MOCGuard: Automatically Detecting Missing-Owner-Check Vulnerabilities in Java Web Applications

Fengyu Liu*, Youkun Shi*, Yuan Zhang, Guangliang Yang, Enhao Li, Min Yang
*Fudan University, * co-first authors*

***Abstract*—Java web applications have been extensively utilized for hosting and powering high-value commercial websites. However, their intricate complexities leave them susceptible to a critical security flaw, named Missing-Owner-Check (MOC), that may expose websites to unauthorized access and data breaches. However, the research on identifying and analyzing MOC vulnerabilities has been limited over the years.**

In this work, we propose a novel end-to-end vulnerability analysis approach, called `MOCGuard`, that can effectively vet Java web applications against MOC issues. Different from related techniques, `MOCGuard` pinpoints MOC vulnerabilities from a new perspective of database-centric analysis. `MOCGuard` first applies database structure analysis to infer user table and user-owned data. Then, `MOCGuard` conducts insecure access checks across both the Java and SQL layers. To thoroughly evaluate the effectiveness of `MOCGuard`, we collaborated with a world-leading tech company. Through our evaluation of 30 high-profile open-source Java web applications and 7 industrial Java web applications, we demonstrate that `MOCGuard` is automatic and effective. Consequently, it successfully uncovered 161 (confirmed) 0-day MOC vulnerabilities, leading to the assignment of 73 CVE identifiers.

1. Introduction

With the robust ecosystem and reliable performance, Java is extensively utilized in the development of web applications, including notable examples such as Amazon [1], Paypal [2] and Shein [3]. These applications store and manage a vast array of sensitive user resources, including account details, financial information, and personal data, making them prime targets for web attackers. To safeguard these sensitive and valuable assets (namely *user-owned data*), implementing ownership verification or *owner checks* for every access request to user-owned data is critical. Overlooking these owner checks may lead to serious security flaws, referred to as Missing-Owner-Check (MOC), posing a substantial risk to user privacy and potentially jeopardizing financial security [4], [5]. For example, a MOC vulnerability was recently found in the U.S. Postal Service (USPS). It allowed any logged-in user to access the account details belonging to other users, ultimately leading to the exposure of the privacy data of up to 60 million users [4].

To detect MOC vulnerabilities in real-world Java web applications, an intuitive vulnerability detection idea is identifying *user-owned data* and vetting its corresponding *owner checks*, i.e., verifying whether the accessed data belongs to the currently logged-in user. Although this idea is straightforward, it is difficult to directly utilize or extend existing techniques to achieve the goal in practice. Two key perspectives should be carefully considered:

- *Q1: How to infer user-owned data within the context of Java web applications?* Accurate identification of user-owned data is crucial for effective detection of MOC vulnerabilities. High inaccuracies can lead to false positives or negatives during vulnerability detection. Applications usually utilize variables to access or represent user-owned data. However, it is challenging to pinpoint user-owned data from thousands of variables and data items available in modern web applications.
- *Q2: How to detect the missing of owner checks when accessing user-owned data?* Even assuming we successfully identify all user-owned data, it is still difficult to analyze whether the access to this data is secure. In particular, the owner-check implementation is quite flexible and may be performed through SQL (e.g., `WHERE` clause) or Java level (e.g., if-statement).

However, existing techniques faced significant difficulties in addressing the above critical challenges. Their methods can be divided into two groups according to how they analyze user-owned data and owner-checks. The first type of techniques (e.g., RoleCast [6] and MPChecker [7]) utilized heuristics to identify crucial variables holding user-owned data through code-level analysis. However, such heuristics are ineffective or infeasible in the domain of the Java web area. For example, RoleCast depended on JSP (Java Server Pages [8]) file structures, which were unavailable in modern template-based web applications. MPChecker relied on system logs tailored for distributed systems. Moreover, in addressing the second challenge, they overlooked the owner check occurring at the SQL level. They potentially treated a large number of safe owner-checks as vulnerable, thereby potentially resulting in lots of false positives.

The second category (e.g., MACE [9]) leveraged an insight that user variables are essential to manipulate user-owned data. Therefore, the existing technique tracked the data flow of the manually annotated user variables (that store user identities) flowing to SQL statements and identified the user-owned data. However, this approach is still limited. First, they heavily relied on the manual annotation of the user variables. Such manual markings are labor-intensive and prone to errors. Second, the result (of the found user-owned data) fell short of completeness when analyzing SQL statements to infer user-owned data from user identities. In practice, the design of application databases is often sophisticated, where much user-owned data is manipulated through data element IDs, rather than user identities. Thus, existing coarse-grained methods may result in a high rate of false negatives for identifying user-owned data, significantly impacting the effective detection of MOC issues (70.81% false negatives in our dataset).

In this paper, we propose MOCGuard, a novel security analysis approach for detecting MOC vulnerabilities in Java web applications. The key observation behind our MOCGuard approach is that modern Java web applications need to manage a huge amount of user data using relational databases. Their database structures (e.g., database tables) are often meticulously designed. This means the complex database structures encode their semantics, reflecting both the crucial details of used-owned data and associated ownership information. Following this key observation, we design a novel database-centric analysis technique to understand user-owned data and its corresponding users, and vet the security of owner checks in Java web applications.

Our MOCGuard approach consists of two primary phases. In the first phase, MOCGuard conducts the database semantic analysis to pinpoint user-tables that store user information, and then infer other tables housing associated user-owned data. In the user-owned data inference, MOCGuard performs two-pronged analysis in terms of data structure analysis and cross-layer code analysis. In the second phase, MOCGuard effectively identifies its owner checks based on the found user-owned data, and carefully examines the security of these owner checks at both the SQL and Java layers. MOCGuard effectively verifies whether the necessary owner checks are in place to protect user-owned data, enabling MOCGuard to identify potential insecure access points and find MOC vulnerabilities effectively.

Last, we evaluate the effectiveness and performance of MOCGuard in 30 widely-used open-source Java web applications and 7 industrial applications. As a result, MOCGuard successfully discovered 161 (confirmed) high-risk zero-day MOC vulnerabilities. In contrast to the state-of-the-art work (i.e., the improved Java version of MACE [9]), MOCGuard demonstrates equally impressive performance, detecting 114 more vulnerabilities and surpassing it by 31.31% in precision and 242.55% in recall.

These newly-found MOC vulnerabilities may be exploited to compromise user privacy or even delete data stored within applications, thereby severely compromising data confidentiality and integrity. Moreover, attackers can leverage identified payment hijacking vulnerabilities to facilitate unauthorized transactions, which can result in substantial financial losses. Considering the significant security impact of these vulnerabilities, we responsibly reported them to the developers of the vulnerable applications. As of now, these vulnerabilities have been assigned 73 CVE identifiers.

We release the source code of MOCGuard to facilitate further research on the security of Java web applications.

The contributions of this paper are summarized as follows:

- In this paper, we study the semantics of database structure, and propose several new key observations and insights about how to leverage database structure semantics to address the problem of MOC detection.
- Following the 'database-speaks-for-itself' idea, we propose a novel database-centric approach to effectively infer user-owned data and vet the security of corresponding owner checks against MOC vulnerabilities, without manual interventions.
- Our evaluation on 30 real-world popular web applications and 7 industrial web applications shows MOCGuard is automatic and effective, with the discovery and confirmation of 161 (confirmed) 0-day MOC vulnerabilities. As of now, these vulnerabilities have been assigned 73 CVE identifiers.

2. Problem Statement

2.1. MOC Vulnerability Definition

Modern web applications have evolved into sophisticated platforms housing extensive data, with one of the most critical elements being *user-owned data*. User-owned data includes users' high-value assets and privacy information, e.g., ID card, addresses, and order records. Thus, its security is crucial. When web application developers do not appropriately safeguard user-owned data, a serious missing-owner-check (MOC) vulnerability may arise, leading to serious consequences[4], [5], [10], [11].

Below we first define several important MOC related concepts.

Definition 1 (Users)**.** U is a set of users available in the target web application (namely A). Thus, $U = \sum_{i=1}^{m} u_i$.

Definition 2 (User-Owned Data)**.** The rich array of data D is owned by users U, and is stored and managed by A. Hence, $D = \sum_{j=1}^{n} d_j$.

Definition 3 (Data Ownership)**.** We define $\text{own}(u_i, d_j)$ to indicate data d_j belongs to user u_i within the context of A.

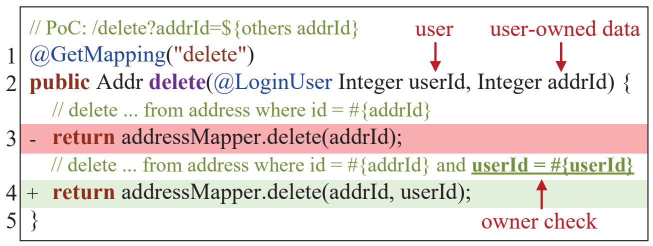
Definition 4 (Data Access)**.** We define $\text{access}(u_i, d_j)$ as a data access operation, where requester u_i accesses data d_j in A.

Upon these concepts, we define MOC vulnerability:

Definition 5 (MOC Vulnerability)**.** A $\text{MOC}(u_i, d_j)$ vulnerability exists when

$$\exists u_i \in U, \exists d_j \in D, \neg\text{own}(u_i, d_j) \land \text{access}(u_i, d_j)$$

holds. When user u_i accesses data d_j, u_i should be d_j's owner. Otherwise, an illegitimate access occurs.



```
  // PoC: /delete?addrId=${others addrId}
1 @GetMapping("delete")
2 public Addr delete(@LoginUser Integer userId, Integer addrId) {
     // delete ... from address where id = #{addrId}
3 -  return addressMapper.delete(addrId);
     // delete ... from address where id = #{addrId} and userId = #{userId}
4 +  return addressMapper.delete(addrId, userId);
5 }
```

Figure 1: A real-world MOC vulnerability patch on litemall; the owner check implemented at SQL-layer.

2.2. Threat Model

The user-owned data in Java web applications are the major attack targets of remote malicious users [7]. In general, these resources are stored in databases and accessed through database operations. Intuitively, the access and manipulation of user-owned data should be verified by testing the *owner check*. Otherwise, attackers can unauthorizedly access sensitive resources belonging to other users, i.e., access(u_i, sr_j) is true and own(u_i, sr_j) is false, ultimately resulting in MOC vulnerabilities.

2.3. Real-World MOC Example

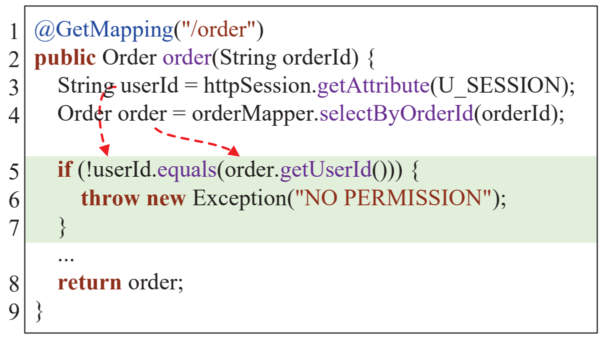
We explore a real-world vulnerable web application to demonstrate MOC vulnerability. Figure 1 shows its simplified code snippet. At line 3, `"addressMapper.delete()"` executes SQL to access and delete user-owned data, which is saved in the variable `"addrId"`. However, crucial data ownership, i.e. `own(userId, addrId)`, is not appropriately verified. Thus, this causes a serious MOC vulnerability and may break the integrity and availability of the vulnerable web application.

Line 4 demonstrates a security patch. It checks the data ownership, with the 'WHERE' clause in SQL, i.e., `userId = #`{`userId`}. Thus, own(u_i, sr_j) is true, which indicates that the MOC vulnerability is fixed.

2.4. Detection Challenges

Given the serious security hazards that MOC vulnerabilities pose to user assets and privacy, early detection and remediation are of utmost importance. In general, MOC vulnerability detection can follow MOC's definition and consider the following two key perspectives.

❶ *How to infer user-owned data within the context of Java web applications?* Accurately identifying user-owned data is the first crucial step in detecting MOC vulnerabilities. Any errors or oversights in this process can directly lead to false positives or false negatives during vulnerability detection. However, this is not a trivial task. As demonstrated, user-owned data is often represented or accessed through variables (e.g., `addrID`) within the target application. Pinpointing these specific variables among thousands present in the application can be complex and challenging.



```
1 @GetMapping("/order")
2 public Order order(String orderId) {
3    String userId = httpSession.getAttribute(U_SESSION);
4    Order order = orderMapper.selectByOrderId(orderId);

5    if (!userId.equals(order.getUserId())) {
6       throw new Exception("NO PERMISSION");
7    }
     ...
8    return order;
9 }
```

Figure 2: The owner check implemented at Java-layer (the lines on light green background).

❷ *How to detect the missing of owner checks when accessing user-owned data?* After identifying the user-owned data, the next step is to analyze whether the access to this data is secure. Intuitively, source-to-sink analysis is quite suitable. First of all, we can identify which operation (i.e., MOC sink) can manipulate the user-owned data. Then, we need to analyze which MOC sinks are reachable for user input (i.e., MOC source) from the program entry. Last, we should analyze whether there are appropriate protections on the program path from MOC source to sink (i.e., owner check).

Nevertheless, achieving high accuracy in MOC vulnerability detection, especially analyzing owner checks, is challenging. The reasons come from two aspects. On one hand, identifying the owner check on the source-to-sink path is difficult. Java language does not provide built-in functions for owner checking. Instead, these are often implemented by the developers themselves, allowing for great flexibility. As demonstrated in Section 2.3, Java web applications can implement owner checks at the SQL-layer. Beyond this, it can be achieved at the Java code-layer as well. For example, as shown in lines 5-7 of Figure 2, developers use if conditional statements to verify data ownership, i.e. verifying that the requester `"userId"` should be the data owner `"orderId.order.getUserId()"`.

On the other hand, analyzing whether the owner check is appropriate for user-owned data is also difficult. Given the possible existence of various user identities in the system, different data have different owners (e.g., products owned by sellers and orders owned by buyers in the mall application). A coarse-grained approach that only checks for the existence of protection, without considering whether the protection is appropriate or not, will directly lead to inaccuracies in vulnerability detection.

2.5. Existing Techniques and Limitations

In recent years, a set of existing works have explored various techniques to protect user-owned data. However, they exhibited significant limitations and faced challenges when being extended to Java web applications. We categorize these works into two groups and separately analyze the root reasons for their limitations.

The first category of work (e.g., RoleCast [6] and MPChecker [7]) utilized heuristics to identify crucial variables holding user-owned data through code-level analysis. However, such heuristics are ineffective or infeasible in the domain of the Java web area. For example, as Java has evolved rapidly, the methodology used in RoleCast, which centers on inferring critical variables through analyzing JSP (Java Server Pages) file structures, is no longer applicable for today's template-based Java web applications, which are primarily developed through frameworks, e.g., Spring Boot [12]. Similarly, MPChecker relied on specific formatted system runtime logs to infer user-owned data, which are unique to distributed systems and do not exist in Java web applications. Moreover, in addressing the second challenge, their technique overlooked the owner check occurring at the SQL level. This could potentially lead to a large number of false positives as demonstrated in our ablation study §5.4.

The second category (e.g., MACE [9]) leveraged an insight that user variables are an essential bridge to manipulate user-owned data, which is usually saved in databases. Thus, it first manually annotated user variables that store user identities, such as super-global variables like `$_SESSION` in PHP. Then, it tracked the data flow of these user variables to `INSERT` statement variables (as sinks) for identifying the SQL statements associated with user-owned data. Last, it understood the SQL statements and found the data items to be manipulated as user-owned data.

However, this approach is still limited. First, it heavily relies on the manual annotation of user variables, which requires significant human effort. The original paper noted that annotating two applications in their evaluation took about 50 minutes. This time increases for Java web applications due to fundamental differences between Java and PHP. In PHP, user or user-owned data is often represented by super-global variables with distinct characteristics (e.g., `$_SESSION`) and limited types. Java lacks this concept, requiring security experts to rely entirely on code context, making annotations for inferring user-owned data very challenging. Furthermore, the analysis to infer user-owned data from user variables is incomplete. Application databases are often complex, with much user-owned data manipulated via data element IDs rather than user variables. Consequently, this coarse-grained approach results in a high rate of false negatives for identifying user-owned data and detecting MOC issues (70.81% false negatives in our dataset).

3. `MOCGuard` Methodology

In this paper, we aim to address the problem of MOC detection. It is clear that relying solely on code-layer analysis struggles to address the two key concerns that were raised for MOC vulnerability detection in §2.4. We believe the fundamental challenge here is that, at the code layer, the ownership relationship between the user and data is not explicitly defined and is intertwined within diverse code contexts, making it quite challenging to identify.

In this work, instead of a standalone code-level analysis, we propose a novel database-centric approach, which has natural advantages in addressing this fundamental challenge. Inspired by Spider-Scents [13], the key idea behind the approach is that as web applications need to manage a significant amount of user-owned data, they often meticulously design their *database structures*, i.e., the organization of database tables and each table's columns. This indicates that complex database structures encode rich dependency relationships among user data, reflecting both the crucial details of these data and their associated ownership information.

Below we refine our database-centric idea. We first present our key observations about database structure. Then, we introduce our main idea for MOC vulnerability detection. Last, we illustrate its workflow by running a real-world example.

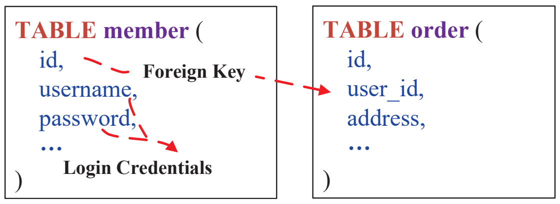
3.1. Key Insights and Observations

In this section, we conclude four key observations about database structure by following the 'database-speaks-for-itself' idea, and explain why they can help design a novel and effective vulnerability detection approach.

Observation#1: *Database structure contains user credentials and can help identify users.* Web applications, given their complex functionalities and vast user bases, frequently manage user authentication through user registration and login functionalities. Correspondingly, to validate user identity during the login process, developers tend to store authentication-related information for each user in the database. We refer to the table containing authentication data related to user identity as *user-table*. Consequently, the semantics of the column names often conceal evidence pertaining to the authentication processes. This evidence suggests that the *user-table* typically exhibits distinct characteristics. A typical example is the `member` table shown in Figure 3. It stores user credentials through columns such as `username` and `password`, and records unique user identifiers using the primary key `id`.

Observation#2: *Database structure can help explicitly infer user-owned data from user-table.* To conveniently record user-owned data D and its corresponding users U, D and U can be stored within the same table in the database, and linked to the user-table through structural connections, such as foreign keys. We refer to all the database tables containing user-owned data as *user-owned tables*. As depicted in Figure 3, the '`order`' user-owned table explicitly includes the '`user_id`' key to denote the order's owner. The '`user_id`' column is a foreign key linked to the primary key '`id`' in the '`member`' user table.

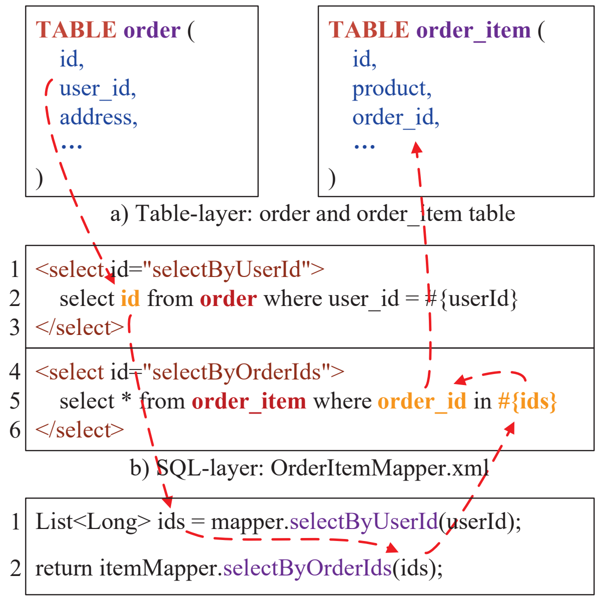
Observation#3: *There are also user-owned data with implicit connection to the user table, due to database normalization [14].* We observe that developers frequently transfer data between different user-owned tables through the code level. Specifically, developers tend to use variables in the code level to accommodate user-owned data read from databases, and then pass these variables as arguments to other methods performing different database operations. We refer to these related database tables as implicit user-owned



```
TABLE member (
    id,
    username,
    password,
    ...
)
```

```
TABLE order (
    id,
    user_id,
    address,
    ...
)
```

Figure 3: User Table and Explicit User-Owned Table.



```
TABLE order (
    id,
    user_id,
    address,
    ...
)
```

```
TABLE order_item (
    id,
    product,
    order_id,
    ...
)
```

a) Table-layer: order and order_item table

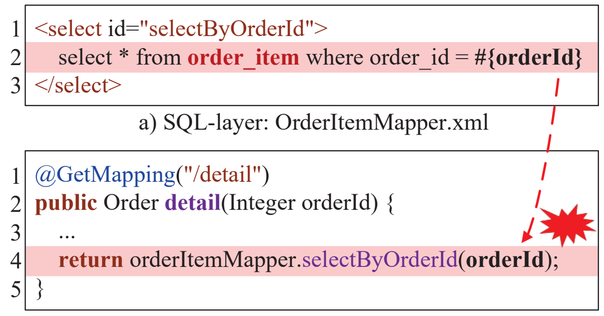
```
1 <select id="selectByUserId">
2     select id from order where user_id = #{userId}
3 </select>
4 <select id="selectByOrderIds">
5     select * from order_item where order_id in #{ids}
6 </select>
```

b) SQL-layer: OrderItemMapper.xml

```
1 List<Long> ids = mapper.selectByUserId(userId);
2 return itemMapper.selectByOrderIds(ids);
```

c) Java-layer: OrderController.java

Figure 4: Real-World Implicit User-Owned Table.



```
1 <select id="selectByOrderId">
2     select * from order_item where order_id = #{orderId}
3 </select>
```

a) SQL-layer: OrderItemMapper.xml

```
1 @GetMapping("/detail")
2 public Order detail(Integer orderId) {
3     ...
4     return orderItemMapper.selectByOrderId(orderId);
5 }
```

b) Java-layer: OrderController.java

Figure 5: Simplified Real-World 0-Day Vulnerable Code.

tables. Figure 4 demonstrates implicit user-owned tables over sensitive variables. Developers retrieve the order 'id' belonging to a specific user from the '`order`' table, and save the content to an '`ids`' variable (line 1 of Figure 4(c)). Subsequently, developers pass the '`ids`' variable into a subsequent Java SQL-operation statement (line 2 of Figure 4(c)) to query the detailed order entries stored within the `order_item` table. Through the sensitive variable '`ids`', we can link implicit user-owned tables. Moreover, the sensitive variable is also crucial for determining its owner checks (i.e., the security of data access requests).

Observation#4: *Operations on user-owned data that lack owner check protections can lead to insecure access.* Upon the data D being accessed, developers should carefully verify D's owner against the requester, i.e., *owner check*. We find the owner check can be conducted in two layers: SQL (e.g., the WHERE clause) and the Java code (e.g., if-statement). If owner check is neglected or incomplete, malicious users will be allowed to access user-owned data belonging to others without proper authorization. Figure 5 showcases a real-world 0-day vulnerability (See more details in Section 3.3). In line 4 of Figure 5 (b), the `selectByOrderId` method accesses the `order_item` table, which is identified as a user-owned table. However, due to the absence of owner check protections, this results in insecure access.

3.2. Approach Overview

Drawing on our observations, we propose a novel database-centric approach that can effectively address the MOC detection from the two key perspectives, i.e., inferring user-owned data and vetting the security of data access. Below, we delve into these two phases with detailed explanations.

3.2.1. User-owned data inference. In the first stage, utilizing the key observations (Section 3.1), we design a database-semantic analysis technique to identify user-table and further infer user-owned data. First, we conduct database semantic analysis to pinpoint the columns in the database table that function as login credentials. The tables housing these columns can be classified as user-table. Then, our approach performs a two-pronged database structure analysis on the previously identified user tables to further explore user-owned tables: one via foreign key analysis, and the other through cross-layer code analysis. The former aids in identifying the explicit user-owned tables, whereas the latter performs the recognition of the implicit ones. With this two-level analysis, the data within the user-owned tables can be treated as user-owned data. Please note that through the implicit user-owned table analysis, sensitive variables can also be identified, which are crucial for the second analysis phase of owner checks.

Specifically, for user table reference, our approach parses the database files (e.g., `.sql` files) in the target web application to extract all columns of database tables. We apply a list of keywords to match the columns associated with user authentication, and treat the tables hosting these crucial columns as user tables. For explicit user-owned tables, we pinpoint foreign keys by conducting syntactic analysis against the `CREATE TABLE` statements in the database file. Upon these foreign keys, we can recognize the tables that are linked via identified foreign keys to the primary key of the user table. Thus, these tables can be classified as user-owned tables.

For implicit user-owned tables, our approach leverages the above explicit user-owned tables to infer the implicit tables through code-level analysis. Since there exist data flows between sensitive variables (accommodating user-owned data) in the Java code layer and the corresponding columns in the SQL layer, we propose a cross-layer user-own data tracking analysis. In the Java layer, our approach employs data flow analysis to identify sensitive variables reading and holding data from user-owned tables, and track them. In the SQL layer, our approach directly analyzes the SQL statements to establish data flow connections between columns in the SQL statements, as well as marking the variables passed into them.

3.2.2. Insecure access detection. In the second stage, adhering to the viewpoint outlined in §2.4, our approach performs a source-to-sink analysis to detect MOC vulnerabilities. This process is divided into three steps.

First, our approach analyzes all SQL statements within the target application to single out those specifically querying user-owned tables. Following this, our approach treats the database operations in Java code that manipulate these identified SQL statements as MOC sinks. Second, our approach locates the call sites of the MOC sinks within the application using a call graph. Following the call sites, we conduct a backward dataflow analysis on their parameters, and trace back to the program's entry point for further ascertaining the MOC sources. Finally, our approach conducts a two-tiered analysis from both the Java code and SQL layer to determine whether each source-to-sink path has appropriate owner-check protection, thereby detecting MOC vulnerabilities.

Specifically, for the owner checks at the SQL layer, developers implemented these through `WHERE` clauses in SQL statements. Therefore, our approach focuses on each SQL statement that queries a user-owned table, analyzing whether its `WHERE` clauses contain only user-owned columns and not user columns. If this is the case, the database operation that manipulates the SQL statement is marked as lacking SQL-layer owner check protection. For the owner checks at the Java code layer, developers generally implemented them with if-conditional statements at the code level to protect the operations on variables representing user-owned data [9], [7]. Therefore, our approach can treat the if-conditional statements that meet the following constraints as owner checks: 1) verifying variables accommodating user-owned data, and 2) disrupting the execution of operations in the control flow level (when denied).

Ultimately, by employing the identified owner checks, our approach assesses whether the owner checks exist on the source-to-sink path and reports paths where such checks are absent as MOC vulnerabilities.

3.3. Running Real-World Example

We explore a real-world example to further illustrate our approach. The Java web application *mall* is an open-source and high-profile e-commerce system (more than 70,000 stars in open-source communities). We identified and verified a critical previously unknown MOC vulnerability (CVE-2023-33***), allowing a remote attacker to access all user-owned data without any restrictions.

Its technical details have been discussed in §3.1 with three key figures: Figure 3, Figure 4 and Figure 5. Upon them, we further explain how our approach 1) refers explicit user-owned data from the user table (Figure 3), i.e., inferring the `order` table as a user-owned table, 2) conducts implicit user-owned table (Figure 4), i.e., identifying the `order_item` table as a user-owned table, and 3) detects the MOC vulnerability caused by insecure access when accessing the `order_item` table (Figure 5), i.e., the owner checks missing both in the SQL and Java layers. Combining these three key workflows, we discuss how our approach successfully detects the MOC vulnerability, showcasing an end-to-end MOC vulnerability detection process.

User-owned Data Inference. We first describe how our approach to infer `order_item` table as a user-owned table. First, in Figure 3, our approach infers the `member` table as a user table, by directly matching its column names with a list of keywords related to authentication, e.g., "username" and "password". Second, our approach infers explicit user-owned tables. As described in §3.2, by analyzing the `CREATE TABLE` SQL statements in the database file, we discover a foreign key connection between the `member` table and the `order` table in Figure 3, thus recognizing the `order` table as an explicit user-owned table.

Third, our approach infers implicit user-owned tables. The red line in Figure 4 illustrates the data flow between the `order` table and the `order_item` table. Specifically, the `selectByUserId` database operation queries the `id` column of the `order` table (line 2 of Figure 4(b)) and stores the results in the `ids` variable within the Java code (line 1 of Figure 4(c)). Thus, we link the `id` column of `order` table in the SQL layer with the `ids` variable in the Java layer, i.e., `order.id` $\rightarrow$ `${ids}`. Then, in line 2 of Figure 4(c), the `ids` variable is passed as a parameter to the `selectByOrderIds` method and used as a condition for the `order_id` column, i.e., `where order_id in ${ids}`(line 5 of Figure 4(b)). Consequently, we establish a data flow connection between the `ids` parameter and the `order_id` column of the `order_item` table, i.e., `${ids}` $\rightarrow$ `order_item.order_id`. By integrating these data flow connections, we successfully establish the connection between the `order` table and the `order_item` table, thus deducing the `order_item` table as a user-owned table. Please note that we mark the '`ids`' variables as sensitive variables.

MOC Vulnerability Detection. We now describe how our approach detects the MOC vulnerability in Figure 5. First, as described using §3.2. Our approach extracts all source-to-sink paths that access user-owned tables. Figure 5(b) shows that the `detail` method on line 2 is a program entry point, where users can view order details by passing the `orderId` parameter. Through data flow analysis, we found that the user-input `orderId` parameter flows into the `selectByOrderId` method on line 7, establishing a source-

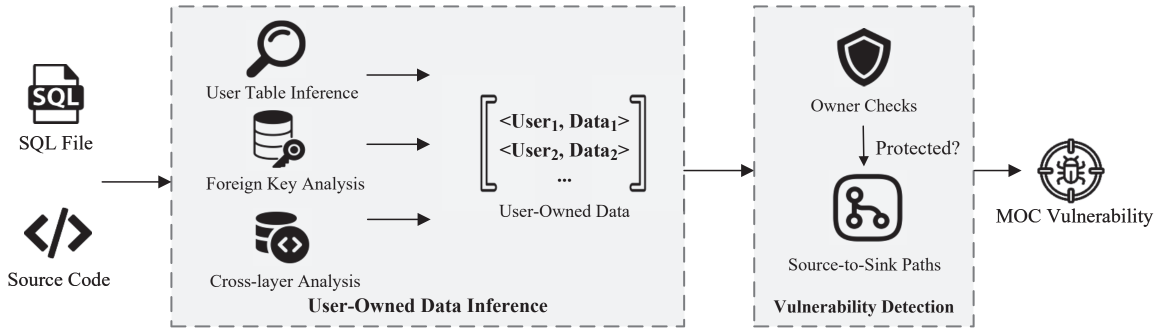


Figure 6: The Architecture of `MOCGuard`.

to-sink path. Then, by delving into the method body of the `selectByOrderId` method, we found that it executes the SQL statement shown in Figure 5(a), i.e., `select * from order_item where order_id = #{orderId}`. It operates on the previously inferred implicit user-owned table, namely `order_item`.

Second, our approach determines whether the MOC sinks are protected by owner checks and reports vulnerabilities where these checks are absent. By examining the context of the operation, we found that there are no owner checks at either the Java layer or the SQL layer. Therefore, malicious users can retrieve the order details of other users by supplying any order number. This means, access(u_i, d_j) holds while own(u_i, d_j) does not, thereby leading to a MOC vulnerability.

4. MOCGuard Design

Following our idea presented in §3, we propose `MOCGuard`, an end-to-end approach for detecting MOC vulnerabilities in Java web applications. As illustrated in Figure 6, `MOCGuard` primarily consists of two phases.

- *User-owned Data Inference (§4.1)* employs data structure analysis and cross-layer data flow analysis to infer user-owned data automatically.
- *MOC Vulnerabilities Detection (§4.2)* identifies the owner checks and detects MOC vulnerabilities within the target application.

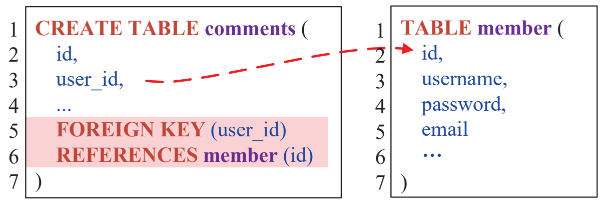
4.1. User-owned Data Inference

4.1.1. User Table Identification. In this phase, `MOCGuard` proceeds with database semantic analysis to infer user tables. Specifically, `MOCGuard` first parses the SQL file within the target application to pinpoint all the tables and columns. The SQL File denotes the files ending with a `.sql` suffix, intended for the initialization of the database structure within the application. It is noteworthy that the `.sql` files for database creation are readily available in our dataset, encompassing 37 applications.

Then, `MOCGuard` identifies user tables through keyword matching. We find that web developers often use similar column-naming styles to save credentials and authentication information. Thus, we create an expandable list of associated keywords (e.g. password) to pinpoint authentication-related columns in database tables. These keywords originate from our observation in §3.1, i.e., the column names used to represent login credentials often have consistent naming words across different applications, such as ‘password’. Inspired by this, our dictionary consists of three commonly used word stems for naming user login credentials, i.e., ‘pass’, ‘token’, and ‘auth’. We can directly search the table columns using these keywords and thus find user tables. Furthermore, to improve the matching rate, `MOCGuard` employs program analysis to locate the call sites to database operations that manipulate these tables, and extract these caller methods' names. If these method names are authentication-related, the corresponding database tables (these methods access) can also be user tables.

4.1.2. Foreign Key Analysis. In this phase, upon the found user tables, `MOCGuard` utilizes foreign key analysis to infer explicit user-owned tables. The first step is to extract the foreign keys between tables in the target application. To ensure referential integrity of the data stored in the database and enhance performance, development guidelines often recommend adding foreign keys to columns that have data dependency with other tables [15]. Typically, these foreign keys are explicitly defined in `.sql` database files. Therefore, `MOCGuard` extracts the foreign key relationships in the target application by parsing the user-provided database files. Specifically, `MOCGuard` extracts SQL statements from the database file and performs syntactic analysis on the `CREATE TABLE` statements. Then, `MOCGuard` identifies foreign key relationships between two tables through the `FOREIGN KEY` and `REFERENCES` clauses. Taking Figure 7 as an example, by parsing the foreign key clause (lines 5-6) in the create table statement, `MOCGuard` identifies that there is a foreign key relationship between the `user_id` column of the `comments` table and the `id` column of the `member` table. Then, leveraging the extracted foreign key constraints, `MOCGuard` identifies tables that are associated via the foreign key to the primary key of the user table as the explicit user-owned tables.

4.1.3. Cross-layer Code Analysis. In the third phase, `MOCGuard` further infers implicit user-owned data within



```
CREATE TABLE comments (
    id,
    user_id,
    ...
    FOREIGN KEY (user_id)
    REFERENCES member (id)
)
```

```
TABLE member (
    id,
    username,
    password,
    email
    ...
)
```

Figure 7: An example of foreign key.

the application using cross-layer code analysis, which effectively bridges the data flow between SQL and Java layers.

❶ In the Java layer, the data flow analysis is applied. One important issue is the data flow interruption caused by broken edges in these tools significantly affects the effectiveness of data flow analysis. To mitigate this problem, we have expanded the data flow capabilities by incorporating additional taint steps. We have modeled typical scenarios of broken edges, i.e., transfer rules [16]. For instance, for the frequently used `valueOf` method in Java, `MOCGuard` adds a data flow edge between the method's parameters and its return value, allowing the parameters of `valueOf` to taint its return value. In the case of the commonly used for-loop statement `for (Long num : nums)`, which implicitly declares the num variable, `MOCGuard` adds a data flow edge between `nums` and `num`.

❷ In the SQL layer, `MOCGuard` establishes the data flow association between the variables flowing from/into the SQL statements and the corresponding columns, such as the queried column of database operations. Specifically, the data flow in the SQL layer can be categorized into outflows and inflows. For outflows, `MOCGuard` identifies and extracts the tables and columns queried through `SELECT` statements. It then establishes data flow associations between the variables used in Java code to store results and the columns queried in these SQL statements. It is important to note that `MOCGuard` primarily focuses on `SELECT` statements because the return values from other SQL statement types, such as `UPDATE` and `DELETE`, generally only indicate the success or failure of the operations rather than providing data outputs.

Regarding inflows, `MOCGuard` initially analyzes the structure of SQL statements related to database operations to ascertain whether parameters from the database operations are integrated into the SQL statements, thereby bridging the gap between the two layers. Following this, `MOCGuard` examines which parts of the SQL statement receive parameters and establishes data flow relationships between these incoming variables and the corresponding columns. For instance, for variables that flow into the `WHERE` clause, `MOCGuard` analyzes the syntax of the `WHERE` clause to identify the specific columns associated with these incoming variables.

Inferring Implicit User-Owned Table. Now, we utilize cross-layer analysis to infer implicit user-owned tables within the application. To more clearly illustrate the design of `MOCGuard`, we use Algorithm 1 to demonstrate the algorithm for inferring implicit user-owned data. Based

Algorithm 1: Implicit User-owned Data Inference

```
   Input  : Set of inferred user-owned tables $T$
   Output: Set of implicit user-owned data $D$
 1 Function CrossLayerAnalysis($vars$):
 2     $pt \leftarrow \emptyset$;
 3     $relatedTables \leftarrow$ TraceDataFlowToSQL($vars$);
 4     foreach $rt \in relatedTables$ do
 5         $pt \leftarrow pt \cup \{rt\}$;
 6     end
 7     return $pt$;

 8 $D \leftarrow \emptyset$;
 9 $pending \leftarrow T$;
10 while $pending \neq \emptyset$ do
11     $t \leftarrow$ Pop($pending$);
12     $vars \leftarrow$ RetrieveSQLResult($t$);
13     $pt \leftarrow$ CrossLayerAnalysis($vars$);
14     foreach $p \in pt$ do
15         if $p \notin D$ then
16             $D \leftarrow D \cup \{p\}$;
17             $pending \leftarrow pending \cup \{p\}$;
18         end
19     end
20 end
```

on key observations described in §3.1, `MOCGuard` infers these implicit user-owned tables by conducting cross-layer data flow analysis centered around the inferred explicit user-owned tables. Specifically, this analysis examines the data flow between the SQL and Java layers, proceeding through the following three steps.

- First, `MOCGuard` correlates database operations with SQL statements within the application, and extracts all database operations that interact with inferred explicit user-owned tables. As shown in lines 11-12 of algorithm 1, the `RetrieveSQLResult` function retrieves all database operations that manipulate explicit user-owned tables.
- Next, `MOCGuard` considers the return value of callsites for these database operations that manipulate user-owned tables as the starting points for cross-layer data flow analysis, and takes the variables in `WHERE` clauses of other SQL statements as the endpoints. Lines 1-7 of algorithm 1 describe the `CrossLayerAnalysis` function. This function performs the cross-layer data flow analysis to track variable data flows between the Java layer and SQL layer, effectively tracking all data flows from the starting points to these endpoints.
- Lastly, drawing from the results of cross-layer data flow analysis, `MOCGuard` identifies tables that have data flow associations with explicit user-owned tables as implicit user-owned tables(Line 14-19 of algorithm 1).

4.2. MOC Vulnerability Detection

Based on the user-owned tables inferred from the previous stage, `MOCGuard` identifies the owner checks and detects MOC vulnerabilities. Below we present more technical details.

4.2.1. Source-to-Sink. According to §2.4, the first step in detecting MOC vulnerabilities is to identify source-to-sink paths. It is evident that only database operations accessing user-owned tables can potentially have MOC vulnerabilities. Therefore, `MOCGuard` leverages inferred user-owned tables to identify MOC sinks. Specifically, `MOCGuard` first extracts all database operations within the application, and then analyzes the corresponding SQL statements to determine whether they manipulate user-owned tables. For database operations that manipulate user-owned tables, `MOCGuard` conducts a backward data-flow dependency analysis on their parameters, tracing back to the program's entry point to further identify the MOC sources. Finally, `MOCGuard` extracts these identified source-to-sink paths that contain user-controllable variables for further analysis in the next step.

4.2.2. Owner Check Identification. `MOCGuard` identifies the owner checks within the application. Typically, these checks should include two components: the user-owned tables and their corresponding users. When interacting with user-owned tables, owner checks are implemented to protect them by verifying whether the currently logged-in user matches the designated owner of these data. However, identifying these checks based on the inferred user-owned tables within the application is not a straightforward task. This complexity arises from the diverse practices of developers, leading to the owner check being implemented in various highly flexible ways. Below we detail how `MOCGuard` utilizes inferred user-owned tables to identify owner checks at the SQL and Java layers, respectively.

SQL Layer Analysis. SQL-layer check is a common practice for owner checks. When operating on user-owned data, developers restrict the user of these data within the conditional expression of the SQL statement, thereby limiting operations to user-owned data that belong to others. Take the `WHERE` clause as an example, for each database operation, `MOCGuard` extracts the `WHERE` clause of the SQL statement. Then, `MOCGuard` analyzes the columns contained in the `WHERE` clause. Through column name matching analysis, if it contains restrictions related to the user column, `MOCGuard` considers it as an owner check. As previously illustrated in line 4 of Figure 1, developers can restrict users to only delete their owned address by including the `user_id` column in the `WHERE` clause, i.e., `where id=#{addrId} and user_id=#{userId}`. Note that `MOCGuard` requires the variable passed into the user column for SQL-layer checks to be beyond user control. If this criterion is not met, `MOCGuard` will not recognize it as an owner check.

Java Layer Analysis. Java-layer check is another type of owner check conducted at the source code level, which is often achieved through conditional statements [7], [9]. Specifically, before accessing user-owned data through database operations, developers utilize common permission-checking methods in the Java layer, such as `if` statements, to ensure the accessed user-owned data belongs to the currently logged-in user.

To identify these Java-layer owner checks, `MOCGuard` first extracts conditional statements commonly used for permission checks. Then, `MOCGuard` employs data flow analysis to examine the association between these statements and the return value of `SELECT` database operations. Take if statements as an example, which are commonly used for permission checks [6]. `MOCGuard` extracts methods used to evaluate equality within if-conditions, such as `equals` method, and conducts data flow analysis on the arguments of these methods. For one of the arguments, `MOCGuard` requires that it has data flow association with the return value of the database operation. Furthermore, `MOCGuard` leverages the inferred user column to determine whether the argument accesses the corresponding field of the class, which represents the user column, via `getter` method in Java. For the other argument, `MOCGuard` requires it to be uncontrollable by the user. Besides, for developer-defined functions, `MOCGuard` can determine whether it is a wrapper for the owner check by analyzing whether an identified if-statement post-dominates the function's entry point [7]. Finally, `MOCGuard` considers conditional statements that satisfy the above criteria as an owner check.

We illustrate the owner check analysis using lines 5-7 of Figure 2. The `equals` method in line 5 involves two parameters. One of them is derived from user credentials that represent the identity of the currently logged-in user. The other comes from database operations, representing the user corresponding to the operated data. By comparing these two, the equals method can verify whether the data operated by the `SELECT` operation belongs to the currently logged-in user. Specifically, the `order` variable in the `equals` comes from the result of the `SELECT` database operation, which retrieves order details based on the order number (line 4). The `userId` variable represents the identity of the currently logged-in user. By comparing the value of two variables, the `equals` method can verify whether the queried order belongs to the current user. If the two values are not equal, the `throw` statement in the `if` block will raise an exception and terminate the execution of the program, thereby restricting unauthorized operation to the user-owned data.

4.2.3. MOC Vulnerability Determination. Based on the identified owner checks and source-to-sink paths, `MOCGuard` conducts a two-tiered analysis from both the Java code and SQL layer to detect MOC vulnerabilities in the target application. Specifically, for each source-to-sink path, `MOCGuard` analyzes whether there are owner checks protecting user-owned data. For the SQL layer, `MOCGuard` examines whether the SQL statement executed by the database operation includes an SQL-layer check. For the Java layer, `MOCGuard` inspects whether a Java-layer check is conducted within the control flow of the database operation. If neither type of check is present, MOCGuard will report a MOC vulnerability.

5. Evaluation

Our evaluation is organized by answering the following four research questions:

- RQ1: How effective is MOCGuard at detecting MOC vulnerabilities in real-world applications? (in §5.2)
- RQ2: How does the effectiveness of MOCGuard compare with the state-of-the-art approach? (in §5.3)
- RQ3: How do the different layer owner checks considered by MOCGuard contribute to its success? (in §5.4)
- RQ4: How efficient is MOCGuard in performing the end-to-end analysis? (in §5.5)

5.1. Experimental Setup

Implementation. We implemented a prototype of MOCGuard for Java web applications. As for now, our prototype supports commonly used Java web technologies, such as J2EE Servlets [17] and the Spring framework [12]. For database operations, the prototype can process JDBC APIs [18] and commonly used ORM frameworks, such as MyBatis [19] and Hibernate [20]. For various static analysis tasks, we utilized the CodeQL static analysis framework [21]. For tasks involving database semantics and foreign key analysis, Python was employed. Specifically, the CodeQL scripts are primarily used for data flow analysis and for collecting essential code and database features. The prototype stores the information obtained from CodeQL scripts in SARIF files [22], which are then parsed by Python scripts for further analysis. In total, the entire prototype consists of 4,520 lines of code. All experiments run on a Ubuntu 20.04 machine, equipped with 64 cores CPU and 245 GB memory.

Dataset. Our dataset includes 30 open-source Java web applications and 7 industrial Java web applications. We provide detailed information about these applications in Table 5 of the Appendix §A.1.

For the open-source applications, we collected 30 widely-used Java web applications from popular open-source repositories (e.g., GitHub [23]) based on the following steps. First, we applied GitHub's Java language filter and sorted them by their star numbers, which reflected their popularity. Note that collecting popular open-source applications as datasets based on stars on GitHub is a common practice [24], [25]. Next, we used various keywords to gather open-source web applications from different categories, such as 'CMS', 'blog', 'development', and 'e-commerce'. Finally, we selected the 30 applications with the highest star counts, and excluded tutorial applications, such as demos and study projects, to ensure the practicality of the applications. As a result, our dataset includes 18 applications with over 1k stars and 4 applications with over 10k stars. These applications span across multiple domains, including e-commerce websites, content management systems, blog systems, backend management systems, and development platforms, which we believe are representative. It is worth noting that the majority of these applications have been utilized in existing research [24], [26], [27], thereby validating the reliability of the dataset.

We also included 7 industrial Java web applications in the dataset. Specifically, to evaluate the effectiveness of MOCGuard in an industrial context, we collaborated with a world-leading tech company that provides services to over a billion users. For code security considerations, the company selected 7 of its core applications which represent complex business scenarios, and applied MOCGuard to analyze them. Therefore, we believe that evaluating such a representative dataset can effectively assess MOCGuard's precision rates, thereby demonstrating its generality and effectiveness.

5.2. Effectiveness: MOC Detection (RQ1)

In this experiment, we evaluated the effectiveness of MOCGuard in detecting MOC vulnerabilities across the entire dataset.

Results Overview. Table 1 presents the detailed detection results of MOCGuard. Overall, MOCGuard successfully identified 180 vulnerable MOC sinks that lack proper ownership verification, which were reported as potential MOC vulnerabilities. Finally, through manual PoC construction and testing, we confirmed 161 MOC vulnerabilities across the entire dataset, achieving a recall rate of 89.44%.

Table 2 provides a detailed breakdown of the distribution of these vulnerabilities in both open-source and industrial applications. Specifically, for the 30 open-source applications, MOCGuard reported a total of 128 MOC vulnerabilities within 17 open-source applications, of which 116 were true positives, resulting in an accuracy rate of 90.63%. For the 7 industrial applications, MOCGuard reported a total of 52 MOC vulnerabilities with an accuracy rate of 86.54%.

Vulnerability Verification. To verify the accuracy of the tool's report, we established the runtime environment for each application. For open-source applications, we opted for local deployment testing during the vulnerability verification phase. For the industrial web applications, we conducted testing in a mirrored environment in collaboration with our

TABLE 1: Effectiveness of MOCGuard in MOC detection (RQ1).

Type	# MOC Sinks	# Reported Vulnerabilities		
		TP	FP	Prec(%)
SELECT	1,113	79	19	80.61%
DELETE	482	23	0	100.00%
UPDATE	637	42	0	100.00%
INSERT	361	17	0	100.00%
Total	2,593	161	19	89.44%

TABLE 2: Distribution of detected MOC vulnerabilities in open-source and industrial applications (RQ1).

Application	TP	FP	Prec(%)
Open-source	116	12	90.63%
Industrial	45	7	86.54%
Total	161	19	89.44%

partner company. It is important to note that this process did not involve any user privacy or sensitive data.

Moreover, the output of `MOCGuard` provides the key details needed to verify the detected vulnerabilities, including the exact path from user input to database operations and the user-controlled parameters. This report facilitates the efficient construction of PoC. For instance, the output for the MOC vulnerability depicted in Figure 5 includes the path to access the endpoint (`"/detail"`) and the user-controlled parameter (`orderId`). As a result, constructing a PoC becomes straightforward using the provided details: `/detail?orderId=${OrderId owned by others}`.

Vulnerability Disclosure. After evaluating these vulnerabilities through specifically crafted exploits, we believe they pose a serious security risk. For open-source applications, attackers can exploit these vulnerabilities to cause the leakage of user privacy or even delete data stored within the application, thereby severely compromising data confidentiality and integrity. Moreover, attackers can leverage identified payment hijacking vulnerabilities to facilitate unauthorized transactions, which can result in substantial financial losses. These security breaches highlight the critical need for effective vulnerability detection mechanisms to safeguard both user privacy and assets. Therefore, we promptly reported all the confirmed vulnerabilities to the developers of the vulnerable applications where they were found. As of now, 73 vulnerabilities have been granted CVE identifiers. For industrial applications, the detected 45 MOC vulnerabilities are all newly identified. These vulnerabilities pose a significant threat to the company's data security, potentially resulting in the leakage of substantial employee and user information and adversely impacting the business functionality of applications. These findings demonstrate `MOCGuard`'s practical utility.

False Positive Analysis. Next, we introduce the 19 false positives that `MOCGuard` reported during vulnerability detection. After conducting a thorough analysis, we found that they all stem from the same reason: *it's hard to discern the intentions of developers regarding data accessibility.* Specifically, these false positives all originate from database query operations (i.e., SELECT type). Let us consider a scenario involving an e-commerce website. When sellers update product prices, it is crucial to perform an owner check to prevent them from arbitrarily changing the prices of products that belong to others. However, buyers can view the price of any product, which implies that the database operation of viewing product prices does not require an owner check. Therefore, for the product table, `MOCGuard` would consider the database operations querying product information via `SELECT` as a MOC vulnerability, which could result in false positives. We argue that discerning developers' intentions regarding resource access is an inherent challenge in static analysis, which has been mentioned in many existing works [9], [28].

5.3. Effectiveness: Comparison (RQ2)

In this experiment, we compare the effectiveness of `MOCGuard` with the state-of-the-art technique (i.e., MACE [9]) across the entire dataset.

Baseline Setup: MACE-Java. MACE [9] is the current state-of-the-art approach, which considers protection at both the code-layer and the SQL-layer during MOC vulnerability detection. Therefore, we adopt it as our baseline. Given its design for PHP and not open-source, we followed the methodology detailed in its original paper and implemented a Java version of MACE, named `MACE-Java`. However, we find there are several difficulties that should be carefully dealt with.

Applying MACE to Java is not a trivial task, the challenges come from two aspects. On one hand, the inputs required by MACE are very difficult to provide, within the context of Java web applications. MACE relies on PHP language features, such as super-global variables, and requires manual annotation to identify users. However, similar super-global variables do not exist in Java, and manually identifying variables that represent users from the thousands of variables in Java code is undoubtedly a challenging task. Therefore, we use *the user columns automatically inferred by `MOCGuard` as input to `MACE-Java`.*

On the other hand, the implementation of MACE's inconsistency analysis detection strategy requires a large amount of engineering work compared to `MOCGuard`. Due to the flexible coding practices adopted by developers, determining whether owner checks are consistent across two paths is a complex task. To achieve this, `MACE-Java` first formalizes the extracted owner checks to enable accurate determination of consistency in checks across different paths during inconsistent analysis. Specifically, `MACE-Java` extracts variable and method names from the owner checks and replaces variable names with specific class names. Then, `MACE-Java` sorts these symbols lexicographically and performs a hash operation on the sorted list. If the hashes are identical, it is considered that the same owner check is present in both paths. In total, the implementation of `MACE-Java` used 941 lines of code.

Ground Truth Construction. Comparing the effectiveness of each work in vulnerability detection ideally requires a comprehensive enumeration of all vulnerabilities within the dataset, which is infeasible [29], [30]. Therefore, to ensure a fair comparison, we followed the widely used method [31] of constructing a ground truth aggregating all vulnerabilities detected by both `MOCGuard` and `MACE-Java` in our dataset. It's worth noting that all the vulnerabilities

TABLE 3: Comparison between MOCGuard and MACE-Java (RQ2).

Baselines	TP	FP	FN	Prec(%)	Recall(%)
MACE-Java	47	22	114	68.12%	29.19%
MOCGuard	161	19	0	89.44%	100.00%

TABLE 4: Ablation study for two variants of MOCGuard (RQ3).

Baselines	TP	FP	Prec(%)
MOCGuard-NoSQLCheck	161	91	63.89%
MOCGuard-NoJavaCheck	161	72	69.10%
MOCGuard	161	19	89.44%

involved in the ground truth are carefully manually confirmed and tested with PoC, ensuring that they are indeed real vulnerabilities. In all, our ground truth consists of 161 vulnerabilities.

Result Overview. Table 3 describes the results of the effectiveness comparison between MOCGuard and MACE-Java across the entire dataset. Overall, MOCGuard surpasses MACE-Java by 31.31% in precision and 242.55% in recall. Notably, against the ground truth of 161 vulnerabilities, MOCGuard successfully identifies all of them, whereas MACE-Java detects only 47 MOC vulnerabilities and reports 22 false positives. This clearly illustrates the superior capability of MOCGuard in effectively identifying MOC vulnerabilities.

False Positive Analysis. In total, MACE-Java reported 22 false positives. After rigorous analysis, we identified that apart from 10 cases also reported by MOCGuard, MACE-Java detected 12 additional false positives. The primary reason for this is that MACE-Java lacks the capability to comprehend the ownership relationship between the user and the user-owned data. Consequently, MACE-Java is unable to precisely determine whether there are owner checks when a database operation accesses user-owned data, ultimately leading to false positives.

False Negative Analysis. For 114 false negatives, we now detail why MACE-Java failed to detect them. As previously mentioned in §2.5, MACE employs an inconsistent protection analysis strategy to detect vulnerabilities. However, this strategy relies on the assumption that developers aim to get most checks right, with only some occasional checks performed incorrectly. Therefore, in cases where the developer fails to implement any of the checks correctly, MACE will be unable to detect all vulnerabilities in the target application. As described in [7], more than 60% of vulnerabilities are triggered by operations that are not guarded anywhere in the program. Consequently, this has led to 114 false negatives in MACE-Java.

5.4. Ablation Study (RQ3)

In this experiment, we separately disabled the owner check analysis module of MOCGuard at the Java-code layer and SQL-code layer, to demonstrate its importance for high-precision detection of MOC vulnerabilities. The details of the two variants are as follows:

- **MOCGuard-NoSQLCheck**: In this variant, we disabled the owner check analysis module at the SQL-layer of MOCGuard, making it consistent with many existing techniques [6], [28], focusing only on the code layer's protection during vulnerability detection.
- **MOCGuard-NoJavaCheck**: In this variant, we disabled the owner check analysis module at the Java code-layer of MOCGuard, to evaluate the precision of MOC detection when only considering protection at the SQL-layer.

Table 4 presents the comparison results of MOCGuard and its two variants against the entire dataset. It is evident that the owner checks analysis modules at both the Java code-layer and the SQL-layer are crucial for the precise detection of vulnerabilities by MOCGuard. When the owner-check analysis at the SQL-layer is disabled, the detection precision decreases by 28.57% compared to the original one. Similarly, when the owner checks analysis at the Java code-layer is disabled, the detection precision also drops significantly by 22.74%. The newly introduced false positives would greatly increase the analysis requirements for MOCGuard end-users, involving substantial human efforts.

5.5. Efficiency (RQ4)

In this experiment, we evaluated the efficiency of MOCGuard in performing the end-to-end analysis across the entire dataset. In total, MOCGuard took about 47 minutes to finish the analysis task of 37 target applications, with an average of 76.22 seconds per application. The efficiency of the analysis is attributed to our database-semantic analysis approach. Compared to traditional static analysis, MOCGuard does not require heavyweight analysis of the source code of the target application, nor does it rely on any human efforts to provide application-specific inputs. In contrast, the related work, MACE [9], as described in its original paper, necessitates manual annotation, with each application averaging several tens of minutes for this process. In comparison, MOCGuard is more lightweight and efficient in detecting MOC vulnerabilities.

6. Case Study

Here, we showcase two MOC vulnerabilities identified by MOCGuard in highly popular applications, further illustrating the high risk posed by MOC vulnerabilities and demonstrating the practical utility of MOCGuard in real-world scenarios.

```
1  //PoC:/pay?orderId=${OrderId owned by others}
2  public Integer pay(Long orderId) {
3      Order order = new Order();
4      order.setId(orderId);
5      order.setPayStatus(1);
6      ...
7      // update order set pay_status=1 where id=#{
       ↪ id}
8      orderMapper.updateByPrimaryKey(order);
9      ...
10 }
```

Figure 8: The vulnerable code of payment hijacking in mall application (over 70k stars on GitHub).

6.1. Payment Hijacking in mall Application (over 70k stars on GitHub)

The *mall* is an open-source and widely used e-commerce application with over 70,000 stars on GitHub. As shown in Figure 8, MOCGuard detected a MOC vulnerability within the application that could lead to payment hijacking. The vulnerable database operation updateByPrimaryKey is located within the pay() method, which manipulates the user-owned table order. This database operation is responsible for setting the payment status of an order, achieved by modifying the pay_status column through an update statement (line 8). The parameter of the setPayStatus method is set to ”1”, which signifies that the order has been paid for (line 5). Due to the lack of owner checks for the order, attackers can input any order number to set an unpaid order to a paid status, which can cause serious financial loss to the merchant. Given the extensive potential damage posed by this vulnerability, we immediately reported this critical issue to the developers and received a CVE (CVE-2023-49***).

```
//PoC:/detail?orderId=${OrderId owned by others}
@PostMapping("detail")
public Object detail(Integer orderId) {
    ...
    Map orderGoodsParam = new HashMap();
    orderGoodsParam.put("id", orderId);
    // select * from nideshop_order_goods where
    ↪ order_id=#{id}
    List<OrderGoods> orderGoods =
    ↪ orderGoodsService.queryList(
    ↪ orderGoodsParam);
    ...
}
```

Figure 9: The vulnerable code of arbitrary order details leakage in platform application (over 20k stars on Gitee).

6.2. Arbitrary Order Details Leakage in platform Application (over 20k stars on Gitee)

The *platform* is a highly popular application with over 20k stars on Gitee, widely used for deploying e-commerce websites. Figure 9 illustrates a MOC vulnerability reported by MOCGuard in this application that could lead to arbitrary order details being stolen. In the detail() method, users can query goods in order by providing the orderId parameter, which is stored in the nideshop_order_goods table. Due to the developer’s oversight, there was no validation of ownership for the nideshop_order_goods table. Attackers can arbitrarily access goods in other users’ orders by iterating through all stored order numbers in the database. Given the severity of this vulnerability, we immediately reported it to the developers of the vulnerable application and actively discussed a solution. As a result, we were granted a CVE identifier, i.e., CVE-2023-37***.

7. Discussion

Adaptability. We devised MOCGuard to efficiently detect MOC vulnerabilities. Since Java web applications typically store user-owned data in relational databases to ensure their integrity and reliability, our design primarily targets relational databases. Currently, the prototype of MOCGuard is capable of detecting MOC issues in Java web applications that are built upon mainstream relational database management systems (DBMS), such as MySQL and PostgreSQL. Since our MOCGuard approach is general, it can be easily extended to other relational databases. Furthermore, due to its database-centered analysis, MOCGuard’s dependency on the programming language is minimal. Consequently, we believe that the key idea of the MOCGuard approach can be extended to other domains, including PHP applications.

Owner Inference Improvement. Authentication-related semantic analysis has significantly assisted us in inferring the owner column in MOCGuard. However, a minor portion of false positives still occur due to unusual naming conventions. One important reason is the difficulty in determining whether an owner column name pertains to authentication, making it challenging to identify as an owner column. In the future, we plan to employ more robust analysis technologies (such as Large Language Models (LLM) [32] for comprehending column name meanings) to mitigate this issue.

Legality and ethicality. This study has not presented any legal or ethical issues. We obtained the source code for local analysis and responsibly reported all detected vulnerabilities in open-source applications to the CVE Numbering Authority (CNA) [33], and also assisted companies in fixing vulnerabilities in industrial applications. Additionally, we have contacted all the developers regarding the MOC vulnerabilities found in §5.2, and will continue to communicate with them throughout the vulnerability disclosure process.

8. Related Work

Broken Access Control Detection. There are several tools [34], [35], [36], [6], [7], [28], [37], [38], [39], [40], [9] that employed various techniques to detect access control vulnerabilities. Specifically, some of them [34], [35], [36], [6], [7], [28], [40] utilized language-specific native methods or software engineering patterns to identify security checks and detect broken access control vulnerabilities through inconsistent or missing security checks detection. However, these tools overlooked the critical owner check within applications. Another category of tools [37], [39] paid attention to the user-owned data, using this insight to identify broken access control vulnerabilities with a dynamic approach. Specifically, they access privileged operations with different user identities and infer the user-owned tables based on whether the user-specific data in the databases appears in the response of the web pages. However, this method relies heavily on the website's runtime environment and configured user information, which significantly limits its scalability. Unlike these previous efforts, `MOCGuard` eliminates the need for extra input or runtime environment, employing a novel approach to accurately detect MOC vulnerabilities within Java web applications.

Web Vulnerabilities Detection. In recent years, the techniques for automatically detecting vulnerabilities within web applications have been extensively studied. A commonly used technique is static analysis [41], [42], [43], [44], [45], [46]. However, the scope of these tools has generally been confined to identifying injection-based vulnerabilities, thereby lacking effectiveness in detecting access control vulnerabilities that require analysis of access control-related features within the program. Another widely employed technique is dynamic analysis [13], [47], [29], [48]. This technique, however, encountered inherent limitations stemming from the code coverage, which may lead to numerous false negatives. To harness the benefits of both static and dynamic analysis, hybrid analysis [49], [50], [31], [30], [51] has gained increasing popularity in recent years. Similarly, this technique is still constrained by inherent limitations of the code coverage, and therefore faces difficulties in the detection of access control vulnerabilities.

9. Conclusion

In this paper, we propose `MOCGuard`, a novel security-vetting approach that can automatically and effectively detect the MOC vulnerabilities in Java web applications. Leveraging a novel database-centric analysis technique, `MOCGuard` can effectively infer user-owned data and verify the security of target applications. Overall, `MOCGuard` successfully detected 161 high-risk 0-day vulnerabilities in real-world applications, with 73 CVE identifiers assigned.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (62172105, 62202106, 62102093, 62302101, 62172104, 62102091, 62472096, 62402114, 62402116). Yuan Zhang and Min Yang are the corresponding authors. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012). Min Yang is a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

References

[1] "Amazon Official Website," https://www.amazon.com.

[2] "Paypal Official Website," https://www.paypal.com.

[3] "Shein Official Website," https://us.shein.com.

[4] "USPS Exposed Data on 60 Million Users," https://krebsonsecurity.com/2018/11/usps-site-exposed-data-on-60-million-users/, 2018.

[5] "Taobao Data Breach," https://www.cpomagazine.com/cyber-security/web-scraping-on-alibabas-taobao-resulted-in-data-leak-of-1-1-billion-records/, 2021.

[6] S. Son, K. S. McKinley, and V. Shmatikov, "Rolecast: Finding Missing Security Checks When You Do Not Know What Checks Are," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, 2011, pp. 1069–1084.

[7] J. Lu, H. Li, C. Liu, L. Li, and K. Cheng, "Detecting Missing-Permission-Check Vulnerabilities in Distributed Cloud Systems," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[8] "Java Server Pages," https://en.wikipedia.org/wiki/Jakarta_Server_Pages.

[9] M. Monshizadeh, P. Naldurg, and V. Venkatakrishnan, "MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 690–701.

[10] "Facebook Photo Bug," https://techcrunch.com/2018/12/14/facebook-photo-bug/, 2018.

[11] "Twitter Data Breach Timeline," https://cybersecurityforme.com/twitter-data-breaches-timeline/, 2021.

[12] "The Official Document of Spring Framework," https://spring.io/projects/spring-framework.

[13] E. Olsson, B. Eriksson, A. Doupé, and A. Sabelfeld, "{Spider-Scents}: Grey-box database-aware web scanning for stored {XSS}," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 6741–6758.

[14] W. W. Armstrong, "Dependency Structures of Database Relationships," in *IFIP congress*, 1974.

[15] "Usage of Foreign Key," https://cloud.google.com/blog/products/data-analytics/join-optimizations-with-bigquery-primary-and-foreign-keys.

[16] N. Grech and Y. Smaragdakis, "P/taint: Unified Points-to and Taint Analysis," *Proceedings of the ACM on Programming Languages*, 2017.

[17] "The Official Document of J2EE Servlet," https://www.oracle.com/java/technologies/java-servlet-tec.html.

[18] "JDBC," =https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/.

[19] "The Official Document of MyBatis," https://mybatis.org/mybatis-3/.

[20] "The Official Document of Hibernate," https://hibernate.org/orm/.

[21] "The Official Website of CodeQL in Github," https://codeql.github.com/.

[22] "SARIF of CodeQL," https://docs.github.com/en/code-security/codeql-cli/using-the-advanced-functionality-of-the-codeql-cli/sarif-output#about-sarif-output.

[23] "The Official Website of Github," https://github.com/.

[24] M. Chen, T. Tu, H. Zhang, Q. Wen, and W. Wang, "Jasmine: A Static Analysis Framework for Spring Core Technologies," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[25] Y. Shi, Y. Zhang, T. Luo, X. Mao, Y. Cao, Z. Wang, Y. Zhao, Z. Huang, and M. Yang, "Backporting security patches of web applications: A prototype design and implementation on injection vulnerability patches," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1993–2010.

[26] Y. Ouyang, K. Shao, K. Chen, R. Shen, C. Chen, M. Xu, Y. Zhang, and L. Zhang, "Mirrortaint: Practical non-intrusive dynamic taint tracking for jvm-based microservice systems," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2514–2526.

[27] A. Antoniadis, N. Filippakis, P. Krishnan, R. Ramesh, N. Allen, and Y. Smaragdakis, "Static analysis of java enterprise applications: frameworks and caches, the elephants in the room," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 794–807.

[28] S. Son, K. S. McKinley, and V. Shmatikov, "Fix Me Up: Repairing Access-Control Bugs in Web Applications." in *NDSS*. Citeseer, 2013.

[29] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[30] E. Güler, S. Schumilo, M. Schloegel, N. Bars, P. Görz, X. Xu, C. Kaygusuz, and T. Holz, "Atropos: Effective Fuzzing of Web Applications for Server-Side Vulnerabilities," in *USENIX Security Symposium*, 2024.

[31] E. Trickel, F. Pagani, C. Zhu, L. Dresel, G. Vigna, C. Kruegel, R. Wang, T. Bao, Y. Shoshitaishvili, and A. Doupé, "Toss a fault to your witcher: Applying Grey-box Coverage-guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities," in *2023 IEEE symposium on security and privacy (SP)*, 2023.

[32] "Large Language Model," https://en.wikipedia.org/wiki/Large_language_model.

[33] "CVE Program," https://www.cve.org/About/Overview.

[34] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, "AutoISES: Automatically Inferring Security Specification and Detecting Violations." in *USENIX Security Symposium*, 2008.

[35] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "Pex: A Permission Check Analysis Framework for Linux Kernel," in *28th USENIX Security Symposium*, 2019.

[36] F. Sun, L. Xu, and Z. Su, "Static Detection of Access Control Vulnerabilities in Web Applications," in *20th USENIX Security Symposium (USENIX Security 11)*, 2011.

[37] X. Li and Y. Xue, "Block: A Black-box Approach for Detection of State Violation Attacks Towards Web Applications," in *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[38] ——, "Logicscope: Automatic discovery of logic vulnerabilities within web applications," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, 2013, pp. 481–486.

[39] X. Li, X. Si, and Y. Xue, "Automated Black-box Detection of Access Control Vulnerabilities in Web Applications," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, 2014.

[40] J. Zhu, B. Chu, H. Lipford, and T. Thomas, "Mitigating Access Control Vulnerabilities through Interactive Static Analysis," in *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, 2015.

[41] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in *2014 IEEE Symposium on Security and Privacy*, 2014.

[42] C. Luo, P. Li, and W. Meng, "TChecker: Precise Static Inter-Procedural Analysis for Detecting Taint-Style Vulnerabilities in PHP Applications," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[43] P. Li and W. Meng, "Lchecker: Detecting Loose Comparison Bugs in PHP," in *Proceedings of the Web Conference 2021*, 2021.

[44] J. Dahse and T. Holz, "Simulation of Built-in PHP Features for Precise Static Code Analysis," in *NDSS*, 2014.

[45] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of php application vulnerabilities," in *2017 IEEE european symposium on security and privacy (EuroS&P)*. IEEE, 2017, pp. 334–349.

[46] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*, 2006.

[47] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black widow: Blackbox Data-driven Web Scanning," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[48] C. Rossow, "jAk: Using Dynamic Analysis to Crawl and Test Modern Web Applications," in *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings*, 2015.

[49] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[50] F. Gauthier, B. Hassanshahi, B. Selwyn-Smith, T. N. Mai, M. Schlüter, and M. Williams, "Backrest: A Model-based Feedback-driven Greybox Fuzzer for Web Applications," *arXiv preprint arXiv:2108.08455*, 2021.

[51] O. van Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides, and E. Athanasopoulos, "webfuzz: Grey-box Fuzzing for Web Applications," in *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26*, 2021.

Appendix A.

A.1. Dataset Details

Table 5 presents detailed information about the 37 Java web applications in the dataset, including their application names, popularity (i.e., Github stars), total lines of code, the number of MOC vulnerabilities, the number of assigned CVEs, and the categories of the applications.

TABLE 5: Breakdown of our evaluation dataset.

Open-source Applications	**# Stars**	**# LoCs**	**# CVEs / Vulns**	**# Description**
mall	73,362	68,681	2 / 2	E-commerce
platform	20,340	26,466	4 / 5	E-commerce
wechat_applet	11,871	32,215	9 / 11	Development Platform
newbee-mall	10,643	4,937	1 / 1	E-commerce
paascloud	9,796	25,149	1 / 3	Backend Management System
basemall	8,907	92,242	7 / 9	E-commerce
xmall	7,056	27,806	8 / 13	E-commerce
SpringBlade	6,397	6,337	0 / 0	Backend Management System
lamp-cloud	5,369	33,478	1 / 2	Development Platform
RuoYi	4,945	22,440	0 / 0	Backend Management System
ForestBlog	4059	5435	0 / 0	Blog
hope-boot	3,254	5,498	0 / 0	Development Platform
dts-shop	2,892	61,128	3 / 5	E-commerce
PaasJava-Platform	2,564	10,075	0 / 5	Development Platform
xbin-store	2,145	16,098	0 / 0	E-commerce
youlai-mall	1,932	26,830	2 / 2	E-commerce
SuperMarket	1,925	3,268	7 / 8	E-commerce
weiit-saas	1,898	26,130	0 / 0	Content Management Systems
mogu_blog_v2	1,572	26,136	0 / 0	Blog
myblog	1,272	4,887	0 / 0	Blog
novel-cloud	1,170	6,154	0 / 1	Content Management Systems
opsli-boot	1,134	40,177	0 / 0	Development Platform
SpringBootBlog	776	3,582	0 / 0	Blog
itranswarp	774	10,222	0 / 0	Content Management Systems
abixen-platform	679	17852	0 / 0	Development Platform
newbee-mall-plus	588	7,482	5 / 6	E-commerce
my-shop	433	31,772	9 / 19	E-commerce
zscat_sw	294	82,200	0 / 0	Content Management Systems
tesco-mall	147	34,309	14 / 23	E-commerce
shop-mall	124	18,985	0 / 1	E-commerce
Total	/	777,971	73 / 116	/
Industrial Applications	**# Stars**	**# LoCs**	**# Vulns**	**# Description**
r***	/	14,382	3	Development Platform
s***	/	207,327	29	Development Platform
h***	/	262,732	1	Development Platform
u***	/	28,610	2	Backend Management System
d***	/	21,043	1	Backend Management System
p***	/	24,077	2	Backend Management System
d***	/	37,807	7	Backend Management System
Total	/	595,976	45	/

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper proposes `MOCGuard`, an approach to detect missing owner checks in Java applications. The authors rely on the central understanding that databases are a reasonable indicator of data ownership, e.g., when a table has a reference to an element in the user table or when the code flow involves taking user IDs from a user-owned table and querying another. MOCGuard leverages this by identifying user-related tables based on keywords and following data dependencies both explicitly through the CREATE TABLE statements and implicitly by analyzing source code. In doing so, the authors find several vulnerabilities in popular open-source projects.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Identifies an Impactful Vulnerability.
- Provides a Valuable Step Forward in an Established Field.

B.3. Reasons for Acceptance

1) Database-centric approach provides an interesting vector for finding vulnerabilities.
2) This paper improves the state-of-the-art by introducing a new database-centric approach to detect MOC vulnerabilities in Java web applications.
3) Successfully identifies many zero-day vulnerabilities that have been confirmed and assigned CVEs.
4) Authors plan to open-source tool for reproducibility and future science.

B.4. Noteworthy Concerns

1) Ground truth evaluation: rather than looking at an actual real-world ground truth dataset of known vulnerabilities, the paper chooses to take the union of its findings and of the tool to which it is compared.
2) The comparative analysis against MACE could be affected by implementation oversights in the custom port to Java applications.
3) A noteworthy concern regarding the definition of MOC (Definition 5): This definition flags any access request from a user to another user's data as a vulnerability. However, the definition does not account for permissions and user capabilities. On the other hand, it is hard to distinguish between the need for MOC and access control checks.