Detecting Broken Object-Level Authorization Vulnerabilities in Database-Backed Applications

Yongheng Huang
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China
huangyongheng20s@ict.ac.cn

Chenghang Shi
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China
shichenghang21s@ict.ac.cn

Jie Lu*
SKLP, Institute of Computing Technology, CAS
Beijing, China
lujie@ict.ac.cn

Haofeng Li
SKLP, Institute of Computing Technology, CAS
Beijing, China
lihaofeng@ict.ac.cn

Haining Meng
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China
menghaining@ict.ac.cn

Lian Li*
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Zhongguancun Laboratory
Beijing, China
lianli@ict.ac.cn

Abstract

Broken object-level authorization (BOLA) vulnerabilities are among the most critical security risks facing database-backed applications. However, there is still a significant gap in our systematic understanding of these vulnerabilities. To bridge this gap, we conducted an in-depth study of 101 real-world BOLA vulnerabilities from open-source applications. Our study revealed the four most common object-level authorization models in database-backed application.

The insights gained from our study inspired the development of a new tool called BolaRay. This tool employs a combination of SQL and static analysis to automatically infer the distinct types of object-level authorization models, and subsequently verify whether existing implementations enforce appropriate checks for these models. We evaluated BolaRay using 25 popular database-backed applications, which led to the identification of **193** true vulnerabilities, including **178** vulnerabilities that have never been reported before, at a false positive rate of 21.86%. We reported all newly identified vulnerabilities to the corresponding maintainers. To date, **155** vulnerabilities have been confirmed, with **52** CVE IDs granted.

CCS Concepts

• **Security and privacy → Software and application security**.

Keywords

Broken Object-Level Authorization; Database-Backed Applications

*Corresponding author

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690227

ACM Reference Format:
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 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690227

1 Introduction

This (BOLA) has been the most common and impactful attack on APIs.

— The Open Web Application Security Project (OWASP) [33]

Database-backed applications utilize a database to manage data, often accompanied by a front-end to interact with the user and perform database operations upon the user's requests. Such applications are widely adopted across various industries, including content management systems, e-commerce websites, and hospital management platforms. However, given the vast amounts of sensitive data managed by these applications, they have also become prime targets for cybersecurity attacks.

A variety of vulnerabilities, such as SQL injection [35] and XSS [36], can be exploited to compromise database-backed applications and steal sensitive data. Among them, *broken object-level authorization* (BOLA) vulnerabilities, also known as *insecure direct object reference* (IDOR) [4], have gained the top position in the OWASP API Top 10 rankings [34] due to their common occurrence and high risk. Notably, many widely-used applications, such as PayPal, Twitter, and TikTok, have suffered from BOLA vulnerabilities, as reported in HackerOne [39].

Figure 1 illustrates CVE-2022-31295, a BOLA vulnerability in the application Odfs-1.0 (*Online Discussion Forum Site*). As the name suggests, this application allows users to create posts which are stored in the `post` table—a database table utilizing `id` as its primary

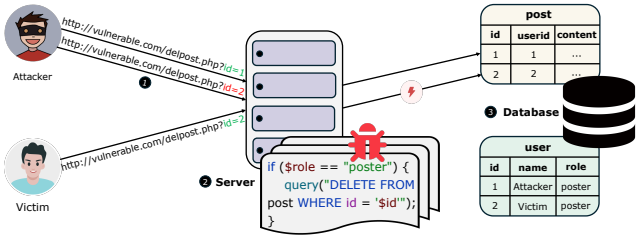

Figure 1: The BOLA vulnerability CVE-2022-31295

key. The URL "`http://vulnerable.com?delpost.php?id=1`" requests the deletion of a post whose `id` equals 1.

The authorization model of this application should be fine-grained at the *object level*: only the creator of a post should have the necessary permission to delete it. However, due to the lack of object-level authorization checks, an attacker can delete other users' posts at will. Let us analyze such an attack. ❶ The attacker attempts to delete a post belonging to the victim by tampering with the `id` from 1 to 2. ❷ The modified request is sent to the application server, which processes it into a SQL statement. ❸ The database executes the SQL statement to perform the deletion operation, albeit undesirably. In this example, it is noteworthy that despite the server's validation of the attacker's role as a `poster` with DELETE privileges, it fails to further verify whether the attacker is indeed the creator of the post to be deleted.

1.1 Challenges

To detect BOLA vulnerabilities, it is vital to understand the underlying *object-level authorization model*, which determines the appropriate policy for accessing an object. For instance, Figure 1 illustrates that a `post` object can only be deleted by its creator. However, it is challenging, if not impossible, to precisely infer object-level authorization models and efficiently detect BOLA vulnerabilities that violate those models. Here, we summarize two main challenges:

Challenge 1: How can we precisely and automatically infer object-level authorization models? Although there have been efforts to automatically infer coarse-grained, function-level [32] authorization models [15, 41, 43, 44, 53], object-level authorization models are much more intricate and require finer-grained, application-specific semantics. This requirement poses a significant challenge to automating this process. As a result, existing approaches are incapable of deriving complex object-level authorization models and often rely on the manual annotation of each object operation with authorization rules, which is tedious and error-prone [13].

Challenge 2: How can we efficiently and precisely detect BOLA vulnerabilities? To detect BOLA vulnerabilities, we need to accurately compute whether each object access is checked properly against its authorization models or not. This often demands expensive path-sensitive analyses, such as model checking and symbolic execution [7, 9, 12–14, 28, 30], which are difficult to scale to real-world applications.

1.2 Solutions

To tackle these two challenges, it is imperative to gain a deep understanding of the real-world BOLA vulnerabilities. For this purpose, this paper constructs a comprehensive dataset consisting of 101 BOLA vulnerabilities from the CVE database [1] and the bug bounty platform Huntr [48]. To the best of our knowledge, this is the first in-depth study of BOLA vulnerabilities. During our study, we have obtained two interesting findings, which can respectively help to address the two challenges mentioned above.

Automatically inferring authorization models. We observe that there are four distinct kinds of object-level authorization models in real-world applications. Among these models, only the ownership model has been studied before [28, 30], with the other three models remaining unexplored in the literature. Furthermore, we observe that all object-level authorization models can be derived from relations across database tables. For instance, in Figure 1, the column `userid` of the `post` table is the foreign key referencing the primary key (Column `id`) of the `user` table.

In light of this, we propose to infer object-level authorization models by reasoning about relations between different database tables. In simple cases, relations between distinct database tables are directly declared as *foreign keys* in the database schema. However, such relations are often not explicitly specified in the schema, but are instead implicitly implemented in the source code. Hence, to address this challenge, we have designed a set of rigorous rules to deduce implicit foreign key references by examining the complex interaction between program code and database queries.

Efficiently detecting BOLA vulnerabilities. We observe that all studied BOLA vulnerabilities are due to *missing object-level authorization checks*. Thus, instead of analyzing whether the authorization model for each object access is consistently enforced or not (which often requires extensive constraint solving of path conditions), we focus on the common cases where accesses to sensitive objects lack object-level authorizations. For instance, in Figure 1, an access to object `post` needs to be checked against both the object itself and its creator, `user`. This approximation leads to an efficient yet precise approach to hunt BOLA vulnerabilities, which addresses challenge 2.

It is crucial to understand that *missing object-level authorization checks* not only includes simple cases where permission checks are entirely missing, but also encompasses cases with inconsistent or incomplete checks that fail to verify the correct corresponding object. For instance, as shown in Figure 1, although role permissions are checked, object-level authorization is missing. Additionally, object-level authorization checks may involve multiple sub-checks. Any missing sub-checks leads to incomplete authorization, which is also considered a lack of object-level authorization checks. We will further elaborate on the details in Section 3.3.

Putting it all together, we have devised an efficient yet precise static approach for detecting BOLA vulnerabilities in database-backed applications.

1.3 Contributions

We realize our approach in a new tool named BolaRay and evaluate it on 25 popular database-backed applications. BolaRay precisely infers object-level authorization models in all evaluated applications, and reports 193 vulnerabilities, including 178 new vulnerabilities that have never been found before, with only 54 false positives. To date, 155 newly reported vulnerabilities have been confirmed and 52 of them have been assigned CVE IDs.

The contributions of this paper are summarized as follows:

- We provide the first in-depth study of BOLA vulnerabilities in real-world database-backed applications. Our study sheds light on new detection techniques for BOLA vulnerabilities within such applications.
- We introduce a novel static analysis approach to identify BOLA vulnerabilities in database-backed applications. Our analysis efficiently uncovers object-level authorization models with high accuracy by statically analyzing the relationships between database tables.
- We have implemented our approach as a tool dubbed BolaRay, and evaluated it using 25 real-world applications. BolaRay accurately reported 193 vulnerabilities, including 178 new critical BOLA vulnerabilities. Out of these, 155 have been confirmed, and 52 CVE IDs have been granted.
- To facilitate future research, we have released the source code of BolaRay, together with all studied vulnerabilities, at https://github.com/BolaRay-d/BolaRay.

2 An Illustration Example

Figure 2 and Figure 3 depict an example of a content management application that is used throughout this paper. Figure 2 presents the 7 tables in this application, where each table uses `id` as its primary key, and implicit foreign key references are highlighted in yellow. For instance, `forumid` is an implicit foreign key referencing `forum::id` – the primary key `id` of table `forum`. Recall that those implicit foreign key references need to be derived from the source code. Relationships between distinct tables, derived from foreign key references, are connected by arrows.

Among the 7 tables, the `user` table manages registered user accounts and the `profile` table stores the profile for each user. The two tables share the identical primary key, meaning the profile of a user can be queried from the `profile` table using their key. The table `forum` manages forums, and a user can participate in none or many forums, as indicated in the `user_forum` table. Each user is granted the role `"manager"` or `"poster"` in their participating forums, where a manager can delete forum notices and a poster can create new posts or manage their own posts. Forum notices and posts are stored in the `notice` table and `post` table, respectively. Finally, the `comment` table stores comments and a user can comment on posts whose status is `open`.

Figure 3 illustrates four BOLA vulnerabilities, each with distinct root causes, manifesting in different APIs.

- The API `close_post` (Figure 3(a)) allows users to set the status of a given post to `Close`. A vulnerability arises because this API does not check whether the requester is the owner of the target post. Consequently, an attacker can close any post at will. The fix (line 10) involves adding an object-level authorization check in the SQL statement to confirm ownership of the target post.
- The API `update_forum` (Figure 3(b)) allows a forum manager to update the `topic` of a forum. The check at line 2 ensures that the requester holds the `manager` role. However, despite performing this role check, the API does not validate whether the requester is a member of the target forum. This oversight leads to a BOLA vulnerability, enabling a manager to update the topic of any forum. This vulnerability is addressed by implementing an object-level authorization check in lines 3-9, to verify that the requester is a member of the target forum.
- In Figure 3(c), the API `delete_notice` deletes a notice of a forum as requested by the forum manager. A vulnerability manifests if the API does not check whether the requester is a member of the forum to which the deleted notice belongs. This authorization rule is enforced through lines 4-9 in Figure 3(c) and lines 1-9 in Figure 3(b), which verifies the relationship between the target notice and its parent forum, and the membership between the requester and the forum, respectively.
- In Figure 3(d), the API `add_comment` enables users to comment on posts. The vulnerability stems from the API's failure to verify whether the post is in an `open` status. This issue is rectified in lines 4-9 of Figure 3(d).

Each vulnerability in Figure 3 necessitates a distinct object-level authorization check. These checks verify various relations between the accessing object and the requester, or between related objects, each corresponding to a distinct authorization model. To detect such vulnerabilities, we need to precisely analyze the fine-grained authorization model for each object access and then verify whether the authorization model has been properly implemented.

3 Empirical Study

The USPS hack is a classic example of a broken authorization vulnerability. User A was able to authenticate to the API and then pivot and access user B's and 60 million other people's information.

— Dan Barahona, Head of Marketing at Biz Dev at APIsec [6]

In this section, we first present the vulnerability collection process for our empirical study, then conduct a comprehensive study on 101 BOLA vulnerabilities. This study aims to answer the following questions: what object-level authorization models are present in real-world applications, and what are the root causes of BOLA vulnerabilities?

3.1 Vulnerability Collection and Analysis

We conducted an empirical study on BOLA vulnerabilities, analyzing data from the CVE database [1] and Huntr platform [48] following previous vulnerability studies [47, 52]. After filtering for open-source applications with valid patches, we identified 101 BOLA vulnerabilities for in-depth analysis.Three authors independently examined each vulnerability, annotating the authorization model, root cause, and fix.Any disagreements were resolved through discussions involving a fourth author. This process was completed over two months. For a detailed methodology and discussion of study limitations, please refer to our appendix[1].

3.2 Authorization Models

***Finding 1*:** There are four types of object-level authorization models, all of which can be derived by reasoning about relationships between tables from implicit foreign key references.

[1] https://github.com/BolaRay-d/BolaRay/blob/main/appendix-study.pdf

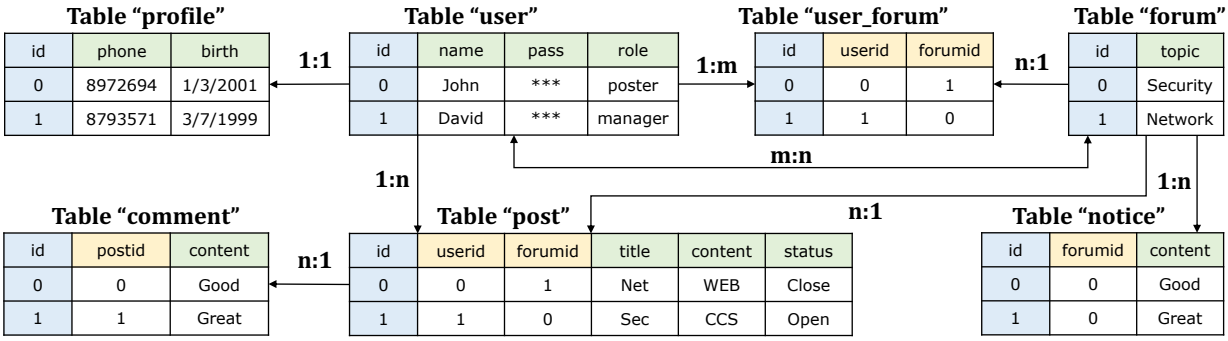

Figure 2: Tables and and their relationships in a content management system.

```
1  $pid = escape($_POST["pid"]);
2  ......
3  $curr_user = $_SESSION["uid"];
4  $role = getRole($curr_user, ......);
5  if ($role != "poster") {
6    die("Not poster");
7  } else {
8    query("UPDATE post SET status
9      = "Close" WHERE id = '$pid'
10 + AND userid = '$curr_user'
11   ");
12 }
```

(a) close_post.php

```
1  $fid = escape($_POST["fid"]);
2  if ($role != "manager") {......}
3  + $row = query("SELECT userid
4  + FROM user_forum WHERE
5  + forumid = '$fid'");
6  + if (!in_array($curr_user,
7  + $row)) {
8  +   die("Not member in forum");
9  + }
10 ...
11 query("UPDATE forum SET topic
12 = ..... WHERE id = '$fid'");
```

(b) update_forum.php

```
1  // same as Lines 1-9 in (b)
2  $nid = escape($_POST["nid"]);
3  ......
4  + $row2 = query("SELECT
5  + forumid FROM notice
6  + WHERE id = '$nid'");
7  + if ($row2[0] != $fid) {
8  +   die("Notice not found");
9  + }
10 ......
11 query("DELETE FROM notice
12 WHERE id = '$nid'");
```

(c) delete_notice.php

```
1  $pid = escape($_POST["pid"]);
2  $cont = escape($_POST["cont"]);
3  ......
4  + $row = query("SELECT status
5  + FROM post WHERE id = '$pid'");
7  + if ($row[0] == "Close") {
8  +   die("Post closed");
9  + }
10 ......
11 query("INSERT INTO comment
12 (postid, content) VALUES
13 ('$pid', '$cont')");
```

(d) add_comment.php

Figure 3: Four examples of BOLA vulnerabilities from Figure 2. Lines 1-9 of the code snippet in (b) has been inlined in the code snippet in (c).

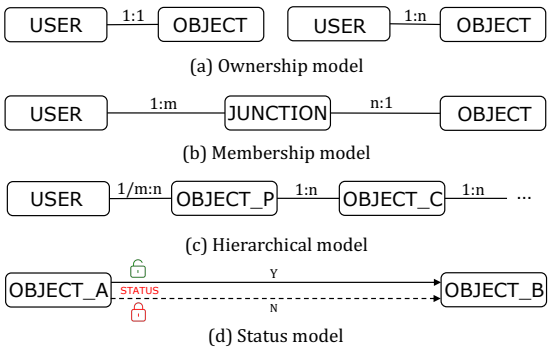

Figure 4: Four different object-level authorization models.

Figure 4 illustrates the four authorization models in database-backed applications. The first three models depict distinct relationships between a user and an object. The last model, named *the status model*, relates an object to the status of another object, suggesting the necessity to recognize the status column in detecting violations of this specific model.

3.2.1 Ownership model. The ownership model is perhaps the most studied authorization model, wherein a user directly owns an object as characterized by a direct one-to-one (1:1) or one-to-many (1:n) relationship between the user and the object, or more specifically, between a user table and an object table. Under this model, an object can only be manipulated by its owner. For instance, in Figure 2, a user owns their personal profile (1:1) and multiple posts created by them (1:n). Consequently, a post can only be deleted by its owner, as exemplified in Figure 3(a).

3.2.2 Membership model. In this model, the relationship between users and objects is modeled as many-to-many (m:n), indicating that (1) an object can be accessed by a specific group of users, and (2) a user can have access to multiple objects. The relationship between a `user` and a `forum` in Figure 2 is such an example: a user can participate in multiple forums, and conversely, a forum can accommodate multiple users.

To capture this membership relationship, a *junction table* (such as `user_forum` in Figure 2) is often introduced to join the user table and the object table together, establishing the connections between users and objects.

3.2.3 Hierarchical model. This model is a combination of an ownership or membership model with one or multiple parent-child relationships between objects, where a parent object can own multiple child objects. For instance, in Figure 2, a user owns their post, and a post owns all comments on itself. Thus, the owner of a post also indirectly owns all comments on that post. Consequently, only the owner of the post can manipulate comments on it. Another example involves the `user` table, the `forum` table, and the `notice` table: a user can manage forum notices (child objects of a forum) only if he or she is a member of that forum.

3.2.4 Status model. In this model, objects possess statuses, and users are only permitted to execute actions on objects when they are in specific states. For instance, in Figure 2, after a post is closed by

Table 1: The root causes and fix strategies.

Root causes	Code fix	SQL fix	Total
Missing Ownership Check	38	25	63(62.37%)
Missing Membership Check	7	0	7(6.93%)
Missing Hierarchical Check	9	0	9(8.91%)
Missing Status Check	22	0	22(21.78%)
Total	76 (75.24%)	25 (24.75%)	101

its owner, other users are barred from commenting on it. Another common scenario involves items becoming unavailable for purchase once they have been removed from the shelves.

3.3 Root Causes and Fixes

***Finding 2*:** Each authorization model is accompanied by its own set of authorization rules. Violating any of these rules leads to a BOLA vulnerability. In our study, all BOLA vulnerabilities stem from *missing object-level authorization checks*, which can be addressed by adding checks in the source code (75.24%) or in SQL statements (24.75%).

As shown in Table 1, all BOLA vulnerabilities are caused by missing distinct types of object-level authorization checks:

- *Missing ownership checks* arises because an application fails to verify whether the current user is indeed the owner of the object being accessed. Figure 3(a) is an example where the ownership of the target post is not properly checked.
- *Missing membership checks* is caused by the lack of check on whether a requester belongs to the group authorized to operate on a specific object. The vulnerability in Figure 3(b) is such an example, due to the fact that the application fails to confirm if the requester is a member of the target forum.
- *Missing hierarchical checks* can be triggered by missing ownership or membership checks, or by the lack of a check against the parent-child relationship between distinct objects. The hierarchical authorization model is often realized through multiple sub-checks which respectively verify ownership or membership of corresponding objects and the relationship between parent objects and child objects. Any missing sub-checks can lead to incomplete authorization. For instance, in Figure 3(c), in addition to verifying the membership between the requester and the forum, the application needs to further ensure that the target notice is a child of the target forum. The two vulnerabilities, CVE-2021-4194 and Huntr-e6144554, are triggered in the same fashion, i.e., missing checks for parent-child relationships. Hence, to avoid such vulnerabilities, we need to ensure that all required sub-checks have been enforced.
- *Missing status checks* occurs because the application does not verify the current status of an object before performing specific operations. This issue is exemplified in Figure 3(d), where, despite the target post being closed, the oversight in checking the target post's status enables attackers to leave comments undesirably. It is important to note that multiple status checks may be required to safely perform an operation. In our study, four vulnerabilities–CVE-2022-0170, CVE-2022-0574, CVE-2022-0726, and CVE-2022-0727–are caused by checking only one status variable while ignoring others. For instance, in CVE-2022-0574, the flawed application publify – a publishing platform – only verifies that the post is `commentable` when adding comments on it, without checking whether the post is in `draft` state or not. To precisely detect such vulnerabilities, it is necessary to identify all required status checks for a specific operation, which is quite challenging. A cautious approach is to require all statuses of an object to be checked, which ensures safety but may result in false positives. However, as showcased in Section 5, we did not encouter such false positives in our experiments.

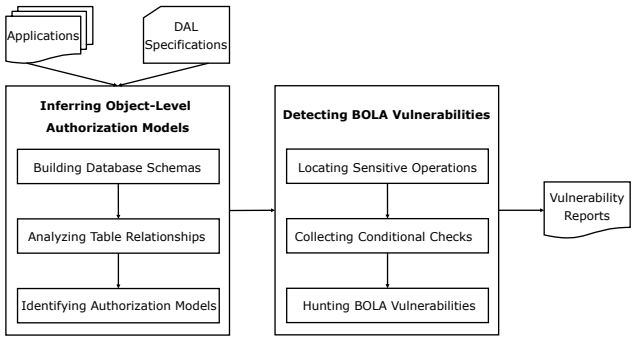

Figure 5: Overview of BolaRay.

All studied vulnerabilities were fixed by adding extra object-level authorization checks: 75.24% of fixes introduced extra checks in the source code, and 24.75% of vulnerabilities were addressed by introducing extra checks in the `WHERE` clauses of corresponding SQL statements. In code-based fixes, data stored in database tables are retrieved into program variables via SQL statements, and object-level authorization is performed by checking those variables in conditional statements. Such a fixing strategy is suitable for complex authorization models, which often involve multiple sub-checks, including the hierarchical model (Figure 3(c)) and the status model (Figure 3(d)). On the other hand, SQL-based fixes directly patch existing SQL queries with additional predicates in their `WHERE` clauses. This approach is often favored in relatively simple authorization models, such as the ownership model (Figure 3(a)). Conversely, to detect such vulnerabilities, we need to identify the authorization model for each object access and further verify whether the object-level authorization checks are properly implemented in both the source code and SQL statements.

4 BolaRay

It is vital that we check all objects and that we check them for read, update and delete actions. We need to check every functionality that has access to these objects.

— Stepan Ilyin, Verified expert from wallarm [17]

In light of our empirical study, we propose BolaRay, a new BOLA vulnerability detection tool. As depicted in Figure 5, the tool comprises two primary modules. The first module automatically infers object-level authorization models in three steps. Subsequently, the second module detects BOLA vulnerabilities by verifying that the set of checks for an object access enforces the authorization model of that object.

name	columns name & type	primary key	foreign keys	unique keys
user	[(id, int[4]), (name, char[20]), ...]	id	[]	[id]
profile	[(id, int[4]), (phone, int[11]), ...]	id	[(id::user)]	[id]
forum	[(id, int[4], (topic, char[20]), ...]	id	[]	[id]
...	...	...	...	...

Figure 6: The schema of our example in Figure 2.

4.1 Inferring Object-Level Authorization Models

Guided by Finding 1, we automatically infer object-level authorization models in three steps: the first step constructs a database schema from table creation statements; the second, also the key step, derives table relationships by analyzing implicit foreign key references; and the third step infers authorization models from these table relationships.

4.1.1 Building Database Schemas. Given a database-backed application, we build the database schema by considering all table creation statements in this application. Table creation statements specify the name, columns, and keys of each table. These statements may be declared in SQL scripts (`.sql` files) or incorporated into source code, either as direct query strings or through database manipulation APIs provided by the underlying Data Access Layer (DAL) framework. Section 4.3 will provide a detailed explanation of how table creation statements and other SQL statements are handled.

The schema summarizes all tables managed by the database. Figure 6 shows the schema of our example in Figure 2. For simplicity, not all tables are given. Each table has a unique name and consists of a set of columns in the form of <name, type>. The primary key is the column uniquely indexing the table, foreign keys declare those columns that refer to the primary keys of other tables, and unique keys are those columns that can only contain unique values.

Relationships explicitly declared as foreign keys are directly encoded in the schema. For instance, in Figure 6, the primary key of table `profile` is also a foreign key referring to the table `user`, reflecting the 1-to-1 mapping between a user and their profile. However, the foreign keys for other tables are set to empty, and their relationships need to be inferred in the next step.

4.1.2 Analyzing Table Relationships. The crux of deducing table relationships lies in precisely identifying implicit foreign keys, which are columns that refer to the primary keys of other tables. Due to various reasons [18], these foreign keys often are not declared in the schema but instead implemented in the source code. Without loss of generality, we hereafter assume that a foreign key contains only one column. Our formulation can be easily extended to support cases with multiple columns constituting a foreign key.

Definition 1. *(Foreign key). A foreign key takes the form of* $(t_1, t_2 :: c2)$*, where column* c_2 *of table* t_2 *refers to the primary key of table* t_1*. The notation* $t :: c$ *is used to denote column* c *of table* t*, and it is simply written as* c *if the table* t *is understood.*

Identifying implicit foreign keys can be challenging since such information lies in the complex interaction between source code and database operations. For instance, if a variable holding a primary key value of table t_1 is used in a `WHERE` clause of a subsequent query

Program	p	::=	$\overline{T}; \overline{s}; \overline{v}$
Tables	T	::=	$t\ \{\overline{c}; c_p; \overline{(c_f : t)}; \overline{c_u}\}$
Statements	s	::=	$l^C : \langle k, t, c, (c, v)\rangle \mid l^C : v \leftarrow \langle k, t, c, (c, v)\rangle$
Keywords	k	::=	SELECT | DELETE | INSERT | UPDATE
Variables	v	::=	x
Checks	C	::=	$\overline{t :: c \mid (t :: c, t :: c)}$
Identifiers	t, c, x		
Locations	l		

Figure 7: Domains used in formalism.

$$\frac{l^C : \langle -, t, -, (c, v)\rangle}{(l, t :: c, v) \in \textbf{Binding}} \quad \text{[Key-value]}$$

$$\frac{l^C : v \leftarrow \langle \text{SELECT}, t, c, -\rangle}{(l, t :: c, v) \in \textbf{Binding}} \quad \text{[Select]}$$

$$\frac{(l_1, t_1 :: c_p, v_1) \in \textbf{Binding} \quad (l_2, t_2 :: c_2, v_2) \in \textbf{Binding} \quad v_1 \text{ aliases to } v_2 \quad l_1 \text{ dominates } l_2}{(t_1, t_2 :: c_2) \in \textbf{ForeignKey}} \quad \text{[Connect]}$$

Figure 8: Rules for foreign key analysis.

on column c_2 of table t_2, it is a strong indication of the foreign key $(t_1, t_2 :: c_2)$. To address this, we design a set of rigorous rules, which will be discussed shortly, to derive the set of foreign keys.

For illustrative purposes, Figure 7 presents the domain used in our formalism. A program p consists of a set of tables T, a set of statements s, and a set of program variables v. A table t consists of a set of columns $\overline{c}$, where $c_p \in \overline{c}$ is the column for primary key; $\overline{(c_f : t)}$ is the set of schema-declared foreign keys, where column $c_f \in \overline{c}$ refers to the primary key of another table; $\overline{c_u}$ denotes the set of unique keys $c_u \in \overline{c}$.

A SQL statement s takes the form of $l^C : \langle k, t, c, (c', v')\rangle$ or $l^C : v \leftarrow \langle k, t, c, (c', v')\rangle$, denoting the statement at location l applies a query operation k to table t with c as the target column (only when k is a `SELECT` operation), and v holding the resulting value. Note that location l is guarded by a set of checks C (to be computed in Section 4.3), where each check either validates a column $t :: c$ or compares two columns $(t :: c, t :: c)$. Finally, the pair (c', v') relates column c' to variable v', as illustrated in the following cases:

- Key-value pairs in a `WHERE` clause: "`... WHERE` c' `=` v'", or "`... WHERE` c' `IN` v'".
- Key-value pairs in an `UPDATE` operation: "`UPDATE ... SET` c' `=` v' `...`".
- Key-value pairs in an `INSERT` operation: "`INSERT INTO ...` (`c'`, `...`) `VALUES` (`v'`, `...`)".

A SQL query containing multiple such pairs is normalized into multiple statements, one for each pair. For example, a SQL statement l: "`UPDATE comment SET pid = $p WHERE status = $s`" is represented by two statements: l : ⟨ `UPDATE, comment, -, (pid, $p)` ⟩ and l : ⟨ `UPDATE, comment, -, (status, $s)` ⟩.

For clarity, Figure 7 considers only database query statements. Those statements concerning data and control flows are processed in a separate analysis, to compute data and control dependencies, as detailed in Section 4.3. Prior to exploring the specific rules for determining foreign keys, let us first introduce the two following sets:

Step 1:

$$\frac{(t_1, t_2 :: c_2) \in \textbf{ForeignKey} \quad (t_3, t_2 :: c_2') \in \textbf{ForeignKey} \quad (t_1, t_3 :: -) \notin \textbf{ForeignKey} \quad (t_3, t_1 :: -) \notin \textbf{ForeignKey} \quad (t_2, - :: -) \notin \textbf{ForeignKey}}{(t_1, t_2 :: c_2, t_3, t_2 :: c_2') \in \textbf{MNR} \quad t_2 \in \textbf{JunctionTable}} \quad \text{[M-N-Rel]}$$

$$\frac{(t_1, t2 :: c_2, t_3, t_2 :: c_2') \in \textbf{MNR} \quad (t_3, t_4 :: c_4, t_5, t_4 :: c_4') \in \textbf{MNR}}{(t_1, t_2 :: c_2, t_5, t_4 :: c_4') \in \textbf{MNR}} \quad \text{[M-N-Rel-R]}$$

Step 2:

$$\frac{(t_1, t_2 :: c_u) \in \textbf{ForeignKey} \quad t_2 \notin \textbf{JunctionTable}}{(t_1, t_2 :: c_u) \in \textbf{OOR}} \quad \text{[1-1-Rel]}$$

$$\frac{(t_1, t_2 :: c_2) \in \textbf{ForeignKey} \quad c_2 \notin \overline{t_2 :: c_u} \quad t_2 \notin \textbf{JunctionTable}}{(t_1, t_2 :: c_2) \in \textbf{ONR}} \quad \text{[1-N-Rel]}$$

Figure 9: Rules for table relation analysis.

- **Binding**, a set of triples in the form of $(l, t :: c, v)$, which means that program variable v may hold values from column $t :: c$ due to the statement at location l.
- **ForeignKey**, a set of foreign keys (Definition 1) in the form $(t_1, t_2 :: c_2)$.

Figure 8 outlines the rules for deducing foreign keys. Specifically, $(l, t :: c, v) \in$ **Binding** if at least one of the following conditions holds true: (1) there exists a SQL statement $l^C : \langle -, t, -, (c, v)\rangle$, where the pair (c, v) relates column c to variable v ([Key-value]), including three cases as discussed above. (2) v receives results from column c (of table t) via `SELECT` statements ([Select]).

In [Connect], $(t_1, t_2 :: c_2)$ is regarded as a foreign key if the binding variable v_1 of $t_1 :: c_p$ (i.e., $(l_1, t_1 :: c_p, v_1) \in$ **Binding**) aliases to the binding variable v_2 of $t_2 :: c_2$ (i.e., $(l_2, t_2 :: c_2, v_2) \in$ **Binding**), indicating that $t2 :: c2$ refers to the primary key of t_1. To filter false foreign keys, inspired by [8], we also require that the two involving statements need to be executed together. This requirement is approximated by the *dominance relationship* [3] between the two statements, i.e., l_1 dominates l_2.

Next, Figure 9 computes table relationships based on deduced foreign keys. We further define four sets, **OOR**, **ONR**, **MNR** and **JunctionTable**, as follows:

- **OOR** (**ONR**), a set of pairs in the form of $(t_1, t_2 :: c_2)$, which means that there is a 1:1 (1:n) relationship between t_1 and t_2, and c_2 is a foreign key referring to t_1.
- **JunctionTable**, a set of junction tables used to join two tables together, as utilized in a m:n relationship.
- **MNR**, a set of tuples in the form of $(t_1, t_2 :: c_2, t_3, t_2' :: c_2')$, denoting the m:n relationship between tables t_1 and t_3: the two tables are joined through one or more junction tables, $t_2, \ldots, t_2'$, with $t_2 :: c_2$ and $t_2' :: c_2'$ being the foreign keys referencing t_1 and t_3, respectively. In the simple two-table join case, t_2 and t_2' are the same.

The rules in Figure 9 are processed through two distinct steps. Initially, the rules [M-N-Rel] and [M-N-Rel-R] are exclusively applied to deduce m:n relationships and to calculate junction tables. Subsequently, in the second step, these junction tables facilitate the application of the two rules [1-1-Rel] and [1-N-Rel].

In [M-N-Rel], the m:n relationship between t_1 and t_3 is concluded only if the following conditions are met: 1) t_1 and t_3 are not directly related, meaning $(t_1, t_3 :: -) \notin$ **ForeignKey** and $(t_3, t_1 :: -) \notin$ **ForeignKey**; and 2) t_2 is introduced solely to join t_1 and t_3 together, indicated by $(t_2, - :: -) \notin$ **ForeignKey**. In our example in Figure 2, similar to the junction table `user_forum`, the table `post` also contains two foreign keys `userid:user` and `forumid:forum`. However, this table is not considered a junction table since it is also referenced by the foreign key `comment:postid`. The rule [M-N-Rel-R] recursively deduce m:n relationships to handle multiple-table joins.

The rules for 1:1 ([1-1-Rel]) and 1:n ([1-N-Rel]) relationships are self-explanatory. They are distinguished based on whether the foreign key $t_2 :: c_2$ is unique. It is important to recall that a junction table is solely introduced to join two other tables and it does not store actual object; thus, all junction tables are *excluded* from consideration (indicated by $t_2 \notin$ **JunctionTable**) when deriving 1:1 and 1:n relationships.

4.1.3 Identifying Authorization Models. Figure 10 outlines the rules for inferring authorization models. Before diving into its details, we first introduce four additional sets **UserTable**, **OwnerModel**, **MemberModel**, and **StatusModel**, as explained below:

- **UserTable**, a set of user tables.
- **OwnerModel**, a set of tuples in the form of $(t_1, t_2 :: c_2)$, signifying a 1:1 or 1:n relationship between t_1 and t_2 (**OOR** or **ONR**), which forms the foundation of an ownership model.
- **MemberModel**, a set of tuples in the form of $(t_1, t_2 :: c_2, t_3, t_2' :: c_2')$, signifying a m:n relationship (**MNR**) in the same form.
- **StatusModel**, a set of triples like $(t_1, t_2 :: c_2, t_1 :: c_1')$, denoting a 1:1 or 1:n relationship between t_1 and t_2, and the column $t_1 :: c_1'$ signifies a status value.

For simplicity, we have not introduced sets for hierarchical models, which by construction, can be computed by joining the set **OwnerModel** or **MemberModel** with **OOR** and **ONR**.For detailed information on hierarchical models, please refer to our appendix[2].

The first two rules in Figure 10 are used to compute user tables. As a common practice, the current user ID is stored in the global session for future authorization purposes. Consider the following code snippet as an example: after a user logs in, the user ID is stored in the global variable `$_SESSION["uid"]` (line 4), which is then accessible from anywhere in the source code.

```
1 $row = query("SELECT id FROM user WHERE name = '$name'
  AND pass = '$pass'");
2 if ($row == null)
3     die("Invalid user");
4 $_SESSION["uid"] = $row[0];
```

In light of this heuristic, the rule [User-Table] constructs an initial set of user tables in **UserTable**. We further extend the set **UserTable** with rule [User-Ext], which also regards those tables as user tables if they have a 1:1 relationship with an existing user table. As a result, in our example in Figure 2, the `profile` table is also considered as a user table.

The next two rules, [Own-Model] and [Mem-Model], infer the ownership model and membership model, respectively, which are self-explanatory. The rule [Stat-Model] relies on the recognition of status columns. We consider that column $t :: c$ is a status column if all the following conditions are met:

[2]https://github.com/BolaRay-d/BolaRay/blob/main/appendix-hm.pdf

$$\frac{(-, t :: c, v) \in \textbf{Binding} \quad v \text{ aliases to } v' \quad v' \text{ is stored in the global session}}{t \in \textbf{UserTable}} \quad \text{[User-Table]}$$

$$\frac{t_1 \in \textbf{UserTable} \quad (t_1, t_2 :: -) \in \textbf{OOR}}{t_2 \in \textbf{UserTable}} \quad \text{[User-Ext]}$$

$$\frac{t_1 \in \textbf{UserTable} \quad (t_1, t_2 :: c_2) \in (\textbf{OOR} \cup \textbf{ONR})}{(t_1, t_2 :: c_2) \in \textbf{OwnerModel}} \quad \text{[Own-Model]}$$

$$\frac{t_1 \in \textbf{UserTable} \quad (t_1, t_2 :: c_2, t_3, t_2' :: c_2') \in \textbf{MNR}}{(t_1, t_2 :: c_2, t_3, t_2' :: c_2') \in \textbf{MemberModel}} \quad \text{[Mem-Model]}$$

$$\frac{(t_1, t_2 :: c_2) \in (\textbf{OOR} \cup \textbf{ONR}) \quad t_1 :: c_1' \text{ is a status column}}{(t_1, t_2 :: c_2, t_1 :: c_1') \in \textbf{StatusModel}} \quad \text{[Stat-Model]}$$

Figure 10: Rules for authorization model inference.

(1) c holds a type with a small range of values, such as `BOOLEAN`, `TINYINT(1)`, or `ENUM`. This condition ensures that the number of states represented by c is finite and enumerable. For example, the `status` column of the table `post` in Figure 2 can only have two values: `Open` or `Close`.
(2) c is modifiable, which ensures that c can be dynamically changed by the application.
(3) c is used to control the visibility or behavior of data presented to the user by the frontend, as shown in line 2 in the code snippet below. Note that this condition is specific to web applications only.

```
1 $row = query("SELECT * FROM post WHERE id = '$pid' AND
  forumid = '$fid'");
2 if ($row[status] == "Open") {
3     echo "<h1> $row[title]</h1>";
4     echo "<body> $row[content] </body>";
5     ......
6 }
```

Finally, we want to point out that a hierarchical model involves three (or more) tables t_1, t_2 and t_3, where t_1 and t_2 constitute an ownership or membership model, and t_2 and t_3 form one or multiple 1:1 or 1:n relationships. Thus, we can reuse the two rules [Own-Model] and [Mem-Model] to deduce hierarchical models. For simplicity, these rules are not given in Figure 10.

4.2 Detecting BOLA Vulnerabilities

In this module, we first locate all sensitive database query statements and calculate the set of checks C enforced for each query statement l. Section 4.3 will demonstrate how the two steps are performed in detail. Here we present the rules for detecting BOLA vulnerabilities in Figure 11. For illustrative purposes, we first introduce the following two sets:

- **AdminColumn**, a set of admin columns, which represent administrators that can perform privileged operations.
- **SafeOp**, a set of safe operations.

The **AdminColumn** set is introduced to model privileged users such as *system administrators* of an application. Intuitively, system administrators can perform any privileged operations, irrespective

$$\frac{l^C : \langle \texttt{DELETE}, t_1, -, - \rangle \quad t_1 \in \textbf{UserTable} \quad t_2 :: c_2 \in C}{t_2 :: c_2 \in \textbf{AdminColumn}} \quad \text{[Admin-Col]}$$

$$\frac{l^C : \langle -, t_2, -, - \rangle \quad t_1 :: c_1 \in C \quad t_1 :: c_1 \in \textbf{AdminColumn}}{l^C \in \textbf{SafeOp}} \quad \text{[Admin-Check]}$$

$$\frac{(t_1, t_2 :: c_2) \in \textbf{OwnerModel} \quad l^C : \langle -, t_2, -, - \rangle \quad (t_1 :: c_p, t_2 :: c_2) \in C}{l^C \in \textbf{SafeOp}} \quad \text{[Own-Check]}$$

$$\frac{(t_1, t_2 :: c_2, t_3, t_2' :: c_2') \in \textbf{MemberModel} \quad l^C : \langle -, t_3, -, - \rangle \quad (t_1 :: c_p, t_2 :: c_2) \in C \quad (t_3 :: c_p, t_2' :: c_2') \in C}{l^C \in \textbf{SafeOp}} \quad \text{[Mem-Check]}$$

$$\frac{(t_1, t_2 :: c_2, t_1 :: c_1') \in \textbf{StatusModel} \quad l^C : \langle -, t_2, -, - \rangle \quad t_1 :: c_1' \in C}{l^C \in \textbf{SafeOp}} \quad \text{[Stat-Check]}$$

Figure 11: Rules for checking safety of sensitive operations.

of the four authorization models discussed previously. We have observed that verifying whether the current user is an administrator often involves specific columns, termed *admin columns*. In this context, we initially identify the set of admin columns, **AdminColumn**, by applying the [Admin-Col] rule. Specifically, we start by identifying privileged operations that require administrator permissions, which, in our implementation, are those `DELETE` operations on a user table. Then, for a privileged operation $l^C : \langle \texttt{DELETE}, t_1, -, - \rangle$, where t_1 is a user table, any columns involved in its condition checks (i.e., $t_2 :: c_2 \in C$) are deemed admin columns.

Subsequently, if a sensitive operation l^C is secured by a condition check that involves an admin column (i.e., $t_1 :: c_1 \in C$ and $t_1 :: c_1 \in$ **AdminColumn**), it is deemed safe without further consultation of our authorization models. This is established by the [Admin-Check] rule.

The next three rules, [Own-Check], [Mem-Check] and [Stat-Check], enforce authorization rules for the corresponding ownership, membership, and status models, respectively.

Given $(t_1, t_2 :: c_2) \in$ **OwnerModel**, the [Own-Check] rule ensures that each sensitive operation l^C on table t_2 ($l^C : \langle -, t_2, -, - \rangle$) is guarded by proper ownership checks. Specifically, we determine whether the two columns $t_1 :: c_p$ and $t_2 :: c_2$ are involved in the same condition check for l^C, i.e., $(t_1 :: c_p, t_2 :: c_2) \in C$. For instance, in Figure 3 (a), this is indicated by the condition at Line 10, i.e., `userid = $curr_user`, where `userid` refers to the column `post::userid` and `$curr_user` refers to `user::id`.

The [Mem-Check] rule is somewhat intricate: it requires that two groups of foreign and primary key pairs – specifically, $t_1 :: c_p$ and $t_2 :: c_2$, along with $t_3 :: c_p$ and $t_2' :: c_2'$ – be involved in two separate conditions checks, respectively. That is, $(t_1 :: c_p, t_2 :: c_2) \in C$ and $(t_3 :: c_p, t_2' :: c_2') \in C$. In our example in Figure 3(b), the two conditions at lines 5-7 – `forumid = $fid` and `in_array($curr_user, $row)` – implement the authorization rules of the membership model (`user`, `user_forum::userid`, `forum`, `user_forum::forumid`) for the query statement at line 11. Here, `forumid` refers to `user_forum::forumid`, `$fid` refers to `forum::id`, `$curr_user` refers to `user::id`, and `$row` refers to `user_forum::userid`.

```
1                    DAL Specification
2  {
3      "name": "$wpdb->update",
4      "op": "UPDATE",
5      "table": {"name": "$0"},
6      "columns": {"name": "$1"},
7      "where": {"name": "$2"}
8  }
```

Figure 12: A DAL specification example in the benchmark SP Manager-4.57.

In the rule [Stat-Check], we have $t_1 :: c'_1 \in C$, ensuring that the status column is considered. For example, in Figure 3 (d), the condition at Line 7 checks the status of the post before adding a comment.

Finally, those sensitive operations that are not in the **SafeOp** set are reported as BOLA vulnerabilities violating corresponding authorization models. It is noteworthy that the two rules, [Own-Check] and [Mem-Check], can be easily extended to validate the existence of checks for the hierarchical authorization model. For clarity, we do not incorporate these rules in Figure 11. Furthermore, Figure 11 illustrates the simple case where each sensitive operation corresponds to a single authorization model. In reality, when operating on the `post` table in Figure 2, since (`user, post::userid`) $\in$ **OwnerModel**, and the tables `user`, `forum`, and `post` together constitute a hierarchical membership model, the statement must be validated for each model separately.

4.3 Implementation

We have developed BolaRay on top of TChecker [25], an inter-procedural static analysis tool for PHP applications. Specifically, TChecker performs inter-procedural data-flow analysis on PHP objects to infer their types or values. This enables precise identification of method call targets, facilitating the incremental construction of an precise call graph, which further enhances inter-procedural data-flow analysis. TChecker supports context-sensitivity but not path-sensitivity, which is also not required by BolaRay.Our implementation consists of 12K lines of Groovy code. Here, we elaborate on the implementation details of the techniques used in Section 4.1 and Section 4.2.

Parsing SQL Statements. We utilize regular expressions (e.g., `CREATE TABLE.*`, `SELECT.*FROM.*`, `UPDATE.*SET.*`, `INSERT INTO.*`, `DELETE FROM.*`) to identify SQL statements (as direct query strings) within the source code. Subsequently, we leverage existing research on log analysis [49, 54] to recursively analyze variables within these statements, aiming to distinguish static text from variables in SQL statements. After this initial step, we employ JsqlParser [19] to parse all SQL statements, which builds the schema as shown in Figure 6 and normalizes each statement in the form of $\langle k, t, c, (c', v')\rangle$, as depicted in Figure 7. Any SQL statement that cannot be successfully parsed is disregarded.

DAL Specifications. BolaRay relies on manual input Data Access Layer (DAL) specifications to handle SQL statements encapsulated in database manipulating framework APIs. The DAL, akin to an ORM (Object-Relational Mapping) framework, is tasked with direct database interactions, offering an abstract API for querying and manipulating data. The DAL specifications detail how these abstract APIs relate to SQL queries, with our DAL specifications drawing on existing methodology [42] for implementation.

Figure 12 presents a simplified DAL specification example in JSON format. The specification declares the API name (line 3), its corresponding SQL operation (line 4), as well as API parameters denoting the target table (line 5), the operating columns (line 6), and the conditions for the operation (line 7). Such a configuration enables BolaRay to reconstruct SQL statements from invocations of DAL APIs. In this paper, we have manually written Data Access Layer (DAL) specifications for 100 APIs, with an average of 13 lines per method.

Computing Aliases. We leverage the use-def based data dependency analysis in TChecker to compute aliases, where two variables are considered as aliases if they depend on a common variable. We enhanced TChecker's dependency analysis with three specific extensions: 1) We handle loops in the form of '`foreach ($arr as $key⇒$value)`', common in PHP, by linking `$arr` to both `$key` and `$value` in the def-use analysis, thus treating `$arr` as an alias for both `$key` and `$value`; 2) We introduce function summaries for common library functions, including `array_push` and `compact`; 3) We differentiate accesses to the same array with distinct indexes, e.g., `$row[0]` and `$row[1]`. In BolaRay, aliases are instrumental in associating a variable with a table column, which also forms the foundation to infer implicit foreign keys ([Connect]). In Figure 3(a), `$curr_user` refers to `user::id` if it aliases to v' and (l, `user::id`, v') $\in$ **Binding**.

Locating Sensitive Operations. Theoretically, all database query statements are sensitive operations that need to be checked by Figure 11. However, this overly conservative approach may introduce numerous false positives. Specifically, we have summarized three scenarios that may not require object-level authorizations:

- `SELECT` statements. It is too restrictive to check every `SELECT` statement since only those statements querying confidential information require object-level authorization. Revisiting our example in Figure 2: a user can read posts created by another user from the `post` table but cannot query his or her profile in the `profile` table. How to automatically deduce whether the queried information is confidential or not is an interesting topic worthy of separate investigation.
- `INSERT` operations in hierarchical models. In Figure 2, due to the absence of a hierarchical membership check involving the three tables – `user`, `forum`, and `post` – a user can insert a post into any forum regardless of their membership in that forum. We reported this issue to the original developer, but they do not perceive this as a security issue, considering it merely a functional bug. It remains debatable whether such violations could lead to vulnerabilities.
- `DELETE` operations in Status models. Typically, `DELETE` operations do not consider the impact on status.

In conclusion, we consider database operation not falling into the above three categories as sensitive operations.

Collecting Conditional Checks. The set of checks, C, guarding a statement $l :< -, t, -, (c, v) >$ includes conditions in the `WHERE` clause of the statement (i.e., checks implied by (c, v)) as well as

those in conditional branches on which l is control-dependent. We associate each program variable v in a condition with its corresponding table column (if applicable), and additional checks in the `WHERE` clauses of those statements binding v to a specific column are incorporated. Consequently, (c, v) implies that $(t :: c, t' :: c') \in C$ if both $(l, t' :: c', v') \in$ **Binding** and v' aliases to v hold, indicating a comparison between two columns. Otherwise, $t :: c$ is included in C, validating the column $t :: c$. Ultimately, the set of checks in C comprises each check that either validates a single column or compares two columns.

We follow the standard algorithm that computes the iterative post-dominance frontier of l as its control dependencies [10]. According to the standard definition, c is a post-dominance frontier of l if l post-dominates one of its successors but not c itself. In performing such control flow analysis, we also consider those aborting statements (e.g., `exit` and `die` in PHP) as an exit node in their corresponding control flow graphs.

4.4 Discussions

4.4.1 Soundness and Precision. We employ a series of heuristic-based rules to infer object-level authorization models. Those rules are summarized from common coding practices, and although they work well in practice, they are neither sound nor complete.

The foreign key analysis rules presented in Figure 8 accurately capture the semantics of foreign keys, i.e., table columns that directly or indirectly refer to primary keys of other tables (Definition 1). The precision and soundness of foreign key analysis are determined by the underlying alias analysis, which may produce false positives or false negatives. However, our experiments, as shown in Table 3, did not reveal any incorrect foreign keys (no false positives) or any missing known foreign keys (no false negatives).

In Figure 10, the rule [User-Table] deduces user tables based on the common design practice that user IDs must be stored in the global `SESSION` variable. To distinguish from other values also stored in `SESSION`, we further require that user tables must contain a column named with the substring '`password`', '`passwd`', or '`pwd`'. This restriction results in no false positives in our experiments (Table 3). However, applications may not follow this design, leading to potential false negatives.

The heuristics for determining status and admin columns are based on observations in our studied applications. Consequently, applications that deviate from these observations may experience false positives and false negatives. For instance, developers might use arbitrary types like `INT` to represent a status, which violates the first condition for identifying status columns. Additionally, applications might permit non-admin roles to manage users or lack user management entirely, leading to false positives and false negatives in the discovery of admin columns, respectively. Such false positives and false negatives will be further discussed in Section 5.1.

4.4.2 Threat Model. BolaRay detects those BOLA vulnerabilities that violate the four types of object-level authorization models outlined in Section 3.3. These four authorization models were developed from our empirical study of 101 known BOLA vulnerabilities. In practice, other types of vulnerability patterns may exist, which are beyond the detection of BolaRay.

The effectiveness of BolaRay relies on two key assumptions. First, we presume that most functional code, specifically the SQL statements manipulating database tables, is correct. Otherwise, BolaRay will fail to reconstruct correct table relations and infer authorization models based on those relations. This assumption is reasonable because any malfunctions would likely be reported promptly. Second, we assume that applications adhere to common programming practices observed in our study. BolaRay follows these practices to formulate a series of heuristic-based rules. If an application deviates from to these practices, its BOLA vulnerabilities may not be detected by BolaRay.

5 Evaluation

We evaluated BolaRay using 25 open-source database-backed PHP applications (see Table 2), which include 19 applications (from rows 3 to 21) that have been widely evaluated in previous works [2, 8, 28, 31, 45] and 6 applications (from rows 22 to 27) from our empirical study. These applications span various industries, containing diverse tables and SQL statements. The experiments were conducted on a MacBook Pro laptop equipped with an 8-core 2.0 GHz M1 Pro processor, 16 GB of memory, and MacOS Sonoma 14.4.1.

Our evaluation aims to answer the following research questions:

- RQ1. How precise is BolaRay in identifying object-level authorization models?
- RQ2. How effective is BolaRay in detecting BOLA vulnerabilities?
- RQ3. How efficient is BolaRay?
- RQ4. How does BolaRay compare with other existing approach?

5.1 Identifying Authorization Models

Accurately identifying authorization models is essential for the effectiveness of BolaRay. Table 3 outlines the counts of user tables, foreign keys, table relations, status columns, admin columns, and object-level authorization models inferred by BolaRay for each application. To evaluate BolaRay 's precision, we manually examined the generated reports. As depicted in Table 3, BolaRay correctly identifies 35 user tables (Columns 2-3) and 1,151 foreign keys (Columns 4-5, 100 of which are explicitly declared in the schema, all from the benchmark Rosariosis) with no false positives. We further manually checked each table and confirmed that there were no known false negatives. This confirms that the rules in Figure 10 and Figure 8 are effective and precise in practical scenarios. It is noteworthy that the rule [Connect] binds a variable to a table column with the necessary condition that the involved statements must be executed together. This filtering strategy, inspired by [8], did not produce false negatives in our experiments while effectively eliminating 273 false foreign keys.

Out of the 1,099 inferred table relations (Columns 6-11), only one false m:n relation in Scarf was reported (Column 11). This false positive is caused by the fact that BolaRay mistakenly classified an object table as a junction table, resulting in a false many-to-many relation. The identification of status columns exhibited a lower precision, with a false positive rate of 25% (Columns 12-13). This discrepancy is attributed to the fact that some status columns

Table 2: Statistics on Evaluation Applications.

Application	Source Code		Database		# SQL Queries					Description
	# Files	# LLOC	# Tables	# Columns	SELECT	INSERT	DELETE	UPDATE	Total	
Mybloggie-2.1.4	57	3,894	4	24	78	7	7	7	99	Content management system
Scarf-1.0	19	840	7	37	50	7	6	12	75	Conference system
Phpns-2.1.1alpha	30	2,012	13	100	68	11	9	8	96	News platform
Webid-1.2.2	239	16,304	57	367	415	83	72	197	767	Auction platform
SchoolMate-1.5.4	63	1,877	15	104	215	16	33	30	294	School management system
PhpNews-1.3.0	20	3,488	6	37	61	39	5	13	118	News platform
Timeclock-1.04	63	12,373	8	35	257	20	7	22	306	Employment management system
Hospital MS-4.0	73	1,843	11	88	54	10	3	18	85	Hospital management system
Doctor Apt-1.0.0	26	820	4	32	24	2	0	4	30	Doctor management system
Wheatblog-1.1	42	1,495	6	35	33	6	5	8	52	Content management system
PHP7-Webchess	30	3,307	7	48	63	14	12	20	109	Web game
Hocms-1.0	38	1,386	7	52	35	9	4	12	60	Home collection management system
Collabtive-3.1	74	18,152	20	141	139	20	43	35	237	Project management system
Oscommerce-2.4.2	436	25,287	51	358	374	465	190	124	1,153	Ecommerce platform
Piwigo-14.4.0	681	120,653	39	221	335	12	31	55	433	Photo management system
PhpBB-3.3.12	1,219	71,781	70	605	856	14	145	298	1,313	Online forum
SMF-2.1.4	329	87,394	73	525	745	24	229	219	1,217	Online forum
Opencart-4.0.2.3	1,005	56,785	154	937	700	173	259	138	1,270	Ecommerce platform
Zencart-2.0.1	1,312	63,107	110	896	746	319	102	103	1,270	Ecommerce platform
Odfs-1.0	44	1,734	5	35	34	8	7	15	64	Online discussion forum system
Admidio-4.1.12	241	20,659	38	366	352	40	69	79	540	Online user management system
TeamPass-3.0.0.22	156	21,558	45	331	576	147	86	195	1,004	Collaborative passwords manager
Rosariosis-8.9.4	374	36,437	92	937	1,010	29	92	88	1,219	School management system
SP Manager-4.57	1,399	163,932	29	204	303	70	44	133	550	Wordpress project management plugin
Openemr-7.0.0	795	95,373	255	3,351	1,413	162	72	262	1,909	Medical practice management system
Total	8,765	832,491	1,126	9,866	8,936	1,707	1,532	2,095	14,270	

Table 3: Number of Tables and Relations.

Application	# User Tables		# Foreign Keys		# Table Relationships						# Status Columns		# Admin Columns		# Ownership Models		# Membership Models		# Hierarchical Models		# Status Models	
					1:1		1:n		m:n													
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
Mybloggie-2.1.4	1	0	3	0	0	0	3	0	0	0	0	0	1	0	2	0	0	0	1	0	0	0
Scarf-1.0	1	0	7	0	0	0	3	0	1	1	1	1	1	0	2	0	1	1	4	0	0	1
Phpns-2.1.1alpha	1	0	12	0	0	0	12	0	0	0	2	0	1	0	5	4	0	0	4	0	1	0
Webid-1.2.2	2	0	72	0	1	0	70	0	0	0	11	2	1	0	12	4	0	0	65	6	18	6
SchoolMate-1.5.4	1	0	26	0	3	0	17	0	2	0	0	0	1	0	6	2	1	0	6	0	0	0
PhpNews-1.3.0	1	0	3	0	0	0	3	0	0	0	1	1	1	0	2	0	0	0	1	0	1	0
Timeclock-1.04	1	0	10	0	1	0	6	0	1	0	1	3	1	0	5	1	1	0	1	0	2	2
Hospital MS-4.0	3	0	10	0	2	0	8	0	0	0	2	0	0	0	6	2	0	0	2	0	0	0
Doctor Apt-1.0.0	1	0	2	0	0	0	2	0	0	0	1	0	0	0	2	0	0	0	0	0	0	0
Wheatblog-1.1	1	0	2	0	0	0	2	0	0	0	2	0	1	0	1	0	0	0	0	0	1	0
PHP7-Webchess	1	0	9	0	0	0	9	0	0	0	0	2	0	0	4	0	0	0	8	0	0	3
Hocms-1.0	1	0	5	0	0	0	5	0	0	0	6	0	0	0	1	0	0	0	0	0	3	0
Collabtive-3.1	1	0	33	0	0	0	21	0	6	0	2	0	1	0	5	1	3	1	21	1	8	1
Oscommerce-2.4.2	3	0	64	0	4	0	55	0	1	0	3	0	0	0	19	0	0	0	10	0	13	2
Piwigo-14.4.0	1	0	34	0	1	0	29	0	1	0	1	0	0	0	10	0	1	0	1	0	0	0
PhpBB-3.3.12	1	0	151	0	2	0	142	0	2	0	1	1	0	0	37	8	0	0	568	118	4	0
SMF-2.1.4	1	0	127	0	3	0	114	0	3	0	0	0	1	0	37	9	3	0	196	46	0	0
Opencart-4.0.2.3	2	0	91	0	1	0	87	0	1	0	0	0	1	0	15	1	0	0	0	0	0	0
Zencart-2.0.1	2	0	91	0	3	0	82	0	1	0	0	0	0	1	22	0	1	0	15	0	0	0
Odfs-1.0	1	0	4	0	0	0	4	0	0	0	4	1	0	0	3	0	0	0	1	0	2	2
Admidio-4.1.12	1	0	39	0	0	0	39	0	0	0	1	0	1	0	18	3	0	0	8	0	2	0
TeamPass-3.0.0.22	1	0	106	0	0	0	102	0	2	0	4	0	2	1	15	4	0	0	135	7	32	0
Rosariosis-8.9.4	2	0	102	0	7	0	93	0	1	0	0	0	1	2	47	2	2	0	80	0	0	0
SP Manager-4.57	1	0	18	0	0	0	15	0	1	0	1	0	1	0	6	0	0	0	7	0	0	0
Openemr-7.0.0	3	0	130	0	0	0	124	0	1	0	7	6	1	0	36	3	0	0	68	8	34	6
Total	35	0	1,151	0	28	0	1,047	0	24	1	51	17	17	4	318	44	13	2	1,202	186	121	23

are irrelevant to authorization, such as those columns controlling the frontend display language. Additionally, BolaRay accurately inferred 21 admin columns (Columns 14-15), with only 4 false positives, indicating the efficacy of the [Admin-Col] rule in Figure 11.

Ultimately, BolaRay identified 1,909 accurate models with only 255 false positives (Columns 16-23). It is noteworthy that neither user tables nor 1:n relationships, which were utilized for inferring ownership models, yielded false positives. However, 44 ownership models were misreported (Column 17). The reason for these discrepancies is that these applications define their own authorization models, which are inconsistent with the ownership models we inferred. These false positives were further propagated to false hierarchical models (Column 21), stemming from incorrect ownership models. The false positives related to membership models (Column 19) and status models (Column 23) originated from incorrect m:n relationships (Column 11) and incorrect status columns (Column 13), respectively.

Table 4: Results on 15 existing vulnerabilities.

Detected	CVE-2022-31295 CVE-2022-31294 CVE-2023-3063 CVE-2022-1551 CVE-2023-3303 CVE-2023-3304 HUNTR-24ae402f CVE-2023-1463 HUNTR-3bf6999c CVE-2023-2946 CVE-2023-2945 CVE-2023-2944 CVE-2023-2942 CVE-2022-2824 HUNTR-52da52b8

5.2 Effectiveness

We evaluate the effectiveness of BolaRay in terms of its ability to detect existing and new vulnerabilities.

5.2.1 Existing Vulnerabilities. Table 4 summarizes the results in detecting existing vulnerabilities from our study. Out of the 101 vulnerabilities examined, 31 are PHP vulnerabilities. After excluding 16 SELECT-related cases, we are left with 15 vulnerabilities for further analysis. BolaRay successfully identified all 15 existing BOLA vulnerabilities across the six PHP applications analyzed in our study, achieving a recall rate of 100%.

Table 5: BOLA vulnerabilities reported by BolaRay. MOC, MMC, MHC, and MSC denote missing ownership checks, missing membership checks, missing hierarchical checks, and missing status checks, respectively.

Application	# MOCs						# MMCs						# MHCs				# MSCs				# Total		# CVEs
	INSERT		DELETE		UPDATE		INSERT		DELETE		UPDATE		DELETE		UPDATE		INSERT		UPDATE				
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	
Mybloggie-2.1.4	0	0	0	0	0	0	0	0	0	0	0	0	1	0	2	0	0	0	0	0	3	0	1
Scarf-1.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Phpns-2.1.1alpha	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	3	0	3
Webid-1.2.2	0	1	7	0	8	9	0	0	0	0	0	0	3	0	0	0	17	3	0	0	35	13	8
SchoolMate-1.5.4	0	0	0	0	10	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	12	1	3
PhpNews-1.3.0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1	0	1	0	0	0	3	1	3
Timeclock-1.04	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	4	0	1
Hospital MS-4.0	0	1	2	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	1	4
Doctor Apt-1.0.0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
Wheatblog-1.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1
PHP7-Webchess	0	0	1	0	7	0	0	0	0	0	0	0	1	0	0	0	0	4	0	0	9	4	3
Hocms-1.0	0	0	1	0	2	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0	8	0	2
Collabtive-3.1	0	1	3	0	3	0	1	0	3	0	8	0	2	0	3	0	8	0	1	0	32	1	13
Oscommerce-2.4.2	0	0	0	2	0	7	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	11	0
Piwigo-14.4.0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
PhpBB-3.3.12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SMF-2.1.4	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2
Opencart-4.0.2.3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Zencart-2.0.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Odfs-1.0	2	0	2	0	5	0	0	0	0	0	0	0	0	0	0	0	2	0	2	0	13	0	4
Admidio-4.1.12	0	0	2	6	2	5	0	0	0	0	0	0	0	0	0	0	1	0	0	0	5	11	0
TeamPass-3.0.0.22	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
Rosariosis-8.9.4	0	0	2	3	2	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	5	3	0
SP Manager-4.57	8	0	0	0	9	0	4	0	0	0	5	0	0	0	0	0	0	0	0	0	26	0	3
Openemr-7.0.0	4	0	2	1	10	3	0	0	0	0	0	0	0	0	1	0	4	2	1	0	22	6	0
Total	14	4	23	12	67	27	5	0	3	0	13	0	10	0	9	1	45	10	4	0	193	54	52

5.2.2 New Vulnerabilities. Table 5 provides details about vulnerabilities reported by BolaRay. Overall, BolaRay reports a total of 247 vulnerabilities, of which 193 vulnerabilities have been manually confirmed as real vulnerabilities, resulting in a false positive rate of 21.86% (Columns 22-23). This total includes the 15 known vulnerabilities mentioned in table 4. Subsequently, we reported the 178 new vulnerabilities to their corresponding maintainers. To date, 155 vulnerabilities have been confirmed, with 52 CVE IDs granted.

BolaRay detects at least one BOLA vulnerability in 21 applications, except for Scarf, PhpBB, Opencart and Zencart. Upon examining Scarf, it was discovered that all sensitive operations are restricted to admin users, making object-level authorization models irrelevant. Nevertheless, by effectively identifying the admin column for Scarf, as detailed in Table 3, BolaRay reports no false BOLA vulnerabilities in this benchmark. The other three applications did not reveal vulnerabilities because they had already implemented proper object-level authorization checks for each object.

5.2.3 False Positives. BolaRay reports 54 false positives at a rate of 21.86%. Through a thorough investigation, we summarize the common causes of these false positives below:

Incorrect Status Model. There are 10 false missing status checks (MSCs) (Column 19), all caused by incorrectly inferred status columns, as discussed in Table 3. Such false positives can be addressed by manually annotating real status columns.

Application-level Authorization. BolaRay simplifies application-level access control policies with the *admin* role, and this simplification leads to 14 false positives. For example, in Admidio, BolaRay accurately derives an ownership relationship between users and photos, and consequently reported a missing ownership check (MOC) when editing photos. Nevertheless, this application does not implement object-level authorization and allows any user with the `editPhotoRight` permission to edit any photo at will. This discrepancy, in contrast to the [Own-Check] rule in Figure 11, leads to 11 false positives (Column 23) reported by BolaRay for this benchmark.

Column-level Authorization. BolaRay enforces object-level authorization, while some applications require the authorization model to be more finely-grained, restricting to specific columns. The auction system, Webid, is such an application. BolaRay correctly established an ownership relationship between the user table and the `auction` table, which stores auction items. Consequently, BolaRay flagged a MOC in scenarios when updates to the column `auction::current_bid` lack ownership verification. However, this particular column records the current highest bid of auction items which, by design, could be updated by any user, making this specific column universally changeable. BolaRay reported 9 false positives (Column 7) due to this discrepancy.

Limitations of TChecker. TChecker does not support variable functions, which are commonly used for implementing callbacks in PHP. Consequently, BolaRay reported 11 false positives in Oscommerce because its permission-checking APIs are implemented through variable functions. This limitation results in BolaRay 's inability to infer the admin column in this application.

5.2.4 Case Studies. Here we investigate three intriguing new BOLA vulnerabilities identified by BolaRay.

CVE-2024-1693. In SP Manager, there is an ownership relationship between the `user` table and the `sp_cu_project` table. However, the application only verifies ownership when deleting objects in the `sp_cu_project` table, while it fails to do so during updates to this table, allowing attackers to arbitrarily update other users' `sp_cu_project` entries, thereby violating system integrity. It is important to note that, due to the frequent updates of numerous APIs, developers can easily overlook object-level authorization for some APIs. This oversight undergoes the need for an automated tool like BolaRay to systematically scan each API.

CVE-2024-23009. The Collabtive application contains a *missing hierarchical check* vulnerability, wherein upon deleting a project folder, it only verifies the membership relationship between the user and the project, neglecting to validate the parent-child relationship between the project and its containing folder. As a result, attackers may delete any project folder. Such *missing hierarchical check* vulnerabilities demand multiple sub-checks, all of which need to be enforced to ensure safety.

CVE-2024-32166. In Webid, the `auction` table, which stores auction items, contains two status columns: `closed` and `suspend`. These two columns collaboratively determine whether an auction item can be purchased or not. However, Webid only checks the `closed` column, ignoring the `suspend` column. As a result, a suspended auction item can still be purchased. To avoid such vulnerabilities, developers need to check all required statuses for a specific operation. BolaRay detects such vulnerabilities by enforcing checks on all status columns of an object, and the missing check of any status column will signify a report. This conservative strategy guarantees safety, and we did not encounter any false positives due to this over-approximation.

5.2.5 Responsible Disclosure. All the new vulnerabilities detected by BolaRay can lead to serious consequences, including data loss, data tampering, and system instability. Recognizing the potential risks, we took the responsibility of disclosing all newly identified vulnerabilities in 25 database-backed applications, with detailed reports. We reached out to the corresponding organizations to report the total new vulnerabilities through their dedicated email addresses and security vulnerability reporting forms. Adhering to responsible disclosure practices, we will refrain from publicly releasing any unresolved vulnerabilities until they have been addressed by developers. As of now, 155 of the identified vulnerabilities have either been confirmed or resolved, and 52 CVE identifiers have been assigned to these reports.

5.3 Efficiency

Figure 13 showcases the times required to analyze the 25 applications listed in Table 2. Among them, SP Manager (a WordPress plugin) stands out for taking the longest time, amounting to 427.5 seconds. This underscores the efficiency of BolaRay in conducting its analyses. By integrating the data from Figure 13 and Table 2, a clear positive correlation emerges between the overall analysis time and the size of the codebase.

This analysis time can be divided into two main components: the time devoted to inferring authorization models and the time allocated to detecting BOLA vulnerabilities. Notably, a majority of the time is consumed by the model inference process. It is crucial to emphasize that the duration of model inference is directly related to the number of tables and SQL statements in the application, rather than its sheer volume of code. For instance, SP Manager, despite having a larger codebase, takes less time in model inference than Openemr, owing to its fewer table relations. This observation aligns with our expectations, as the process of model inference primarily involves analyzing SQL queries to infer table relationships.

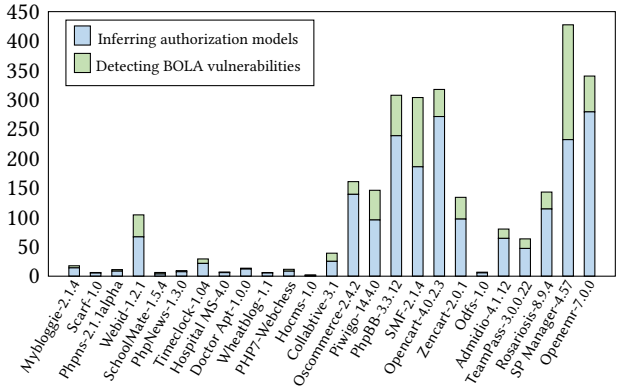

Figure 13: Analysis time (s) of BolaRay

Table 6: Ownership Models and BOLA vulnerabilities reported by Mace.

Application	# Ownership Models		# MOCs						# CVEs
			DELETE		UPDATE		Total		
	TP	FP	TP	FP	TP	FP	TP	FP	
Mybloggie-2.1.4	2	0	0	2	0	2	0	4	0
Scarf-1.0	1	0	0	0	0	0	0	0	0
Phpns-2.1.1alpha	5	4	0	0	1	0	1	0	1
Webid-1.2.2	10	2	0	0	1	0	1	0	1
SchoolMate-1.5.4	1	0	0	0	0	0	0	0	0
PhpNews-1.3.0	2	0	0	0	0	1	0	1	0
Timeclock-1.04	2	1	0	0	0	0	0	0	0
Hospital MS-4.0	4	2	2	0	3	0	5	0	4
Doctor Apt-1.0.0	1	0	0	0	0	0	0	0	0
Wheatblog-1.1	1	0	0	0	0	0	0	0	0
PHP7-Webchess	3	0	1	0	4	0	5	0	2
Hocms-1.0	1	0	0	0	0	0	0	0	0
Collabtive-3.1	4	5	3	0	3	0	6	0	4
Oscommerce-2.4.2	10	0	0	2	0	7	0	9	0
Piwigo-14.4.0	2	0	0	0	1	0	1	0	0
PhpBB-3.3.12	3	1	0	0	0	0	0	0	0
SMF-2.1.4	1	0	0	0	0	0	0	0	0
Opencart-4.0.2.3	3	0	0	0	0	0	0	0	0
Zencart-2.0.1	1	0	0	0	0	0	0	0	0
Odfs-1.0	1	0	0	0	0	0	0	0	0
Admidio-4.1.12	0	0	0	0	0	0	0	0	0
TeamPass-3.0.0.22	11	4	0	0	0	0	0	0	0
Rosariosis-8.9.4	2	0	0	0	0	0	0	0	0
SP Manager-4.57	2	0	0	0	0	0	0	0	0
Openemr-7.0.0	10	0	1	0	5	0	6	0	0
Total	83	19	7	4	18	10	25	14	12

5.4 Comparison

Mace [28] is the most closely relevant work to BolaRay. This tool relies on the manual annotation of user variables to identify ownership relationships, where the target table of an `INSERT` statement is owned by the inserted user variable (if it exists). Consequently, Mace checks whether the target tables of any `UPDATE` or `DELETE` statements enforce ownership checks in their `WHERE` clauses. Although Mace is closed-source, we reproduced the tool for a direct comparison with BolaRay. Instead of manually specifying user variables for Mace, we used the user variables inferred by BolaRay.

The results of this comparison are summarized in Table 6. Mace correctly identified 83 true ownership models and 25 true BOLA vulnerabilities for 7 applications, all of which were also disclosed by BolaRay. However, Mace missed the rest 235 ownership models and 79 MOC vulnerabilities reported by BolaRay because it identifies ownership relation only from `INSERT` statements. In contrast, BolaRay analyzes aliases together with all forms of SQL statements to infer such relationship.

Mace reported 19 false ownership models and 14 false vulnerabilities, with 15 of the false ownership models and 10 of the false vulnerabilities also being reported by BolaRay. The additional

4 false ownership models occurred because Mace fails to recognize junction tables, which BolaRay can handle. This difference in capability leads to discrepancies in analyzing database table relationships, further affecting the accuracy of inferred authorization models. Moreover, Mace reported 4 more false vulnerabilities than BolaRay. This because Mace only verifies checks in the `WHERE` clauses of SQL statements, whereas BolaRay also considers checks in conditional branches.

5.5 Limitations and Future work

Although effective in practice, BolaRay has the following limitations:

- *Manual DAL Specifications.* Currently, BolaRay requires manual annotation of DAL specifications to detail how SQL queries are encapsulated in framework APIs. On average, we need to write 13 lines of specifications per API. Although this is a one-time effort, it is considered a barrier to adopting the tool for new applications.
- *Generalizability.* BolaRay supports PHP applications; however, the approach is suitable for general database-backed applications. The rules developed for inferring authorization models and applying authorization checks (Figure 8 to Figure 11) can be readily implemented in a multi-language code analysis engine like CodeQL, and applied to applications written in other languages.
- `SELECT` *Statements.* BolaRay does not consider `SELECT` statements as sensitive operations, meaning it cannot detect BOLA vulnerabilities causing sensitive information leakage. This limitation is also present in all existing static detection tools [7, 9, 12–14, 28, 30] for BOLA vulnerabilities. Determining which data should be classified as sensitive in an application is a challenging topic worthy of further investigation [28].
- *Incorrect object-level authorization checks.* We developed BolaRay based on the finding that all studied BOLA vulnerabilities were caused by missing object-level checks. Thus, BolaRay is not designed for detecting those incorrect object-level authorization checks against a wrong variable, which are rare in practice.

In the future, we plan to overcome some of the limitations by utilizing large language models to automatically generate DAL specifications and identify sensitive data for handling `SELECT` statements.

6 Related Work

There have been a number of static tools capable of identifying BOLA vulnerabilities, including Waler [12], SPACE [30], ANOVUL [13], Cancheck [7], UrFlow [9], and FINAD [14]. These tools require the manual provision of authorization models and then apply various static analysis techniques, such as model checking and symbolic execution, to identify flaws in authorization implementation. To express the underlying authorization model, different specifications were proposed. For instance, Waler uses invariant variables, UrFlow employs SQL query constraints, and FINAD utilizes activity flow graphs to annotate authorization rules. In contrast, BolaRay automatically infers authorization models, making it distinctively innovative. The most closely relevant work to BolaRay is Mace [28], which assumes that an ownership relation exists between a user and a table as long as the `INSERT` statement of a table contains a user ID. Unlike Mace, our approach utilizes all types of SQL statements and expands the concept of ownership to include three additional authorization models.

Dynamic tools [20–22, 27, 38, 40, 46, 55] for BOLA vulnerability detection focus on identifying tamperable IDs and vulnerability triggers. For example, Authscope [55] analyzes user requests to identify tamperable IDs and examines responses to determine whether vulnerabilities are triggered. The effectiveness of these efforts relies on the ability to trigger as many code paths as possible through comprehensive requests.

A number of dynamic defense approaches have been proposed to prevent attacks that exploit BOLA vulnerabilities at runtime. FlowWatcher [29] detects violations of object-level authorization models at the HTTP proxy level. Conversely, SafeD [11], CLAMP [37], and Nemesis [26] focus on identifying these violations within the SQL server environment. All of these works rely on manual descriptions of object-level authorization models. BolaRay can enhance these existing methods by automatically inferring object-level authorization models.

General access control vulnerabilities have been studied extensively in the literature. AutoISES [44] statically infers security specifications of Linux systems by analyzing the correlation between data structure accesses and security checks. PeX [53] detects access vulnerabilities in the Linux kernel using a crafted indirect call analysis to associate permission checks with privileged functions. ACHyb [15] enhances the precision of PeX through combined static-dynamic analysis. Sun et al. [43] detect access control vulnerabilities in web applications by first constructing a sitemap for different roles, then checking whether accesses from unprivileged pages can successfully reach privileged pages. RoleCast [41] proposes a role-specific consistency analysis to detect inconsistent authorization checks in web applications. MPChecker [24] automatically identifies privileged operations in distributed systems by inferring user- and system-related variables via log-based analysis, then checks whether the privileged operations are guarded by permission checks. Compared to the above works, BolaRay targets the more granular BOLA vulnerabilities.

There have been numerous empirical studies targeting different issues in database-backed applications, including performance bugs [23, 51] and data constraint bugs [5, 16, 50]. To the best of our knowledge, this is the first paper to conduct an in-depth study of BOLA vulnerabilities in such systems.

7 Conclusion

We conducted the first in-depth study on BOLA vulnerabilities in database-backed applications and developed BolaRay, a novel tool for detecting such vulnerabilities. The key idea behind BolaRay is the use of a combined SQL and static analysis to automatically infer object-level authorization models. Our evaluation of BolaRay encompassed 25 popular database-backed applications, revealing 178 new critical vulnerabilities. Notably, 155 of these vulnerabilities have been confirmed, and 52 of them are documented with CVE identifiers.

Acknowledgement

We thank all reviewers for their valuable feedback. This work is supported by the National Key R&D Program of China (2022YFB3103900), the National Natural Science Foundation of China (62402474, 62132020 and 62202452), and the China Postdoctoral Science Foundation (2024M753295).

References

[1] 2020. Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/.
[2] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. 2018. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In *USENIX Security 18*. 377–392.
[3] F. E. Allen. 1970. Control flow analysis. *ACM Sigplan Notices* 5, 7 (1970), 1–19.
[4] apisecurity.io. 2022. API1:2019 — Broken object level authorizati. https://apisecurity.io/encyclopedia/content/owasp/api1-broken-object-level-authorization.htm.
[5] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2015. Feral concurrency control: An empirical investigation of modern application integrity. In *SIGMOD 15*. 1327–1342.
[6] Dan Barahona. 2022. What is Broken Object Level Authorization (BOLA) and How to Fix It. https://www.apisec.ai/blog/broken-object-level-authorization.
[7] Ivan Bocić and Tevfik Bultan. 2016. Finding access control bugs in web applications with CanCheck. In *ASE 16*. 155–166.
[8] An Chen, JiHo Lee, Basanta Chaulagain, Yonghwi Kwon, and Kyu Hyung Lee. 2023. SynthDB: Synthesizing Database via Program Analysis for Security Testing of Web Applications.. In *NDSS*.
[9] Adam Chlipala. 2010. Static Checking of {Dynamically-Varying} Security Policies in {Database-Backed} Applications. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*.
[10] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS* 13, 4 (1991), 451–490.
[11] Kevin Eykholt, Atul Prakash, and Barzan Mozafari. 2017. Ensuring Authorized Updates in Multi-user {Database-Backed} Applications. In *26th USENIX Security Symposium (USENIX Security 17)*. 1445–1462.
[12] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2010. Toward automated detection of logic vulnerabilities in web applications. In *19th USENIX Security Symposium (USENIX Security 10)*.
[13] Mahmoud Ghorbanzadeh and Hamid Reza Shahriari. 2020. ANOVUL: Detection of logic vulnerabilities in annotated programs via data and control flow analysis. *IET Information Security* 14, 3 (2020), 352–364.
[14] Mahmoud Ghorbanzadeh and Hamid Reza Shahriari. 2020. Detecting application logic vulnerabilities via finding incompatibility between application design and implementation. *IET Software* 14, 4 (2020), 377–388.
[15] 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 *ESEC/FSE 21*. 316–327.
[16] Haochen Huang, Bingyu Shen, Li Zhong, and Yuanyuan Zhou. 2023. Protecting data integrity of web applications with database constraints inferred from application code. In *ASPLOS 23*. 632–645.
[17] Stepan Ilyin. 2024. What is Broken Object Level Authorization? https://www.wallarm.com/what/broken-object-level-authorization.
[18] Lan Jiang and Felix Naumann. 2020. Holistic primary key and foreign key detection. *Journal of Intelligent Information Systems* 54 (2020), 439–461.
[19] JSQLParser. 2024. Java SQL Parser. https://jsqlparser.github.io/JSqlParser/.
[20] Ajay KumarShrestha, Pradip Singh Maharjan, and Santosh Paudel. 2015. Identification and illustration of insecure direct object references and their countermeasures. *International Journal of Computer Applications* 114, 18 (2015), 39–44.
[21] Malte Kushnir, Olivier Favre, Marc Rennhard, Damiano Esposito, and Valentin Zahnd. 2021. Automated black box detection of HTTP GET request-based access control vulnerabilities in web applications. In *ICISSP 2021*. SciTePress, 204–216.
[22] Xiaowei Li, Xujie Si, and Yuan Xue. 2014. Automated black-box detection of access control vulnerabilities in web applications. In *CODASPY 14*. 49–60.
[23] Xiaoxuan Liu, Shuxian Wang, Mengzhu Sun, Sicheng Pan, Ge Li, Siddharth Jha, Cong Yan, Junwen Yang, Shan Lu, and Alvin Cheung. 2023. Leveraging Application Data Constraints to Optimize Database-Backed Web Applications. *Proc. VLDB Endow.* 16, 6 (feb 2023), 1208–1221.
[24] Jie Lu, Haofeng Li, Chen Liu, Lian Li, and Kun Cheng. 2022. Detecting missing-permission-check vulnerabilities in distributed cloud systems. In *CCS 22*. 2145–2158.
[25] Changhua Luo, Penghui Li, and Wei Meng. 2022. TChecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in PHP applications. In *CCS 22*. 2175–2188.
[26] N. Zeldovich M. Dalton, C. Kozyrakis. 2009. Nemesis: Preventing authentication and access control vulnerabilities in web applications. (2009).
[27] S. Ghasemi M.A. Hadavi, A. Bagherdaei. 2021. IDOT: Black-Box Detection of Access Control Violations in Web Applications. *ISeCure* 13, 2 (2021).
[28] Maliheh Monshizadeh, Prasad Naldurg, and VN Venkatakrishnan. 2014. Mace: Detecting privilege escalation vulnerabilities in web applications. In *CCS 14*. 690–701.
[29] Divya Muthukumaran, Dan O'Keeffe, Christian Priebe, David Eyers, Brian Shand, and Peter Pietzuch. 2015. FlowWatcher: Defending against data disclosure vulnerabilities in web applications. In *CCS 15*. 603–615.
[30] Joseph P Near and Daniel Jackson. 2016. Finding security bugs in web applications using a catalog of access control patterns. In *Proceedings of the 38th International Conference on Software Engineering*. 947–958.
[31] Eric Olsson, Benjamin Eriksson, Adam Doupé, and Andrei Sabelfeld. [n. d.]. Spider-Scents: Grey-box Database-aware Web Scanning for Stored XSS. ([n. d.]).
[32] owsap. 2023. API1:2023 Broken Object Level Authorization. https://owasp.org/API-Security/editions/2023/en/0xa1-broken-object-level-authorization/.
[33] owsap. 2023. API5:2023 Broken Function Level Authorization. https://owasp.org/API-Security/editions/2023/en/0xa5-broken-function-level-authorization/.
[34] owsap. 2023. OWASP Top 10 API Security Risks – 2023. https://owasp.org/API-Security/editions/2023/en/0x11-t10/.
[35] owsap. 2024. SQL Injection. https://owasp.org/www-community/attacks/SQL_Injection.
[36] owsap. 2024. XSS. https://owasp.org/www-community/attacks/xss/.
[37] Bryan Parno, Jonathan M McCune, Dan Wendlandt, David G Andersen, and Adrian Perrig. 2009. CLAMP: Practical prevention of large-scale data leaks. In *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 154–169.
[38] I Putu Agus Eka Pratama and Alvin Maulana Rhusuli. 2022. Penetration Testing on Web Application Using Insecure Direct Object References (IDOR) Method. In *2022 International Conference on ICT for Smart Society (ICISS)*. IEEE, 01–07.
[39] reddelexc. 2023. Top IDOR reports from HackerOne. https://github.com/reddelexc/hackerone-reports/blob/master/tops_by_bug_type/TOPIDOR.md.
[40] Marc Rennhard, Malte Kushnir, Olivier Favre, Damiano Esposito, and Valentin Zahnd. 2022. Automating the detection of access control vulnerabilities in web applications. *SN Computer Science* 3, 5 (2022), 376.
[41] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. 2011. Rolecast: finding missing security checks when you do not know what checks are. In *OOPSLA 11*. 1069–1084.
[42] He Su, Feng Li, Lili Xu, Wenbo Hu, Yujie Sun, Qing Sun, Huina Chao, and Wei Huo. 2023. Splendor: Static Detection of Stored XSS in Modern Web Applications. In *ISSTA 23*. 1043–1054.
[43] Fangqi Sun, Liang Xu, and Zhendong Su. 2011. Static Detection of Access Control Vulnerabilities in Web Applications.. In *USENIX Security Symposium*, Vol. 64.
[44] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. 2008. AutoISES: Automatically Inferring Security Specification and Detecting Violations.. In *USENIX Security Symposium*. 379–394.
[45] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. 2023. 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)*. IEEE, 2658–2675.
[46] Nisal Madhushan Vithanage and Neera Jeyamohan. 2016. WebGuardia-An integrated penetration testing system to detect web application vulnerabilities. In *WiSPNET 16*. IEEE, 221–227.
[47] Enze Wang, Jianjun Chen, Wei Xie, Chuhan Wang, Yifei Gao, Zhenhua Wang, Haixin Duan, Yang Liu, and Baosheng Wang. 2024. Where URLs Become Weapons: Automated Discovery of SSRF Vulnerabilities in Web Applications. In *S&P 24*. IEEE Computer Society, 216–216.
[48] The world's first bug bounty latform for AI/ML. 2024. Common Vulnerabilities and Exposures (CVE). https://huntr.com/.
[49] W. Xu, L. Huang, A. Fox, D. Patterson, and M.I. Jordan. 2009. Detecting large-scale system problems by mining console logs. In *SOSP 09*. 117–132.
[50] Junwen Yang, Utsav Sethi, Cong Yan, Alvin Cheung, and Shan Lu. 2020. Managing data constraints in database-backed web applications. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1098–1109.
[51] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *ICSE 18*. 800–810.
[52] Chendong Yu, Yang Xiao, Jie Lu, Yuekang Li, Yeting Li, Lian Li, Yifan Dong, Jian Wang, Jingyi Shi, Defang Bo, et al. [n. d.]. File Hijacking Vulnerability: The Elephant in the Room. ([n. d.]).
[53] 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 *28th ${USENIX}$ Security Symposium*. 1205–1220.
[54] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A non-intrusive request flow profiler for distributed systems. In *OSDI 14*. 629–644.
[55] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2017. Authscope: Towards automatic discovery of vulnerable authorizations in online services. In *CCS 17*. 799–813.