In-Depth Investigation of SQL Snippets From Communities for Mutation-Based-Fuzzing

Anonymous Author(s)

1 Motivation

Database Management Systems (DBMSs) are integral to modern software applications serving as the backbone for data storage and query processing across a wide range of domains. However, DBMSs are susceptible to vulnerabilities, which may lead to severe consequences, like service denials[4], data leaks[12], data loss[3], and even system failure [2]. In light of the significant damage these vulnerabilities can cause, it is imperative to proactively identify and mitigate security flaws within DBMSs to maintain system security and safeguard data integrity.

Mutation-based fuzzing [1, 6, 8, 10, 17, 18, 20] is recognized as an effective technique for uncovering vulnerabilities in software programs, and it has been successfully applied to DBMSs for effective testing and bug finding. In mutation-based fuzzing approaches, fuzzers maintain a pool of initial seeds and iteratively generate new test cases by mutating these seeds. These mutated test cases are then executed on the target programs or systems, exploring new code regions and potentially exposing vulnerabilities. Specifically in DBMS fuzzing, A seed typically refers to a test case that comprises a series of SQL statements. Mutation-based DBMS fuzzing continuously mutates these SQL statements to generate new test cases. The quality of these initial seeds is of utmost importance for the effectiveness of fuzzing, since they serve as the starting point for fuzzing and form the foundation for the subsequent fuzzing iterations. Additionally, the diversity of features contained in the initial seeds affects the variety of functionalities that can be explored within the DBMS. High quality initial seeds help increase test coverage and enhance the effectiveness of bug detection in mutation-based fuzzing.

In the DBMS fuzzing, the initial seeds are typically acquired from the DBMSs' built-in test cases[7, 11, 14], such as unit tests, regression tests, performance tests or tests for other purpose, which are designed to cover specific functionalities and test scenarios. With access to these test suites, engineers can collect initial seeds and guide the mutation process effectively.Although built-in test cases provide a solid foundation for generating meaningful mutations, there are some limitations to consider. Firstly, they may not cover all possible use cases, particularly those involving complex interactions or custom configurations. Secondly, since built-in test cases are designed to test expected behavior, they might not explore more unconventional or unexpected SQL patterns, which can limit the diversity of fuzzing input space. Lastly, built-in tests might not fully reflect real-world usage patterns that actual users employ, potentially reducing the effectiveness of the fuzzing process in uncovering real-world issues.

Therefore, collecting more various test cases from communities as initial seed can effectively enhance the fuzzing process. SQL snippets from communities like StackOverflow [15] or Postgres mail list [13], often cover a wide range of scenarios, including various query patterns, optimizations, and problem-specific solutions. Additionally, these snippets reflect real-world problems and use cases, providing a more diverse and potentially practical set of queries. We believe utilizing these SQL snippets from communities as initial seeds will improve the result of fuzzing and detect more possible bugs. However, SQL snippets in communities are not structured, and there are challenges to extracting and utilizing them in the mutation-based fuzzing process.

Firstly, SQL snippets in communities are hidden in dialogue texts, in which people pose technical questions and volunteers give some answers. Successfully fetching the SQL snippets from the text data requires developing a robust approach to identify, preprocess, validate, and even fix these snippets while preserving the context and structure of the SQL code. This process must ensure the specific database and its accurate version corresponding to the extracted SQL statements while maintaining the integrity and functionality of the SQL snippets. Secondly, the grammar support of DBMS fuzzers is often limited, which can pose challenges when working with SQL snippets from communities. These extracted snippets might include a wide range of SQL statements, some of which may not be fully supported by the fuzzer's grammar. Even if the snippets pass the syntax checker and are valid for execution in the database, fuzzers may still encounter difficulties in mutating them due to the lack of support for certain grammar.

In consequence, the primary object of this work is to investigate a solution for extracting SQL snippets from community sources, followed by preprocessing and organizing them into well-structured initial seeds for mutation-based fuzzing. Furthermore, we will conduct fuzz testing using these seeds and generate a report to validate our hypothesis that community-derived seeds can significantly enhance the effectiveness of the mutation-based fuzzing process.

2 Related Work

Mutation-Based DBMS Fuzzing. Mutation-based fuzzing techniques have been effectively utilized to test many widely-used DBMSs, such as SQLite, MySQL, MariaDB, and PostgreSQL. In these approaches, initial SQL seeds are mutated to generate various input cases. Specifically, *SQUIRREL* [20] mutates SQL statements by converting them into the Intermediate Representation (IR), applying mutation to the IRs, and translating the mutated IRs back into syntactically and semantically correct SQL statements. *LEGO* [9] introduces the type affinity of SQL sequences and employs it to guide the generation of SQL statements with abundant SQL type sequences. *GRIFFIN*[6] employs a grammar-free way to mutate SQL statements through statement reshuffle and metadata-based substitution. *SQLRight* [10] enhances mutation-based fuzzers with test oracles, using branch coverage to detect logical bugs.

Initial Seed Generation. Given the importance of the initial seed quality, there are several existing works that concentrate on generating high-quality seeds for general-purpose mutation-based

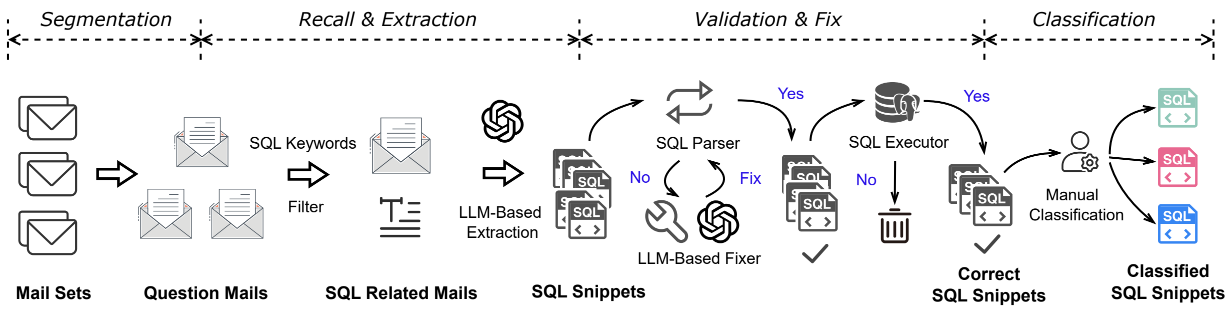


Figure 1: Workflow of processing

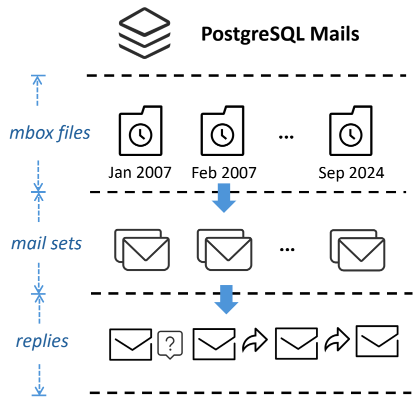


Figure 2: The structure of PostgreSQL mail data

fuzzing. *SAFL* [16] employs symbolic execution for qualified initial seeds that provide valuable exploration directions. *SLF* [19] generates valid seeds by automatically identifying input validity checks of programs and input fields related to these checks. 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 and various differences between databases. Typically, the DBMS fuzzing still take the built-in test cases as the initial seeds. Considering the lack of built-in test cases in certain databases, the *SEDAR* [5] bridges the test cases across different databases and generate high-quality seeds under the power of large language models.

3 Method

There are two community sources provided: Stack Overflow [15] and the PostgreSQL bugs mail list [13]. Stack Overflow offers a wider range of discussions on various database-related issues, including application and performance aspects across multiple databases. In contrast, the PostgreSQL bugs mail list specifically focuses on issues that lead to bugs within PostgreSQL. While leveraging diverse sources can potentially generate more valuable seeds for mutation-based fuzzing, we initially focus on mining SQL snippets from the PostgreSQL bugs mail list and limit our fuzzing efforts to the PostgreSQL database for simplicity.

The workflow is divided into four steps, as illustrated in Figure 1: (a). Segmentation, (b). Recall and Extraction, (c). Validation and Fix, and (d). Classification. They are elucidated in the following parts respectively.

3.1 Segmentation

In this step, we collect the raw data and do some preprocessing. The source data is downloaded from the official PostgreSQL bugs mailing list [1]. We have collected all relevant email files from 2016 year to 2024 year. The data structure is shown in Figure 2, which is arranged in chronological order, with each month's emails stored in an *mbox* file. under the assistance of the python built-in library *mailbox*, the *mbox* files are easily parsed and separated into mail sets. Each *mbox* file contains multiple sets of emails, where each set resolves one discussion topic. A mail set typically includes an initial question email and several reply emails. The initial question email proposes and highlights one issue related to a possible PostgreSQL bug and following reply emails discuss and attempt to resolve the problem. The main distinction between question mails and reply mails is that the latter includes the "RE:" prefix in the subject line. Empirically, question mails are more likely to contain SQL snippets, as the sender often needs to provide examples to explain and reproduce the reported issues and bugs. Given the large volume of mails, we prioritize the question mail. By filtering whether the title contains the prefix "RE:", we can easily filter out the reply mails and gather question mails as candidates. In addition, some unrelated information in the mail is removed, such as the sender details, email format, and attachments. Only the messages of the question mail are kept for subsequent processing. The static information of collected data is shown in Table 1. In this step, we download 31,379 mails from the PostgreSQL bug mailing list and segment 6,633 question mail above them.

3.2 Recall & Extraction

At this stage, we aim to leverage the capabilities of a Large Language Model (LLM) to extract SQL snippets from email text. Compared to rule-based methods, which rely on crafting complex regular expressions and a deep understanding of SQL syntax, the LLM-based approach is more user-friendly and delivers significantly better results. For example, the LLM can easily identify complex SQL

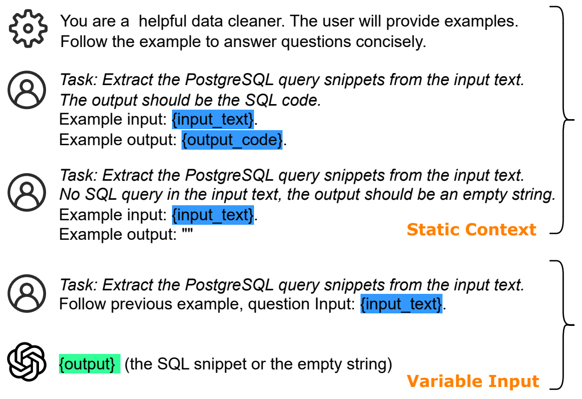
[1]https://www.postgresql.org/list/pgsql-bugs

Table 1: The size of data in each step

Type	Mail	Question Mail	SQL-Related Mail	Raw SQL Snippet	Syntax Correct SQL Snippet	LLM-Fixed SQL Snippet	Valid Fuzzing Seeds	Final Fuzzing Seeds
Number	31,379	6,633	4,845	3,558	2,753	309	505	118

snippets, such as those involving User-Defined Functions (UDFs). However, the main drawbacks are its lower efficiency and the additional costs associated with using LLM services. To optimize the extraction process, we introduce a recall stage that filters out emails lacking PostgreSQL keywords [2], such as SELECT, TABLE, and others. This stage ensures that only relevant emails proceed to the extraction step. In our implementation, the recall stage successfully excludes 1,788 emails from further processing.

Subsequently, we construct hard prompts for LLM to align the output with the standard result and conduct multi-round conversations to avoid long sequence input. The prompt design is presented in Figure 3. Initially, the agent is assigned the role of a "data cleaner," with a brief description of its task. Two examples are then provided to guide the formulation of the LLM's output. In the first example, where the input text contains an SQL snippet, the expected output is SQL code. In the second example, where the input text lacks an SQL snippet, the expected output is an empty string. Each user input consists of four components: a task description, an output hint, an example input, and an example output. These conversations are collected as static context and used as background information for every LLM inference. Then, the candidate mail messages are sent to LLM for processing. We extract 3,558 SQL snippets from 4,845 SQL-related mail messages at this stage.


You are a helpful data cleaner. The user will provide examples.
Follow the example to answer questions concisely.

Task: Extract the PostgreSQL query snippets from the input text.
The output should be the SQL code.
Example input: {input_text}.
Example output: {output_code}.

Task: Extract the PostgreSQL query snippets from the input text.
No SQL query in the input text, the output should be an empty string.
Example input: {input_text}.
Example output: ""

Task: Extract the PostgreSQL query snippets from the input text.
Follow previous example, question Input: {input_text}.

{output} (the SQL snippet or the empty string)

Figure 3: The prompt design for SQL snippet extraction

3.3 Validation & Fix

After extracting the raw SQL snippets, their correctness and compatibility with the PostgreSQL SQL parser cannot be guaranteed. While the snippets are generated with the assistance of an LLM, potential inaccuracies remain for several reasons. First, the LLM operates as a black box—despite its demonstrated capabilities across various domains, it is not immune to generating errors. Second, errors may originate from the questioner when crafting the SQL examples in the mail. To ensure the validity of these snippets, it is essential to use the PostgreSQL parser to convert them into Abstract Syntax Trees (ASTs) for thorough validation. This step helps identify and address potential issues before execution. To validate the syntax of each SQL snippet, we employ the *libpg_query* [3] library. For snippets that fail to pass the PostgreSQL parser, the library analyzes the parsing failures and provides possible reasons for the errors. Through this process, we identified 2,753 syntax-correct SQL snippets. For the remaining 805 snippets, potential human input errors were considered. These snippets, along with their corresponding failure reasons, were provided as input to the LLM, requesting assistance in fixing the errors. The recalibrated SQL snippets were then re-validated using *libpg_query*, resulting in an additional 309 syntax-correct SQL snippets. We can get 3,062 syntax-correct SQL snippets so far.

Despite being syntax-correct, these SQL snippets differ significantly from the initial seeds typically used in fuzzing tests. Built-in test cases generally consist of a sequence of SQL queries with complete contextual information. For example, they often include *CREATE* and *INSERT* statements to define the schema and establish the initial database state before executing other queries. In contrast, there is no constrain on SQL snippets. the extracted SQL snippets often lack this context, as authors typically assume that community members possess the background knowledge needed to fill in gaps in the provided examples. For example, the SQL snippets may about *SELECT* data from an non-existent table in database, leading to execution failures. As a result, these snippets cannot be directly used as testing seeds. To address this problem, we initialize a PostgreSQL server and wrap each SQL snippet within a transaction for execution. Snippets that fail within their transaction are discarded, while those that execute successfully are collected for further use. In this stage, we get 505 valid fuzzing seeds from 3,062 syntax-correct SQL snippets.

3.4 Classification

In the final stage, we manually review the 505 valid fuzzing seeds, removing overly simple SQL snippets, such as *SELECT version();*, and sampling a subset from the remaining set. This process resulted in 118 fuzzing seeds which are classified into three categories: **Standard** seeds: 46 seeds that involve only standard SQL keywords. **PG-featured** seeds: 59 seeds that utilize PostgreSQL-specific built-in functions and attributes. **TS-featured** seeds:13 seeds that involve timestamp-related functions and attributes.

[2]https://www.postgresql.org/docs/12/sql-commands.html
[3]https://github.com/pganalyze/libpg_query

Table 2: The effectiveness of various fuzzing seeds

Seeds	Map Coverage (%)	New Edges
base	15.36	2711
std	15.27	2681
pg	15.68	2828
std + pg	15.55	2647
std w/o base	12.56	1588
pg w/o base	11.01	1077

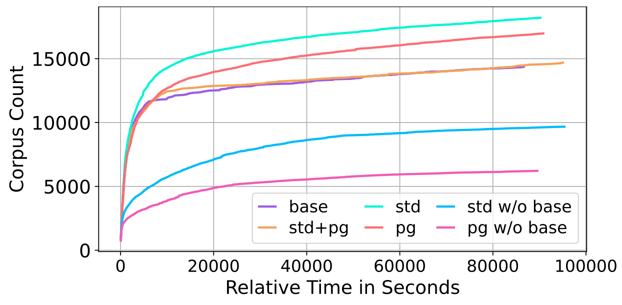


Figure 4: The corpus size over fuzzing time

4 Evaluation

All processing code and collected SQL Snippets are open-sourced in https://github.com/Zrealshadow/ComFuzzSeeds. To evaluate the effectiveness of the extracted fuzzing seed, we choose the **Squirrel** [20] as the base fuzzing tool to test the PostgreSQL database.

4.1 Setup

We mainly focus on evaluating the effectiveness of the Standard seeds and PG-featured seeds. We construct six groups of experiments, which involves different groups of fuzzing seeds.

- **base** test based on Squirrel built-in seeds.
- **std** test based on Squirrel built-in seeds and Standard seeds.
- **pg** test based on Squirrel built-in seeds and PG-featured seeds.
- **std w/o base** test based on Standard seeds.
- **pg w/o base** test based on PG-featured seeds.
- **std + pg** test based on Squirrel built-in seeds, Standard seeds, and PG-featured seeds

As for evaluation metrics, we choose the **Map Coverage** and **New Edges**. **Map Coverage** is typically displayed as a percentage, indicating how much of the program's control flow graph has been exercised relative to the instrumentation granularity. This metric can help assess how thoroughly the program has been tested. **New Edges** represents the number of test cases that result in better edge coverage. Each new edge suggests the fuzzer is making progress in discovering unexplored code paths. This metric indicates the exploration of new program behaviors in the fuzzing process. In simplification, the larger the map coverage and new edges , the better the fuzzing seeds are.

Our experiments run on a machine with an Intel Xeon W-1290P CPU (10 cores, 20 threads, 3.7 GHz base, 5.3 GHz turbo), 20 MB L3 cache, and support for AVX2 and FMA instructions. The system used a 64-bit Ubuntu 20.04 (kernel 5.13.0). The time budget for the fuzzer is set to 24 hours.

4.2 Results Analysis

Table 2 shows the results of fuzzing processing using different fuzzing seeds. First, we compare the results between *base*, **std** ,and **pg**. Among these, *pg* achieves the best map coverage and identifies the most new edges. This is attributed to the inclusion of SQL snippets in **pg** that involve many PostgreSQL built-in tables, schemas, and functions, which are absent in *base*. These additions diversify the input seeds, aligning with the conclusion in Squirrel[20] paper, which states SQL dialect unique to different DBMSs can enhance testing performance. Surprisingly, the performance of *std* is worse than *base*, even though the SQL snippets in **std** are a subset of those in *base*. However, Figure 4 presents that the generated test cases in *std* are more than others. This suggests that the additional test cases generated by *std* do not effectively improve map coverage. In this case, having more fuzzing seeds appears to negatively impact performance. A similar phenomenon also occurs between *pg* and *std + pg*. As discussed in the paper, we attribute this to the limitations in the efficiency of **Squirrel** testing tool. A larger number of seeds expands the mutation space, making it more challenging to identify effective test cases. Extending the fuzzing time beyond 24 hours could mitigate this issue by allowing more time to explore the expanded space. Furthermore, Comparing the results between *std*, *std w/o base*, *pg*, and *pg w/o base* highlights the critical role of built-in fuzzing seeds. These built-in seeds are specifically designed to cover as much of the standard SQL grammar as possible. When built-in seeds are excluded, relying solely on extracted seeds significantly reduces fuzzing performance. Thus, extracted seeds are most effective when used as a supplement to standard fuzzing seeds. Finally, *pg w/o base* is worse than *std w/o base*, likely because standard SQL snippets tend to reveal broader map coverage than PostgreSQL-specific SQL snippets.

5 Limitation and Future work

Evaluation Metric. Unlike other work that uses code coverage, we choose map coverage to evaluate the effectiveness of fuzzing seeds. It operates at a coarse level, focusing on transitions between basic blocks in the control flow graph, and lacks the granularity of code coverage, which tracks lines, branches, and functions. Map coverage cannot distinguish between different executions within the same basic block, misses subtle data-dependent behaviors, and provides no insight into execution depth or frequency. Therefore, better to evaluate the extracted fuzzing seeds using code coverage metrics in future work.

Fuzzing Tools. We use a single fuzzing tool, *Squirrel*, to evaluate the extracted fuzzing seeds. *Squirrel* converts SQL snippets into an intermediate representation (IR) for mutation and tests the DBMS based on coverage feedback. However, relying on a single fuzzing tool makes the experimental results less convincing, as different fuzzing tools may employ varied testing strategies and methods, leading to differing outcomes. In future work, we plan to use *SQL-Right* to further validate the effectiveness of our extracted seeds.

References

[1] Jinsheng Ba and Manuel Rigger. 2023. Testing database engines via query plan guidance. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2060–2071.

[2] Philip A Bernstein and Nathan Goodman. 1983. The failure and recovery problem for replicated databases. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*. 114–122.

[3] Gautam Bhargava and Shashi K Gadia. 1993. Relational database systems with zero information loss. *IEEE Transactions on Knowledge and Data engineering* 5, 1 (1993), 76–87.

[4] Scott A Crosby and Dan S Wallach. 2003. Denial of service via algorithmic complexity attacks. In *12th USENIX Security Symposium (USENIX Security 03)*.

[5] Jingzhou Fu, Jie Liang, Zhiyong Wu, and Yu Jiang. 2024. Sedar: Obtaining High-Quality Seeds for DBMS Fuzzing via Cross-DBMS SQL Transfer. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[6] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-free DBMS fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[7] PostgreSQL Global Development Group. [n. d.]. PostgreSQL. https://github.com/postgres/postgres. Accessed: 2024-09-06.

[8] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. {DynSQL}: Stateful Fuzzing for Database Management Systems with Complex and Valid {SQL} Query Generation. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4949–4965.

[9] 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 *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 668–681.

[10] 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.

[11] MySQL. [n. d.]. MySQL Test Suite. https://dev.mysql.com/doc/extending-mysql/8.4/en/mysql-test-suite.html. Accessed: 2024-09-06.

[12] Bryan Parno, Jonathan M McCune, Dan Wendlandt, David G Andersen, and Adrian Perrig. 2009. CLAMP: Practical prevention of large-scale data leaks. In *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 154–169.

[13] PostgreSQL Global Development Group. [n. d.]. PostgreSQL Bugs Mailing List. https://www.postgresql.org/list/pgsql-bugs/. Accessed: 2024-09-06.

[14] SQLite. [n. d.]. Testing SQLite. https://www.sqlite.org/testing.html. Accessed: 2024-09-06.

[15] Inc. Stack Exchange. [n. d.]. Authentication - Stack Exchange API. https://api.stackexchange.com/docs/authentication. Accessed: 2024-09-06.

[16] 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*. 61–64.

[17] 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.

[18] 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 *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 251–262.

[19] 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.

[20] 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 *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 955–970.