applied sciences

MDPI

Article

Squill: Testing DBMS with Correctness Feedback and Accurate Instantiation

Shihao Wen [1], Peng Jia [1,*], Pin Yang [1] and Chi Hu [2]

[1] School of Cyber Science and Engineering, Sichuan University, Chengdu 610207, China; wenshihao@stu.scu.edu.cn (S.W.); yangpin@scu.edu.cn (P.Y.)
[2] China Academy of Engineering Physics, Mianyang 621900, China; huchi16@nudt.edu.cn
* Correspondence: pengjia@scu.edu.cn

Abstract: Database Management Systems (DBMSs) are the core of management information systems. Thus, detecting security bugs or vulnerabilities of DBMSs is an essential task. In recent years, grey-box fuzzing has been adopted to detect DBMS bugs for its high effectiveness. However, the seed scheduling strategy of existing fuzzing techniques does not consider the seeds' correctness, which is inefficient in finding vulnerabilities in DBMSs. Moreover, current tools cannot correctly generate SQL statements with nested structures, which limits their effectiveness. This paper proposes a fuzzing solution named Squill to address these challenges. First, we propose correctness-guided mutation to utilize the correctness of seeds as feedback to guide fuzzing. Second, Squill embeds semantics-aware instantiation to correctly fill semantics to SQL statements with nested structures by collecting the context information of AST nodes. We implemented Squill based on Squirrel and evaluated it on three popular DBMSs: MySQL, MariaDB, and OceanBase. In our experiment, Squill explored 29% more paths and found 3.4× more bugs than the existing tool. In total, Squill detected 30 bugs in MySQL, 27 in MariaDB, and 6 in OceanBase. Overall, 19 of the bugs are fixed with 9 CVEs assigned. The results show that Squill outperforms the previous fuzzer in terms of both code coverage and bug discovery.

Keywords: coverage-based grey-box fuzzing; database testing; vulnerability

Citation: Wen, S.; Jia, P.; Yang, P.; Hu, C. Squill: Testing DBMS with Correctness Feedback and Accurate Instantiation. *Appl. Sci.* **2023**, *13*, 2519. https://doi.org/10.3390/app13042519

Academic Editor: Luis Javier García Villalba

Received: 22 November 2022
Revised: 10 February 2023
Accepted: 13 February 2023
Published: 15 February 2023

1. Introduction

Database management systems (DBMSs) are widely used worldwide as the core of modern information systems. Like other complicated computer applications, the security and reliability of DBMSs face severe challenges. Malicious attacks on DBMSs, such as remote code execution or denial of service, will seriously harm the information system. Therefore, it is of great significance to efficiently detect DBMS vulnerabilities to improve their robustness and the security of the information system built on them.

Black-box fuzzing, or generation-based fuzzing, has been extensively used in finding DBMS bugs, such as SQLsmith [1] and SQLancer [2–4]. Security researchers have found a considerable number of bugs using this technique. A black-box fuzzer treats the program as a black box and is unaware of internal program structure [5]. It randomly generates a large number of SQL statements and executes them in the DBMS. The current input is saved for subsequent analysis when unexpected behavior occurs, such as a crash. The disadvantage of black-box fuzzing has been thoroughly discussed by the academic circle, which is inefficiency. Since the generation of SQL statements is entirely random, considering the complexity of the DBMS, most of the inputs generated by the black-box fuzzer will be difficult to trigger the deep program logic, in which bugs often hide. Despite inefficiency, this technique still has a wide range of uses. Since black-box fuzzing does not require the source code of the DBMS, it can test some commercial DBMSs that are not open source.

Researchers have studied grey-box fuzzing actively in recent years. The main difference between grey-box fuzzing and black-box fuzzing is that the former leverages

instrumentation to glean information about the program [5], such as code coverage. With an initial seed queue, the grey-box fuzzer performs a series of mutations on seeds to generate new inputs and saves the inputs that trigger a new state (or crash) of the program for future mutation. Therefore, compared with black-box fuzzer, grey-box fuzzer can explore the deep states of the program gradually. The well-known AFL [6] collects the code coverage of the program during fuzzing by instrumentation, and DBMS vendors have applied it to DBMS testing. For example, SQLite used AFL as a standard part of the testing strategy until it was superseded by better fuzzers [7]. However, since fuzzer [8–10], like AFL, was not initially designed for DBMS fuzzing, the SQL statements generated by AFL often have syntactic or semantic errors, making it hard to trigger the deep logic of DBMSs (such as the optimizer). Squirrel [11], a recent work focusing on DBMS fuzzing, has solved this problem to some extent, making it the state-of-art grey-box DBMS fuzzer. It introduces the structure-aware mutator for SQL statements into AFL. After mutation, it fills inputs with new semantics to improve the syntactic and semantic correctness.

In recent years, many new solutions have been proposed for grey-box fuzzer to improve fuzzing efficiency. An important one is improving the seed scheduling strategy. However, less attention has been paid to the seed scheduling strategy in the DBMS fuzzing area. In DBMS fuzzing, different seeds have different correctness, and seeds with different correctness contribute differently to fuzzing. Hence, scheduling seeds by speed and size, the seed scheduling strategy in the existing grey-box DBMS fuzzer, is inefficient. Another challenge in grey-box DBMS fuzzing is the semantics filling of SQL statements. In order to make the SQL statement generated by mutation pass the semantic check of DBMSs, Squirrel proposes a method called Semantics-Guided Instantiation to fill the SQL skeleton with concrete semantics. However, the instantiation method of Squirrel does not perform well on SQL statements with nested structures due to design issues. A significant reason is that Squirrel cannot distinguish between nodes with the same type but at different levels. The problem of instantiation makes Squirrel hard to generate complex SQL statements, limiting its effectiveness in finding DBMS bugs.

In this paper, we implement a grey-box fuzzer, Squill, to address the challenges faced in current DBMS fuzzing. As the particularity of DBMS fuzzing scenarios, we propose correctness-guided mutation, which utilizes the correctness of SQL statements as feedback to guide fuzzing. We design two heuristic methods to improve the fuzzing efficiency by collecting the correctness (valid, syntax-error, semantics-error) of each seed. First, we prioritize mutating valid seeds because of their effectiveness in generating new paths and crashes. Second, we give some seeds with syntactic or semantic errors more opportunities to participate in mutation as material to activate interesting SQL structures in them more rapidly. In addition, we propose semantics-aware instantiation, which has the ability to guarantee the semantic correctness of the inputs with nested structures. We design a new instantiation stage in which we fill the nodes with semantics according to the predetermined constraints. During instantiation, we traverse each node of the AST in turn and parse according to the node type. While traversing, we collect the context information of each node so that we can distinguish nodes of the same type but at different levels and assign different dependencies to them. For example, with the context information of a node, we can distinguish whether it is at the beginning of a SELECT statement or a subquery in FROM clause and treat it differently.

We implemented Squill based on Squirrel. To understand the effectiveness of Squill, we evaluated it on three popular databases: MySQL [12], MariaDB [13], and OceanBase [14]. Squill successfully found 63 memory error issues, including 30 bugs in MySQL, 27 bugs in MariaDB, and 6 bugs in OceanBase. We have reported all of our findings to the developers of the appropriate DBMS. At the time of paper writing, 19 bugs have been fixed, and 9 CVE numbers have been assigned due to the danger of these vulnerabilities. Our evaluation shows that correctness-guided mutation helps to improve the efficiency of fuzzers in path exploration and bug finding. We also compare our work with the current state-of-the-art tool, Squirrel. After 24 h of testing, Squill found 15, 17, and 2 bugs in each of the

three DBMSs, while Squirrel found only 3, 7, and 0 bugs. Furthermore, results show that semantics-aware instantiation outperforms the instantiation of Squirrel in the correct semantic filling of complex SQL statements.

In this paper, we first introduce Squirrel's mutation and instantiation method. Then we illustrate the necessity of scheduling seeds according to correctness through experiments and illustrate the drawbacks of Squirrel's instantiation method with examples. In addition, we introduce our solutions Squill to these two problems, including correctness-guided mutation and semantics-aware instantiation. Eventually, we prove the effectiveness of Squill through experiments.

In conclusion, this paper makes the following contributions:

- We investigated the drawbacks of the current seed scheduling strategy and the problem of Squirrel's instantiation method. We conclude that seeds should be scheduled based on correctness, and a new instantiation method that can correctly generate semantics for SQL statements with nested structures is demanded.
- We propose correctness-guided mutation, which utilizes the correctness of seed execution as feedback to guide fuzzing and improve efficiency. Moreover, we propose semantics-aware instantiation to address the challenge of correct semantics generation for SQL statements with nested structures. We implement Squill, a coverage-guided DBMS fuzzer that applies the two solutions above.
- We evaluated Squill on several real-world DBMSs and found 63 bugs.The results show that Squill outperforms the previous fuzzer in terms of both code coverage and bug discovery. We have released the source code of Squill at https://github.com/imbawenzi/Squill (accessed on 22 November 2022).

2. Background

Our proposed solution, Squill, is built on the state-of-the-art DBMS fuzzer, Squirrel. In this section, we first present an overview of Squirrel. We also introduce the challenges that current grey-box DBMS fuzzing faces and illustrate the motivation of Squill.

2.1. Overview of Squirrel

Squirrel is a recent work that aims to detect memory errors in DBMSs. Based on AFL, Squirrel modifies the mutation component so that the fuzzer can guarantee the syntactic correctness of SQL statements when mutating. As the input may be a combination of multiple parts from different SQL statements, there is a considerable probability for its semantics to be wrong. After mutation, Squirrel fills the skeleton of the SQL query with concrete operands (such as table name) through query instantiation to improve the semantic correctness.

A fuzzing loop of Squirrel starts with an empty database and inputs a set of SQL statements into DBMS, which generally include CREATE, INSERT, UPDATE, and SELECT statements. After Squirrel completes one execution, it will empty the database. Squirrel will add the input to the seed queue when it triggers new code coverage. So that Squirrel can mutate based on previous seeds, triggering the deep logic of DBMSs, compared with black-box DMBS fuzzer.

2.1.1. Mutation of Squirrel

Squirrel implements a SQL parser that converts SQL statements into AST. The mutation of the seeds (SQL statements) is based on the AST. Each node has an associated type (or grammar type), such as `SelectStmt` for the root node of a SELECT statement. Squirrel proposes three new mutation operators, including insertion, deletion, and replacement of an AST node. There is an AST subtree library in Squirrel, which we call the mutation material library. Squirrel will convert the original input and new seeds into AST and add all subtrees of these AST to the mutated material library. When performing a replace or insert mutation, Squirrel randomly selects a subtree whose root node has the same type as the target node from the mutation material library to mutate. In this way, Squirrel can

maintain SQL statements in a structural manner and guarantee syntactic correctness during mutation. In the AST parser, Squirrel additionally assigns a refined data type, used in the instantiation, to nodes with semantics, such as table name.

2.1.2. Instantiation of Squirrel

The new SQL statement generated by mutation is a syntax-correct skeleton with semantics stripped. Squirrel fills it with concrete values in the process called instantiation. For data definition nodes, such as table name and column name in the CREATE statement, Squirrel directly generates concrete data to fill the node and record it. For other nodes, Squirrel will first construct the dependency graph of nodes according to the preset dependency rules of different refined data types. For the node in the graph with more than one parent, Squirrel randomly picks one to establish the edge. After that, the dependency graph is filled from top to bottom to complete the semantics filling of each node.

Figure 1 is an instantiation example of a SELECT statement in Squirrel, where `x*` is the placeholder for the semantics to be filled, and `v*` represents the semantics after filling. For the CREATE statement, we assume that the semantics has been assigned. The SELECT statement has two types of nodes that need to be instantiated, the node whose refined data type is `kDataColumnName` and the node whose refined data type is `KDataTableName`. Squirrel specifies the dependencies between these two refined data types. That is, the column name depends on the table name. Since the column name `x1` can come from both table `x3` and table `x4`, and `x2` is the same, the dependency graph in the figure can be constructed. Then Squirrel randomly selects a parent for each node, assuming that `x1`, `x2` depend on `x3`. Finally, the dependency graph is filled from top to bottom. For the table name, Squirrel randomly selects one from the existing tables(`v1` and `v5`). For the column name, Squirrel randomly selects a column name from the table it depends on. At this point, the SELECT statement is filled with semantics.

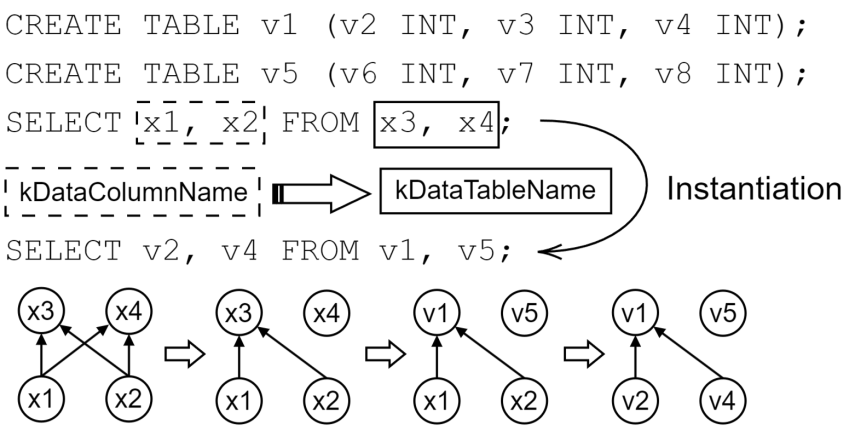


Figure 1. An instantiation example of Squirrel.

2.2. Motivation

2.2.1. Correctness Feedback

In grey-box fuzzing, fuzzers usually collect some information to guide fuzzing. For example, AFL collects seeds' size and execution speed and prioritizes mutating the smaller and faster seeds. Some studies [15–17] have shown that information, such as the rareness of branches, the number of memory reads or writes, and the number of branches that seed changed can guide fuzzer to perform better. In DBMS fuzzing, there is a noticeable difference in the correctness of the seeds. For example, Listing 1 shows some SQL statements with different correctness. An intuitive assumption is that seeds with different correctness contribute differently to fuzzing.

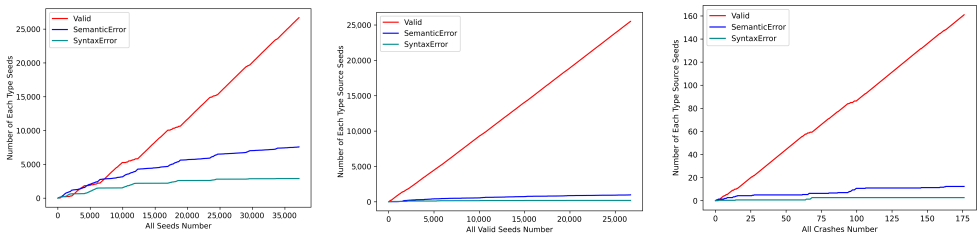
Listing 1. SQL statements with different correctness.

```
--- Valid
SELECT row_number() OVER w, v1 FROM v2 WINDOW w AS (PARTITION BY
v3 ORDER BY v4);

--- Semantics-error
SELECT row_number() OVER w, v1 FROM v2;
--- ERROR: Window name 'w' is not defined.

--- Syntax-error
SALECT row_number() OVER w, v1 FROM v2 WINDOW w AS (PARTITION BY
v3 ORDER BY v4);
--- ERROR: MySQL server version for the right syntax to use
near 'SALECT\ldots'
```

To verify our hypothesis, we conducted experiments on Squirrel to evaluate the contribution of seeds with different correctness. The result is demonstrated in Figure 2. According to the correctness of seeds, we divided the seeds into three types: valid (or semantics-correct), syntax-error, and semantics-error. We counted the seeds number of each type in a DBMS fuzzing process, as shown in Figure 2a. The abscissa indicates the total number of seeds in the process of fuzzing, and the ordinate indicates the number of different correctness seeds during the period. We found that most of the seed increments come from valid seeds. In other words, most of the paths explored by fuzzing were the program logic of DBMS after the syntactic and semantic check. It is because only inputs that are syntactic and semantic correct can proceed to the following phases, such as optimization and execution, triggering new code coverage.



(**a**) Correctness of seeds (**b**) Source seed type of valid seeds (**c**) Source seed type of crashes

Figure 2. Contributions of seeds with different correctness in a DBMS fuzzing process.

We also counted the correctness of the valid seed's source seed in this fuzzing, as shown in Figure 2b. The abscissa indicates the total number of valid seeds in the fuzzing process, and the ordinate indicates the number of different correctness valid seed's source seeds during the period. A seed's source seed means that the seed was generated by the mutation based on its source seed. It can be seen that the majority of valid seeds are mutated from valid seeds. Considering the proportion of valid seeds in all seeds, it shows that valid seeds have a greater probability of generating valid seeds than seeds with syntactic and semantic errors. It is because if seeds with syntactic and semantic errors want to generate valid seeds, they need to mutate the wrong structures into correct ones, which is more difficult.

Moreover, we counted the correctness of the crash source seed, as shown in Figure 2c. The abscissa indicates the total number of crashes in the fuzzing process, and the ordinate indicates the number of different correctness crash source seeds during the period. The result shows that valid seeds are more likely to generate crash inputs than seeds with syntactic and semantic errors, as crashes often hide in the deep logic of the DBMS. To cause

a crash, the input needs to pass the DBMS's syntactic and semantic check so that the DBMS can execute it. Therefore, the input that causes a crash is often valid.

Motivation. According to the analysis above, we can conclude that seeds with different correctness have different contributions to fuzzing. Hence, the seed schedulers in existing fuzzers, which schedules seeds by speed and size, are not efficient. Ideally, valid seeds should be mutated prior to invalid seeds because of their effectiveness in generating new paths and crashes. Therefore, a better seed scheduling strategy is demanded.

2.2.2. Limitation of Squirrel's Instantiation

The instantiation method of Squirrel works well on simple SQL statements. However, when faced with complex SQL statements, this method shows its limitation. In this paper, we define complex SQL statements as long SQL statements with nested structures, such as subqueries. When Squirrel translates SQL statements into AST, it will initialize the node containing semantics with a corresponding refined data type. It means that when recursive parsing, such as a subquery, nodes at different levels will be assigned with the same refined data type, for Squirrel parses them with the same grammar. However, there may be dependencies between nodes at different levels. Therefore, an error occurs when using the refined data type to determine the dependencies between nodes in nested structures. From another point of view, this problem is caused by Squirrel defining the dependency between nodes in the syntax analysis stage, in which the information about SQL statements is not enough to construct a complicated dependency.

Suppose there are SQL statements shown in Figure 3, which are similar to that in Figure 1, except the SELECT statement has a subquery. For descriptive convenience, the subquery does not have an alias here. Repeating the instantiation described in Section 2.1.2, the refined data type of `x1`, `x2`, `x3`, and `x4` is kDataColumnName. Hence, they all depend on the table name nodes x5 or x6 in the same statement. Assuming that `x1` depends on `x5`, and `x2`, `x3`, `x4` depend on `x6`, the dependency graph in the figure can be constructed and filled. We can see that `x1` is filled with an invalid column name v2 that does not exist in the subquery result because `x1` comes from table v1 while `x3`, `x4` come from table v5. Even if there is only one subquery, Squirrel still has a high probability of filling in the wrong semantics, let alone in the case of multiple subqueries.

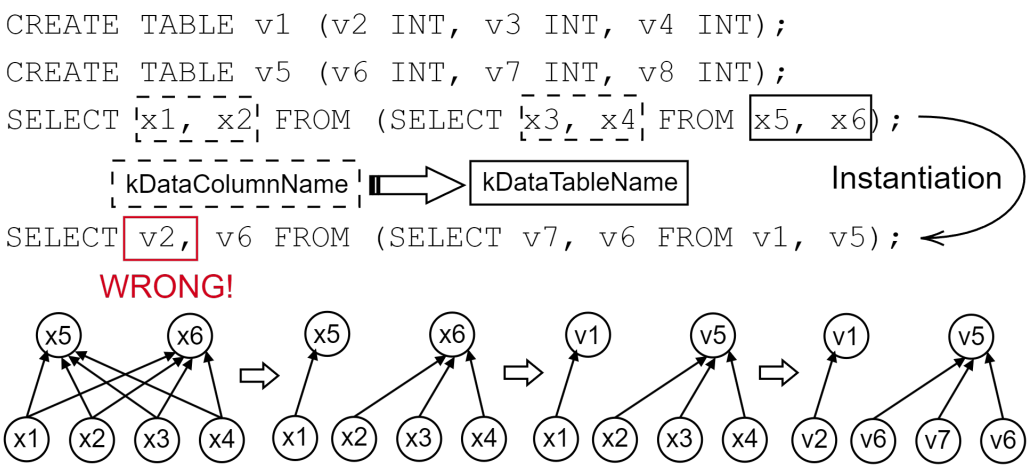


Figure 3. Squirrel's instantiation of SQL statements with a subquery.

In fact, `x1` and `x2` should depend on `x3` and `x4`, as `x1` and `x2` should come from the result of subquery in the FROM clause. Squirrel cannot do that by defining more data relation rules because it initializes both column name nodes in the subquery and the main SELECT statement with the same refined data type. During instantiation, it appears to Squirrel that these nodes are all the same. In this example, Squirrel has no way of distinguishing between `x1` and `x3` and has difficulty establishing a dependency that makes

`x1` depend on `x3` and `x4`. Because of the above problem, in practice, Squirrel will discard the input with multiple subqueries for their low semantics-correct rate after instantiation.

Motivation. A new instantiation method that can correctly generate semantics for SQL statements with nested structures is demanded. In order to achieve this goal, the new instantiation method should not rely on the refined data type defined in the AST translator to construct the dependency graph.

3. Design of Squill

We propose two practical solutions to address the above challenges. First, we provide correctness-guided mutation, which contains two heuristic methods, utilizing the correctness of seeds as feedback to improve the efficiency of fuzzing (Section 3.1). Second, we introduce semantics-aware instantiation (Section 3.2). During instantiation, we collect the context information of nodes. So we can know the level of the node according to the context information and build dependencies across levels when traversing to a nested structure.

Figure 4 shows an overview of Squill, where the white components are the original Squirrel, and our design is marked in grey. Squill follows the general flow of grey-box DBMS fuzzing, which mainly includes mutation, instantiation, and fuzzing. First, Squill selects the next seed to mutate from the seed queue. Squill will preferentially select the seeds with syntactic and semantic correctness. Then, the seed is translated into AST. The mutator randomly performs replacement, insertion, and deletion mutations on the AST. Squill adds an interesting material library to participate in the mutation. During the instantiation phase, new inputs generated by mutation will be filled with semantics to maintain semantic correctness. We design a new instantiator to address the challenge of correct semantics generation for SQL statements with nested structures. In the end, Squill will take these test cases as input to the DBMS, detect whether the DBMS has crashed, and add the input that triggers new code coverage to the seed queue.

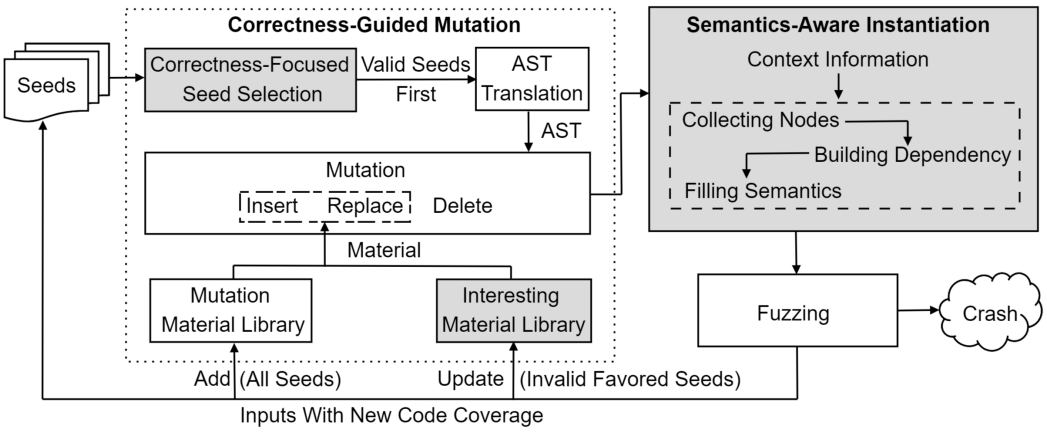


Figure 4. Overview of Squill.

3.1. Correctness-Guided Mutation

Since seeds with different correctness contribute differently to fuzzing, the fuzzer should not treat them equally. In this section, we propose two correctness-guided heuristic methods to improve the efficiency of fuzzing in path exploration and bug finding.

3.1.1. Correctness-Focused Seed Selection

In DBMS fuzzing, most of the seed increments come from valid seeds. Valid seeds can trigger deeper logic of DBMS than those with syntactic and semantic errors, exploring more paths. In addition, valid seeds have a higher probability of generating valid seeds, producing more crashes. Based on the conclusion above, we propose a correctness-focused seed selection strategy. We mutate valid seeds first, then seeds with semantics-error, and finally seeds with syntax-error. Because mutating valid seeds is more likely to generate

valid seeds, leading to more path exploration and bug finding. The process of seed selection is shown in Algorithm 1.

Algorithm 1 Correctness-focused seed selection.

```
1:  // Run when fuzzer generates new seed
2:  function UPDATE_BITMAP_SCORE(new_seed)
3:      for i from 0 to MAP_SIZE do
4:          // trace_bits is the bitmap of current seed
5:          if trace_bits[i] is not 0 then
6:              if top_rated[i] is not 0 then
7:                  if new_seed has better correctness than top_rated[i] then
8:                      top_rated[i] = new_seed;
9:                  end if
10:                 // Compare speed and size with top_rated[i]
11:                 // when they have same correctness
12:                 if new_seed has the same correctness as top_rated[i] then
13:                     if new_seed is faster and smaller than top_rated[i] then
14:                         top_rated[i] = new_seed;
15:                     end if
16:                 end if
17:             else
18:                 top_rated[i] = new_seed;
19:             end if
20:         end if
21:     end for
22: end function
```

We implement correctness-focused seed selection based on AFL's original seed selection mechanism. AFL updates `top_rated` whenever it finds a new seed `top_rated` is an array that has the same size as the bitmap, where each value records the seed with the highest score on the corresponding edge in the bitmap. The faster, smaller seed will have a higher score. Then, AFL uses a greedy algorithm to select a minimum subset of seeds that contain all edges in the bitmap from seeds recorded in `top_rated`. Seeds in this subset are marked as `favored`. The `favored` seeds have a higher probability of mutating. AFL uses this mechanism to reduce the seed queue and improve the efficiency of fuzzing.

In the original seed selection mechanism, AFL prioritizes fuzzing faster and smaller seeds. This mechanism tends to select more syntax-error or semantics-error seeds to participate in fuzzing. The seeds with syntactic or semantic errors usually have a faster execution speed, as they terminate at the syntactic and semantic check phase of DBMS. The subsequent phases of DBMS, such as the optimization and storage phase, are often time-consuming. In our method, we preferentially update seeds with better correctness into `top_rated`, as shown in Algorithm 1 Line 7–9. We define that valid seeds are better than semantics-error seeds, which are better than syntax-error seeds. We first compare the correctness of seeds and then consider their execution speed and size only if they have the same correctness (Line 12–16). It ensures that valid seeds are mutated preferentially.

3.1.2. Mutation with Interesting Material Library

Valid seeds usually trigger deep program logic. In contrast, seeds with syntactic or semantic errors (in other words, invalid) often terminate at the early phase of DBMS, such as the syntactic and semantic check. It means that the optimizer and executor of DBMS do not actually process these invalid SQL statements. So some SQL structures in the syntax-error or semantics-error seed are unactivated, as subsequent phases of DBMS do not actually process them. That is, although some SQL structures can trigger new DBMS logic, they are not actually executed because they are in an invalid seed. We call these SQL structures interesting structures here. For example, the query in Line 4 of Listing 1

is an invalid input, which uses a not existing window `w`. The valid one is shown in Line 2. The function `row_number()` in Line 4 may be an interesting structure. It is not actually executed since it is in a semantics-error SQL statement that does not pass the semantic check of DBMS.

Therefore, we designed a method to filter out these interesting structures and activate them, as shown in Algorithm 2. We maintain an interesting material library, which contains subtrees of all current `favored` and invalid seeds (Line 15–17). When the fuzzer needs a material (subtree) from the mutation material library to participate in mutation, it has a certain probability of obtaining the material from the interesting material library (Line 21–30). The variable `probability` in Line 25 is an input parameter, which is set to 5 by default.

Algorithm 2 Mutation with interesting material library.

```
1:  // Run when top_rated changed
2:  function CULL_QUEUE(void)
3:      temp_bitmap[MAP_SIZE] = 0;
4:      set_empty(interesting_library);
5:      for i from 0 to MAP_SIZE do
6:          if top_rated[i] is not 0 and temp_bitmap[i] is 0 then
7:              // Favored means mutating first
8:              top_rated[i] is favored;
9:              // Record the bitmap of top_rated[i] to temp_bitmap
10:             for j from 0 to MAP_SIZE do
11:                 if top_rated[i].bitmap[j] is not 0 then
12:                     temp_bitmap[j] = 1;
13:                 end if
14:             end for
15:             if top_rated[i] is invalid then
16:                 add_into_interesting_library(top_rated[i]);
17:             end if
18:         end if
19:     end for
20: end function
21: // Run when insertion or replacement
22: function GET_IR_FROM_LIBRARY(type)
23:     // Get a random number from 1 to 100
24:     rand_int = get_rand_int(100);
25:     if rand_int<probability then
26:         get_from_interestring_library(type);
27:     else
28:         get_from_all_library(type);
29:     end if
30: end function
```

We utilize the `favored` mechanism in AFL to select seeds that may contain interesting structures. After correctness-focused seed selection, the `favored` and invalid seed must trigger the program state (edge) that valid seeds have not triggered. For example, some SQL structures in these seeds might trigger a unique logic of the DBMS parser. When these SQL structures are actually executed, it is likely to bring path exploration or bug finding in the optimizer or executor of DBMS. Since it is difficult to generate valid seeds from the mutation of seeds with syntactic and semantic errors, we give the mutation material (subtrees) of these seeds more opportunities to participate in mutation, making interesting structures executed in valid seeds after insertion or replacement.

3.2. Semantics-Aware Instantiation

We design an instantiation algorithm to address the challenge of correct semantics generation for SQL statements with nested structures. In instantiation, while traversing AST nodes in the order of SQL statements, we parse nodes according to the node's type and context information (such as the type of parent and adjacent nodes). In this way, we can distinguish nodes of the same type but at different levels, as their context information is different. For example, the type of the parent node of a main SELECT statement and a subquery is distinct. With this information, we can construct a series of detailed constraints on nodes based on prior knowledge (the relationship between semantics in SQL statements) and then fill them with semantics correctly according to these constraints. For example, for a table name node in a CREATE statement, we can know whether it comes from a CREATE TABLE statement or a CREATE TRIGGER statement according to the context information when parsing it. For the former, we will fill it with a newly generated unique table name. For the latter, we will randomly assign a table name to it from the currently existing table name (created in the previous SQL statement).

We divide semantics into simple and complicated semantics, depending on the complexity of constraints. When traversing to a node, if we can instantly assign semantics to it without error, we call the semantics that the node has as simple semantics. The dependency constraints of nodes with such semantics are relatively simple, usually across statements. For example, the table name dropped in the DROP statement is from tables created in the previous CREATE statements. When filling a DROP statement with semantics, the previous CREATE statement has been traversed and instantiated. At this point, the existing table names are determined, which can be instantly assigned to the table name node in the DROP statement. When traversing to a node, if we cannot instantly assign semantics to it but need to wait until the entire SQL statement is parsed and fill it with consideration of the semantics of other nodes, we call the semantics that the node has as complicated semantics. For example, the column name in a SELECT clause depends on one of the tables in the FROM clause, which means that the former should be a column of the latter, and we need to instantiate the latter before the former.

3.2.1. Instantiation of Simple Semantics

Simple semantics mainly exist in CREATE, DROP, and ALTER statements, as well as nodes that do not have dependencies, such as function names. In instantiation, Squill maintains a data structure called the information table that stores the current database information, which mainly contains the table name and column name of the created tables. This information table also stores information of indexes, views, and triggers. When instantiating CREATE, DROP, and ALTER statements, we perform creating, deleting, and modifying operations in the information table correspondingly, such as in real DBMS. For the CREATE statement, we generate and assign a unique table name and column name (or index name) to the corresponding node. We record this information in the information table described above. For the ALTER statement, we will randomly choose a table name from the currently existing table name. Whether it is to modify, delete, or add a column name, Squill randomly assigns a column name from the table chosen above and modifies the corresponding information in the information table. Similarly to the DROP statement, we randomly assign a table name and delete it in the information table. For nodes without dependency, including function, integer, and floating point number, Squill will randomly assign a predefined value to them. In addition, the alias node will be assigned a unique name when traversed.

3.2.2. Instantiation of Complicated Semantics

Instantiation of complicated semantics is performed in SQL statements with column-table dependency, including SELECT, INSERT, and UPDATE statements. It is performed within one SQL statement, as the dependency between column and table is not across statements. For example, there are two independent SELECT statements. The column

name in the former and the table name in the latter are irrelevant. The instantiation of complicated semantics includes three stages, collecting nodes, building dependency, and filling semantics, as shown in Algorithm 3. Note that since an input of Squill is composed of multiple SQL statements, the instantiation of complicated semantics is often performed multiple times for an input.

Algorithm 3 Instantiation of complicated semantics.

```
1:  ColumnList ← Column[];
2:  VirtualTableList ← VirtualTable[];
3:  DependencyList ← map(Column, VirtualTable);
4:  function INSTANTIATION(root)
5:      // A. Collecting Nodes
6:      for each ColumnNode in SelectTarget, Where... do
7:          // Convert node to Column
8:          ColumnList.add(ColumnParser(ColumnNode));
9:      end for
10:     for each TableNode in From, InsertTable, UpdateTable do
11:         // Convert node to VirtualTable
12:         VirtualTableList.add(TableParser(TableNode));
13:     end for
14:     for each SubQueryNode do
15:         // Process subquery recursively
16:         Instantiation(SubQueryNode);
17:         if SubQueryNode is in From then
18:             VirtualTableList.add(SubQueryParser(SubQueryNode));
19:         end if
20:     end for
21:
22:     // B. Building Dependency
23:     for each Column in ColumnList do
24:         DependencyList[Column]=RandChoose(VirtualTableList);
25:     end for
26:
27:     // C. Filling Semantics
28:     for each (Column, VirtualTable) in DependencyList do
29:         if VirtualTable is not filled then
30:             FillVirtualTable(VirtualTable);
31:         end if
32:         FillColumn(Column);
33:     end for
34:     FillVirtualTableNotInDependencyList();
35: end function
```

A. Collecting Nodes. While traversing AST, we collect the node with complicated semantics based on the type and context information of the current node and store it in the corresponding data structure (Line 5–20). For the SELECT statement, we collect column name nodes in select target, function parameter, WHERE, GROUP BY, ORDER BY, and WINDOW clauses, storing them in the data structure called `Column` (Line 6–9). `Column` stores not only the column name node but also the alias node and the table name node corresponding to the column name node, if they exist. With the help of context information, we can describe a column abstractly. Similarly, we collect the column name node in the insert and update the target clause of the INSERT and UPDATE statement.

For the table name, since we want to treat the result of the subquery as a table, we define a data structure called `VirtualTable` to represent a table. `VirtualTable` includes a table name node, an array of `Column`, and the alias node of the table, which can describe a table or the result of a subquery. For the SELECT statement, we collect the table name

node in the FROM clause. For INSERT and UPDATE statements, we collect their target table name nodes (Line 10–13). For subqueries, we process them recursively, instantiating them from inner to outer (Line 14–20). For the subquery in FROM clause of the SELECT statement, we treat the result of it as a table in the subsequent dependency construction. For the subquery in other clauses, such as in the WHERE clause, we instantiate it like a SELECT statement since there is no external dependency within it.

Figure 5 shows the data structure that contains nodes in the main `SELECT` statement, where `Column_x4` and `Column_x5` are the data structure `Column` which contains nodes `x4` and `x5`. `VirtualTable` describes a table (`x10`) or the result of a subquery (`s1`) by filling different fields (`TableNameNode` or `ColumnList`). When parsing, we recursively processed subqueries, which means that the subquery and the main `SELECT` statement will be parsed with the same function, and the subquery will be instantiated before the outer query. Therefore, the corresponding data structure (such as `Column_x4` and `Column_x5`) is created while parsing the subquery.

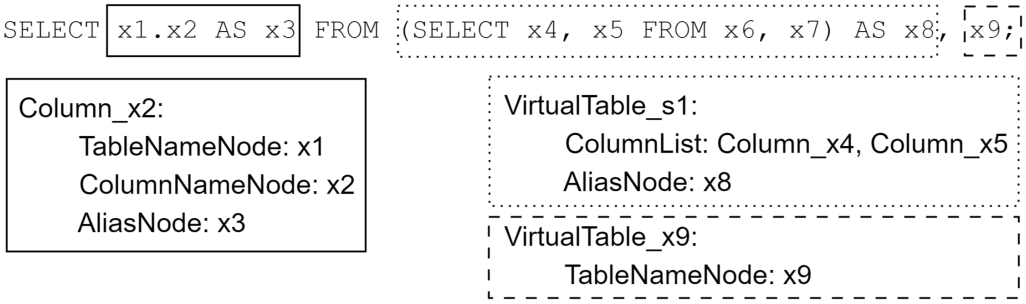


Figure 5. An example of collecting nodes.

B. Building Dependency. After creating the corresponding data structure, we construct the dependency between `Column` and `VirtualTable` (Line 22–25). Obviously, after processing, the dependency is very clear, which is that all `Column` depends on the `VirtualTable` in `FROM` clause. We randomly select a `VirtualTable` for each `Column` to depend on and record the dependency.

C. Filling Semantics. For each `Column-VirtualTable` dependency recorded, we fill nodes in it with semantics (Line 27–34). We fill `VirtualTable` first, and then the `Column` which depends on it. If the `VirtualTable` describes a table, we randomly assign the table name node in it with a table name from currently existing table names. The column name node in the `Column` which depends on the `VirtualTable` will be assigned a random column name from the table selected. The table name node in the `Column` will be filled with the same table name in `VirtualTable`, if it exists. If the `VirtualTable` describes the result of a subquery, we do not need to instantiate nodes of the `Column` in it, as they have been filled with semantics in the instantiation of the subquery (Line 16). The column name node in the `Column` will be filled with a column name in a random one of the `Column` in the `VirtualTable`. The table name node in the `Column` will be filled with the alias of the `VirtualTable`. At last, we fill semantics of the `VirtualTable` that is not depended on by any `Column` (Line 34).

Example. Figure 6 is an example of the instantiation of complicated semantics in which the SQL statement is the same as that in Section 2.2.2. Suppose that the first two CREATE statements have been instantiated, where the table names and column names have been generated, filled in nodes, and recorded in the information table. For the SELECT statement, there are two instantiation processes, one is the instantiation of the subquery, and the other is the instantiation of the main SELECT statement. Since the process is similar, here we focus on the instantiation of the main SELECT statement. Assume that the instantiation result of the subquery is as in `step1`. This SELECT statement contains two `Column` and a `VirtualTable`, where the `VirtualTable` describes the result of a subquery. Obviously, both `Column_x1` and `Column_x2` depend on `VirtualTable_s1`. We can construct

a dependency graph as shown in the figure. Compared with Squirrel, the dependency graph here is more abstract. The dependency graph in Squirrel is constructed with AST nodes, while the dependency graph in Squill consists of abstract data structures, such as Column and VirtualTable. As `VirtualTable_s1` contains the result of a subquery, assuming that we randomly choose `Column_v7` for `Column_x1`, and `Column_v6` for `Column_x2`, we can fill nodes in them with semantics based on the dependencies. The result after filling in semantics is shown in `step2`. Compared with Squirrel's method, semantics-aware instantiation can effectively handle the SQL statement with nested structures like subquery.

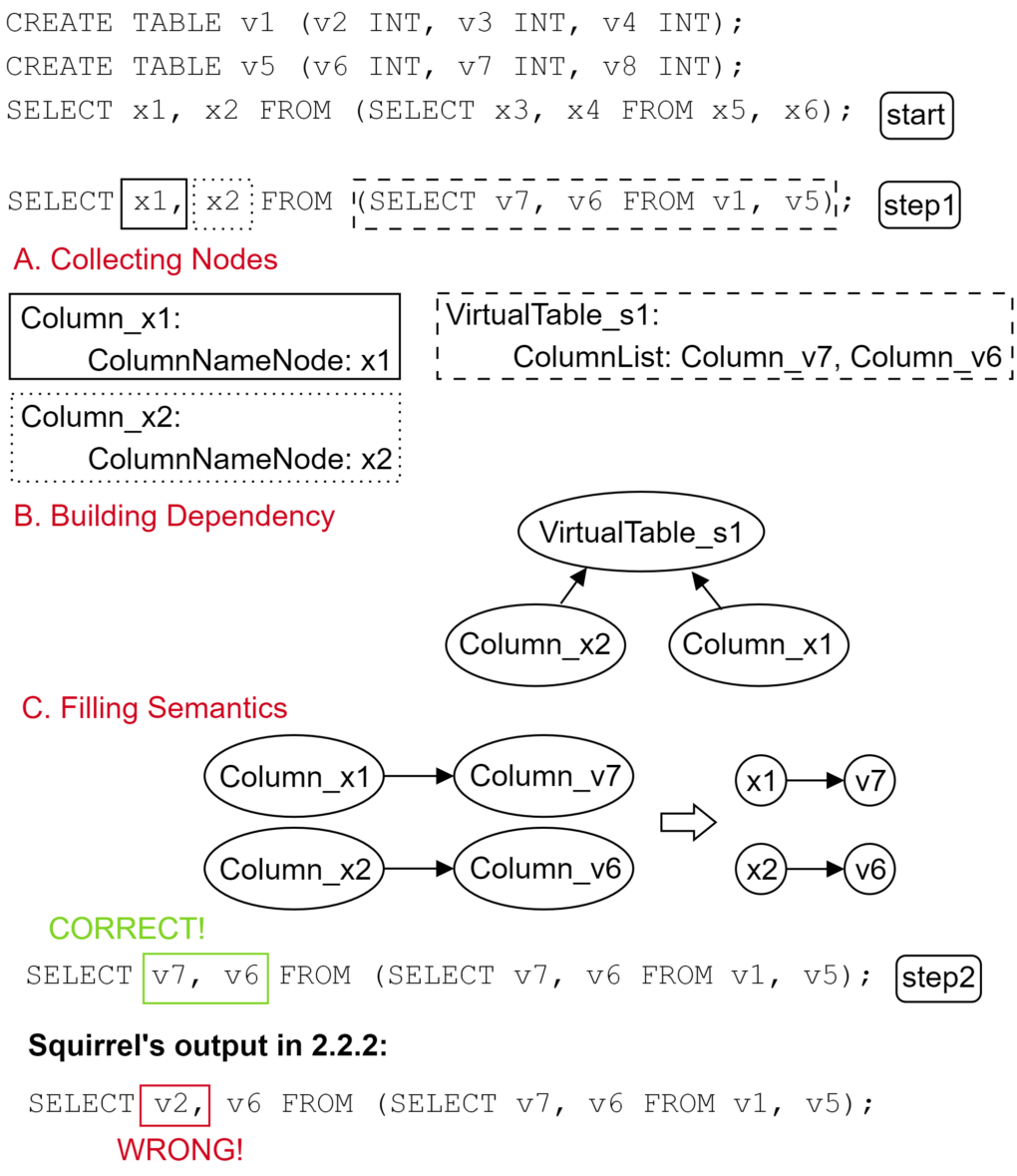


Figure 6. An example of instantiation of complicated semantics.

4. Implementation

Squill is implemented based on Squirrel. Since Squirrel is at the top of AFL, we implement the correctness-guided mutation based on the seed selection mechanism of AFL. In the implementation, we judge the correctness of the input according to the error code returned by the DBMS after executing. Additionally, the interesting material library has the same structure as the mutation material library in Squirrel, where the main difference between them is that the former stores subtrees of seeds that are invalid and `favored` while the latter stores subtrees of all seeds. We implement a new instantiation stage after mutation

to replace the instantiator of Squirrel for its fundamental limitation in design. We improve the grammar of the AST parser since there are omissions and errors in Squirrel's grammar, and we remove the code that defines the refined data type.

5. Evaluation

We applied our tool Squill on real-world DBMSs to verify its effectiveness. The evaluation was designed to answer the following questions:

Q1. Can Squill detect bugs from well-tested DBMSs? (Section 5.1).
Q2. Can Squill perform better than existing tools? (Section 5.2).
Q3. How does correctness-guided mutation help fuzzing? (Section 5.3).
Q4. What is the contribution of semantics-aware instantiation? (Section 5.4).

We selected three popular real-world DBMSs for evaluation, including MySQL, MariaDB, and OceanBase. We mainly compared Squill with Squirrel, as Squirrel had been shown to outperform other mutation-based fuzzers, such as AFL, and generation-based fuzzers, such as SQLsmith. We did not compare Squill with SQLRight [18] and SQLancer, because their target is the logic bug of DBMSs, while Squill, like Squirrel, focuses on the memory error of DBMSs. We perform the experiments on three computers with Ubuntu 18.04 system, Intel(R) Core(TM) i7-10700 (2.90 GHz) CPU, and 32 GB memory. We used the llvm mode of AFL to instrument the DBMS. Because of the large codebase of DBMSs, we set the bitmap size to 256 K and used a 20% ratio instrumentation. The DBMS versions in the experiment are all the latest, including MySQL 8.0.29, MariaDB 10.10.0, and OceanBase 3.1.4. In the experiments, due to resource bottleneck, we ran one DBMS and a fuzzer on each machine for 24 h at a time and repeated three times. Squill and Squirrel used the same seed and initial library in experiments.

5.1. DBMS Bugs

In total, Squill found 63 bugs, including 30 bugs from MySQL, 27 from MariaDB, and 6 from OceanBase. The details of these bugs are shown in Table 1. We have reported all bugs to the developers of the appropriate DBMS. At the time of paper writing, 19 of all bugs have been fixed, with 9 CVEs assigned. The type of bugs found by Squill are listed in the second column of Table 1. Specifically, Squill found 10 bugs related to buffer overflows and use-after-free.

Table 1. Bugs detected by Squill.

ID	Type	Description	Status	Reference
MySQL 8.0.27				
1	SEGV	Common_table_expr::clone_tmp_table()	Fixed	CVE-2022-21509
2	HOF	make_join_readinfo()	Fixed	CVE-2022-21526
3	SEGV	Item_field::fix_outer_field()	Fixed	CVE-2022-21527
4	SEGV	push_new_name_resolution_context()	Fixed	CVE-2022-21528
5	SEGV	QEP_shared_owner::table()	Fixed	CVE-2022-21529
6	SEGV	Item_field::used_tables()	Fixed	CVE-2022-21530
7	SEGV	QEP_shared_owner::idx()	Fixed	CVE-2022-21531
8	HOF	compare_fields_by_table_order()	Fixed	CVE-2022-21438
9	AF	Query_expression::accumulate_used_tables()	Fixed	CVE-2022-21459
10	AF	MoveCompositeIteratorsFromTablePath()	Fixed	BUG106045
11	SEGV	Query_block::next_query_block()	Fixed	BUG106047
12	AF	temptable::Handler::position()	Verified	BUG106048
13	SEGV	Item_subselect::exec()	Verified	BUG106050
14	SEGV	Bitmap::merge()	Verified	BUG106051
15	SEGV	TABLE::empty_result_table()	Verified	BUG106058
16	AF	SubqueryWithResult::single_query_block()	Verified	BUG106061
17	AF	TABLE_LIST::create_materialized_table()	Verified	BUG106055

Table 1. *Cont.*

ID	Type	Description	Status	Reference
MySQL 8.0.29				
18	HUAF	Item_field::used_tables_for_level()	Verified	BUG108241
19	AF	handler::ha_index_next_same()	Verified	BUG108242
20	AF	Bounds_checked_array::operator[]()	Verified	BUG108243
21	SEGV	KEY::records_per_key()	Verified	BUG108244
22	AF	add_key_fields()	Verified	BUG108246
23	AF	add_key_field()	Verified	BUG108247
24	AF	Query_block::get_derived_expr()	Verified	BUG108248
25	AF	Item_func_case::find_item()	Verified	BUG108249
26	SBOF	Query_expression::prepare()	Verified	BUG108251
27	AF	Item_func_in::val_int()	Verified	BUG108252
28	SEGV	Item_ref::walk()	Verified	BUG108253
29	AF	copy_contexts()	Verified	BUG108254
30	SBOF	Item_func::fix_fields()	Verified	BUG108255
MariaDB 10.3.35				
31	SEGV	update_depend_map_for_order()	Verified	MDEV-28501
32	SEGV	st_select_lex::next_select()	Verified	MDEV-28502
33	SEGV	get_addon_fields()	Verified	MDEV-28503
34	SEGV	With_element::get_name()	Verified	MDEV-28504
35	SEGV	sub_select()	Verified	MDEV-28505
36	AF	find_field_in_table_ref()	Verified	MDEV-28506
37	SEGV	Item_field::fix_outer_field()	Verified	MDEV-28507
38	AF	create_tmp_table()	Fixed	MDEV-28508
39	SEGV	Bitmap::merge()	Verified	MDEV-28509
40	SEGV	get_sort_by_table()	Verified	MDEV-28510
41	SEGV	Item_subselect::init_expr_cache_tracker)	Verified	MDEV-28614
42	AF	handler::ha_rnd_next()	Verified	MDEV-28615
43	SEGV	Item_ref::fix_fields()	Verified	MDEV-28616
44	SEGV	TABLE_LIST::set_check_materialized()	Fixed	MDEV-28617
45	SEGV	Item_equal::val_int()	Verified	MDEV-28618
46	SEGV	Window_funcs_sort::setup()	Verified	MDEV-28619
47	SEGV	Item_subselect::get_cache_parameters()	Verified	MDEV-28620
48	AF	Item_subselect::exec()	Verified	MDEV-28621
49	SEGV	Item_exists_subselect::exists2in_processor()	Verified	MDEV-28622
50	AF	resolve_ref_in_select_and_group()	Verified	MDEV-28623
51	AF	Item_field::fix_fields()	Verified	MDEV-28624
MariaDB 10.10.0				
52	SBOF	st_select_lex_unit::set_unique_exclude()	Verified	MDEV-29358
53	HUAF	Field::is_null()	Verified	MDEV-29359
54	SEGV	grouping_field_transformer_for_where()	Verified	MDEV-29360
55	SBOF	resolve_references_to_cte()	Verified	MDEV-29361
56	AF	Item_singlerow_subselect::val_int()	Verified	MDEV-29362
57	HUAF	calc_group_buffer()	Verified	MDEV-29363
OceanBase 3.1.4				
58	AF	ObInsertResolver::resolve_insert_values()	Fixed	issues 986
59	HOF	ABitSet::myffsl()	Fixed	issues 987
60	SEGV	ObLatchMutex::try_lock()	Fixed	issues 988
61	AF	ObSelectStmtPrinter::print_with()	Fixed	issues 989
62	SEGV	sql::ObExpr::count()	Fixed	issues 995
63	SEGV	ObMergeJoinOp::ChildRowFetcher::next()	Fixed	issues 1000

HUAF: heap-use-after-free. SBOF: stack-buffer-overflow. HOF: heap-buffer-overflow. SEGV: segmentation violation. AF: assertion failure

Case Study. Squill detected a bug in MariaDB (ID 44 in Table 1, PoC in Listing 2), which can cause a DBMS crash by a null pointer accessing. This bug happened in INSERT...SELECT statements whose WHERE condition contains an IN/ANY/ALL predicand with a special GROUP clause, which can be eliminated and contains a subquery over a mergeable derived table referencing the updated table. It is caused by the incorrect access to the derived table which has been eliminated. The bug can cause a similar crash when executing a single-table DELETE statement with EXISTS subquery whose WHERE condition is like this. Executing this kind of query will cause a crash of DBMS in the preparation phase. The stability of the DBMS is critical, as it is usually the infrastructure for some information systems which require high availability, such as business systems in banks. Denial-of-service attacks based on such vulnerabilities can make the DBMS crash, resulting in serious consequences.

Listing 2. A PoC of ID 44 in Table 1.

```
CREATE TABLE v0 ( v1 BOOLEAN, v2 INT, v3 INT );
CREATE TABLE v4 ( v5 INT NOT NULL, v6 INT, v7 INT );
INSERT INTO v4 ( v7 ) VALUES ( ( ( TRUE ,v5 ) NOT IN
( SELECT ( - 49 ) AS v8, -128 FROM v0 GROUP BY ( TRUE, v3 )
NOT IN ( SELECT v5, ( SELECT v2 FROM ( WITH v9 AS ( SELECT v7
FROM ( SELECT NOT v5 <= 'x' , FROM v4 GROUP BY v7 ) AS v10 )
SELECT v7, ( v7 = 67 OR v7 > 'x' ) FROM v4 ) AS v11 NATURAL JOIN
v0 WHERE v7 = v3 ) AS v12 FROM v4 ), v2 ) OR v5 > 'x' ) );
```

5.2. Comparison with Existing Tools

We evaluate Squill and Squirrel on three real-world DBMSs, MySQL, MariaDB, and OceanBase, to help us better understand the performance of Squill. As shown in Figure 7, we compare the capability of bug finding and path exploration between the two tools. The number details are listed in Table 2. More program paths explored and more bugs found per unit of time means better fuzzer performance. We also compared the type of bugs they found, which represents how harmful the bug is. Since Squirrel will drop long inputs with multiple subqueries, we disable the length and the subquery check of Squirrel, denoted as Squirrel$_{!check}$.

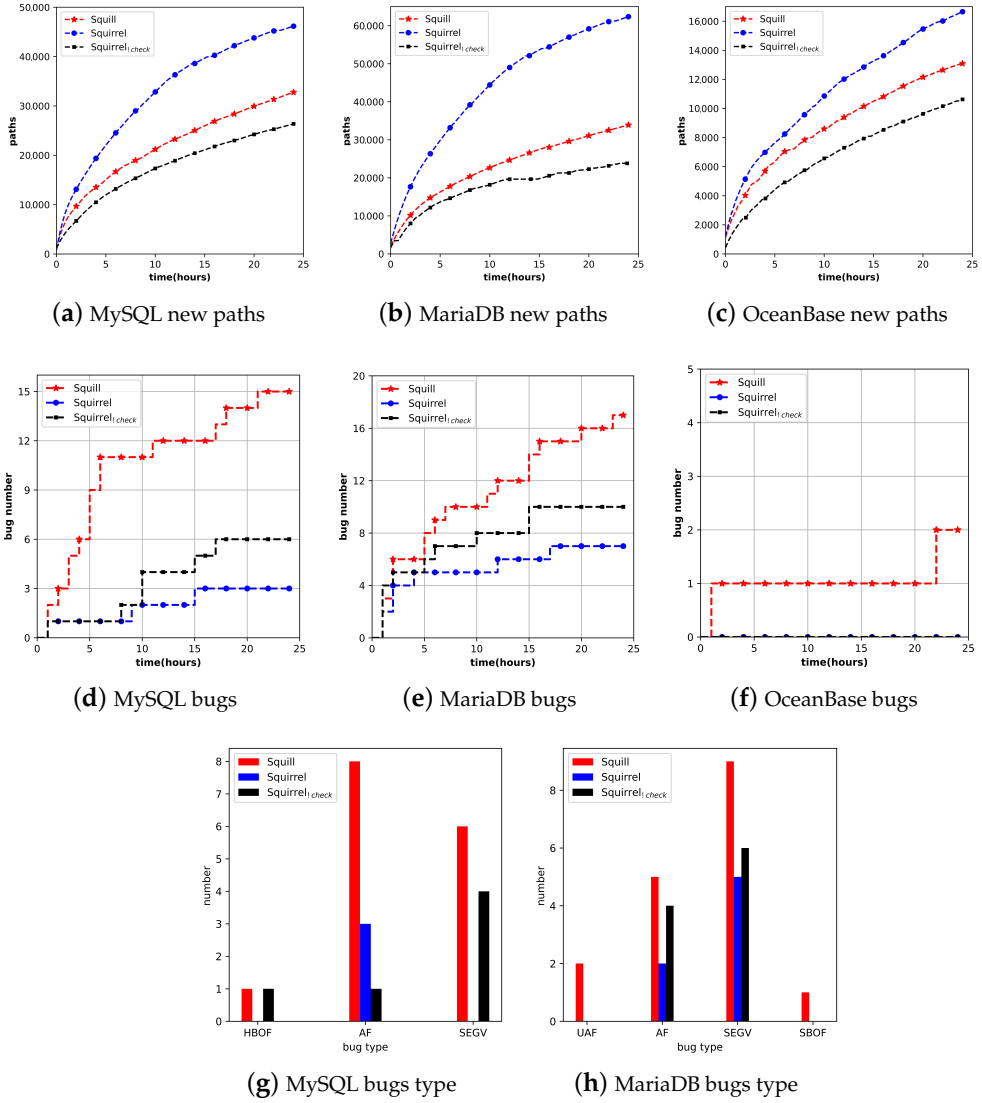
Table 2. The number of paths and bugs explored by each fuzzer in 24 h.

DBMS	Squill		Squirrel		Squirrel$_{!check}$	
	Paths	Bugs	Paths	Bugs	Paths	Bugs
MySQL	32,827	15	46,232	3	26,387	6
MariaDB	33,904	17	62,465	7	23,835	10
OceanBase	13,081	2	16,684	0	10,630	0

In statistics, we deduplicate crashes to the corresponding bug since a bug often causes hundreds of crashes and summarize the number of bugs by the hour. Due to the multithreading feature of DBMSs, the unique crash mechanism of AFL is hard to deduplicate DBMS crashes accurately. For MySQL and MariaDB, we deduplicate crashes according to the report output by ASan [19]. For OceanBase, we use GDB [20] to debug each crash after fuzzing and deduplicate according to the information, such as the call stack of functions, at the time of the DBMS crash.

Path Exploration. Figure 7a–c show the number of paths explored by Squill, Squirrel, and Squirrel$_{!check}$ over time in MySQL, MariaDB, and OceanBase. As we can see, Squirrel explored more paths than Squill and Squirrel$_{!check}$. It is because Squirrel drops long inputs with multiple subqueries for their low semantics-correct rate after instantiation. The input generated by Squirrel is very short and simple and with a fast execution speed. However,

with semantics-aware instantiation, we do not need to limit the number of subqueries in inputs generated by Squill. Thus Squill can generate long and complex inputs, which means a slow execution speed. Faster execution usually means more paths. So we tested Squirrel$_{!check}$, which can also generate long and complex inputs, to evaluate Squill more comprehensively. Compared with Squirrel$_{!check}$, Squill explored 24% more paths in MySQL, 40% in MariaDB, and 23% in OceanBase. Moreover, Squill and Squirrel found many more paths on MySQL and MariaDB than OceanBase. We think this may be caused by the feature of OceanBase as a distributed database and the bad grammar compatibility of the fuzzer with OceanBase.



(**a**) MySQL new paths (**b**) MariaDB new paths (**c**) OceanBase new paths

(**d**) MySQL bugs (**e**) MariaDB bugs (**f**) OceanBase bugs

(**g**) MySQL bugs type (**h**) MariaDB bugs type

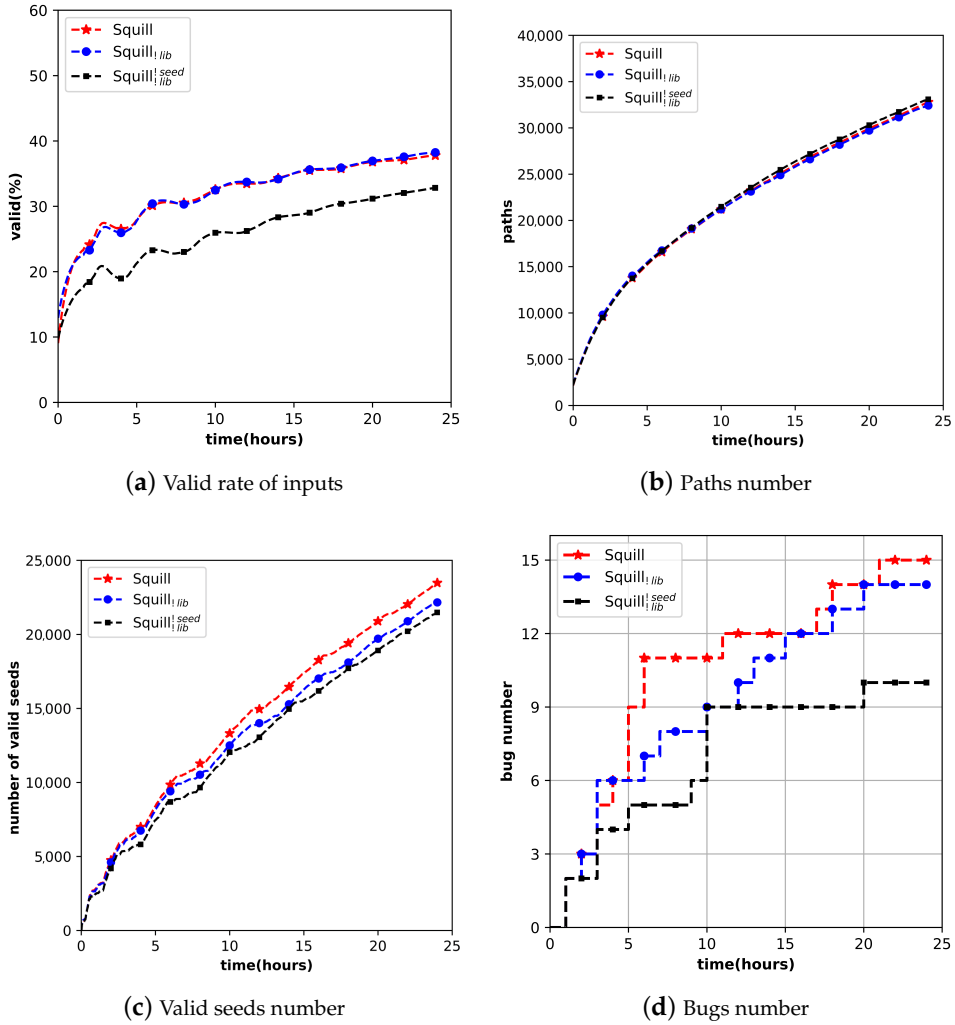
Figure 7. Comparison with existing tools.

Bug Finding. Figure 7d–f show the number of bugs found by Squill, Squirrel, and Squirrel$_{!check}$ over time in MySQL, MariaDB, and OceanBase. In total, Squill found 3.4x and 2x more bugs than Squirrel and Squirrel$_{!check}$, which shows the effectiveness of Squill in bug finding. Note that Squill and Squirrel$_{!check}$ found more bugs than Squirrel in MySQL and MariaDB. The result proves that there is no fundamental reason that maximizing the number of paths (or seeds) is directly connected to finding bugs [21]. Figure 7g,h show the type of bugs found by Squill, Squirrel, and Squirrel$_{!check}$. The main types of bugs are assertion fails and SEGV. It shows that Squill found a total of four buffer-related errors, while Squirrel and Squirrel$_{!check}$ only found one.

Overall, Squill outperforms Squirrel in finding memory error bugs of real-world DBMSs. Because Squill has the ability to generate valid complex SQL while Squirrel cannot. Moreover, Squill embeds correctness-guided mutation, which can improve the efficiency of fuzzing. Squill can also explore more paths than $\text{Squirrel}_{!check}$, which shows the effectiveness of Squill.

5.3. Contribution of Correctness-Guided Mutation

To understand the contribution of different factors in correctness-guided mutation, we disable each factor to perform unit tests in MySQL and measure various aspects of the fuzzing process. In addition to the capabilities of bug finding and path exploration, we also compare the correctness of the input. Figure 8 shows the result, where $\text{Squill}_{!lib}^{!seed}$ means we disable both correctness-focused seed selection and mutation with interesting material library, and $\text{Squill}_{!lib}$ means we only disabled interesting material library. Since the implementation of the mutation with interesting material library relies on correctness-focused seed selection, we do not disable the latter and keep the former.



(**a**) Valid rate of inputs (**b**) Paths number

(**c**) Valid seeds number (**d**) Bugs number

Figure 8. Contributions of correctness-guided mutation. The experiment is performed on MySQL.

Correctness of Inputs. Figure 8a shows the valid rate of inputs when fuzzing, which means the proportion of valid inputs in all inputs. Higher valid rate of inputs when fuzzing is better, because we want the input to pass the validity check of the DBMS. The result is $\text{Squill} \approx \text{Squill}_{!lib} > \text{Squill}_{!lib}^{!seed}$, where $\text{Squill}_{!lib}$ is 6% higher than $\text{Squill}_{!lib}^{!seed}$. The result shows

that the correctness-focused seed selection improves the ability of the fuzzer to generate more valid inputs because of its strategy to prioritize mutating seeds which is valid.

Path Exploration. Figure 8b shows the number of paths explored by each fuzzer. Squill, Squill$_{!lib}$, and Squill$^{!seed}_{!lib}$ are almost equal in the number of paths, and Squill$^{!seed}_{!lib}$ is slightly higher than the other two. Due to that Squirrel$^{!seed}_{!lib}$ generates more syntactically and semantically incorrect inputs, as shown in Figure 8a, its inputs are executed faster, leading to more paths. In addition, we count the number of valid seeds generated during fuzzing, as shown in Figure 8c. A seed represents a path since only if an input triggers a new path will it be saved into the seed queue as a seed. Therefore, the number of valid seeds reflects the capability of exploring the path which passes the syntactic and semantic check of DBMS. The result is Squill > Squill$_{!lib}$ > Squill$^{!seed}_{!lib}$, where Squill is approximately 9% higher than Squill$^{!seed}_{!lib}$, and Squill$_{!lib}$ is about 3% higher than Squill$^{!seed}_{!lib}$. The result shows that both two mechanisms can help fuzzing in path exploration.

Bug Finding. Figure 8d shows the number of bugs found by Squill with each setting, where the original Squill achieves the best results. Squill and Squill$_{!lib}$ found 15 and 14 bugs in MySQL, while Squill$^{!seed}_{!lib}$ only found 10. The results show that the correctness-guided mutation plays an important role in bug finding.

Overall, both mechanisms of correctness-guided mutation improve the effectiveness of Squill in path exploration and bug finding, where correctness-focused seed selection improves the ability of Squill to generate more valid inputs and mutation with interesting material library helps Squill explore more DBMS states after the syntactic and semantic check.

5.4. Contribution of Semantics-Aware Instantiation

In this section, we evaluate semantics-aware instantiation introduced in Section 3.2. We perform instantiation method of Squill and Squirrel to instantiate SQL statements of the same dataset and input the SQL statements with semantics to DBMS. We evaluate the instantiation of Squill by comparing the correctness of these inputs. The higher valid rate of input after instantiation is, the better instantiation method is. In the end, we illustrate the advantages of Squill instantiation through a practical example.

The dataset contains all valid seeds in one MySQL fuzzing of Squill because we want to compare the two methods' capability to instantiate some critical inputs and ensure that these inputs can be correctly instantiated. We normalize the seeds before adding them to the dataset, that is, removing the semantics in them. Due to the design of translating AST to string, the input generated by Squill has a very tiny probability that it cannot be parsed by itself (same with Squirrel). Moreover, there are differences between the grammar of Squill and Squirrel, and Squirrel cannot parse some inputs of Squill. So we remove the seeds that both Squill and Squirrel cannot parse. The evaluation results are shown in Table 3.

Table 3. The comparison between instantiation of Squill and Squirrel.

Fuzzer	Seed Size	Valid	Invalid	Total	Valid Rate
Squill	**<1 kb**	5246	373	5619	93.36%
	1∼1.5 kb	15,810	1328	17,138	92.25%
	>1.5 kb	8280	676	8956	92.45%
	total	29,336	2377	31,713	92.5%
Squirrel	**<1 kb**	4304	1315	5619	76.6%
	1∼1.5 kb	9747	7391	17,138	56.87%
	>1.5 kb	4419	4537	8956	49.34%
	total	18,470	13,243	31,713	58.24%

The results show that the instantiation of Squill (92.5% valid rate) outperforms Squirrel's (58.24% valid rate). In addition, we make separate statistics according to the file size of the seeds. The file size of seeds corresponds to the length of the SQL statement, which we think is positively related to the complexity of the SQL statement. Long SQL statement

usually means more complicated dependencies between nodes and more nested structures, such as subquery. With the increased complexity of SQL statements (file size), the valid rate of Squirrel's instantiation is significantly reduced, while the correct rate of Squill's instantiation changes less. This shows the advantage of Squill's instantiation in processing complex SQL statements. Because of the randomness in semantics filling, the valid rate of Squill's instantiation is not 100%, though the input was instantiated correctly before.

Listing 3 is a PoC of ID 23 in Table 1. It can be seen that there is a nested structure containing subqueries in the SELECT statement in Line 3. This kind of nested structure is pervasive in SQL statements generated by mutation, which may be closely related to overflow vulnerabilities. It is difficult for Squirrel to instantiate such type of structure since Squirrel is hard to build correct dependencies between subqueries, such as the semantics of the first two v4 positions in Line 3.

Listing 3. A PoC of ID 23 in Table 1.

```
CREATE TABLE v0 ( v1 NUMERIC UNIQUE, v2 BIGINT ) ;
CREATE TABLE v3 ( v4 INT, v5 INT ) ;
SELECT 1 FROM v3 GROUP BY ( SELECT v4 FROM
( SELECT v4 FROM ( SELECT v4 FROM v3 UNION SELECT v1 FROM v0 )
AS v7 WHERE (v4 = 0 AND v4 = -1 AND v4 = 67 ) ) AS v9);
```

6. Discussion

In this section, we discuss several limitations of our current implementation and possible future directions.

Universality of Fuzzer. In this paper, the instantiation of Squill is based on the grammar of MySQL, which has low universality. So we chose MariaDB and OceanBase, which are compatible with MySQL grammar, for evaluation. The cost of migrating this approach to other DBMSs is slightly higher than Squirrel. Moreover, the universality of the method is also very important [22,23]. In the future, we plan to achieve the universality of the fuzzer by implementing an instantiation method that satisfies the intersection of most SQL grammars and then writing extensions for each DBMS based on this universal method.

Mode of Input. Both Squill and Squirrel start with an empty database, and the input is a combination of CREATE, INSERT, and SELECT statements. We observed that most of the seeds that triggered new code coverage were mutated in SELECT statements. Changing the data inserted and the table structure created usually does not bring new paths. We think there is room for optimization. For example, we can construct a series of tables with complex structure and data as the initial database and only input SELECT statements in fuzzing. This can save the overhead of table creation and data insertion of each input.

Mutation Operator. Squill and Squirrel use the same mutation operators, including insertion, deletion, and replacement of AST nodes. We think there are other mutation operators suitable for DBMS fuzzing scenarios. For example, the random recursive mutation operator mentioned in Nautilus [24] randomly selects a recursive tree and repeats the recursion 2^n times. Such mutation operators may help trigger buffer overflow vulnerabilities of DBMSs.

Fuzzing Partial. Most of the vulnerabilities detected by Squill and Squirrel are located in the parser and optimizer components of the DBMS. It means the main target of the current DBMS fuzzer is the parser and optimizer rather than the executor of the DBMS. However, the storage process in the executor is time-consuming, as it involves the disk IO. So one optimization idea is to separate the parser and optimizer by analyzing the source code of the DBMS. Fuzzing these separated-out functions can significantly reduce the overhead during the execution phase of the DBMS, improving the efficiency of fuzzing.

7. Related Work

In this section, we discuss the recent DBMS testing technologies related to Squill.

Black-box DBMS Fuzzing. Black-box fuzzing, or generation-based fuzzing, has been widely used to detect DBMS bugs. With a specific predefined schema, continuously generating a large number of SQL statements into the DBMS to trigger abnormal behaviors (usually crashes) of the DBMS is one method of black-box DBMS fuzzing. Sqlsmith [1] is a representative of this kind of black-box DBMS fuzzer. Based on AST, it randomly generates SQL query statements for the initial database through a series of highly customized rules. In addition, differential testing is another standard method used to detect DBMS vulnerabilities in black-box DBMS fuzzing. Rags [25] and Sparkfuzz [26] send the same SQL query to different DBMSs and detect correctness bugs by comparing the differences in the results. Sqlancer [2–4] constructs different SQL statements of functionally equivalent through several different patterns and inputs them into the same DBMS. If the results are different, the DBMS might have a logical bug. Similarly, AMOEBA [27] constructs query pairs that are semantically equivalent to each other and then compares their response time on the same database system to detect performance bugs. The main difference between Squill and the works above is that Squill is a grey-box fuzzer with feedback like code coverage. Compared with blind fuzzing, fuzzing with feedback can comprehensively explore program states and trigger the deep logic of DBMSs.

Grey-box DBMS Fuzzing. In recent years, grey-box or mutation-based fuzzing has shown its effectiveness in memory error bug detection [28–37]. AFL [6], which is an important milestone in the area of software security testing [38], has been applied to DBMS fuzzing. However, the fuzzer, like AFL, performs poorly in generating structural inputs, such as SQL statements. Though there are many works trying to address this challenge, such as Zest [39], GRIMOIRE [40], and Nautilus [24]. Their ability to generate syntactically and semantically correct SQL queries is still not good enough due to the strict syntactic and semantic requirements of the DBMS. The recent work Squirrel [11] focuses on the DBMS fuzzing scenarios. Through a customized parser based on Bison [41] and Flex [42], Squirrel translates SQL statements into AST and mutates based on the AST to guarantee the syntax correctness of the inputs. After mutation, Squirrel fills the newly generated inputs with semantics to increase their semantic correctness. There are many works based on Squirrel. With its industry-oriented design, Ratel [43] improves the feedback precision in DBMS fuzzing and enhances the robustness of input generation. SQLRight [18] combines differential testing and mutation-based fuzzing to detect logic bugs of the DBMS. Squill is also based on Squirrel, using the correctness of seeds as feedback to guide fuzzing. Moreover, Squill introduces an instantiation method that can generate correct semantics for SQL statements with nested structures.

8. Conclusions

In this paper, we design and implement Squill to find memory errors in DBMSs. We introduce the correctness of seeds into DBMS fuzzing as feedback and propose two methods: correctness-focused seed selection and mutation with interesting material library. Additionally, we investigate the challenge of semantics filling in DBMS fuzzing and design a new instantiation method to address this challenge. We evaluated Squill on popular real-world DBMSs and found 30 bugs in MySQL, 27 in MariaDB, and 6 in OceanBase, with 9 CVEs assigned. The evaluation showed that Squill could find more bugs in DBMSs than existing tools.

Author Contributions: Conceptualization, P.J.; Data curation, P.Y.; Formal analysis, S.W.; Funding acquisition, P.J.; Methodology, S.W.; Software, S.W.; Supervision, C.H.; Validation, P.Y.; Visualization, S.W.; Writing—original draft, S.W.; Writing—review and editing, P.J. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by National Key R&D projects of China OF FUNDER grant number 2021YFB3101803.

Institutional Review Board Statement: Not appliable.

Informed Consent Statement: Not appliable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Seltenreich, A. SQLSmith. Available online: https://github.com/anse1/sqlsmith (accessed on 22 November 2022).
2. Rigger, M.; Su, Z. 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, ESEC/FSE 2020, Virtual Event, 8–13 November 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 1140–1152. [CrossRef]
3. Rigger, M.; Su, Z. Testing Database Engines via Pivoted Query Synthesis. In Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation, OSDI'20, Virtual, 4–6 November 2020; USENIX Association: Berkeley, CA, USA, 2020.
4. Rigger, M.; Su, Z. Finding Bugs in Database Systems via Query Partitioning. *Proc. ACM Program. Lang.* **2020**, *4*, 1–30. [CrossRef]
5. Wikipedia. Fuzzing. Available online: https://en.wikipedia.org/wiki/Fuzzing (accessed on 22 November 2022).
6. Zalewski, M. American Fuzzy Lop. Available online: https://github.com/google/AFL (accessed on 22 November 2022).
7. Consortium, S. How SQLite Is Tested. Available online: https://www.sqlite.org/testing.html (accessed on 22 November 2022).
8. LLVM. LibFuzzer. Available online: https://www.llvm.org/docs/LibFuzzer.html (accessed on 22 November 2022).
9. Google. Honggfuzz. Available online: https://github.com/google/honggfuzz (accessed on 22 November 2022).
10. Fioraldi, A.; Maier, D.; Eißfeldt, H.; Heuse, M., AFL++: Combining Incremental Steps of Fuzzing Research. In Proceedings of the 14th USENIX Conference on Offensive Technologies, Berkeley, CA, USA, 11 August 2020; USENIX Association: Berkeley, CA, USA, 2020.
11. Zhong, R.; Chen, Y.; Hu, H.; Zhang, H.; Lee, W.; Wu, D. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In Proceedings of the CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, 9–13 November 2020; Ligatti, J., Ou, X., Katz, J., Vigna, G., Eds.; ACM: New York, NY, USA, 2020; pp. 955–970. . [CrossRef]
12. Oracle. MySQL Server. Available online: https://github.com/mysql/mysql-server (accessed on 22 November 2022).
13. Foundation, M. MariaDB. Available online: https://github.com/MariaDB/server (accessed on 22 November 2022).
14. Group, A. OceanBase. Available online: https://github.com/oceanbase/oceanbase (accessed on 22 November 2022).
15. Lemieux, C.; Sen, K. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, 3–7 September 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 475–485. [CrossRef]
16. Wang, Y.; Jia, X.; Liu, Y.; Zeng, K.; Bao, T.; Wu, D.; Su, P. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In Proceedings of the 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, CA, USA, 23–26 February 2020; The Internet Society: Washington, DC, USA, 2020.
17. Yue, T.; Wang, P.; Tang, Y.; Wang, E.; Yu, B.; Lu, K.; Zhou, X. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In Proceedings of the 29th USENIX Conference on Security Symposium, SEC'20, San Diego, CA, USA, 12–14 August 2020; USENIX Association: Berkeley, CA, USA, 2020.
18. Liang, Y.; Liu, S.; Hu, H. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In Proceedings of the 31st USENIX Security Symposium (USENIX Security 22), Boston, MA, USA, 10–12 August 2022; USENIX Association: Boston, MA, USA, 2022; pp. 4309–4326.
19. Google. AddressSanitizer. Available online: https://github.com/google/sanitizers/wiki/AddressSanitizer (accessed on 22 November 2022).
20. Foundation, F.S. GDB: The GNU Project Debugger. Available online: http://www.sourceware.org/gdb/ (accessed on 22 November 2022).
21. Klees, G.; Ruef, A.; Cooper, B.; Wei, S.; Hicks, M. Evaluating Fuzz Testing. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18, Toronto, ON, Canada, 15–19 October 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 2123–2138. [CrossRef]
22. Yan, L.; Ahmad, M.W.; Jawarneh, M.; Shabaz, M.; Raffik, R.; Kishore, K.H.; Azeem, I. Single-Input Single-Output System with Multiple Time Delay PID Control Methods for UAV Cluster Multiagent Systems. *Secur. Commun. Netw.* **2022**, *2022*, 3935143. [CrossRef]
23. Gao, H.; Kareem, A.; Jawarneh, M.; Ofori, I.; Raffik, R.; Kishore, K.H. Metaheuristics Based Modeling and Simulation Analysis of New Integrated Mechanized Operation Solution and Position Servo System. *Math. Probl. Eng.* **2022**, *2022*, 1466775. [CrossRef]
24. Aschermann, C.; Frassetto, T.; Holz, T.; Jauernig, P.; Sadeghi, A.; Teuchert, D. NAUTILUS: Fishing for Deep Bugs with Grammars. In Proceedings of the 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, CA, USA, 24–27 February 2019; The Internet Society: Washington, DC, USA, 2019.
25. Slutz, D. *Massive Stochastic Testing of SQL*; Technical Report MSR-TR-98-21; Publisher: Burlington, MA, USA, 1998.

26. Ghit, B.; Poggi, N.; Rosen, J.; Xin, R.; Boncz, P. SparkFuzz: Searching Correctness Regressions in Modern Query Engines. In Proceedings of the Workshop on Testing Database Systems, DBTest '20, Portland, OR, USA, 19 June 2020; Association for Computing Machinery: New York, NY, USA, 2020. [CrossRef]
27. Liu, X.; Zhou, Q.; Arulraj, J.; Orso, A. Automatic Detection of Performance Bugs in Database Systems Using Equivalent Queries. In Proceedings of the 44th International Conference on Software Engineering, ICSE '22, Pittsburgh, PA, USA, 25–27 May 2022; Association for Computing Machinery: New York, NY, USA, 2022; pp. 225–236. [CrossRef]
28. Gan, S.; Zhang, C.; Qin, X.; Tu, X.; Li, K.; Pei, Z.; Chen, Z. CollAFL: Path Sensitive Fuzzing. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 679–696. [CrossRef]
29. Manès, V.J.M.; Kim, S.; Cha, S.K. Ankou: Guiding Grey-Box Fuzzing towards Combinatorial Difference. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20, Seoul, Republic of Korea, 27 June–19 July 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 1024–1036. [CrossRef]
30. Lyu, C.; Ji, S.; Zhang, C.; Li, Y.; Lee, W.H.; Song, Y.; Beyah, R. MOPT: Optimized Mutation Scheduling for Fuzzers. In Proceedings of the 28th USENIX Conference on Security Symposium, SEC'19, Berkeley, CA, USA, 14–16 August 2019; USENIX Association: Berkeley, CA, USA, 2019; pp. 1949–1966.
31. Zhou, C.; Wang, M.; Liang, J.; Liu, Z.; Jiang, Y. Zeror: Speed up Fuzzing with Coverage-Sensitive Tracing and Scheduling. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE '20, Melbourne, Australia, 21–25 September 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 858–870. [CrossRef]
32. Aschermann, C.; Schumilo, S.; Blazytko, T.; Gawlik, R.; Holz, T. REDQUEEN: Fuzzing with Input-to-State Correspondence. In Proceedings of the 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, CA, USA, 24–27 February 2019; The Internet Society: Washington, DC, USA, 2019.
33. Chen, P.; Chen, H. Angora: Efficient Fuzzing by Principled Search. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 711–725. [CrossRef]
34. Park, S.; Xu, W.; Yun, I.; Jang, D.; Kim, T. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 1629–1642. [CrossRef]
35. Yun, I.; Lee, S.; Xu, M.; Jang, Y.; Kim, T. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Vancouver, BC, Canada, 16–18 August 2017; USENIX Association: Baltimore, MD, USA, 2018; pp. 745–761.
36. Pham, V.T.; Böhme, M.; Santosa, A.E.; Căciulescu, A.R.; Roychoudhury, A. Smart Greybox Fuzzing. *IEEE Trans. Softw. Eng.* **2021**, *47*, 1980–1997. [CrossRef]
37. Wüstholz, V.; Christakis, M. Harvey: A Greybox Fuzzer for Smart Contracts. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, Virtual Event, 8–13 November 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 1398–1409. [CrossRef]
38. Fioraldi, A.; Maier, D.; Zhang, D.; Balzarotti, D. LibAFL: A framework to build modular and reusable fuzzers. In Proceedings of the CCS 2022, 29th ACM Conference on Computer and Communications Security, Los Angeles, CA, USA, 7–11 November 2022; ACM: New York, NY, USA, 2022.
39. Padhye, R.; Lemieux, C.; Sen, K.; Papadakis, M.; Le Traon, Y. Semantic Fuzzing with Zest. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, 15–19 July 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 329–340. [CrossRef]
40. Blazytko, T.; Aschermann, C.; Schlögel, M.; Abbasi, A.; Schumilo, S.; Wörner, S.; Holz, T. GRIMOIRE: Synthesizing Structure while Fuzzing. In Proceedings of the 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, USA, 14–16 August 2019; USENIX Association: Santa Clara, CA, USA, 2019; pp. 1985–2002.
41. Foundation, F.S. Gnu Bison. Available online: https://www.gnu.org/software/bison (accessed on 22 November 2022).
42. Paxson, V. Flex. Available online: https://github.com/westes/flex (accessed on 22 November 2022).
43. Wang, M.; Wu, Z.; Xu, X.; Liang, J.; Zhou, C.; Zhang, H.; Jiang, Y. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '21, Madrid, Spain, 22–30 May 2021; IEEE Press: Hoboken, NJ, USA, 2021; pp. 328–337. [CrossRef]