can be explained by the flexible typing feature of SQLite [36]. SQLite is a library-feature DBMS and is designed with a flexible type check feature. For the SQL statements, it stores and executes without checking the real semantic correctness of the

Table 5: The semantic correctness ratios achieved and number of branches covered by GRIFFIN- and GRIFFIN in 12 hours.

	Semantic Correctness Ratios			Number of Covered Branches		
	GRIFFIN-	GRIFFIN	Improvement	GRIFFIN-	GRIFFIN	Improvement
SQLite	0.7187	0.9589	33.42 %↑	15243	22019	44.45%↑
DuckDB	0.3776	0.9429	149.68%↑	14568	30159	107.02%↑
MariaDB	0.3378	0.7598	124.93%↑	77755	122219	57.18%↑
PostgreSQL	0.3204	0.6166	92.44 %↑	102359	127521	24.58%↑

data type. As a result, many SQL queries which contain data type mismatching are still recognized by SQLite as semantic-correct. In contrast, PostgreSQL, MariaDB, and DuckDB check the semantic information to formulate query plans before execution. Therefore, any SQL statement containing semantic errors is considered invalid before execution. Thus, GRIFFIN achieves more improvement on PostgreSQL, MariaDB, and DuckDB than on SQLite.

Table 6: P-values and effect sizes of the comparison with other fuzzers. All p-values are statistically significant ($p < 0.05$), and all effect sizes are large ($\hat{A}_{12} > 0.71$).

		P-Value		Effect Size	
v.s. Fuzzer	DBMS	Branch	Crash	Branch	Crash
SQLsmith	SQLite	**0.000003**	**0.00007**	**1**	**0.8667**
	DuckDB	**0.00000001**	**0.0000003**	**1**	**1**
	PostgreSQL	**0.00000001**	**0.0000005**	**1**	**1**
SQLancer	SQLite	**0.00000001**	**0.00007**	**1**	**0.8667**
	DuckDB	**0.000003**	**0.0000003**	**1**	**1**
	MariaDB	**0.00000001**	**0.0000006**	**1**	**1**
	PostgreSQL	**0.00000001**	**0.0000005**	**1**	**1**
SQUIRREL	SQLite	**0.00000001**	**0.00007**	**1**	**0.8667**
	MariaDB	**0.00000001**	**0.000003**	**1**	**1**
	PostgreSQL	**0.00000001**	**0.0000005**	**1**	**1**

6 DISCUSSION

This section shows some limitations of GRIFFIN and evaluation, and the plans to address them in future work.

Threats to validity of the evaluation. In the evaluation of comparing GRIFFIN with other fuzzers, GRIFFIN achieved the highest coverage branch counts and crash numbers in 12 hours on SQLite, DuckDB, MariaDB, and PostgreSQL, with 15 times repeated as well as significant p-values and large effect sizes shown in Table 6. However, for *threats to external validity*, we only conducted experiments on relational DBMSs because other fuzzers only support the grammar of these DBMSs. For other DBMSs, because of the metadata-based mutation, GRIFFIN could still test them. For *threats to internal validity*, first, the bug finding comparison shown in Table 4 only considered unique crash bugs de-duplicated by the top stack frame when DBMS crashing; it leads to non-crash bugs being ignored and maybe some crash bugs being misclassified. Second, the experiment used the same initial seeds between multiple runs when fuzzing each DBMS with GRIFFIN and Squirrel, but it did not consider the impact of seed choices on the performance of fuzzers.

Domain-specific bugs. With the metadata-based mutation, GRIFFIN can generate numerous semantically-correct SQL statements and improve the code coverage to trigger bugs in the deep logic of DBMS during fuzzing. Following the practice of mutation-based fuzzing [7, 10, 11, 13, 38], GRIFFIN uses ASAN to detect memory safety bugs. In the same way, GRIFFIN can also integrate some other specific sanitizers to detect domain-specific bugs like the ACID counterexample of a DBMS. Nevertheless, a prerequisite for triggering these bugs is to cover the corresponding specific code during fuzzing. GRIFFIN ensures the semantic correctness of SQL statements with the metadata-based mutation and improves the code coverage of fuzzing, which could help to trigger more logic of the DBMS, and finally trigger more bugs.

Dependency on built-in test cases. Designed with grammar-free mutation, GRIFFIN is highly dependent on the initial built-in test cases because it generates new SQLs by reshuffling the SQL statements from the original test cases and ensuring semantic correctness with metadata-guided substitution. Unlike other grammar-based DBMS fuzzers, GRIFFIN cannot generate new SQL structures that are not contained in the initial statements. As a result, when only limited initial seeds are given and cover only a little SQL grammar of DBMS, GRIFFIN's performance will decrease because it is designed without any SQL grammar. However, for commonly used DBMS, it is not difficult to collect grammar-rich SQL queries. The built-in test cases of DBMS are used to test various database functionalities and can cover almost all the SQL grammar supported by DBMSs. Among the 391 DBMS in the market [19], 206 open-source DBMS' built-in test cases can all be directly collected from their repositories. Based on these built-in test cases, GRIFFIN generated abundant semantic-correct test cases and finally found 13 CVEs in the evaluation. Nevertheless, it may still make sense to handle the situation when there is a limited number of original seeds. One possible solution is to transform SQL queries from other DBMSs into the initial seeds. We plan to extend it in the future.

Semantic errors. Although GRIFFIN can improve semantic correctness through metadata-based mutation, it still generates some semantically incorrect test cases due to the specific constraint feature of the DBMS. For example, if a test case tries to insert the same value twice in a primary key column, the DBMS cannot execute the second insertion statement due to the primary key constraint. However, GRIFFIN cannot correct such a semantic error because GRIFFIN only considers metadata dependencies, while primary key constraints are related to both metadata dependencies and data dependencies. This kind of semantic error is detrimental to fuzzing data-sensitive DBMS components such as storage engines and optimizers. We plan to add support for data-based checks in the future.

7 RELATED WORK

In this section, we will talk about the works about DBMS Testing and the main difference between GRIFFIN and them.

Generation-based DBMS fuzzing. Generation-based fuzzing [21, 33, 35] generates enormous SQL queries for execution with predefined generation models. SQLsmith [33], one of the state-of-the-art generation-based DBMS fuzzers, can generate an amount of semantic-correct SQL queries continuously at a high speed based on the predefined AST model. However, it can only generate limited varieties of SQL queries (e.g., only SELECT statements) restricted by the grammar rules of the generation model. Some other works detect logic bugs in DBMS by different testing. RAGS [35] detects bugs by comparing the results of different DBMS with the same SQL queries. Due to the difference in the SQL grammar in the two DBMSs, only part of the functionalities can be tested. APOLLO [21] generates queries and checks the performance on different versions of target DBMS to find performance bugs. SQLancer [32] synthesizes two kinds of different SQL queries which are the same in semantics but evaluated by an optimizing and a non-optimizing version of DBMS for execution. If the two queries return different result sets, a logic bug in the optimizer of DBMS is detected. These generated-based fuzzers focus on detecting different particular logic bugs in DBMS instead of exploring more state-space of DBMS. They can only generate limited types of SQL statements that conform to the predefined generation model.

GRIFFIN is different from these works. It focuses on preserving both the syntactic and semantic correctness of SQL queries without SQL grammar. Unlike prebuilding the generation model based on the SQL grammar, it analyzes the metadata in DBMS and mutates new SQL queries with the metadata information. Consequently, with the metadata mutation, GRIFFIN can generate an amount of semantic correctness SQL queries based on the initial test case from DBMS, and explore various state spaces of the target DBMS.

Mutation-based DBMS fuzzing. Mutated-based fuzzing has been widely used in software testing and finds many bugs [4, 8, 9, 23–27, 39, 48–51]. Traditional mutation-based fuzzers (e.g., AFL) mutate new test cases by random mutation such as flipping the bytes of original inputs. Without prebuilding the generation model for specification, they can easily adapt to test library-feature DBMS like SQLite [5]. However, the mutated test cases contain a large amount of semantic errors and can not trigger complex behaviors of DBMS. To improve the performance of DBMS fuzzing, some works focus on improving the syntactic correctness of the mutated SQL queries. SQUIRREL [51] proposes to improve the syntactic correctness of the SQL test cases with the language validity based on the AFL. It designs an intermediate representation (IR) according to SQL grammar for type-based mutation to maintain syntactic correctness. RATEL [39] enhances the robustness of the SQL generation by combining the SQL dictionary and grammar-based mutation. It also improves the feedback precision with inter-binary coverage linkage and bijective block mapping to improve the performance of DBMS fuzzing. Nevertheless, the improvement of the syntactic and semantic correctness still strongly depends on prebuilding the SQL parser based on the SQL grammar.

Unlike these works, GRIFFIN focuses on improving semantic correctness without SQL grammar. Different from random mutation, GRIFFIN first identifies the metadata structure of the SQL queries and mutates new SQL test cases by analyzing the metadata of DBMS in real time. With the metadata-based mutation, GRIFFIN can improve the semantic correctness and trigger various behaviors of DBMS for detecting bugs.

Automatic grammar mining. Program input grammars, which formally describe the structures of program inputs, are already used in white-box fuzzing [15], black-box fuzzing [17] and grey-box fuzzing [3]. Automatic grammar mining can extract the input grammars of target programs by observing how they process inputs during execution [16, 18, 47]. With the source code of the target, REINAM [47] can generate initial inputs for the target with symbolic execution and iteratively infer input grammars generalized from these inputs. With a set of sample inputs given, AUTOGRAM [18] can extract the grammars by tracing the dynamic data flow of each input character, while Mimid [16] infers readable grammars from dynamic control flow.

GRIFFIN differs from these works. It does not extract the input grammars from the target DBMS. Instead, it constructs the metadata graph for each input to describe the semantic constraints specific to DBMS such as the dependencies between metadata and statements, which cannot be represented by input grammars directly.

Testing for DBMS schemas. To improve the correctness of DBMS, DBMS schema testing has been applied for decades, which usually generates data or queries on an existing schema for detecting vulnerabilities [1, 6, 22, 29]. With the target schema along with a SQL query given, QAGen [6] can generate data for the schema and the parameter values for the query to fulfill user-defined query constraints, and ADUSA [22] can not only generate the data but also provide the expect execution result of the given query. With the integrity constraints of schema considered, Mcminn et al. [29] formally defined the test coverage criteria for relational database schema integrity constraints, and DOMINO [1] introduced an automatic data generation method to effectively test the integrity constraints of target schemas.

Different from these works, GRIFFIN is not designed for testing the target schema of DBMS. It focuses on generating abundant semantic-correct SQL queries to test DBMS without the specified grammar. While schema testing approaches preserve the structure of schema and generate corresponding data or queries, the SQL statements generated by GRIFFIN may change the schema, such as `CREATE TABLE` statements.

8 CONCLUSION

In this paper, we propose GRIFFIN, a grammar-free mutation based DBMS fuzzer. It tracks the metadata of the statements in built-in SQL test cases as they are executed, and constructs the metadata graph to describe the dependencies between metadata and statements iteratively. Based on the graphs, it reshuffles statements and employs metadata-guided substitution to correct semantic errors. Compared with SQLancer, SQLsmith, and SQUIRREL, GRIFFIN covered 73.43%-274.70%, 80.47%-312.89%, 43.80%-199.11% more branches and found 27, 27, and 22 more bugs in 12 hours on SQLite, DuckDB, MariaDB, and PostgreSQL, respectively. It found 55 previously unknown bugs in total. Among them, 13 bugs are assigned with CVE IDs due to their security influences.

ACKNOWLEDGMENTS

This research is sponsored in part by NSFC Program (No. 62022046, 92167101, U1911401, 62021002, 62192730), National Key Research and Development Project (No. 2019YFB1706203, No2021QY0604).

REFERENCES

[1] Abdullah Alsharif, Gregory M Kapfhammer, and Phil McMinn. 2018. DOMINO: Fast and effective test data generation for relational database schemas. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 12–22.

[2] ANSI. 2022. ANSI Standard. https://www.ansi.org/. Accessed: September 3, 2022.

[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars.. In *NDSS*.

[4] 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)*.

[5] 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.

[6] Carsten Binnig, Donald Kossmann, Eric Lo, and M Tamer Özsu. 2007. QAGen: generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 341–352.

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based Greybox Fuzzing As Markov Chain. In *IEEE Transactions on Software Engineering*.

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

[10] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1580–1596.

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

[12] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[13] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. {GREYONE}: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2577–2594.

[14] Github. 2022. Squirrel GitHub. https://github.com/s3team/Squirrel. Accessed: September 3, 2022.

[15] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*. 206–215.

[16] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining input grammars from dynamic control flow. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 172–183.

[17] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. 445–458.

[18] Matthias Hoschele and Andreas Zeller. 2017. Mining input grammars with AUTOGRAM. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 31–34.

[19] Solid IT. 2022. DB-Engines Ranking. https://db-engines.com/en/ranking. Accessed: September 3, 2022.

[20] Solid IT. 2022. DB-Engines Ranking of Relational DBMS. https://db-engines.com/en/ranking/relational+dbms. Accessed: September 3, 2022.

[21] 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.

[22] Shadi Abdul Khalek, Bassem Elkarablieh, Yai O Laleye, and Sarfraz Khurshid. 2008. Query-aware test generation using a relational constraint solver. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 238–247.

[23] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. In *ACM International Conference on Automated Software Engineering (ASE)*.

[24] 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.

[25] 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).

[26] LibFuzzer 2022. LibFuzzer. https://www.llvm.org/docs/LibFuzzer.html. Accessed: September 3, 2022.

[27] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security Symposium*.

[28] Chirag Manghnani. 2022. Metadata in DBMS – Overview and Types. https://www.thecrazyprogrammer.com/2019/12/metadata-in-dbms.html. Accessed: September 3, 2022.

[29] Phil Mcminn, Chris J Wright, and Gregory M Kapfhammer. 2015. The effectiveness of test coverage criteria for relational database schema integrity constraints. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 1 (2015), 1–49.

[30] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990).

[31] NVD. 2022. NVD Home. https://nvd.nist.gov/. Accessed: September 3, 2022.

[32] 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.

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

[34] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. *USENIX Annual Technical Conference (ATC)*.

[35] Donald R Slutz. 1998. Massive stochastic testing of SQL. In *VLDB*, Vol. 98. Citeseer, 618–622.

[36] SQLite. 2022. The Advantages Of Flexible Typing. https://www.sqlite.org/flextypegood.html. Accessed: September 3, 2022.

[37] The Clang Team. 2022. Source-based Code Coverage. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html. Accessed: September 3, 2022.

[38] Dmitry Vyukov. 2015. google/syzkaller: syzkaller is an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller. Accessed: September 3, 2022.

[39] 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.

[40] Wikipedia. 2022. Comparison of relational database management systems. https://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems. Accessed: September 3, 2022.

[41] Wikipedia. 2022. Database. https://en.wikipedia.org/wiki/Database. Accessed: September 3, 2022.

[42] Wikipedia. 2022. Database catalog. https://en.wikipedia.org/wiki/Database_catalog. Accessed: September 3, 2022.

[43] Wikipedia. 2022. databases. https://en.wikipedia.org/wiki/Database. Accessed: September 3, 2022.

[44] Wikipedia. 2022. Metadata in database management. https://en.wikipedia.org/wiki/Metadata#Database_management. Accessed: September 3, 2022.

[45] Wikipedia. 2022. Relational Database. https://en.wikipedia.org/wiki/Relational_database. Accessed: September 3, 2022.

[46] Wikipedia. 2022. SQL. https://en.wikipedia.org/wiki/SQL. Accessed: September 3, 2022.

[47] Zhengkai Wu, Evan Johnson, Wei Yang, Osbert Bastani, Dawn Song, Jian Peng, and Tao Xie. 2019. REINAM: reinforcement learning for input-grammar inference. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 488–498.

[48] 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 *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 251–262. https://doi.org/10.1145/3533767.3534364

[49] 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*.

[50] Michal Zalewski. 2017. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: September 3, 2022.

[51] 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.*