# BACScan: Automatic Black-Box 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

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

Wei Meng
The Chinese University of Hong Kong
Hong Kong SAR, China
wei@cse.cuhk.edu.hk

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

Qianheng Wang
Fudan University
Shanghai, China
21307130075@m.fudan.edu.cn

Chenlin Wang
The Chinese University of Hong Kong
Hong Kong SAR, China
clwang23@cse.cuhk.edu.hk

Zihan Lin
Fudan University
Shanghai, China
zhlin22@m.fudan.edu.cn

Min Yang
Fudan University
Shanghai, China
m_yang@fudan.edu.cn

## Abstract

Broken-Access-Control (BAC) vulnerabilities have consistently been ranked among the most critical security risks in web applications, occupying the top positions in the OWASP Top 10 over the past several years. These vulnerabilities allow attackers to bypass access control mechanisms and perform unauthorized operations, posing security and privacy threats to sensitive business and user data. Despite substantial attention given to BAC vulnerabilities, effective and reliable approaches to detecting these issues remain limited.

In this work, we present BACScan, a novel black-box approach to detect BAC vulnerabilities in web applications. Unlike existing response similarity-based oracles that check only unauthorized read accesses, BACScan introduces an innovative feedback-driven oracle, which determines whether unauthorized read or modification operations have occurred by inferring operationally-dependent web pages and analyzing the operational feedback. We evaluated BACScan on 20 real-world applications and successfully identified 89 vulnerabilities, including 54 previously unreported ones, outperforming state-of-the-art tools. We reported all newly identified vulnerabilities to the affected vendors. To date, 35 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, 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 (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3719027.3744825

## 1 Introduction

*Broken-Access-Control moves up to the category with the most serious web application security risk.*

*— The OWASP Top 10 [19].*

With the rapid advancement of web applications, numerous commercial platforms (e.g., Amazon [1], PayPal [9]) now store substantial volumes of critical privacy-sensitive user data, including identity and payment information, thereby making them attractive targets for web attackers. To protect this sensitive data, developers implement and deploy access control mechanisms to prevent unauthorized operations. However, inadequate or improperly configured access control mechanisms can give rise to serious vulnerabilities, known as *Broken-Access-Control* (BAC) vulnerabilities.

In recent years, BAC vulnerabilities have become increasingly severe. According to the OWASP Top 10, BAC vulnerabilities have consistently ranked among the most critical issues over the past five years [18, 19, 21]. In OWASP 2023, BAC vulnerabilities even accounted for four out of the five most severe vulnerabilities [21]. Furthermore, numerous widely-used applications, including PayPal, Twitter, and TikTok, have been reported to exhibit BAC vulnerabilities, as documented in HackerOne [15, 16]. Attackers may exploit these vulnerabilities to perform unauthorized operations, including arbitrary read, modification, or even deletion of sensitive data, thereby posing significant threats to the availability and integrity of the vulnerable applications [20, 22].

Although BAC vulnerability detection is a well-explored research topic, the increasing severity of BAC vulnerabilities highlights the long-lasting lack of highly effective detection techniques. Recent detection approaches can generally be categorized into white-box and black-box methods. White-box approaches exhibit significant limitations, including high false positive rates, the inability to generate proof-of-concept (PoC), and scalability issues associated with static analysis techniques [45]. To address these limitations, researchers have explored dynamic black-box scanning techniques (i.e., scanners), which are widely used methods for penetration testing and are particularly effective in detecting security flaws in real-world scenarios [24, 30]. Given these advantages, we adopted a black-box approach for detecting BAC vulnerabilities and conducted a thorough study of existing black-box techniques. Specifically, these techniques [28, 35, 37, 49, 50, 62] rely on a *response similarity-based oracle* (shortened to response-based oracle) for detecting BAC vulnerabilities. In particular, the security analysts set up an attacker and a victim user accounts with the scanners. The scanner uses the attacker's credentials to send HTTP requests to access the victim's personal data. If the response received by the attacker is similar to the victim's, a BAC vulnerability is reported.

However, the design of this oracle exhibits significant inherent flaws. Oracle is a crucial and fundamental component of black-box scanners, playing a pivotal role in determining the effectiveness of black-box vulnerability detection [32]. Unfortunately, the response-based oracle is based on a flawed assumption: the *direct response* to a BAC attack request always provides evidence indicating whether the exploit is successful or not. This assumption holds true in scenarios like unauthorized data leakage via Read-based Broken Access Control (RBAC) vulnerabilities, where the leaked data is directly returned to the attacker via the HTTP response. However, in other scenarios such as unauthorized data deletion via Modification-based Broken Access Control (MBAC) vulnerabilities, this assumption fails as the direct HTTP response to the attack request does not necessarily indicate whether the unauthorized operation is successful. This erroneous oracle leads to a significant rate of false negatives and false positives in the detection of MBAC vulnerabilities (70.97% and 86.96% in our evaluation, respectively).

In this paper, we propose BACScan, a novel black-box approach to detecting BAC vulnerabilities in web applications. Specifically, BACScan employs the state-of-the-art (SoTA) response-based oracle for detecting RBAC vulnerabilities where the direct HTTP response provides feedback. We design a novel feedback-driven oracle for MBAC vulnerabilities where the direct HTTP response cannot reflect the modification status. It determines whether unauthorized modifications have succeeded by observing the feedback from other operationally-dependent pages that indicate the modification status. While the idea is intuitively simple, effectively implementing and applying this feedback-driven oracle for MBAC vulnerability detection presents significant challenges. To achieve this, two main challenges must be carefully addressed:

- *C1: How to accurately and efficiently identify status pages that provide feedback in a black-box context?* Accurately identifying the corresponding status pages that provide feedback for modification pages is critical for vulnerability detection. However, in a black-box scenario, only HTTP requests and responses are available. Given the vast number of requests and responses in modern web applications, accurately and efficiently pinpointing these operationally-dependent status pages is non-trivial.
- *C2: How to navigate the application into the correct state to collect effective feedback for MBAC vulnerability detection?* Once the status page providing feedback is identified, the next step is to replay modification requests using an attacker's session and verify if unauthorized changes to data occur. However, the target application may not be in the right state, and issues like attempts to delete non-existent data may cause operations to fail, resulting in failed request replays and affecting detection accuracy.

Inspired by Black Widow [32], we observe that the data access and operational dependency between web pages can help us identify the corresponding status pages of the modification page. The status pages could provide feedback for detecting unauthorized modifications. Therefore, BACScan introduces a novel data structure to represent the data dependency relationship, called the Inter-page Data Dependency Graph (IDDG). Specifically, it utilizes SoTA crawling techniques with the victim user's credentials to construct the IDDG based on the navigation graph. The process involves intercepting requests to modification pages, generating and inserting unique tokens, and leveraging the hierarchical strategy to efficiently traverse the navigation graph to establish data dependencies between modification and status pages. Then, BACScan leverages the constructed IDDG to apply the novel feedback-driven oracle to detect MBAC vulnerabilities. Specifically, it follows the navigation edge to replay modification requests with the attacker's credentials and follows the data dependency edge to revisit corresponding status pages to obtain feedback. It then detects potential MBAC vulnerabilities based on the presence or absence of unique tokens.

We evaluate the effectiveness and performance of BACScan on 6 open-source web applications with 44 known BAC vulnerabilities and 14 applications with no known vulnerabilities. These applications have been widely evaluated in previous studies [26, 34, 48, 56, 58] and each has at least 100 stars on GitHub, demonstrating their representativeness and popularity. Moreover, they are implemented in various languages, including Java, PHP, and Go, further demonstrating the language-independent advantage of BACScan. As a result, BACScan successfully discovered 89 vulnerabilities, including 54 (verified) high-risk 0-day and 35 known BAC vulnerabilities. Besides, we compare BACScan to state-of-the-art techniques (i.e., BurpSuite [12] and EvoCrawl [35]). BACScan performs significantly better, detecting up to 49 additional vulnerabilities. These newly discovered BAC vulnerabilities have the potential to leak sensitive user information, delete or modify critical operational data and records (e.g., changing a patient's prescription), posing serious security and privacy threats to individual users and business operations. Given their significant security impact, we responsibly reported all the newly detected vulnerabilities to the relevant developers. At the time of writing, 39 vulnerabilities have been confirmed by the developers, and 35 have been assigned new CVE IDs.

In summary, our paper makes the following contributions:

- We design a new feedback-driven oracle that leverages inter-page data dependency for black-box detection of BAC vulnerabilities.
- We propose a novel black-box detection approach—BACScan—to effectively detect BAC vulnerabilities in web applications. We will open-source our prototype upon publication.
- Our evaluation with 20 real-world popular web applications demonstrates the effectiveness of BACScan, with the discovery

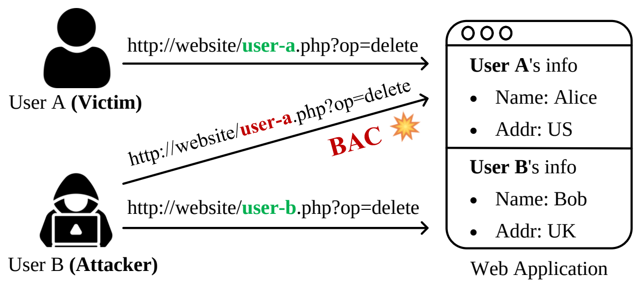


**Figure 1: Threat Model of BAC Vulnerability.**

of 54 (verified) 0-day BAC vulnerabilities and the assignment of 35 new CVE IDs.

## 2 Problem Statement

### 2.1 Broken-Access-Control Vulnerability

Web applications store substantial amounts of sensitive user data, such as identity and payment information, which is protected by applications' access control mechanisms. However, misconfigurations in these mechanisms would result in *Broken-Access-Control* (BAC) vulnerabilities [18], including issues like CWE-284 (Improper Access Control) [4]. These vulnerabilities allow attackers to bypass access controls and perform unauthorized operations, such as reading, modifying, or deleting sensitive data of other users, posing risks to the application's confidentiality, integrity, and availability [20, 22]. For example, as shown in Figure 1, although both users should only have access to their own data, User B (the *attacker*) can exploit the vulnerability to delete the data of User A (the *victim*) without authorization.

### 2.2 Existing Detection Techniques

Existing white-box approaches [43, 44, 46, 52–54] utilize various application-specific inputs (e.g., runtime logs and code annotations) to identify access control checks within applications, treating inadequately safeguarded security-sensitive operations as BAC vulnerabilities. While the high-level idea is reasonable, these approaches still present notable limitations due to high false positives, inability to generate PoCs, and reliance on specific programming languages.

Recently, numerous studies have explored black-box techniques for detecting BAC vulnerabilities. Black-box scanning is widely employed in security penetration testing. This approach uses scanners to simulate real-world attacks from an external adversary's perspective, where the attacker lacks prior knowledge of the system's internal logic and implementation but can observe the system's external behavior. Black-box techniques are particularly effective in detecting security flaws in live or production environments due to their low reliance on source code, significantly increasing their practicality in real-world scenarios [24, 30]. Given these advantages, we decided to adopt a black-box approach for detecting BAC vulnerabilities and conducted a thorough study of existing techniques.

Many works (e.g., [28, 35, 37, 49, 50, 62]) introduced automated black-box scanners for BAC vulnerability detection. Specifically, they typically adopt a *response similarity-based oracle* and employ a three-step approach. First, they configure two users of the target application for the scanner, with one acting as an attacker and the other serving as a victim user. Second, the scanner utilizes both the victim's and the attacker's sessions to explore the web application, separately collecting the pages accessible to each, along with the corresponding page content (i.e., HTTP responses) for each user. Third, the scanner filters out public pages accessible to both the attacker and the victim and selects the pages that are only accessible to the victim. It then replaces the victim's session with the attacker's, keeping all other request parameters identical, and revisits the pages that are exclusively accessible to the victim. If the content for a given page is highly similar between the attacker and the victim, the scanner reports a potential BAC vulnerability.

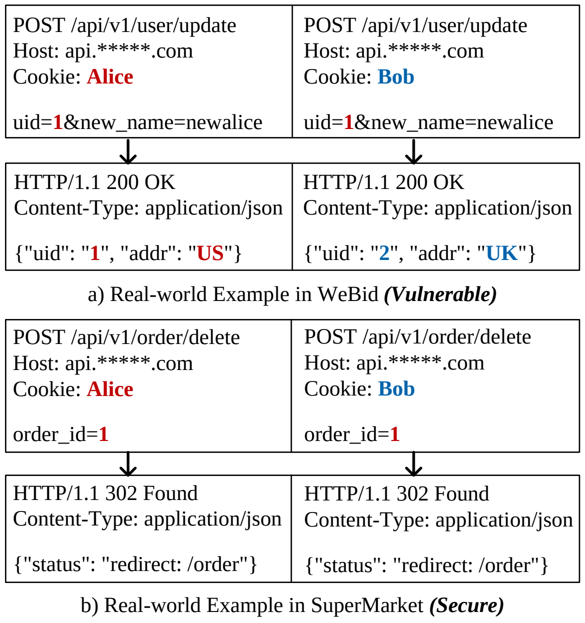
### 2.3 Limitations of Existing Black-box Scanner

Undoubtedly, the oracle is a crucial and fundamental component of black-box scanners, playing a pivotal role in determining the effectiveness of vulnerability detection [32, 34]. At first glance, the response-based oracle appears intuitively reasonable and well-suited for BAC vulnerability detection. However, upon an in-depth analysis of the oracle's underlying mechanism, we observe that it is predicated on a flawed assumption: *the direct response to a BAC attack always contains evidence indicating whether the exploit is successful or not.* This assumption significantly limits the scanner's applicability to only a narrow subset of BAC vulnerabilities.

Specifically, in certain scenarios, this assumption holds true. For example, the presence of a victim's order details within the attacker's HTTP response clearly indicates a BAC vulnerability. Nevertheless, in other scenarios, this assumption fails and results in significant issues such as false negatives and false positives. To provide a more clear illustration, we classify BAC vulnerabilities based on their impact on data integrity and organize them into two distinct categories: Read-based Broken-Access-Control (i.e., RBAC) and Modification-based Broken-Access-Control (i.e., MBAC).

❶ RBAC vulnerabilities are characterized by read-only operations (e.g., `SELECT`) through which attackers can achieve unauthorized read access to data without modifying it. The direct response-based oracle typically works well for detecting RBAC vulnerability, as the goal of exploiting RBAC is to leak sensitive data. Consequently, the accessed private data is directly leaked to the attacker via the HTTP response, which serves as an indicator of whether unauthorized data access has occurred.

❷ MBAC vulnerabilities, on the other hand, involve modification operations (e.g., `INSERT`, `UPDATE`, and `DELETE`) and represent a significant portion of BAC vulnerabilities. Given that MBAC can severely compromise data integrity (e.g., deleting critical private user data), its potential impact is much more severe than that of RBAC. However, for MBAC vulnerabilitis, the direct response-based oracle is generally ineffective. In some cases, the status of data modification is present in the direct HTTP response, indicating whether the unauthorized modification is successful (e.g., `{"status":"success"}` in HTTP response). Nevertheless, in most cases, the status is not available in direct response to modification requests. This is due to the intrinsic nature of MBAC, which primarily focuses on data modification. The modification status might not be observable in the direct response to the user and be present in other dependent responses. Consequently, treating the direct response as an indicator of modification status (i.e., response-based oracle) for detecting MBAC vulnerability leads to a significant number of false positives



```
POST /api/v1/user/update
Host: api.*****.com
Cookie: Alice

uid=1&new_name=newalice
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{"uid": "1", "addr": "US"}
```

```
POST /api/v1/user/update
Host: api.*****.com
Cookie: Bob

uid=1&new_name=newalice
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{"uid": "2", "addr": "UK"}
```

a) Real-world Example in WeBid ***(Vulnerable)***

```
POST /api/v1/order/delete
Host: api.*****.com
Cookie: Alice

order_id=1
```

```
HTTP/1.1 302 Found
Content-Type: application/json

{"status": "redirect: /order"}
```

```
POST /api/v1/order/delete
Host: api.*****.com
Cookie: Bob

order_id=1
```

```
HTTP/1.1 302 Found
Content-Type: application/json

{"status": "redirect: /order"}
```

b) Real-world Example in SuperMarket ***(Secure)***

**Figure 2: Real-world BAC Vulnerability Examples.**

and false negatives. As demonstrated in §5.3, our evaluation reveals that the false positive and false negative rates can reach as high as 70.97% and 86.96%, respectively. Therefore, it is crucial to identify the correct dependent indicators of modification status to accurately detect MBAC vulnerabilities.

## 2.4 Real-World BAC Example

To highlight the limitations of existing work, we present two real-world examples illustrated in Figure 2.

Figure 2 (a) illustrates a 0-day MBAC vulnerability within WeBid [17]. The response-based oracle is ineffective in detecting this vulnerability, resulting in a false negative. Specifically, user Alice (the victim) sends a request to the `/api/vi/user/update` page to update her username. The parameters `uid` and `new_name` represent Alice's user ID and the new username, respectively. User Bob (the attacker) replaces the cookie with his own while leaving the rest of the request unchanged, and then replays the request. Since the developer failed to implement an appropriate access control policy to verify the relationship between the currently logged-in user and the value of the `uid` parameter, Bob can modify Alice's username without authorization, exposing an MBAC vulnerability. However, the response to this modification request does not contain any indicators showing whether the update is successful. The application only returns the information related to the user who makes the request, such as the uid and address. Consequently, even though the attacker successfully modified the victim's data without authorization, the response-based oracle incorrectly concludes that no vulnerability exists due to the dissimilarity between the attacker and victim responses, ultimately leading to a false negative.

Figure 2 (b) illustrates a false positive caused by the response-based oracle in SuperMarket [11]. In this scenario, user Alice deletes her own order by sending a request to the `/api/v1/order/delete` page. User Bob, acting as the attacker, replicates this request but replaces the cookie with his own before replaying it. Due to the developer having implemented proper access control measures, Bob cannot delete Alice's order without authorization, meaning no MBAC vulnerability actually exists. Nevertheless, since the responses to both requests are highly similar, the response-based oracle erroneously concludes that a vulnerability is present, resulting in a false positive.

## 3 Overview of Our Method

Although the response-based oracle is effective for RBAC, it proves fundamentally unsuitable for detecting MBAC vulnerabilities due to the intrinsic differences in their operational characteristics. Therefore, in this work, rather than relying on the traditional response-based oracle, we propose a novel oracle for effective black-box MBAC vulnerability detection. In this section, we first introduce the design of our new oracle (in §3.1), then discuss the challenges encountered (in §3.2), and finally present our solution (in §3.3).

### 3.1 MBAC Detection Oracle

Given that attackers aim to modify data by exploiting MBAC vulnerabilities, detecting MBAC vulnerabilities hinges on determining whether an unauthorized data modification is successful. Building on this understanding, the key insight behind our approach is that *the data altered by a modification page can be displayed on a status page*. In other words, there exists a status page that provides modification feedback in most cases, although it can be one different from the modification page. This makes sense because web applications would usually present user-specific data that can be modified through multiple channels on some dedicated pages. Such pages, which are used to view data, can therefore function as status pages providing feedback. For example, consider an application that has two pages: an order submission page and an order list page. When a user submits an order through the order submission page, the details of that order can be displayed on the order list page. By examining the order list page, we can determine whether the order submission is successful, as it displays the data regarding the submitted order. If the data shown on the order list page changes unexpectedly or is modified by another user without the original user's consent, it becomes apparent that unauthorized modifications have occurred. Thus, the order list page serves not only as a display for submitted orders but also as a crucial indicator for detecting any unauthorized data manipulation on the order submission page.

Therefore, we design a *feedback-driven oracle* for MBAC vulnerability detection. Specifically, the intuitive detection workflow can be outlined in the following two steps: (1) we crawl and explore the application with the victim's credential, analyze the HTTP requests and responses, and identify the status page that displays the data related to the modification request. (2) we replay the modification request using the attacker's credential and revisit the status page to obtain feedback, which indicates whether the modification is successful. By leveraging this feedback-driven oracle, we can confirm the presence of an MBAC vulnerability if the data has been altered without authorization.

## 3.2 Challenges

This feedback-driven oracle and detection workflow may appear straightforward. However, implementing this oracle and effectively applying it for MBAC vulnerability detection presents significant difficulties. Two main challenges must be addressed carefully.

**Challenge I: How to accurately and efficiently identify status pages that provide feedback in a black-box context?** Accurately identifying the corresponding status pages that provide feedback for a modification request is the first crucial step. Any errors or oversights in this process can directly lead to false positives or false negatives during vulnerability detection. However, this is not a trivial task. As demonstrated, we cannot access the source code or query the database in a black-box context. The only information available comes from HTTP requests and their corresponding responses. In web applications, there are typically thousands of such requests and responses. Manually analyzing each one or exhaustively traversing them would be highly time-consuming. Therefore, accurately and efficiently identifying the corresponding status pages that provide feedback from a large volume of requests and responses is undoubtedly a challenging task.

**Challenge II: How to navigate the application into the correct state to collect effective feedback for MBAC vulnerability detection?** After identifying the status page that provides feedback, the next step involves replacing the session with those of an attacker, replaying the modification requests to the target page, and revisiting the corresponding status page to verify whether unauthorized modifications to the victim's data have occurred. However, several critical issues must be addressed to ensure the application is in the correct state, allowing for the successful replay of modification requests. For instance, during data dependency construction, DELETE-type operations may permanently remove certain data, making it unmodifiable. In this case, when the scanner replays the delete request to detect vulnerabilities, the data to be deleted no longer exists, causing the replayed request to fail and resulting in false negatives. Furthermore, modification requests may contain parameters such as CSRF tokens, which could cause the replayed request to fail, thus impacting the accuracy of vulnerability detection. Consequently, it is essential to interact with the application in the correct state, thereby collecting effective feedback for MBAC vulnerability detection.

## 3.3 Our Solution

To address these key challenges, we propose our novel solutions, which consist of two key techniques.

**Technique I: Inter-page Data Dependency Construction.** Inspired by Black Widow [32], we observe that *there exists a data dependency between the data manipulated by the modification page and the data displayed by the status page.* This dependency helps identify the status page that indicates the state of the modified data, thus acting as feedback for detecting unauthorized modifications. To establish such dependency, we introduce a novel graph data structure called Inter-page Data Dependency Graph (IDDG), which connects modification pages and their corresponding status pages through data dependency edges (detailed in §4.1).

**Algorithm 1:** IDDG Construction

```
  Input: Initial Page ($P_0$), Victim's Session $VS$
  Output: IDDE Set ($E$)
1 $E \leftarrow \emptyset$
2 foreach $p \in$ ExplorePage($P_0, VS$) do
3     $p \leftarrow$ InterceptRequest($p$)
4     if Type($p$) $\in$ {POST, PUT, DELETE} then
5         $token \leftarrow$ GenerateToken($p$)
6         $p_{\mathrm{m}} \leftarrow$ ReplaceParams($p, token$)
7         ReleaseRequest($p_{\mathrm{m}}$)
8     end
9     foreach $p_r \in$ HierarchicalTraverse($p$) do
10        if CheckToken($p_r, token$) then
11            $E \leftarrow E \cup \{(p_{\mathrm{r}}, p_{\mathrm{m}})_{\mathrm{Type}}\}$
12        end
13    end
14 end
```

Specifically, we demonstrate the IDDG construction process through Algorithm 1: ① Our approach utilizes SoTA crawling techniques [32] to explore the web pages of the target application, using both the victim's and attacker's sessions to construct separate navigation graphs for each user. We then analyze the URLs and contents of all pages, filtering out public pages that are accessible to both attackers and victims. ② During the victim's page exploration (line 2), our approach intercepts requests to modification pages only accessible to the victim, replaces parameter values with randomly generated tokens, and then releases the request (lines 3-8). ③ Next, we employ a hierarchical strategy to traverse the navigation graph, prioritizing in pages that are more likely to serve as feedback (line 9). Notably, our approach stops graph traversal once the first status page capable of providing feedback is identified, thus improving efficiency. ④ Our approach revisits the status pages to record any tokens. When a token inserted by a modification page appears (for INSERT and UPDATE) or disappears (for DELETE) on a status page, a data-dependency edge is established between the status and modification pages, along with its corresponding operation type, i.e., INSERT, UPDATE, or DELETE (lines 10-12). By following these steps, our approach can accurately represent the inter-page data dependency relationships within the IDDG.

**Technique II: IDDG-based MBAC Vulnerability Detection.** Utilizing the constructed IDDG, our approach applies the novel feedback-driven oracle to detect MBAC vulnerabilities. The workflow for this phase is shown in Algorithm 2. ⑤ We attempt to replay the victim requests sent to modification pages in the IDDG using the attacker's session. For INSERT- and UPDATE-type modification pages, we directly replay the request (lines 3-5). For DELETE-type modification pages, unlike other types, we first locate the page that originally inserted the token into the current delete page, and then replay the insertion operation to insert the token back onto the page (lines 6-10). ⑥ Next, leveraging the inter-page data dependency relationships recorded in the IDDG, we follow the data dependency edge to locate and revisit the corresponding status page using the victim's session, detecting vulnerabilities by obtaining

**Algorithm 2:** MBAC Vulnerability Detection

```
Input: IDDG $G$, Attacker's Session $AS$, Victim's Session $VS$,
Output: Vulnerabilities $V$
1  $V \leftarrow \emptyset$
2  foreach $p_m \in G$ do
3      if InsertPage($p_m$) or UpdatePage($p_m$) then
4          ReplayReq($p_\mathrm{m}, AS$)
5      end
6      if DeletePage($p_m$) then
7          $token \leftarrow$ GenerateToken($p$)
8          ReInsert($p_\mathrm{m}, token, VS$)
9          ReplayReq($p_\mathrm{m}, AS$)
10     end
11     $p_\mathrm{r} \leftarrow$ GetDependencyNode($p_\mathrm{m}, G$)
12     if CheckVuln($p_r, token, VS$) then
13         $V \leftarrow V \cup p_\mathrm{m}$
14     end
15 end
```

feedback. Specifically, we designed distinct oracles for each type of modification page (see §4.2). For INSERT and UPDATE types, the appearance of a new token on the status page indicates an MBAC vulnerability. For DELETE, the absence of a previously existing token signals an MBAC vulnerability (lines 11-14).

## 4 BACScan

In this section, we provide the design details of our BAC (including MBAC and RBAC) vulnerability detection approach, named BACScan. Figure 3 illustrates the architecture of BACScan, which consists of two key modules.

- *IDDG Construction (§4.1).* This module constructs the IDDG to represent the data dependency relationships between web pages, thereby facilitating vulnerability detection.
- *BAC Vulnerability Detection (§4.2).* This module initiates HTTP requests and leverages the feedback-driven oracle and response-based oracle to detect both MBAC and RBAC vulnerabilities.

### 4.1 IDDG Construction

In this section, we first present the formal definition of the IDDG, followed by a detailed explanation of how BACScan constructs the IDDG using one example.

*4.1.1 IDDG Definition.* The IDDG is constructed on top of the navigation graph and represents the data dependency relationship between pages through special nodes and edges. We introduce the structure of the navigation graph and utilize graph notation to rigorously define the formal representation of the IDDG.

**Navigation Graph Definition.** Navigation Graph is a data structure widely used in traditional tasks such as web vulnerability scanning [29, 31, 32]. It represents how pages within a web application are accessible from one to another through navigation edges. Following existing work, we introduce the key components of the navigation graph, i.e., the nodes and edges. The nodes represent individual web pages, such as the login page. Web crawlers typically collect newly accessible web pages by exploring already visited pages. The edges capture the relationships between page nodes, indicating how one page can navigate to another, which may occur through hyperlinks, event handling, form submissions, etc.

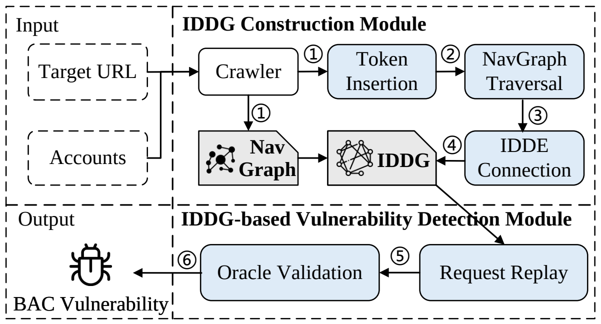


**Figure 3: Architecture of BACScan.**

We use $n$ to represent the nodes in the navigation graph, and $NE$ to represent the set of navigation edges. The formal definition of $NE$ is as follows:

$$NE = \{(n_\mathrm{prev}, n_\mathrm{next}) \mid Navigate(n_\mathrm{prev}) = n_\mathrm{next}\}$$

**IDDG Definition.** The IDDG is constructed on top of the navigation graph. IDDG represents the data dependency relationships between web pages by connecting page nodes through a new type of directed edge—inter-page data dependency edge (IDDE). Specifically, an IDDE connects from a modification page node to its corresponding status page node. The modification pages process requests that alter the data, e.g., submitting orders. The status pages process requests that retrieve data and display this data to end users, e.g., listing orders. The IDDE indicates that data modified by the modification page can be displayed on the connected status page. Besides, the IDDE records the modification type performed on the data (i.e., INSERT, UPDATE, and DELETE), which facilitates further analysis of data dependencies. The formal definition of an IDDE is as follows:

$$IDDE = \{(n_\mathrm{mod}, n_\mathrm{status}) \mid Dependency(n_\mathrm{mod}) = n_\mathrm{status}\}$$

In this paper, consistent with Black Widow [32], we treat data read pages that process GET requests as status pages and pages that process POST, PUT, or DELETE requests as modification pages. While this assumption is not always true, it is generally accepted in the context of web applications. As defined in the HTTP RFC [33], GET requests are idempotent and do not modify data, making them suitable for categorization as data read operations. Besides, our evaluation results in §5.5 demonstrate that this assumption has no significant impact on the accuracy of the IDDG or the effectiveness of vulnerability detection. We discuss this issue further in §6.

*4.1.2 IDDG Construction.* We now describe how BACScan constructs the navigation graph and connects the IDDE between pages.

**Page Exploration.** BACScan first explores each web page using a crawler to construct the navigation graph. In accordance with the state-of-the-art crawler [31, 32], BACScan explores the web pages through four types of actions, i.e., extracting static URLs, submitting forms, interacting with iframes, and triggering JavaScript event handlers (e.g., clicks) within the pages. This method ensures that BACScan conducts a comprehensive exploration of the current web pages. Subsequently, BACScan treats the pages generated through these actions as new page nodes, linking them to the currently explored page through navigation edges. These edges represent how the actions facilitate navigation from one page node to another, thereby forming the navigation graph of the target application.

**Token Insertion.** During the exploration of each page, BACScan intercepts all issued modification requests. Then, apart from certain structured parameters, such as `date` and `email`, BACScan replaces other parameter values in the request with unique tokens, and subsequently releases the intercepted requests one by one. Specifically, we generate these unique tokens as pseudo-random strings of six lowercase characters, e.g., axdvhn. This randomness ensures sufficiently high entropy to prevent them from being mistaken for other strings within the application.

It must be acknowledged that some data dependencies arise from structured data, e.g., `email`, and replacing only unstructured parameters with unique tokens may miss these relationships. Nevertheless, pages with MBAC vulnerabilities typically contain numerous unstructured parameters (e.g., `username`, `address`) that can be replaced. Therefore, we can leverage these parameters to establish the data dependency relationships between pages, which do not significantly affect the detection of MBAC vulnerabilities. As shown in §5.2, our evaluation results demonstrate that the false negatives caused by the structured parameters of BACScan are only 2.90%.

**Hierarchical Navigation Graph Traversal.** For each intercepted modification request with an inserted token, BACScan attempts to traverse the navigation graph to find the corresponding status page. One straightforward method is to revisit all web pages to search for the status page. However, this approach is highly time-consuming and impractical. Therefore, BACScan employs a hierarchical traversal strategy to efficiently identify the corresponding status page, thereby establishing data dependency. Specifically, the hierarchical strategy involves a comprehensive evaluation process of all pages to prioritize in revisiting the most promising ones. The traversal stops once the first page with a data dependency on the intercepted modification request is identified. This approach helps avoid indiscriminate traversal of all pages, thus facilitating a more efficient identification of the status pages that provide feedback.

To achieve this, a *scoring algorithm* is designed to assess the likelihood of being a status page of specific modification pages. This algorithm integrates multiple factors, including URL similarity, request parameter relevance, and the distance to the intercepted modification page within the navigation graph. BACScan then selects the page with the highest score for prioritized access. The details of the scoring algorithm and its breakdown are as follows:

$$S(m,r) = \frac{w_1}{1+Sim(m,r)} + w_2 \cdot \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}(p_i \in r) + \frac{w_3}{Dist(m,r)} \quad (1)$$

❶ *URL Similarity.* BACScan analyzes the response of the intercepted modification request and attempts to extract the redirect URL provided in the response, which may appear either in the response header (e.g., `Location:/order`) or the response body (e.g., `redirect:/order`). This design choice is based on the observation that developers tend to redirect users to a page displaying the modified data, i.e., the status page, after a modification is performed. Given that redirect URLs may contain various path parameters that could cause direct URL matching to fail, BACScan thus leverages the Levenshtein distance algorithm [59] to assess the similarity between the redirect URL and the URLs of previously accessed pages (denoted as $Sim(m,r)$ in Equation 1), assigning higher scores to more similar pages. Additionally, in cases where no redirection information is found in the response, BACScan defaults to using the URL of the modification page as a substitute for the redirect URL.

❷ *Request Parameter.* BACScan then analyzes the relationship between the parameters of the intercepted modification page (i.e., $p_i$) and the content (i.e., HTTP response) of previously accessed pages. Specifically, as shown in Equation 1, BACScan parses the parameter names from the modification request and checks whether they appear in the content of accessed pages. The result of $\mathbb{I}(p_i \in r)$ is set to 1 if the parameter is found in the content, or 0 otherwise. This metric is well-founded. For instance, consider a modification request that contains an `address` parameter, and another page presents information related to addresses. It is reasonable to infer that, compared to other pages, this data read page is more likely to exhibit a data dependency with the modification request.

❸ *Navigation Distance.* The distance between the previously accessed status pages and the intercepted modification page in the navigation graph is another key factor considered by BACScan, denoted as $Dist(m,r)$. Typically, modification pages are closely associated with their corresponding status pages regarding business functionality, and developers design them to require minimal interaction, thus positioning them relatively close to each other in the navigation graph. Therefore, BACScan assigns higher scores to pages that are closer in distance.

As shown in §5.4, this hierarchical strategy enables BACScan to reduce the total number of page visits, thereby improving the efficiency of navigation graph traversal and resulting in a 109.10% improvement in overall performance.

**IDDE Connection.** In the final step, BACScan establishes IDDE by analyzing the relationship between the token inserted by the modification page and the content of the revisited data read page. Specifically, based on the type of operation performed on the token, we categorize the IDDE into three types: 1) If the inserted token appears in the content of the read page, BACScan treats it as the corresponding status page and establishes an INSERT-type edge between the two nodes. 2) If the inserted token is found in the read page and a previously existing token disappears, BACScan creates an UPDATE-type edge. 3) If only a previously existing token disappears from the read page, BACScan forms a DELETE-type edge. For modification page nodes that operate on the same token, BACScan groups them into a single node cluster and labels the modification type (i.e., `INSERT`, `UPDATE`, or `DELETE`) for each page to facilitate further BAC vulnerability detection.

### 4.1.3 IDDG Construction Example.

We refer to Figure 4 as an example to provide a detailed explanation of how BACScan constructs the IDDG. Figure 4 (a) illustrates a web page displaying order data. The end user accesses this functionality via the `/orderList` path to view her submitted orders. Figure 4 (b) shows the constructed IDDG of the order data. The IDDG consists of three nodes, each representing adding, viewing, and deleting orders, respectively. To elaborate, we take the IDDE between the "Add Order" and "List Order" pages as an example to describe its construction process in detail. Initially, during the page exploration phase, BACScan identifies that the "List

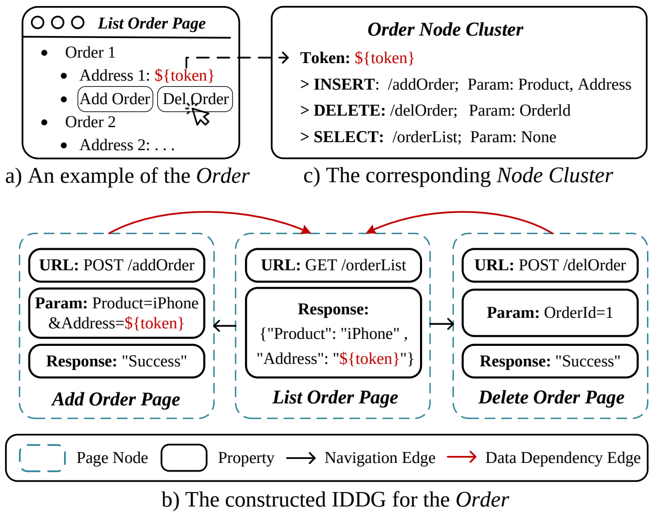


**Figure 4: An example of constructed IDDG and Cluster.**

Order" page can navigate to the "Add Order" pages, thereby connecting the two nodes using navigation edges (black line). Subsequently, BACScan intercepts the order submission request, replaces the order address parameter with a pseudo-randomly generated token, and traverses the navigation graph to locate this inserted token. Finally, by observing and analyzing the token's behavior (e.g., its appearance or removal on the page), BACScan confirms the token's presence on the "List Order" page. As a result, BACScan establishes an IDDE to represent the data dependency relationship between the "Add Order" and "List Order" pages (red line). Figure 4 (c) presents the cluster of nodes that operate on order data. BACScan precisely identifies and marks the token's location within the web page, enabling the grouping of subsequent modifications to this token (e.g., insertion, deletion) into a single operational cluster. This clustering process further facilitates the detection of BAC vulnerabilities.

## 4.2 BAC Vulnerability Detection

In this section, we elaborate on how BACScan achieves effective BAC vulnerability detection. Specifically, BACScan utilizes two distinct oracles, i.e., the feedback-driven oracle for detecting MBAC vulnerabilities and the response similarity-based oracle for identifying RBAC vulnerabilities.

*4.2.1 MBAC Vulnerability Detection.* The process of detecting MBAC vulnerabilities primarily consists of two stages: Modification Request Replay and Feedback-driven Oracle Validation.

**Modification Request Replay.** BACScan attempts to replay the victim requests sent to modification pages in the IDDG using the attacker's session, replacing the token in the parameters with a newly generated token. Given that modification requests are typically POST or similar types that often require CSRF tokens, directly replaying these requests may fail. To address this, BACScan traces back along the navigation edges to locate the preceding data read page of the target modification page. Then, starting from this read page, BACScan follows the navigation edges, executing related actions such as triggering event handlers and submitting forms, until it successfully reaches the target modification page.

Furthermore, BACScan employs distinct strategies to replay the modification request based on the type of operation. For INSERT- and UPDATE-types, it directly replays the request to attempt unauthorized manipulation of the victim's data. For DELETE-types, however, direct replay often fails. This issue arises because, during the construction of the IDDG, the target data associated with the delete page may have already been permanently removed. Consequently, replaying a request to delete non-existent data leads to a server error. For instance, during IDDG construction, if a DELETE-type request has already removed an order with the parameter `OrderId=1`, replaying the same request would fail as the order no longer exists and cannot be deleted again.

To address this issue, BACScan first leverages the node cluster to locate the page that originally inserts the token removed by the DELETE page. It then revisits the corresponding INSERT-type page to reinsert the token into the application, modifying the parameters of the replayed DELETE request to reference the newly inserted data (e.g., `OrderId=2`). This approach ensures the successful replay of DELETE-type modification requests.

**Feedback-driven Oracle Validation.** Then, BACScan follows the IDDE to locate and revisit the status page with a data dependency on the replayed modification request, using the victim's session to obtain feedback on whether the unauthorized modification succeeded. Specifically, BACScan adopts different detection strategies based on the type of modification request.

- For INSERT-type operations, the appearance of a newly inserted token on the status page acts as the feedback, directly indicating an MBAC vulnerability. This feedback confirms that the scanner can observe unauthorized insertion.
- For UPDATE-type operations, the detection of a new token alongside the disappearance of an existing token on the status page signals an MBAC vulnerability.
- For DELETE-type operations, the disappearance of a previously existing token from the status page serves as evidence of an MBAC vulnerability.

For example, in Figure 4 (b), BACScan uses the victim's session to insert `token-A` via the /addOrder modification page. The inserted `token-A` can be viewed on the corresponding status page (i.e., /orderList). Next, BACScan replaces the value of `Address` parameter with `token-B` and replays the /addOrder request using the attacker's session. Finally, BACScan revisits the status page `/orderList` using the victim's session to check for the presence of `token-B`. If found, BACScan reports an MBAC vulnerability.

*4.2.2 RBAC Vulnerability Detection.* For RBAC vulnerability detection, BACScan employs the widely adopted three-step approach described in §2.3. First, BACScan uses both the victim's and the attacker's sessions to explore the web application, separately collecting the pages accessible to each, along with the corresponding page content (i.e., HTTP responses) for each user. Second, BACScan adopts a SoTA approach [35] to filter out public pages accessible to both the attacker and the victim. Specifically, it compares each page in the attacker's and victim's navigation graphs, treating pages with identical URLs and content as public. This process allows BACScan to select only the pages containing sensitive data that are exclusively accessible to the victim. Third, BACScan utilizes the attacker's session to replay the GET requests sent to data read pages within the navigation graph. This step simulates unauthorized access attempts by the attacker. BACScan then applies a SoTA response similarity-based oracle, as described in [35], to detect RBAC vulnerabilities. If the similarity score exceeds a predefined threshold (set to 0.7, following [35]), BACScan concludes that sensitive data intended for the victim is also accessible to the attacker, thereby reporting an RBAC vulnerability.

## 5 Evaluation

### 5.1 Experimental Setup

**Implementation.** Our prototype implementation follows the approach presented in Section §4. For the crawler, our prototype follows the Black Widow [32] to automatically explore web pages (e.g., trigger JavaScript event handlers within pages), thereby constructing the navigation graph. In addition, our prototype uses Python and Playwright (maintained by Microsoft [10]) to interact with the web browser, thereby intercepting requests and constructing the IDDG. For the hyper-parameters in hierarchical traversal strategy, we conducted sensitivity testing using our ground-truth set to fine-tune the parameters, ultimately setting $w_1$, $w_2$, and $w_3$ to 0.4, 0.3, and 0.3, respectively. In total, the entire prototype consists of 4,116 lines of Python code. All experiments run on an Ubuntu 22.04 machine, equipped with a 64-core CPU and 256 GB of memory.

**Experiments.** Our evaluation seeks to answer the following four research questions:

- RQ1: How effective is BACScan in detecting BAC vulnerabilities within real-world applications?
- RQ2: How does BACScan perform compared to SoTA approaches?
- RQ3: How efficient is BACScan in performing the analysis?
- RQ4: How effective is the construction of IDDG?

**Dataset.** Our dataset comprises 20 popular open-source web applications. Among these, 14 applications are designated as the testing set, while 6 applications with 44 known BAC vulnerabilities constitute the ground-truth set. These applications have been widely evaluated in previous studies and are implemented in various languages, including Java, PHP, and Go, further proving that BACScan is language-independent. We manually set up the runtime environments for all 20 applications and applied BACScan to detect BAC vulnerabilities within them. Detailed information about these applications is provided in Table 1. The step-by-step construction process is as follows.

- **Testing Set.** We collected 14 widely-used web applications from popular open-source repositories (e.g., GitHub [14]) based on the following criteria: (1) Considering the importance of dataset reliability and representativeness, each selected application has been widely evaluated in previous studies [26, 34, 43, 48, 56, 58]. (2) To ensure their popularity, we require that the selected applications should have over 100 stars on GitHub. (3) Additionally, to demonstrate that BACScan is programming language-independent, the selected applications include multiple programming languages. Consequently, our testing set comprises 7 PHP-based applications and 7 Java-based applications, among which 13 applications have over 1,000 stars, and 1 application has over 100 stars.
- **Ground-truth Set.** We collected applications containing known BAC vulnerabilities to serve as the ground truth set. Given that many known CVEs lack detailed vulnerability information, and constructing vulnerability proof-of-concept (PoC) through code review requires significant manual effort, we prioritized in gathering BAC vulnerabilities from sources such as Huntr [7], ExploitDB [6], and existing studies [43, 58] that provide PoCs. Specifically, the collection was based on keywords (e.g., broken access control, missing authorization) and CWEs (e.g., CWE-200 [3], CWE-284 [4]). The vulnerability disclosure dates were restricted to January 2022 to January 2024. Additionally, to reduce the manual effort involved in setting up runtime environments, we preferred to include applications with multiple known vulnerabilities in the ground-truth set. In the end, our ground-truth set consists of 44 validated BAC vulnerabilities from 6 applications, including 24 MBAC vulnerabilities and 20 RBAC vulnerabilities. Table 1 presents a detailed breakdown showing the distribution of these 44 vulnerabilities.

**Table 1: Breakdown of our evaluation dataset, including their names, popularity (i.e., stars), the number of established IDDEs, detected BAC vulns, and assigned CVEs.**

| Testing Set | # CVEs / Vulns[1] | # IDDEs | # Stars | # Language |
|---|---|---|---|---|
| Mall-swarm | 0 + 1 / 1 + 1 | 49 | 11,988 | Java |
| Newbee_mall | 2 + 2 / 2 + 3 | 25 | 10,972 | Java |
| Invoiceninja | 1 + 0 / 2 + 0 | 150 | 8,192 | PHP |
| PrestaShop | 0 + 0 / 0 + 0 | 49 | 8,139 | PHP |
| XMall | 5 + 4 / 6 + 4 | 26 | 7,135 | Java |
| SpringBlade | 0 + 0 / 0 + 0 | 6 | 6,298 | Java |
| Ruoyi | 0 + 0 / 0 + 0 | 71 | 6,124 | Java |
| Joomla | 0 + 0 / 0 + 0 | 21 | 4,785 | PHP |
| Ampache | 1 + 0 / 2 + 0 | 181 | 3,500 | PHP |
| OpenEMR | 4 + 2 / 5 + 6 | 92 | 3,140 | PHP |
| Supermarket | 5 + 2 / 6 + 3 | 13 | 2,006 | Java |
| PhpBB | 0 + 0 / 0 + 0 | 32 | 1,836 | PHP |
| Apache_inlong | 2 + 0 / 4 + 2 | 143 | 1,370 | Java |
| Webid | 2 + 2 / 5 + 2 | 59 | 114 | PHP |
| **Total** | 22 + 13 / 34 + 22 | 917 | / | / |
| **Ground Truth Set** | **# Known Vulns[2]** | **# IDDEs** | **# Stars** | **# Language** |
| Memos-0.9.0 | 4 + 5 | 27 | 34,380 | Go |
| WordPress_SPM-4.57 | 6 + 1 | 44 | 19,453 | PHP |
| Snipe_it-5.0.3 | 0 + 3 | 57 | 11,152 | PHP |
| Lunary-1.2.7 | 6 + 6 | 16 | 1,085 | TypeScript |
| MyBloggie-2.1.4 | 2 + 0 | 15 | 281 | PHP |
| Collabtive-2.1 | 6 + 5 | 59 | 215 | PHP |
| **Total** | 24 + 20 | 218 | / | / |

[1]The number of 0-day BAC vulnerabilities discovered by BACScan and the assigned CVEs. For example, '22 + 13 / 33 + 21' means 33 MBAC and 21 RBAC vulnerabilities, with 22 MBAC and 13 RBAC CVEs assigned.
[2]The number of known BAC vulnerabilities in the application. For example, '24 + 20' means 24 MBAC and 20 RBAC vulnerabilities.

**Table 2: The effectiveness of BACScan in BAC vulnerability detection (RQ1).**

| Dataset | MBAC Vulnerability | | | | | RBAC Vulnerability | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | TP | FP | FN | Prec (%) | Recall (%) | TP | FP | FN | Prec (%) | Recall (%) |
| Ground Truth Set | 20 | 0 | 4 | 100.00% | 83.33% | 15 | 3 | 5 | 83.33% | 75.00% |
| Testing Set | 33 | 0 | / | 100.00% | / | 21 | 5 | / | 80.77% | / |
| **Total** | 53 | 0 | / | 100.00% | / | 36 | 8 | / | 81.82% | / |

**BACScan Setup.** For each application, we set up two victim users and one attacker user for BACScan. Among the two victim users, one shares the same role as the attacker user (e.g., a regular user), while the other has a higher role (e.g., an administrator). Then, BACScan constructs the IDDG for each victim user and replays requests using the attacker's cookie to detect BAC vulnerabilities.

## 5.2 RQ1: Effectiveness

We evaluated the effectiveness of BACScan in detecting BAC vulnerabilities on two separate datasets: the ground-truth set and the testing set.

**Result Overview.** Table 2 provides the detailed results. In total, BACScan detected 89 BAC vulnerabilities within the ground-truth and testing sets, of which 54 are potential 0-day BAC vulnerabilities and 35 are known BAC vulnerabilities.

Specifically, for MBAC vulnerabilities, within the ground-truth dataset, BACScan successfully detected 20 MBAC vulnerabilities, with 0 false positives and 4 false negatives. The precision and recall rates were notably high at 100.00% and 83.33%, respectively. Additionally, in the testing set, BACScan reported 33 potential 0-day MBAC vulnerabilities. Owing to BACScan's ability to directly provide URLs and parameters associated with the vulnerabilities, we could efficiently check the vulnerability reports. Ultimately, we verified that all 33 MBAC vulnerabilities are truly exploitable within the testing set. For RBAC vulnerabilities, BACScan achieved a precision rate of 83.33% and a recall rate of 75.00% on the ground truth set. On the testing set, BACScan detected 21 zero-day RBAC vulnerabilities with an accuracy of 80.77%.

**Vulnerability Disclosure.** Attackers can exploit these vulnerabilities to leak private user data or even delete data stored within the application, thereby severely compromising data confidentiality and integrity. For example, a vulnerability identified in OpenEMR [8] allows attackers to modify any patient's diagnosis or medications, posing a serious threat to patient's privacy, health, and even life. These security breaches underscore the urgent need for effective BAC vulnerability detection mechanisms to safeguard both data privacy and integrity in critical web applications.

Therefore, we promptly reached out to the corresponding developers to report all confirmed vulnerabilities through their dedicated email addresses and vulnerability reporting forms. Adhering to responsible disclosure practices, we will refrain from publicly releasing any unresolved vulnerabilities until they have been addressed. To date, 39 vulnerabilities have been confirmed by the developers, and we have received 35 CVE identifiers in acknowledgment, including 22 MBAC and 13 RBAC ones.

**False Positives.** For all 8 false positives of BAC vulnerabilities, we conducted a detailed analysis and found that they are mainly caused by *the limited response similarity algorithm.* Although the response similarity algorithm is highly effective for RBAC vulnerability detection, it can also produce false positives in certain scenarios. These false positives arise from the highly flexible and diverse page content and HTML structures across different applications, which make the predefined fixed similarity threshold less effective in handling these variations, resulting in false positives.

**False Negatives.** For 9 false negatives, the primary causes can be attributed to two main aspects. Firstly, 2 false negatives resulted from *incomplete construction of the IDDG.* For example, consider the missed vulnerability in WeBid [17]. By passing an auction ID in the HTTP request parameters (e.g., `auction=1`), an attacker can exploit this vulnerability to relist any closed auction without authorization. However, since this HTTP request only contains an integer parameter, BACScan cannot replace it with unique string tokens, which is essential for establishing inter-page data dependencies. Consequently, BACScan failed to detect this vulnerability. Secondly, 7 false negatives were caused by *the crawler's limited code coverage.* Achieving comprehensive exploration of all web pages through automated crawlers remains a widely recognized challenge. Although we utilized state-of-the-art crawler technology, consistent with numerous existing studies that rely on crawler-based exploration, certain pages inevitably remained unexplored [32, 39]. This coverage limitation subsequently led to undetected vulnerabilities on those pages.

## 5.3 RQ2: Comparison

In this part, we compare the effectiveness of BACScan with two baselines. For a thorough evaluation, we use the 54 (33 MBAC + 21 RBAC) verified 0-day vulnerabilities reported by BACScan and 44 known BAC vulnerabilities as the *vulnerability ground truth set*, assessing the precision and recall rates of the two baseline approaches across the entire dataset.

**Baseline Setup.** We first describe the setup of these two baselines, i.e., EvoCrawl [35] and BurpSuite [12].

- **EvoCrawl** is the state-of-the-art approach to BAC vulnerability detection and will be presented at NDSS 2025 [35]. Its code is available as open source on GitHub [5]. Given that EvoCrawl itself includes a crawling module and an oracle specifically designed for detecting BAC vulnerabilities, we only needed to provide the target URL and user login credentials to initiate detection. To prevent the crawler from exceeding runtime limits, the crawling process was restricted to a maximum of 8 hours.

**Table 3: Comparison between BACScan and baselines in BAC vulnerability detection (RQ2).**

| Baselines | MBAC Vulnerability | | | | | RBAC Vulnerability | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | TP | FP | FN | Prec (%) | Recall (%) | TP | FP | FN | Prec (%) | Recall (%) |
| EvoCrawl | 17 | 16 | 40 | 51.52% | 29.82% | 23 | 9 | 18 | 71.88% | 56.10% |
| BurpSuite | 31 | 19 | 26 | 62.00% | 54.39% | 35 | 15 | 6 | 70.00% | 85.37% |
| BACScan | 53 | 0 | 4 | 100.00% | 92.98% | 36 | 8 | 5 | 81.82% | 87.80% |

- **BurpSuite** is a comprehensive commercial black-box scanner that supports enhanced vulnerability detection through various extensions. We installed the *Autorize* extension from BurpSuite's BApp Store [13], enabling effective detection of BAC vulnerabilities. Next, we configured the login credentials in BurpSuite's Dashboard module and initiated the vulnerability detection by clicking the "New Scan" button.

**False Positives.** As shown in Table 3, BACScan surpasses EvoCrawl and BurpSuite by 94.12% and 61.29% in the precision rate of MBAC vulnerability detection, respectively. We conducted an in-depth analysis of all the false positives reported by EvoCrawl and BurpSuite. Apart from the false positive causes in BACScan (i.e., insensitive resources and response similarity algorithm), we found that the primary reason for these false positives is *the inherent defect in their response-based oracle*, as described in §2.3. This origin of false positives is understandable, as many MBAC vulnerabilities do not include indicators of successful operations in their direct responses. For example, in Supermarket [11], the response to an order deletion operation always returns a 302 status code, regardless of whether the deletion succeeds. Such outcome-independent responses render the response-based oracle ineffective, leading to false positives in these baseline tools.

**False Negatives.** The recall rates of EvoCrawl and BurpSuite for MBAC vulnerability detection result are 29.82% and 54.39%, respectively. Our comprehensive analysis of all these false negatives revealed that, beyond those caused by limited code coverage in the crawlers, additional false negatives primarily stem from two factors. Firstly, most of the false negatives in EvoCrawl and BurpSuite arise from *inherent limitations in the response-based oracle*. For instance, in OpenEMR [8], responses to modification requests are linked to the identity of the request sender. Specifically, when an attacker submits a request to modify a victim's data, the response reflects only data associated with the attacker, regardless of whether the modification succeeds, and excludes any of the victim's data. This account-specific response leads to low similarity between attacker and victim responses, causing these baselines to mistakenly assume no vulnerability, resulting in false negatives. Secondly, 21 false negatives in EvoCrawl stem from its *user-specific data filtering strategy*. As outlined in Section III.C of the EvoCrawl's paper [35], this strategy filters out user-specific elements (e.g., username) based on page similarity. However, this approach inadvertently excludes substantial amounts of user-specific and sensitive data, such as individual user orders, resulting in numerous false negatives.

## 5.4 RQ3: Efficiency

**Evaluation Setup.** In this part, we evaluated the performance of BACScan in analyzing the entire dataset, focusing on its efficiency in detecting vulnerabilities. To ensure the robustness of our results, each web application was tested in two rounds. The average time taken for vulnerability detection was then calculated, allowing us to accurately assess the performance of BACScan.

Additionally, to evaluate the improvement in performance provided by the hierarchical traversal strategy, we created a variant of BACScan, named BACScan-Random. This variant disables the hierarchical strategy and instead randomly selects pages from the navigation graph. We set an 8-hour timeout for this variant to limit its execution time. By comparing the results of both versions, we can evaluate the specific contribution of the hierarchical traversal strategy to the overall performance.

**Result Analysis.** Figure 5 shows the time taken by BACScan and BACScan-Random to analyze the entire dataset. On average, BACScan requires 1.1 hours to complete the task of detecting BAC vulnerabilities in a given application. This analysis time encompasses both the IDDG construction and the vulnerability detection phases. Compared to other dynamic testing approaches that rely on crawlers [31, 32, 35, 56], we consider BACScan's performance to be acceptable.

In comparison, BACScan-Random requires an average of 2.3 hours for vulnerability detection, representing a performance decrease of 109.10% relative to BACScan. In some applications (e.g., OpenEMR), BACScan-Random's performance dropped by as much as 153.24%, ultimately leading to a timeout. This indicates that BACScan-Random spends significantly more time searching for the inserted token within the navigation graph, while BACScan avoids this overhead by employing the hierarchical traversal strategy. Moreover, as the scale of the applications increases, the performance degradation of BACScan-Random becomes more pronounced. This is because, as the size of the application grows, the navigation graph expands, leading to longer traversal times. These experimental results highlight the significant performance improvements achieved by the hierarchical traversal strategy in BACScan.

## 5.5 RQ4: IDDG

The IDDG is crucial to our feedback-driven oracle, making its precision vital for effective MBAC vulnerability detection. To ensure its reliability, we manually evaluate the correctness of the constructed IDDG and break down the detailed data of the IDDE, thereby demonstrating its importance in vulnerability detection.

**Result Analysis.** As shown in Table 1, we present the breakdown of the 1,135 IDDEs constructed by BACScan across all evaluated applications. Given the considerable manual effort required to verify correctness, we randomly selected 10% (113) of the total 1,135 IDDEs to assess their precision. Our evaluation revealed that only 4 (3.54%) of the sampled IDDEs were incorrectly generated. A detailed analysis of these false positives indicates that they were all caused

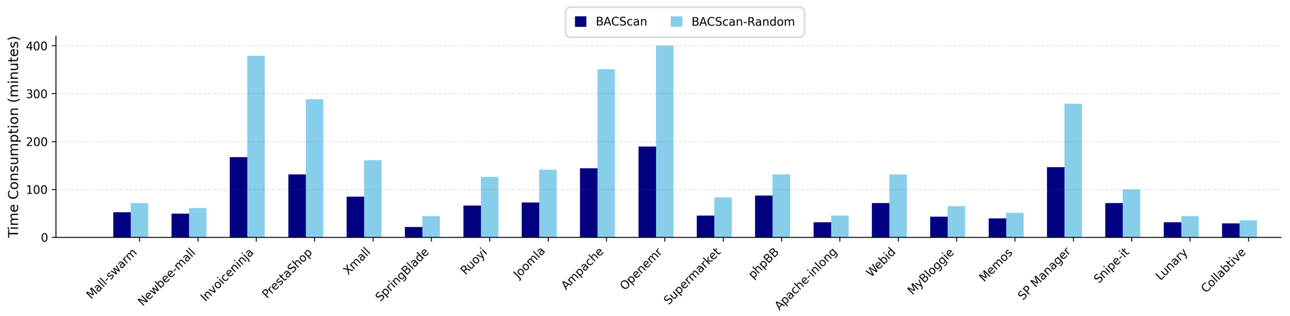


**Figure 5: Time consumption comparison between BACScan and BACScan-Random.**

```
  // PoC send to Modification Page
1 POST /interface/patient_file/add_transaction.php
2 Cookie: Attacker's Cookie

3 form_date=2024-01-01&form_note=${token}
  // Response from Status Page
4 GET /interface/patient_file/transactions.php
5 Cookie: Victim's Cookie

6 <html>
7    <titiel>Patient Transactions</title>
8    ... <td>${token}</td> ...
9 </html>
```

**Figure 6: Arbitrary Patient Note Update vulnerability in OpenEMR application (over 3k stars on Github).**

by GET-type modification pages. As described in §4.1.1, BACScan assumes that only POST requests modify data. Consequently, during the IDDG construction, some GET-type modification pages inadvertently modified the tokens inserted by BACScan. When revisiting the corresponding status page, BACScan detected unexpected token modifications, leading to the incorrect connection of IDDEs. Experimental results show that developers mostly (96.43%) adhere to the definitions in the RFC, avoiding the use of GET requests for data modification operations. Moreover, these four incorrect IDDEs did not result in any false negatives or false positives, further confirming the reliability of our IDDG.

We then analyzed the distribution of the 1,135 IDDEs by type, with 331 for INSERT, 452 for UPDATE, and 352 for DELETE. Interestingly, while DELETE-type IDDEs accounted for 31.40%, DELETE-type MBAC comprised 48.89% of the total detected vulnerabilities. Upon further source code analysis, we found that developers often implement DELETE operations directly via data indices (e.g., deleting data by ID). In contrast, UPDATE and INSERT operations typically involve binding data to a specific user, unintentionally performing authorization checks. This finding also highlights the importance of BACScan's replay strategy for DELETE requests.

## 5.6 Case Study

We now showcase two BAC vulnerabilities detected by BACScan and missed by existing tools in highly popular applications, further illustrating the high risk posed by these vulnerabilities and demonstrating the practical utility of BACScan in real-world scenarios.

```
  // Response from Status Page before Attack
1 GET /member/orderDetail
2 Cookie: Victim's Cookie

3 <html> <titiel>Order Detail</title>
4    <div class="id">1</div>
5    <div class="addr">${token}</div>
6    <div class="phone">xxx</div>
     ...
7 </html>
```

```
   // PoC send to Modification Page
8  POST /member/delOrder
9  Cookie: Attacker's Cookie

10 addressId=1
   // Response from Status Page after Attack
11 GET /member/orderDetail
12 Cookie: Victim's Cookie

13 <html><titiel>Order Detail</title>
14 </html>
```

**Figure 7: Arbitrary Order Deletion vulnerability in XMall application (over 7k stars on Github).**

**Arbitrary Patient Note Update in OpenEMR.** The OpenEMR is an open-source and widely used hospital management system with over 3,000 stars on GitHub. As shown in Figure 6, BACScan successfully detected a BAC vulnerability within the application that could lead to arbitrary updates of patient notes. Specifically, during the vulnerability detection process, BACScan replayed the modification request (i.e., `/add_transaction.php`) and inserted a random token into the `form_note` parameter (`$token` in line 3). Subsequently, BACScan replayed this request using the attacker's cookie. Finally, leveraging the constructed IDDG, BACScan located the corresponding status page (i.e., `/transaction.php`) and identified the inserted token (`$token` in line 8) on the page. Consequently, BACScan reported the presence of an MBAC vulnerability. Attackers could exploit this vulnerability to modify any patient's diagnosis or medications, which poses a severe threat to patient safety. Given the extensive potential damage posed by this vulnerability, we promptly reported this critical issue to the developers and were issued a CVE (CVE-2024-46**1).

**Arbitrary Order Deletion in XMall.** The XMall application is a highly popular e-commerce application with over 7,000 stars on GitHub. Figure 7 illustrates a BAC vulnerability identified by BACScan within this application, which allows deleting arbitrary order. Lines 8–10 showcase the modification page where the vulnerability resides. BACScan replayed the request and subsequently revisited the corresponding status page (i.e., /member/order-Detail). Lines 1–7 and 11–14 present the content of the status page before and after the replay of the modification request, respectively. BACScan identified that the token originally present (line 5) disappeared following the modification request, indicating an MBAC vulnerability. Attackers can exploit this vulnerability to delete any user's orders, posing significant risks to user privacy and potentially resulting in financial losses. We immediately reported it to the developers of the vulnerable application. As a result, we were granted a CVE identifier, i.e., CVE-2024-36**0.

## 6 Discussion

**Limitations and Future works.** While BACScan worked well in the evaluations, we see several potential improvements.

- *Web Crawler.* For a long time, the question of how to enable crawlers to fully explore web applications has been a popular research topic. In practice, we have observed that state-of-the-art crawlers (e.g., Black Widow [32]) are indeed capable of effectively exploring the majority of web pages. However, there are still some web pages that remain insufficiently explored, which leads to false negatives in BACScan. In the future, as crawling technologies continue to advance, we believe that the performance of BACScan will be further improved.
- *Performance Trade-off.* As described in §4.1.2, BACScan assumes that tokens are only inserted into POST requests triggered during the page exploration process. It is important to note that excluding token insertion for GET requests is primarily a performance trade-off. While intercepting GET requests and inserting tokens would be effective, it would also significantly increase the analysis time required by BACScan. Hooking database operations to determine which database actions are triggered by HTTP requests is a highly effective approach. However, in the context of black-box testing, we do not have access to the database. Therefore, we can only make every effort to infer the triggered database actions from the evidence in HTTP requests and responses.

**Ethics Consideration.** This study has not presented any legal or ethical issues. We obtained the source code for local analysis and responsibly reported all detected vulnerabilities in open-source applications to the CVE Numbering Authority (CNA) [2]. Additionally, we have contacted all the developers regarding the BAC vulnerabilities found in §5.2, and will continue to communicate with them throughout the vulnerability disclosure process.

## 7 Related Work

**BAC Vulnerability Detection.** There are numerous studies [28, 35, 37, 40–44, 46, 49, 50, 52–55, 58, 60–62] that have employed various techniques to detect BAC vulnerabilities. These studies are commonly categorized into two main types: dynamic approaches and static approaches. The dynamic approaches [28, 35, 37, 40–42, 49, 50, 62] typically simulate multiple users through login sessions and use cross-user forced browsing combined with response-based oracles to detect vulnerabilities. However, these oracles are inadequate for detecting MBAC vulnerabilities due to their inability to capture data dependencies, affecting precision and recall. Static approaches [43, 44, 46, 52–55, 58, 60, 61] model user credentials (e.g., $_SESSION in PHP) and use predefined rules to identify permission checks. While effective in some cases, they still present notable limitations. On one hand, they suffer from high false positives due to the lack of runtime context to validate vulnerability reports. On the other hand, they require static analysis of the source code, limiting their applicability to programs developed in specific programming languages. Unlike these previous efforts, BACScan eliminates the need for additional input and leverages a novel IDDG-based approach to accurately detect BAC vulnerabilities, addressing the limitations of both static and dynamic methods.

**Web Vulnerability Detection.** In recent years, the techniques for automatically detecting vulnerabilities within web applications have been extensively studied. These techniques also can be categorized into static and dynamic approaches. The static approaches [25, 27, 36, 38, 45, 57] identify user inputs as sources and predefined security-sensitive operations as sinks. They then analyze whether a data flow path exists from the source to the sink, indicating a potential vulnerability. The dynamic approaches [23, 29, 32, 34, 47, 51, 56] design oracles specifically tailored to the security-sensitive operations relevant to different types of vulnerabilities. For example, an exception thrown by an SQL execution function may serve as an oracle to detect SQL injection vulnerabilities. These approaches detect vulnerabilities by monitoring runtime behavior and determining whether the corresponding oracle is triggered during the execution of the application. However, these approaches mainly focus on taint-style vulnerabilities, which rely on well-defined security operations for modeling. In contrast, BAC vulnerabilities are tied to business logic and cannot be effectively modeled. 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 BACScan, a novel black-box approach for detecting Broken-Access-Control (BAC) vulnerabilities in web applications. By introducing a feedback-driven oracle based on inter-page data dependency, BACScan addresses the limitations of existing detection methods, particularly for modification-based BAC vulnerabilities. We evaluate BACScan on 20 open-source web applications, discovering 89 BAC vulnerabilities, including 54 previously unknown high-risk 0-day vulnerabilities and the assignment of 35 new CVE IDs. These findings demonstrate the practical utility of BACScan in BAC vulnerability detection. We hope our work can assist the community in addressing the growing threats posed by BAC vulnerabilities.

## 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, 62172105). Yuan Zhang and Min Yang are the corresponding authors. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012). Min Yang is a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

## References

[1] Amazon Official Website. https://www.amazon.com.
[2] CVE Program. https://www.cve.org/About/Overview.
[3] CWE200. https://cwe.mitre.org/data/definitions/200.html.
[4] CWE284. https://cwe.mitre.org/data/definitions/284.html.
[5] Evocrawl on Github. https://github.com/dlgroupuoft/evocrawl.
[6] Exploit DB. https://www.exploit-db.com/.
[7] Huntr platform. https://huntr.com/.

[8] Openemr on Github. https://github.com/openemr/openemr.
[9] Paypal Official Website. https://www.paypal.com.
[10] Playwright on Github. https://playwright.dev/python/.
[11] Supermarket on Github. https://github.com/ZongXR/SuperMarket.
[12] The Official Website of BurpSuite. https://portswigger.net/burp.
[13] The Official Website of BurpSuite's BApp Store. https://portswigger.net/bappstore.
[14] The Official Website of Github. https://github.com/.
[15] Top BAC reports from HackerOne. https://github.com/reddelexc/hackerone-reports/blob/master/tops_by_bug_type/TOPIDOR.md.
[16] Top BAC reports from HackerOne. https://github.com/reddelexc/hackerone-reports/blob/master/tops_by_bug_type/TOPAUTHORIZATION.md.
[17] WeBid on Github. https://github.com/renlok/WeBid.
[18] OWASP Top 10 - 2019. https://owasp.org/API-Security/editions/2019/en/0x11-t10/, 2019.
[19] OWASP Top 10 - 2021. https://owasp.org/Top10/A01_2021-Broken_Access_Control/, 2021.
[20] Notorious Hacks in History. https://securityboulevard.com/2023/03/23-most-notorious-hacks-history-that-fall-under-owasp-top-10/, 2023.
[21] OWASP Top 10 - 2023. https://owasp.org/API-Security/editions/2023/en/0x11-t10/, 2023.
[22] Serious Data Breach News. https://www.apisec.ai/blog/5-real-world-examples-of-business-logic-vulnerabilities-that-resulted-in-data-breaches, 2023.
[23] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
[24] Brad Arkin, Scott Stender, and Gary McGraw. Software penetration testing. *IEEE Security & Privacy*, 3(1):84–87, 2005.
[25] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE european symposium on security and privacy (EuroS&P)*, pages 334–349. IEEE, 2017.
[26] Miao Chen, Tengfei Tu, Hua Zhang, Qiaoyan Wen, and Weihang Wang. Jasmine: A static analysis framework for spring core technologies. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.
[27] Johannes Dahse and Thorsten Holz. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *NDSS*, 2014.
[28] G Deepa, P Santhi Thilagam, Amit Praseed, and Alwyn R Pais. Detlogic: A black-box approach for detecting logic vulnerabilities in web applications. *Journal of Network and Computer Applications*, 109:89–109, 2018.
[29] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A {state-aware}{black-box} web vulnerability scanner. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 523–538, 2012.
[30] Adam Doupé, Marco Cova, and Giovanni Vigna. 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*, pages 111–131. Springer, 2010.
[31] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. Rescan: A middleware framework for realistic and robust black-box web application scanning. In *Network and Distributed System Security (NDSS) Symposium*, 2023.
[32] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. Black widow: Blackbox data-driven web scanning. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1125–1142. IEEE, 2021.
[33] Roy Fielding and Julian Reschke. Hypertext transfer protocol (http/1.1): Semantics and content. RFC 7231, 2014.
[34] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. Atropos: Effective fuzzing of web applications for server-side vulnerabilities. In *USENIX Security Symposium*, 2024.
[35] Xiangyu Guo, Akshay Kawlay, Eric Liu, and David Lie. Evocrawl: Exploring web application code and state using evolutionary search. In *the Network and Distributed System Security Symposium (NDSS)*. Accepted for publication.
[36] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, 2006.
[37] Malte Kushnir, Olivier Favre, Marc Rennhard, Damiano Esposito, and Valentin Zahnd. Automated black box detection of http get request-based access control vulnerabilities in web applications. In *ICISSP 2021, online, 11-13 February 2021*, pages 204–216. SciTePress, 2021.
[38] Penghui Li and Wei Meng. Lchecker: Detecting Loose Comparison Bugs in PHP. In *Proceedings of the Web Conference 2021*, 2021.
[39] Penghui Li, Wei Meng, Mingxue Zhang, Chenlin Wang, and Changhua Luo. Holistic concolic execution for dynamic web applications via symbolic interpreter analysis. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (Oakland). San Francisco, CA, USA*, 2024.
[40] Xiaowei Li, Xujie Si, and Yuan Xue. Automated Black-box Detection of Access Control Vulnerabilities in Web Applications. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, 2014.
[41] Xiaowei Li and Yuan Xue. Block: A Black-box Approach for Detection of State Violation Attacks Towards Web Applications. In *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.
[42] Xiaowei Li and Yuan Xue. Logicscope: Automatic discovery of logic vulnerabilities within web applications. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 481–486, 2013.
[43] Fengyu Liu, Youkun Shi, Yuan Zhang, Guangliang Yang, Enhao Li, and Min Yang. Mocguard: Automatically detecting missing-owner-check vulnerabilities in java web applications. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 10–10. IEEE Computer Society, 2024.
[44] Jie Lu, Haofeng Li, Chen Liu, Lian Li, and Kun Cheng. Detecting Missing-Permission-Check Vulnerabilities in Distributed Cloud Systems. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.
[45] Changhua Luo, Penghui Li, and Wei Meng. TChecker: Precise Static Inter-Procedural Analysis for Detecting Taint-Style Vulnerabilities in PHP Applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.
[46] Maliheh Monshizadeh, Prasad Naldurg, and VN Venkatakrishnan. MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 690–701, 2014.
[47] Eric Olsson, Benjamin Eriksson, Adam Doupé, and Andrei Sabelfeld. {Spider-Scents}: Grey-box database-aware web scanning for stored {XSS}. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6741–6758, 2024.
[48] Yicheng Ouyang, Kailai Shao, Kunqiu Chen, Ruobing Shen, Chao Chen, Mingze Xu, Yuqun Zhang, and Lingming Zhang. Mirrortaint: Practical non-intrusive dynamic taint tracking for jvm-based microservice systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2514–2526. IEEE, 2023.
[49] Jiadong Ren, Mingyou Wu, Bing Zhang, Ke Xu, Shangyang Li, Qian Wang, Yue Chang, and Tao Cheng. Detac: Approach to detect access control vulnerability in web application based on sitemap model with global information representation. *International Journal of Software Engineering and Knowledge Engineering*, 33(09):1327–1354, 2023.
[50] Marc Rennhard, Malte Kushnir, Olivier Favre, Damiano Esposito, and Valentin Zahnd. Automating the detection of access control vulnerabilities in web applications. *SN Computer Science*, 3(5):376, 2022.
[51] Christian Rossow. jAk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings*, 2015.
[52] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Rolecast: Finding Missing Security Checks When You Do Not Know What Checks Are. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 1069–1084, 2011.
[53] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Fix Me Up: Repairing Access-Control Bugs in Web Applications. In *NDSS*. Citeseer, 2013.
[54] Fangqi Sun, Liang Xu, and Zhendong Su. Static Detection of Access Control Vulnerabilities in Web Applications. In *20th USENIX Security Symposium (USENIX Security 11)*, 2011.
[55] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. AutoISES: Automatically Inferring Security Specification and Detecting Violations. In *USENIX Security Symposium*, 2008.
[56] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. Toss a fault to your witcher: Applying Grey-box Coverage-guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities. In *2023 IEEE symposium on security and privacy (SP)*, 2023.
[57] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*, 2014.
[58] Huang Yongheng, Shi Chenghang, Lu Jie, Li Haofeng, Meng Haining, and Li Lian. Detecting broken object-level authorization vulnerabilities in database-backed applications. In *Proceedings of the 31st ACM Conference on Computer and Communications Security (CCS)*, October 2024.
[59] Li Yujian and Liu Bo. A normalized levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence*, 29(6):1091–1095, 2007.
[60] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. Pex: A Permission Check Analysis Framework for Linux Kernel. In *28th USENIX Security Symposium*, 2019.
[61] Jun Zhu, Bill Chu, Heather Lipford, and Tyler Thomas. Mitigating Access Control Vulnerabilities through Interactive Static Analysis. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, 2015.
[62] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. Authscope: Towards automatic discovery of vulnerable authorizations in online services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 799–813, 2017.