BACAgent: LLM-Powered Detection of Broken-Access-Control Vulnerabilities in Web Applications

Fengyu Liu
Fudan University
Shanghai, China
fengyuliu23@m.fudan.edu.cn

Yuan Zhang
Fudan University
Shanghai, China
yuanxzhang@fudan.edu.cn

Zheng Lou
Fudan University
Shanghai, China
zlou25@m.fudan.edu.cn

Tian Chen
Fudan University
Shanghai, China
tianchen23@m.fudan.edu.cn

Youkun Shi
Fudan University
Shanghai, China
21110240048@m.fudan.edu.cn

Jiarun Dai
Fudan University
Shanghai, China
jrdai@fudan.edu.cn

Enhao Li
Fudan University
Shanghai, China
ehli23@m.fudan.edu.cn

Guangyu Zhou
ByteDance Inc.
Shanghai, China
zhouguangyu@bytedance.com

Zhongfu Su
ByteDance Inc.
Shanghai, China
suzhongfu.13@bytedance.com

Zequn Fang
ByteDance Inc.
Shanghai, China
fangzequn@bytedance.com

Abstract

Broken-Access-Control (BAC) vulnerabilities have long been recognized as one of the most severe security threats in web applications and have consistently ranked among the top risks in the OWASP Top 10. These vulnerabilities arise when applications fail to enforce intended access control policies, allowing attackers to perform unauthorized operations on protected resources and leading to serious security and privacy consequences. Despite extensive research on BAC vulnerability detection, existing approaches remain limited in effectiveness, particularly in accurately determining what access control is intended for a given resource.

In this paper, we present BACAgent, a novel white-box approach to detect BAC vulnerabilities in web applications. BACAgent revisits BAC vulnerability detection from the perspective of inferring intended Access Control Policies (ACPs). Our key insight is that, although explicit ACP documentation is often absent, real-world applications implicitly encode rich authorization intent throughout the codebase. Based on this insight, BACAgent adopts a three-step approach to infer intended ACPs and detect BAC vulnerabilities. First, BACAgent leverages LLM together with the RAG technique to identify and interpret implemented access control, treating them as code-level evidence. Next, BACAgent extracts semantic-level evidence from the codebase and jointly reasons over both code-level and semantic-level evidence to infer the intended ACPs. Finally, BACAgent detects BAC vulnerabilities by identifying inconsistencies between the inferred ACPs and implemented access control. We evaluate BACAgent on 25 open-source applications and 5 industrial applications developed in Java, PHP, and Go. BACAgent successfully detects 157 BAC vulnerabilities, including 65 previously unknown ones, significantly outperforming state-of-the-art techniques. We responsibly disclosed all newly discovered vulnerabilities. To date, 51 new CVE IDs have been assigned.

CCS Concepts

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

Keywords

Broken Access Control; Web Security

ACM Reference Format:
Fengyu Liu, Yuan Zhang, Zheng Lou, Tian Chen, Youkun Shi, Jiarun Dai, Enhao Li, Guangyu Zhou, Zhongfu Su, and Zequn Fang. 2026. BACAgent: LLM-Powered Detection of Broken-Access-Control Vulnerabilities in Web Applications. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3832642

CCS '26, The Hague, Netherlands.

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832642

1 Introduction

With the rapid evolution of web applications, modern online services (e.g., Google [6], X [24]) increasingly manage large volumes of privacy and business data, such as personal identities, financial records, and transactional information. These applications have therefore become attractive targets for attackers seeking unauthorized data access or manipulation. To protect sensitive resources,

developers commonly deploy access control checks to prevent unauthorized operations. However, when access control is inadequately designed or incorrectly implemented, applications may expose serious security flaws, known as Broken-Access-Control (BAC) vulnerabilities. According to the OWASP Top 10 reports, BAC vulnerabilities have steadily risen in importance and were ranked as the most critical category in OWASP 2025 [27].

Conceptually, BAC vulnerabilities can be broadly classified into two types. Broken Vertical Access Control (BVAC) occurs when applications fail to properly enforce role-based privilege boundaries, allowing low-privileged users (e.g., normal user) to access functionality intended for higher-privileged roles (e.g., administrator). Broken Horizontal Access Control (BHAC) arises when applications neglect ownership checks, enabling malicious users to access or manipulate data belonging to other users with the same role. Exploiting such vulnerabilities typically requires little technical effort, yet can rapidly compromise large amounts of sensitive data, disrupt normal business operations, and lead to substantial financial and reputational losses.

Given the severity of these threats, BAC vulnerability detection has attracted significant research attention. Existing approaches generally fall into black-box and white-box categories. Black-box techniques [36, 43] rely on crawling web applications at runtime, which often suffer from limited code coverage [36, 43] and require fully deployed execution environments, making large-scale analysis difficult. To overcome these limitations, recent research [37, 41, 45, 47, 50, 51] has explored static white-box techniques that analyze application source code to detect BAC vulnerabilities. However, these approaches primarily rely on heuristic rules to identify missing or inconsistent access control checks, without understanding what type of access control a resource is actually intended to enforce. As a result, they often misclassify benign accesses or overlook risky ones, leading to substantial false positives and false negatives.

Therefore, in this paper, we revisit the problem of detecting BAC vulnerabilities in white-box settings, aiming to achieve more effective detection. In real-world applications, developers implicitly design *Access Control Policies (ACP)* that define how different roles are allowed to interact with specific resources under different operations. BAC vulnerabilities arise when the implemented access control checks deviate from these intended policies. However, while implementations are fully visible in white-box settings, the intended ACP is rarely explicitly documented. Large applications often contain hundreds of resources whose access control requirements vary substantially across roles and operations, making it difficult to accurately infer the developer's intent. As a result, the core research question in BAC vulnerability detection becomes: *how to accurately infer the intended ACP?*

To achieve this, we observe that although explicit ACP documentation is rarely available in real-world applications, the intended ACP is not absent. As applications evolve to manage large amounts of user-owned data, developers inevitably encode their authorization intent implicitly throughout the codebase. As a result, applications contain abundant implicit evidence that reflects how resources are intended to be protected. In particular, such evidence manifests at different levels. Implemented access control checks reflect code-level authorization behavior, while naming conventions in APIs and database schemas convey semantic-level evidence. By jointly reasoning over both code-level and semantic-level evidence, it becomes possible to infer the intended ACP in a systematic manner. While this approach may seem intuitive, realizing it in practice introduces two non-trivial challenges.

- *C1: How to accurately identify and interpret implemented access controls?* In real-world applications, both vertical and horizontal access control are implemented in highly flexible and diverse ways, ranging from framework-provided mechanisms to developer-defined conditional checks. The lack of uniform syntactic patterns makes it difficult not only to reliably identify access controls, but also to correctly interpret their authorization behavior, such as which roles are being checked.
- *C2: How to effectively leverage multiple sources of evidence for ACP inference?* After identifying implemented access control checks, the next step is to infer the intended ACP by jointly considering code-level and semantic-level evidence. An intuitive approach is to use static analysis to extract semantic evidence from multiple sources (e.g., table names and code comments) and combine them with identified access controls. However, this naive integration is often ineffective, as only a small portion of the extracted evidence is authorization-related. Simply aggregating all evidence can easily distract the reasoning process and hinder accurate ACP inference.

To overcome these challenges, we propose BACAgent, a novel white-box approach for detecting BAC vulnerabilities in web applications. BACAgent aims to accurately infer the intended Access Control Policy (ACP) by jointly analyzing code-level (e.g., implemented checks) and semantic-level (e.g., table name) evidence embedded in the application. Specifically, BACAgent first performs behavior-oriented access control analysis to identify and interpret framework-based and developer-defined access control checks along resource access paths. For framework-based checks, it leverages static analysis to locate relevant APIs and employs retrieval-augmented generation (RAG) [38] with framework documentation to accurately interpret their enforced authorization behavior. For developer-defined checks, BACAgent extracts candidate conditional checks via static analysis and relies on the LLM's code understanding capability to determine whether they enforce role-based or ownership-based authorization. Next, BACAgent infers the intended ACP through an on-demand multi-source reasoning process. Instead of directly concatenating all available evidence, it incrementally provides carefully selected code-level and semantic-level evidence to the LLM, allowing the model to focus on authorization-related information while avoiding distraction from irrelevant context. Together, these techniques enable BACAgent to accurately infer resource-specific ACPs and detect BAC vulnerabilities by identifying inconsistencies between the intended ACPs and the implemented access control.

We evaluate the effectiveness of BACAgent on 25 open-source web applications and 5 industrial applications, with 106 known BAC vulnerabilities in total. These applications are implemented in multiple programming languages, including PHP, Java, and Go, demonstrating the language-independent advantage of BACAgent. As a result, BACAgent successfully discovered 157 vulnerabilities, including 65 (verified) high-risk 0-day and 92 known BAC vulnerabilities, achieving a precision rate of 78.89% and a recall rate of

86.79%. Besides, we compare BACAgent to state-of-the-art techniques (i.e., PCFinder [49], MOCGuard [41] and BOLARay [37]). BACAgent significantly outperforms state-of-the-art approaches, improving precision and recall by up to 49.72% and 78.51%, respectively. These newly discovered vulnerabilities enable unauthorized access to sensitive user data or modification of critical application resources. We responsibly disclosed all newly identified vulnerabilities. As of now, 51 have been assigned CVE IDs.

In summary, the paper makes the following main contributions:

- We revisit the problem of BAC vulnerability detection and formulate it as inferring the intended Access Control Policies from application code.
- We propose BACAgent, a novel white-box framework that effectively detects BAC vulnerabilities in web applications. We will open-source our BACAgent prototype upon publication.
- We evaluate BACAgent on 25 widely used open-source and 5 industrial applications implemented in PHP, Java, and Go, and successfully detect 65 verified 0-day BAC vulnerabilities, resulting in the assignment of 51 new CVE IDs.

2 Problem Statement

2.1 Broken-Access-Control (BAC) Vulnerability

Access control is the primary mechanism that web applications rely on to safeguard sensitive user information, including personal identities, financial records, and various business-critical resources. However, misconfigurations in these mechanisms would result in Broken-Access-Control (BAC) vulnerabilities [27], including issues like CWE-284 (Improper Access Control) [2]. These vulnerabilities enable attackers to bypass access control checks and carry out unauthorized operations such as deleting sensitive data, thereby threatening the confidentiality, integrity, and availability of web applications [25, 26]. According to existing research [43, 47], BAC vulnerabilities can be divided into *Broken Vertical Access Control (BVAC)* and *Broken Horizontal Access Control (BHAC)*.

Broken Vertical Access Control (BVAC). Vertical Access Control (VAC) regulates access to resources based on user roles, such as distinguishing administrator-level operations from those accessible to normal users. A BVAC vulnerability arises when the application fails to enforce the appropriate VAC checks, allowing users of one role to perform operations intended for higher-privileged roles (e.g., administrators). As illustrated in Figure 1, the `list` interface should be accessible only to administrators. However, without the VAC check, a normal user would be able to invoke this administrator interface, leading to a BVAC vulnerability. The highlighted line `@RequiresPermissions("admin")` (line 11) represents the missing VAC and shows the patch that restores the required check.

Broken Horizontal Access Control (BHAC). Horizontal Access Control (HAC) enforces resource ownership constraints by ensuring that a user can access or modify only the resource they own. In practice, this protection is typically achieved through an ownership check [41], where the application verifies that the currently authenticated user's identifier (e.g., $_SESSION['user_id']) matches the ownership field of the accessed resource (e.g., member_id). When this ownership check is missing or incorrectly implemented, the application fails to prevent users from accessing or manipulating other users' resources, resulting in a BHAC vulnerability. As shown in Figure 1, the `user_detail` interface should ensure that a user can view only their own order. The HAC check (lines 6-8) compares the current user's identifier (`SessionUtil.getCurrentMember()`) with the owner of the order (`getMemberId()`). The green highlighted area reflects the patch that introduces this HAC check. Without these lines, a user would be able to retrieve other users' order details, which results in a BHAC vulnerability.

```
1  @GetMapping("/user/order/detail")
2  /* For users to view their orders */
3  public OrderDetail user_detail(String orderId) {
4     OmsOrder order = orderMapper.selectById(orderId);
5     UmsMember member = SessionUtil.getCurrentMember();
6  +  if (!member.getId().equals(order.getMemberId())){
7  +     Asserts.fail("Failed");
8  +  }
9     return CommonResult.success(order);      HAC
10 }
                                  VAC
11 +@RequiresPermissions("admin")
12 @GetMapping("/admin/order/list")
13 /* For admin to manage orders */
14 public OrderDetail list() {
15    List<OmsOrder> orders = orderMapper.listOrder();
16    return CommonResult.success(orders);
17 }
```

Figure 1: Real-world BAC Vulnerability Examples.

2.2 Existing Works and Limitations

In recent years, existing works have explored various techniques for detecting BAC vulnerabilities. The first line of work adopts black-box techniques [36, 43], which are widely used in penetration testing [29, 33]. These approaches send HTTP requests to the target application and analyze the corresponding responses to determine whether BAC vulnerabilities are present. However, they require deploying a full runtime environment of the target application, which requires substantial manual effort. In addition, due to the limited code coverage achieved by current crawling techniques [34, 36, 43, 52], these approaches often suffer from a high false negative rate.

The second line of work relies on white-box techniques such as static analysis to examine the source code of the target application [37, 41, 45, 47, 50, 51]. These approaches leverage various application-specific inputs (e.g., runtime logs and code annotations) to identify access control checks and treat inadequately protected accesses to sensitive resources as BAC vulnerabilities. While the high-level idea is reasonable, these approaches face a fundamental limitation: *they cannot determine what type of access control the target resource is intended to enforce.* For example, as shown in Figure 1, reading the `order` resource should enforce HAC when accessed by normal users. In contrast, reading the `product` resource does not require HAC for read operations, since users are allowed to view and select any product. In other words, it is designed to be publicly accessible. Existing works rely primarily on heuristic rules, which cannot capture the semantics or intended properties of the

accessed data. As a result, they cannot determine which access control checks should be enforced for each resource, leading to substantial false positives and false negatives.

2.3 Rethinking the BAC Vulnerability Detection

Existing approaches struggle to understand what access control should actually be enforced and thus cannot effectively detect BAC vulnerabilities. Even in white-box settings where the program logic is fully visible, current techniques still cannot reliably determine whether the data is protected as intended. Therefore, in this work, we rethink the problem of detecting BAC vulnerabilities in white-box settings, aiming to develop a more effective approach for detecting BAC vulnerabilities.

Root Cause Analysis. To better understand BAC vulnerabilities, let us first revisit the underlying root cause of BAC vulnerability. In real applications, developers generally define *Access Control Policies (ACP)* that specify how different roles are intended to interact with specific data. For instance, as shown in Figure 1, an administrator may view all orders, while a normal user is restricted to viewing only their own. However, when developers fail to implement access control in accordance with the intended ACP, the authorization guarantees break down, resulting in a BAC vulnerability. This mismatch between the intended ACP and the implemented checks fundamentally explains how BAC vulnerabilities occur.

Terminology and Formalization. To clearly define and reason about BAC vulnerabilities, we first formalize the ACP used throughout this work. The ACP specifies the access control requirements for each data resource under different roles and operations. Formally, an ACP is represented as a four-element tuple: $ACP = \langle D, O, R, AC \rangle$

- **Data (D)**. The protected data item, represented by a database table that stores the application's data. For example, an `order` table contains all orders placed by users, and a `product` table records all products available in the store.
- **Operation (O)**. The database operation performed on the data item. In general, such operations include reading, creating, updating, and deleting. Different operations may require different types of access control to ensure proper protection.
- **Role (R)**. The role of the user performing the operation. Typical roles include a normal user who can only access resources associated with their own account, and an administrator who is allowed to access broader system-wide resources.
- **Access Control (AC)**. The type of access control checks that must be satisfied for the given (D, O, R). This may include vertical access control (VAC) that restricts operations based on roles, horizontal access control (HAC) that restricts operations to data owned by the same user, a combination of both (e.g., a logged-in user reading their own order), or none for public resources (e.g., a blog page accessible without login).

Key Research Question. As described in our root cause analysis, detecting BAC vulnerabilities fundamentally relies on the intended ACP for each resource and then examining whether the implementation follows that policy. Therefore, the effectiveness of any detection approach hinges on the ability to obtain accurate ACP.

However, in real-world scenarios, applications rarely provide formal specifications or documentation that clearly describe the intended ACP for each data item. Large applications often manage a substantial number of resource data (D), and the appropriate access control (AC) for each resource varies significantly depending on the interacting role (R) and operation (O). Therefore, determining whether a particular database operation should enforce HAC, VAC, a combination of both, or is publicly accessible is difficult when the ACP is not explicitly documented. As a result, inferring the developer's intended ACP solely from the application code remains a challenging task and becomes the core problem in white-box BAC detection. This leads to the core research question in BAC vulnerability detection: *How to accurately infer the intended ACP?*

3 Overall Idea

In this paper, we propose a novel approach that accurately infers the ACP of resources in real-world web applications, thereby enabling effective detection of BAC vulnerabilities. In the following, we first introduce our key observations about ACP inference (in §3.1), then discuss the challenges encountered and present our solution (in §3.2).

3.1 Key Insight and Observation

In practice, real-world applications rarely provide explicit documentation describing their intended ACP, which makes it difficult to reason about whether a given resource access is properly protected. Nevertheless, this does not imply that the intended ACP is absent. Instead, as applications evolve to manage large amounts of user-owned data, developers inevitably encode their access control intention implicitly throughout the codebase during design and implementation.

Therefore, the key insight behind our approach is that, *although ACP documentation is missing, the application itself contains abundant implicit evidence that reflects how resources are intended to be protected*. By systematically collecting and reasoning over such evidence, it becomes possible to infer the intended ACP in a principled manner. Next, we summarize three key observations that motivate our approach to infer ACP, and explain how these observations enable the design of our approach.

Observation I: *Correctly implemented access controls provide code-level evidence for inferring the ACP.* In real-world applications, most operations are implemented without BAC flaws, indicating that the access control checks along these execution paths are largely correct. Such correctly implemented access controls implicitly encode the intended ACP requirements for the underlying resources, thereby providing reliable code-level evidence about what type of access control should be enforced for a given resource. For example, an application may access the `order` resource in multiple code locations, where most accesses are properly protected by access controls. In this case, these correctly protected accesses reveal the authorization requirements for the `order` resource, serving as references for inferring the intended ACP. Therefore, by identifying and interpreting these implemented access control checks, we can obtain reliable code-level evidence for inferring the ACP.

Observation II: *Naming conventions embedded in the codebase provide semantic-level evidence for inferring the ACP.* Beyond implemented access control checks, the codebase itself conveys developers' authorization intent through a variety of natural-language evidence. This evidence appears across multiple sources, such as

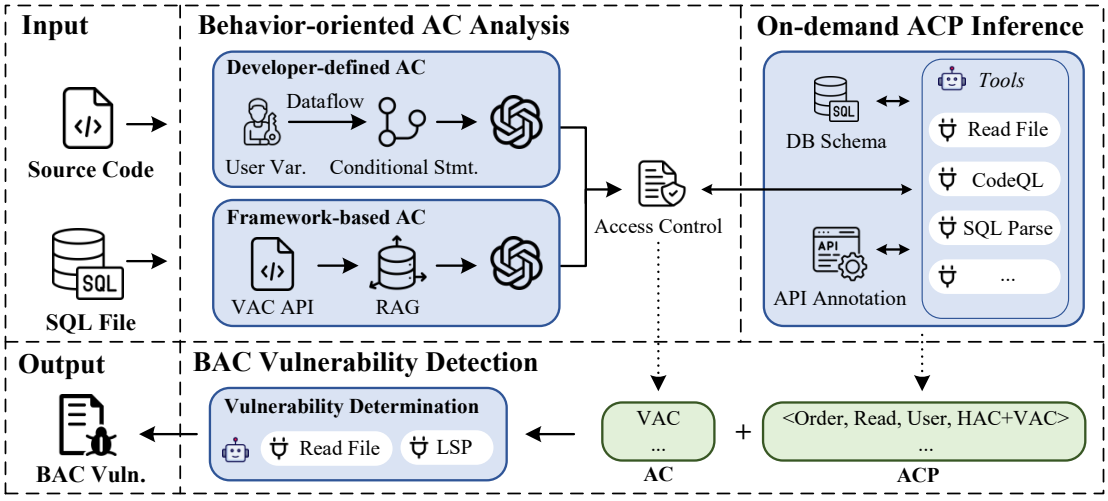

Figure 2: Architecture of BACAgent.

the naming of web interfaces (e.g., endpoints under `/admin/*` typically indicate administrator-only access), database schema design (e.g., table and column names like `order.user_id` indicating user ownership of order resource), as well as code comments. Such semantic-level evidence frequently reveals whether a resource is user-owned, role-restricted, or publicly accessible. Although any individual piece of evidence may be incomplete or ambiguous, collectively interpreting these natural-language semantics provides mutually reinforcing insight that is highly informative for inferring the intended ACP.

Observation III: *LLM can understand both code and natural language, and thus jointly leverage code-level and semantic-level evidence to infer the ACP.* LLM has demonstrated strong capabilities in understanding code logic and natural-language semantics. This makes them particularly well-suited for identifying and interpreting implemented access control logic, such as identifying what roles are being explicitly checked in code. At the same time, LLM can incorporate semantic-level evidence embedded throughout the application codebase, such as comments, naming conventions, and database schema. By jointly reasoning over these code-level and semantic-level evidence, LLMs can assist in inferring the intended ACPs of specific resources.

3.2 Challenges and Solutions

Guided by the above observations, our approach to infer ACP follows three intuitive steps: (1) identifying and interpreting code-level evidence, i.e., access controls already implemented in the application, (2) collecting and organizing semantic-level evidence from multiple sources, and (3) leveraging the LLM to infer the intended ACP by jointly analyzing code-level and semantic-level evidence. While this approach may seem intuitive, realizing it in practice introduces two non-trivial challenges.

Challenge I: How to accurately identify and interpret implemented access controls? In real-world applications, both VAC and HAC are implemented in highly flexible and diverse ways. Access control logic may be realized through framework-provided mechanisms or developer-defined conditional checks, often without explicit or uniform syntactic patterns. Consequently, traditional modeling and heuristic-based approaches [37, 41, 45, 49] generalize poorly when attempting to identify access control across various implementations. More importantly, even when such checks are successfully identified, accurately interpreting their authorization behavior, such as which roles are being validated, remains challenging. This understanding is essential for ACP inference, yet traditional approaches fail to capture it. Moreover, directly applying LLMs to raw code is also ineffective. As shown in §5.5, such direct application achieves a precision rate of only 58.30%.

Solution: To accurately identify and interpret implemented access controls, we propose a behavior-oriented access control analysis approach. Our key insight is that, although access controls are implemented in diverse syntactic forms, they follow a small set of stable authorization behaviors. In practice, access controls can be broadly categorized into two types, i.e., framework-based and developer-defined. For framework-based access controls, authorization logic is encapsulated in framework-provided APIs or middleware (e.g., security filters [5] and annotations [22]). While the concrete APIs vary across frameworks, their enforced checks are explicitly specified in framework documentation. We locate such callsites via static analysis and leverage the RAG technique [38] with framework documentation to accurately analyze what access control is enforced. Developer-defined access controls are implemented through custom conditional statements and can be divided into VAC and HAC. VAC validates user roles, while HAC checks whether the accessed resource belongs to the current user. We extract candidate conditions via static analysis and use the LLM's code understanding capability to analyze whether they enforce role validation (VAC) or ownership validation (HAC).

Challenge II: How to effectively leverage multiple sources of evidence for ACP inference? After identifying implemented access controls, the next step is to infer the intended ACP by jointly considering multiple sources of evidence. An intuitive idea is to leverage static analysis techniques to extract semantic-level evidence from multiple sources such as table names and code comments, and then feed this evidence together with the code-level

evidence, i.e., the identified access controls, into LLM to infer the ACP based on its code and semantic understanding capabilities. However, this naive concatenation strategy is ineffective in practice. Although the codebase contains abundant information, only a small subset is relevant to access control and useful for inferring the ACP of a specific resource. Naively concatenating all extracted evidence can easily distract the model, triggering the lost-in-the-middle phenomenon [44]. As a result, the model may fail to focus on information related to access control, making it difficult to accurately infer resource-specific access control policy.

Solution: To effectively leverage multi-source evidence for ACP inference, we propose on-demand ACP inference, which controls how evidence from multiple sources is provided to the LLM during ACP inference. For each resource (e.g., the `order` table), we first collect two categories of evidence relevant to ACP inference. Semantic-level evidence is extracted via static analysis and includes the resource's database schema, API paths that access the resource, and relevant comments. Code-level evidence consists of the identified access control code snippets, which are executed on the path before the resource is accessed. Next, instead of directly concatenating all collected evidence and feeding it to LLM, we adopt an on-demand inference strategy. The LLM is initially provided with a small set of high-confidence evidence, such as identified access control checks and the resource's database schema. This evidence jointly constrains how access controls are intended to be enforced and characterizes the inherent properties of the protected resource, thereby grounding the model's reasoning and enabling accurate ACP inference. When the high-confidence evidence is insufficient or ambiguous, additional information is incrementally supplied upon request. This design allows the LLM to selectively consult code-level or semantic-level evidence only when needed, reducing distraction from irrelevant information and enabling focused reasoning over the ACP of each resource.

4 BACAgent Design

In this section, we present the design details of our approach, BACAgent, for detecting BAC vulnerabilities in web applications. As shown in Figure 2, BACAgent primarily consists of three phases.

- *Behavior-oriented Access Control Analysis* (§4.1) identifies and interprets access control logic already implemented in the application, covering both framework-based and developer-defined mechanisms, and extracts their authorization behavior.
- *On-demand ACP Inference* (§4.2) infers the intended access control policy (ACP) for each resource by selectively integrating multi-source evidence, including database schemas, API naming information, and identified access control, while avoiding unnecessary attention dilution.
- *BAC Vulnerability Detection* (§4.3) detects BAC vulnerabilities by comparing the inferred ACP with the implemented access control along each resource-operation path.

4.1 Behavior-oriented Access Control Analysis

In this phase, BACAgent identifies and interprets access control logic already implemented in the application. As observed in §3.1, access control in real-world applications is primarily implemented in two forms: *framework-based access control*, which is enforced by security frameworks through predefined APIs, and *developer-defined access control*, which is implemented using application-specific conditional statements. Next, we describe how BACAgent identifies and interprets these two access control mechanisms.

```
1  @RequiresPermissions("user:delete")
2  @RequestMapping("/user/delete")
3  public String deleteUserInfo(String userId) {
     ...
4    return userService.delete(userId);
5  }

6  public class AppRealm extends AuthorizingRealm {
7    protected AuthorizationInfo doGetAuthorizationInfo() {
       …
8      if ("admin".equals(role))
9        roleInfo.addStringPermission("user:delete");
10   }
11 }
```

Figure 3: Framework-based VAC Example.

4.1.1 Framework-based Access Control. Framework-based access control is widely adopted in modern web applications, where authorization logic is delegated to security frameworks (e.g., Spring Security [19] and Apache Shiro [22]). These frameworks typically enforce VAC by validating the roles of users accessing protected resources. In such cases, access control is implemented through framework-provided APIs or configuration rules, rather than conditional checks written by developers. However, although it is often possible for LLM to identify that such code is performing framework-based access control, it is still unable to interpret which roles are being checked by understanding the code alone. This limitation arises because framework-based access control relies on complex framework logic and configurations. As a result, even though LLM has been trained on relevant knowledge, it is prone to context distraction and struggles to focus on the code that configures roles, leading to incorrect role inference. As reflected in our evaluation (see §5.5), directly prompting LLM with framework-based access control code achieves a precision rate of only 58.30%.

To bridge this gap, BACAgent adopts the retrieval-augmented generation (RAG) technique. By grounding LLM reasoning with security framework documentation, BACAgent enables LLM to accurately understand the behavior of framework-provided access control APIs, and thus reliably identify framework-based access control in application code and interpret which roles they validate. Specifically, this process consists of two steps: Knowledge Base Construction and RAG-based Access Control Analysis.

Knowledge Base Construction. RAG has proven effective in retrieving relevant information from a knowledge base, thereby significantly enhancing the accuracy of LLM-generated responses. This technique has been extensively explored in a variety of code-related tasks [54, 55]. To leverage RAG, BACAgent first constructs a knowledge base by collecting documentation from widely-used security frameworks that implement access control mechanisms (e.g., Spring Security [19] for Java, and Symfony Security [20] for PHP). These documents typically describe security annotations,

configuration rules, and API usage patterns related to authorization. The collected framework documentation is parsed and converted into vectors using an embedding model, and then stored in a vector database for efficient retrieval during analysis.

Table 1: Framework Access Control APIs

Framework/Lib	Class	APIs	Lang
Java EE	Filter Interceptor	doFilter	Java
Spring Security	AccessDecisionManager OncePerRequestFilter MethodInterceptor	decide doFilter invoke	Java
Apache Shiro	Subject ShiroFilter MethodInterceptor	isPermitted doFilter invoke	Java
Laravel	Middleware	handle	PHP
Symfony	Event Listener	onKernelRequest	PHP
PSR-15	MiddlewareInterface	process	PHP
Gin	Middleware	/	Go
Hertz	Middleware	/	Go
Casbin	Enforcer Model	Enforce	Go

RAG-based Access Control Analysis. After constructing the knowledge base, BACAgent performs RAG-based analysis to identify and interpret framework-based access control checks in application code. To begin, BACAgent models common access control framework APIs across different programming languages, such as Filter [5] in Java and Middleware [12] in Go (as defined in Table 1). Next, BACAgent employs static analysis to extract relevant access control code snippets, treating them as candidate access control for further analysis. For each candidate, BACAgent queries the knowledge base to retrieve the most relevant documentation fragments that describe how the corresponding framework enforces access control. These retrieved fragments, along with the local code context, are provided to the LLM as in-context references. Guided by this augmented context, even if an access control code snippet lacks clear authorization semantics on its own, the LLM can leverage the RAG-retrieved documentation to identify and interpret its access control behavior.

Running Example. We refer to Figure 3 as a running example to further illustrate RAG-based Access Control Analysis. As shown in lines 1–5, the `delete` API provides functionality for administrators to delete users from the application. This operation is protected by the annotation `@RequiresPermissions("user:delete")` (line 1), which implements VAC at the framework level. However, based solely on this code snippet, BACAgent cannot determine which specific role is being validated. By leveraging RAG, BACAgent retrieves relevant framework documentation, which reveals that developers typically extend the `AuthorizingRealm` class to define role-based access control logic [18]. Guided by this knowledge, BACAgent invokes static analysis to locate the corresponding `AppRealm` class and identifies the role configuration defined therein (line 10). As a result, BACAgent determines that the `delete` API enforces VAC for the administrator role.

4.1.2 Developer-defined Access Control. Developer-defined access control is typically implemented through conditional statements. Unlike framework-based access control, which relies on predefined APIs and configuration rules, it offers greater flexibility and can be tailored to the specific needs of applications. There are two main categories of developer-defined access control: *VAC*, which enforces authorization based on roles, and *HAC*, which enforces authorization through ownership checks. This section describes how BACAgent analyzes both VAC and HAC, focusing on identifying relevant code snippets and interpreting authorization behavior.

Table 2: Session APIs modeled by BACAgent

Framework/Lib	Class	APIs	Lang
Jakarta Servlet	HttpSession	getAttribute	Java
Spring Security	Authentication, SecurityContext	getPrincipal, getAuthentication	Java
Apache Shiro	SecurityUtils	getSubject	Java
Java JWT	Jwts	parser, claims	Java
Native JDK	ThreadLocal	get	Java
Native PHP	/	$_SESSION	PHP
Laravel	SessionManager	get, put	PHP
Symfony	SessionInterface	get, set	PHP
Gin	Context	Get, MustGet	Go
Hertz	RequestContext	Get, MustGet	Go
Go JWT	jwt.Token	Parse, Claims	Go

VAC Analysis. Vertical access control (VAC) ensures that users have the necessary roles to access specific resources, thus requiring the retrieval of the current user's role and reflecting role-related semantics. To identify VAC, BACAgent employs a three-step approach. Specifically, BACAgent first models common session management APIs such as PHP's `$_SESSION` and Java's `HttpSession` (as defined in Table 2), which typically store information about user roles. Next, BACAgent performs data flow analysis to extract conditional statements (e.g., `if` statements) that involve data flow from session APIs to the `if` condition. BACAgent extracts these conditional statements and treats them as candidate VAC checks. Finally, BACAgent uses LLM to determine whether these statements involve behavior that retrieves role-related information from session APIs and checks the role. If so, the statement is identified

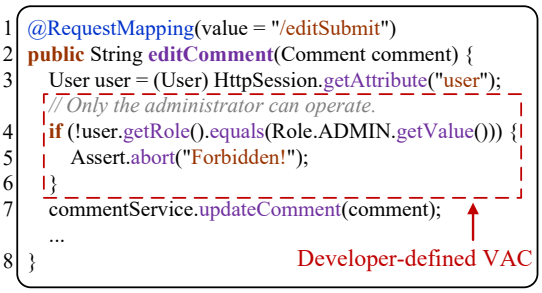

Figure 4: Developer-defined VAC Example.

as VAC. For example, as shown in Figure 4, the `if` statement (line 4) involves data flow from the session API (i.e., `HttpSession`), retrieves the role, and checks whether the user has an `admin` role. BACAgent therefore identifies it as a VAC.

HAC Analysis. Horizontal access control (HAC) ensures that users can only access resources they own. Similar to VAC, HAC also retrieves the current logged-in user's identity through session APIs and checks it using conditional statements. However, unlike VAC, which focuses on role-based authorization, HAC verifies ownership of the resource by comparing the logged-in user's ID with the resource's owner ID. Therefore, to identify HAC, BACAgent first extracts a candidate set of conditional statements that check session data, similar to the process in VAC analysis. Next, BACAgent performs data flow analysis to further determine whether the variable involved in the `if` condition originates from a database operation, specifically the resource's owner ID. Finally, BACAgent leverages the LLM to interpret whether the conditional statement compares the user ID obtained from the session with the owner ID retrieved from the database, and classifies it as HAC if such an ownership check is performed. For example, as shown in Figure 1, the code retrieves the `memberId` from the database and compares it with the logged-in user's ID. BACAgent therefore identifies it as a HAC.

4.2 On-demand ACP Inference

Access control policy (ACP) inference is a critical step for identifying BAC vulnerabilities. In this phase, BACAgent extracts semantic-level evidence from multiple sources and combines it with code-level evidence, i.e., the identified access control checks, to construct structured inputs for assisting LLM-based ACP inference. However, as discussed in §3.2, naively providing all extracted evidence to the LLM often leads to the lost-in-the-middle problem, which significantly degrades the accuracy of ACP inference.

To address this, BACAgent adopts an on-demand inference strategy that incrementally provides only the evidence required for the current inference, rather than presenting all available information at once. This design enables the LLM to effectively utilize multi-source evidence and accurately infer the intended ACP. Next, we describe how BACAgent extracts semantic-level evidence, followed by our on-demand ACP inference technique.

4.2.1 Semantic-level Evidence Extraction. In this step, BACAgent extracts semantic-level evidence from two sources: database schema and API annotations. The database schema captures how resources are stored and structured in the database, while API annotations describe the functionality of endpoints that operate on specific resources. Together, these two types of evidence provide semantic information for ACP inference beyond code-level access controls.

Database Schema. The database schema plays a pivotal role in understanding the ACP of a specific resource. It primarily provides two types of semantic information: naming semantics and structural semantics. For naming semantics, BACAgent leverages the SQL parsing tool to extract table names and column names associated with the target resource. These naming conventions encode rich natural-language semantics and provide useful evidence for LLM to reason about the intended usage of a resource. For example, a table named `order` typically stores user order records, while columns such as `phone` and `address` indicate that the resource contains sensitive user information. Such resources usually require strict enforcement of both HAC and VAC to prevent unauthorized access. For structural semantics, BACAgent analyzes foreign key constraints defined in the database schema. As observed in prior work [37, 41], such structural information captures explicit relationships between resources and their associated owners, and thus provides effective evidence for identifying resource ownership and inferring access control policies.

API Annotations. API annotations include information such as comments (e.g., line 2 in Figure 1), URL paths (e.g., `/user/order` in Figure 1), method signatures, and call chains to specific resources. These annotations follow naming conventions that reflect the functionality of each web API endpoint and indicate which underlying resources are exposed or manipulated. For example, method names such as `deleteUser` explicitly indicate the accessed resource and the performed operation, which provides useful evidence for understanding the intended access control requirements. BACAgent leverages the static analysis technique to extract API annotations from these sources. By leveraging this information, BACAgent can better understand the operational context of a specific resource, which in turn supports more accurate ACP inference.

In addition, BACAgent employs static analysis to extract call chains from API entry points to resource-specific operations, providing the LLM with richer semantic information for inferring API functionality. To support this analysis, BACAgent is equipped with a file-reading tool that enables it to retrieve the source code of callsites along the extracted call chains. By examining this contextual information, BACAgent associates API endpoints with the resources they access or modify, thereby enabling a clearer understanding of API functionality and resource ACPs.

4.2.2 On-demand ACP Inference. Given the extracted multi-source evidence, BACAgent infers the ACP in an on-demand manner. Instead of directly concatenating all available evidence and providing it to the LLM at once, BACAgent incrementally supplies evidence based on the needs of the current inference step. Specifically, BACAgent starts with a small set of high-confidence evidence, including the identified access control statements and the database schema of the target resource. These inputs jointly constrain how access control is enforced in the code and characterize the inherent properties of the protected resource, providing a grounded context for ACP inference. When the LLM determines that the provided

information is insufficient to infer the ACP or leads to ambiguous reasoning, BACAgent selectively retrieves additional evidence from other sources, such as API annotations.

In addition, to avoid overwhelming the LLM with excessive context, BACAgent incrementally supplies additional evidence in a step-by-step manner. For example, when multiple call chains exist between an API entry point and a resource operation, or when call chains are excessively long, BACAgent provides them to the LLM incrementally rather than all at once. For long call chains, BACAgent applies a length threshold (set to 20 callsites) and summarizes intermediate segments, preserving callsites near both the API entry and the resource operation, where API functionality semantics typically reside. This design enables the LLM to focus on the most relevant code while avoiding attention dilution caused by lengthy and less informative call chains. Similarly, for long code comments, BACAgent first asks LLM to summarize the content and remove information irrelevant to access control, retaining only semantic evidence that is useful for ACP inference. This process allows the LLM to access semantic-level and code-level evidence on demand, avoiding overwhelming context and enabling focused reasoning about the ACP of a specific resource.

4.3 BAC Vulnerability Detection

In this phase, BACAgent detects BAC vulnerabilities by comparing the inferred ACPs with the identified access control checks along resource access paths. Specifically, the detection process consists of three main steps. First, for each operation (e.g., deleting an order), BACAgent analyzes which resource is accessed and retrieves the corresponding intended ACP. Second, BACAgent uses data-flow analysis to collect the implemented access control checks along the execution paths from user input (e.g., $_GET in PHP) to the target operation. It then interprets the authorization behavior actually enforced by these checks, such as which role is validated. Finally, BACAgent compares the access control requirements in the ACP with those enforced by the implemented checks. If any inconsistency is identified, BACAgent reports a BAC vulnerability.

Take the first API in Figure 1 as an example. This API allows users to read the `order` resource. BACAgent infers that the intended ACP for this operation is `<Order, Read, User, HAC+VAC>`, indicating that the operation should be restricted to authenticated users who own the target order. However, BACAgent identifies that the implemented access control fails to enforce ownership validation (HAC), resulting in a mismatch between the intended and implemented checks. Consequently, BACAgent reports a broken access control vulnerability.

5 Evaluation

Our evaluation seeks to answer the following research questions:

- RQ1: How effective and efficient is BACAgent at detecting BAC vulnerabilities in real-world applications?
- RQ2: How effective is the inference of ACP?
- RQ3: How does BACAgent perform compared to state-of-the-art approaches?
- RQ4: How do the different components of BACAgent contribute to its success?

5.1 Experimental Setup

Implementation. The implementation of BACAgent can be divided into static analysis and agent framework, each playing a crucial role in enabling accurate BAC vulnerability detection. In total, the entire prototype consists of 10,185 lines of Python and 2,350 lines of CodeQL. Details are presented below.

Static Analysis. The static analysis in BACAgent focuses on two core tasks. First, it locates and extracts evidence from the codebase. For example, it leverages modeled APIs to identify framework-based access controls, and extracts semantic evidence such as API annotations from the code. Second, BACAgent performs data-flow analysis to support tasks such as identifying candidate developer-defined access control checks. To support applications written in different programming languages, we leverage CodeQL [1] for Java and Go, and PHPJoern [30] for PHP.

Agent Framework. BACAgent uses the LangGraph framework [10] to interact with LLMs. To support on-demand analysis, we implement a set of tool functions, e.g., file-reading, SQL parsing, and lightweight LSP-based static analysis [11]. For RAG, BACAgent uses pretrained embedding models [4] to encode framework documentation, which are stored in the Milvus vector database [13] for efficient retrieval. Throughout the analysis process, the LLM's temperature is set to zero to ensure consistent outputs.

Dataset Construction. In total, our dataset consists of 25 popular open-source web applications and 5 industrial web applications, covering three programming languages: PHP, Java, and Go. We provide details about these applications in Table 6 of the Appendix B. Specifically, the construction process is as follows:

Open-source Applications. We collected web applications from popular open-source repositories (e.g., GitHub [21]) according to the following criteria. (1) To ensure dataset reliability, each selected application has been widely evaluated in prior studies [31, 37, 41–43, 48, 53]. (2) To guarantee sufficient popularity and real-world relevance, we require that each application has more than 100 stars on GitHub. (3) To demonstrate that BACAgent is programming language–independent, the selected applications cover multiple programming languages. (4) In addition, to evaluate the recall rate of BACAgent, we prioritize applications with known BAC vulnerabilities, for which proof-of-concept exploits can be obtained from public sources such as NVD [14] or Huntr [8].

Industrial Applications. We collaborated with a world-leading technology company that provides services to billions of users. They generously supplied us with 5 Go-based industrial applications, which contain 29 known BAC vulnerabilities. These applications are also included in our evaluation dataset. Note that the company has a well-established SDL (Secure Development Lifecycle) [17] team. Each application deployed online undergoes multiple rounds of manual inspection, including comprehensive code audits and extensive black-box testing. As a result, discovering previously unknown vulnerabilities in these applications is really challenging.

In all, our dataset includes 10 Java-based applications, 10 PHP-based applications, and 10 Go-based applications. These applications contain 106 known BAC vulnerabilities. We evaluate the precision and recall rate of BACAgent on this dataset to assess its effectiveness in detecting known vulnerabilities and discovering previously unknown ones.

Table 3: The effectiveness of BACAgent in detecting BAC vulnerabilities (RQ1).

LLM Backends	# TP[1]	# FP	# FN	# Prec (%)	# Recall (%)
Qwen3-Plus	157 (92 + 65)	42	14	78.89%	86.79%
GPT-4.1	139 (77 + 62)	43	29	76.37%	72.64%
DeepSeek-V3	148 (85 + 63)	50	21	74.75%	80.19%

[1]The number of known and unknown BAC vulnerabilities discovered by BACAgent. For example, '157 (92 + 65)' means 92 known and 65 unknown vulnerabilities.

5.2 RQ1: Vulnerability Detection

In this part, we evaluate the effectiveness and efficiency of BACAgent in detecting BAC vulnerabilities across the dataset.

Experimental Setup. We run BACAgent on each application in our dataset and report the number of discovered vulnerabilities, along with precision and recall rate. To evaluate efficiency, we also report the average analysis time and token cost. Additionally, to further evaluate the generality of BACAgent across different LLM backends, we repeat the experiments using three LLMs: GPT-4.1 [7], Qwen3-Plus [15], and Deepseek-V3 [3].

Result Overview. Table 6 of Appendix B presents the detailed detection results for each application. Overall, in the best case across different LLMs, BACAgent detected a total of 157 BAC vulnerabilities across the dataset, achieving the highest precision rate of 78.89% and recall rate of 86.79%. Among them, 92 are known BAC vulnerabilities, and 65 are previously unknown ones. These results demonstrate the effectiveness of BACAgent in detecting BAC vulnerabilities in real-world applications. Table 3 further illustrates the detection results under different LLM backends. Among all evaluated models, Qwen3-Plus achieves the best performance. Nevertheless, BACAgent maintains comparable performance when using alternative LLMs, indicating its robustness and practical applicability.

Regarding efficiency, BACAgent requires an average analysis time of 27 minutes per application. In terms of LLM token cost, when using GPT-4.1 as the backend model, BACAgent achieves the lowest token usage, averaging 3,329,973 tokens per application, which corresponds to a cost of $9.98 based on GPT-4.1 pricing [7]. In contrast, when using Qwen as the backend, BACAgent consumes the largest number of tokens, averaging 9,655,788 tokens per application. However, due to the substantially lower token pricing of Qwen [15], the overall cost per application is reduced to only $1.10. These results indicate that BACAgent can effectively detect BAC vulnerabilities with reasonable computational and token costs.

Vulnerability Disclosure. Attackers can exploit these vulnerabilities to leak sensitive user information, delete data, or even hijack payment, thereby severely compromising data integrity and causing substantial financial losses. Therefore, for the vulnerabilities detected in open-source applications, we promptly notified the developers of all affected applications. As of now, we have received 51 CVE identifiers in acknowledgment. For the vulnerabilities detected in industrial applications, we responsibly reported them to the corresponding SDL teams and closely collaborated with them to fix all identified issues.

False Positive Analysis. We further analyzed all false positives reported by BACAgent across three LLMs. Overall, Deepseek-V3 generated 50 false positives and covered all the false positives from the other LLMs, which are mainly caused by three factors. First, 21 false positives stem from *inaccurate ACP inference for public resources*. In practice, certain resources are intentionally designed to be publicly accessible and do not require strict HAC. For example, blog posts may be owned by individual users but are readable by everyone. In these cases, BACAgent may infer that the resource requires ownership checks and thus incorrectly reports BAC vulnerabilities. Although BACAgent can correctly infer ACPs for most public resources, such cases still lead to a small portion of false positives. Second, 14 false positives are caused by *inaccurate identification of existing access control statements*. Some access control checks are implemented in flexible ways (e.g., `lambda` statement [9]) and are difficult to reliably identify by static analysis. When such checks are missed, BACAgent may mistakenly conclude that no access control is enforced and report a vulnerability. Third, 15 false positives are caused by *context distraction during LLM reasoning*. Although existing access control logic is correctly identified, the LLM may fail to consistently retain and incorporate this information when reasoning over long code paths, leading to secure operations being incorrectly flagged as vulnerable.

False Negative Analysis. For 29 false negatives across three LLMs, the primary causes can be attributed to two aspects. First, 11 false negatives result from *inaccurate ACP inference for administrator-only resources*. Some resources are intended to be accessible only to administrators (e.g., system log tables such as `sys_log`), but BACAgent may incorrectly infer them as user-owned resources that only require user-level VAC. As a result, BACAgent may fail to recognize that the endpoint should enforce administrator-level VAC, and thus misses BAC vulnerabilities. Second, 18 false negatives are caused by *early termination of the taint analysis*. Achieving precise inter-procedural taint propagation has long been recognized as a challenging problem in static analysis [46]. Despite extensive prior efforts, indirect calls [39] (e.g., reflection [16]) can still interrupt taint propagation, causing BACAgent to misclassify parameters of sensitive operations as attacker-uncontrollable and consequently miss BAC vulnerabilities.

5.3 RQ2: ACP Inference

In this part, we further evaluate the accuracy of the ACPs inferred by BACAgent and analyze the root causes of incorrect inferences.

Experimental Setup. For each protected resource, BACAgent infers an ACP represented as a four-tuple, which specifies the required access control under different operation contexts. Since accurate ACP inference is fundamental to effective BAC vulnerability detection, we conduct a dedicated evaluation of the inferred ACPs. Specifically, two authors of this paper, each with over five years of experience in system and web security research, independently examined the correctness of the inferred ACP. In cases of disagreement, a third author was involved to conduct further discussion and reach a final consensus.

Result Analysis. In total, BACAgent inferred ACPs for 1,028 resources. To evaluate its precision, we randomly sampled 200 inferred ACPs and found that BACAgent achieves a precision rate

Table 4: Comparison between BACAgent and baselines in BAC vulnerability detection (RQ3).

Baselines	Java					PHP				
	# TP	# FP	# FN	# Prec (%)	# Recall (%)	# TP	# FP	# FN	# Prec (%)	# Recall (%)
PCFinder	36	38	29	48.65%	55.38%	/	/	/	/	/
MOCGuard	30	15	35	66.67%	46.15%	/	/	/	/	/
BOLARay	/	/	/	/	/	19	12	16	61.29%	54.29%
BACAgent	59	22	6	72.84%	90.77%	29	10	6	74.36%	82.86%

of 94.50%. We further conducted an in-depth analysis of the 11 incorrectly inferred cases out of the 200 sampled ACPs and found that they can be categorized into two main causes.

First, 7 cases stem from *insufficient evidence in the code.* Although BACAgent extracts evidence from multiple sources whenever possible, for some resources, the available evidence is still insufficient for the LLM to accurately infer the intended access control policy. For example, the `cms_subject` table is used to display public content and is intended to be accessible by all users. However, despite the provided evidence, BACAgent incorrectly inferred that it should be restricted to administrators, resulting in false positives. Our analysis shows that even experienced security researchers often need to inspect extensive code context and conduct in-depth discussions to determine the correct ACP for such resources. Given the inherent ambiguity of these cases and the trade-off between accuracy and analysis cost, we consider these false positives acceptable at the current stage.

Second, 4 cases are caused by *missed identification of existing access control statements.* Existing access control checks serve as a critical evidence source for ACP inference. Although BACAgent leverages LLM and RAG technique to improve the effectiveness of access control identification, some highly flexible or unconventional checks remain difficult to recognize. This limitation directly leads to incorrect ACP inference in these cases. Importantly, when the missed access control statements are manually provided, BACAgent is able to correctly infer the corresponding ACPs, indicating that the errors stem from access control identification rather than ACP reasoning itself. We leave a more precise and robust identification of access control checks as future work.

5.4 RQ3: Comparison

In this part, we compare BACAgent against three baseline approaches over the entire dataset.

Experimental Setup. We first describe the setup of these three baselines, i.e., PCFinder [49], MOCGuard [41], and BOLARay [37].

- **PCFinder** is an approach for detecting BAC vulnerabilities. It analyzes application code to identify access control checks and detects vulnerabilities through missing-check and inconsistent-check analysis. Since the prototype of PCFinder is designed for Java-based web applications, we apply it only to the Java applications in our dataset.
- **MOCGuard** detects BHAC vulnerabilities by analyzing application source code together with database schemas provided as `.sql` files. It identifies vulnerabilities by reasoning about ownership encoded in the database structure. As MOCGuard is also designed for Java-based web applications, we apply it only to the Java applications in our dataset.
- **BOLARay** is another approach for detecting BHAC. It similarly relies on application source code and `.sql` files for analysis. Since the prototype of BOLARay targets PHP-based web applications, we apply it only to the PHP applications in our dataset.

For the vulnerability ground-truth set, to ensure a fair comparison, we constructed a ground truth aggregating all vulnerabilities detected by both baselines and BACAgent. Ultimately, our ground-truth set consists of 65 verified BAC vulnerabilities in Java and 35 verified BAC vulnerabilities in PHP web Applications.

Result Overview. Table 4 presents a detailed comparison between BACAgent and three baseline approaches across the entire vulnerability dataset. Overall, BACAgent demonstrates superior performance in BAC vulnerability detection, achieving higher precision and recall than all baseline methods. Specifically, PCFinder, MOCGuard, and BOLARay achieve precision rates of 48.65%, 66.67%, and 61.29%, and recall rates of 55.38%, 46.15%, and 54.29%, respectively. These comparison results indicate that BACAgent significantly outperforms existing approaches in accuracy for BAC vulnerability detection.

Result Analysis. We further conduct an in-depth analysis of all false positives and false negatives produced by the three baseline approaches. First, these approaches rely heavily on heuristic rules, which are insufficient for accurately identifying and interpreting access control checks. For example, MOCGuard and BOLARay focus on detecting HAC and are unable to recognize VAC enforced through role checks. PCFinder identifies access control primarily through pattern matching on conditional statements, but lacks the capability to interpret their authorization semantics, such as distinguishing whether a check enforces VAC or HAC. Second, these approaches fundamentally overlook ACP inference and therefore cannot determine what type of access control a resource is intended to enforce. As a result, they apply uniform detection strategies to all resources regardless of their intended access control requirements. These coarse-grained treatments cause the baselines to overestimate the need for access control in some cases and underestimate it in others, ultimately resulting in both false positives and false negatives.

Case Study. We now showcase a BVAC vulnerability that is missed by MOCGuard and PCFinder. As shown in Figure 5, the `delNotice` method provides an interface for administrators to delete system announcements by operating on the `notice` resource. BACAgent accurately infers the intended ACP for the notice resource, which specifies that this operation should only be accessible to administrators. However, although the developer enforces access control

```
1 @GetMapping("/admin/notice/delete")
2 public String delNotice(Integer id, HttpSession session) {
3     if(session.getAttribute("user") == null) {
4         response.sendRedirect("/login");
5         return "/error/403";
6     }
7     noticeService.deleteNotice(id);
8     return "redirect:/admin/notice";
9 }
```

Figure 5: A BVAC vulnerability detected by BACAgent in ForestBlog (4.7k GitHub stars).

checks (lines 3–6), the implementation only verifies whether the user is authenticated, without validating the user's role. As a result, any logged-in user can invoke this operation, leading to a BVAC vulnerability. By inferring the intended ACP, BACAgent precisely identifies this issue as a BVAC case. In contrast, due to its inability to reason about the intended ACP, MOCGuard blindly assumes that an ownership check is required and incorrectly reports this issue as a BHAC vulnerability. PCFinder detects the presence of a VAC but fails to determine which specific role should be checked for the `notice` resource, and therefore misses the vulnerability entirely.

5.5 RQ4: Ablation Study

In this part, we conducted an ablation study to demonstrate the necessity of each key component of BACAgent.

Experimental Setup. In this experiment, we construct a ground-truth vulnerability dataset consisting of 106 known vulnerabilities (as described in §5.1) and 65 0-day vulnerabilities newly discovered by BACAgent. Next, we constructed four variants, each of which disables a key component of BACAgent and uses the rest of the system as is. We then evaluate all variants on the ground-truth vulnerability dataset and compare their effectiveness against BACAgent. The variant construction details are as follows.

- BACAgent-NoRAG. We disable the retrieval-augmented generation (RAG) capability in the *Framework-based Access Control* component. As a result, BACAgent-NoRAG relies solely on the LLM to interpret framework-based access control by reading application code, without grounding its reasoning in framework documentation.
- BACAgent-NoSem. We disable the use of semantic-level evidence in *On-demand ACP Inference* component, including database schemas and API annotations. Therefore, BACAgent-NoSem infers ACPs using only code-level evidence, i.e., the identified access control statements, without additional semantic context.
- BACAgent-NoAC. We disable the use of code-level evidence in the *On-demand ACP Inference* component. In this setting, BACAgent-NoAC infers ACPs solely based on semantic-level evidence, including database schemas and API annotations, without leveraging the identified access control statements.
- BACAgent-NoOnDemand. We disable the on-demand inference technique in the *On-demand ACP Inference* component. Instead of selectively providing evidence to LLM, this variant concatenates all available evidence and provides it to LLM in a single input, thereby illustrating the importance of our on-demand strategy.

Table 5: Ablation study for four variants of BACAgent (RQ4).

Variants	# TP	# FP	# FN	# Prec (%)	# Recall (%)
BACAgent-NoRAG	144	103	27	58.30%	84.21%
BACAgent-NoSem	89	97	82	47.85%	52.05%
BACAgent-NoAC	116	67	55	63.39%	67.84%
BACAgent-NoOnDemand	131	91	40	59.01%	76.61%
BACAgent	157	42	14	78.89%	91.81%

Result Analysis. Table 5 provides a breakdown of the comparison results between BACAgent and its four variants. It is clear that these four key components are essential for effective BAC vulnerability detection. A detailed analysis of the results is as follows:

❶ *BACAgent vs. BACAgent-NoRAG.* BACAgent-NoRAG introduces *61* additional false positives compared to BACAgent. This is primarily because BACAgent-NoRAG lacks access to framework documentation, and thus fails to correctly interpret framework-based (VAC) mechanisms adopted by the target application. As a result, BACAgent-NoRAG frequently misclassifies VACs intended for administrator roles as user-level checks, leading to false positives. This result clearly highlights the critical role of the RAG component in accurately interpreting framework-based access control statements.

❷ *BACAgent vs. BACAgent-NoSem.* As shown in Table 5, compared to BACAgent, BACAgent-NoSem produces additional 55 false positives and 68 false negatives in BAC vulnerability detection. Specifically, without semantic-level evidence from database schemas and API annotations, BACAgent-NoSem lacks an understanding of resource properties and API contexts. Consequently, ACP inference becomes error-prone, which directly degrades the effectiveness of BAC vulnerability detection.

❸ *BACAgent vs. BACAgent-NoAC.* BACAgent-NoAC also performs worse than BACAgent. Specifically, BACAgent-NoAC removes code-level evidence of existing access control from the ACP inference process. As a result, BACAgent-NoAC fails to correctly infer the authorization requirements of resources whose semantics are not explicit from natural-language evidence alone. For example, the `oms_order_return_apply` table in the mall application stores order return information that is intended to be accessible only by administrators. After removing existing access control evidence, BACAgent-NoAC incorrectly infers that this resource can be accessed by normal users, leading to inaccurate detection results.

❹ *BACAgent vs. BACAgent-NoOnDemand.* As shown in Table 5, BACAgent-NoOnDemand performs worse than BACAgent because BACAgent-NoOnDemand eagerly feeds all available evidence into the LLM at once. Although the information is comprehensive, it overwhelms the LLM and causes severe context distraction. As a result, the LLM struggles to focus on the ACP inference task for a specific resource, leading to degraded inference accuracy.

6 Discussion

Mitigation. Our findings indicate that many BAC vulnerabilities stem from inconsistencies between designed access control policies and their concrete implementations. To mitigate such issues, developers should ensure that access control logic is implemented

in strict accordance with the intended policy, covering both vertical and horizontal access control requirements. In practice, this includes consistently enforcing role-based as well as ownership checks for security-sensitive resources. Moreover, maintaining explicit access control policy documentation at the resource level can greatly facilitate both manual review and automated checking. Such documentation enables developers and analysis tools to reason about authorization semantics more precisely, reducing the risk of broken access controls as applications evolve over time.

Limitation and Future Work. While BACAgent demonstrates strong effectiveness in inferring intended access control policies across diverse real-world applications, its accuracy still depends on how access control checks are expressed in the codebase. In practice, some applications implement access control in highly customized or implicit ways, where authorization decisions are deeply intertwined with complex business logic, dispersed across long call chains, or encoded through unconventional programming patterns. In such scenarios, even with LLM-assisted reasoning, BACAgent may fail to fully capture the precise authorization semantics, which can affect both ACP inference and vulnerability detection results. This limitation reflects a broader challenge shared by existing white-box techniques rather than a shortcoming unique to BACAgent. As future work, we plan to incorporate more advanced program analysis techniques, such as deeper inter-procedural reasoning, more precise control- and data-flow analysis, and improved modeling of complex authorization patterns, to further enhance the robustness and accuracy of access control identification and interpretation.

7 Related Work

Broken Access Control Detection. There are numerous works that have explored various techniques for detecting broken access control vulnerabilities in web applications [36, 37, 41, 43, 45, 47, 50, 51, 57]. Many of these approaches [37, 41, 45, 50, 51, 57] rely on language-specific APIs, framework conventions, or software engineering patterns to identify access control checks and detect broken access control through missing or inconsistent authorization logic. However, such techniques typically focus on the presence of access control checks themselves, without reasoning about whether these checks actually enforce the intended ACP for a given resource. Another line of work [36, 43] adopts dynamic analysis to detect access control flaws. While effective in certain scenarios, they heavily depend on a fully configured runtime environment and carefully prepared user accounts, which significantly limit their scalability and applicability to large, real-world systems. Unlike these previous works, BACAgent employs a novel white-box approach to infer ACP and accurately detect BAC vulnerabilities.

Web Vulnerabilities Detection. In recent years, techniques for automatically detecting vulnerabilities in web applications have been extensively studied. A widely adopted approach is static analysis [30, 42, 46, 56], which models user inputs as sources and security-sensitive operations as sinks, and detects vulnerabilities by analyzing data-flow paths between them. These techniques are effective for taint-style vulnerabilities such as SQL injection, but largely operate at the dataflow level and lack the ability to reason about authorization semantics, limiting their applicability to access control vulnerabilities. Another common approach is dynamic analysis [28, 32, 34, 35, 40, 53], which detects vulnerabilities by monitoring runtime behavior using vulnerability-specific oracles. While dynamic techniques detect vulnerabilities by monitoring runtime behavior and checking whether predefined oracles (e.g., exceptions or error responses) are triggered, such oracles are typically designed for taint-style vulnerabilities. In contrast, broken access control vulnerabilities are tied to business logic, where unauthorized operations often execute successfully without triggering any runtime error or exception. This fundamental difference leads to false positives and false negatives when these methods are applied to BAC vulnerability detection.

8 Conclusion

In this paper, we propose BACAgent, a novel white-box approach for detecting Broken-Access-Control (BAC) vulnerabilities in web applications. BACAgent infers intended Access Control Policies by combining behavior-oriented access control analysis with on-demand multi-source reasoning over code-level and semantic-level evidence, enabling precise BAC vulnerability detection. We evaluate BACAgent on 25 widely used open-source applications and 5 industrial applications, discovering 157 BAC vulnerabilities, including 65 previously unknown high-risk 0-day vulnerabilities and the assignment of 51 new CVE IDs. These findings demonstrate the practical utility of BACAgent in BAC vulnerability detection. We hope our work can help the community better understand and address the persistent and growing threats posed by BAC vulnerabilities in real-world web applications.

Ethical Considerations

Vulnerability Verification Environment. All static analyses were performed locally on the collected applications and did not involve any real user data or privacy-sensitive information. Then, after BACAgent reported potential vulnerabilities, we built these applications on a local server for vulnerability verification. This process also did not involve any data related to real users.

Vulnerability Disclosure. In terms of vulnerability disclosure, all our vulnerability reports have strictly adhered to the timeline of the CVE Numbering Authorities (CNA), along with proactive communication with all developers. Specifically, after we manually confirmed the vulnerabilities, we immediately contacted the developers, including raising issues in the Github repository and via email. We carefully explained to the developers the cause of the vulnerability, the details of vulnerability exploitation, and the corresponding recommended solutions for fixing the issues. Although some vulnerabilities were still being fixed at the time we submitted this paper, we did not mention any important information about these vulnerabilities in the paper, and anonymized all vulnerabilities and affected applications. Therefore, the release of this paper will not cause any harm to real-world users.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (U2436207) and the Shanghai Pilot Program for Basic

Research - FuDan University 21TQ1400100 (21TQ012). Yuan Zhang is the corresponding author.

References

[1] [n. d.]. CodeQL on Github. https://codeql.github.com/.
[2] [n. d.]. CWE284. https://cwe.mitre.org/data/definitions/284.html.
[3] [n. d.]. Deepseek-V3. https://api-docs.deepseek.com/news/news250325.
[4] [n. d.]. Embedding Model. https://platform.openai.com/docs/guides/embeddings/embedding-models.
[5] [n. d.]. Filter in Java. https://docs.oracle.com/javaee/7/api/javax/servlet/Filter.html.
[6] [n. d.]. Google Official Website. https://www.google.com/.
[7] [n. d.]. GPT-4.1. https://openai.com/index/gpt-4-1/.
[8] [n. d.]. Huntr platform. https://huntr.com/.
[9] [n. d.]. Lambda Expression in Java. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.
[10] [n. d.]. LangGraph. https://www.langchain.com/langgraph.
[11] [n. d.]. Language Server Protocol. https://microsoft.github.io/language-server-protocol/.
[12] [n. d.]. Middleware in Go. https://gin-gonic.com/en/docs/examples/using-middleware/.
[13] [n. d.]. Milvus. https://github.com/milvus-io/milvus.
[14] [n. d.]. National Vulnerability Database. https://nvd.nist.gov/.
[15] [n. d.]. Qwen3-Plus. https://www.alibabacloud.com/help/en/model-studio/models.
[16] [n. d.]. Reflection in Java. https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/index.html.
[17] [n. d.]. SDL. https://www.microsoft.com/en-us/securityengineering/sdl/practices.
[18] [n. d.]. Shiro Documentation. https://shiro.apache.org/realm.html.
[19] [n. d.]. Spring Security Document. https://spring.io/projects/spring-security.
[20] [n. d.]. Symfony Security Document. https://symfony.com/doc/current/security.html.
[21] [n. d.]. The Official Website of Github. https://github.com/.
[22] [n. d.]. The Official Website of Shiro. http://shiro.apache.org.
[23] [n. d.]. The Source Code of BACAgent. https://github.com/LFYSec/BACAgent.
[24] [n. d.]. Twitter Official Website. https://x.com/.
[25] 2023. Notorious Hacks in History. https://securityboulevard.com/2023/03/23-most-notorious-hacks-history-that-fall-under-owasp-top-10/.
[26] 2023. Serious Data Breach News. https://www.apisec.ai/blog/5-real-world-examples-of-business-logic-vulnerabilities-that-resulted-in-data-breaches.
[27] 2025. OWASP Top 10 - 2025. https://owasp.org/Top10/2025/A01_2025-Broken_Access_Control/.
[28] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. 2018. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *27th USENIX Security Symposium (USENIX Security 18)*.
[29] Brad Arkin, Scott Stender, and Gary McGraw. 2005. Software penetration testing. *IEEE Security & Privacy* 3, 1 (2005), 84–87.
[30] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. [n. d.]. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE european symposium on security and privacy (EuroS&P)*. IEEE.
[31] Miao Chen, Tengfei Tu, Hua Zhang, Qiaoyan Wen, and Weihang Wang. 2022. Jasmine: A Static Analysis Framework for Spring Core Technologies. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.
[32] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the state: A {state-aware}{black-box} web vulnerability scanner. In *21st USENIX Security Symposium (USENIX Security 12)*. 523–538.
[33] Adam Doupé, Marco Cova, and Giovanni Vigna. 2010. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 111–131.
[34] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. 2021. Black widow: Blackbox data-driven web scanning. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1125–1142.
[35] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. 2024. Atropos: Effective fuzzing of web applications for server-side vulnerabilities. In *USENIX Security Symposium*.
[36] Xiangyu Guo, Akshay Kawlay, Eric Liu, and David Lie. [n. d.]. EvoCrawl: Exploring Web Application Code and State using Evolutionary Search. In *the Network and Distributed System Security Symposium (NDSS)*. Accepted for publication. https://security.csl.toronto.edu/wp-content/uploads/2024/09/xguo-ndss2025-evocrawl.pdf
[37] Yongheng Huang, Chenghang Shi, Jie Lu, Haofeng Li, Haining Meng, and Lian Li. 2024. Detecting Broken Object-Level Authorization Vulnerabilities in Database-Backed Applications. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2934–2948.
[38] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.
[39] Guoren Li, Manu Sridharan, and Zhiyun Qian. 2025. Redefining Indirect Call Analysis with KallGraph. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2957–2975.
[40] Zihan Lin, Yuan Zhang, Jiarun Dai, Xinyou Huang, Bocheng Xiang, Guangliang Yang, Letian Yuan, Lei Zhang, Tian Chen, and Min Yang. 2025. Effective directed fuzzing with hierarchical scheduling for web vulnerability detection. In *34th USENIX Security Symposium (USENIX Security 25)*. 8349–8366.
[41] Fengyu Liu, Youkun Shi, Yuan Zhang, Guangliang Yang, Enhao Li, and Min Yang. 2024. MOCGuard: Automatically Detecting Missing-Owner-Check Vulnerabilities in Java Web Applications. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 10–10.
[42] Fengyu Liu, Yuan Zhang, Tian Chen, Youkun Shi, Guangliang Yang, Zihan Lin, Min Yang, Junyao He, and Qi Li. 2025. Detecting Taint-Style Vulnerabilities in Microservice-Structured Web Applications. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 972–990.
[43] Fengyu Liu, Yuan Zhang, Enhao Li, Wei Meng, Youkun Shi, Qianheng Wang, Chenlin Wang, Zihan Lin, and Min Yang. 2025. BACScan: Automatic Black-Box Detection of Broken-Access-Control Vulnerabilities in Web Applications. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. 1320–1333.
[44] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.
[45] Jie Lu, Haofeng Li, Chen Liu, Lian Li, and Kun Cheng. 2022. Detecting Missing-Permission-Check Vulnerabilities in Distributed Cloud Systems. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*.
[46] Changhua Luo, Penghui Li, and Wei Meng. 2022. TChecker: Precise Static Inter-Procedural Analysis for Detecting Taint-Style Vulnerabilities in PHP Applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*.
[47] Maliheh Monshizadeh, Prasad Naldurg, and VN Venkatakrishnan. 2014. MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 690–701.
[48] Yicheng Ouyang, Kailai Shao, Kunqiu Chen, Ruobing Shen, Chao Chen, Mingze Xu, Yuqun Zhang, and Lingming Zhang. 2023. MirrorTaint: Practical Non-intrusive Dynamic Taint Tracking for JVM-based Microservice Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2514–2526.
[49] Youkun Shi, Fengyu Liu, Guangliang Yang, Yuan Zhang, Yinzhi Cao, Enhao Li, Xin Tan, Xiapu Luo, Min Yang, and Siyi Chen. 2025. Facilitating Access Control Vulnerability Detection in Modern Java Web Applications With Accurate Permission Check Identification. *IEEE Transactions on Information Forensics and Security* (2025).
[50] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. 2011. Rolecast: Finding Missing Security Checks When You Do Not Know What Checks Are. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 1069–1084.
[51] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. 2013. Fix Me Up: Repairing Access-Control Bugs in Web Applications.. In *NDSS*. Citeseer.
[52] Aleksei Stafeev, Tim Recktenwald, Gianluca De Stefano, Soheil Khodayari, and Giancarlo Pellegrino. 2025. YuraScanner: Leveraging LLMs for Task-driven Web App Scanning.. In *NDSS*.
[53] 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)*.
[54] Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 146–158.
[55] Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2024. Coderag-bench: Can retrieval augment code generation? *arXiv preprint arXiv:2406.14497* (2024).
[56] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*.
[57] Qiyi Zhang, Fengyu Liu, Zihan Lin, and Yuan Zhang. 2025. Be Aware of What You Let Pass: Demystifying URL-based Authentication Bypass Vulnerability in Java Web Applications. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. 1874–1888.

A Open Science

In alignment with the open science policy, we are committed to fully following the conference's artifact evaluation guidelines. We publicly release the code of BACAgent, along with the datasets used for evaluation in our research [23].

B Dataset

Table 6 presents the breakdown of our evaluation dataset, including application names, popularity (i.e., GitHub stars), lines of code (LoC), the number of known and unknown BAC vulnerabilities, and the effectiveness of BACAgent.

Table 6: Breakdown of our evaluation dataset.

Open-source	Stars	LoCs	Known Vuln.	Detected Vuln. (Known / Unknown)	Prec (%)	Recall (%)	Language
mall	80,981	68,681	2	2 / 3	62.50%	100.00%	Java
platform	20,340	26,466	4	4 / 1	71.43%	100.00%	Java
mall-swarm	11,988	65,681	1	1 / 4	83.33%	100.00%	Java
wechat-applet	11,871	32,215	10	8 / 1	69.23%	80.00%	Java
newbee-mall	10,972	4,937	4	4 / 1	71.43%	100.00%	Java
xmall	7,135	27,806	6	5 / 1	75.00%	83.33%	Java
novel	5,364	7,067	1	0 / 1	100.00%	0.00%	Java
forestblog	4,697	5,435	0	0 / 2	50.00%	-	Java
supermarket	2,084	3,268	8	8 / 1	90.00%	100.00%	Java
icecms	1,885	7,410	9	7 / 5	70.59%	77.78%	Java
gin-vue-admin	22,565	32,511	1	0 / 9	90.00%	0.00%	Go
go-admin	11,836	15,326	0	0 / 10	71.43%	-	Go
paopao-ce	4,461	21,168	4	4 / 1	100.00%	100.00%	Go
gin-mall	674	4,807	0	0 / 3	100.00%	-	Go
newbee-mall-api-go	573	4,555	0	0 / 8	80.00%	-	Go
prestashop	8,892	449,107	1	1 / 0	100.00%	100.00%	PHP
invoiceninja	8,192	206,003	3	2 / 1	100.00%	66.67%	PHP
openemr	4,587	558,321	4	3 / 3	85.71%	75.00%	PHP
drupal	4,229	582,273	2	1 / 0	20.00%	50.00%	PHP
piwigo	3,656	179,097	1	1 / 0	100.00%	100.00%	PHP
microweber	3,373	144,463	1	1 / 0	100.00%	100.00%	PHP
wordpress	3,188	10,934	9	8 / 1	75.00%	88.89%	PHP
mantisbt	1,741	78,600	1	0 / 0	0.00%	0.00%	PHP
zencart	415	308,339	0	0 / 1	33.33%	-	PHP
webid	119	35,872	5	4 / 2	100.00%	80.00%	PHP
Industrial	**Stars**	**LoCs**	**Known Vuln.**	**Detected Vuln. (Known / Unknown)**	**Prec (%)**	**Recall (%)**	**Language**
a****_********	/	70,628	7	7 / 0	87.50%	100.00%	Go
a*_*****	/	41,778	3	3 / 2	100.00%	100.00%	Go
d*****_*****_***	/	2,740	4	4 / 0	100.00%	100.00%	Go
o**_******	/	223,323	14	13 / 4	89.47%	92.86%	Go
s*****_***	/	29,581	1	1 / 0	100.00%	100.00%	Go
Total	**/**	**/**	**106**	**92 / 65**	**78.89%**	**86.79%**	/