CERT: Finding Performance Issues in Database Systems Through the Lens of Cardinality Estimation

Jinsheng Ba  
National University of Singapore  
Singapore  
bajinsheng@u.nus.edu

Manuel Rigger  
National University of Singapore  
Singapore  
rigger@nus.edu.sg

ABSTRACT
Database Management Systems (DBMSs) process a given query by creating a query plan, which is subsequently executed to compute the query’s result. Deriving an efficient query plan is challenging, and both academia and industry have invested decades into researching query optimization. Despite this, DBMSs are prone to performance issues, where a DBMS produces an unexpectedly inefficient query plan that might lead to the slow execution of a query. Finding such issues is a longstanding problem and inherently difficult, because no ground truth information on an expected execution time exists. In this work, we propose Cardinality Estimation Restriction Testing (CERT), a novel technique that finds performance issues through the lens of cardinality estimation. Given a query on a database, CERT derives a more restrictive query (e.g., by replacing a LEFT JOIN with an INNER JOIN), whose estimated number of rows should not exceed the estimated number of rows for the original query. CERT tests cardinality estimation specifically, because it was shown to be the most important part for query optimization; thus, we expect that finding and fixing cardinality-estimation issues might result in the highest performance gains. In addition, we found that other kinds of query optimization issues can be exposed by unexpected estimated cardinalities, which can also be found by CERT. CERT is a black-box technique that does not require access to the source code; DBMSs expose query plans via the EXPLAIN statement. CERT eschews executing queries, which is costly and prone to performance fluctuations. We evaluated CERT on three widely used and mature DBMSs, MySQL, TiDB, and CockroachDB. CERT found 13 unique issues, of which 2 issues were fixed and 9 confirmed by the developers. We expect that this new angle on finding performance bugs will help DBMS developers in improving DBMSs’ performance.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging: Software performance.

KEYWORDS
Database, Performance Issue, Cardinality Estimation

1 INTRODUCTION
Database Management Systems (DBMSs) are fundamental software systems that allow users to retrieve, update, and manage data [25, 48, 56]. Most DBMSs support the Structured Query Language (SQL), which allows users to specify queries in a declarative way. Subsequently, the DBMSs translate the query into a concrete execution plan. To balance the trade-off between spending little time on optimization, which is performed at run time, and finding an efficient execution plan, researchers and practitioners have invested decades of effort into query optimization, covering directions such as search space exploration for join ordering [12, 13, 35], index data structures [16], execution time prediction [1, 57], or parallel execution on multi-core CPUs [15] and GPUs [37].

Finding performance issues in DBMSs—also referred to as optimization opportunities or performance bugs—is challenging. Given a query Q and a database D, we want to determine whether executing Q on D results in unexpectedly suboptimal performance. In general, no ground truth is available that specifies whether Q executes within a reasonable time. To exacerbate this issue, DBMSs use various heuristics and cost models during optimizations, or make trade-offs in optimizing specific kinds of queries over others. Second, the execution time of Q might be significant if D is large, making it time-consuming to measure Q’s actual performance. Given that the execution time depends on various factors of the execution environment [32] (e.g., the state of caches), it might even be necessary to execute Q multiple times to obtain a reasonably reliable measure of its execution time. Cloud environments are in particular prone to noise [24]; a report on testing SAP HANA [2] has recently stressed that performance testing for cloud offerings of DBMSs—such as SAP HANA Cloud, which runs in Kubernetes pods—is one of the main challenges in testing DBMSs due to inherently noisy environments.

Benchmark suites such as TPC-DS [51] or TPC-H [33] are widely used in practice to monitor DBMSs’ performance over versions through predetermined performance baselines, which could be specified [41, 61, 62]. However, deriving an appropriate baseline is challenging and might result in false alarms. Automated testing techniques have been proposed to find performance issues without the need of curating a benchmark suite. APOLLO [21] generates databases and queries automatically and validates whether executing the query on different versions of the DBMS results in significantly different execution times. However, APOLLO can only find regression bugs. AMOEBA [31] finds performance issues by examining discrepancies in the execution time of a pair of semantically equivalent queries. However, semantically equivalent queries do not necessarily exhibit a similar performance as the issues found by AMOEBA have a high false positive rate—only 6 of 39 issues
were confirmed by the developers, 5 of which were fixed [31]. For the above methods, queries need to be executed on sufficiently large databases to detect significant performance discrepancies.

In this work, we propose Cardinality Estimation Restriction Testing (CERT), a general technique that finds performance issues by testing the DBMSs’ cardinality estimation. Cardinality estimation is the process in which cardinality estimator computes estimated cardinalities, the estimated numbers of rows that will be returned. Since estimated cardinalities are approximate, it is infeasible to check for a specific number. Rather, the core idea of our approach is that making a given query more restrictive should cause the cardinality estimator to estimate that the more restrictive query should fetch at most as many rows as the original query. More formally, given a query $Q$ and a database $D$, $Card(Q, D)$ denotes the actual cardinality, that is, the exact number of records to be fetched by $Q$ on $D$. If we derive a more restrictive query $Q'$ from $Q$, $Card(Q', D) \leq Card(Q, D)$ always holds. $EstCard(Q, D)$ denotes the estimated cardinality for $Q$, and we expect $EstCard(Q', D) \leq EstCard(Q, D)$ to also hold for any DBMS. We refer to this property as cardinality restriction monotonicity. Any violation of this property indicates a potential performance issue.

CERT addresses the aforementioned challenges. Cardinality estimation accuracy was shown to be the single most important component for deriving an efficient execution plan [26]. Therefore, we believe that pinpointing issues in cardinality estimation would help developers focus on the most relevant issues, addressing which might result in significant performance gains. Additionally, this idea is applicable to finding a broader range of performance issues. For example, we found that other kinds of query optimization issues can be exposed by unexpected estimated cardinalities, as shown in Listing 3. Estimated cardinalities can be readily obtained by DBMSs without executing $Q$: DBMSs typically provide a SQL `EXPLAIN` statement that provides this information as part of a query plan, allowing our technique to achieve high throughput. Furthermore, since our method does not measure run-time performance, it can be used in noisy environments, and minimal test cases that demonstrate the performance issue can be automatically obtained [65]. Finally, CERT is a black-box technique, because applying it requires no access to the source code or binary.

Listing 1 shows a running example demonstrating CERT. We randomly generate SQL statements as shown in lines 1–4 to create a database state and ensure that each table’s data statistics is up to date in lines 5–6. Then, we randomly generate a query with a `LEFT JOIN` and derive a more restrictive query by replacing the `LEFT JOIN` with an `INNER JOIN` as shown in lines 8–9. The second query is more restrictive than the first query as `INNER JOIN` should always fetch no more rows than `LEFT JOIN`. We examined the estimated cardinalities in their query plans, obtained by using an `EXPLAIN` statement. Despite having made the query more restrictive, the cardinality estimator estimated that the first query fetches 20 rows, while the second one fetches 60 rows, which is unexpected. The root cause was an incorrect double-counting when estimating the selectivity of `OR` expression in the `ON` condition of the `INNER JOIN`. The estimated cardinality of the first query with `LEFT JOIN` should be no less than 60; after the developers fixed this issue, the estimated cardinality was changed to 60. This fix improved the performance of

```
Listing 1: This running example demonstrates a performance issue found by CERT in CockroachDB. It is due to an incorrect double-counting when estimating the selectivity of OR expressions in join ON conditions.
```

```{language=sql}
1 CREATE TABLE t1 (c0 INT);
2 CREATE TABLE t1 (c0 INT);
3 INSERT INTO t0 VALUES (1), (2), (3), (4), (5), (6), (7), (8), (9), (10), (11), (12), (13);
4 INSERT INTO t1 VALUES (21), (22), (23), (24), (25);
5 ANALYZE t0;
6 ANALYZE t1;
7
8 EXPLAIN SELECT * FROM t0 LEFT JOIN t1 ON t0.c0<1 OR t0.c0>1; -- estimated rows: 20
9 EXPLAIN SELECT * FROM t0 INNER JOIN t1 ON t0.c0<1 OR t0.c0>1; -- estimated rows: 60
10
11 • cross join(left outer) • cross join
12 | estimated row:20 | estimated row:60
13 | pred:(c0<1)OR(c0>1) | • filter
14 | • scan | • estimated row:12
15 | estimated row:13 | | filter:(c0<1)OR(c0>1)
16 | table: t0@t0_pkey | •• scan
17 | •• scan | estimated row:13
18 | estimated row:5 | table: t0@t0_pkey
19 | table: t1@t1_pkey | •• scan
20 | Estimated row:5
21 | Estimated row:20
```

Overall, we make the following contributions:

- We present a motivational study to investigate the causes of previous performance issues.
- We propose a novel technique, CERT, to test cardinality estimation for finding performance issues in query optimization
Listed 2: The EBNF representation of a query.

1. `SELECT [DISTINCT]`
2. `select_expression (, select_expression)*`
3. `FROM table_reference (INNER | LEFT | RIGHT | FULL | CROSS JOIN table_reference)*`
4. `(WHERE predicate)*`
5. `(GROUP BY predicate)*`
6. `(HAVING predicate)*`
7. `(LIMIT row_count)*`

We show a concrete realization of the technique by proposing 12 query-restriction rules.
- We implemented CERT in SQLancer and evaluated it on multiple aspects. CERT found 13 unique issues of cardinality estimation in widely-used DBMSs, and 11 issues were confirmed or fixed. The source code of CERT is publicly available, and has been integrated into SQLancer.1

2 BACKGROUND

Structured Query Language: Structured Query Language (SQL) [5] is a declarative programming language that expresses only the logic of a computation without specifying its specific execution. SQL is widely supported by DBMSs; for example, according to a popular ranking,2 the 10 most popular DBMSs support it. Listing 2 shows the EBNF representation [36], a metasyntax notation to express context-free grammars, of a SQL query, whose features we considered in this work. A query starts with the `SELECT` keyword. It can optionally be succeeded by a `DISTINCT` clause that specifies that only unique records should be returned. A `JOIN` clause joins two tables or views; various joins exist that differ on whether and how rows should be joined when the join predicate evaluates to false. A query can contain a single `WHERE` clause; only rows for which its predicate evaluates to true are included in the result set. Similar to `DISTINCT`, the `GROUP BY` clause groups rows that have the same values into a single row. It can be followed by a `HAVING` clause that excludes records after grouping them. The `LIMIT` clause is used to restrict the number of records that are fetched. More advanced features, such as window functions, common table expressions (CTEs), and subqueries can be used. While we did not consider them in this work, we believe that our proposed approach could be extended to support them.

Query optimization: DBMSs include query optimizers that, after parsing the SQL query, determine an efficient query plan, which specifies how a SQL query is executed. Determining an efficient query plan is challenging, since many factors might influence the plan’s performance. The most commonly used models are cost-based [6]—the query plan with the lowest projected performance cost is chosen. Cardinality estimation was found to be the most important factor that affects the quality of query optimization [26]. Cardinality estimators typically obtain data statistics of the tables to be queried by sampling [18], through histograms [46], or machine learning algorithms [10, 23, 59]. Then, they enumerate all sub-plan queries, which are queries that process only a subset of tables in a query, and estimate how many rows they fetch. For example, for a query $A \bowtie B \bowtie C$ ($\bowtie$ denotes a join), cardinality estimators could estimate the cardinalities of $A, B, C$ respectively, and then estimate the cardinalities of $A \bowtie B, B \bowtie C, A \bowtie C$, and $A \bowtie B \bowtie C$. Lastly, the estimated cardinalities of these sub-plan queries help to decide the join order—whether $A \bowtie B$ or $B \bowtie C$ should be executed first.

Query plan. A query plan is a tree of operations that describes how a specific DBMS executes a SQL statement. A query plan can be obtained by executing a query with the prefix `EXPLAIN`. Query plans typically include estimated cardinalities for operations that affect the number of subsequent rows. Listing 1 shows the two query plans for the two queries of the running example in lines 11–21. The first query uses a `LEFT JOIN`, and its query plan includes three operations: `cross join`, `scan`, and `limit`. The second query uses an `INNER JOIN`, and its query plan includes four operations: `cross join`, `filter`, `scan`, and `limit`. The structural difference between both query plans is the location of the operation `filter`, as the predicate $(\text{c0}<1)$ or $(\text{c0}=1)$ in the `ON` clause of both queries is part of the `cross join` of the first query plan, but is a separate operation `filter` in the second query plan. For the first query plan, the estimated cardinality 20 of the root node `cross join` (left outer) is determined based on the predicate as well as the two estimates for the number of records in tables `t0` and `t1`, which are 13 and 5. For the second query plan, the estimated cardinality for `cross join` is 60; here, the estimate is partly based on the `filter` operation, which is estimated to return 12 rows. In this work, when we refer to the estimated cardinality of a query, we refer to the query’s root node.

3 PERFORMANCE ISSUE STUDY

As a motivating study, to investigate if performance issues are caused by incorrect cardinality estimation in practice, we examined previous performance issues related to query optimization.

Subjects. We studied the issues reported for MySQL, TiDB, and CockroachDB. MySQL is the most popular relational DBMS according to a survey in 2021.3 TiDB and CockroachDB are popular enterprise-class DBMSs, and their open versions on GitHub are highly popular as they have been starred more than 33k and 26k times. They are widely used and have thus been studied in other DBMS testing works [30, 43, 44].

Methodology. We searched for performance issues using the keywords “slow” or “suboptimal” in the above-stated DBMSs, aiming to obtain issues that relate to either slow execution or suboptimal query plans. For MySQL, we chose issues whose status was closed, the severity was (S5) performance, and the type was MySQL Server: Optimizer in the bug tracker.4 Considering MySQL was first released in 1995 and some issues are too old to be reproduced, we investigated the issues in version 5.5 or later. For TiDB, we searched its repository5 by the filter `is/issue:closed linked:pr label:C-bug slow in:title`. For CockroachDB, we searched its repository6 by the filter `is/issue:closed linked:pr label:C-bug slow in:title`. Then, we

1https://github.com/sqlancer/sqlancer/issues/822
we identified six issues in three DBMSs, and three of them were bonds. Column incurs a slowdown of execution time from 0.01 seconds to 3.02 seconds. A performance issue caused by cardinality after fixing the implementation of the operation.

the right child, and the estimated cardinality remained unchanged operations. For example, issue #71790 was due to the inefficient cardinality for correlated columns. The other three issues, which caused by the estimated cardinalities, so we deemed both were caused by cardinality estimation. Issue #9067 was caused by the cardinality estimation due to an issue in calculating cardinality estimators, they are still observable through estimated cardinalities, so we deemed both were caused by cardinality estimation. Issue #9067 was caused by incorrect cardinality estimation. We attribute the lower number of performance issues to the difficulty in identifying and resolving performance issues in query optimization. For issues #61631 and #56714, they produce inefficient query plans that have higher estimated cardinalities than the optimal query plans. Although both issues are not directly due to the faults in cardinality estimators, they are still observable through estimated cardinalities, so we deemed both were caused by cardinality estimation. Issue #9067 was caused by incorrect cardinality estimation. We attribute the lower number of performance issues to the difficulty in identifying and resolving performance issues in query optimization. For issues #61631 and #56714, they produce inefficient query plans that have higher estimated cardinalities than the optimal query plans. Although both issues are not directly due to the faults in cardinality estimators, they are still observable through estimated cardinalities, so we deemed both were caused by cardinality estimation. Issue #9067 was caused by incorrect cardinality estimation. We attribute the lower number of performance issues to the difficulty in identifying and resolving performance issues in query optimization. For issues #61631 and #56714, they produce inefficient query plans that have higher estimated cardinalities than the optimal query plans. Although both issues are not directly due to the faults in cardinality estimators, they are still observable through estimated cardinalities, so we deemed both were caused by cardinality estimation. Issue #9067 was caused by incorrect cardinality estimation. We attribute the lower number of performance issues to the difficulty in identifying and resolving performance issues in query optimization. For issues #61631 and #56714, they produce inefficient query plans that have higher estimated cardinalities than the optimal query plans. Although both issues are not directly due to the faults in cardinality estimators, they are still observable through estimated cardinalities, so we deemed both were caused by cardinality estimation. Issue #9067 was caused by incorrect cardinality estimation. We attribute the lower number of performance issues to the difficulty in identifying and resolving performance issues in query optimization. For issues #61631 and #56714, they produce inefficient query plans that have higher estimated cardinalities than the optimal query plans. Although both issues are not directly due to the faults in cardinality estimators, they are still observable through estimated cardinalities, so we deemed both were caused by cardinality estimation. Issue #9067 was caused by incorrect cardinality estimation. We attribute the lower number of performance issues to the difficulty in identifying and resolving performance issues in query optimization. For issues #61631 and #56714, they produce inefficient query plans that have higher estimated cardinalities than the optimal query plans. Although both issues are not directly due to the faults in cardinality estimators, they are still observable through estimated cardinalities, so we deemed both were caused by cardinality estimation. Issue #9067 was caused by incorrect cardinality estimation. We attribute the lower number of performance issues to the difficulty in identifying and resolving performance issues in query optimization. For issues #61631 and #56714, they produce inefficient query plans that have higher estimated cardinalities than the optimal query plans. Although both issues are not directly due to the faults in cardinality estimators, they are still observable through estimated cardinalities, so we deemed both were caused by cardinality estimation. Issue #9067 was caused by incorrect cardinality estimation. We attribute the lower number of performance issues to the difficulty in identifying and resolving performance issues in query optimization. For issues #61631 and #56714, they produce inefficient query plans that have higher estimated cardinalities than the optimal query plans. Although both issues are not directly due to the faults in cardinality estimators, they are still observable through estimated cardinalities, so we deemed both were caused by cardinality estimation. Issue #9067 was caused by incorrect cardinality estimation.
to them. Our rules are not exhaustive; we believe that practitioners could propose additional rules depending on their testing focus.

*Rule overview.* In Table 2, CERT derives a *Target* statement from a *Source* statement by applying a restriction on the shown *Clause*, as demonstrated through *Example*. <Predicate> refers to a boolean expression and <Natural number> to a natural number. These examples are based on a database state with two tables t0 and t1, both of which have only one column c0. For each test to be generated, we randomly choose a SQL clause, of which one or more rules are randomly applied to restrict a query. In Figure 1, we choose the JOIN clause and apply only rule 1, which replaces a *LEFT JOIN* with an INNER JOIN in the JOIN clause of a query.

*JOIN clause.* Our key insight for testing the JOIN clause is the partial inequality relationship in terms of cardinalities between different kinds of joins. For a fixed join predicate, the following inequalities for the different joins’ cardinalities hold: INNER JOIN \(\leq\) LEFT JOIN/RIGHT JOIN \(\leq\) FULL JOIN \(\leq\) CROSS JOIN. Figure 2 illustrates this using a JOIN diagram [11] on two tables, each of which has three rows, with the same color denoting the rows that can be matched in the JOIN predicate. As determined by the SQL standard, INNER JOIN fetches rows that have matching values in both tables; LEFT JOIN/RIGHT JOIN fetch all rows from the left/right table and the matching rows from the respectively other table; FULL JOIN fetches all rows from both tables; CROSS JOIN fetches all possible combinations of all rows from both tables without an OR clause. A corner case for rule 5 concerning the CROSS JOIN is that this join may fetch fewer rows than FULL JOIN if either of the tables is empty, in which case CROSS JOIN fetches zero rows. To avoid potential false alarms, we ensure that each table contains at least one row.

*WHERE clause.* For the WHERE clause, our insight is that we can restrict the predicate that is used for filtering rows. If the query contains no WHERE clause, we restrict the query by adding one with a random predicate. If the WHERE clause’s predicate has an OR operator, we restrict it by removing either of the OR’s operands. Otherwise, we add an AND operator with a randomly generated predicate. Restricting predicates would also apply to testing JOIN clauses; in this work, we aimed to introduce the general idea behind CERT and illustrate it on a small set of promising rules. We believe that practitioners who adopt the approach will propose many additional rules.
which is more efficient. However, because the predicate is pushed down before joining two tables and is applied in the second query. The calculation is based on sequences rather than trees, because the computation of the edit distance of trees was shown to be NP-hard [50].

Table 2: The rules to restrict queries.

<table>
<thead>
<tr>
<th>Clause</th>
<th>Source</th>
<th>Target</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 JOIN</td>
<td>LEFT JOIN</td>
<td>INNER JOIN</td>
<td>SELECT * FROM t0 LEFT JOIN t1 ON ...;</td>
</tr>
<tr>
<td>2 JOIN</td>
<td>RIGHT JOIN</td>
<td>INNER JOIN</td>
<td>SELECT * FROM t0 RIGHT JOIN t1 ON ...;</td>
</tr>
<tr>
<td>3 JOIN</td>
<td>FULL JOIN</td>
<td>LEFT JOIN</td>
<td>SELECT * FROM t0 FULL LEFT JOIN t1 ON ...;</td>
</tr>
<tr>
<td>4 JOIN</td>
<td>FULL JOIN</td>
<td>RIGHT JOIN</td>
<td>SELECT * FROM t0 FULL RIGHT JOIN t1 ON ...;</td>
</tr>
<tr>
<td>5 JOIN</td>
<td>CROSS JOIN</td>
<td>FULL JOIN</td>
<td>SELECT * FROM t0 CROSS FULL JOIN t1;</td>
</tr>
<tr>
<td>6 SELECT</td>
<td>ALL</td>
<td>DISTINCT</td>
<td>SELECT ALL DISTINCT * FROM t0;</td>
</tr>
<tr>
<td>7 GROUP BY</td>
<td>&lt;Empty&gt;</td>
<td>&lt;Predicate&gt;</td>
<td>SELECT * FROM t0 GROUP BY c0 HAVING c0&gt;0;</td>
</tr>
<tr>
<td>8 HAVING</td>
<td>&lt;Empty&gt;</td>
<td>&lt;Predicate&gt;</td>
<td>SELECT * FROM t0 HAVING c0&gt;0;</td>
</tr>
<tr>
<td>9 WHERE</td>
<td>&lt;Empty&gt;</td>
<td>&lt;Predicate&gt;</td>
<td>SELECT * FROM t0 WHERE c0&gt;0 AND c0&lt;8;</td>
</tr>
<tr>
<td>10 WHERE</td>
<td>&lt;Predicate&gt;</td>
<td>AND &lt;Predicate&gt;</td>
<td>SELECT * FROM t0 WHERE c0&gt;0 AND c0&lt;8;</td>
</tr>
<tr>
<td>11 WHERE</td>
<td>&lt;Predicate&gt; OR</td>
<td>&lt;Predicate&gt;</td>
<td>SELECT * FROM t0 WHERE c0&gt;0 OR c0&lt;8;</td>
</tr>
<tr>
<td>12 LIMIT</td>
<td>&lt;Natural number&gt;</td>
<td>- &lt;Natural number&gt;</td>
<td>SELECT * FROM t0 LIMIT 10 5;</td>
</tr>
</tbody>
</table>

† Rule 5 holds when both tables are not empty.

Listing 4: An example of query plans that are not structurally similar, which is why we exclude them for testing.

1 CREATE TABLE t0 (c0 INT);
2 CREATE TABLE t1 (c0 INT, c1 INT);
3 INSERT INTO t1 VALUES(1,2), (3,4), (5,6), (NULL, NULL);
4 INSERT INTO t0 VALUES(1,2), (2);
5 ANALYZE t0;
6 ANALYZE t1;
7 EXPLAIN SELECT * FROM t0 FULL JOIN t1 ON t1.c1 IN (t1.c1)
   WHERE CASE WHEN t1.rowid > 2 THEN false ELSE t1.c1=1 END; -- estimated rows: 2
8 EXPLAIN SELECT * FROM t0 RIGHT JOIN t1 ON t1.c1 IN (t1.c1)
   WHERE CASE WHEN t1.rowid > 2 THEN false ELSE t1.c1=1 END; -- estimated rows: 3
9 ...

RIGHT JOIN, the sequence of operations is cross join, scan, filter. scan, in which the operation filter is applied before the operation cross join. The difference is due to a SQL optimization mechanism called predicate pushdown [28], which moves a filter to be executed before joining two tables and is applied in the second query. The predicate pushdown does not affect the final result, but can reduce the estimated cardinalities to be joined in the operation RIGHT JOIN, which is more efficient. However, because the predicate is pushed down, the structure of the query plans changes. Therefore, the estimated cardinalities are calculated in a different manner than that of the first query, and the developers consider the estimated cardinalities of both query plans as incomparable. In Listing 4, the estimated cardinalities of both operations FULL JOIN and RIGHT JOIN are calculated as the sum of the estimated cardinalities in the last step and the operation filter is calculated as one-third of the estimated cardinalities in the last step. The estimated cardinality of the first query plan is calculated by \((2 + 4)/3 = 2\), while that of the second query plan is calculated by \(4/3 + 2 = 3\).

We identify comparable estimated cardinalities by checking for structural similarity. In a query plan \(P\), the operation of a node \(N\) is denoted as \(O_N\). Suppose \(N\) has \(k\) children, denoted as \(C_1, C_2, ..., C_k\). The node’s flattened operation sequence is an array obtained by concatenating the flattened child nodes: flatten \((N) = [O_N, flatten(C_1), flatten(C_2), ..., flatten(C_k)]\). The flattening of the root node is also denoted as flatten \((P)\). For a pair of query plans \(P_a\) and \(P_b\), we define both are structurally similar only if \(ED(flatten(P_a), flatten(P_b)) \leq 1\), in which \(ED\) represents the edit distance [34], a common way of quantifying the dissimilarity of two strings, of both operation sequences. For example, in the query plans of Figure 1, the sequences of operations are \([\text{cross join}, \text{scan}, \text{scan}]\) and \([\text{cross join, filter, scan, scan}]\). The first sequence can be edited to the second sequence by inserting a filter only, and vice versa. Both query plans are structurally similar and we validate the cardinality restriction monotonicity property. If they are not structurally similar, we continue testing with a new query. The calculation is based on sequences rather than trees, because the computation of the edit distance of trees was shown to be NP-hard [50].

4.4 Validating Cardinality Estimation

Finally, we validate the cardinality restriction monotonicity property on the estimated cardinalities extracted from the query plans. If the estimated cardinality of the original query is lower than that of the more restrictive query, we report the query pair as an issue. In Figure 1, the estimated cardinality of the original query is 20, which is lower than that of the other restricted query, which is 60. This indicates an unexpected result, and we report both queries to developers. Recall that we deem the estimated cardinality in the root operation of the query plan as the estimated cardinality of the query and ignore the estimated cardinalities of other operations.
5 EVALUATION

To evaluate the effectiveness and efficiency of CERT in finding performance issues through estimated cardinalities, we implemented CERT in SQLancer, which is an automated testing tool for DBMSs, and, based on our prototype SQLancer+CERT, we sought to answer the following questions:

Q.1 Effectiveness. Can CERT identify previously unknown issues?
Q.2 Historic Bugs. Can CERT identify historic performance issues?
Q.3 Efficiency. How does CERT compare to the state-of-the-art approach in terms of accuracy and efficiency?
Q.4 Sensitivity. Which rules proposed in Table 2 contribute to finding issues? Do DBMSs adhere to the cardinality restriction monotonicity property in practice?

Implementation. We reused SQLancer and its implementation of a rule-based random generation method to generate queries and database states. For each table, we generated 100 INSERT statements and ensured that every table contained at least one row. Then, the generated queries and database states are passed to CERT for validating the cardinality restriction monotonicity property. The core logic of CERT is implemented in only around 200 lines of Java code for each DBMS, suggesting that its low implementation effort might make the approach widely applicable. We used pattern matching to extract operations and estimated cardinalities, and implemented the Dynamic Programming (DP) algorithm [54] to calculate the structural similarity.

Tested DBMSs. We tested the same DBMSs, MySQL, TiDB, and CockroachDB as we studied in Section 3. For Q1, Q3, and Q4, we used the latest available development versions (MySQL: 8.0.31, TiDB: 6.4.0, CockroachDB: 22.2.0). For Q3, in an attempt of a fairer comparison to AMOEBA, we chose the historical version of CockroachDB 20.2.19, which is the version that AMOEBA [31] tested.

Baselines. To the best of our knowledge, no existing work can be applied to specifically test cardinality estimation. The most closely related work is AMOEBA, which finds performance issues in query optimizers. We did not consider APOLO [21], because it finds only performance regressions. Ensuring a fair comparison with AMOEBA is challenging, as the approaches are not directly comparable. AMOEBA validates that semantic-equivalent queries exhibit similar performance characteristics, while CERT validates the cardinality restriction monotonicity property. Furthermore, both tools support a different set of DBMSs; AMOEBA supports CockroachDB and PostgreSQL, while CERT supports CockroachDB, TiDB, and MySQL. Thus, we performed the comparison in Q3 using only CockroachDB, which is supported by both tools.

Experimental infrastructure. We conducted all experiments on a desktop computer with an Intel(R) Core(TM) i7-9700 processor that has 8 physical cores clocked at 3.00GHz. Our test machine uses Ubuntu 20.04 with 8 GB of RAM, and a maximum utilization of 8 cores. We run all experiments 10 runs for statistical significance.

Q.1 Effectiveness

Method. We ran SQLancer+CERT to find performance issues. Each automatically generated issue report usually includes many SQL statements, making it challenging for developers to analyze the root reason for the issue. To alleviate this problem and better demonstrate the underlying reasons for these issues, we adopted delta debugging [65] to minimize test cases before reporting them to developers. The steps to minimize the test case are 1) incrementally removing some of the SQL statements in the test case and 2) ensuring that the cardinality restriction monotonicity property is still violated. After submitting the issue reports with minimized test cases to developers, we submitted follow-up issues only if we believed them to be unique, such as those identified by different rules than previous issues, to avoid duplicate issues. MySQL has its own issue-tracking system, and developers add a label Verified for the issues that they have confirmed. TiDB and CockroachDB use GitHub’s issue tracker. TiDB’s developers assign labels, such as affected versions and modules; we considered the issue as Confirmed after such a label was assigned. CockroachDB’s developers typically directly replied whether they planned on fixing the issue which we consider Fixed or whether the issue was considered a false alarm. In some cases, they added a Backlogged label to indicate that they would investigate this issue in the future. For all DBMSs, based on historic reports, we observed that, typically, developers directly reject duplicate issue reports.

Results. Table 3 shows the unique issues that CERT found in three tested DBMSs. The Bug ID column shows the bug id in respective bug trackers. The Version column shows the versions or git commits of the DBMSs in which we found corresponding issues. The Rules column shows which rules identified this issue. In total, we have reported 13 unique issues on unexpected estimated cardinalities. 9 issues have been confirmed by developers in three days, 2 issues have been fixed in one week, and 2 issues were backlogged. No false alarm was generated. We speculate that many confirmed bugs remain unfixed, because 1) fixing performance issues requires comprehensive consideration which usually consumes much time, and 2) performance issues might have lower priority than other issues, such as correctness bugs, which cause a query to compute an incorrect result. Among all 13 unique issues, the only known issue that we found in CockroachDB had been backlogged for around 10 months since it was first found, and our test case clearly demonstrated the root reason for the issue, which allowed developers to quickly fix it. Apart from the reported issues, CERT continuously generates more than ten issue reports per minute. We did not report the additional bug-inducing test cases to the developers to avoid burdening them, because deciding their uniqueness would be challenging. Therefore, we believe that CERT could help identify additional performance issues in the future. Overall, all issues were exposed in various SQL clauses and predicates, which may imply no common issues across tested DBMSs. We give two examples of minimized test cases to explain the issues we found as follows.

| Issue #108852 identified by rule 6. | Listing 5 shows a test case exposing issue #108852 in MySQL. Rule 6, which replaces ALL with DISTINCT in Table 2, exposed this issue. In Listing 5, the first query |

---

6https://github.com/sqlancer/sqlancer/releases/tag/v2.0.0
Table 3: The issues found by CERT. The red strike-through text is removed, while the content in green color is added.

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Bug ID</th>
<th>Version</th>
<th>Rules</th>
<th>Modifications to the query</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>108833</td>
<td>8.0.31</td>
<td>9</td>
<td>WHERE t0.c0 &gt; t0.c1</td>
<td>Verified</td>
</tr>
<tr>
<td>MySQL</td>
<td>108851</td>
<td>8.0.31</td>
<td>9</td>
<td>WHERE t1.c1 BETWEEN (SELECT 1 WHERE FALSE) AND (t1.c0)</td>
<td>Verified</td>
</tr>
<tr>
<td>MySQL</td>
<td>108852</td>
<td>8.0.31</td>
<td>6</td>
<td>DISTINCT</td>
<td>Confirmed</td>
</tr>
<tr>
<td>TiDB</td>
<td>38319</td>
<td>51a6684f</td>
<td>11</td>
<td>WHERE (TRUE) OR(TO_BASE64(t0.c0))</td>
<td>Confirmed</td>
</tr>
<tr>
<td>TiDB</td>
<td>38747</td>
<td>3ef8352a</td>
<td>7</td>
<td>GROUP BY t0.c0</td>
<td>Confirmed</td>
</tr>
<tr>
<td>TiDB</td>
<td>38479</td>
<td>3ef8352a</td>
<td>3 &amp; 5</td>
<td>CROSS LEFT JOIN</td>
<td>Confirmed</td>
</tr>
<tr>
<td>TiDB</td>
<td>38482</td>
<td>3ef8352a</td>
<td>8</td>
<td>HAVING (t1.c0)REGEXP(NULL)</td>
<td>Confirmed</td>
</tr>
<tr>
<td>TiDB</td>
<td>38665</td>
<td>6e55fa0</td>
<td>2</td>
<td>RIGHT INNER JOIN</td>
<td>Confirmed</td>
</tr>
<tr>
<td>TiDB</td>
<td>38721</td>
<td>6e55fa0</td>
<td>9</td>
<td>WHERE v0.c2</td>
<td>Confirmed</td>
</tr>
<tr>
<td>CockroachDB</td>
<td>88455</td>
<td>7cde315d</td>
<td>1</td>
<td>LEFT INNER JOIN</td>
<td>Fixed</td>
</tr>
<tr>
<td>CockroachDB</td>
<td>89161</td>
<td>f188d21d</td>
<td>11</td>
<td>WHERE (t0.c0 IS NOT NULL) OR (1 &lt; ALL (t0.c0 &amp; t0.c0))</td>
<td>Fixed (Known)</td>
</tr>
<tr>
<td>CockroachDB</td>
<td>89462</td>
<td>81586f62</td>
<td>8</td>
<td>HAVING (t1.c0 :CHAR) = 'a'</td>
<td>Backlogged</td>
</tr>
<tr>
<td>CockroachDB</td>
<td>90113</td>
<td>fbfb71b9</td>
<td>2</td>
<td>RIGHT INNER JOIN</td>
<td>Backlogged</td>
</tr>
</tbody>
</table>

Listing 5: Rule 6, which identified #108852 in MySQL, replaces ALL with DISTINCT.

1 CREATE TABLE t0 (c0 INT, c1 INT UNIQUE) ;
2 INSERT INTO t0 VALUES (-1, NULL), (1, 2), (NULL, NULL), (3, 4);  
3 ANALYZE TABLE t0 UPDATE HISTOGRAM ON c0, c1;
4 5 EXPLAIN SELECT ALL t0.c0 FROM t0 WHERE t0.c1; -- estimated rows: 3
6 EXPLAIN SELECT DISTINCT t0.c0 FROM t0 WHERE t0.c1; -- estimated rows: 4

Listing 6: Rule 11, which identified #89161 in CockroachDB, removes either operand of an OR expression.

1 CREATE TABLE t0 (c0 INT);
2 INSERT INTO t0 VALUES (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
3 ANALYZE t0;
4 5 EXPLAIN SELECT t0.c0 FROM t0 WHERE (t0.c0 IS NOT NULL) OR (1 < ALL (t0.c0, t0.c0)); -- estimated rows: 3
6 EXPLAIN SELECT t0.c0 FROM t0 WHERE (t0.c0 IS NOT NULL); -- estimated rows: 10

in line 5 fetches the rows including duplicate rows, while the second query in line 6 excludes duplicate rows, so the cardinality of the second query should be no more than that of the first query. However, the estimated cardinality of the second query is greater than that of the first query, which is unexpected. Suppose a query q with ALL is a subquery of another query Q with DISTINCT, this issue affects whether DISTINCT should be pushed down to the execution of q for an efficient query plan that aims to retrieve the fewest rows from q. This issue was confirmed by the MySQL developers already three hours after we reported it.

Listing 7: The performance improvement by fixing our found issues #88455 and #89161.

1 CREATE TABLE t0 (c0 INT);
2 CREATE TABLE t1 (c0 INT);
3 CREATE TABLE t2 (c0 INT);
4 INSERT INTO t0 SELECT * FROM generate_series(1,1000);
5 INSERT INTO t1 SELECT * FROM generate_series(1001,2000);
6 INSERT INTO t2 SELECT * FROM generate_series(1,333100);  
7 ANALYZE t0;
8 ANALYZE t1;
9 ANALYZE t2;
10 11 SELECT COUNT(*) FROM t0 LEFT OUTER JOIN t1 ON t0.c0 <1 OR t0.c0>1 FULL JOIN t2 ON t0.c0=t2.c0; -- 399ms → 321ms
12 SELECT COUNT(*) FROM t0 LEFT JOIN t1 ON t0.c0>0 WHERE (t0.c0 IS NOT NULL) OR (1 < ALL (t0.c0, t0.c0)); -- 131ms → 109ms

Issue #89161 identified by rule 11. Listing 6 shows a test case exposing issue #89161 in CockroachDB by rule 11, which removes either operand of an OR expression. The predicate (t0.c0 IS NOT NULL) in the WHERE clause of the second query should fetch no more rows than the predicate (t0.c0 IS NOT NULL) OR (1 < ALL (t0.c0, t0.c0)) of the first query. However, the estimated cardinality of the second query is greater than that of the first query, which is unexpected. This issue was caused by a buggy logic to handle the OR clause. In CockroachDB, given predicates A and B, the estimated cardinality of predicate A OR B is calculated by: P(A OR B) = P(A) + P(B) − P(A AND B). However, when A and B depend on the same table or column, the estimated cardinality is unexpected. We found this issue by rule 11 in Table 2. Although this issue was known, it had been backlogged for around 10 months since it was first found. When we reported our test case, the developer opened a pull request in their git repository to fix it after three days.
We checked whether any pair violated the cardinality restriction monotonicity. We identified in Table 1, we attempted using CERT which allows for more optimal joining orders.

**Performance analysis.** To investigate the extent to which the issues we found affect performance, we evaluated the query performance of the fixed issues #88455 and #89161 on a test case as shown in Listing 7 that involves joining multiple tables. We could not consider unfixed issues, as it would be unclear how to determine the potential speedup. We executed both queries in lines 12 and 13 before and after the fixes of #88455 and #89161 respectively. After executing either query ten times, we found that the fixes improved the performance by an average of 20% and 17%, respectively. This improvement is due to the more accurate estimated cardinality which allows for more optimal joining orders.

```
EXPLAIN SELECT MAX(a) FROM test; -- estimated rows: 1
EXPLAIN SELECT MAX(a) FROM test WHERE b=0; -- estimated rows: 500190
```

Q.2 Historic Bugs

**Method.** To evaluate whether cardinality restriction monotonicity is sufficiently general to identify previous performance issues that we identified in Table 1, we attempted using CERT to identify all three performance issues whose fixes changed the estimated cardinalities, namely issues #61631, #56714, and #9067. Specifically, based on the queries in the issue reports, we followed step 2 in Figure 1 to randomly construct 10,000 pairs of queries. Then, we checked whether any pair violated the cardinality restriction monotonicity before the fix, and adhered to the cardinality restriction monotonicity after the fix. If so and both query plans are structurally similar, we concluded that cardinality restriction monotonicity could have identified the performance issue.

**Results.** All three previous performance issues caused by cardinality estimation can be found by CERT. For example, considering the performance issue #56714 in Listing 3, Listing 8 shows the pair of queries that CERT produces to identify the performance issue. The second query has an additional WHERE clause compared to the first query, so the estimated cardinality of the second query should be no more than that of the first query. However, due to incorrect usage of the index in column b, the second query scans all rows and has a higher estimated cardinality. After the fix, the estimated cardinality of the second query decreases to 1.

Using CERT, we have found 13 unique issues in MySQL, TiDB, and CockroachDB. The fixes improve query performance by 19% on average.

Q.3 Efficiency

**Accuracy.** We evaluated whether CERT has higher accuracy in confirmed issues than AMOEBA. We evaluated this aspect based on the observation that around five in six reported bugs by AMOEBA were false alarms. A high rate of false alarms significantly limits the applicability of an automated testing technique. Recall that it is challenging to make a fair comparison as CERT and AMOEBA find different kinds of issues affecting performance.

**Results.** Table 4 shows the number of all and confirmed/ fixed unique performance issues found by CERT and AMOEBA. The authors of AMOEBA reported 25 issues in CockroachDB, but only 6 issues (24% accuracy) were confirmed or fixed by developers. In comparison, for CERT, 50% of issues in CockroachDB and 100% of issues in MySQL and TiDB were confirmed or fixed. For CockroachDB, CERT found fewer performance issues than AMOEBA, because we did not report all found issues to avoid duplicate reports. Overall, these results suggest that CERT has high accuracy and is a practical technique for finding relevant performance issues. Despite these promising results, on a conceptual level, similar to AMOEBA, we cannot ensure that the performance issues would be considered as such by the developers.

**Throughput.** We evaluated whether CERT has a higher testing throughput than AMOEBA. State-of-the-art benchmarks and approaches, such as TPC-H [33], AMOEBA [31], and APOLLO [21] execute queries, which results in relatively low throughput. Therefore, we expect that CERT will validate more queries per second. To evaluate this, we determined the average number of test cases per second processed by SQLancer+CERT and AMOEBA in one hour.

**Results.** In CockroachDB, on average across 10 runs and one hour, CERT validates 714.54 test cases while AMOEBA exercises 1.85 test cases per second. This suggests a 386× performance improvement over AMOEBA. Recall that the throughput results are not directly comparable, as different approaches can find different kinds of issues. In addition, CERT is immune to performance fluctuation, because the estimated cardinalities are not affected by execution time. Therefore, CERT yields the same results in different hardware and network environments.

Q.4 Sensitivity

**Sensitivity of rules.** We evaluated which rules presented in Table 2 contribute to finding the issues in Table 3. Specifically, we recorded which rules were applied to the bug-inducing test cases that we reported. We considered reported bug-inducing test cases,
We discuss some key considerations on the design of CERT, its characteristics, as well as the evaluation’s results.

Evaluating performance gains. It is challenging to measure the overall performance gain that fixing the issues reported by CERT could achieve in practice. One issue is that measuring the overall performance impact might be misleading, because many other components in query optimizers can affect performance as well. For example, research on the Join Order Benchmark [26] demonstrated that a worse cardinality estimator might lead to better performance due to the issues in other components. In addition, 9 issues were confirmed, but remained unfixed. Since we lack domain knowledge to address the underlying issues, we cannot determine the performance gains that fixing these issues would cause.

Generality. While we focused on important SQL clauses in this work, the 12 rules shown in Table 2 are not comprehensive. They could be extended to cover various other features, such as window functions. Additionally, the cardinality restriction monotonicity property might be applicable also to other kinds of data models than the traditional relational data model. For example, Neo4j, a graph DBMS, also uses a concept similar to query plans—termed execution plans—and cardinality estimation9 (EstimatedRows field in execution plans). Its query optimization also depends on cardinality estimation, which we expect to comply with the cardinality restriction monotonicity property. More work is required to explore cardinality restriction monotonicity in other DBMSs in the future.

7 RELATED WORK

Automatically testing for performance issues. The most related strand of research is on finding performance issues in DBMSs automatically. Jung et al. proposed APOLLO [21], which compares the execution times of a query on two versions of a database system to find performance regression bugs. Liu et al. proposed AMOEBA [31], which compares the execution time of a semantically-equivalent pair of queries to identify an unexpected slowdown. In contrast, CERT specifically tests cardinality estimation, which is most critical for query optimization [26]. Thus, we believe that issues found by CERT might be most relevant for DBMSs’ performance. Unlike these works, CERT avoids executing queries by inspecting only query

plans, allowing for higher throughput and making the approach robust against unanticipated performance fluctuations.

**Performance benchmarking.** Benchmarking DBMSs is a common practice to identify performance regressions, and to continuously improve the DBMSs’ performance on a set of benchmarks. TPC-H [33] and TPC-DS [51] are the most popular benchmarks and are considered the industry standard. Boncz et al. studied and identified 28 “chokepoints” (i.e., optimization challenges) of the TPC-H benchmark [4]. Poess et al. modified and analyzed the TPC-DS benchmark [38] to measure SQL-based big data systems. Karimov et al. proposed a benchmark for stream-data-processing systems [22]. Boncz et al. proposed an improved TPC-H benchmark, JCC-H [3], which introduces Join-Crossing-Correlations (JCC) to evaluate the scenarios where data in one table can affect the behaviors of operations involving data in other tables. Leis et al. proposed the Join Order Benchmark (JOB) [26], which uses more complex join orders. Raasveldt et al. described common pitfalls when benchmarking DBMSs and demonstrated how they can affect a DBMS’s performance [40]. CERT is complementary to benchmarking; while benchmarking focuses on workloads deemed relevant for users, CERT can find performance issues through the lens of cardinality estimation even on previously unseen workloads.

**DBMS testing.** Besides testing DBMSs’ performance, automated testing approaches have been proposed to find other types of bugs. Many fuzzing works on finding security-relevant bugs, such as memory errors, were proposed. SQLSmith [53], Griffin [14], DynSQL [20], and ADUSA [31] used grammar-based methods to generate test cases for detecting memory errors. Squirrel [67], inspired by grey-box fuzzers such as AFL [52], used code coverage as guidance to find memory errors. Various approaches have been proposed to find logic bugs in DBMSs. The PQS [44], NoREC [42], and TLP [43] oracles detect logic bugs in the implementation of SELECT statements. DQE [47] detects logic bugs in UPDATE and INSERT statements. Transactional properties have also been tested. Cobra [49] used formal methods to verify the serializability of executions in key-value stores. Troc [9] defines an oracle to detect isolation bugs in transactional procedures. Similar to testing approaches that find logic bugs and test properties, our core contribution is a new test oracle, which, however, finds performance issues that violate the cardinality restriction monotonicity property.

**Cardinality estimation.** Various cardinality-estimation techniques have been proposed. Han et al. [17] comprehensively evaluated various algorithms for cardinality estimation, of which we describe important ones below. PostgreSQL [55] and MultiHist [39] applied one-dimensional and multi-dimensional histograms to estimate cardinality. Similarly, UniSample [27, 66] and WiSample [29] used sampling-based methods to estimate cardinality. Apart from these traditional approaches, machine learning-based methods have gained attention recently. MSCN [23, 64], LW-XGB [10], and UAEQ [59] used deep neural networks, classic lightweight regression models, and deep auto-regression models to learn to map each query directly to its estimated cardinality. In addition, NeuroCard [63], BayesCard [58], DeepDB [19], and FSPN [60, 68] utilized deep auto-regression models and three probabilistic graphical models BN, SPN, and FSPN to predict the data distribution for cardinality estimation. CERT is a black-box technique that could, in principle, be applied to any cardinality estimator. However, we believe that finding relevant issues in learning-based estimators that can be fixed might be challenging.

**Metamorphic testing.** At a conceptual level, CERT can be classified as a metamorphic testing approach [7, 8], which is a method to generate both test cases and validate results. Metamorphic testing uses an input $I$ to a system and its output $O$ to derive a new input $I'$ (and output $O'$), for which a test oracle can be provided that checks whether a so-called Metamorphic Relation holds between $O$ and $O'$. For CERT, $I$ corresponds to the original query, $O$ is the estimated cardinality, $I'$ is the restricted query, and $O'$ is its estimated cardinality. The metamorphic relation that we validate is the cardinality restriction monotonicity property. Metamorphic testing has been applied successfully in various domains [8, 45]. TLP [43] and NoREC [42] are other metamorphic testing approaches that were proposed for testing DBMSs, as discussed above.

## 8 CONCLUSION

We have presented CERT, a novel technique for finding performance issues through the lens of cardinality estimation in DBMSs. Our key idea is to, given a query, derive a more restrictive query, and validate that the restriction is reflected by the DBMSs’ estimated cardinalities; we refer to this property as cardinality restriction monotonicity. Specifically, we validate that the original query’s cardinality is at least as high as the more restrictive query. Our evaluation has demonstrated that this technique is effective. Of the 13 unique issues that we reported, 2 issues were fixed, 9 issues were confirmed, and 2 issues require further investigation. The fixes improved query performance by 19% on average. Unlike other testing approaches for performance issues, CERT avoids executing queries, achieving a speedup of $386\times$ compared to the state of the art. Finally, it is readily applicable. DBMSs expose estimated cardinalities as part of query plans to users; thus, CERT is a black-box technique that is applicable without modifications, even if the DBMSs’ source code is inaccessible to the testers. Furthermore, since no queries are executed, CERT is resistant to performance fluctuations. Overall, we believe that CERT is a useful technique for DBMS developers and testers and hope that the technique will be widely adopted in practice.

## 9 DATA AVAILABILITY

Our implementation and experimental data are publicly available at https://zenodo.org/records/10476847.

## ACKNOWLEDGMENTS

This research is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing <NRF-NCR25-Fuzz-0001>). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and Cyber Security Agency of Singapore.