SEDAR: Obtaining High-Quality Seeds for DBMS Fuzzing via Cross-DBMS SQL Transfer

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

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

ABSTRACT

Effective DBMS fuzzing relies on high-quality initial seeds, which serve as the starting point for mutation. These initial seeds should incorporate various DBMS features to explore the state space thoroughly. While built-in test cases are typically used as initial seeds, many DBMSs lack comprehensive test cases, making it difficult to apply state-of-the-art fuzzing techniques directly.

To address this, we propose SEDAR which produces initial seeds for a target DBMS by transferring test cases from other DBMSs. The underlying insight is that many DBMSs share similar functionalities, allowing seeds that cover deep execution paths in one DBMS to be adapted for other DBMSs. The challenge lies in converting these seeds to a format supported by the grammar of the target database. SEDAR follows a three-step process to generate seeds. First, it executes existing SQL test cases within the DBMS they were designed for and captures the schema information during execution. Second, it utilizes large language models (LLMs) along with the captured schema information to guide the generation of new test cases based on the responses from the LLM. Lastly, to ensure that the test cases can be properly parsed and mutated by fuzzers, SEDAR temporarily comments out unparsable sections for the fuzzers and uncomments them after mutation. We integrate SEDAR into the DBMS fuzzers SQUIRREL and GRIFFIN, targeting DBMSs such as Virtuoso, MonetDB, DuckDB, and ClickHouse. Evaluation results demonstrate significant improvements in both fuzzers. Specifically, compared to SQUIRREL and GRIFFIN with non-transferred seeds, SEDAR enhances code coverage by 72.46%-214.84% and 21.40%-194.46%; compared to SQUIRREL and GRIFFIN with native test cases of these DBMSs as initial seeds, incorporating the transferred seeds of SEDAR results in an improvement in code coverage by 4.90%-16.20% and 9.73%-28.41%. Moreover, SEDAR discovered 70 new vulnerabilities, with 60 out of them being uniquely found by SEDAR with transferred seeds, and 19 of them have been assigned with CVEs.

*Jie Liang and Yu Jiang are the corresponding authors.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639210

KEYWORDS

DBMS Fuzzing, Initial Seeds, Vulnerability Detection

ACM Reference Format:
Jingzhou Fu, Jie Liang, Zhiyong Wu, and Yu Jiang. 2024. SEDAR: Obtaining High-Quality Seeds for DBMS Fuzzing via Cross-DBMS SQL Transfer. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639210

1 INTRODUCTION

Database Management Systems (DBMSs) play a crucial role in modern software applications as they store data and process queries across various domains. However, DBMSs are not immune to vulnerabilities, which can have severe consequences such as service denials, data leaks, data loss, and even complete system failure. Given the potential harm these vulnerabilities can inflict, it becomes essential to proactively identify and address any security issues within DBMSs to ensure system security and data integrity.

Mutation-based fuzzing is widely recognized as an effective technique for uncovering vulnerabilities in software programs. Fuzzers utilizing mutation-based approaches maintain a pool of initial seeds and iteratively generate new test cases by mutating these seeds. These mutated inputs are then executed on the target programs, exploring new code regions and potentially exposing vulnerabilities. However, when it comes to fuzzing Database Management Systems (DBMSs), which primarily rely on Structured Query Language (SQL) as inputs, the complexity of SQL grammar and the intricacies of DBMS executable binaries pose unique challenges. Traditional mutation-based fuzzing tools often struggle to effectively fuzz DBMSs because the inputs generated by them often result in syntax errors or semantic errors, making it difficult to test the underlying logic and detect vulnerabilities within the DBMS. Therefore, many works [9, 14, 19, 44, 48] are proposed to ensure that the generated inputs for DBMS fuzzing adhere to the expected input formats, enabling comprehensive testing of the deep logic and discovering potential vulnerabilities within DBMSs.

The effectiveness of fuzzing technologies for DBMS heavily depends on the quality of initial seeds. The initial seeds play a crucial role as the starting point for mutations, enabling comprehensive exploration of the state space of programs. To achieve effective DBMS fuzzing, it is essential that the initial seeds encompass a diverse range of DBMS features, allowing for thorough coverage of the target DBMS's functionality and potential vulnerabilities.

Typically, existing mutation-based DBMS fuzzing approaches rely on collecting SQL seeds from built-in unit test cases and regression test suites. By leveraging these existing test cases, these fuzzing techniques obtain initial seeds that have already been designed to cover specific functionalities and test scenarios. This method guarantees that the initial seeds are representative of valid inputs and can help guide the mutation process effectively.

However, a significant obstacle arises when many DBMSs lack comprehensive test cases, leaving fuzzers without a reliable source for collecting SQL statements as initial seeds. Although several works [21, 42, 43, 46] have proposed initial seed generation approaches, these techniques may not be suitable for generating high-quality SQL seeds due to the unique complexities of SQL grammar and the inherent dependencies present in SQL statements. SQL grammar exhibits intricate structures and semantic rules that differ significantly from general-purpose markup languages. These complexities make it challenging to apply traditional seed generation approaches directly to SQL statements. Additionally, SQL statements often involve dependencies, such as table relationships and query constraints, which further complicate the generation of meaningful and representative SQL seeds.

To address the issue of lacking high-quality initial seeds in DBMS fuzzing, a potential solution could be to gather test cases from established DBMSs and transfer them into appropriate initial seeds for the target DBMS. However, the transfer of SQL test cases across DBMSs poses significant challenges. First, the grammar differences between DBMSs make it impractical to directly utilize these test cases as initial seeds. Doing so would result in numerous syntactic and semantic errors when applied to the target DBMS. To effectively trigger deep code regions within the target DBMS, it is crucial to ensure that the SQL statements in the transferred seeds adhere to the grammatical requirements of the target DBMS. Moreover, compatibility issues arise when attempting to use these transferred seeds with existing DBMS fuzzers. The mutators employed by these fuzzers are designed to process the SQL test cases that conform to their specifically supported grammar. As a result, these mutators may fail to parse or mutate the transferred seeds due to the presence of unsupported grammatical constructs. This limitation hinders the effective testing of target DBMS using these fuzzers.

To overcome the challenges, we propose a solution called SEDAR. It guarantees compatibility with the target DBMS by a three-step process: ① First, the existing SQL test cases are executed within their original DBMS environments, allowing for the collection of essential schema information during the execution. This schema information provides valuable insights into the database structure and characteristics. ② Second, the SQL test cases, along with the collected schema information, are input to large language models (LLMs) for further processing. The LLMs leverage the provided schema information to guide the generation of new test cases that align with the expected input formats and behavior of the target DBMS. These newly generated test cases benefit from the extensive knowledge of LLMs to be compatible with the target DBMS. ③ Finally, to ensure mutability by DBMS fuzzers, specific measures are taken. When certain sections of the generated test cases cannot be parsed by the fuzzers' mutators, these sections are marked as comments within the SQL statements. This marking instructs the mutators to ignore those sections while focusing on mutating the rest of the SQL statements. After the mutation, the previously marked sections are uncommented, resulting in test cases that are both compatible and mutable, ready to be used by DBMS fuzzers.

For evaluation, we apply SEDAR to generate initial seeds for four DBMSs: MonetDB, Virtuoso, DuckDB, and ClickHouse. We use two state-of-the-art mutation-based DBMS fuzzers, namely SQUIRREL and GRIFFIN, to assess the quality of the generated seeds. By utilizing the initial seeds provided by SEDAR, these fuzzers were able to achieve significant coverage improvements in the target DBMSs. Specifically, in MonetDB, Virtuoso, DuckDB, and ClickHouse, the fuzzers covered 40.58%-195.45%, 90.82%-126.87%, 62.05%-136.20%, and 72.46%-214.84% more branches, respectively, compared to using non-transferred seeds. When using these transferred seeds to augment the native seeds of these DBMSs for fuzzing, the fuzzers enhance code coverage by 10.77%-28.41%, 12.53%-16.20%, and 4.90%-9.73% in MonetDB, DuckDB, and ClickHouse, respectively. Moreover, the fuzzers identified a total of 70 previously unknown bugs across the evaluated DBMSs. Among them, 60 bugs were uniquely found by SEDAR with transferred seeds, and 19 bugs have been assigned CVE IDs due to their severity. In summary, our paper makes the following contributions:

(1) We identify the critical dependency of fuzzing techniques on the availability of high-quality initial seeds derived from built-in test cases. However, many DBMSs lack such test cases, resulting in hindering effective DBMS fuzzing.
(2) We propose SEDAR, a novel approach that overcomes the limitation of missing initial test cases by enabling the generation of seed inputs for targeted DBMS fuzzing through cross-DBMS SQL transfer. These seeds can also be utilized to augment the native seeds, thereby increasing code coverage and facilitating the detection of unknown bugs.
(3) SEDAR successfully uncovered 70 bugs in real-world DBMSs, 19 of which have been assigned CVE identifiers.

2 BACKGROUND AND MOTIVATION

DBMSs and SQL. Most DBMSs use structured query language (SQL) as the language to manage data. Users can utilize various functionalities of DBMSs by executing corresponding SQL statements. Typically, DBMSs adhere to basic features of ANSI SQL standard [1] for common uses, while they support more advanced features with their unique SQL dialects. The grammars of SQL dialects are not compatible with each other, since similar features may be implemented with distinct grammar between different DBMSs.

Initial Seeds for DBMS Fuzzing. A seed for DBMS fuzzing typically refers to a test case that comprises a series of SQL statements. Mutation-based DBMS fuzzing continuously mutates existing seeds to generate new test cases. The quality of these initial seeds is of utmost importance for the effectiveness of fuzzing. First, the initial seeds serve as the starting point for fuzzing and form the foundation for the subsequent fuzzing iterations. The richness of the features contained in the initial seeds affects the variety of functionalities that can be explored within the DBMS in the following mutation. High-quality initial seeds help increase test coverage and enhance the effectiveness of bug detection. Moreover, due to the unique grammar and behaviors of different DBMSs, it is important to tailor the initial seeds to match the specific characteristics of the

target DBMS, ensuring that the fuzzing is well-suited to explore that particular system effectively.

Test engineers have utilized mutation-based fuzzers to extensively test many widely-used DBMSs, such as SQLite, MySQL, MariaDB, and PostgreSQL. To facilitate the testing process, their initial seeds are often acquired from the DBMSs' built-in test cases. Specifically, these DBMSs have maintained extensive test suites [24, 28, 38], containing thousands of built-in test cases for unit tests, regression tests, performance tests, and other purposes. With access to these test suites, test engineers can effectively collect initial seeds and perform efficient fuzzing on these popular DBMSs.

Lack of Quality Initial Seeds Limits DBMS Fuzzing. Although the acquisition of initial seeds from the extensive test suites is standard practice for those popular DBMSs, it is worth noting that not all DBMSs provide such large and high-quality test suites. The lack of comprehensive built-in test cases in some DBMSs poses a challenge for fuzzers as there is no feasible way to directly collect the suitable initial seeds for these DBMSs. This limitation can adversely impact the performance of DBMS fuzzers due to the absence of high-quality initial seeds.

```
1:CREATE TABLE a (p_id INT, p_name BLOB);
2:INSERT INTO a VALUES (1,NULL);
3:EXPLAIN SELECT * FROM a WHERE p_name='Lilu';
-- In other DBMSs: successfully executed
3:TRACE   SELECT * FROM a WHERE p_name='Lilu';
-- In MonetDB: server crashed
```

Figure 1: A crash bug in MonetDB found by SEDAR.

Figure 1 depicts a crash bug that cannot be detected by current DBMS fuzzers due to the absence of appropriate initial seeds, which is triggered by a `TRACE` statement. The `TRACE` statement is a specialized feature in MonetDB intended for analyzing statement execution, akin to the `EXPLAIN` statement in other DBMSs. However, MonetDB does not officially provide test cases that include the `TRACE` statements. As a result, current DBMS fuzzers cannot discover this bug because none of the initial seeds contain the `TRACE` statement, making it challenging to generate subsequent mutations for triggering the bug. One potential method to test the `TRACE` feature in MonetDB is by transferring the test cases from other DBMSs that use the `EXPLAIN` feature, as their grammars are similar. For example, we can simply replace the keyword `EXPLAIN` with `TRACE` to test MonetDB. However, such transfer faces two challenges:

(1) *The substantial differences in grammar among DBMSs.* While the `TRACE` and `EXPLAIN` statements in the example differ only in one keyword, there are significant differences in grammar across other features, such as storage engines, data types, and function names, among others. Taking MonetDB and SQLite as an example, MonetDB has 225 functions, while SQLite has 165 functions, with only 59 of them sharing the same function names in both DBMSs. Such significant differences make the transfer difficult and laborious.

(2) *The limited grammar support of DBMS fuzzers.* Many DBMS fuzzers are implemented to test specific popular DBMSs, which means their supported grammar is also tailored to those specific systems. Even if test cases from other DBMSs are successfully transferred to match the grammar of MonetDB, the fuzzers may still encounter difficulties in parsing or mutating these transferred test cases due to the lack of support for certain grammars in them.

Basic Idea of SEDAR. To address the first challenge, SEDAR leverages large language models (LLMs) to facilitate the transfer process. LLMs have demonstrated exceptional performance in natural language tasks and programming language tasks [7, 16, 17]. For SQL-related tasks, LLMs have been successfully utilized to generate SQL statements from texts, showcasing outstanding effectiveness [29]. SEDAR utilizes LLMs to transfer SQL statements. It generates prompts that include the individual SQL statements along with their corresponding descriptions, aiding the process of LLMs.

To overcome the second challenge, SEDAR takes steps to ensure that the transferred test cases remain mutable by the fuzzers. It accomplishes this by concealing any unparsable keywords and clauses via commenting them out. This strategy enables the fuzzers to focus solely on the remaining parsable parts, allowing them to efficiently mutate the test cases. Through a combination of SQL transfer and mutability refinement, SEDAR successfully generates high-quality initial seeds compatible with the DBMSs and fuzzers.

3 DESIGN

The overall design of SEDAR is shown in Figure 2, which contains three steps: schema capture, SQL transfer, and mutability refinement. The following paragraphs present the details of each step.

3.1 Schema Capture

Schema capture aims to collect the schema information from the test cases of other DBMSs. The schema refers to the structure or blueprint that delineates the organization of the database. It outlines the logical arrangement of database objects, including tables, columns, indexes, functions, constraints, and others. Schema provides a comprehensive overview of SQL statements, which can be used to describe them and serve as prompts for the subsequent SQL transfer process via LLMs. The schema information provides the context of statements, which enhances the accuracy of the transfer.

The lack of schema information can lead to inaccuracies of the output by LLMs due to the ambiguity of a single SQL statement. For example, in a single `SELECT` statement "`SELECT... WHERE a=b`", the data types of the columns "a" and "b" are ambiguous when schema information is not available. If the statement is to be transferred to a DBMS requiring explicit type casting such as PostgreSQL, the expression "a=b" needs a rewrite with type casting. This could be as "`a=b::BOOL`" when the column "a" is the boolean type or "`a=b::TEXT`" for the string type. Due to the ambiguity, LLMs cannot infer the data types of columns as well as the correct formats of type casting, leading to incorrect transfer. In contrast, when providing schema information, LLMs can generate correct expressions according to the data types from schema information.

Schema information could be retrieved by executing specific queries. DBMSs typically support queries on system tables that can list all database objects with their corresponding names and types, effectively representing the entire schema. However, collecting the entire schema is excessive for describing SQL statements. While databases can contain hundreds of objects such as user-defined or built-in tables, columns, triggers, and functions, A single SQL statement may merely reference several database objects. Only the database objects referenced by the SQL statements in test cases are necessary for retrieving the schema that describes them.

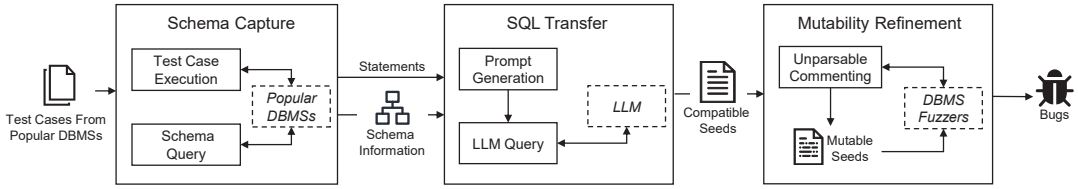


Figure 2: The overall design of SEDAR. First, SEDAR executes the statements in test cases of other DBMSs and collects the schema information of the DBMS during execution. Second, to make the test cases compatible with the grammar of the target DBMS, it feeds the test cases along with the schema information to the LLMs to transfer them into those satisfying the grammar of the target DBMS. Third, to make the test case parsable and mutable by the DBMS fuzzer, it parses the test case with the fuzzer's parser, comments out the unparsable sections, and then uncomments them after mutation.

To collect the database objects referenced by each statement of test cases, SEDAR follows these steps: (1) First, SEDAR executes the test cases within their respective DBMS. It records each executed statement and splits it into tokens. (2) Second, SEDAR enumerates the identifier tokens in the statement and queries the current system tables to locate the corresponding database objects that match these identifiers. (3) Third, for the database objects found by the query, SEDAR constructs the referenced sub-schema according to the information of these objects.

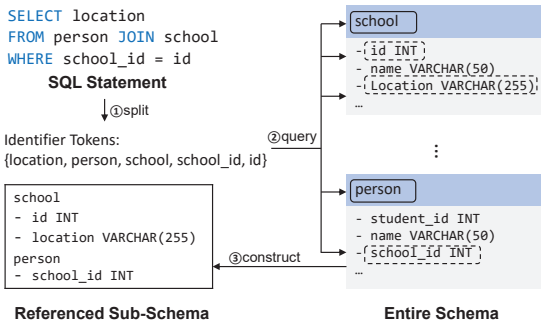


Figure 3: The workflow of schema capture.

Figure 3 illustrates an example of the workflow of schema capture. Upon the execution of a statement, SEDAR first splits the statement into identifier tokens, namely "location", "person", "school", "school_id", and "id". Next, SEDAR queries the system table to locate the corresponding database objects, and the result of the query reveals that "school" and "person" are table names, while "id", "location", and "school_id" are column names. Finally, SEDAR constructs the sub-schema referenced by the statement according to the query result. After the above steps are finished, we obtain the SQL statements in test cases paired with their corresponding sub-schemas. The schema information of each statement will be used to describe the statement, which is introduced in the next component of SEDAR.

3.2 SQL Transfer with LLM

SQL transfer aims to transform the SQL statements in test cases from popular DBMSs into statements that match the grammar of the target DBMSs via LLMs. First, SEDAR generates prompts for the LLMs based on the SQL statements and their corresponding schema information for the transfer task. Next, it queries the LLMs and processes the responses into the test cases that are compatible with the target DBMS.

Prompt Generation. The inputs given to LLMs, known as prompts, determine their outputs. To drive LLMs to process the transferring task, it is required to design suitable prompts (i.e., prompt engineering). For the purpose of transferring the test cases of popular DBMS to target DBMS, a straightforward design of the prompt is to submit the entire test cases to LLMs. However, the test cases can be very large, potentially including thousands of SQL statements. Due to the limitations of LLMs in handling large inputs and generating extensive outputs within a single prompt, SEDAR adopts a different approach. It generates individual prompts for each statement in test cases, addressing the task of transferring a single statement per prompt. This allows for more efficient processing and ensures better compatibility with the LLMs.w

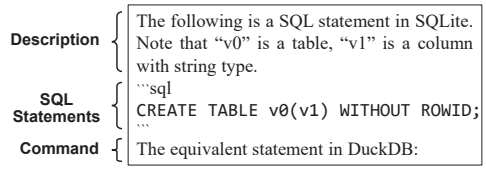


Figure 4: An example of the prompt to transfer a statement from SQLite grammar to DuckDB grammar.

Figure 4 shows a sample prompt for the transfer of a single statement. The prompt contains the following parts: ① The description of the statement waiting to transfer, which includes the introduction of the sub-schema referenced by the statement. It provides the context of the statement, which enables the LLM to generate accurate output without requiring knowledge of entire test cases. ② The content of the statement. It is enclosed within a markdown code block with the 'sql' hint. ③ The command part, namely "The equivalent SQL statement in DuckDB". It is to drive the LLMs to transfer the given statement into the one matching DuckDB grammar.

In the prompt, both the SQL statement part and the command part can be generated directly from the content of the SQL statement and the name of the target DBMS. For the description part, SEDAR needs to generate a natural language summary of the sub-schema captured by the schema capture component. It is achieved

by converting each item in the sub-schema into a sentence with the format "{item_name} is a {item_type}", and concatenating these sentences to form the description part.

LLM Query. Upon the generation of prompts, SEDAR enumerates the prompts and queries the LLMs. Since every prompt already contains the context of the statement, the order of these prompts during querying does not impact the process. Thus, for LLMs with parallel support, SEDAR can concurrently query multiple prompts for speeding up. Upon these prompts are sent to LLMs, the LLMs generate responses that contain the transferred SQL statements. For LLMs that have been pretrained with data containing the knowledge of target DBMS, they can directly provide the result of transferring. Otherwise, the LLMs require fine-tuning with the documentation and tutorials of target DBMS to enable them to transfer SQL statements properly. Finally, when receiving the responses from LLMs, SEDAR extracts the transferred statements and combines them in their original sequences to create new test cases. These new test cases are called *compatible seeds* since they should now be compatible with the grammar of the target DBMS.

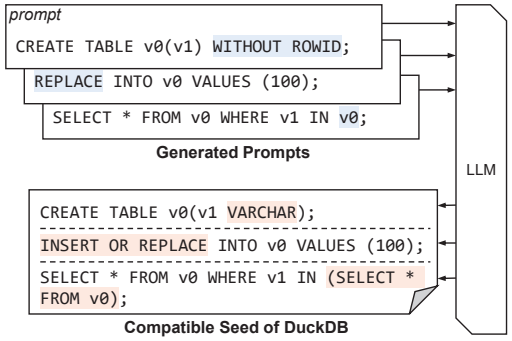


Figure 5: An example of a compatible seed of DuckDB. It is combined with the responses of LLM driven by prompts.

Figure 5 illustrates the transfer from a SQLite test case into a compatible seed for DuckDB. SEDAR first generates three prompts for the three statements of the test case, similar to the example shown in Figure 4. Then, SEDAR queries the LLM with these prompts in parallel. The responses of LLM present the following modifications to the statements: ① The clause `WITHOUT ROWID` in the table creation statement is removed, as DuckDB does not support this feature. ② The data type definition is added since the column types of table creation in DuckDB require explicit definitions. ③ The `REPLACE` keyword and the `IN` clause are replaced into the equivalent grammar of DuckDB. Finally, SEDAR joins these statements together to form the compatible seed of DuckDB.

3.3 Mutability Refinement

Although the compatible seeds are suitable to the grammar of the target DBMS, they are currently not mutable for the mutation-based DBMS fuzzer. During the mutation process, the DBMS fuzzer's mutator is required to parse the SQL statements in the seed, analyze their syntactic structures (e.g., the IRs in SQUIRREL), modify the structures, and convert the modified structures back to SQL statements to form a new test case. When the seeds contain the grammar supported by the target DBMS but not by the mutator's parser, the mutator is unable to parse them, impeding further mutation.

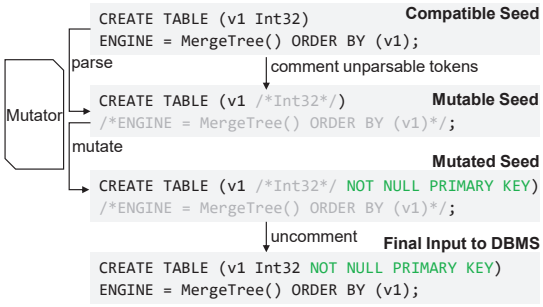


Figure 6: The workflow of mutability refinement. SEDAR generates mutable seeds for the mutators of fuzzers by commenting out the unparsable sections in compatible seeds. These comments for unparsable sections will be uncommented after the mutation finishes.

For instance, SQUIRREL is unable to mutate the seed of ClickHouse as shown in Figure 6. In ClickHouse, a `CREATE TABLE` statement always specifies the storage engine, such as `MergeTree` and `Memory`. However, SQUIRREL lacks support for the grammar of engine declaration clauses. If such statements are fed to SQUIRREL for mutation, the SQUIRREL's mutator will fail to parse the statements and simply skip the entire seed. As a result, SQUIRREL cannot mutate any ClickHouse seed that contains `CREATE TABLE` statements, which limits SQUIRREL's performance on ClickHouse.

To make the mutator able to parse and mutate the seed, SEDAR tries to extract the parsable part of the seed with Algorithm 1. Specifically, SEDAR first breaks the content of the seed into a list of tokens, and then these tokens are sequentially fed to the mutator's parser. (lines 2-6). When a token encounters an error of the parser (e.g., no rules can handle the token appended), SEDAR marks this token as an unparsable section, recovers the parser from its error state, and continues with the next token (lines 7-14). After all of the tokens have been processed, SEDAR comments out these unparsable sections, instructing the parser to ignore them (line 15). The "/*" and "*/" strings are inserted around these unparsable sections of the seed, which is a common format of SQL comments. Upon commenting out the unparsable sections, we refer to the seed as a *mutable seed*, since it can now be parsed for further mutation.

These mutable seeds are subsequently delivered to the DBMS fuzzer's mutator to mutate, which introduces a variety of mutated structures within the parsable sections (line 16). Next, SEDAR uncomments the unparsable sections in these seeds to restore the unsupported grammar (lines 17-19). The seeds thus consist of both the mutated parsable part and uncommented unparsable sections of the seed, which can trigger new behaviors in the target DBMS.

Figure 6 is an example to show how SEDAR refines the mutability of a compatible seed of ClickHouse. First, SEDAR uses the mutator of SQUIRREL to parse the statement in the seed and identify two unparsable sections: the Int32 data type and the engine clause. Second, SEDAR comments out these unparsable sections, thereby forming a mutable seed that contains only the `CREATE TABLE` keyword and

Algorithm 1: Mutate the unparsable seed

```
Input  : the mutator of DBMS fuzzer: M,
         the seed containing unparsable parts: S.
Output : The mutations of the unparsable seed.
1  begin
2      P ← GetParserOf(M);
3      tokens ← SplitIntoTokens(S);
4      state ← InitStateOfParser(P);
5      unparsables ← an empty list;
6      foreach t in tokens do
7          prevState ← state;
8          state ← Parse(t, P, state);
9          if IsErrorState(nextState) then
10             Append(unparsables, t);
11             state ← prevState;
12         end
13     end
14     S* ← S;
15     foreach t in unparsables do S* ← Comment(S*, t);
       // S* here is called the mutable seed.
16     mutated ← MutateTestCase(S*, M);
17     foreach m in mutated do
18         foreach t in unparsables do m ← UnComment(m, t);
19     end
20     return mutated;
21 end
```

the column name "v1". Third, Squirrel mutates the mutable seed into a statement with the keywords `NOT NULL` and `PRIMARY KEY`. Last, the unparsable sections are uncommented to get the final seed. The mutated seed is still valid for ClickHouse and contains new keywords produced by the mutation to trigger new behaviors.

4 IMPLEMENTATION

As Figure 2 shows, Sedar consists of three components with respect to three steps: schema capture, SQL transfer, and mutability refinement. We implemented the three components with 3,213 lines of C++ code and 1,072 lines of Java code.

For the schema capture component, we modified the source code of the DBMS clients [23, 27, 39] to capture the referenced sub-schema whenever a statement is executed. Once the collections of the statements and their schema information of a DBMS's test cases are finished, these results are reusable when transferring from the DBMS to multiple DBMSs. For the SQL transfer component, we employed the gpt-3.5-turbo-0301 model as the LLM to transfer SQL statements. The prompt generation is implemented in Java, and the interaction with gpt-3.5-turbo-0301 is run via its online Applications Program Interface (API). For the mutability refinement component, its implementation depends on the parser of the DBMS fuzzer. For example, Squirrel's parser is written with Lex and Yacc [18] code. We implemented the refinement of mutability by adding the error recovery rules [11] to Squirrel's parser, which enables it to automatically record and skip unparsable tokens.

5 EVALUATION

We evaluate Sedar with the improvement of code coverage and bug detection after employing Sedar to existing DBMS fuzzers, as well as the efficiency of the seeds generated by Sedar in DBMS fuzzing. Our evaluation aims to answer the following research questions:

- **RQ1:** With Sedar employed, can DBMS fuzzers find new vulnerabilities in real-world DBMSs?
- **RQ2:** How is the improvement of Sedar for the performance of mutated-based fuzzers on new DBMS?
- **RQ3:** What is the contribution of SQL transferring on the initial seeds for fuzzing?
- **RQ4:** What is the effectiveness of refining mutability to DBMS fuzzers?

5.1 Evaluation Setup

Tested DBMSs. We selected four popular DBMSs for the evaluation, including MonetDB [3], Virtuoso [36], DuckDB [6], and ClickHouse [10]. These DBMSs are all widely used open-source DBMSs according to DB-Engine Ranking [12, 13]. They support similar DBMS features (e.g. the relational data model) but use different SQL dialects. The versions under evaluation are MonetDB v11.46.0, Virtuoso-opensource v7.2.9, DuckDB v0.7.1, and ClickHouse v23.5.3.24.

DBMS Fuzzers. To evaluate the quality of seeds generated by Sedar, we employed two mutation-based DBMS fuzzers, Griffin and Squirrel. Specifically, Griffin applies mutation by reshuffling statements and metadata-guided substitutions, while Squirrel performs syntax-preserved mutations and semantic-guided instantiations on SQL test cases. Both Squirrel and Griffin rely on the initial seeds for DBMS fuzzing and aim to detect crash bugs.

Table 1: Information on collected open-source test cases.

	SQLite	MySQL	PostgreSQL
Number of Test Cases	2,643	2,685	1,367
Number of Statements	308,292	98,862	19,814
Total Size of Test Cases	31 MiB	15 MiB	8.4 MiB

Collected Test Cases. As Sedar aims to produce high-quality initial seeds for target DBMS by transferring test cases from other DBMSs, we collected built-in test cases from other DBMS repositories [23, 27, 39] for SQL transfer, including SQLite, MySQL, and PostgreSQL. Table 1 shows the information on collected test cases in each DBMS. We also collected the native open-source test cases from the repositories of MonetDB, DuckDB, and ClickHouse for comparison. Virtuoso does not offer open-source test cases.

Basic Setups. We performed the evaluation on a 128-core AMD EPYC 7742 Processor @ 2.25 GHz machine with 504 GiB of main memory running 64-bit Ubuntu 20.04. The DBMSs to test are compiled by AFL++ [8] for coverage feedback. We ran the DBMS fuzzers with their default configurations and different sets of initial seeds, and utilized ptrace to monitor whether the DBMSs crashed during fuzzing. Each fuzzing instance ran for 24 hours with 2 CPUs on testing target DBMSs.

5.2 DBMS Vulnerabilities

With Sedar employed, Squirrel and Griffin have found 70 unknown bugs in 3 days, and 60 of them were uniquely discovered by Sedar with transferred seeds. Table 2 lists the details of these discovered bugs. Specifically, Sedar found 31, 25, 6, and 8 bugs in MonetDB, Virtuoso, DuckDB, and ClickHouse, respectively. These new bugs found by Sedar have various types, including 33 null

Table 2: List of previously-unknown bugs detected by SQUIRREL and GRIFFIN with SEDAR employed, 70 confirmed with 19 CVEs assigned.
[**UAF**: Use-After-Free, **SO**: Stack Overflow, **HBOF**: Heap Buffer Overflow, **SBOF**: Stack Buffer Overflow, **AF**: Assertion Failure, **NPD**: Null Pointer Dereference, **DBZ**: Divide-by-Zero, **SEGV**: Segmentation Violation]

Fixed/Detected	Component	Bug Type
MonetDB (19/31)	storage	UAF(3), NPD(3)
	relation	NPD(10), SO(1)
	select	NPD(5)
	optimizer	NPD(1)
	mal	SBOF(1), HBOF(1), NPD(2)
	gdk	DBZ(1), SEGV(1), UAF(1), HBOF(1)
Virtuoso (25/25)	sqlstmts	SEGV(4), NPD(2), HBOF(1)
	sql	DBZ(2), UAF(1), NPD(5), SEGV(2), HBOF(5)
	parser	SEGV(1), NPD(1)
	chash	SEGV(1)
DuckDB (5/6)	catalog	UAF(1)
	storage	NPD(1), AF(1)
	transaction	AF(1)
	odbc	SEGV(1)
	optimizer	SO(1)
ClickHouse (8/8)	function	NPD(2), SEGV(1)
	interpreter	AF(1)
	processor	DBZ(1), NPD(1), SEGV(1)
	common	SO(1)
Total	19 components	57 fixed, 70 confirmed

pointer dereferences, 6 use-after-free, 3 stack overflows, 8 heap buffer overflows, 1 stack buffer overflow, 4 divide-by-zero, 12 segmentation violations, and 3 assertion failures. All of these bugs are confirmed by the vendors, and 57 of them are already fixed. In addition, 19 of them are assigned CVE IDs.

Case Study: the "interval" operator in ClickHouse. SEDAR detected a null pointer dereference bug in ClickHouse, triggered by an insertion statement with the INTERVAL operator. Ordinarily, in SQL grammar, the INTERVAL operator is for the calculation of date and timestamp values. ClickHouse supports a non-standard INTERNAL grammar, but such a feature can induce a crash bug when executing the INSERT statement shown in Figure 7. In ClickHouse, The expression "INTERVAL '1 day 1 hour 1 minute 1 second"' was processed as a tuple with four values, corresponding to "(toIntervalDay(1), toIntervalHour(1), toIntervalMinute(1), toIntervalSecond(1))". However, the insertion only accepts exactly one value per column with the DATE type. The mismatch of the number of values leads to a null pointer dereference of ClickHouse during the processing of the insertion statement.

The Reason for Finding the Bug by SEDAR. ClickHouse lacks testing of the non-standard interval expression within insertion statements, while other DBMSs have tested similar features. In MySQL, it supports string literals like "1 01:01:01" to denote time intervals and the "ADDTIME" function to add a date and a time interval. MySQL includes a test case testing this feature in an insertion statement (shown in blue in Figure 7). Nonetheless, such a test case in MySQL is invalid for ClickHouse due to the differences in the time interval grammar. Executing the MySQL's INSERT statement on ClickHouse will result in a "Cannot parse expression of type Date" error, failing to reveal the crash bug. However, SEDAR provides the valid time interval expression by transferring the statement in MySQL to ClickHouse grammar (shown in red in Figure 7).

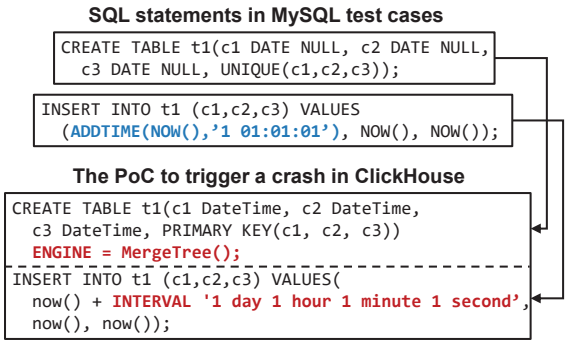


Figure 7: Case study: a crash bug in ClickHouse when processing the INTERVAL clause found by SEDAR.

Similarly, SEDAR also transfers a table creation statement to ClickHouse's grammar. Thus, with initial seeds containing the specific INTERNAL expression in insertion statements and a correct table creation statement of ClickHouse, a fuzzing instance succeeded in finding the bug in several hours by combining these two statements.

5.3 Overall Experiments

To evaluate the improvement of DBMS fuzzers by SEDAR, we performed GRIFFIN and SQUIRREL on DBMSs, and configured them with five distinct sets of initial seeds on each DBMS: the first one is the built-in initial seeds of the default configurations of fuzzers, labeled as GRIFFIN and SQUIRREL in the evaluation; the second one is the test cases we collected from other popular DBMSs without transferring, labeled as GRIFFIN^P and SQUIRREL^P; the third one is the seeds generated by SEDAR, labeled as SEDAR-GRIFFIN and SEDAR-SQUIRREL; the fourth one is the native test cases of the DBMSs to test, with the mutability refinement of SEDAR applied, labeled as SEDAR-GRIFFIN^{N-} and SEDAR-SQUIRREL^{N-}; the last one is the combination of the seeds generated by SEDAR and their native test cases, labeled as SEDAR-GRIFFIN^N and SEDAR-SQUIRREL^N. It should be noted that the fuzzers are not able to directly parse or mutate their native test cases, so we apply the mutability refinement to the native test cases in the experiment.

We measured their effectiveness with the number of branches and the count of detected bugs. All the branch coverage was captured by the SanitizerCoverage [40] of LLVM for a fair comparison. We additionally repeated the experiment 5 times and computed the p-values for statistical tests.

Code Coverage. Figure 8 presents the number of covered code branches on DBMSs by DBMS fuzzers when using different test cases as initial seeds. It illustrated that DBMS fuzzers deployed with SEDAR outperform the fuzzers with non-transferred seeds in terms of branch coverage. Specifically, SEDAR-GRIFFIN covered more branches by 21.40%-40.58%, 90.82%-126.87%, 62.05%-74.26%, and 172.99%-194.46% on MonetDB, Virtuoso, DuckDB, and ClickHouse, respectively, compared to GRIFFIN and GRIFFIN^P. Similarly, SEDAR-SQUIRREL enhanced branch coverage by 160.30%-195.45%, 107.68%-111.00%, 81.71%-136.20%, and 72.46%-214.84% on these DBMSs compared with SQUIRREL and SQUIRREL^P.

The enhancements in branch coverage can be attributed to two primary reasons. On one hand, the seeds for SEDAR-GRIFFIN and

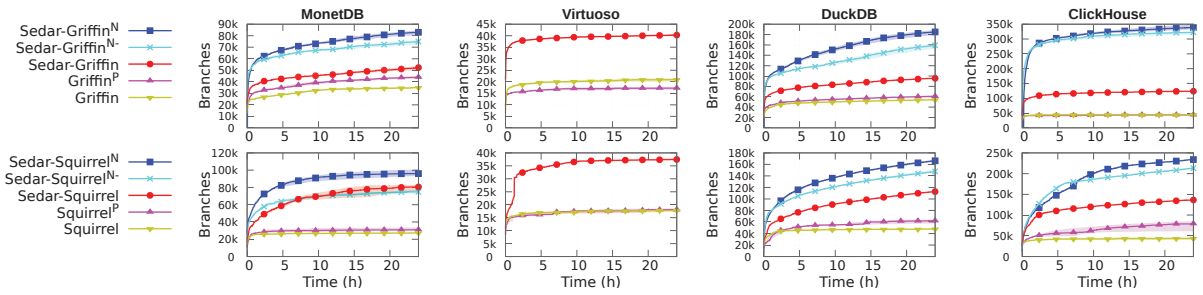


Figure 8: The branch coverage of DBMS fuzzers on MonetDB, Virtuoso, DuckDB, and ClickHouse. Displayed are the medians and the 95% confidence intervals of 5 repeated times.

SEDAR-SQUIRREL have high qualities for the target DBMSs. The GRIFFIN, GRIFFIN^P, SQUIRREL, and SQUIRREL^P only use the default seeds of fuzzers or the test cases of other DBMSs as the initial seeds, which are currently incompatible with the grammar of target DBMSs. As a result, the new test cases generated by SQUIRREL and GRIFFIN from these seeds do not satisfy the grammar of target DBMSs, limiting their code coverage on the four DBMSs. Instead, the initial seeds processed by SEDAR match the grammar of target DBMSs, which helped fuzzers cover more branches. The other reason is that the mutability refinement of SEDAR helps enhance the correctness of SQL statements during fuzzing. Without the mutability refinement, the SQL statements generated by SQUIRREL and GRIFFIN contain syntax errors for new target DBMS, limiting the fuzzers exploring DBMS behaviors. In contrast, SEDAR employs the mutability refinement, which helps DBMS fuzzers generate correct SQL statements and thereby deeply test the DBMSs. Consequently, with the compatible initial seeds and the mutability refinement component, SEDAR enables the fuzzers to generate more syntactically and semantically correct statements, and cover more deep logic and code regions in target DBMSs by these statements.

It can also be observed from Figure 8 that SEDAR-GRIFFIN^{N-} and SEDAR-SQUIRREL^{N-} achieved substantially higher code coverage compared to the ones without native test cases in their initial seeds. This is because the native seeds, as maintained by their developers, have already covered a wide range of features and functionality of DBMSs. The mutability refinement also helps the fuzzers to adapt these seeds and finally cover more code branches. Furthermore, when introducing the transferred seeds into initial seeds additionally, SEDAR-GRIFFIN^N and SEDAR-SQUIRREL^N cover more branches than SEDAR-GRIFFIN^{N-} and SEDAR-SQUIRREL^{N-} by 10.77%-28.41%, 12.53%-16.20%, and 4.90%-9.73% for MonetDB, DuckDB, and ClickHouse, respectively. The reason is that these transferred seeds bring valuable test cases from other DBMSs to the target DBMSs, such as the edge cases of DBMS features like the example in Figure 7. They help SEDAR-GRIFFIN^N and SEDAR-SQUIRREL^N to trigger more code regions than using only native test cases.

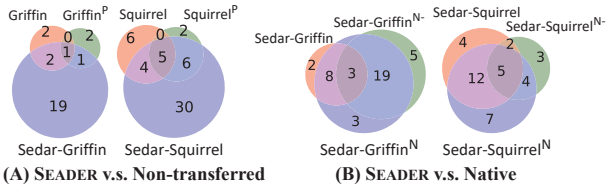
Bugs. Table 3 exhibits the bugs detected by fuzzers with different seeds after 24 hours. Compared to the fuzzers using non-transferred seeds, We can see that SEDAR improves the bug detection ability of GRIFFIN and SQUIRREL on the four DBMSs. With SEDAR employed, SEDAR-GRIFFIN found a total of 18 and 19 more bugs in the four DBMSs than GRIFFIN and GRIFFIN^P, and SEDAR-SQUIRREL found 30

Table 3: Number of bugs detected by DBMS fuzzers in 24 hours. Shown are the medians of 5 runs.

Fuzzer	MonetDB	Virtuoso	DuckDB	ClickHouse
GRIFFIN	2	2	1	0
GRIFFIN^P	3	0	1	0
SEDAR-GRIFFIN	10	10	2	1
SEDAR-GRIFFIN^{N-}	27	-	0	0
SEDAR-GRIFFIN^N	30	-	2	1
SQUIRREL	9	6	0	0
SQUIRREL^P	5	8	0	0
SEDAR-SQUIRREL	18	22	3	2
SEDAR-SQUIRREL^{N-}	13	-	0	1
SEDAR-SQUIRREL^N	23	-	3	2

and 32 more bugs than SQUIRREL and SQUIRREL^P. The relationships between these bugs are shown in Figure 9 (A). With different initial seeds, the GRIFFIN series uncovered 27 bugs in total, with 19 of them being detected only when SEDAR is employed. Similarly, the SQUIRREL series discovered 53 bugs, with 30 of them uniquely found by SEDAR-SQUIRREL. It shows that with the initial seeds of SEDAR, the fuzzers GRIFFIN and SQUIRREL can detect more bugs compared to their previous performance. We also noticed that several bugs were only found with non-transferred seeds. It is because few of the incompatible test cases in these seeds crashed the DBMSs.

Furthermore, compared to the fuzzers utilizing the native test cases as seeds, SEDAR can help fuzzers to find new bugs in MonetDB, DuckDB, and ClickHouse. As Figure 9 (B) shows, the GRIFFIN series and SQUIRREL series found 40 and 37 bugs in the three DBMSs in total. Out of these, 13 bugs of GRIFFIN series and 23 bugs of SQUIRREL series were detected only when the transferred seeds of SEDAR were integrated into the initial seeds. Additionally, we can see that SEDAR-GRIFFIN^N, which combines transferred seeds and native test cases as initial seeds, found 20 and 6 more



(A) SEADER v.s. Non-transferred **(B) SEADER v.s. Native**

Figure 9: Relationships between the bugs in Table 3.

bugs than SEDAR-GRIFFIN and SEDAR-GRIFFIN^{N-}. Similarly, SEDAR-SQUIRREL^{N} found 6 and 14 more bugs than the other two. We also noticed that SEDAR-GRIFFIN^{N-} and SEDAR-SQUIRREL^{N-} only discovered one bug in DuckDB and ClickHouse in total, although they achieved high branch coverage. It is because DuckDB and ClickHouse have already supported multiple fuzzers [4, 5] and have been fuzzed with their native test cases for a long time, which makes it hard to discover new bugs with these test cases only. In contrast, the transferred seeds of SEDAR provide valuable test cases from other DBMSs, such as the edge cases that are not considered by the native ones. They help fuzzers to detect new bugs in DuckDB and ClickHouse.

P-value. The experiment is repeated 5 times for statistical tests. From the p-values displayed in Table 4, we can infer that SEDAR-GRIFFIN, SEDAR-SQUIRREL, SEDAR-GRIFFIN^{N}, and SEDAR-SQUIRREL^{N} achieved significantly higher branch coverage and detected a significantly larger number of bugs.

Table 4: P-values for SEDAR-GRIFFIN and SEDAR-SQUIRREL vs GRIFFIN, GRIFFIN^{P}, SQUIRREL, and SQUIRREL^{P}, and p-values for SEDAR-GRIFFIN^{N} and SEDAR-SQUIRREL^{N} vs SEDAR-GRIFFIN^{N-} and SEDAR-SQUIRREL^{N-}. All p-values are statistically significant ($p < 0.05$).

Fuzzer	MonetDB		Virtuoso		DuckDB		ClickHouse	
	Branch	Bug	Branch	Bug	Branch	Bug	Branch	Bug
GRIFFIN	**0.0079**	**0.0109**	**0.0079**	**0.0106**	**0.0079**	**0.0056**	**0.0079**	**0.0040**
GRIFFIN^{P}	**0.0079**	**0.0117**	**0.0079**	**0.0099**	**0.0079**	**0.0056**	**0.0079**	**0.0040**
SQUIRREL	**0.0079**	**0.0112**	**0.0079**	**0.0119**	**0.0079**	**0.0073**	**0.0079**	**0.0065**
SQUIRREL^{P}	**0.0079**	**0.0117**	**0.0079**	**0.0119**	**0.0079**	**0.0073**	**0.0079**	**0.0065**
SEDAR-GRIFFIN^{N-}	**0.0079**	**0.0278**	-	-	**0.0079**	**0.0065**	**0.0079**	**0.0086**
SEDAR-SQUIRREL^{N-}	**0.0079**	**0.0116**	-	-	**0.0079**	**0.0086**	**0.0079**	**0.0200**

5.4 Contribution of SQL Transfer

To understand the advantage of the compatible seeds generated by SQL transfer of SEDAR, we compared the original test cases collected from other popular DBMSs, labeled as S^{-}, and the compatible seeds generated by SEDAR with SQL transfer, labeled as S. The compatible seeds S consist of the statements generated by LLM of the SQL transfer component. We evaluated the two sets of seeds in terms of syntactic correctness, semantic correctness, and code coverage.

Table 5 shows the syntactic correctness ratios and semantic correctness ratios, as well as the number of covered branches between S^{-} and S on four DBMSs. From the table, we can see that S contains 27%, 66%, 29%, and 29% more syntactic-correct statements, includes 56%, 64%, 24%, and 62% more semantic-correct statements, and covers 13%, 66%, 13%, 99% more code branches than S^{-} in MonetDB, Virtuoso, DuckDB, and ClickHouse, respectively. It indicates that the SQL transfer with LLMs can convert some invalid SQL statements into valid ones for the target DBMS. These transferred statements achieve higher code coverage since more valid statements can trigger more code logic of DBMS.

We can also notice the high improvements in code coverage of Virtuoso and ClickHouse. Virtuoso, which is a multi-model DBMS primarily focusing on the graph data model, offers only basic support to SQL grammar for the relational model. This limitation explains why S^{-}, the collected SQL test cases from other DBMSs, has a low syntactic correctness ratio of only 0.32 on Virtuoso. By comparison, within the compatible seeds S, some statements with unsupported grammar are transferred into the equivalent expressions of basic SQL grammar that Virtuoso does support. Some are even transferred into the SPARQL [26] dialect for its graph data model. Consequently, S achieves a syntactic correctness ratio of 0.53 and covers 66% more code branches than S^{-}.

Table 5: The syntactic correctness ratios, the semantic correctness ratios, and branch coverage of the test cases of other DBMSs S^{-} and the compatible seeds S of SEDAR.

DBMS	Syntactic Correctness Ratios		Semantic Correctness Ratios		Branch Coverage	
	S^{-}	S	S^{-}	S	S^{-}	S
MonetDB	0.589	0.754 (+27%)	0.236	0.370 (+56%)	34,189	38,493 (+13%)
Virtuoso	0.324	0.539 (+66%)	0.097	0.160 (+64%)	22,176	36,915 (+66%)
DuckDB	0.674	0.871 (+29%)	0.284	0.352 (+24%)	61,356	69,217 (+13%)
ClickHouse	0.613	0.793 (+29%)	0.182	0.297 (+62%)	56,497	112,494 (+99%)

For ClickHouse, the improvement partially comes from `CREATE TABLE` statements. ClickHouse requires an explicit specification of the storage engine when creating a table. However, in S^{-}, most table creation statements lack the engine declaration, leading to execution failures when ClickHouse runs test cases containing these statements. Moreover, it impacts all subsequent `INSERT` and `SELECT` statements that reference the table, resulting in a low semantic correctness ratio and branch coverage. In contrast, the table creation statements in S successfully include the engine declaration benefiting from SQL transfer. It enables table creation statements in S to be valid and executable in ClickHouse, as well as the corresponding `INSERT` and `SELECT` statements, leading to higher code coverage.

Table 6: The correct syntactic structures in the semantically correct statements of S^{-} and S.

DBMS	Function		Data Type		Keyword	
	S^{-}	S	S^{-}	S	S^{-}	S
MonetDB	107	**132**	29	29	235	**244**
Virtuoso	28	**52**	16	**17**	102	**136**
DuckDB	185	**214**	34	**39**	291	**340**
ClickHouse	83	**227**	36	**61**	124	**152**
Total	403	**625**	115	**146**	752	**872**
Increment	222	-	31	-	120	-

Furthermore, we examined the syntactic structures contained by the semantically correct statements of S^{-} and S in the four DBMSs, focusing on the functions, data types, and keywords supported by these DBMSs. From the result shown in Table 6, we can see that S contains 222 more functions, 31 more data types, and 120 more keywords in total than S^{-}. This improvement allows the compatible seeds S to achieve higher correctness ratios and code branches in the four DBMSs.

5.5 Effectiveness of Mutability Refinement

To measure the effectiveness of the mutability refinement, we performed SQUIRREL on four DBMSs with the compatible seeds generated just by the SQL transfer component of SEDAR, labeled as S, and the mutable seeds generated by SQL transfer and mutability refinement, labeled as S^{+}. We recorded the number of successfully parsed and mutated statements by SQUIRREL with the compatible seeds S and the mutable seeds S^{+} as the metric. We also ran SQUIRREL for 24 hours for the comparison of branch coverage.

Table 7: The number of statements in compatible seeds and mutable seeds that can be parsed by SQUIRREL, and the branch coverage achieved by SQUIRREL with these seeds.

DBMS	Parsable Statements		Branch Coverage	
	S	S^+	S	S^+
MonetDB	166,410	239,495	61,981	77,194
Virtuoso	106,118	200,209	24,017	37,191
DuckDB	125,655	201,997	96,513	116,638
ClickHouse	149,843	230,766	87,389	143,368
Total	548,026	872,467	269,900	374,391
Increment	324,441	-	104,491	-

Table 7 presents the statistical results of four tested DBMSs. Compared to S, the numbers of parsable statements in S^+ increased by 44%, 89%, 61%, and 54% on MonetDB, Virtuoso, DuckDB, and ClickHouse, respectively. The improvement is mainly attributed to the mutability refinement, which helps DBMS fuzzers mutate the grammar-incompatible SQL statements. Since S is generated by LLM for SQL transfer, it includes the unique grammar of target DBMS, such as unique keywords and clauses, which prevents SQUIRREL from parsing and mutating these seeds. By contrast, in S^+, these unparsable parts with unique grammar are temporarily commented out. It allows SQUIRREL to parse the remaining parsable parts and mutate them to generate more test cases. Thus, with more seeds able to be parsed and mutated, the mutated seeds are expected to trigger more behaviors in DBMSs and cover more code branches. Table 7 shows the code coverage after running SQUIRREL with S and S^+ respectively. From the table, we can see that SQUIRREL with S^+ achieves 25%, 55%, 21%, and 64% higher code coverage on four DBMSs than S. The result indicates that the mutability refinement component of SEDAR can improve the performance by enabling it to parse and mutate the test cases containing unique grammar, which could generate more semantically correct SQL statements and thereby cover more code branches.

6 DISCUSSION

In this section, we discuss several limitations of our implementation of SEDAR and our plan to address them in future work.

The Limited Syntactic and Semantic Correctness Ratios of Transferred Seeds. Although there is a noticeable improvement in the syntactic and semantic correctness of transferred seeds compared to original ones as indicated in Table 5, the absolute values of the correctness ratios are still low. There are several reasons. ① One is the different support of DBMS features. When the target DBMS does not support similar features in the original statement, such as unsupported data types and functions, the SQL transfer will produce a non-executable statement on target DBMS. For example, Virtuoso is primarily a graph database, providing only basic support for SQL grammar and lacking similar advanced features in popular DBMSs. Thus, many statements in popular DBMSs do not have equivalent ones in Virtuoso, and it causes significantly lower correctness ratios for Virtuoso than the other DBMSs. ② The second is the hallucinations of LLMs. LLMs sometimes output inaccurate or incorrect results, leading to the SQL transfer generating incorrect statements for target DBMSs. ③ Furthermore, we can also notice that the semantic correctness ratios are notably lower than syntactic correctness ratios. This is because SQL statements within a test case might have contextual dependencies. A single incorrectly transferred statement will affect the subsequent statements that depend on it, rendering them semantically incorrect as well.

A way to improve these correctness ratios is to update the SQL transfer process with real-time feedback from target DBMSs and regeneration. For example, upon the transfer of a SQL statement by the LLMs, we can immediately execute the transferred statement on the target DBMS. If there is an execution failure, we can feed the error message back to the LLMs to generate an amended version. We will implement this feedback component in future work.

Table 8: The transfer results of other LLMs on the SQL transfer cases in Figure 5 and Figure 7.

Model	Figure 5 create table	Figure 5 replace	Figure 5 select	Figure 7 create table	Figure 7 insert
`Llama-2-7B`	✗	✗	✗	✗	✗
`Llama-2-13B`	✗	✓	✓	✗	✗
`Llama-2-70B`	✓	✓	✓	✗	✓
`ChatGLM2-6B`	✗	✓	✗	✗	✗
`ChatGLM2-130B`	✓	✓	✓	✓	✓
`CodeLlama-7B`	✓	✓	✓	✗	✓
`CodeLlama-13B`	✓	✓	✓	✓	✓
`CodeLlama-34B`	✓	✓	✓	✓	✓

✓: transferred correctly; ✗: transferred wrongly.

External Validity. In the implementation and evaluation of LLMs, we only employed gpt-3.5-turbo-0301 in the SQL transfer component of SEDAR, leading to external validity. To demonstrate the ability of other LLMs in SQL transfer, we tested the transfer examples in Figure 5 and Figure 7 on several state-of-the-art LLMs including `Llama 2` [41], `ChatGLM2` [35, 47], and `CodeLlama` [33], with the results listed in Table 8. As illustrated, the LLMs fine-tuned for code tasks (e.g. `CodeLlama`) and the LLMs with more parameters (e.g. `ChatGLM2-130B`) produce better results. We believe that SEDAR can perform well with these models on the SQL transfer task. Moreover, to enhance the performance, we can fine-tune the LLMs with the documentation, tutorials, and native test cases of the target DBMSs, especially for new DBMS features or newly developed DBMSs.

7 RELATED WORK

Mutation-Based DBMS Fuzzing. The mutation-based fuzzing techniques [2, 9, 14, 19, 20, 44, 45, 48] have been successfully applied to DBMSs for effective testing and bug finding. SQUIRREL [48] mutates SQL statements by translating SQL statements into the Intermediate Representation (IR) for SQL, mutating the IRs, and re-translating the mutated IRs into syntactically and semantically correct SQL statements. LEGO [19] proposes the type affinity of SQL sequences and employs it to guide the generation of SQL statements with abundant SQL type sequences. GRIFFIN [9] utilizes a grammar-free way to mutate SQL statements with statement reshuffle and metadata-based substitution. SQLRight [20] employs test oracles to mutation-based fuzzers to detect logic bugs. While the above are all coverage-guided approaches, QPG [2] proposes the concept of query plan guidance to guide the fuzzing other than code coverage.

Generation-Based DBMS Testing. Some generation-based approaches [15, 22, 30–32, 34, 37] are proposed to find specific types of DBMS bugs. SQLsmith [34] generates `SELECT` statements with

complex structures from a predefined Abstract Syntax Tree (AST) to find crash bugs. NoREC [30] is designed to detect optimization bugs by generating SQL queries as well as the equivalent ones that cannot be optimized. PQS [32] generates queries that should fetch a specific row, and indicates a bug triggered when the row is not included in result sets. DQE [37] focuses on logic bugs in `UPDATE` and `DELETE` queries by verifying that the same rows are accessed with the same predicate. AMOEBA [22] discovers performance bugs by generating equivalent SQL queries and comparing their execution times.

Initial Seed Generation. Several existing works [21, 25, 43, 46] concentrate on generating high-quality seeds for general-purpose mutation-based fuzzing. SAFL [43] employs symbolic execution to generate qualified initial seeds that provide valuable exploration directions. SLF [46] generates valid seeds by automatically identifying input validity checks of programs and input fields related to these checks. TENSILEFUZZ [21] further denotes the input fields as string variables and generates seeds through string constraint solving. While the seed generation works gain good effectiveness on general-purpose fuzzing, they are not suitable for DBMS fuzzing due to the complexity of SQL grammar. Other works can provide valid initial seeds for specific fuzzing targets. Moonshine [25] distills seeds for Operating System (OS) fuzzers from real-world system call traces by analyzing their dependencies. Skyfire [42] learns a probabilistic context-sensitive grammar (PCSG) from existing seeds and grammars, and generates well-distributed initial seeds.

Main Differences. SEDAR is a tool to provide high-quality initial seeds via cross-DBMS SQL transfer. Its seeds could be utilized for popular DBMS fuzzers. Even for DBMSs that are not supported by them in grammar, SEDAR can still utilize the mutability refinement component to extend these fuzzers to complete the testing process. The generation-based tools typically generate SQLs based on predefined rules and may only encompass a limited set of features. In contrast, SEDAR transfers SQLs from well-tested DBMS's built-in test cases. These test cases encompass the rich features of these popular databases, which are conducive to discovering bugs. As a result, SEDAR found bugs in 19 components of four DBMSs shown in Table 2, including components such as parsers, optimizers, interpreters, and transactions. Compared to other seed generation methods, SEDAR does not generate seeds from scratch. Instead, SEDAR obtains seeds by transferring test cases from other DBMSs with LLMs. It does not rely on the test cases of target DBMSs.

8 CONCLUSION

In this paper, we propose SEDAR, which generates the initial seeds for target DBMS fuzzing through cross-DBMS SQL transfer. First, to obtain the SQL test cases compatible with target DBMSs, it queries LLMs to transfer the test cases prompting with the collected schema information. Then, to make the seed mutable by the DBMS fuzzers, it comments out the unparsable parts for the parsers of mutators, and uncomments them after mutation. With SEDAR employed, mutation-based fuzzers SQUIRREL and GRIFFIN found 31, 25, 6, and 8 bugs in MonetDB, Virtuoso, DuckDB, and ClickHouse, respectively. Moreover, 19 bugs of them are assigned with CVE IDs due to their severity.

ACKNOWLEDGMENTS

This research is sponsored in part by the NSFC Program (No. 62302256, 62022046, 92167101, 62021002), Chinese Postdoctoral Science Foundation(2023M731953), and National Key Research and Development Project (No. 2022YFB3104000).

REFERENCES

[1] ANSI. 2022. ANSI Standard. https://www.ansi.org/. Accessed: January 4, 2024.
[2] Jinsheng Ba and Manuel Rigger. 2023. Testing database engines via query plan guidance. In *Proceedings of International Conference on Software Engineering (ICSE)*.
[3] MonetDB B.V. 2023. MonetDB Website. https://www.monetdb.org. Accessed: January 4, 2024.
[4] ClickHouse. 2021. Fuzzing: Practical Approaches in ClickHouse. https://presentations.clickhouse.com/cpp_siberia_2021/index_en.html. Accessed: January 4, 2024.
[5] DuckDB. 2023. DuckDB Fuzzers. https://github.com/duckdb/duckdb-fuzzer/issues. Accessed: January 4, 2024.
[6] DuckDB. 2023. DuckDB WebSite. https://www.duckdb.org/. Accessed: January 4, 2024.
[7] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.
[8] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*.
[9] 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*. 1–12.
[10] ClickHouse Inc. 2023. ClickHouse Website. https://clickhouse.com. Accessed: January 4, 2024.
[11] Free Software Foundation Inc. 2023. Error Recovery. https://www.gnu.org/software/bison/manual/html_node/Error-Recovery.html. Accessed: January 4, 2024.
[12] Solid IT. 2022. DB-Engines Ranking. https://db-engines.com/en/ranking. Accessed: January 4, 2024.
[13] Solid IT. 2022. DB-Engines Ranking of Relational DBMS. https://db-engines.com/en/ranking/relational+dbms. Accessed: January 4, 2024.
[14] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation.
[15] 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.
[16] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.
[17] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models. In *International conference on software engineering (ICSE)*.
[18] John R Levine, Tony Mason, and Doug Brown. 1992. *Lex & yacc.* " O'Reilly Media, Inc.".
[19] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. 2023. Sequence-oriented DBMS fuzzing. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*.
[20] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of {DBMS} with Coverage-based Guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. 4309–4326.
[21] Xuwei Liu, Wei You, Zhuo Zhang, and Xiangyu Zhang. 2022. TensileFuzz: facilitating seed input generation in fuzzing via string constraint solving. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 391–403.
[22] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic Detection of Performance Bugs in Database Systems using Equivalent Queries. (2022).
[23] MySQL. 2023. MySQL Github. https://github.com/mysql/mysql-server. Accessed: January 4, 2024.
[24] MySQL. 2023. The MySQL Test Suite. https://dev.mysql.com/doc/dev/mysql-server/latest/PAGE_MYSQL_TEST_RUN_PL.html. Accessed: January 4, 2024.
[25] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*. 729–743.
[26] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)* 34, 3 (2009), 1–45.

[27] PostgreSQL. 2023. PostgreSQL GitHub. https://github.com/postgres/postgres. Accessed: January 4, 2024.
[28] PostgreSQL. 2023. Regression Tests. https://www.postgresql.org/docs/current/regress-run.html. Accessed: January 4, 2024.
[29] Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. Evaluating the text-to-sql capabilities of large language models. *arXiv preprint arXiv:2204.00498* (2022).
[30] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1140–1152.
[31] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (2020). https://doi.org/10.1145/3428279
[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] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
[34] Andreas Seltenreich, Bo Tang, and Sjoerd Mullender. 2018. SQLsmith: a random SQL query generator. https://github.com/anse1/sqlsmith
[35] OpenLink Software. 2023. ChatGLM2 repository. https://github.com/THUDM/ChatGLM2-6B/blob/main/README_EN.md. Accessed: January 4, 2024.
[36] OpenLink Software. 2023. Virtuoso Open-Source Edition. https://vos.openlinksw.com/owiki/wiki/VOS. Accessed: January 4, 2024.
[37] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing Database Systems via Differential Query Execution. In *Proceedings of International Conference on Software Engineering (ICSE)*.
[38] SQLite. 2023. How SQLite Is Tested. https://www.sqlite.org/testing.html. Accessed: January 4, 2024.
[39] SQLite. 2023. SQLite Github. https://github.com/sqlite/sqlite. Accessed: January 4, 2024.
[40] The Clang Team. 2023. SanitizerCoverage. https://clang.llvm.org/docs/SanitizerCoverage.html. Accessed: January 4, 2024.
[41] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
[42] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 579–594.
[43] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden*. ACM, 61–64.
[44] 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.
[45] 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
[46] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. 2019. SLF: Fuzzing without valid seed inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 712–723.
[47] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. 2022. Glm-130b: An open bilingual pre-trained model. *arXiv preprint arXiv:2210.02414* (2022).
[48] 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*.