Accio: Detecting Privilege Escalation Vulnerabilities in DBMSs via Minimum Privilege Set Calculation

Jingzhou Fu
KLISS, BNRist, School of Software
Tsinghua University
Beijing, China

Zongrui Peng
KLISS, BNRist, School of Software
Tsinghua University
Beijing, China

Jie Liang
School of Software
Beihang University
Beijing, China

Zhiyong Wu*
KLISS, BNRist, School of Software
Tsinghua University
Beijing, China

Chi Zhang*
KLISS, BNRist, School of Software
Tsinghua University
Beijing, China

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

Abstract

Access control mechanisms in DBMSs enforce user privileges for executing SQL operations. They are critical for protecting the confidentiality and integrity of stored data. However, implementation errors can lead to privilege escalation, allowing unauthorized actions that appear legitimate to the system. We refer to such errors as *privilege escalation vulnerabilities (PEVs)*, which can result in severe consequences, including denial of service, data leakage, and data tampering. Despite their serious impact, the characteristics of PEVs remain largely unexplored, and systematic detection methods are still lacking.

In this paper, we present a motivation study of 63 real-world privilege escalation cases from 5 widely used DBMSs. Our analysis shows that many vulnerabilities stem from missing privilege checks, incorrect privilege resolution, or operation-privilege mismatches. Detecting these PEVs requires careful reasoning about the interactions between SQL operations, user privileges, and execution contexts. Guided by these insights, we design and implement Accio, which detects privilege escalation vulnerabilities via minimum privilege set calculation. It models privilege hierarchies, computes the minimum set of privileges required for each SQL command, and checks consistency against expected policies. We evaluated Accio on seven widely used DBMSs, discovering 26 previously unknown PEVs, all confirmed, and 23 of which have been fixed. Accio also successfully detects historical privilege escalation cases and has been positively recognized by developers for its practical impact.

CCS Concepts

• **Security and privacy → Database and storage security**.

Keywords

DBMS, Access Control, Privilege Escalation

*Zhiyong Wu and Chi Zhang are the corresponding authors.

CCS '26, The Hague, Netherlands

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832623

ACM Reference Format:
Jingzhou Fu, Zongrui Peng, Jie Liang, Zhiyong Wu, Chi Zhang, and Yu Jiang. 2026. Accio: Detecting Privilege Escalation Vulnerabilities in DBMSs via Minimum Privilege Set Calculation. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3832623

1 Introduction

Database Management Systems (DBMSs) are fundamental components of modern software infrastructure, serving as the backend for a wide range of applications to store, query, and manage data [29, 35, 47]. As multi-user systems, DBMSs implement fine-grained access control mechanisms [31] that allow administrators to assign different privileges to different users, ensuring that regular users can execute only authorized SQL operations. To enforce this policy, each database operation is checked against the current user's assigned privileges, and the DBMS should enforce these checks correctly and consistently.

In practice, however, DBMSs may allow users to perform operations beyond their assigned privileges. Such behaviors are known as *privilege escalation* [26, 36], which enables unauthorized actions to be executed while appearing legitimate to the DBMS [39]. Privilege escalation can lead to serious consequences, including unauthorized access, data leakage, and data tampering, thereby threatening the confidentiality and integrity of stored data. A key reason for privilege escalation is the complexity of correctly implementing access control in DBMSs. Privilege semantics must be enforced consistently across multiple semantic layers, database objects, and execution contexts, which makes the implementation prone to subtle errors or oversights. We refer to such implementation errors as **privilege escalation vulnerabilities (PEVs)**, which may be exploited by unprivileged or malicious users to obtain capabilities beyond the intended design. *In this paper, we specifically focus on PEVs in SQL DBMSs, where privileges are defined over database objects and configured using SQL commands, such as* `GRANT` *commands.*

For example, Figure 1 illustrates a PEV in OceanBase, a widely deployed SQL DBMS. OceanBase provides the `SHOW PROCESSLIST` command to display commands executed by logged-in users. Under the default privilege configuration, its output should be limited to the user's own sessions. However, in this case, the incorrect privilege enforcement allows a regular user to observe commands

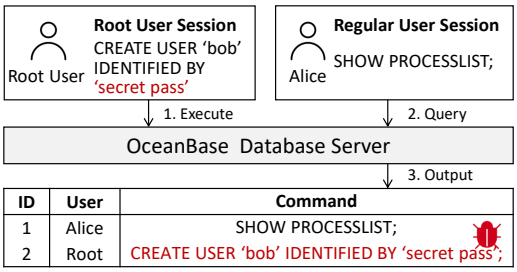

Figure 1: A PEV in OceanBase detected by Accio. It enables regular users to access command content from the root session (e.g., other users' passwords in plain text), which exceeds their intended privileges.

from all sessions, including those of the root user. Consequently, when the root user executes `CREATE USER 'bob' IDENTIFIED BY 'secret pass'`, a regular user with low privileges (e.g., Alice) can directly observe the command and obtain the plaintext password "secret pass". With valid credentials, the attacker can log in as the compromised user, inherit its privileges, and further manipulate database objects or user grants, leading to complete compromise of the database system. The root cause of this vulnerability is an implementation error in OceanBase's access control logic, which fails to correctly enforce the privilege checks of `SHOW PROCESSLIST` command for session visibility at runtime.

Despite the severe consequences of PEVs in DBMSs, the understanding of their characteristics remains limited, and effective techniques for systematically detecting them are still lacking. DBMS developers primarily rely on large-scale regression test suites [29] to validate command behavior under different privilege configurations, yet these tests are costly to develop and maintain, and are inherently limited in covering boundary cases and subtle privilege interactions, leaving many security-critical behaviors undetected. Researchers have also explored automated approaches such as static analysis to examine privilege enforcement [24]. However, access control in DBMSs is highly dynamic and context-dependent, as privilege checks may vary with execution phases, runtime database states, user sessions, and internal components. Such characteristics make static analysis imprecise in this domain, often leading to a large number of false positives.

To better understand PEVs and facilitate their detection, we present a motivation study of 63 real-world cases from 5 popular DBMSs, namely MySQL, MariaDB, TiDB, Firebird, and ClickHouse. We carefully examined each bug report along with the corresponding source code to analyze two main aspects: the symptoms of PEVs, such as the types of unauthorized actions observed, and the trigger conditions, including which SQL operations and database objects cause PEVs. We found that PEVs are triggered by various SQL commands: about 41.3% by DQL commands, 25.4% by administration commands, 19.0% by DDL commands, and 14.3% by DML commands. Regarding root causes, about 85.7% of the vulnerabilities are caused by missing privilege checks, while the remaining cases are due to incorrect privilege resolution or operation-privilege mismatch. Our findings indicate that detecting such vulnerabilities requires consideration of the interactions between SQL operations, user privileges, and execution contexts.

Even with these findings, detecting PEVs in DBMSs remains challenging. First, DBMSs manage multiple types of privileges organized hierarchically across various objects, making it difficult to precisely model and reason about the privileges required for each operation. Second, determining which privileges are actually required to execute a given SQL command is non-trivial, especially when commands interact with multiple objects or depend on dynamic, context-sensitive conditions. Finally, the documentation of required privileges for some commands is ambiguous or incomplete, making it difficult to determine whether observed behavior constitutes a true access control violation.

To address these challenges, we designed a tool named Accio to detect PEVs in SQL DBMSs in three stages. First, it constructs a privilege model that captures privilege types and database objects. Second, Accio generates SQL commands and calculates the minimum privilege set required to execute each command using an iterative, feedback-driven strategy. Finally, it performs consistency analysis by comparing the calculated privilege set against expected policies, identifying privilege escalation such as missing privilege checks or incorrect privilege scopes.

We implemented and evaluated Accio on seven widely used DBMSs, namely MySQL [29], MariaDB [5], OceanBase [28], TiDB [37], ClickHouse [15], Firebird [10], and Oracle [35]. The tool discovered 26 previously unknown privilege escalation vulnerabilities, all of which have been confirmed, and 23 have been fixed. We also tested Accio against historical versions with known privilege escalation cases, successfully detecting 91.7% of them. In summary, the paper makes the following contributions:

- We find that privilege escalation vulnerabilities in DBMSs pose substantial risks, yet their characteristics remain largely unexplored, and systematic methods for detecting such vulnerabilities are still lacking.
- We analyze the root causes and trigger conditions of 63 real-world PEVs and design Accio to detect them via minimum privilege set calculation, systematically reasoning about the privileges required for SQL commands.
- Using Accio, we discovered 26 previously unknown PEVs across 7 widely deployed SQL DBMSs, all of which have been confirmed, and 23 have been fixed.

2 Background

2.1 Basic Concepts

Database Management Systems (DBMSs) [47] are software systems that enable users to define, create, maintain, and control access to databases. They serve as the backbone of modern information infrastructure, managing vast amounts of data for various applications. Structured Query Language (SQL) [52] is the standard language for interacting with DBMSs, used for querying, updating, and managing data.

SQL commands can be classified into four main types based on their operations and functionality:

- Data Definition Language (DDL) commands (e.g., `ALTER`, `CREATE`) are used to define or modify the database structure or schema [49].
- Data Manipulation Language (DML) commands (e.g., `UPDATE`, `DELETE`) are used to manage data within schema objects [50].

- Data Query Language (DQL) commands (e.g., `SELECT`) are used to retrieve data from the database [51].
- Data Control Language (DCL) commands (e.g., `GRANT`, `REVOKE`) are used to control access to data [48].

Most DBMSs support multiple users with different privileges, including *administrators* and *regular users*. Administrators are able to execute most types of SQL commands and configure the privileges of regular users through the corresponding DCL commands. In contrast, regular users can execute only a limited set of DQL, DDL, and DML commands according to the privileges assigned to them.

2.2 Access Control in DBMSs

Access control in DBMSs determines whether each SQL command is allowed to execute under the privileges of the current session. In this section, we describe how DBMS access control works in practice, including privilege types, privilege configurations for users and roles, and privilege checks during SQL command execution.

Privilege types. The privilege types specify different operations that a user is allowed to perform on target database objects, such as `CREATE`, `SELECT`, and `INSERT` [33]. Some systems further distinguish privilege scopes, such as the entire database or schema [33]. In addition, many systems provide system-scope administrative privileges for tasks such as the `CREATE USER` privilege, which apply at the global scope rather than to a single table.

Privilege configurations. Privileges in DBMSs are configured using commands such as `GRANT` and `REVOKE` [32]. These commands modify the DBMS's internal privilege configuration. In DBMSs, privileges can be attached not only directly to users, but also via roles [34]: A role in DBMSs is a group of privileges that can be granted to users or other roles, and users can choose which roles to activate in a session. In this case, DBMSs check both the privileges owned by the user and the privileges from all active roles when executing SQL commands.

Privilege checks. When the DBMS receives a SQL command, it applies privilege checks by validating the command against the current user's privilege configuration. The real-world privilege checks are often complex, since the checks are determined by multiple factors, including the SQL semantics, the current database state, and the assigned privileges of the current user. Moreover, depending on the implementation of the DBMS's access control, these privilege checks may be performed at different stages, such as during parsing, optimization, or execution of the command. These factors increase the complexity of privilege checks in DBMSs.

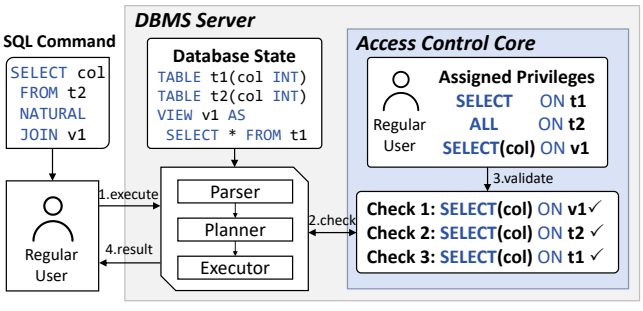

Figure 2: An example of the privilege check in DBMS.

Figure 2 illustrates how a `SELECT` command induces a series of privilege checks during execution. The regular user tries to execute the command `SELECT col FROM t2 NATURAL JOIN v1`, where `v1` is a view defined as `SELECT * FROM t1`. When the DBMS receives this command, it processes it through the parser, planner, and executor components, based on the current database state. During this process, the DBMS may apply privilege checks at any of these stages, depending on its detailed implementation. It needs to check three privileges: the `SELECT` privilege for column `col` on table `t2`, view `v1`, and its base table `t1`. Then, it validates whether the current user has the required privileges to execute the command according to its assigned privileges. The user owns the table-scope `SELECT` privilege on `t1`, `ALL PRIVILEGES` on `t2`, and the `SELECT` privilege for column `col` on `v1`. Its privileges cover all required checks for the command, and the DBMS allows the command to execute.

The example shows that even a simple SQL command may involve complex privilege checks due to the complexity of DBMS access control. Such complexity makes it challenging to ensure the correctness of privilege checks in DBMSs. An implementation flaw at any step in this process could result in an incorrect privilege check, leading to privilege escalation. In this paper, we specifically focus on *privilege escalation vulnerabilities* (PEVs), where the DBMS incorrectly allows a user to execute a command without satisfying the necessary privilege requirements.

3 Motivation Study

To better understand privilege escalation behavior in DBMSs, we conducted an in-depth motivation study across five widely used DBMSs: MySQL, MariaDB, TiDB, Firebird, and ClickHouse. We collected PEV reports from their issue trackers and commit histories.

Threat Model. In this paper, we focus on detecting PEVs caused by flaws in the DBMS access control implementations. The attacker's goal is to exploit such PEVs to perform operations beyond their assigned privileges, resulting in: (1) Accessing or tampering with database objects (e.g., tables, views) without the necessary privileges (e.g., `SELECT`, `INSERT`). (2) Executing restricted administrative commands (e.g., `CREATE USER`) that should be denied.

Our attack model assumes that the attacker operates through a legitimate low-privileged DBMS user or a valid low-privileged session. The attacker can submit SQL commands through the interface exposed to that account or session. This mainly covers two scenarios: (1) The attacker may already own a legitimate but low-privileged DBMS account. Such accounts commonly exist in enterprise data warehouses and multi-tenant database services [8]. (2) The attacker may obtain the ability to interact with the DBMS through a valid low-privileged session by other means, such as application-level vulnerabilities (e.g., SQL injection) or weakly protected database access (e.g., guest accounts) [7].

We assume that user privileges are correctly configured by administrators and exclude attacks on authentication mechanisms and the host operating system. Our focus is strictly on the correctness of the internal access control implementation within the DBMS.

Study Methodology. Table 1 summarizes the PEVs collected in our study. Since manually inspecting all reports to identify PEVs is time-consuming, we applied filtering rules to identify relevant reports. We used keywords like "GRANT", "REVOKE", "privilege",

Table 1: The number of collected PEVs in popular DBMSs.

DBMSs	MySQL	MariaDB	TiDB	Firebird	ClickHouse
Studied Bugs	10	7	20	22	4
Age (years)	30	16	9	25	9
GitHub Stars	11.9k	6.7k	39.4k	1.4k	44.4k

"permission", and "access control" to retrieve potentially relevant reports, as DBMS developers typically do not label reports as related to PEVs. We then filtered to include only reports with a status of confirmed or fixed, excluding potential false positives. Then, we reviewed the descriptions and developer comments. If a report ultimately indicated that a command could execute beyond its intended privileges, we classified it as a PEV. Through this process, we identified 63 PEVs in total.

Threats to Validity. Similar to other studies, our research has several limitations that should be considered. (1) Regarding the selection of target DBMSs, we chose the bug trackers of MySQL [30], MariaDB [6], TiDB [38], Firebird [10], and ClickHouse [15] as our collection sources. This is because their bug trackers clearly indicate whether a bug's status is confirmed and fixed, and their bug descriptions are detailed, which facilitates analysis. Furthermore, their access control designs generally follow the SQL standard, so an investigation into their privilege escalation issues has general applicability. (2) Regarding bug accessibility, some reports involving severe privilege escalation issues may be restricted from public access by MySQL and MariaDB [29]. Consequently, our sample may not include some of the most critical privilege escalation issues. Nevertheless, since these issues are also typically caused by incorrect privilege checks, these publicly accessible bugs can still provide an effective understanding of privilege escalation.

3.1 Manifestations of PEVs

To understand the potential damage of PEVs, we examined the manifestations of the collected bugs by manually reviewing the bug descriptions and reproducing them. Regarding their manifestations, we have the following findings:

> Finding 1. *Among the 63 studied bugs, about 41.3% (26/63) manifest as unauthorized data modification, about 38.1% (24/63) as unauthorized information disclosure, and about 20.6% (13/63) as database state corruption.*

Unauthorized Data Modification. Unauthorized data modification is a prominent issue, potentially leading to integrity violations, occurring in 41.3% of the studied cases. This problem typically arises when a DML command (i.e., `INSERT`, `UPDATE`, and `DELETE`) or DDL command (e.g., `ALTER`) bypasses the required privilege checks. If the write privilege is not properly maintained, the regular user may modify other users' tables to which it do not have access, causing data tampering or data loss.

Unauthorized Information Disclosure. Our investigation indicates that approximately 38.1% of the bugs result in unauthorized information disclosure, undermining the confidentiality of returned data. These PEVs allow attackers to access database objects without the necessary privileges. Such manifestations arise when a DQL command (e.g., `SELECT`) or a restricted administration command (e.g., `SHOW GRANTS`) fails to enforce read access control correctly. As a result, users can view sensitive user information or system status that they are not authorized to see, leading to data leakage.

Database State Corruption. Database state corruption is another severe manifestation, accounting for 20.6% of the studied cases. These cases do not directly cause data leakage or overwrite, but they can still lead to data loss, denial-of-service attacks, or functional failures. This includes issues such as unauthorized DDL operations (e.g., `DROP TABLE`), administration commands (e.g., `KILL`), or incorrect privilege enforcement that blocks legitimate administrative actions. This typically occurs when the DBMS engine fails to check administrative privileges or incorrectly resolves them, affecting the entire DBMS instance or preventing authorized operations.

3.2 Root Causes of PEVs

We manually checked each bug report and further analyzed the root causes of these bugs. We categorize the root causes into three categories, and we have the following findings:

> Finding 2. *Among the 63 studied bugs, the most common root cause is missing privilege checks (about 85.7% (54/63)), while the remaining cases are due to incorrect privilege resolution or operation-privilege mismatch.*

Missing Privilege Checks. The absence of required privilege checks is a prevalent root cause, responsible for 54 cases. This often occurs in implicit internal operations where the security context is lost, or when developers overlook the need for checks in complex execution paths. For example, developers may correctly implement checks for standard SQL commands but fail to enforce them when the same functionality is triggered via internal functions. This indicates that most privilege issues are context-specific.

Incorrect Privilege Resolution. This category involves logic errors in calculating the user's effective privileges (7 cases). The system correctly identifies the required privilege type and object for the target command, but fails to correctly resolve whether the user possesses that privilege. This typically occurs when the system mishandles complex authorization rules, such as scope hierarchies, role inheritance, or context matching.

Operation-Privilege Mismatch. This category represents a semantic error where the system performs a check, but enforces a privilege type that does not align with the semantic nature of the operation (2 cases). It involves vulnerabilities where the privilege type is mismatched or the object binding is incorrect.

3.3 Trigger Conditions of PEVs

Following the identification of root causes, we analyze the specific conditions that trigger these bugs.

> Finding 3. *Among the 63 studied bugs, about 41.3% (26/63) are triggered by DQL commands, about 25.4% (16/63) by administration commands, about 19.0% (12/63) by DDL commands, and about 14.3% (9/63) by DML commands.*

DQL Commands. Our investigation indicates that approximately 41.3% of the bugs are triggered by DQL commands. These bugs often occur when the system incorrectly evaluates privileges for reading data or metadata. For `SELECT` commands, the bugs frequently involve complex query structures, such as views, subqueries, or joins, where the permission checks fail to propagate correctly to the underlying tables or columns.

Administration Commands. Administration commands (e.g., `SHOW`, `SET`, `KILL`) trigger about 25.4% of the bugs. These often involve system-level operations where privilege checks are missed or incorrectly enforced for specific parameters or contexts.

DDL Commands. DDL commands account for a significant portion of the triggers (about 19.0%). `CREATE` commands are particularly problematic, often causing privilege escalation when modifying table structures or renaming objects. Similarly, `DROP` and `ALTER` commands can lead to inconsistent privilege states if the system fails to clean up permissions correctly.

DML Commands. DML commands trigger about 14.3% of the bugs. Although less frequent than query issues, these bugs are critical because they directly affect data integrity. Bugs in this category often involve fine-grained access controls, where the system fails to verify permissions for specific columns being modified.

We further examine the target objects of the trigger commands in the collected PEVs.

> Finding 4. *For the affected object types among the studied bugs, about 23.8% (15/63) affect the access to user tables, about 19.0% (12/63) affect the system tables, and about 11.1% (7/63) affect user views.*

Tables. Our investigation indicates that tables are the most frequently targeted objects in privilege escalation (about 23.8%). This is expected, as tables are the fundamental storage units in DBMSs and are involved in most data operations.

System Tables. System tables account for a significant portion of PEVs (about 19.0%). These bugs are critical because they can lead to unauthorized access to metadata or system configurations.

Other Objects. A variety of other objects are affected (about 36.5%), including sequences, procedures, functions, users/roles, and global system variables, showing the diverse attack surface of privilege escalation vulnerabilities.

3.4 Status of Existing Approaches

DBMS developers have made efforts to test access control in DBMSs. They usually design various unit tests and regression tests to avoid incorrect privilege checks. Current access control testing in DBMSs still primarily relies on these unit test cases. These manually constructed test cases can cover specific functionalities, but they cannot systematically or comprehensively test the complex logic of the access control system, leaving many PEVs undetected.

Existing DBMS fuzzing and testing approaches focus on finding crashes, logic errors, and performance bugs during SQL execution. However, these approaches do not specifically target access control testing. Although they can generate various SQL commands, they cannot distinguish whether a command execution is correctly checked against the expected privileges. For example, DBMS fuzzing usually relies on AddressSanitizer [41] to find memory bugs and customized test oracles based on query results to detect logic bugs. Such test oracles cannot validate whether the current user has the necessary privileges to execute the command. The lack of test oracles or ground truth models for access control makes them ineffective in identifying PEVs.

Other access control testing approaches often rely on static analysis to locate privileged operations and then check whether the corresponding permission checks appear on all reachable execution paths in the source code. However, their assumptions are hard to validate in DBMS access control. In DBMSs, privileges are defined over rich SQL-specific types and multiple object scopes, and the required checks depend on the SQL semantics, runtime database state, and execution context. Moreover, these privilege checks can be enforced across multiple stages of SQL processing. A SQL command is usually processed through complex components such as the parser, optimizer, and executor. This makes it difficult for static analysis to precisely track the execution path and determine whether all required privilege checks are performed. Such characteristics limit the effectiveness of these approaches for detecting PEVs in DBMSs.

Basic Idea of Accio. Finding 2 shows that most PEVs are caused by missing privilege checks. Based on this observation, the core insight of Accio is that PEVs in DBMSs arise when a command succeeds without enforcing privilege checks on some referenced objects. For each command, Accio determines whether there exists a privilege configuration under which the command still executes successfully while certain objects lack the required privileges. Any object that remains accessible without a covering privilege reveals a potential PEV. Findings 3 and 4 further indicate which commands and objects should be prioritized during PEV detection. Specifically, Finding 3 suggests that Accio should mainly generate DDL, DQL, DML, and administration commands. Finding 4 suggests that Accio should focus on the database objects that are most frequently involved in PEVs, including tables, system tables, views, sequences, and procedures. These findings together guide the design of Accio, which is presented in the following section.

4 Accio Design

Design Goal. A practical PEV detection approach should have the following properties:

- **General:** Accio targets DBMSs that follow the SQL standard (e.g., MySQL, TiDB, and Oracle). It assumes that the target DBMS exposes privilege information through system tables and configures privileges via `GRANT` commands. Accio can be adapted to other DBMSs with minor adjustments.
- **Efficient:** Accio is able to frequently exercise the privilege checking logic and effectively detect PEVs in real-world DBMSs within a reasonable time.
- **Accurate:** Accio is designed to have satisfactory precision to avoid reporting false positives.

Workflow. Figure 3 shows the overall workflow of Accio, which consists of three key components: Privilege Model Construction, Minimum Privilege Set Calculation, and Missing Privilege Check Detection. (1) Privilege Model Construction: Accio first analyzes the target DBMS to obtain essential authorization information,

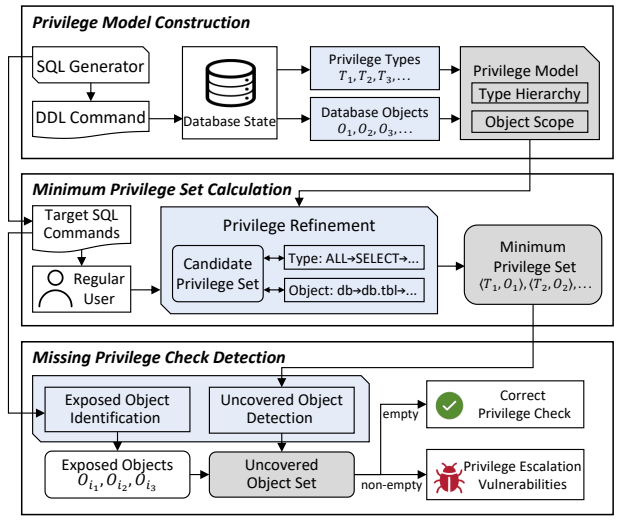

Figure 3: The overall workflow of Accio.

including privilege types and object scopes. Based on this, it constructs a comprehensive privilege model that represents the full set of privileges and their inclusion relationships. (2) Minimum Privilege Set Calculation: For a given SQL command, Accio calculates the minimum set of privileges required for execution. It employs a top-down approach, utilizing the constructed model to decompose high-level privileges and iteratively refine object scopes to identify precise permissions. (3) Missing Privilege Check Detection: Accio analyzes the consistency between the calculated minimum privilege set and the observed behavior. By checking for missing coverage and uncovered exposed objects, it identifies potential PEVs.

4.1 Privilege Model Construction

In this section, we detail the construction of the Dual-Hierarchy Privilege Model, which serves as the foundation for precise privilege analysis. Formally, we define a privilege as a tuple $P = \langle T, O\rangle$, where T represents the Privilege Type and O denotes the Object Scope. To effectively capture the complex authorization logic of DBMSs, Accio establishes two orthogonal hierarchical structures: a static Privilege Type Hierarchy and a dynamic Object Scope Hierarchy. This dual-hierarchy design enables Accio to model the inclusion relationships between privileges comprehensively, i.e., whether a privilege P_i implies another privilege P_j.

Static Privilege Type Hierarchy Construction. Accio first parses the DBMS system tables (e.g., `system.privileges`) to obtain essential information about privilege definitions and their static dependencies. For DBMSs that do not expose such information through system tables, the required privilege information can be obtained from the official documentation. It then extracts the inclusion relationships where a high-level privilege (e.g., `ALL`) encompasses multiple sub-privileges (e.g., `SELECT`, `ALTER`). Based on these relationships, Accio constructs a static directed tree to represent the Type Hierarchy. In this hierarchy, the root node represents the full privilege, while leaf nodes correspond to atomic operations. For example, in ClickHouse, the `ALTER` privilege implicitly includes `ALTER TABLE`, `ALTER VIEW`, and `ALTER INDEX`. Our model represents this by creating directed edges from the `ALTER` node to its child nodes. This static structure is built once during the initialization phase and remains constant throughout the analysis.

Dynamic Object Scope Hierarchy Construction. Simultaneously, Accio constructs the Object Scope Hierarchy. To ensure the hierarchy covers a wide range of objects, Accio first prepares the database state by executing a sequence of object creation commands. According to Finding 4, Accio employs a SQL generator to produce creation commands for a variety of user objects, including `TABLE`, `VIEW`, `PROCEDURE`, and `SEQUENCE`. Then, Accio executes these creation commands to update the database state, and traverses the system tables to retrieve the current schema information. Unlike privilege types, the objects (e.g., databases, tables, columns) are dynamic and user-defined. It initializes a root node representing the global scope and then recursively identifies all existing databases, tables, and columns, adding them as child nodes of their respective parents. For instance, a table `t1` within database `db1` is modeled as a child of the `db1` node, which in turn is a child of the global node. Accio incorporates this structural information to enable granular privilege analysis at different object scopes.

Model Instantiation. To construct the final privilege graph, Accio instantiates privilege tuples by mapping privilege types to their compatible object scopes. This process ensures that only valid combinations are included, as not all privilege types are applicable to every object scope (e.g., `CREATE TABLE` applies to table scopes but not to column scopes). Accio employs a dynamic probing strategy to determine these valid combinations efficiently. Specifically, Accio enumerates all available privilege types. It generates test privilege configuration commands on representative objects for each object-scope granularity according to the grammar of the target DBMS (e.g., the `GRANT` grammar). It then executes these commands and observes the feedback. By observing the execution feedback, Accio identifies valid privilege–scope combinations and incorporates them into the privilege graph. Since the compatibility rules are constant for a given DBMS version, this model is constructed once and reused throughout the subsequent testing process.

Figure 4 illustrates how Accio constructs the Dual-Hierarchy Privilege Model from a DBMS's privilege semantics. Accio first

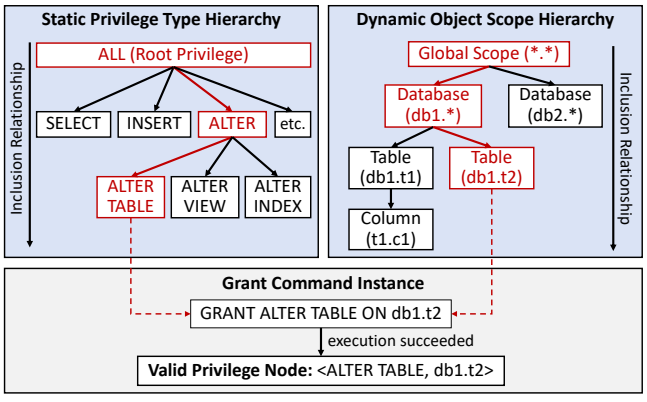

Figure 4: The Privilege Model Construction in Accio.

builds a *privilege type hierarchy* that captures static inclusion relationships among privilege types (e.g., `ALL` subsumes `SELECT`, `ALTER`, and their refined variants such as `ALTER TABLE`), enabling type-level implication during reasoning. Then, for each privilege type, Accio constructs an object scope hierarchy that represents the dynamic object scopes of privileges, organized from the global scope (e.g., `*.*`) down to database, table, and column scopes, where privileges naturally inherit along containment. Given these two hierarchies, Accio synthesizes a privilege node for each concrete grant by pairing its privilege type with its object scope; for example, `GRANT SELECT ON db1.t1` is mapped to the node $\langle \texttt{SELECT}, db_1.t_1 \rangle$. This unified representation allows Accio to reason consistently about privilege subsumption via the type hierarchy and scope coverage via the object scope hierarchy when analyzing whether a SQL command is properly covered by current privileges.

4.2 Minimum Privilege Set Calculation

Based on the privilege model, the second component is Minimum Privilege Set Calculation. For a specific SQL command, identifying the precise set of privileges required for its execution is challenging due to the complexity of SQL syntax and authorization logic. Accio addresses this by calculating the minimum set of privileges through a divide-and-conquer process.

First, Accio generates target SQL commands for privilege analysis. According to Findings 3 and 4, Accio focuses on the SQL commands and database objects most commonly involved in PEVs. Specifically, it generates DDL, DML, DQL, and administration commands, and focuses on objects including tables, views, sequences, and procedures. During generation, Accio first chooses a command type and constructs its base AST. It then adds different clauses according to the command types. Finally, Accio binds them to existing database objects to ensure that the generated commands are valid under the current schema.

Second, Accio validates the command with the superuser privilege. It checks if the command can be executed successfully with the privilege `ALL ON *.*`. If the execution fails even with the highest privilege, the command is likely invalid or contains syntax errors, and Accio discards it. If successful, Accio proceeds to decompose the `ALL` privilege into its constituent sub-privileges according to the constructed model.

Then, Accio determines which specific privilege types are necessary to execute the command. It iterates through the sub-privileges and tests whether the command can still be executed with each subset. When the number of privilege types is large, checking them one by one is inefficient. To accelerate this process, Accio employs a divide-and-conquer strategy. It divides the privilege set into two halves and tests if the required privilege lies within one of them. By recursively narrowing down the candidate privilege set, Accio efficiently isolates the necessary privilege types. This results in a minimum set of required privilege types for the SQL command.

Next, Accio refines the object scopes to find the most precise scope. It starts by splitting the global scope (`*.*`) into database-scoped entries (e.g., `db1.*`, `db2.*`). For each database, it calculates the necessary privilege set. Subsequently, it further drills down to table-scoped entries (e.g., `db1.tbl1`, `db2.tbl2`). Through this iterative splitting and checking process, Accio obtains the minimum set of required privileges in terms of both privilege type and object scope. For example, if a command accesses a table `t1`, Accio will determine that the `SELECT` privilege on `t1` is required, rather than `SELECT` on the entire database.

Algorithm 1: Minimum Privilege Set Calculation

```
Input  : SQL command: cmd, Privilege model: model
Output : Minimum privilege set: min_set
1  min_set ← ∅;
2  foreach p ∈ model.all_types do
3  |   min_set ← min_set ∪ {⟨p, GLOBAL⟩};
4  end
5  min_set ← refine_priv(min_set, ∅);
6  foreach ⟨p, obj⟩ ∈ min_set do
7  |   cur ← min_set − ⟨p, obj⟩;
8  |   child_privs ← get_children(model, ⟨p, obj⟩);
9  |   refine ← refine_priv(child_privs, cur);
10 |   min_set ← cur ∪ refine;
11 end
12 return min_set;
13 Function refine_priv(privs, ctx):
14 |   if |privs| == 1 then
15 |   |   return execute (ctx) ? ∅ : privs;
16 |   end
17 |   L, R ← split(privs);
18 |   res_L ← execute(R ∪ ctx) ? ∅ : refine_priv(L, R ∪ ctx);
19 |   res_R ← execute(L ∪ ctx) ? ∅ : refine_priv(R, L ∪ ctx);
20 |   return res_L ∪ res_R;
```

Algorithm 1 illustrates how Accio calculates the minimum privilege set for a given SQL command. Accio first initializes an over-privileged configuration by granting all supported privilege types at the global scope, ensuring the command is executable before refinement (Lines 1–4). It then invokes a feedback-driven minimization procedure that iteratively removes privilege entries and re-executes the command to retain only those that are necessary (Line 5). After minimizing required privilege *types*, Accio further refines privilege *object scopes* by replacing coarse-grained entries with their finer-grained children in the object hierarchy (e.g., from database to table and column), again guided by execution outcomes (Lines 6–11). To reduce trials, the refinement procedure applies a divide-and-conquer strategy that splits candidates and recursively minimizes only the subset whose removal causes execution failure (Lines 13–20). Finally, Accio returns a compact privilege set that is sufficient for executing the command and is minimum for the target DBMS (Line 12).

4.3 Missing Privilege Check Detection

After Accio computes the minimum privilege set required by a SQL command, it further checks whether the DBMS enforces this set consistently during execution. In particular, according to Finding 2, we target *missing privilege checks*, where the system executes a command successfully but fails to validate the privileges needed to protect the objects that the command may access or expose. To this

end, Accio combines the command semantics, the observed execution outcome, and the computed minimum set to decide whether the behavior constitutes a privilege escalation.

Accio performs the detection in two steps. First, it identifies *exposed objects* by expanding the SQL command and collecting all object references that are potentially revealed through execution. This step is necessary because the surface form of a SQL command often omits the concrete objects actually accessed at runtime. Accio therefore normalizes the command into an expanded form and then extracts object names from the expanded SQL command, using the current database state as the reference to resolve which tokens correspond to real objects. Accio also collects exposed objects from the returned result sets of the SQL commands. Specifically, the result set typically consists of two parts: the column headers and the row values. Accio records an object as exposed if its name appears in either the headers or the returned data.

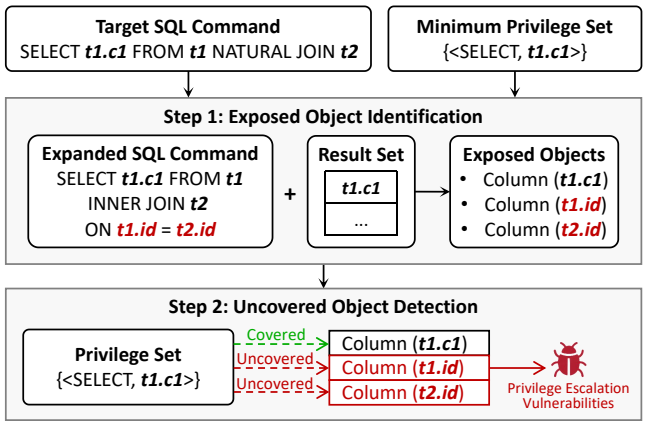

Figure 5: The Missing Privilege Check Detection in Accio.

The expansion step is necessary for handling implicit object usage. For example, `SELECT * FROM tbl` is incomplete from an access-control perspective, as `*` represents all columns of `tbl`; the effective access is closer to `SELECT col1, col2, col3 FROM tbl`, and the exposed objects should include these concrete columns. Similarly, `SELECT col FROM tbl1 NATURAL JOIN tbl2` implicitly depends on the join keys: the DBMS matches columns with the same name across the two tables, meaning that the query references those columns even if they are not explicitly listed. By unfolding such implicit semantics, Accio obtains a precise exposed object set that reflects what the command actually reveals under the current schema.

Second, Accio cross-checks the exposed-object set against the computed minimum privilege set. Specifically, Accio verifies that every exposed object is covered by at least one matching privilege entry in the minimum set. Coverage is defined with respect to object scopes and the object hierarchy: a column can be covered by a column-scope privilege on that column, a table-scope privilege on its parent table, a database-scope privilege on the containing database/schema, or a global-scope privilege, depending on the DBMS's privilege model. Intuitively, an object is considered protected if some privilege in the minimum set grants the required operation at the same or a broader scope that subsumes the object.

Figure 5 shows an example of how Accio detects a missing privilege check after obtaining the minimum privilege set. Given the target query `SELECT t1.c1 FROM t1 NATURAL JOIN t2` and its computed minimum set `SELECT` on `t1.c1`, Accio first performs exposed object identification by expanding the query into an equivalent form with explicit join predicates, e.g., `... INNER JOIN t2 ON t1.id = t2.id`, and then extracting all object names involved in the expanded command. It also collects the column names as exposed columns from the result set. In this case, besides the selected column `t1.c1`, the natural join implicitly references the join keys `t1.id` and `t2.id`. Accio then conducts uncovered object detection by checking whether each exposed object is covered by at least one entry in the minimum privilege set. Since only `t1.c1` is covered while `t1.id` and `t2.id` remain uncovered, Accio reports such execution as a PEV candidate.

This detection becomes effective precisely because we have already minimized the required privileges. The minimum set removes privilege types and scopes that are unrelated to executing the command in the target DBMS, making the remaining entries a tight approximation of what should be checked. Under this condition, if Accio still observes that a command succeeds and exposes objects that are not covered by any privilege entry in the minimum set, the most plausible explanation is that the DBMS omitted (or incorrectly bypassed) the corresponding privilege check along the execution path. Accio then reports the case as a candidate PEV, together with the uncovered exposed objects and the execution evidence, to facilitate developer confirmation and debugging.

5 Implementation

We implemented Accio in approximately 18,500 lines of Python on top of the SQLGlot framework [25].

SQL Generation. To create database states and target SQL commands, we use SQLGlot to generate valid SQL commands for different DBMSs. Accio's SQL generator constructs ASTs for different command types and converts them into SQL commands in the dialect of the target DBMS. In the current implementation, the SQL generator covers 21 SQL command types and 27 clause types, and each command contains up to 10 clauses. It currently supports objects including `TABLE`, `VIEW`, `PROCEDURE`, and `SEQUENCE`.

Privilege Configuration Generation. For the Privilege Model Construction component, Accio synthesizes `GRANT` commands to instantiate privilege nodes and check their validity against the target DBMS. We summarized the documented forms of `GRANT` commands from the official documentation of each DBMS and grouped them by object scope, such as global, database, table, and column scopes. We then implemented these forms as context-free grammars using Python's NLTK library [40]. During model construction, Accio instantiates these forms with representative objects in the current database state and executes the resulting `GRANT` commands to check whether a privilege type is valid on a specific object scope. Based on the execution feedback, Accio retains valid privilege–scope combinations.

Parallel Execution. For the Minimum Privilege Set Calculation component, we implemented a multi-threaded framework to improve performance. We implemented a connection pool to manage

Table 2: The PEVs detected by Accio in seven DBMSs within 24 hours, including 1 in MySQL, 3 in MariaDB, 7 in OceanBase, 10 in TiDB, 2 in ClickHouse, 2 in Firebird, and 1 in Oracle.

ID	DBMS	Command	Description	Status
1	MySQL	`CREATE`	Privilege checks for `CREATE TABLE ... LIKE` are missing	Confirmed
2	MariaDB	`CHECK TABLE`	Privilege checks for `CHECK TABLE` are inconsistent	Fixed
3	MariaDB	`INSERT`	`INSERT ... RETURNING` exposes columns for which the user lacks `SELECT` privilege	Fixed
4	MariaDB	`SHOW TABLES`	`SHOW TABLES` allows users with only `GRANT OPTION` privilege to read all table names	Fixed
5	OceanBase	`CHECK TABLE`	Privilege checks for `CHECK TABLE` are missing	Fixed
6	OceanBase	`SHOW PROCESSLIST`	Results of `SHOW PROCESSLIST` leak unauthorized information	Fixed
7	OceanBase	`CALL`	Privilege checks for procedures in the `DBMS_UDR` packages are missing	Fixed
8	OceanBase	`CALL`	Privilege checks for procedures in the `DBMS_WORKLOAD_REPOSITORY` packages are missing	Fixed
9	OceanBase	`CALL`	Privilege checks for procedures in the `DBMS_XPLAN` packages are missing	Fixed
10	OceanBase	`INSERT`	`INSERT` command can expose unauthorized column names	Fixed
11	OceanBase	`SELECT`	Results of `information_schema.parameters` leak unauthorized information	Fixed
12	TiDB	`DROP STATS`	Privilege checks for `DROP STATS` are missing	Fixed
13	TiDB	`SPLIT TABLE`	Privilege checks for `SPLIT TABLE` are missing	Fixed
14	TiDB	`ALTER RANGE`	Privilege checks for `ALTER RANGE` are missing	Fixed
15	TiDB	`SHOW TABLE`	Results of `SHOW TABLE ... REGIONS` leak unauthorized information	Fixed
16	TiDB	`ADMIN`	Privilege checks for `ADMIN PLUGINS` are missing	Fixed
17	TiDB	`INSERT`	Privilege checks for columns in `INSERT` values are missing	Fixed
18	TiDB	`ALTER TABLE`	Privilege checks for `TRUNCATE PARTITION` are insufficient	Fixed
19	TiDB	`SELECT`	Results of `information_schema.statements_summary` leak unauthorized command content	Fixed
20	TiDB	`SELECT`	Results of `information_schema.statements_summary_history` leak unauthorized command content	Fixed
21	TiDB	`SELECT`	Results of `information_schema.user_attributes` leak unauthorized attributes	Fixed
22	ClickHouse	`SYSTEM`	Privilege checks for `SYSTEM SYNC FILE CACHE` are missing	Fixed
23	ClickHouse	`EXPLAIN`	Privilege checks for `EXPLAIN` are broken	Confirmed
24	Firebird	`COMMENT`	Privilege checks for `COMMENT ON PARAMETER` are missing	Fixed
25	Firebird	`DROP FILTER`	Privilege checks for `DROP FILTER` are bypassed	Fixed
26	Oracle	`SELECT`	Results of table `SYS.ORA_KGLR7_DEPENDENCIES` leak unauthorized information	Confirmed

the database connections for multiple users and sessions concurrently. Since each validation task executes the same target command under a different privilege subset, these tasks can be processed independently. We therefore assign each validation task to a separate worker thread and use a thread pool to maximize parallelism. This implementation can reduce the time cost of calculating the minimum privilege set for complex SQL commands.

6 Evaluation

We evaluated Accio in terms of its ability and efficiency to detect PEVs on real-world DBMSs. Our evaluation aims to answer the following research questions:

- **RQ1:** Can Accio successfully discover new and critical PEVs in widely-used DBMSs?
- **RQ2:** Can Accio perform better on DBMSs than other PEV detection approaches?
- **RQ3:** What is the effectiveness of components of Accio?
- **RQ4:** Can Accio reproduce historical PEVs?

6.1 Experimental Setup

Test DBMSs. To demonstrate the generality and PEV-finding ability of Accio across various database architectures, we applied Accio to seven widely used DBMSs: MySQL [29], MariaDB [5], TiDB [37], OceanBase [28], ClickHouse [15], Firebird [10], and Oracle [35]. These DBMSs represent different categories of database systems, including traditional relational databases (MySQL, MariaDB, Firebird, Oracle), distributed HTAP databases (TiDB, OceanBase), and OLAP databases (ClickHouse). They are extensively used in industrial production environments and have been rigorously tested by their vendors and the open-source community. Specifically, the chosen versions for evaluation were MySQL v9.2.0, MariaDB v12.1.1, TiDB v8.5.2, OceanBase v4.3.5, ClickHouse v25.3.10, Firebird v5.0.3, and Oracle Database Free 26ai. These versions were the latest stable releases at the time of our experiments.

Environment. The evaluation was performed on a machine running 64-bit Ubuntu 22.04 LTS with 128 cores (AMD EPYC 7763 Processor) and 504 GiB of main memory. To ensure a clean and consistent experimental environment, each DBMS was deployed in an isolated Docker container. We allocated 10 CPU cores and 50 GiB of RAM for each DBMS instance. We ran Accio on each DBMS for 24 hours with 5 times repeated under identical conditions, and the median results are used in this paper.

6.2 PEV Detection

Overall Results. We evaluated Accio in terms of its ability to find new PEVs in real-world DBMSs. Despite these seven tested DBMSs being mature systems that have undergone rigorous testing by their vendors, Accio demonstrated effective PEV-finding capabilities. Accio successfully detected 26 previously unknown PEVs. Table 2 summarizes the statistics of these PEVs across the seven DBMSs. Accio found 1 PEV in MySQL, 3 in MariaDB, 7 in OceanBase, 10 in TiDB, 2 in ClickHouse, 2 in Firebird, and 1 in Oracle. These detected PEVs enable unprivileged users to perform restricted operations. Specifically, 8 of them cause unauthorized data modifications, 9 lead to sensitive information leakage, and 9 allow database state corruption. We have reported all the detected PEVs to the corresponding DBMS vendors through their official

bug trackers and security contact email addresses. All of these PEVs have been confirmed by the developers, with 23 of them fixed.

Case Studies. To provide insights into how Accio detects subtle PEVs, we present detailed case studies of three vulnerabilities found in OceanBase, MariaDB, and TiDB. We selected these cases because they represent diverse categories of privilege escalation vulnerabilities (e.g., unauthorized state modification, metadata leakage, data deletion) and demonstrate the capability of Accio to discover PEVs in different DBMSs.

Listing 1: Unauthorized creation of query rewrite rules in OceanBase via the `CALL` command.

```
-- Privilege: SELECT ON testdb.*
CALL DBMS_UDR.CREATE_RULE('rule1', 'testdb',
     'SELECT␣?', 'SELECT␣?+1');
--Expected: Permission denied to create rule
--Actual: Query OK, rule 'r1' created successfully
```

Case Study 1: Unauthorized creation of query rewrite rules in OceanBase. Accio detected a missing privilege check in OceanBase in the `DBMS_UDR` package. In this scenario, when the user-defined rewrite rule feature is enabled, a user with only the `SELECT` privilege on the target database can execute procedures such as `DBMS_UDR.CREATE_RULE`. This allows an attacker to define rules that rewrite legitimate queries (e.g., `` `SELECT ?' ``) into arbitrary forms (e.g., `` `SELECT ? + 1' ``), tampering with the query logic of other users. Accio identified this vulnerability because the system successfully executed the command despite the user lacking the necessary administrative privileges to modify global rewrite rules.

Listing 2: Unauthorized disclosure of user table names and definitions in MariaDB.

```
--Privilege: USAGE ON *.* WITH GRANT OPTION
SHOW TABLES FROM other_db;
SHOW CREATE TABLE other_db.other_tbl;
--Expected: Access denied for database 'other_db'
--Actual: Returns all table names in 'other_db'
    and the definition of 'other_tbl'
```

Case Study 2: Unauthorized disclosure of table names and definitions in MariaDB. Accio discovered an information leakage vulnerability in MariaDB involving the `SHOW TABLES` command. This vulnerability allows a user possessing only the `GRANT OPTION` privilege to list all table names in any restricted database (`other_db` in the case study). Moreover, `SHOW CREATE TABLE` has a similar issue, which allows the user to view the definitions of tables in the `other_db` database. In this way, users with the `GRANT OPTION` privilege can first retrieve all table names in the database and then obtain their table definitions, leading to unauthorized disclosure. Normally, accessing the metadata of system tables requires specific `SELECT` or `SHOW TABLE` privileges. Accio detected this issue because the command returned a list of sensitive system tables that were not covered by the user's assigned privileges.

Case Study 3: Unauthorized deletion of table statistics in TiDB. Accio identified a missing privilege check in TiDB regarding the

Listing 3: Unauthorized deletion of table statistics in TiDB.

```
--Privilege: SELECT ON test.another_tbl
DROP STATS test.tbl;
--Expected: ERROR: DROP command denied
--Actual: Query OK, statistics for 'tbl' deleted
```

`DROP STATS` command. In this case, a user with the `SELECT` privilege on any table in the database can successfully execute `DROP STATS` to remove the statistical data of other tables. This operation causes unauthorized deletion of other users' statistics information, which can impact the query optimizer's performance and stability. Accio detected this vulnerability because the command succeeded on a target table for which the user had no `DROP` privileges on the target table, violating the expected access control policy.

6.3 Comparison with Existing Approaches

To evaluate the effectiveness of Accio compared to static analysis approaches, we conducted a comparative experiment by adapting MPChecker [24], a missing permission check vulnerability detector originally designed for Java distributed systems. MPChecker models privileged operations in the system and the permission checks that are expected to guard them. It reports a potential vulnerability when a privileged operation is reachable without a required check on the corresponding execution path. Since MPChecker's implementation is not directly compatible with DBMSs, we implemented its core detection logic using CodeQL, referred to as CodeQL$^{\text{MPChecker}}$. We applied CodeQL$^{\text{MPChecker}}$ to MySQL, MariaDB, TiDB, ClickHouse, and Firebird to detect missing permission check vulnerabilities. We skipped the evaluation of MPChecker on OceanBase due to the difficulty in compiling OceanBase with CodeQL support. The Oracle database is also excluded since it does not provide source code.

PEV detection. Among the five DBMSs, CodeQL$^{\text{MPChecker}}$ reported 57, 14, 11, 47, and 23 missing permission check issues MySQL, MariaDB, TiDB, ClickHouse, and Firebird, respectively. However, upon manual verification, we found that only 5 in TiDB and 1 in ClickHouse of these reports were true positives. By contrast, within 24 hours, Accio detected 1, 3, 10, 2, and 2 PEVs in the five DBMSs, respectively, including the 6 PEVs identified by CodeQL$^{\text{MPChecker}}$. CodeQL$^{\text{MPChecker}}$ misses many PEVs for two reasons: On the one hand, CodeQL$^{\text{MPChecker}}$ cannot identify cases where a privilege check exists but is still insufficient for DBMS privilege semantics. The tool treats a path as safe once it observes any privilege check, but DBMSs often require checks with multiple types and object scopes. For example, we observed a PEV case detected by Accio where a database-scope check exists, while the required table-scope check is missing. CodeQL$^{\text{MPChecker}}$ cannot detect it since this execution path already contains a permission check, although this check is insufficient. On the other hand, some DBMSs enforce access control implicitly through SQL query logic rather than explicit check functions in programming languages. For example, for some SQL commands in Firebird, the DBMS checks the users' privileges via specific SQL commands in the system table. PEVs in these commands are caused by incorrect privilege-check queries (e.g., missing

necessary `WHERE` clauses to filter out those objects where the user does not have privileges). CodeQL$^{\text{MPChecker}}$ is unable to identify such mistakes in SQL commands, leading to missing PEVs.

False Positives. CodeQL$^{\text{MPChecker}}$ reports 146 false positives, including 57, 14, 6, 46, and 23 in the five DBMSs, respectively. We attribute the high false positive rate of CodeQL$^{\text{MPChecker}}$ to the way privilege enforcement is implemented in DBMS access control. In DBMSs, privilege checks are often not placed at a single entry point. Instead, they are distributed across different execution stages, including parsing, optimization, and execution. As a result, some basic privilege checks may be applied during parsing the SQL command, while the other checks happen deeper in the optimization or execution stage. Due to the complex logic of the SQL parser and optimizer, it is difficult to correctly capture all of the privilege checks together. In addition, DBMS execution has complex control flow. It uses dynamic dispatch based on object types, operational modes, and configuration switches. Some commands also enforce privileges through loops that iteratively verify objects. These patterns are hard to recognize as guaranteed checks by static analysis, which further increases false positives of CodeQL$^{\text{MPChecker}}$.

In contrast, at 24 hours, Accio generated two false positives in total when detecting PEVs in the tested DBMSs, including one in ClickHouse and one in OceanBase. both of them are caused by silently failed SQL commands. In these cases, the command does not report an error when the privileges are insufficient, but it also skips the actual execution and does not affect the database state. For example, in ClickHouse, the `SYSTEM DROP REPLICA` command does not report an error when the user lacks the required privilege, but it performs no operation. Since Accio determines PEVs based on the observed execution status, such silently failed commands are treated as successful executions by Accio, leading to false positives. This behavior originates from the special design of specific DBMS commands. In contrast, most SQL commands in the tested DBMSs explicitly report errors when privilege checks fail, and Accio produces no false positives on these commands.

6.4 Effectiveness of Accio Components

To understand the contribution of the Privilege model Construction in Accio, we applied the Privilege Model Construction component to seven DBMSs. Note that model construction is used to identify privilege types and object scopes and to ensure model correctness by hierarchy checking. Table 3 shows the statistics of the privilege models constructed by Accio on seven DBMSs. Across the seven DBMSs, Accio covers 659 privilege types in total, accounting for 97.5% (659/676) of all documented privilege types supported by these DBMSs. This indicates that the Privilege Model Construction component is effective in constructing accurate privilege models for diverse DBMSs. For instance, Accio identified all 66 distinct privileges and four object scopes in MySQL. It also modeled 206 distinct privileges in ClickHouse and 217 in Oracle, which have complex access control systems. The remaining 17 privilege types currently remain unsupported because they require object types or preconditions that are not covered by the current generator. This result shows that the privilege model constructed by Accio covers most of the privilege types exposed by real-world DBMSs. For example, ClickHouse's `ALTER_COLLECTION` privilege requires user-defined collections, while the current database-state generator does not create such objects.

Table 3: Statistics of privilege types in vendor documentation and in the privilege models constructed by Accio on seven DBMSs. Note that the same privilege types can appear in multiple object scopes.

DBMS	# Documented Privs.	# Modeled Privs. by Each Object Scope			
		GLOBAL	DATABASE	TABLE	COLUMN
MySQL	66	66	20	14	4
MariaDB	42	42	22	15	4
OceanBase	38	38	16	13	4
TiDB	49	49	20	14	–
ClickHouse	218	206	105	98	19
Firebird	41	41	–	5	2
Oracle	222	217	–	11	3
Total	676	659	183	170	36

To understand the effectiveness of the Privilege Model Construction and Minimum Privilege Set Calculation in Accio, we implemented two variants: Accio$^{!Min}_{!Model}$ and Accio$^{!Min}$. Accio$^{!Min}_{!Model}$ disables both Privilege Model Construction and Minimum Privilege Set Calculation. Accio$^{!Min}$ disables Minimum Privilege Set Calculation. Due to the absence of Privilege Model Construction, Accio$^{!Min}_{!Model}$ can only construct `GRANT` commands that do not depend on database states (i.e., privileges with the global scope). In contrast, Accio$^{!Min}$ can construct `GRANT` commands with different scopes based on database states. However, since both variants lack Minimum Privilege Set Calculation, they enumerate all possible `GRANT` commands for each target command to attempt execution and detect potential PEVs under different privileges. We compare Accio against these two variants on the seven DBMSs for 24 hours, using the number of tested SQL commands and the number of detected PEVs as the metrics.

Table 4: The number of tested SQL commands by Accio and its variants in 24 hours.

DBMS	Accio$^{!Min}_{!Model}$	Accio$^{!Min}$	Accio
MySQL	3,851	844	12,312
MariaDB	6,574	890	13,043
OceanBase	1,628	1,171	8,605
TiDB	6,662	623	21,948
ClickHouse	1,608	188	11,976
Firebird	5,446	1,513	24,785
Oracle	1,680	1,209	38,760
Total	27,449	6,438	131,429

Table 4 shows the number of tested commands by Accio and its variants. From the table, we can see that Accio tested more commands than Accio$^{!Min}_{!Model}$ and Accio$^{!Min}$. Specifically, Accio$^{!Min}$ tested the fewest commands. This difference is mainly caused by the number of privilege sets that need to be tested for each SQL command. Without Minimum Privilege Set Calculation, Accio$^{!Min}$ needs to enumerate all possible privileges and test whether the command can be executed under each of them. For example, for a table

with one column, MySQL exposes 104 possible privileges across different object scopes, and Accio$^{!Min}$ needs to test all of them separately. Consequently, the number of tested SQL commands in 24 hours is much reduced compared to Accio. Accio$^{!Min}_{!Model}$ tests more commands than Accio$^{!Min}$ since it can only enumerate global-scope privilege nodes that are independent of database states, which reduces the number of privilege attempts required for each command. In contrast, Accio completes the detection of each target command with fewer attempts through Minimum Privilege Set Calculation. In our evaluation, the minimum privilege set for each command in MySQL can be computed within 20 privilege attempts by Accio's divide-and-conquer strategy, which highly improves the overall testing throughput.

Table 5: The number of PEVs detected by Accio and its variants in 24 hours.

DBMS	Accio$^{!Min}_{!Model}$	Accio$^{!Min}$	Accio
MySQL	0	0	1
MariaDB	0	1	3
OceanBase	0	1	7
TiDB	3	1	10
ClickHouse	1	0	2
Firebird	0	0	2
Oracle	0	0	1
Total	4	3	26

Table 5 shows the number of PEVs detected by Accio and its variants. From the table, we can see that Accio detected the most PEVs, followed by Accio$^{!Min}_{!Model}$, and Accio$^{!Min}$ detected the least. Although Accio$^{!Min}_{!Model}$ tested more commands than Accio$^{!Min}$, it cannot construct `GRANT` commands with specific scopes due to the lack of Privilege Model Construction. Consequently, it fails to trigger the PEVs that depend on specific database states. As for Accio$^{!Min}$, the number of detected PEVs is less than Accio due to the much slower execution speed. By contrast, Accio detects the most PEVs because it supports more privileges and achieves higher throughput during testing. We also measured how quickly Accio detects unique PEVs under the same time budget. Figure 6 shows that Accio detected 8 unique PEVs within 2 hours, while both Accio$^{!Min}_{!Model}$ and Accio$^{!Min}$ detected only one PEV. Within 24 hours, Accio detected 26 unique PEVs, compared with 4 detected by Accio$^{!Min}_{!Model}$ and 3 detected by Accio$^{!Min}$.

6.5 Rediscovery of Historical PEVs

To evaluate Accio's ability to rediscover known historical PEVs, we ran historical versions of the DBMSs that contain known PEVs and used Accio to attempt to reproduce them. Note that we excluded 21 PEVs because they are more than 10 years old, and the corresponding DBMS binaries cannot run on current OS environments. We further excluded 6 PEVs because they were introduced and fixed only in internal development builds and did not appear in any released version of DBMSs. Among the remaining 36 reproducible PEVs across the five studied DBMSs, Accio successfully reproduced

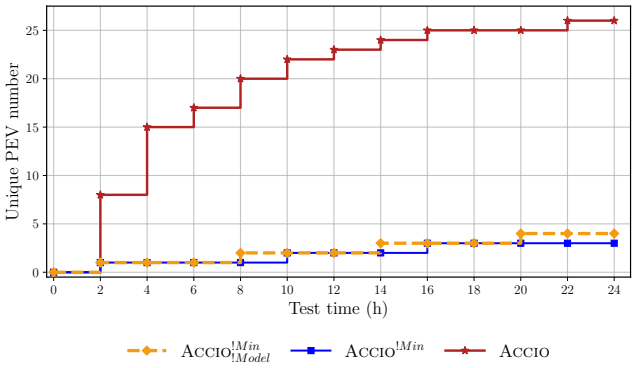

Figure 6: The number of unique PEVs detected over time by Accio in 24 hours.

33 of them, showing that Accio is effective in detecting historical PEVs in their corresponding DBMS versions.

Table 6: Rediscovery of historical PEVs by Accio. Among the 63 historical PEVs collected in our motivation study, we exclude 21 cases whose corresponding DBMS versions are too old to run on current OS environments, and 6 cases that only existed in internal development builds and never appeared in released versions.

DBMS	MySQL	MariaDB	TiDB	ClickHouse	Firebird	Total
Historical	3	4	17	2	10	36
Reproduced	3	4	17	2	7	33

Accio failed to reproduce 3 PEVs in the study since these cases fall outside the detection boundary of Accio. Accio is designed to identify missing privilege checks by detecting uncovered database objects, while these three PEVs do not trigger the uncovered-database-object condition. One missed case is rooted in an operation-privilege mismatch, where the DBMS applies the privilege check to the command but enforces an incorrect privilege type. Accio misses this PEV because it cannot distinguish cases where the user already has some privileges on the objects, even if the privilege type is incorrect. The other two missed cases depend on specific database environments and configurations. For example, one of them requires the presence of a specific shared library (`.so`) file in the file system. It can only be triggered by crafting SQL commands that reference the file path to this external library file. Such PEVs are difficult for Accio to detect, since Accio's privilege model is constructed over database objects in the database state, while the external files cannot be modeled by Accio's current design.

7 Discussion

Coverage of Accio's SQL Generator. Currently, Accio focuses on the SQL command types that are most likely to trigger PEVs according to our motivation study, including DDL, DML, DQL, and administration commands on tables, views, sequences, and procedures. Although this coverage already enables Accio to detect previously unknown PEVs in widely used DBMSs, it does not cover

all SQL grammars and privilege–object combinations supported by modern DBMSs. In particular, some DBMS-specific SQL commands, object types that require special database states, and commands that depend on specific runtime environments remain outside the current SQL generator and privilege model. As a result, Accio may miss PEVs that can only be triggered by these unsupported SQL commands. A possible direction for improvement is to extend the SQL generator with more command types and clause combinations. It can also further combine coverage-guided DBMS fuzzing so that more complex SQL commands can be explored, with minimum privilege set calculation deployed as the test oracle.

Adapting Accio to other DBMSs. Accio mainly targets SQL DBMSs whose privileges are defined over database objects and can be configured through `GRANT` commands. Adapting Accio to these DBMSs requires minimal effort and involves two primary steps. (1) The first step is to configure the system table queries. We need to write the SQL commands to retrieve the privilege types, object scopes, and schema information from the DBMS's system tables (e.g., `information_schema`). This step typically requires understanding the DBMS's metadata structure. (2) The second step is to implement the client interface. This interface encapsulates the functions for connection management, query execution, and error handling. Since most DBMSs provide standard Python drivers, developers only need to implement wrappers around these drivers to adapt Accio to the target system.

For DBMSs that do not directly follow the Information Schema and the `GRANT` privilege model, Accio can still be adapted. However, this adaptation requires more vendor-specific effort in privilege model construction. In such cases, the privilege types and their dependencies may need to be collected from vendor-specific system tables or documentation. The grammar rules of privilege configuration commands may also need adaptation. Therefore, the current design of Accio is general for SQL DBMSs with object-scoped privileges, but its adaptation effort still depends on the metadata interface and privilege design of the target DBMS. Beyond SQL DBMSs, the idea of minimum privilege set calculation may still be useful for non-SQL DBMSs and other systems with configurable privilege entries. However, these systems usually define privileges over different resources and operations. They need a different privilege model and a corresponding oracle for checking whether the required privileges are enforced correctly.

8 Related Work

Access Control Testing. Access control testing has been extensively explored in systems such as the Linux kernel, operating systems, and distributed cloud systems. Existing techniques generally fall into two categories: static analysis and dynamic analysis. Static methods typically perform rule-based analysis of code paths and policy specifications [13, 14, 22–24, 27, 56]. For example, BolaRay automatically infers object-level authorization models from database schemas and application code, then reports object accesses that miss the required object-level checks [14]. MPChecker combines log analysis and static analysis to identify privileged operations in distributed cloud systems and then detects missing permission checks [24]. PeX statically maps Linux kernel permission checks to their privileged functions, and scans all user-reachable paths to find incorrect checks [56]. Dynamic methods run the target systems with different users or roles, and check at runtime whether the observed permissions match basic security expectations [4, 9, 23, 57, 58]. BACScan relies on runtime feedback from HTML responses to tell whether attacker-issued requests succeed in accessing or altering victim-specific data [23]. DYNAMO uses runtime instrumentation to log all security checks on each API call, then detects missing or inconsistent checks as permission vulnerabilities [9].

Accio differs from these works because it neither analyzes source code nor relies on HTTP-level interaction. Focusing on DBMSs rather than applications or OSs, Accio bases its test oracle on a privilege model and the minimum privilege set that allows the execution of SQL commands, instead of on response similarity, taint tracking, or manually crafted scenarios.

Permission Mining and Refinement. Researchers have explored approaches that infer or refine permission configurations from source code, historical logs, audit records, and existing policies [19, 46, 54, 55]. These works aim to help administrators or system components obtain reasonable permission settings in deployed systems. For example, AutoArmor automatically generates inter-service access control policies for microservices from invocation logic extracted by static analysis [19]. P-DIFF infers policies from historical access logs to support continuous validation and forensics [54]. CASPR recommends and refines SELinux policy rules based on context-aware information such as policy rules, file locations, audit logs, and attribute information [55]. These approaches are useful for policy generation and policy repair.

However, these works do not test whether a target system correctly enforces permissions during execution. They derive an approximate privilege set from the available context rather than a minimum one, which is not suitable for PEV detection. In contrast, Accio targets PEVs caused by incorrect access control implementation in DBMSs. It calculates the minimum privilege set required for each SQL command through execution feedback, and then detects PEVs by checking for uncovered exposed database objects.

DBMS Testing. Due to their complexity, DBMSs often suffer from a wide range of vulnerabilities. Accordingly, researchers have developed various testing frameworks to detect bugs, primarily focusing on performance, logic, and crash issues. Performance bug testing aims to find unexpected slowdowns or inefficient execution in DBMSs [2, 21, 53]. For instance, Mozi transforms the configurations of DBMSs into equivalent variants and compares query runtime to uncover performance bugs [21]. CERT checks whether estimated cardinalities decrease when a query is rewritten to be more restrictive [2].

Logic bug testing focuses on finding cases where the DBMS finishes execution but produces wrong results [1, 3, 42–44]. These works usually rely on test oracles, such as comparing outputs of equivalent queries or cross-checking multiple executions, to expose inconsistencies. For example, PQS generates queries that must return a randomly chosen pivot row and reports a bug whenever the pivot row is missing from the result [44]. QPG guides the optimizer to produce different query plans for the same SQL command and then checks whether all these plans return identical results [1]. EET applies equivalent expression transformation and checks whether the transformed query still returns the same result as the original query [18]. TxCheck detects logic bugs in database transaction by

constructing semantically equivalent test cases from fine-grained statement-level dependencies in transactions [17].

Crash bug testing aims to identify inputs or runtime conditions that cause a DBMS to crash, hang, or exit unexpectedly [11, 12, 20, 45, 59]. These works typically focus on generating many valid and diverse SQL commands, feeding them to DBMSs, and reporting a bug when an abnormal termination is observed. For example, SQLsmith generates test cases by modeling the grammar rules of the target DBMSs and constructing ASTs [45]. Griffin utilizes a grammar-free mutation-based method to generate SQL commands [11]. Lego learns affinities between SQL command types and uses them to generate diverse SQL sequences [20]. DynSQL proposes a stateful fuzzing approach that incrementally generates complex and valid SQL commands by collecting DBMS-specific state information after each command execution [16].

Accio differs from these works by treating access control as the primary test target rather than query logic, crash, or performance. It varies privilege assignments and object scopes instead of fixing a single user configuration. Accio builds a privilege model, computes the minimum privilege set for each SQL command, and detects potential missing privilege checks. This process enables Accio to expose subtle PEVs that are invisible to prior DBMS fuzzers.

9 Conclusion

In this paper, we focus on detecting privilege escalation vulnerabilities (PEVs) caused by implementation flaws in DBMS access control. We first conduct an in-depth motivation study of such PEVs across several widely-used DBMSs. Our study revealed that the main root cause of PEVs is missing privilege checks of related database objects in the access control implementation. Based on these findings, we designed Accio, a testing framework aimed at detecting PEVs in DBMS implementations. Accio constructs a privilege model to capture privilege semantics and employs a divide-and-conquer strategy to calculate the minimum privilege set required for each SQL command. Finally, it performs missing privilege check detection, which checks for uncovered objects in the SQL commands and result sets against the calculated minimum privilege set. We evaluated Accio on seven widely used DBMSs and discovered 26 previously unknown PEVs. All these PEVs have been confirmed, and 23 have been fixed by the developers. Our future work will focus on enhancing Accio to support more complex access control scenarios and applying it to more DBMSs.

Acknowledgments

We thank the reviewers and the shepherd for their valuable comments. This research is partly sponsored by NSFC Program (No. 62302256, U2441238, 625B2100), the Fundamental Research Funds for the Central Universities (No. JKF-2026052809520), CCF-Tencent Rhino-Bird Open Research Fund (No. RAGR20250134), and Huawei-Tsinghua Database Testing Research Project (No. 20252001670).

Ethical Considerations

In conducting our research, we carefully considered the ethical implications of detecting PEVs in DBMSs. All discovered vulnerabilities were promptly reported to the respective DBMS vendors through their official security channels or bug tracking systems. We provided detailed reproduction steps to assist developers in verifying and fixing the issues. Furthermore, to prevent potential misuse, we have withheld specific exploit scripts and technical details for vulnerabilities that have not yet been published or fixed.

Regarding CVE assignment, for the PEVs whose bug reports and patches have been made public by the vendors, we have submitted CVE requests. For the remaining reported PEVs, we do not disclose additional technical details before the vendors complete their fix and disclosure process. For example, Oracle Database informed us that a CVE will be assigned after the patched version is released.

Additionally, all experiments were performed in a strictly controlled, isolated environment. We deployed the target DBMSs using Docker containers hosted on our local servers. Our research did not involve interactions with production systems or unauthorized operational services. This ensured that our vulnerability detection was conducted without any risk to the stability or security of real-world database deployments or their users.

Compliance with Open Science Policy

We fully support the open science policy and provide the artifact of Accio at https://anonymous.4open.science/r/Accio-C598/.

References

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

[2] 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.

[3] Jinsheng Ba and Manuel Rigger. 2024. Keep It Simple: Testing Databases via Differential Query Plans. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[4] Yunang Chen, Yue Gao, Nick Ceccio, Rahul Chatterjee, Kassem Fawaz, and Earlence Fernandes. 2022. Experimental security analysis of the app model in business collaboration platforms. In *31st USENIX Security Symposium (USENIX Security 22)*. 2011–2028.

[5] MariaDB Corporation. 2025. MariaDB. https://mariadb.org/. Accessed: July 28, 2026.

[6] MariaDB Corporation. 2025. MariaDB's JIRA Instance. https://jira.mariadb.org/. Accessed: July 28, 2026.

[7] National Vulnerability Database. 2016. NVD - CVE-2016-6662. https://nvd.nist.gov/vuln/detail/CVE-2016-6662. Accessed: July 28, 2026.

[8] National Vulnerability Database. 2025. NVD - CVE-2025-8107. https://nvd.nist.gov/vuln/detail/CVE-2025-8107. Accessed: July 28, 2026.

[9] Abdallah Dawoud and Sven Bugiel. 2021. Bringing balance to the force: Dynamic analysis of the android application framework. *Bringing balance to the force: dynamic analysis of the android application framework* (2021).

[10] Firebird. 2025. Firebird. https://firebirdsql.org/. Accessed: July 28, 2026.

[11] 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.

[12] 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 20th European Conference on Computer Systems (EuroSys)*. 1061–1076.

[13] 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.

[14] 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.

[15] ClickHouse Inc. 2025. ClickHouse. https://clickhouse.com/. Accessed: July 28, 2026.

[16] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. {DynSQL}: Stateful fuzzing for database management systems with complex and valid {SQL} query generation. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4949–4965.

[17] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting transactional bugs in database engines via {graph-based} oracle construction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 397–417.

[18] Zu-Ming Jiang and Zhendong Su. 2024. Detecting logic bugs in database engines via equivalent expression transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 821–835.

[19] Xing Li, Yan Chen, Zhiqiang Lin, Xiao Wang, and Jim Hao Chen. 2021. Automatic policy generation for {Inter-Service} access control of microservices. In *30th USENIX Security Symposium (USENIX Security 21)*. 3971–3988.

[20] 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.

[21] 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.

[22] Fengyu Liu, Youkun Shi, Yuan Zhang, Guangliang Yang, Enhao Li, and Min Yang. 2025. MOCGuard: Automatically Detecting Missing-Owner-Check Vulnerabilities in Java Web Applications. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 903–919.

[23] Fengyu Liu, Yuan Zhang, Enhao Li, Wei Meng, Youkun Shi, Qianheng Wang, Chenlin Wang, Zihan Lin, and Min Yang. 2025. BACScan: Automatic Black-Box Detection of Broken-Access-Control Vulnerabilities in Web Applications. (2025).

[24] 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.

[25] Toby Mao. 2023. SQLGlot Website. https://sqlglot.com/sqlglot.html. Accessed: July 28, 2026.

[26] MITRE. 2025. Privilege Escalation. https://attack.mitre.org/tactics/TA0004/. Accessed: July 28, 2026.

[27] 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.

[28] OceanBase. 2025. OceanBase. https://en.oceanbase.com/. Accessed: July 28, 2026.

[29] Oracle. 2025. MySQL. https://www.mysql.com/. Accessed: July 28, 2026.

[30] Oracle. 2025. MySQL Bugs. https://bugs.mysql.com/. Accessed: July 28, 2026.

[31] Oracle. 2025. MySQL Documentation: Access Control and Account Management. https://dev.mysql.com/doc/refman/9.2/en/access-control.html. Accessed: July 28, 2026.

[32] Oracle. 2025. MySQL Documentation: GRANT Statement. https://dev.mysql.com/doc/refman/9.2/en/grant.html. Accessed: July 28, 2026.

[33] Oracle. 2025. MySQL Documentation: Privileges Provided by MySQL. https://dev.mysql.com/doc/refman/9.2/en/privileges-provided.html. Accessed: July 28, 2026.

[34] Oracle. 2025. MySQL Documentation: Using Roles. https://dev.mysql.com/doc/refman/9.2/en/roles.html. Accessed: July 28, 2026.

[35] Oracle. 2025. Oracle. https://www.oracle.com/. Accessed: July 28, 2026.

[36] OWASP. 2025. Broken Access Control. https://owasp.org/Top10/A01_2021-Broken_Access_Control/. Accessed: July 28, 2026.

[37] PingCAP. 2025. TiDB. https://www.pingcap.com/. Accessed: July 28, 2026.

[38] PingCAP. 2025. TiDB GitHub. https://github.com/pingcap/tidb. Accessed: July 28, 2026.

[39] PortSwigger. 2025. Access control vulnerabilities and privilege escalation. https://portswigger.net/web-security/access-control. Accessed: July 28, 2026.

[40] NLTK Project. 2025. NLTK: Natural Language Toolkit. https://www.nltk.org/. Accessed: July 28, 2026.

[41] NLTK Project. 2026. AddressSanitizer. https://clang.llvm.org/docs/AddressSanitizer.html. Accessed: July 28, 2026.

[42] 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.

[43] 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.

[44] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 667–682.

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

[46] Ruowen Wang, William Enck, Douglas Reeves, Xinwen Zhang, Peng Ning, Dingbang Xu, Wu Zhou, and Ahmed M Azab. 2015. {EASEAndroid}: Automatic policy analysis and refinement for security enhanced android via {Large-Scale}{Semi-Supervised} learning. In *24th USENIX Security Symposium (USENIX Security 15)*. 351–366.

[47] Wikipedia. 2024. Database. https://en.wikipedia.org/wiki/Database. Accessed: July 28, 2026.

[48] Wikipedia. 2025. Data control language. https://en.wikipedia.org/wiki/Data_control_language. Accessed: July 28, 2026.

[49] Wikipedia. 2025. Data definition language. https://en.wikipedia.org/wiki/Data_definition_language. Accessed: July 28, 2026.

[50] Wikipedia. 2025. Data manipulation language. https://en.wikipedia.org/wiki/Data_manipulation_language. Accessed: July 28, 2026.

[51] Wikipedia. 2025. Data query language. https://en.wikipedia.org/wiki/Data_query_language. Accessed: July 28, 2026.

[52] Wikipedia. 2025. SQL. https://en.wikipedia.org/wiki/SQL. Accessed: July 28, 2026.

[53] 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.

[54] Chengcheng Xiang, Yudong Wu, Bingyu Shen, Mingyao Shen, Haochen Huang, Tianyin Xu, Yuanyuan Zhou, Cindy Moore, Xinxin Jin, and Tianwei Sheng. 2019. Towards continuous access control validation and forensics. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 113–129.

[55] Lifang Xiao, Hanyu Wang, Aimin Yu, Lixin Zhao, and Dan Meng. 2025. CASPR: Context-Aware Security Policy Recommendation.. In *NDSS*.

[56] 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.

[57] Yuan Zhang, Min Yang, Guofei Gu, and Hao Chen. 2016. Rethinking permission enforcement mechanism on mobile systems. *IEEE Transactions on Information Forensics and Security* 11, 10 (2016), 2227–2240.

[58] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. 2013. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 611–622.

[59] 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.