Beacon: Detecting Broken Access Control Vulnerabilities in DBMSs via System Catalog Consistency Validation

ZONGRUI PENG*, Tsinghua University, China
JINGZHOU FU*, Tsinghua University, China
ZHIYONG WU, Tsinghua University, China
JIE LIANG†, Beihang University, China
XIANGDONG HUANG, Tsinghua University, China
DALONG SHI, AVIC International Digital Network Technology Co., Ltd., China
YU JIANG†, Tsinghua University, China

Access control in DBMSs is critical for ensuring data security and integrity. However, the increasing complexity of its implementation often introduces broken access control (BAC) vulnerabilities. These vulnerabilities can lead to severe consequences, including privilege escalation, unauthorized data access, or even full compromise of the DBMS. Existing manual testing for BAC vulnerabilities is time-consuming and incomplete. Automated methods like static analysis also struggle in DBMSs, as static rules are difficult to apply to multi-level and dynamically changing privileges.

In this paper, we propose Beacon, which detects BAC vulnerabilities by validating the consistency between SQL operations and system catalogs. Our key insight is that the visibility of objects in the system catalogs is consistent with the user's access control: if an object is invisible to a user in the system catalogs, the user should not have any access privileges on that. Any inconsistency suggests that a user is exceeding their privileges, indicating a potential BAC vulnerability. We used Beacon to test eight popular DBMSs (e.g., MySQL and MariaDB), uncovering 39 previously unknown BAC vulnerabilities. Among them, 19 result in privilege escalation, and 20 lead to unauthorized information disclosure. Moreover, 7 of them have existed in DBMSs for more than 6 years, with the longest-persisting one lasting 13 years. DBMS vendors took these issues seriously and have already confirmed all of these vulnerabilities. Many vendors provided positive feedback, recognizing the importance of addressing these vulnerabilities. For instance, OceanBase awarded bounties for reported vulnerabilities, underscoring Beacon's role in improving DBMS access control.

CCS Concepts: • **Security and privacy → Database and storage security**.

Additional Key Words and Phrases: DBMS Testing, Access Control

ACM Reference Format:
Zongrui Peng, Jingzhou Fu, Zhiyong Wu, Jie Liang, Xiangdong Huang, Dalong Shi, and Yu Jiang. 2026. Beacon: Detecting Broken Access Control Vulnerabilities in DBMSs via System Catalog Consistency Validation. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 124 (April 2026), 29 pages. https://doi.org/10.1145/3798232

*Zongrui Peng and Jingzhou Fu contributed equally to this work.
†Jie Liang and Yu Jiang are the corresponding authors.

Authors' Contact Information: Zongrui Peng, KLISS, BNRist, School of Software, Tsinghua University, China; Jingzhou Fu, KLISS, BNRist, School of Software, Tsinghua University, China; Zhiyong Wu, KLISS, BNRist, School of Software, Tsinghua University, China; Jie Liang, School of Software, Beihang University, China; Xiangdong Huang, School of Software, Tsinghua University, China; Dalong Shi, AVIC International Digital Network Technology Co., Ltd., China; Yu Jiang, KLISS, BNRist, School of Software, Tsinghua University, China.

ACM 2475-1421/2026/4-ART124
https://doi.org/10.1145/3798232

1 Introduction

Database Management Systems (DBMSs) [84] are essential software solutions for data storage and management, extensively utilized across domains such as business, finance, and healthcare [75]. Access control is a core mechanism in DBMSs that regulates user access to data resources [46]. Through access control, the system effectively restricts access to unauthorized data, preventing data breaches or tampering [4, 13, 73].

Access control in DBMSs is inherently complex [3], often implemented through multi-level authorization mechanisms such as Role-Based Access Control (RBAC) [89] and privilege inheritance, enforced via hierarchical privilege tables (e.g., `mysql.user`) [46]. Its complexity makes the implementation prone to subtle and hard-to-detect flaws. In particular, insufficient permission checks may allow users to access data or perform operations beyond their intended authorization. These implementation issues, arising from failures to enforce proper access control, are collectively known as **Broken Access Control** (BAC) [7] vulnerabilities. For instance, according to Trend Micro [34], a BAC vulnerability in Apache CouchDB [14] enabled hundreds of real-world attacks that abused victim machines in organizations for cryptocurrency mining, causing severe resource exhaustion and device slowdowns [35, 41].

Figure 1 illustrates a BAC vulnerability in OceanBase where a regular user (with only basic privileges to manage their own tables) can bypass access control to retrieve information in tables of other users. Specifically, the user creates a trigger [42] containing `SELECT` operations that require the privileges of another user's table. When the trigger is activated, OceanBase skips permission checks on the `SELECT` command inside, resulting in the unauthorized disclosure of data. The root cause of this BAC vulnerability lies in the implementation of OceanBase, which enforces only the `CREATE TRIGGER` privilege at creation time, without validating the privileges required for the operations inside the trigger's definition. This vulnerability can be exploited to bypass access control during trigger execution. For instance, a regular user can embed `SELECT`, `UPDATE`, or `DELETE` operations on another user's table inside a trigger defined on their own table, which are executed without permission checks when the trigger is invoked. As a result, this allows both unauthorized information leakage and unauthorized data modification.

The main obstacle in detecting BAC vulnerabilities in DBMSs is *the comprehensive modeling of DBMS access control, which is essential for the automated determination of whether a given command violates access policies.* The difficulty of achieving this complete modeling is evident in the following aspects. First, DBMSs are complex due to multi-level privilege models, where privileges vary by level and change dynamically with user roles. This dynamic structure makes access control modeling particularly challenging [51, 54]. Second, the documentation often fails to explain clearly how

```
-- Create a trigger accessing other users' tables
CREATE TRIGGER hack_trigger BEFORE INSERT ON t0 FOR EACH ROW
INSERT INTO t1 SELECT * FROM table_of_other_user;
-- Activate the trigger with an INSERT command
INSERT INTO t0 VALUES ('');
```

Expected result: INSERT fails	***Actual Result: INSERT succeeds*** 🐞
SELECT command in the trigger body denied for table_of_other_user	the content of table_of_other_user is stored into table t1 without any permission check

Fig. 1. A BAC vulnerability in OceanBase Community Edition discovered by Beacon. It allows a regular user with the `TRIGGER` privilege to execute arbitrary `SELECT`, `INSERT`, `UPDATE`, and `DELETE` commands on arbitrary tables without any permission check.

privileges are granted, leaving many aspects vague or poorly described. This lack of clarity makes it difficult to fully understand how access control is implemented and managed. Finally, different DBMSs offer unique SQL commands and privilege options, and handle privilege rules in varying ways, making it challenging to construct a unified and precise checking mechanism [9, 51, 79].

To detect BAC vulnerabilities in DBMSs, developers manually create SQL test suites to verify behavior under specific privilege configurations [80]. While effective in limited cases, this manual process is time-consuming and requires expertise. Researchers have also explored automated methods [22, 31, 39], particularly static analysis, but adapting them to DBMSs is difficult due to the dynamic and context-dependent nature of DBMS access control. Unlike static operations in systems like web applications, DBMS privileges depend on runtime configurations and can vary with object-level or row-level rules, making static analysis ineffective for detecting BAC vulnerabilities.

In this paper, we propose Beacon, which detects BAC vulnerabilities in DBMSs via system catalog consistency validation. In DBMSs, system catalogs consist of system tables and views that describe the database structure (e.g., tables, columns, and indexes). They enforce access policies by filtering query results based on the user's privileges, ensuring that only database objects on which the user has at least one privilege are displayed. In other words, if the user has no privilege on an object, then the object should be invisible to that user in system catalogs, and operations on it should be unauthorized. Based on this observation, our key insight is that objects invisible to the user in system catalogs should not be authorized for the user to operate on. For example, if a user cannot view the system catalog entries for a table, then the user should be unauthorized to execute SQL operations (e.g., `SELECT`) on it. Therefore, any operation on a catalog-invisible object indicates a potential BAC vulnerability. Following this insight, Beacon detects privilege escalation when a command succeeds on objects on which the user holds no privileges, and detects unauthorized information disclosure when information of such non-privileged objects is revealed.

Concretely, Beacon detects BAC vulnerabilities in three phases: ① First, Beacon synthesizes operation sequences, assigns an execution user and a corresponding authorization command (`GRANT`/`REVOKE`) to each operation. ② Second, Beacon queries system catalogs as the root user and the regular user before executing each command, and compares the query results to identify unauthorized objects (i.e., the objects that a regular user cannot access due to privilege constraints). ③ Finally, Beacon executes the test cases and detects BAC vulnerabilities by checking if any unauthorized objects are exposed in SQL commands, query result sets, and error messages. After completing a test case, Beacon returns to phase 1 to generate the next one.

We implemented Beacon and evaluated it on eight popular DBMSs: MySQL, MariaDB, OceanBase, StarRocks, TiDB, Dameng, MonetDB, and PostgreSQL. Although the access control mechanisms of these DBMSs have undergone comprehensive testing, Beacon identified 39 previously unknown BAC vulnerabilities, all of which have been confirmed. These vulnerabilities pose serious security risks, potentially compromising system integrity and confidentiality. Among them, 19 result in *privilege escalation*, and 20 lead to *unauthorized information disclosure*. Moreover, 7 of them have existed for more than 6 years, with the longest-persisting one lasting 13 years. Beacon has been positively reviewed by database vendors, with many recognizing its critical role in addressing BAC vulnerabilities. For example, the bounty program of OceanBase awarded bounties for 10 vulnerabilities found by Beacon. This recognition highlights the value of Beacon in identifying security issues that could potentially be exploited in real-world scenarios.

In summary, our paper makes the following contributions:

- We found that BAC vulnerabilities in DBMSs, despite their critical severity and substantial risks to data integrity, currently suffer from a lack of automated detection methods.

- We propose an automatic BAC vulnerability detection method for DBMSs based on system catalog consistency validation. It leverages the correlation between a user's granted privileges and the database objects visible in the system catalogs to identify BAC vulnerabilities.
- We implemented our approach in Beacon and discovered 39 previously unknown BAC vulnerabilities in eight widely used DBMSs, all of which have been confirmed by vendors.

2 Background

DBMSs and SQL. A Database Management System (DBMS) [84] is software designed to store, organize, and retrieve structured data. DBMSs usually employ Structured Query Language (SQL) [90] to define, manipulate, query, and control data within databases. Users can interact with a DBMS by executing SQL commands to perform operations on database objects and data. SQL commands are typically categorized into four types: data definition language (DDL), data manipulation language (DML), data query language (DQL), and data control language (DCL) [90]. DDL creates and modifies database objects using commands like `CREATE`, `ALTER`, and `DROP` [86]. DML inserts, deletes, and updates data using three commands: `INSERT`, `DELETE`, and `UPDATE` [87]. DQL is primarily used to perform queries on data with commands such as `SELECT` [88]. DCL controls access to data through two fundamental commands: `GRANT`, which grants privileges to a specific user, and `REVOKE`, which revokes these privileges [85].

Access Control in DBMSs. Access control in DBMSs is a core mechanism that regulates user or program access to data resources [3, 75]. Its fundamental logic adheres to the "principle of least privilege" [72], which ensures that users are granted only the minimum privileges necessary to complete specific tasks. Based on this principle, DBMSs enforce strict access control that distinguishes the *root user* from *regular users*. The *root user* holds system-level privileges and has full control over a DBMS instance, including modifying global configurations and managing user accounts [19]. In contrast, *regular users* operate within assigned privileges based on operational needs [51]. This strict separation isolates critical system operations from regular data interactions, enforcing security boundaries through centralized authorization management.

Access control is enforced through DCL commands (`GRANT` and `REVOKE`) in DBMSs. These commands control the execution privileges for database operations, including DDL, DML, and DQL [51]. For example, if a regular user lacks the `SELECT` privilege on table `tbl`, executing the query `SELECT * FROM tbl` will fail with an error like "SELECT command denied for 'tbl' ". After the root user grants the privilege via `GRANT SELECT ON tbl TO regular_user`, the same query executes successfully. Conversely, if the privilege is subsequently revoked with `REVOKE SELECT ON tbl FROM regular_user`, the access is blocked again.

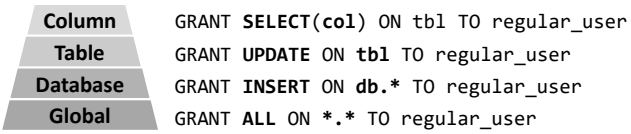

Fig. 2. The access control in MySQL operates across hierarchical privilege levels, with each level having its own format for `GRANT` and `REVOKE` statements.

With the two DCL commands, DBMSs establish hierarchical and dynamic access control with significant complexity: (1) DBMSs often support numerous SQL commands, each requiring specific privileges [51]. For example, `ALTER TABLE` in MySQL needs `ALTER`, `CREATE`, and `INSERT` privileges for the corresponding table [58]. (2) DBMSs utilize a hierarchical privilege structure that spans different privilege levels [47]. For example, as shown in Figure 2, MySQL's access control system

operates across multiple levels, including global, database, table, and column levels. Each level has its own set of applicable privileges for `GRANT` and `REVOKE`. (3) The access control mechanism is often influenced by dynamic factors, such as changes in user roles, variations in operation types, and real-time privilege adjustments, complicating privilege management in DBMSs [51].

System Catalogs. System catalogs are collections of system tables and views where DBMSs store schema metadata, such as information about tables and columns, and internal bookkeeping information [49, 78]. System catalogs are automatically created and maintained by DBMSs and serve as foundational components of database operations.

Table: test.employees

employee_id	name	phone	starting_year
2025011	Alice	4851	2025
2024051	Bob	0339	2024
2021801	Cindy	6931	2021
2021088	Tom	6287	2021

System catalog: information_schema.COLUMNS

column_name	table_name	table_schema	data_type
employee_id	employees	test	int
name	employees	test	varchar
phone	employees	test	varchar
starting_year	employees	test	year

Fig. 3. An example of a user table and its corresponding metadata in a system catalog.

Figure 3 illustrates how metadata for a user table is stored in a system catalog. For example, in MySQL, metadata for all columns in the current database is stored in the `information_schema.columns` table [53]. This system table contains attributes such as `column_name` (the column name), `table_name` (the table name), `table_schema` (the schema to which the table belongs), and `data_type` (the column data type). Users can use the `SELECT` command to retrieve such information from the system catalogs. As a foundational component of DBMSs, system catalogs enable users to efficiently query metadata such as table structures, constraints, and privilege assignments, supporting core functions like query processing and privilege validation.

One crucial property of system catalogs is *the strict enforcement of access control, which inherently aligns with the principle of least privilege.* As mandated by the SQL Standard, the query of each system catalog ensures that a given user can access only those rows that represent objects on which the user has privileges [23]. This principle means that metadata visibility is dynamically filtered based on user privileges. Most DBMSs implement this rule consistently for their system catalogs. For instance, MySQL explicitly states in its documentation: "For most `INFORMATION_SCHEMA` tables, each MySQL user has the right to access them, but can see only the rows in the tables that correspond to objects for which the user has the proper access privileges." [48] This design ensures users access only metadata in their privilege scope and helps prevent unauthorized information disclosure. This privilege-filtering mechanism operates through the DBMS's internal access control system, which aligns with the privilege management of regular table access.

3 Motivation

BAC vulnerabilities in DBMSs refer to *bugs in access control mechanisms that allow unauthorized users to perform actions they should not be allowed to do* [66]. In this paper, we focus on BAC vulnerabilities in the context of a logged-in user executing SQL commands, where the user could perform unauthorized actions due to insufficient permission checks. Such BAC vulnerabilities can be categorized into two types: (1) *Privilege escalation*: users may gain elevated privileges beyond what they are granted, which enables them to execute unauthorized SQL commands such as modifying other users' database structures, and even to take full control of the DBMS instance [37]. (2) *Unauthorized information disclosure*: specific SQL commands may expose sensitive information on other users in the execution output, such as table and column names [36]. This disclosure could

significantly lower the barrier for subsequent attacks [60, 67]. According to PortSwigger [68] and OWASP [59], table and column names are often required to construct SQL injection queries to access and manipulate unauthorized data [60, 67].

```
1. SHOW OPEN TABLES; -- output: empty set {}
2. SHOW COLLATION;
3. SHOW DATABASES;
4. SHOW FUNCTION STATUS;
5. SHOW OPEN TABLES; -- output: {db1.tbl1, db2.tbl2, ... }
```

Fig. 4. A BAC vulnerability of the SHOW command in MySQL. This vulnerability allows a regular user to access information about tables of other users.

An Example of BAC Vulnerability. Figure 4 illustrates a BAC vulnerability in MySQL detected by Beacon, where a regular user exploits the SHOW OPEN TABLES command to obtain information about tables belonging to other users. Specifically, in MySQL, the SHOW OPEN TABLES command enumerates non-temporary tables present in the table cache [52]. Under normal conditions, only tables for which the user has the corresponding SELECT privileges are included in the result. However, a BAC arises when a user executes the sequence of commands shown in Figure 4 in MySQL, which enables the listing of table information from unauthorized tables across all schemas.

```
1. check_db(SELECT, db=information_schema)
2. check_table(SELECT, table=OPEN_TABLES)
3. check_column(SELECT, columns=[OPEN_TABLES.Table,...])
4. check_db(SELECT, db=db1)            // expected: denied, actual: allowed
5. check_table(SELECT,table=db1.tbl1) // expected: denied, actual: allowed
//  additional checks on other user tables and 50+ internal tables
6. check_table(...)
7. read(table=OPEN_TABLES)
//  expected result: empty set,   actual: {db1.tbl1, ...}
```

Fig. 5. The root cause of the BAC vulnerability in Figure 4. The SHOW OPEN TABLES command's permission checks for databases and tables are broken due to the cached privileges of previous commands, causing the user to see unauthorized tables.

Root cause. Figure 5 shows the root cause of this BAC vulnerability. When the SHOW OPEN TABLES command is executed, MySQL first constructs a virtual table named OPEN_TABLE. It then verifies whether the current user has the necessary privileges to access the virtual table and its columns (Lines 1-3). Next, it iterates over all cached user tables and more than 50 internal system tables to perform permission checks (Lines 4-6). Only the databases and tables for which the user holds the SELECT privilege will be listed in the result (Line 7). However, during the processing of the preceding three SHOW commands in Figure 4, MySQL temporarily marks certain tables as selectable and caches these elevated privileges within the system. Consequently, when the user executes SHOW OPEN TABLES, MySQL incorrectly assumes the user has the privilege to list all open tables across the system (Lines 4-6). Therefore, the command's execution results expose table names and other details that should be hidden. This constitutes unauthorized information disclosure: the user sees information about all open tables, even if the user does not have the corresponding privileges.

Obstacles in Detecting BAC Vulnerabilities in DBMSs. The main obstacle in detecting BAC vulnerabilities in DBMSs is the *comprehensive modeling of DBMS access control mechanisms, which is essential for the automated determination of whether a given command violates access policies.*

However, achieving a comprehensive model would require perfectly replicating DBMS access control mechanisms, which is practically unachievable. The difficulty of achieving this complete modeling lies in the following aspects: (1) DBMS access control is complex due to its multi-level and dynamic nature, making it difficult to accurately model access control, particularly when complex privilege combinations are involved [51, 54]. (2) The DBMS specification and documentation often fail to clearly explain how privileges are granted, leaving many aspects vague or poorly described. This lack of clarity makes it difficult to fully understand how access control is implemented and managed. (3) The variation in how different DBMS implementations handle SQL grammar and security policies adds another layer of difficulty in creating a unified model.

Status of Existing Methods. To the best of our knowledge, DBMS BAC vulnerability detection primarily relies on manual analysis using handcrafted SQL test suites to verify system behavior under specific privilege configurations [80]. While effective in some cases, this approach is time-consuming, requires expertise, and cannot systematically cover all privilege configurations, leaving many BAC vulnerabilities undetected.

In addition to manual methods, automated techniques such as static analysis are often employed in Linux kernel [94] and web applications [1]. These techniques involve analyzing source code paths or enforcing predefined policies. However, applying them to DBMSs is problematic due to the complexity of DBMS access control. Specifically, these methods are typically rule-driven, assuming that identical operations or resource accesses require consistent permission checks. Any deviations from these predefined rules are flagged as potential BAC vulnerabilities. In contrast, DBMS access control is multi-layered, dynamic, and context-dependent, making these assumptions impractical. For example, as shown in Figure 4, the permission checks performed can vary depending on the database's current state (e.g., existing tables and schemas), causing the same operation to undergo different checks. This makes these methods ineffective for detecting BAC vulnerabilities.

Insight of Beacon. As mentioned in Section 2, system catalogs are closely aligned with the access control of database objects. In particular, a user's access to metadata (e.g., table structures, column information) directly corresponds to their access privileges to the actual data. In other words, objects on which a user has no privilege should be invisible in system catalogs and unauthorized for operations. Given this observation, Beacon does not attempt to fully replicate or model the entire database access control mechanism. Instead, it focuses on identifying inconsistencies between the visibility of system catalog entries and the access privileges granted during SQL execution.

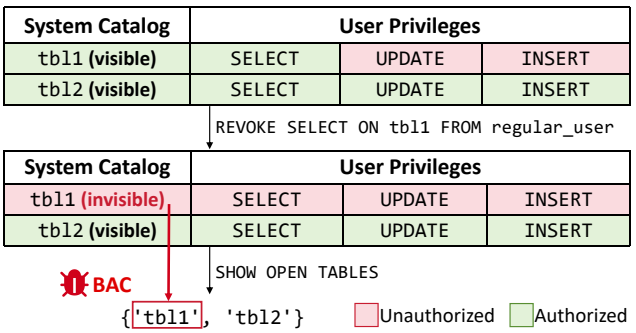

Fig. 6. Beacon detected that 'tbl1' in the result was invisible in system catalogs, revealing a BAC vulnerability.

Our key insight lies in the inherent consistency between the visibility of objects in system catalogs and the user's privileges. Any object invisible to a user in system catalogs should not be authorized for that user to access. Therefore, if a user can perform any operation on an object not visible to

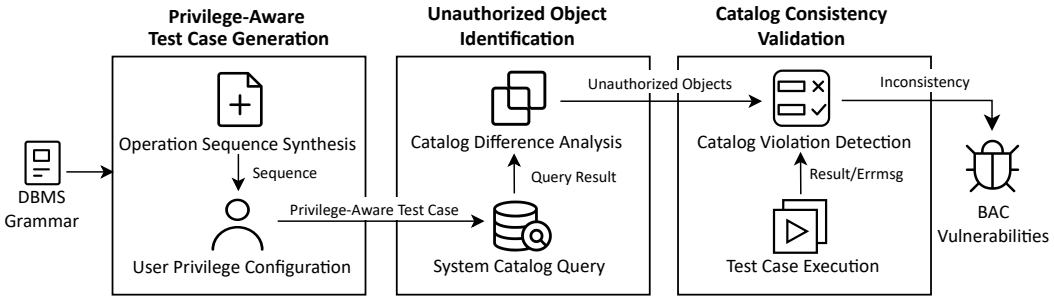

Fig. 7. The workflow of Beacon. It follows three phases: (1) Privilege-Aware Test Case Generation: creating diverse SQL operations and corresponding privilege configurations; (2) Unauthorized Object Identification: finding objects invisible to regular users by performing catalog comparison; (3) Catalog Consistency Validation: detecting unauthorized access or disclosure.

the user in the system catalogs, it indicates a BAC vulnerability in the DBMS. This approach is DBMS-agnostic, bypassing the obstacles of manual rule definition and the complexity of static modeling of all permission checks. For example, consider a regular user initially having privileges on tables `tbl1` and `tbl2`, and both tables appear in system catalogs. As illustrated in Figure 6, after revoking the `SELECT` privilege on `tbl1`, only `tbl2` remains visible to the user in the system catalogs. However, the result set of the `SHOW OPEN TABLES` command unexpectedly includes the invisible `tbl1`, thereby indicating a BAC vulnerability.

4 Beacon Design

Design Goal: A practical BAC vulnerability detection method should have the following properties.

- *General:* Beacon is designed to find BAC vulnerabilities for most practical DBMSs, from relational databases, e.g., MySQL [45], to NewSQL databases, e.g., TiDB [65], and from distributed databases, e.g., OceanBase [43], to analytical databases, e.g., StarRocks [76]. The tool can be deployed to different DBMSs with minor adjustments.
- *Automatic:* The testing process is fully automated, eliminating the need for manual effort.
- *Efficient:* Beacon is able to frequently exercise the access control logic and effectively detect BAC vulnerabilities in real-world DBMSs.
- *Accurate:* Beacon is designed to have satisfactory precision to reduce reporting false positives.

4.1 Beacon Workflow

Figure 7 illustrates the overall workflow of Beacon, which contains 3 phases: (1) *Privilege-Aware Test Case Generation*: Beacon first synthesizes SQL command sequences (DDL, DML, and DQL) and, for each command, adds a corresponding DCL operation to configure the regular user's privilege. Subsequently, Beacon specifies an execution user (*root* or *regular*) for each command. (2) *Unauthorized Object Identification*: Before executing each command, both the root user and the regular user query system catalogs to retrieve available database objects. The catalog query results are then compared to identify unauthorized objects (i.e., those that the regular user is not permitted to access). (3) *Catalog Consistency Validation*: Beacon executes each command with its specified user and monitors any unauthorized objects exposed in SQL commands, query result sets, and error messages. If an exposure is detected, a potential BAC vulnerability is identified. Thus, for each command in the test case (generated in phase 1), Beacon first identifies unauthorized objects

(phase 2), then executes the command and detects BAC vulnerabilities (phase 3). After processing all commands in the test case, Beacon returns to phase 1.

4.2 Privilege-Aware Test Case Generation

Privilege-Aware Test Case Generation constructs SQL operation sequences to systematically explore access control states capable of inducing potential BAC vulnerabilities. To achieve broad coverage of such vulnerabilities, the core challenge is to comprehensively cover diverse SQL operations and privilege configurations while maintaining syntactic and semantic correctness. Beacon addresses this challenge through two steps: (1) Operation Sequence Synthesis: Beacon synthesizes SQL operation sequences by emitting DDL, DML, and DQL commands in dependency order, ensuring that each command is semantically correct and all referenced objects exist. (2) User Privilege Configuration: For each command in the sequence, Beacon augments the sequence by injecting appropriate DCL commands and associating each SQL operation with specific user roles. This enables testing various privilege combinations and their impact on SQL operations.

Operation Sequence Synthesis. Based on the grammar of target DBMSs, Beacon generates SQL sequences composed of DDL (e.g., `CREATE`, `ALTER`), DML (e.g., `INSERT`), and DQL (e.g., `SELECT`) commands to systematically test the access control logic of DBMSs. To effectively exercise access control of DBMSs, Beacon varies combinations of object types (e.g., databases, tables, indexes) and operations (e.g., `CREATE`, `ALTER`, `DROP`) to drive DBMSs through diverse permission checks and corner cases. Meanwhile, in order to guarantee the semantic correctness and executability of these commands, Beacon maintains an internal schema. This schema is implemented as an in-memory nested map that records all active database objects. At the top level, it maps object types (e.g., `TABLE`,

Algorithm 1: SQL Operation Sequence Synthesis

Input : The number of DDL, DML, and DQL commands to generate: N_{ddl}, N_{dml}, N_{dql}.
The supported types of DDL, DML, and DQL commands: $types_{ddl}$, $types_{dml}$, $types_{dql}$.
The supported database object types: $types_{obj}$.
Output : The SQL operation sequence: seq.

```
seq ← empty list; schema ← empty map;
foreach obj_type ∈ dependencyOrderSort(types_obj) do
    create ← synthesizeSQL(CREATE_obj_type, schema); // Synthesize a CREATE for each type
    schema ← updateSchema(schema, create); // Update metadata changes
    seq.add(create);
for i ← 1 to N_ddl do
    ddl ← synthesizeSQL(types_ddl, schema); // Synthesize additional DDL commands
    schema ← updateSchema(schema, ddl); // Update metadata changes
    seq.add(ddl);
// Synthesize DML and DQL commands
for j ← 1 to N_dml do seq.add(synthesizeSQL(types_dml, schema)) ;
for k ← 1 to N_dql do seq.add(synthesizeSQL(types_dql, schema)) ;
return seq;
Function synthesizeSQL (types_sql, schema):
    type ← selectRandomly(types_sql); // Randomly select a SQL command type
    ast ← constructAST(type); // Construct an AST for the type
    if ¬ schema.empty() then ast ← instantiateAST(schema, ast);
    return transformASTtoSQL (ast); // Transform the instantiated AST into a SQL command
```

INDEX) to the set of existing object names, while nested entries track hierarchical details such as columns and constraints. This schema enables dynamic identifier resolution, allowing Beacon to construct commands by selecting valid targets from the schema rather than generating random names. This mechanism ensures that every synthesized operation, whether it is dropping an index or querying a column, strictly targets valid, existing objects recorded in the internal schema.

Beacon uses Algorithm 1 to synthesize SQL operation sequences. The process begins by initializing an internal schema: Beacon sorts the supported database object types by their dependencies (e.g., a column depends on a table), and generates one CREATE command for each type following the dependency order (Lines 2-5). After generating each command, Beacon updates the internal schema so later commands can reference existing objects (Line 4). Next, Beacon synthesizes N_{ddl} additional DDL commands to further evolve the schema (Lines 6-9), and updates the internal schema after every DDL command to reflect the latest metadata (Line 8). Finally, Beacon synthesizes N_{dml} DML and N_{dql} DQL commands based on the current schema (Lines 11-12). Since DML and DQL commands do not change schema metadata, no schema update is needed. For each command, Beacon calls synthesizeSQL to ensure syntactic and semantic correctness (Lines 14-18). The function randomly selects a command type (e.g., SELECT, CREATE, INSERT) from the given set and constructs an abstract syntax tree (AST) for that type (Lines 15-16). If the internal schema is not empty, it instantiates the AST by binding required object references to existing objects in the internal schema, following the dependencies declared in the AST (Line 17). For instance, DROP INDEX must reference an index that already exists in the schema. Beacon then transforms the instantiated AST into a concrete SQL command (Line 18). A simplified example of the generated operation sequences is shown in the top part of Figure 9, where N_{ddl}, N_{dml}, and N_{dql} are set to 1.

User Privilege Configuration. Beacon inserts a distinct DCL command before each command in the initially generated operation sequence. Figure 8 presents a formal description of the constructed DCL commands in MySQL. DCL commands fall into two categories: the GRANT command (i.e., *grant*) and the REVOKE command (i.e., *revoke*). A GRANT (or REVOKE) command consists of two parameters: *priv* and *level*. The *priv* denotes the specific privilege to be granted or revoked, such as CREATE, INSERT, or SELECT. The *target* represents the target objects to which the privilege changes are applied. The *object* refers to the target database object, such as a table, view, or schema, on which the *priv* is either assigned or removed.

$$
\begin{aligned}
dcl &:= \texttt{<}grant\texttt{>} \mid \texttt{<}revoke\texttt{>} \\
grant &:= \texttt{GRANT <}priv\texttt{> ON <}target\texttt{> TO regular\_user} \\
revoke &:= \texttt{REVOKE <}priv\texttt{> ON <}target\texttt{> FROM regular\_user} \\
priv &:= \texttt{CREATE} \mid \texttt{DROP} \mid \texttt{INSERT} \mid \texttt{SELECT} \mid \texttt{SELECT(}col\texttt{)} \mid \ldots \\
target &:= *.* \mid db.* \mid db.object \mid object
\end{aligned}
$$

Fig. 8. MySQL's formal description for DCL commands.

For each command in the initially generated operation sequence (i.e., *cmd*), Beacon constructs the corresponding DCL command through the following steps: (1) Beacon selects a database object from *cmd*. Note that if *cmd* does not contain any object, Beacon selects no object. (2) Beacon selects a privilege level (i.e., global, database, table, or column level) that suits the object or is higher than the object's level. If no object is selected in the previous step, Beacon sets the level to global. (3) Beacon chooses a privilege from the selected level, and constructs the grant or revoke command dcl_{cmd} according to the formal description in Figure 8.

After generating DCL commands to configure the regular user's privileges, BEACON constructs the privilege-aware test case. We model the privilege-aware test case as a sequence of execution tuples $\langle u, s \rangle$, where u is the user account (either the *root* user or the *regular* user), and s is the SQL command. First, since only the root user can execute DCL commands, BEACON appends an execution tuple $\langle root, dcl_{cmd} \rangle$ to the test case. Subsequently, BEACON appends two execution tuples for each SQL operation command *cmd*: $\langle regular, cmd \rangle$ and $\langle root, cmd \rangle$. The first tuple tests whether the regular user can execute *cmd* under the current privileges, enabling the further detection of unauthorized access. Given that the regular user may lack sufficient privileges, certain commands could fail, which would disrupt the dependency order of subsequent operations. For example, if creating table `tbl` fails, subsequent read or write operations on `tbl` will also fail. To ensure that the overall test sequence remains valid and all dependent commands can proceed, BEACON always follows with $\langle root, cmd \rangle$. $\langle root, cmd \rangle$ will be executed only if the regular user fails to execute *cmd*, aiming to ensure that each operation is successfully executed at least once. BEACON produces the privilege-aware test case by repeating these steps for each command in the operation sequence. It contains both valid SQL operations and diverse privilege configurations.

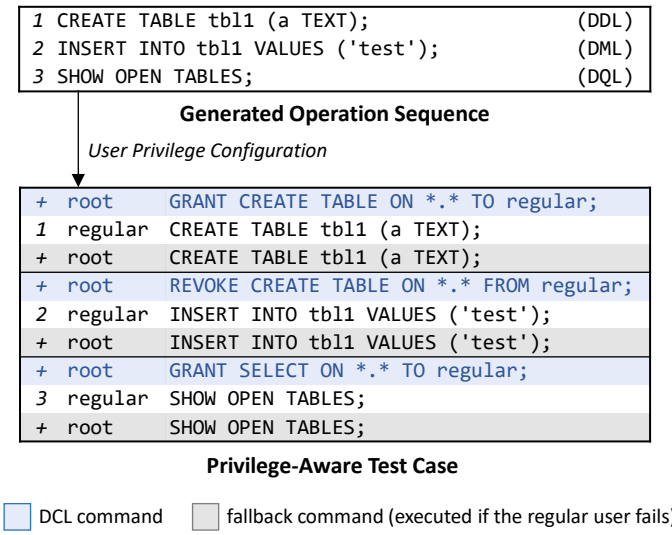

Fig. 9. A simplified sample of privilege-aware test cases.

Figure 9 presents a simplified example of privilege-aware test case generation. The original operation sequence includes `CREATE`, `INSERT`, and `SHOW` commands. For each command, BEACON inserts a DCL command as the root user, and appends the original SQL command for both the regular user and the root user. For example, the `INSERT` command is transformed into three execution tuples: $\langle root, \texttt{REVOKE} \rangle$, $\langle regular, \texttt{INSERT} \rangle$, and $\langle root, \texttt{INSERT} \rangle$. When BEACON later executes the test case, it first has the root user perform the `REVOKE` command, then lets the regular user attempt the `INSERT` operation. If the attempt fails due to insufficient privileges, the root user then executes the `INSERT` command, ensuring that all operations remain valid for subsequent steps.

4.3 Unauthorized Object Identification

Unauthorized objects refer to objects that users are restricted from accessing, such as tables, columns, and triggers. Identifying them aims to collect information on all unauthorized objects

within DBMSs. Accessing unauthorized objects precisely indicates broken access control, which outlines the authorized privilege boundaries of users. Information regarding unauthorized objects is subsequently used to detect unauthorized access in DBMSs via Catalog Consistency Validation.

Beacon identifies unauthorized objects by executing system catalog queries. DBMSs provide functionality for querying system catalogs, which list all database objects along with their names, types, and additional details. Since system catalog queries return only privileged objects for the current user, objects that are unavailable in the query results can be recognized as unauthorized. Specifically, the identification process contains two steps: (1) *System Catalog Query:* Beacon executes system catalog queries under both the root user and the regular user accounts, obtaining the accessible objects of each user. (2) *Catalog Difference Analysis:* Beacon compares the system catalog query results to identify unauthorized objects for the regular user.

System Catalog Query. Beacon executes system catalog queries under both the root user (with administrative privileges) and the regular user (with restricted privileges dynamically configured via User Privilege Configuration) accounts to retrieve information regarding their accessible objects, such as schemas, tables, columns, triggers, functions, and procedures. Concretely, the root user can query metadata for all database objects, whereas the regular user's results only contain the subset of objects for which it currently holds privileges. Note that querying all system catalogs is excessive for Unauthorized Object Identification. While DBMSs typically maintain hundreds of system catalogs, only a subset of them record information on database objects. To efficiently locate the relevant catalogs, Beacon first generates a variety of sample database objects (e.g., tables, views, functions) from an individual empty database. Next, it queries each system catalog as the root user to determine which catalogs expose the names of the created objects. Subsequently, Beacon selects a minimal subset of catalogs sufficient to capture all database objects. For example, in MySQL, `information_schema.tables` already includes information on views in the database, rendering it unnecessary to query `information_schema.views` separately. This selection avoids redundant queries and reduces execution overhead.

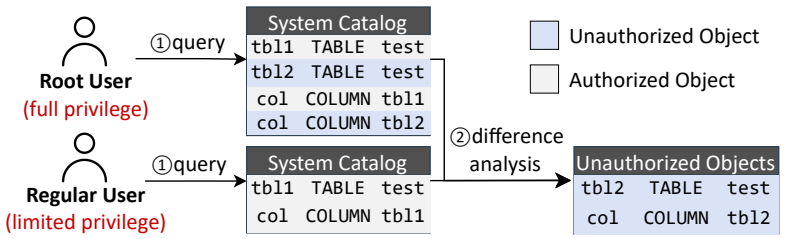

Fig. 10. An example of identifying unauthorized objects.

Catalog Difference Analysis. Beacon conducts a difference analysis between the system catalog query results of the root and the regular user. Specifically, it subtracts the result set of the regular user's query from that of the root user's. To precisely identify these objects, the remaining set contains objects accessible to the root user but not to the regular user. Each database object is represented as a tuple `(object_name, type, dependent_object_name)`, where `dependent_object_name` indicates its dependent object (e.g., a table's dependent object is the schema it belongs to, and a column's dependent object is its table). Two objects are considered identical only when their object name, type, and dependent object name values match. For example, the columns `col` in table `tbl1` and `col` in `tbl2` are not recognized as the same, because they belong to different tables. These identified unauthorized objects will then be used to detect BAC vulnerabilities through Catalog Consistency Validation.

Figure 10 illustrates an overall example of identifying unauthorized objects. Consider a database containing tables `tbl1` and `tbl2`, both with a `col` column. The regular user has the `SELECT` privilege on `tbl2` but lacks privileges on `tbl1`. In such a setting, Beacon first queries system catalogs under both root and regular user accounts. While multiple system catalogs are scanned, the relevant metadata in this case includes table definitions from `information_schema.tables` (containing `tbl1` and `tbl2`) and column definitions from `information_schema.columns` (containing both columns). The root user's query results show both `tbl1` and `tbl2` with all columns, whereas the regular user observes only `tbl2` with its single column `col`. Next, Beacon compares the query results from the root and the regular user. This comparison exposes database objects accessible to the root user but inaccessible to the regular user (`tbl1` and its column `col`), which will be targeted to detect BAC vulnerabilities in Section 4.4.

4.4 Catalog Consistency Validation

Catalog consistency refers to the alignment between database objects visible in system catalogs and those accessible during SQL command processing. If any unauthorized object is detected during the execution of an SQL command, it is considered a **catalog consistency violation**.

For each tuple $\langle u, s \rangle$ in the privilege-aware test case, Beacon executes the SQL command s with user u. For each command executed by the regular user, Beacon monitors the result sets and error messages, and checks for *catalog consistency violations* to identify BAC vulnerabilities. Based on the two categories of BAC, unauthorized information disclosure and privilege escalation, Beacon employs three *detection rules* ($\text{Rule}_{\text{result}}$, $\text{Rule}_{\text{errmsg}}$, and Rule_{cmd}). As Figure 11 illustrates, for objects a user has no privileges on, $\text{Rule}_{\text{result}}$ and $\text{Rule}_{\text{errmsg}}$ detect unauthorized information disclosure if their names are revealed in result sets and error messages, respectively. Meanwhile, Rule_{cmd} identifies privilege escalation when a command succeeds on such non-privileged objects.

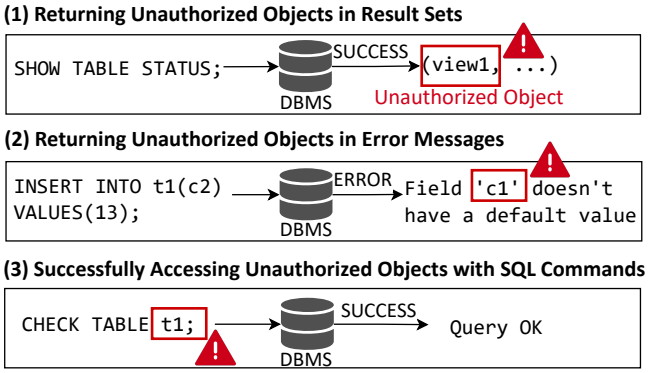

Fig. 11. Examples of catalog consistency violations in result sets, error messages, and SQL commands.

Rule 1: Returning Unauthorized Objects in Result Sets ($\text{Rule}_{\text{result}}$). If unauthorized objects appear in result sets, it suggests that the DBMS is not correctly enforcing access control before returning the results. This enforcement gap can lead to unauthorized information disclosure in query results, such as names of tables and columns owned by other users. $\text{Rule}_{\text{result}}$ supports detecting BAC vulnerabilities in DQL commands. The top part of Figure 11 presents an example of the unauthorized objects within the result sets of a DQL command. Consider the situation where the view `view1` is unauthorized for the regular user. If the result set of the command `SHOW TABLE STATUS`, executed by a regular user, includes `view1`, it is a potential BAC vulnerability.

Rule 2: Returning Unauthorized Objects in Error Messages ($\text{Rule}_{\text{errmsg}}$). If error messages contain unauthorized objects, it implies that the DBMS may have broken access control when generating error messages. The disclosed unauthorized objects in error messages may include tables, columns, etc. $\text{Rule}_{\text{errmsg}}$ can uncover BAC vulnerabilities across DDL, DML, and DQL commands. For instance, consider a table `t1` with columns `c1` and `c2`. As shown in Figure 11, if the regular user lacks sufficient privileges to access `t1`, `c1`, and `c2`, a DML operation such as `INSERT INTO t1(c2) VALUES(13)` may fail, yielding an error message stating: "Field 'c1' doesn't have a default value". This message inadvertently exposes the existence of the unauthorized column `c1`.

Allowlisted cases. Notably, if unauthorized objects appear in both the executed SQL command and its error message, the situation is generally not considered a BAC vulnerability. Such error messages simply echo what the user already submitted in the SQL commands, without actually revealing new information. For example, if the user lacks `DROP` privilege on table `t1`, `DROP TABLE t1` will fail with an error message "DROP command denied for table 't1'". However, in such a condition, the appearance of 't1' in the error message is not considered problematic.

Rule 3: Successfully Accessing Unauthorized Objects with SQL Commands (Rule_{cmd}). If a successful SQL command accesses unauthorized objects at runtime, which is an act of privilege escalation, the DBMS may have broken access control in its authorization or query processing. Accessing unauthorized objects during runtime can result in severe consequences, such as unauthorized data modifications. Rule_{cmd} can identify BAC vulnerabilities in DDL, DML, and DQL commands. The bottom of Figure 11 illustrates an example: if a table `t1` is considered unauthorized for the regular user, then the command `CHECK TABLE t1`, which verifies table integrity and error detection for `t1`, should fail due to insufficient privileges on `t1`. If this command succeeds, it implies that the unauthorized object `t1` is altered or modified during execution.

Allowlisted cases. Note that if a DQL command executes successfully but returns an empty result, even when involving unauthorized objects, it does not qualify as this type of vulnerability. For example, queries with filtering conditions (like `WHERE` or `LIKE` clauses) might return empty results due to insufficient privileges on the database objects referenced in the conditions. Consider the command `SHOW TABLE LIKE 't1'`. If the user lacks access to the table 't1', this query won't fail but

Algorithm 2: Catalog Consistency Validation

Input : The executed SQL command by the regular user: sql,
Unauthorized database objects: unauth_objs
Output : The matched detection rule for violations

```
type, succ, result, errmsg ← status(sql);
foreach obj ∈ unauth_objs do
    // Check the allowlisted cases
    is_allow_errmsg ← contain(sql, obj) and ¬succ; // Allowlisted for Rule errmsg
    is_allow_cmd ← isDQL(type) and isEmpty(result); // Allowlisted for Rule cmd
    // Identify BAC by three detection rules
    if contain(result, obj) then
        return Rule_result
    if contain(errmsg, obj) and ¬is_allow_errmsg then
        return Rule_errmsg
    if contain(sql, obj) and succ and ¬is_allow_cmd then
        return Rule_cmd
return NoRuleMatched; // No violation found
```

will return an empty result set. Therefore, successful DQL commands referencing unauthorized objects but with an empty result set should be distinguished from this type of vulnerability.

Algorithm 2 shows the procedure for catalog consistency validation. First, Beacon captures the SQL command's execution status: its command type, result, error message, and whether the execution is successful (Line 1). Next, for each unauthorized object, Beacon verifies its inclusion in the allowlists of $\text{Rule}_{\text{errmsg}}$ and Rule_{cmd}. (Lines 4-5). Subsequently, the algorithm detects BAC by applying the three detection rules. If the result exposes unauthorized objects, it returns $\text{Rule}_{\text{result}}$ (Lines 7-8). If the error message contains unauthorized and non-allowlisted objects, it reports $\text{Rule}_{\text{errmsg}}$ (Lines 9-10). If a successful SQL command contains unauthorized objects and doesn't fall into the allowlist, it flags Rule_{cmd} (Lines 11-12). Finally, if the SQL command passes all checks, it concludes by reporting that no rule was matched (Line 13).

5 Implementation

We implemented Beacon with the SQLGlot framework [32]. The following details highlight key aspects of our implementation.

To generate diverse types of SQL commands, we implemented a SQL generator based on each target DBMS's SQL grammar. Beacon employs SQLGlot to construct syntactically correct AST for every single SQL command, and uses Beacon's internal schema to instantiate these ASTs into valid SQL sequences. Our generator supports 50, 47, 52, 21, 39, 42, 36, and 54 types of SQL commands, and covers 16, 15, 18, 4, 11, 22, 18, and 13 types of database objects, for MySQL, MariaDB, OceanBase, StarRocks, TiDB, Dameng, MonetDB, and PostgreSQL, respectively. To balance coverage and cost, for each DBMS, we set the test-case length (i.e., N_{ddl}, N_{dml}, and N_{dql}) to twice the number of supported command types. The length of the generated test cases is typically 300-600 commands. During test case generation, Beacon enforces a naming policy that disallows simple or duplicate object names, ensuring that objects referenced in SQL commands and execution outputs can be uniquely identified. To support the generation of grant and revoke statements, we extracted all valid privileges and their permissible levels (e.g., global, database, and table) from official DBMS documentations [47]. We store a privilege list for each level (e.g., `SELECT`, `INSERT`, `UPDATE` for the table level, and `EXECUTE` for the routine level). When granting or revoking privileges for an object, Beacon samples from the object's level-specific privilege list (with a fixed seed for reproducibility).

To identify unauthorized database objects, we implement case-insensitive matching for object names. This approach aligns with the standard practices of most DBMSs, which typically store and compare object names in a case-insensitive manner. For instance, when checking table names like `tbl1` and `TBL1`, Beacon treats them as identical entries. The matching process normalizes database object names to a uniform case format. This enables consistent comparison operations regardless of original letter casing, while maintaining accurate identification of unauthorized objects.

To validate system catalog consistency, we implement an execution monitor and a violation detector. The monitor leverages standard Python client interfaces to connect target DBMSs, execute prepared SQL commands, and retrieve query results and error messages. Error messages are captured by catching exceptions raised by the clients during database interactions. The violation detector then finds unauthorized objects present in query results, errors, and the original SQL commands by whole-word matching. Beacon performs whole-word matching to eliminate false positives caused by substring overlaps. For example, a search term like `tbl1` will only match the exact word, ignoring incidental occurrences within longer words (e.g., `tbl10`).

During testing, the same underlying flaw may be triggered by multiple test cases generated by Beacon, leading to duplicate vulnerability reports. To eliminate such duplicates, we implemented an automatic test case reduction and deduplication approach in Beacon. First, it collapses test cases whose triggering commands share the same SQL structure (i.e., identical ASTs). It then prunes

unnecessary preceding commands from each test case, and groups the remaining cases by the operation type of the triggering statement (e.g., `SELECT`, `INSERT`, `GRANT`). After this automatic processing, we manually merge semantically equivalent test cases within each group before reporting.

6 Evaluation

To the best of our knowledge, Beacon is the first automated testing framework specifically designed to detect BAC vulnerabilities in DBMSs. While other DBMS testing tools test for different types of bugs, such as crashes or logic errors, they lack support for detecting BAC vulnerabilities. Meanwhile, existing access control testing approaches are designed for fixed access control models in operating systems or web applications, which are unsuitable for DBMSs, as detailed in Section 7.

Consequently, our evaluation assesses Beacon's ability to detect BAC vulnerabilities across real-world DBMSs. Specifically, we aim to answer the following research questions:

- **RQ1:** Can Beacon find new BAC vulnerabilities in real-world DBMSs?
- **RQ2:** What are the impacts and root causes of these BAC vulnerabilities?
- **RQ3:** How effective are the Beacon components?

6.1 Evaluation Setup

Tested DBMSs. We targeted Beacon on eight popular DBMSs, including MySQL 9.2.0 [45], MariaDB 11.7.1 [6], StarRocks 3.3.5 [76], TiDB 9.0.0 [65], Dameng 2024.10 [8], OceanBase 4.3.4 [43], MonetDB 25.3.2 [38], and PostgreSQL 18.1 [69]. These DBMSs were selected based on their widespread use in industry, covering different database types (relational, columnar, and distributed). The version of each DBMS was the latest available at the time of our evaluation.

Experiment Setup. We conducted the experiments on a machine running 64-bit Ubuntu 20.04 with 128 cores (AMD EPYC 7742 Processor @ 2.25 GHz) and 504 GiB of main memory. All DBMSs tested were deployed in Docker containers and could be downloaded directly from the Docker Hub websites [12]. For quantitative comparisons, we ran the Docker containers for each DBMS experiment (including DBMS server and Beacon) with 10 CPU cores and 10 GiB of main memory. In the evaluation, we tested Beacon on all eight DBMSs for 24 hours.

6.2 BAC Vulnerability Detection

Beacon successfully exposed **39** previously unknown BAC vulnerabilities on these DBMSs within a 24-hour testing period. We have reported all the vulnerabilities to DBMS developers. As of this writing, all of these reports have been confirmed, and 23 have already been fixed.

Statistics. Table 1 summarizes the distribution of the discovered 39 BAC vulnerabilities across eight DBMSs. Beacon detected 5, 5, 10, 3, 8, 4, 3, and 1 BAC vulnerabilities in MySQL, MariaDB, OceanBase, StarRocks, TiDB, Dameng, MonetDB, and PostgreSQL, respectively. These results indicate that BAC vulnerabilities remain pervasive even in widely used and extensively tested DBMSs. Beacon triggers these vulnerabilities by constructing different permission check scenarios, and employs system catalog consistency validation for identification.

Impact of the Detected Vulnerabilities. We organize the identified vulnerabilities into two major categories based on the impact of BAC vulnerabilities: ① *Privilege Escalation.* This category includes flaws that permit the execution of SQL commands (e.g., `CREATE`, `REPLACE`) without proper privileges on database objects, directly violating access control. We identified 19 vulnerabilities detected by Rule_{cmd} belonging to this category. ② *Unauthorized Information Disclosure.* This category includes vulnerabilities where the result sets or error messages of SQL commands contain unauthorized information (e.g., schema details). Such disclosure can potentially facilitate further targeted attacks. All 20 detected by $\text{Rule}_{\text{result}}$ and $\text{Rule}_{\text{errmsg}}$ were classified under this category. According to DBMS

Table 1. The 39 BAC vulnerabilities detected by Beacon within 24 hours, including 5 in MySQL, 5 in MariaDB, 10 in OceanBase, 3 in StarRocks, 8 in TiDB, 4 in Dameng, 3 in MonetDB, and 1 in PostgreSQL.

DBMS	Command	Rule	Bug Count and Status
MySQL	SELECT	$\text{Rule}_{\text{result}}$	Confirmed (4)
MySQL	SHOW	$\text{Rule}_{\text{result}}$	Confirmed (1)
MariaDB	DROP TRIGGER	$\text{Rule}_{\text{errmsg}}$	Confirmed (1)
MariaDB	LOAD CACHE	Rule_{cmd}	Confirmed (1)
MariaDB	SELECT	$\text{Rule}_{\text{result}}$	Confirmed & Fixed (3)
OceanBase	CALL	Rule_{cmd}	Confirmed (2)
OceanBase	CREATE TRIGGER	Rule_{cmd}	Confirmed & Fixed (1)
OceanBase	CREATE PROCEDURE	Rule_{cmd}	Confirmed & Fixed (1)
OceanBase	CREATE FUNCTION	Rule_{cmd}	Confirmed & Fixed (1)
OceanBase	SELECT	$\text{Rule}_{\text{result}}$	Confirmed & Fixed (3)
OceanBase	SHOW	$\text{Rule}_{\text{result}}$	Confirmed & Fixed (2)
StarRocks	INSERT INTO	$\text{Rule}_{\text{errmsg}}$	Confirmed & Fixed (1)
StarRocks	SHOW	$\text{Rule}_{\text{result}}$	Confirmed & Fixed (2)
TiDB	DELETE	Rule_{cmd}	Confirmed (2)
TiDB	ADMIN	Rule_{cmd}	Confirmed & Fixed (4)
TiDB	REPLACE VIEW	Rule_{cmd}	Confirmed & Fixed (1)
TiDB	SELECT FOR UPDATE	Rule_{cmd}	Confirmed & Fixed (1)
Dameng	CREATE SCHEMA	Rule_{cmd}	Confirmed (1)
Dameng	SELECT	$\text{Rule}_{\text{result}}$	Confirmed (3)
MonetDB	ALTER USER	Rule_{cmd}	Confirmed & Fixed (2)
MonetDB	DROP USER	Rule_{cmd}	Confirmed & Fixed (1)
PostgreSQL	CREATE TYPE	Rule_{cmd}	Confirmed (1)
Total	**39 Detected, with 39 Confirmed and 23 Fixed**		

developers, these flaws could trigger real-world security incidents and result in data breaches or tampering, especially in multi-user environments. Moreover, developers appreciated our efforts to address these vulnerabilities. For instance, the OceanBase team confirmed all 10 of our BAC reports as zero-day vulnerabilities. Notably, the bounty program of OceanBase [44] awarded prizes for these reports, underscoring Beacon's effectiveness in enhancing real-world DBMS access control.

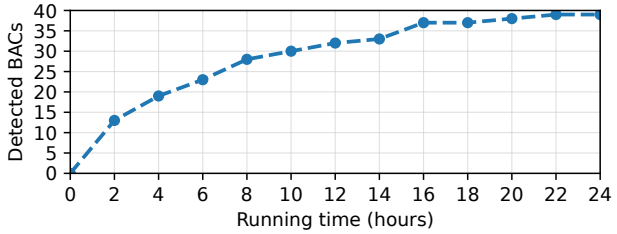

Fig. 12. Cumulative unique BAC vulnerabilities detected over time.

Results of efficiency. To better understand Beacon's BAC detection efficiency, we tracked the cumulative unique BAC vulnerabilities detected by Beacon in Figure 12. Within the first 12 hours, Beacon discovered 32 of the 39 vulnerabilities (approximately 84%), demonstrating its efficiency in

discovering BAC vulnerabilities within a relatively short time. In later hours, Beacon discovered 7 BAC vulnerabilities requiring more complex privilege configurations and longer SQL command sequences to trigger, efficiently exploring the deeper logic of DBMS access control.

Table 2. The existence duration of detected vulnerabilities.

Existence Duration	≤1 year	2	3	4	5	6	>6 years
Total Vulnerabilities	4	9	5	3	5	6	7

Vulnerability Existence Duration. We analyzed how long the detected vulnerabilities had persisted in these DBMSs before discovery. To calculate this duration, we subtracted the release date of the earliest vulnerable version from that of the latest version. The results are shown in Table 2. Among the detected vulnerabilities, the median lifespan was approximately 4 years. Specifically, 7 vulnerabilities had existed for over 6 years, with the longest one lasting 13 years. The most long-lasting one was discovered in PostgreSQL, where `CREATE TYPE` allows a user to create a range type using a subtype on which the user has no privileges, enabling privilege escalation through accessing unauthorized data types. Moreover, 10 of the BAC vulnerabilities have already existed since their corresponding commands were first introduced and supported in the target DBMSs. These vulnerabilities affected all DBMS release versions that support the BAC-inducing commands.

6.3 Selected Vulnerabilities Found by Beacon

To demonstrate the practical impact and the underlying root causes of these BAC vulnerabilities detected by Beacon, we provide three case studies, corresponding to each detection rule. Each study illustrates a unique type of BAC flaw arising from unauthorized objects appearing in successful SQL commands, result sets, and error messages, respectively.

```
root     CREATE USER user2;
         -- Grant the CREATE ROUTINE privilege to user
         'regular'
root     GRANT CREATE ROUTINE ON *.* TO regular;
regular  CREATE PROCEDURE unwant_grant()
         BEGIN GRANT ALL ON *.* TO user2; END
         -- Broken access control: the procedure with an
         unprivileged command is executed successfully
regular  CALL unwant_grant();
```

Expected:	Actual:
Access denied: you need the GRANT privilege	Query OK

Fig. 13. A BAC vulnerability in OceanBase that allows a user to execute arbitrary SQL commands.

Case Study 1: `CREATE PROCEDURE` in OceanBase. Figure 13 illustrates a BAC vulnerability in OceanBase's `CREATE PROCEDURE` command, stemming from flawed permission checks. A user possessing only the `CREATE ROUTINE` privilege can nevertheless embed arbitrary commands in the procedure's routine body. These commands are subsequently executed with elevated privileges via a `CALL` command, even without the necessary privileges, which is an act of privilege escalation. For example, an attacker could embed a `GRANT ALL` statement to escalate privileges by granting full access to an arbitrary user. Beacon triggered this vulnerability by granting the `CREATE PROCEDURE` privilege to the regular user. Beacon then identified the flaw via the unauthorized username 'user2' in the successful `CREATE ROUTINE` command, which violated the system catalog consistency.

The root cause is that OceanBase only checks whether the user holds the CREATE ROUTINE privilege while failing to validate commands in the routine body. As a result, the procedure is allowed to include commands for which the user has no authorization. Moreover, at execution time, OceanBase still does not verify the commands in the routine body. Consequently, a user with the CREATE ROUTINE privilege can execute any SQL command within the procedure body. From this vulnerability, we learned that databases must carefully check privileges for compound commands, such as PL/SQL blocks or PREPARE commands. This check must occur at either the creation or execution stage to ensure all involved commands have appropriate privileges.

root	`CREATE TABLE tbl1 (a TEXT);`
	`-- Disable SELECT privilege from 'regular'`
root	`REVOKE SELECT ON tbl1 FROM regular;`
regular	`SHOW COLLATION;`
regular	`SHOW DATABASES;`
regular	`SHOW FUNCTION STATUS;`
	`-- Broken access control in the result set`
regular	`SHOW OPEN TABLES;`
Expected: Output: Empty Set	Actual: Output: {'tbl1'}

Fig. 14. A BAC vulnerability in MySQL that leaks names of unauthorized tables by SHOW OPEN TABLES.

Case Study 2: SHOW OPEN TABLES in MySQL. Figure 14 shows a BAC vulnerability in MySQL, where SHOW OPEN TABLES can reveal the names of all tables used by any user within the DBMS. This results in unauthorized disclosure of the database structure belonging to other users. Beacon triggered this BAC by revoking the SELECT privilege on tbl1 from the regular user, and identified the vulnerability via the unauthorized table name 'tbl1' in the result set of SHOW OPEN TABLES.

The root cause is that MySQL caches prior permission check results to accelerate performance. When SHOW OPEN TABLES queries this information, it bypasses the appropriate permission checks on the returned tables. Consequently, the query leaks table names that should remain inaccessible to the regular user. From this case, we learn that caching the result of permission checks is a common optimization in DBMSs, yet developers must handle these cached results carefully. Possible measures include separating caches across different users or running renewed permission checks on cached data before exposing it to users.

root	`CREATE TABLE tbl1 (col1 INT, col2 INT, col3 INT);`
	`-- Grant no privilege to user 'regular'`
root	`REVOKE ALL ON tbl1 FROM regular;`
	`-- Broken access control in error messages`
regular	`INSERT INTO tbl1 VALUES (DEFAULT, DEFAULT, DEFAULT);`
	`-- ERROR: Column has no default value, column=col1.`
regular	`INSERT INTO tbl1 VALUES (NULL, DEFAULT, DEFAULT);`
	`-- ERROR: Column has no default value, column=col2.`
regular	`INSERT INTO tbl1 VALUES (NULL, NULL, DEFAULT);`
	`-- ERROR: Column has no default value, column=col3.`

Fig. 15. A BAC vulnerability in StarRocks that leaks column names in error messages of INSERT commands.

Case Study 3: INSERT INTO tbl VALUES(DEFAULT) in StarRocks. Figure 15 represents a BAC vulnerability in StarRocks, where the error messages generated by an INSERT command can

disclose the unauthorized column names owned by other users. This allows an attacker to infer the table structures of these users or deduce the services they might be running on the database. This compromises the confidentiality of the target user's database logic. BEACON triggered this vulnerability by revoking all privileges from the regular user. It then identified the vulnerability via the unauthorized column names 'col1' to 'col3' appearing within the error message of `INSERT`.

The fundamental flaw is that StarRocks performs multiple checks during an `INSERT` operation (e.g., verifying query syntax, looking up column definitions, and checking for default values), but defers the actual permission check until it is too late. Consequently, errors occurring before the privilege validation stage trigger an immediate failure, returning a verbose message containing the unauthorized column names to the user. To mitigate this, developers should enforce permission checks as early as possible, preventing unauthorized objects from leaking through early-stage failure messages. If necessary, a secondary filter can inspect error messages to ensure they do not disclose any unauthorized information.

6.4 Design Choice Analysis

This section analyzes three critical design choices: the synthesis of SQL operation sequences in Privilege-Aware Test Case Generation, the selection of system catalogs in Unauthorized Object Identification, and the allowlisted cases in Catalog Consistency Validation. Specifically, the operation sequence synthesis aims to test the DBMS's access control implementation across diverse SQL command types. The process for selecting system catalogs aims to enhance test efficiency by avoiding time-consuming catalog queries. The allowlisted cases of BEACON's detection rules are to guarantee accuracy by systematically excluding known sources of false positives stemming from inherent command properties or database output formats.

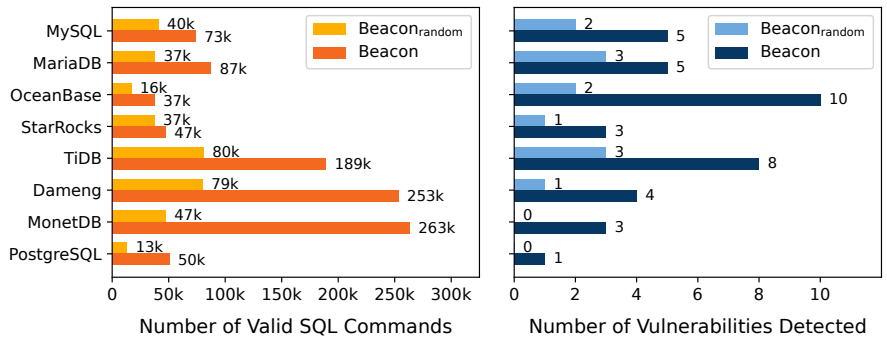

Fig. 16. The number of generated valid SQL commands and detected BAC vulnerabilities between BEACON$_{random}$ and BEACON in 24 hours.

Operation Sequence Synthesis in Privilege-Aware Test Cases. In Section 4.2, we presented the synthesis of SQL operation sequences during the generation of privilege-aware test cases. To cover a wide variety of SQL command types, BEACON first constructs initial DDL commands to create diverse database objects, and then synthesize a series of DDL, DML, and DQL commands based on the internal schema. This approach can generate various types of valid SQL commands, as a SQL command usually depends on some existing database objects. To assess the effectiveness of BEACON's operation sequence synthesis, we implemented BEACON$_{random}$, which generates DDL, DML, and DQL commands randomly without the internal schema, and compared the test case generation and vulnerability detection between BEACON and BEACON$_{random}$. As shown in Figure

16, the comparative results demonstrate that Beacon produced more valid SQL commands across all evaluated DBMSs. More importantly, the number of detected BAC vulnerabilities is also higher when using Beacon. Specifically, Beacon identified 39 BAC vulnerabilities in total, whereas Beacon$_{random}$ detected only 12. These results confirm that the operation sequence synthesis is effective at generating diverse valid test cases and discovering BAC vulnerabilities in DBMSs.

Table 3. Number of system catalogs and the average time required for the associated queries during Unauthorized Object Identification. Beacon$^-$ employs all available system catalogs, while Beacon only uses the minimal subset of catalogs.

DBMS	Number of Catalogs		Query Time (s)	
	Beacon$^-$	Beacon	Beacon$^-$	Beacon
MySQL	233	10	5.410	0.023
MariaDB	300	10	1.449	0.019
OceanBase	1247	10	22.174	0.434
StarRocks	53	9	1.373	0.257
TiDB	156	10	2.803	0.024
Dameng	308	4	9.899	0.228
MonetDB	137	7	2.670	0.028
PostgreSQL	212	8	22.588	0.933
Total	2646	68	68.366	1.946

System Catalog Selection for Identifying Unauthorized Objects. As described in Section 4.3, Beacon selects a minimal subset of system catalogs that expose metadata for all database objects. In practice, each DBMS provides a substantially larger set of system catalogs, offering information on performance, storage, and other aspects. Moreover, when a DBMS processes a system catalog query, it sometimes filters the query results by scanning all user tables for permission checks, which can incur significant latency. Therefore, identifying accessible objects across *all* available system catalogs would be extremely time-consuming. To confirm this, we examined the overlap of system catalog information across different databases, and measured both performance and vulnerability detection when using all system catalogs for unauthorized object identification. We found that expanding the scope to include additional system catalogs significantly slowed execution, as shown in Table 3, far exceeding the runtime when focusing on just the minimal subset of catalogs. Using all system catalogs is 35.1× slower on average, with the largest slowdown observed in MySQL (235.2×). Moreover, the expansion did not reveal new vulnerabilities. Consequently, for unauthorized object identification, we rely on the minimal subset of catalogs to improve overall test efficiency.

Allowlisted Cases of Beacon's Detection Rules. As described in Section 4.4, Rule$_{\text{cmd}}$ and Rule$_{\text{errmsg}}$ each incorporate one condition for allowlisted cases that are not treated as a BAC. Without these two filtering conditions, Beacon would produce many false positives. We verified this by disabling the conditions of allowlisted cases for testing, namely Rule$^-_{\text{cmd}}$ and Rule$^-_{\text{errmsg}}$, and the results are shown in Table 4. The results indicate that removing these conditions generates a significant number of false positives, severely undermining Beacon's overall accuracy. For instance, Rule$^-_{\text{cmd}}$ would classify every command containing `WHERE table_name = 'xxx'` as a BAC vulnerability, even when the returned result set was empty. Similarly, Rule$^-_{\text{errmsg}}$ treated any error message like "DROP command denied to table `'xxx'`" as BAC, but they are actually false positives. These results show the necessity of including these allowlist conditions in Rule$_{\text{cmd}}$ and Rule$_{\text{errmsg}}$.

Nevertheless, Beacon occasionally reports harmless commands as false positives. In our evaluation, it yielded three false positives due to the Rule$_{\text{cmd}}$ logic. These false positives occurred

Table 4. False positives of Beacon's detection rules, where $\text{Rule}^{-}_{\text{cmd}}$ and $\text{Rule}^{-}_{\text{errmsg}}$ utilize the rules of Rule_{cmd} and $\text{Rule}_{\text{errmsg}}$ but disable allowlist conditions.

DBMS	False Positives of Beacon's Detection Rules				
	$\text{Rule}^{-}_{\text{cmd}}$	$\text{Rule}^{-}_{\text{errmsg}}$	Rule_{cmd}	$\text{Rule}_{\text{errmsg}}$	$\text{Rule}_{\text{result}}$
MySQL	4	45	0	0	0
MariaDB	3	46	0	0	0
OceanBase	7	32	2	0	0
StarRocks	2	17	0	0	0
TiDB	2	33	1	0	0
Dameng	4	16	0	0	0
MonetDB	1	30	0	0	0
PostgreSQL	0	24	0	0	0
Total	23	243	3	0	0

when a command returned a successful status ("Query OK") but caused no actual changes in the database. For example, in TiDB, the command "`TRACE INSERT INTO tbl VALUES (100)`" will trace the execution status of the `INSERT` command. This command always reports success and returns a result set to show the insertion status, even if the insertion is canceled due to insufficient privileges. Since Beacon identifies a successful execution targeting an unauthorized object, it mistakenly reports this as a BAC vulnerability. Filtering out such potentially no-operation commands in our detection logic can eliminate the remaining false positives.

7 Discussion

Comparison with Existing Access Control Testing Works. Existing access control testing approaches primarily rely on applying static analysis methods to source code. However, since the loops and recursive function calls occur in the DBMSs' source code that performs permission checks, directly using static analysis is difficult. To assess whether these approaches are effective for DBMSs, we trace runtime function calls when the DBMS executes SQL commands to analyze the execution paths instead of static analysis.

Table 5. The valid vulnerabilities (true positives, TP) and false positives (FP) of the BAC detection when applying existing access control testing methods to MySQL.

Tool	Assumption	TP	FP
PeX [94]	The same operation requires the same permission checks	1	142
MACE [39]	Access to the same resource needs the same permission checks	0	97
MPChecker [31]	Each operation has at least one permission check	1	179
Beacon	The runnable command is consistent with the system catalogs	5	0

Specifically, we instrument the source code of MySQL to trace which permission checks and privileged operations are triggered during SQL execution. The instrumented functions for permission checks cover different object levels of privilege verification, such as check_table_access (table-level) and check_column_privilege (column-level). The instrumented privileged functions are those used for table read-write-update operations, including functions like ha_write_row (inserts a row into a table) and ha_update_row (modifies an existing row). Once we run MySQL with instrumentation

on a test suite, we obtain detailed execution paths of these permission checks and privileged operations for each command. We then adopt the criteria proposed in prior works [31, 39, 94]. For every execution path labeled as BAC by them, we inspect each SQL command's actual runtime behavior to determine whether it is a false positive. As shown in Table 5, these methods produce numerous false positives. For example, the command "`SELECT * FROM (select 1) AS a2`" is incorrectly identified as BAC by these methods, as it reads and writes a table named `a2` without any permission check during execution. However, this reflects the expected behavior of MySQL's access control since the table `a2` is a temporary table used solely for handling the alias in the `SELECT` command.

Test Case Generation Approaches. In our approach, we adopt a bottom-up generation method to compose test cases from DDL, DML, DQL, and DCL commands, thereby covering a wide range of SQL operations and object types in a unified workflow. Moreover, by leveraging catalog consistency as the test oracle, our approach can detect BAC vulnerabilities without explicitly modeling the underlying privilege model of each DBMS. Another test case generation approach for detecting consistency issues is mutation-based testing, which has been widely used in compiler testing and optimization testing [26, 70, 71, 81]. Specifically, mutation-based methods for consistency testing transform a test case into semantically equivalent variants through certain mutation strategies, and then check whether execution results remain consistent before and after mutation. By accumulating non-trivial mutations over multiple iterations, the resulting test inputs can become more complex, which helps exercise deeper execution paths. However, adapting these methods to BAC testing presents several challenges. For instance, the mutations must preserve the privilege-relevant behaviors under the DBMS privilege model, which requires designing privilege-preserving rewrite rules and tuning mutation operators accordingly. Designing an efficient approach that combines bottom-up generation and privilege-preserving mutation to ensure both broad coverage and deeper exploration of permission checking paths is a promising direction for future work.

Scope and More BAC Types Support. Beacon is based on the insight that objects invisible in system catalogs should be unauthorized for users to access. Accordingly, it supports two BAC categories: privilege escalation, when a command succeeds on objects on which the user holds no privileges; and unauthorized information disclosure, when information of such non-privileged objects is revealed through result sets and error messages.

Despite this scope, there are additional types of BAC vulnerabilities that Beacon does not currently support but could address in the future. (1) First, Beacon assumes that catalogs correctly hide unauthorized objects from users. If catalogs expose unauthorized objects, Beacon may observe no inconsistency. Incorporating cross-catalog difference analysis could reduce such misses. (2) In addition, Beacon does not check whether the types of privileges a user actually holds on an object match those required by the operation on that object. Therefore, Beacon may miss BAC vulnerabilities where users have partial privileges on an object but attempt to perform operations requiring other privileges for that object. This gap can be mitigated by systematically modeling the privileges required by different SQL operations and database objects. (3) Moreover, some cache-related BAC vulnerabilities might be masked by executing `REVOKE` commands or querying system catalogs, which may refresh internal caches in DBMSs. To address this, Beacon can be extended to perform difference analysis by comparing outputs before and after an explicit cache refresh. (4) Finally, Beacon targets the grant–revoke model for object privileges and does not cover fine-grained mechanisms such as Row-Level Security (RLS) [20]. Extending Beacon's consistency principle to row visibility by comparing RLS-permitted rows with query results is a possible extension.

Defending against BAC vulnerabilities with database auditing and firewalls. Database auditing and firewalls are standard techniques in modern DBMSs for monitoring and blocking malicious activities. Database auditing records operations for later review, while firewalls proactively block suspicious commands before execution [30, 50, 56]. These solutions can mitigate some

BAC risks by flagging suspicious activity against protected objects or blocking known malicious commands. However, auditing and firewall policies are manually defined and maintained by administrators for each user. They can only guard against known BAC vulnerabilities by manually crafting corresponding allowlists and blocklists. Moreover, while these techniques operate primarily via command-filtering policies (e.g., blocking a `DROP` or `CREATE` command using predefined patterns), they lack mechanisms to prevent unauthorized information disclosure in result sets or error messages. This demonstrates the importance of Beacon, which can automatically detect BAC vulnerabilities to improve the implementation of real-world DBMSs' access control systems.

8 Related Work

Access Control Testing. Access control testing has been applied across diverse systems, such as database-backed applications and distributed cloud environments [21, 22, 31, 39, 94]. Existing approaches primarily focus on analyzing source code paths and enforcing predefined policy rules. In practice, they are rule-driven, operating under the key assumption that identical operations or resource accesses must undergo consistent permission checks. For instance, ACHyb [21] employs static analysis to detect vulnerable paths and utilizes dynamic analysis to minimize false positives. Similarly, MPChecker [31] identifies paths from program entry points to privileged operations that lack proper permission checks, while MACE [39] verifies that an application consistently uses the same authorization context when accessing the same resource along different paths.

Beacon differs from these approaches by employing a dynamic validation technique that operates independently of any fixed permission models. Instead of analyzing code paths, Beacon directly detects inconsistencies by comparing the execution results of SQL commands with the permission information in system catalogs. This dynamic approach allows Beacon to adapt to various flexible authorization scenarios. Additionally, verifying the consistency between system catalogs and actual operations ensures cross-DBMS compatibility.

DBMS Testing. DBMSs are complex software systems and thus prone to problems. DBMS testing efforts can be categorized into three types: testing for logic, crash, and performance bugs. Logic bug testing [10, 15, 18, 25, 28, 70, 71] focuses on verifying the correctness of execution results. These studies typically concentrate on constructing logic test oracles. For example, NoREC [70] detects differences in result sets between optimizing and non-optimizing query versions, and TLP [71] detects whether composed results of partitioning queries are equivalent to the original query's results. Crash bug testing [16, 17, 27, 74, 82, 93, 95] aims to identify circumstances under which DBMSs might crash. These approaches focus on generating valid SQL commands and monitoring DBMS crashes. For instance, SQLsmith [74] generates SQL commands based on syntax modeling, and Squirrel [95] uses coverage feedback to guide the mutation of SQL commands. Performance bug testing [2, 24, 29, 91, 92] checks whether execution results are returned within reasonable time limits. For example, Apollo [24] compares the execution time of different DBMS versions.

In contrast to traditional DBMS testing approaches, Beacon directly targets BAC vulnerabilities. We leverage an observable invariant in DBMS catalogs: each user should only view and operate on the objects that the user is authorized to access. By correlating catalog visibility with the executability of privileged commands, our technique uncovers subtle BAC vulnerabilities that have evaded detection in prior testing frameworks.

DBMS Access Control. Recent research on DBMS access control has primarily focused on developing new access control mechanisms to improve security and effectiveness [5, 11, 40, 61–63, 83]. Pappachan et al. proposed an approach that functions as a security gate to protect against sensitive information disclosure via inferences [63]. Sieve is designed to scale Fine-Grained Access Control (FGAC) to scenarios involving thousands of access control policies [62]. Wang et al. introduced a correctness criterion for FGAC and proposed a new access control algorithm satisfying

it [83]. Paduroiu et al. presented Membrane, an FGAC mechanism for Apache Spark, which supports row-level and column-level access control while maintaining full API compatibility with Apache Spark [61]. All these works significantly enhance DBMS access control, rendering it more robust and adaptable to various application scenarios.

Conversely, unlike these studies centered around new mechanisms, Beacon focuses on detecting BAC vulnerabilities within existing DBMS access control implementations. Beacon discovers these vulnerabilities by automatically validating the consistency between database objects available in system catalogs and those accessible during query execution. This approach complements existing solutions by targeting implementation flaws that they may overlook.

Database Auditing and Firewall. Database auditing and database firewall techniques have witnessed widespread adoption in DBMSs. Database firewall techniques are effective in defending against malicious attacks. They examine SQL statements before execution based on predefined attack patterns or block rules, to determine whether to execute or reject each statement [30, 50, 56]. For example, ProxySQL utilizes allowlists, blocklists, and generic block rules to filter queries [30]. Database auditing records high-risk activities in audit logs according to static rules, such as sensitive data access and user creation [33, 55, 57, 77]. These logs require review by administrators or other external processes. For example, Oracle's Fine-Grained Auditing provides administrators with detailed, query-level audit trails [57]. Typically, database auditing and firewall methods do not detect mismatches between intended and actual permission checks. Consequently, they may miss potential flaws in the implementation of database access control.

In contrast, Beacon focuses on detecting BAC vulnerabilities within the DBMSs' access control, rather than blocking or recording malicious activities externally. Unlike auditing and firewalls, which do not examine whether permission checks are correctly implemented, Beacon automatically identifies mismatches between object visibility in system catalogs and actual command executability. This approach discovers BAC vulnerabilities without relying on block rules or manual log analysis.

9 Conclusion

We proposed Beacon, a systematic approach to detecting broken access control vulnerabilities in DBMSs by comparing system catalog visibility against actual privilege enforcement. By focusing on diverse access control scenarios and cross-user catalog comparisons, Beacon pinpoints discrepancies that indicate unauthorized object access or privilege escalation. Our evaluation of eight popular DBMSs detected 39 previously unknown BAC vulnerabilities. All of these identified issues are confirmed by DBMS vendors, underscoring the severity of BAC vulnerabilities in DBMSs. In future work, we plan to expand Beacon to a broader range of DBMS platforms and further refine its privilege-analysis capabilities.

Data-Availability Statement

The artifact of this paper is available on Zenodo [64].

Acknowledgments

We sincerely thank the anonymous reviewers for their insightful feedback. This research is sponsored in part by the NSFC Program (No.62302256, 62525207, U2441238, 625B2100), the Fundamental Research Funds for the Central Universities (No.JKF-2025023600609), the CCF-NSFOCUS Kun-Peng Scientific Research Fund (No.2025002), and the Huawei-Tsinghua Database Testing Research Project (No.20252001670).

References

[1] Ahmed Anas, Salwa Elgamal, and Basheer Youssef. 2024. Survey on detecting and preventing web application broken access control attacks. *International Journal of Electrical and Computer Engineering (IJECE)* 14, 1 (2024), 772–781. doi:10.11591/ijece.v14i1.pp772-781

[2] Jinsheng Ba and Manuel Rigger. 2024. Cert: Finding performance issues in database systems through the lens of cardinality estimation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 1–13. doi:10.1145/3597503.3639076

[3] Elisa Bertino, Gabriel Ghinita, Ashish Kamra, et al. 2011. Access control for databases: Concepts and systems. *Foundations and Trends® in Databases* 3, 1–2 (2011), 1–148. doi:10.1561/9781601984173

[4] Elisa Bertino and Ravi Sandhu. 2005. Database security-concepts, approaches, and challenges. *IEEE Transactions on Dependable and secure computing* 2, 1 (2005), 2–19. doi:10.1109/TDSC.2005.9

[5] Jianneng Cao, Barbara Carminati, Elena Ferrari, and Kian-Lee Tan. 2009. Acstream: Enforcing access control over data streams. In *2009 IEEE 25th International Conference on Data Engineering (ICDE)*. IEEE, 1495–1498. doi:10.1109/ICDE.2009.25

[6] MariaDB Corporation. 2025. MariaDB. https://mariadb.org/. Accessed: Feb. 27, 2026.

[7] The MITRE Corporation. 2026. CWE-284: Improper Access Control. https://cwe.mitre.org/data/definitions/284.html. Accessed: Feb. 27, 2026.

[8] Dameng. 2025. DM8 - A New-Generation Large General Relational Database. https://en.dameng.com/. Accessed: Feb. 27, 2026.

[9] Dameng. 2025. Privileges for using ArcGIS with Dameng. https://pro.arcgis.com/en/pro-app/latest/help/data/databases/privileges-dameng.htm. Accessed: Feb. 27, 2026.

[10] Wenqian Deng, Jie Liang, Zhiyong Wu, Jingzhou Fu, and Yu Jiang. 2025. Detecting Logic Bugs in DBMSs via Equivalent Data Construction. *Proceedings of the ACM on Management of Data* 3, 6 (2025), 1–25. doi:10.1145/3769779

[11] Sabrina De Capitani Di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. 2007. Over-encryption: Management of access control evolution on outsourced data. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*. 123–134. https://dl.acm.org/doi/10.5555/1325851.1325869

[12] Docker. 2025. Docker Hub. https://hub.docker.com/. Accessed: Feb. 27, 2026.

[13] Joint Task Force. 2020. *Security and privacy controls for information systems and organizations.* Technical Report. National Institute of Standards and Technology. doi:10.6028/NIST.SP.800-53r5

[14] The Apache Software Foundation. 2025. Apache CouchDB. https://couchdb.apache.org. Accessed: Feb. 27, 2026.

[15] 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. doi:10.1145/3597503.3639210

[16] 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 (ASE)*. 1–12. doi:10.1145/3551349.3560431

[17] Jingzhou Fu, Jie Liang, Zhiyong Wu, Yanyang Zhao, Shanshan Li, and Yu Jiang. 2025. Understanding and Detecting SQL Function Bugs: Using Simple Boundary Arguments to Trigger Hundreds of DBMS Bugs. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys)*. 1061–1076. doi:10.1145/3689031.3696064

[18] Ying Fu, Zhiyong Wu, Yuanliang Zhang, Jie Liang, Jingzhou Fu, Yu Jiang, Shanshan Li, and Xiangke Liao. 2025. THANOS: DBMS Bug Detection via Storage Engine Rotation Based Differential Testing. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1–12. doi:10.1109/ICSE55347.2025.00257

[19] The PostgreSQL Global Development Group. 2025. PostgreSQL Documentation: Role Attributes. https://www.postgresql.org/docs/current/role-attributes.html. Accessed: Feb. 27, 2026.

[20] The PostgreSQL Global Development Group. 2025. PostgreSQL Documentation: Row Security Policies. https://www.postgresql.org/docs/current/ddl-rowsecurity.html. Accessed: Feb. 27, 2026.

[21] Yang Hu, Wenxi Wang, Casen Hunger, Riley Wood, Sarfraz Khurshid, and Mohit Tiwari. 2021. ACHyb: A hybrid analysis approach to detect kernel access control vulnerabilities. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 316–327. doi:10.1145/3468264.3468627

[22] Yongheng Huang, Chenghang Shi, Jie Lu, Haofeng Li, Haining Meng, and Lian Li. 2024. Detecting Broken Object-Level Authorization Vulnerabilities in Database-Backed Applications. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2934–2948. doi:10.1145/3658644.3690227

[23] ISO/IEC. 2023. Information technology — Database languages — SQL — Part 2: Foundation (ISO/IEC 9075-2:2023). https://www.iso.org/standard/76584.html. ISO/IEC Standard.

[24] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2020. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. In *Proceedings of the 46th International Conference on Very Large*

Data Bases (VLDB). 1–12. doi:10.14778/3357377.3357382

[25] Jinhui Lai, Chi Zhang, Bingyan Li, Chenglin Liang, Jie Liang, Zhiyong Wu, Jingzhou Fu, Yu Jiang, and Zichen Xu. 2025. SRS: Detecting Logic Bugs of Join Implementation in DBMSs via Set Relation Synthesis. *Proceedings of the ACM on Management of Data* 3, 6 (2025), 1–24. doi:10.1145/3769828

[26] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *Acm Sigplan Notices* 50, 10 (2015), 386–399. doi:10.1145/2858965.2814319

[27] 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)*. 668–681. doi:10.1109/ICDE55515.2023.00057

[28] Jie Liang, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Chengnian Sun, and Yu Jiang. 2024. Mozi: Discovering DBMS Bugs via Configuration-Based Equivalent Transformation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 1–12. doi:10.1145/3597503.3639112

[29] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic detection of performance bugs in database systems using equivalent queries. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. 225–236. doi:10.1145/3510003.3510093

[30] ProxySQL LLC. 2025. ProxySQL: Firewall Whitelist. https://proxysql.com/documentation/Firewall-whitelist/. Accessed: Feb. 27, 2026.

[31] Jie Lu, Haofeng Li, Chen Liu, Lian Li, and Kun Cheng. 2022. Detecting missing-permission-check vulnerabilities in distributed cloud systems. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2145–2158. doi:10.1145/3548606.3560589

[32] Toby Mao. 2023. SQLGlot Website. https://sqlglot.com/sqlglot.html. Accessed: Feb. 27, 2026.

[33] MariaDB. 2025. MariaDB Community Audit. https://mariadb.com/docs/server/reference/plugins/mariadb-audit-plugin. Accessed: Feb. 27, 2026.

[34] Trend Micro. 2026. About Trend Micro. https://www.trendmicro.com/en_us/about.html. Accessed: Feb. 27, 2026.

[35] Trend Micro. 2026. Apache CouchDB Vulnerabilities Permit Monero Mining. https://www.trendmicro.com/en/research/18/b/vulnerabilities-apache-couchdb-open-door-monero-miners.html. Accessed: Feb. 27, 2026.

[36] MITRE. 2025. CWE-200: Exposure of Sensitive Information to an Unauthorized Actor. https://cwe.mitre.org/data/definitions/200.html. Accessed: Feb. 27, 2026.

[37] MITRE. 2025. Privilege Escalation. https://attack.mitre.org/tactics/TA0004/. Accessed: Feb. 27, 2026.

[38] MonetDB 2025. The Database System to speed up your Analytical Jobs. https://www.monetdb.org/. Accessed: Feb. 27, 2026.

[39] Maliheh Monshizadeh, Prasad Naldurg, and VN Venkatakrishnan. 2014. Mace: Detecting privilege escalation vulnerabilities in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 690–701. doi:10.1145/2660267.2660337

[40] Rimma V. Nehme, Hyo-Sang Lim, and Elisa Bertino. 2010. FENCE: Continuous access control enforcement in dynamic data stream environments. In *Proceedings of IEEE 26th International Conference on Data Engineering (ICDE)*. 940–943. doi:10.1109/ICDE.2010.5447899

[41] National Vulnerability Database (NVD). 2025. CVE-2017-12635 Detail. https://nvd.nist.gov/vuln/detail/CVE-2017-12635. Accessed: Feb. 27, 2026.

[42] OceanBase. 2025. CREATE TRIGGER. https://en.oceanbase.com/docs/common-oceanbase-database-10000000001974512. Accessed: Feb. 27, 2026.

[43] OceanBase. 2025. The Multi-cloud Distributed Database for Mission-critical Workloads at Any Scale. https://en.oceanbase.com/. Accessed: Feb. 27, 2026.

[44] OceanBase. 2025. OceanBase Security Response Protocol. https://en.oceanbase.com/security. Accessed: Feb. 27, 2026.

[45] Oracle 2025. MySQL. https://www.mysql.com/. Accessed: Feb. 27, 2026.

[46] Oracle. 2025. MySQL Documentation: Access Control and Account Management. https://dev.mysql.com/doc/refman/9.2/en/access-control.html. Accessed: Feb. 27, 2026.

[47] Oracle. 2025. MySQL Documentation: GRANT Statement. https://dev.mysql.com/doc/refman/9.2/en/grant.html. Accessed: Feb. 27, 2026.

[48] Oracle. 2025. MySQL Documentation: INFORMATION_SCHEMA Introduction. https://dev.mysql.com/doc/refman/9.2/en/information-schema-introduction.html. Accessed: Feb. 27, 2026.

[49] Oracle. 2025. MySQL Documentation: INFORMATION_SCHEMA Table Reference. https://dev.mysql.com/doc/refman/9.2/en/information-schema-table-reference.html. Accessed: Feb. 27, 2026.

[50] Oracle. 2025. MySQL Documentation: MySQL Enterprise Firewall. https://dev.mysql.com/doc/refman/9.2/en/firewall.html. Accessed: Feb. 27, 2026.

[51] Oracle. 2025. MySQL Documentation: Privileges Provided by MySQL. https://dev.mysql.com/doc/refman/9.2/en/privileges-provided.html. Accessed: Feb. 27, 2026.

[52] Oracle. 2025. MySQL Documentation: Show Open Tables Statements. https://dev.mysql.com/doc/refman/9.2/en/show-open-tables.html. Accessed: Feb. 27, 2026.

[53] Oracle. 2025. MySQL Documentation: The INFORMATION_SCHEMA COLUMNS Table. https://dev.mysql.com/doc/refman/9.2/en/information-schema-columns-table.html. Accessed: Feb. 27, 2026.

[54] Oracle. 2025. MySQL Documentation: Using Roles. https://dev.mysql.com/doc/refman/9.2/en/roles.html. Accessed: Feb. 27, 2026.

[55] Oracle. 2025. MySQL Enterprise Audit. https://www.mysql.com/products/enterprise/audit.html. Accessed: Feb. 27, 2026.

[56] Oracle. 2025. Oracle Database Firewall. https://www.oracle.com/technetwork/cn/products/database-firewall/overview/index.html. Accessed: Feb. 27, 2026.

[57] Oracle. 2025. Using Fine-Grained Auditing. https://www.oracle.com/technical-resources/articles/middleware/idm-fga-otn.html. Accessed: Feb. 27, 2026.

[58] Oracle. 2026. MySQL Documentation: ALTER TABLE Statement. https://dev.mysql.com/doc/refman/9.2/en/alter-table.html. Accessed: Feb. 27, 2026.

[59] OWASP. 2026. OWASP Foundation. https://owasp.org/. Accessed: Feb. 27, 2026.

[60] OWASP. 2026. Testing for SQL Injection. https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05-Testing_for_SQL_Injection. Accessed: Feb. 27, 2026.

[61] Andrei Paduroiu, Sungheun Wi, Yan Yan, Roni Burd, Ruhollah Farchtchi, and Giovanni Matteo Fumarola. 2024. Membrane-Safe and Performant Data Access Controls in Apache Spark in the Presence of Imperative Code. *Proc. VLDB Endow.* 17, 12 (2024), 3813–3826. doi:10.14778/3685800.3685808

[62] Primal Pappachan, Roberto Yus, Sharad Mehrotra, and Johann-Christoph Freytag. 2020. Sieve: a middleware approach to scalable access control for database management systems. *Proc. VLDB Endow.* 13, 12 (2020), 2424–2437. doi:10.14778/3407790.3407835

[63] Primal Pappachan, Shufan Zhang, Xi He, and Sharad Mehrotra. 2022. Don't be a tattle-tale: Preventing leakages through data dependencies on access control protected data. *Proc. VLDB Endow.* 15, 11 (2022), 2437–2449. doi:10.14778/3551793.3551805

[64] Zongrui Peng, Jingzhou Fu, Zhiyong Wu, Jie Liang, Xiangdong Huang, Dalong Shi, and Yu Jiang. 2026. Beacon: Detecting Broken Access Control Vulnerabilities in DBMSs via System Catalog Consistency Validation. doi:10.5281/zenodo.18669646

[65] PingCAP. 2026. TiDB. https://github.com/pingcap/tidb. Accessed: Feb. 27, 2026.

[66] PortSwigger. 2025. Access control vulnerabilities and privilege escalation. https://portswigger.net/web-security/access-control. Accessed: Feb. 27, 2026.

[67] PortSwigger. 2026. Examining the database in SQL injection attacks. https://portswigger.net/web-security/sql-injection/examining-the-database. Accessed: Feb. 27, 2026.

[68] PortSwigger. 2026. PortSwigger. https://portswigger.net/. Accessed: Feb. 27, 2026.

[69] PostgreSQL 2026. PostgreSQL. https://www.postgresql.org/. Accessed: Feb. 27, 2026.

[70] 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 (ESEC/FSE)*. 1140–1152. doi:10.1145/3368089.3409710

[71] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30. doi:10.1145/3428279

[72] Jerome H Saltzer and Michael D Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308. doi:10.1109/PROC.1975.9939

[73] Ravi S Sandhu. 1998. Role-based access control. In *Advances in computers*. Vol. 46. Elsevier, 237–286. doi:10.1016/S0065-2458(08)60206-5

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

[75] Abraham Silberschatz, Henry F Korth, Shashank Sudarshan, et al. 2002. *Database system concepts*. Vol. 5. McGraw-Hill New York.

[76] StarRocks 2025. Fast, Fresh, and Flexible: Analytics With No Compromise. https://www.starrocks.io/. Accessed: Feb. 27, 2026.

[77] StarRocks. 2025. Manage audit logs within StarRocks via AuditLoader. https://docs.starrocks.io/docs/administration/management/audit_loader/. Accessed: Feb. 27, 2026.

[78] StarRocks. 2025. StarRocks Documentation: Information Schema. https://docs.starrocks.io/docs/sql-reference/information_schema/. Accessed: Feb. 27, 2026.

[79] StarRocks. 2025. StarRocks Documentation: Privileges supported by StarRocks. https://docs.starrocks.io/docs/administration/user_privs/privilege_item/. Accessed: Feb. 27, 2026.
[80] StarRocks. 2025. StarRocks Test Suite. https://github.com/StarRocks/starrocks/blob/main/test/sql/test_rbac/R/test_view_privilege. Accessed: Feb. 27, 2026.
[81] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*. 849–863. doi:10.1145/3022671.2984038
[82] 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 *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 328–337. doi:10.1109/ICSE-SEIP52600.2021.00042
[83] Qihua Wang, Ting Yu, Ninghui Li, Jorge Lobo, Elisa Bertino, Keith Irwin, and Ji-Won Byun. 2007. On the correctness criteria of fine-grained access control in relational databases. In *Proceedings of the 33rd international conference on Very large data bases (VLDB)*. 555–566. https://dl.acm.org/doi/10.5555/1325851.1325915
[84] Wikipedia. 2024. Database. https://en.wikipedia.org/wiki/Database. Accessed: Feb. 27, 2026.
[85] Wikipedia. 2025. Data control language. https://en.wikipedia.org/wiki/Data_control_language. Accessed: Feb. 27, 2026.
[86] Wikipedia. 2025. Data definition language. https://en.wikipedia.org/wiki/Data_definition_language. Accessed: Feb. 27, 2026.
[87] Wikipedia. 2025. Data manipulation language. https://en.wikipedia.org/wiki/Data_manipulation_language. Accessed: Feb. 27, 2026.
[88] Wikipedia. 2025. Data query language. https://en.wikipedia.org/wiki/Data_query_language. Accessed: Feb. 27, 2026.
[89] Wikipedia. 2025. Role-based access control. https://en.wikipedia.org/wiki/Role-based_access_control. Accessed: Feb. 27, 2026.
[90] Wikipedia. 2025. SQL. https://en.wikipedia.org/wiki/SQL. Accessed: Feb. 27, 2026.
[91] Zhiyong Wu, Jie Liang, Jingzhou Fu, Mingzhe Wang, and Yu Jiang. 2025. Hulk: Exploring Data-Sensitive Performance Anomalies in DBMSs via Data-Driven Analysis. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 2181–2202. doi:10.1145/3728973
[92] Zhiyong Wu, Jie Liang, Jingzhou Fu, Mingzhe Wang, and Yu Jiang. 2025. PUPPY: Finding Performance Degradation Bugs in DBMSs via Limited-Optimization Plan Construction. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1–12. doi:10.1109/ICSE55347.2025.00045
[93] 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. doi:10.1145/3533767.3534364
[94] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. 2019. PeX: A permission check analysis framework for linux kernel. In *Proceedings of the 28th USENIX Conference on Security Symposium (USENIX Security)*. 1205–1220.
[95] 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 (CCS)*. 955–970. doi:10.1145/3372297.3417260