Facilitating Access Control Vulnerability Detection in Modern Java Web Applications with Accurate Permission Check Identification

Youkun Shi, Fengyu Liu, Guangliang Yang, Yuan Zhang, Yinzhi Cao, Enhao Li, Xin Tan, Xiapu Luo, Min Yang, Siyi Chen

***Abstract*—Access-control vulnerabilities have emerged as a significant concern in recent years, posing considerable security risks to a wide range of critical systems. The detection of access-control vulnerabilities in Java web applications poses unique challenges, because heuristics used in the past, e.g., access-control specifications or format-specific runtime logs, may not exist in modern Java web applications using web frameworks. Therefore, to date, there is no effective approach to detecting such vulnerabilities in modern Java web applications.**

In this paper, we introduce a novel approach, called `PCFinder`, which leverages multi-level semantics- and context-analysis to conduct accurate permission-check identifications against real-world Java web infrastructures for access-control vulnerability detection. `PCFinder` successfully discovered 58 high-risk broken access control vulnerabilities, with 30 having been assigned CVE identifiers thus far, in analyzing 50 popular, real-world Java web applications. We also evaluate `PCFinder` on manually constructed ground-truth data and show that `PCFinder` achieved a high level of accuracy, i.e., a precision of 94.12% and a recall of 96.97% in identifying permission checks.

***Index Terms*—Web Vulnerability, Broken Access Control, OSS Security.**

I. Introduction

JAVA web applications have gained widespread popularity and extensive deployment in past years, becoming a cornerstone of modern web infrastructure [1], [2], [3], [4]. Modern Java web applications are often developed using a web framework, e.g., Spring Boot, Spring MVC, and Apache Struts [5], [6], [7]. Harnessing their inherent versatility and scalability, they have been a preferred choice for many high-profile websites, including Amazon [1], Microsoft [2], and eBay [3]. Meanwhile, due to their critical importance, they have been subjected to various attacks, with access-control vulnerabilities being particularly prominent. The access-control vulnerability arises from inadequate verification of permissions for privileged operations, such as modifying database records or altering system files. The OWASP Top 10 ranking lists underscore the access control vulnerability, which ascended from the second position in 2013 and 2017 to claim the top spot in 2021 [8], [9], [10].

To the best of our knowledge, no prior works have attempted to detect access-control vulnerabilities in modern web applications developed using Java web frameworks. Furthermore, previous approaches in other domains (e.g., PHP, Android, or distributed applications) [11], [12], [13], [14], [15], [16], [17], [18] or upon classic Jakarta Server Pages (JSP) [19] *cannot* be easily ported to modern Java web applications. Specifically, access-control vulnerability detection requires the identification of inadequate or missing permission checks using a domain-specific approach requiring either syntactic or semantic knowledge. Such knowledge—e.g., access-control [11], [16], [17] and protocol [18] specifications, file structure-based patterns [19], or format-specific runtime logs [15]—often does not exist in modern Java web applications.

Let us describe in detail how such knowledge was used in the past and why it does not apply to Java web applications. On one hand, prior works have used syntactic knowledge—e.g., expert knowledge of permission-based security mechanisms enforced in target programs [20], [13], [21], [14], file structures [19], or access-control [11], [16], [17] and protocol [18] specifications—to model permission checks. However, Java web applications do not possess such security mechanisms or specifications. On the other hand, prior works laid down permission check semantics as a two-step combination of credential identification and credential-permission correlation. Specifically, they first used heuristic knowledge, e.g., super-global variables in PHP [12] and format-specific runtime logs in distributed systems [15], to identify user credentials. They then considered the if-statements that validate these credentials in their conditions as permission checks. However, user credentials used in Java web applications do not obey such heuristic patterns. Moreover, focusing solely on the if-condition while neglecting the context of the if-body in the analysis might lead to numerous false positives.

In this work, we design a novel end-to-end permission-check analysis approach, called `PCFinder` (Permission Check Finder), which analyzes and leverages user-credential semantics and permission-check contexts in Java web applications, to detect access-control vulnerabilities. Our key observation is that credential names in Java web applications often follow certain semantic meanings (e.g., `"roleId"` and `"password"`). Therefore, we design a novel credential-naming model system, which facilitates effective semantic analysis of

This work was supported in part by the National Natural Science Foundation of China (U2436207, 62172105, 62202106), the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012), Hong Kong RGC Projects (PolyU15224121, PolyU15231223) and HKPolyU Project (No. ZGGG).(Corresponding authors: Yuan Zhang and Min Yang.)

Youkun Shi, Fengyu Liu, Guangliang Yang, Yuan Zhang, Enhao Li, Xin Tan and Min Yang are with the School of Computer Science and Technology, Fudan University, Shanghai, China (e-mail: ykshi21@m.fudan.edu.cn; fengyuliu23@m.fudan.edu.cn; yanggl@fudan.edu.cn; yuanxzhang@fudan.edu.cn; ehli23@m.fudan.edu.cn; xintan19@fudan.edu.cn; m_yang@fudan.edu.cn).

Yinzhi Cao is with the Department of Computer Science, Johns Hopkins University, Baltimore, USA (e-mail: yinzhi.cao@jhu.edu).

Xiapu Luo is with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China (e-mail: csxluo@comp.polyu.edu.hk).

Siyi Chen is with Alibaba Group, Hangzhou, China (e-mail: siyi.csy@alibaba-inc.com).

credentials, and enables precise identification and monitoring of user authentication or authorization data throughout the target application.

While the idea is intuitively simple, it is challenging to achieve high performance with either a naïve keyword or a learning-based approach because of the diversity of credential names used in Java web applications. Therefore, PCFinder innovatively introduces an evolutionary strategy that iteratively increases the vocabulary of names based on newly seen naming conventions discovered via static program analysis. Specifically, PCFinder first constructs and initializes the naming model using root words associated with credentials, such as "pass", "role", and "permiss". Then, PCFinder delves into the propagation of credential namings within the target application via data flow analysis, identifying more credential candidates with diverse names. Lastly, PCFinder expands the model across different applications to integrate additional essential root words by analyzing the naming convention commonalities of the identified credential candidates.

After credential identification, PCFinder detects permission checks associated with credentials in a conditional statement. To mitigate false positives, PCFinder relies on constructing and examining permission-check contexts closely tied to detecting these checks. The context is pivotal and comprises two primary layers of information: (1) the credential names referenced in the conditional statements, and (2) the privileged operations that are conditionally executed based on those statements. Our determination strategy involves thoroughly analyzing these contextual layers to discern whether a conditional statement indeed serves as a permission check by verifying the presence and relevance of credential namings to the guarded privileged operations.

We evaluated PCFinder upon 50 popular, real-world Java web applications. PCFinder successfully discovered 58 critical access-control vulnerabilities, including those with Github [22] stars exceeding 10,000, up to 70,000. We responsibly reported all vulnerabilities to their developers and as of now, 30 of these vulnerabilities have been assigned CVE identifiers. We also evaluate the effectiveness of PCFinder in identifying permission checks. The results demonstrated the promising performance of PCFinder, successfully identifying 486 permission checks with a precision rate of 91.35%.

Lastly, we ported a prior work, namely MACE [12] focusing on PHP web applications, to Java, and compared it with PCFinder. Our evaluation results show that PCFinder successfully identified 32 more unknown vulnerabilities than MACE, while also outperforming MACE in precision and recall by 61.65% and 123.06% respectively. At the same time, PCFinder outperforms MACE in terms of precision and recall in permission checks identification by 21.19% and 63.64% respectively.

Contributions. We summarize our contributions as follows:

- In this paper, we present a novel naming model approach for computing and identifying user-credential variables. This approach lays the foundation for addressing the formidable challenge of accurately identifying user credentials.
- Relying on the naming model, we propose a novel permission check identification approach, called PCFinder, that can effectively facilitate access-control vulnerability detection in modern Java web applications. Notably, PCFinder operates in an end-to-end manner, eliminating the need for manual intervention.
- Our experiments against real-world popular Java web applications show PCFinder is accurate and effective against access-control vulnerabilities, contributing to the discovery of 58 previously unknown severe vulnerabilities, assigned with 30 CVE identifiers.
- We have open-sourced the code of our PCFinder prototype, along with our empirical data and evaluation dataset[1].

II. Design Motivation

A. Definitions

We first define and illustrate two crucial terms using the concrete example in Figure 1, which shows a code snippet for updating user information.

```
1  @PutMapping(value = "/update")
2  public ResponseEntity update(SysUser user){
3     SysUser currentUser = SecurityUtils.getSysUser();
      // Permission Check
4     if (!equals(user.getShopId(), currentUser.getShopId())) {
5        throw new Exception("Permission denied");
6     }
      ...
7     if (!passwordEncoder.matches(
8           user.getPassword(), currentUser.getPassword())){
9        user.setPassword(encode(user.getPassword()));
10    }
11    // Privileged Operation
12    sysUserService.updateUserInfo(user);
13    return ResponseEntity.success();
14 }
```

Fig. 1: An example of the if-implemented permission check and the if-statements that often lead to false positive in permission check identification (highlighted in red background).

- **Permission check** is an effective access control enforcement for safeguarding sensitive resources in web applications. Developers typically implement it as a conditional statement to verify a user's permissions before allowing a privileged operation. For example, the if statement in lines 4-6 of Figure 1 is a permission check. It ensures that a user can only update profiles belonging to the same shop by comparing the shopId with that of the currently logged-in user (currentUser). If the check fails, it interrupts the control flow by throwing an exception, thereby preventing the privileged operation `updateUserInfo()` in line 12.
- **User credentials**, being the primary objects of validation in permission checks, commonly represent the *authentication* or *authorization* information of users. In Figure 1, the values returned by `user.getShopId()` serve as user credentials. They represent the users' authorization information (i.e., their shop affiliation), which is critical for making the access control decision.

[1]https://github.com/seclab-fudan/PCFinder

B. Three Key Observations on Java Permission Checks

To understand common access control patterns, we conducted an empirical study[2] of 324 permission-sensitive paths sampled from our dataset and 26 published security patches. Our analysis identified 361 permission checks, revealing three key observations regarding their implementation in Java web applications:

Observation#1: Permission Check Types. In Java web applications, the vast majority of permission checks are implemented through if-statements (97.51% in our empirical study).

Observation#2: Permission Check Behaviors. In Java web applications, permission checks implemented with if-statements exhibit two distinct characteristics: their conditions validate user credentials, and their bodies exhibit one of two distinct behaviors: it either interrupts the control flow (e.g., by throwing an exception or returning an error) upon a failed check (84.38%) or executes a privileged operation (e.g., database manipulation) upon a successful one (15.62%).

```
1  if (validate_user_credentials)
2      then perform privileged_operation;
3      or fail(...);
```

Observation#3: Credential Characteristics. In Java web applications, user credentials are represented by program variables, their names have semantic meaning and high-frequency naming words (e.g., *pass*, *permiss*, and *role*); their values are user-uncontrollable.

C. Challenges and Existing Limitations

As observed above, the security paradigm of permission checks in Java web applications conforms to the crucial two phases of credential identification and permission check determination. Therefore, a natural idea arises that we can apply or extend existing PCI techniques (i.e., MACE [12] and MPCHECKER [15]) to the area of Java web security. However, we find this is a challenging task. Two main non-trivial challenges must be effectively addressed.

Challenge I: *How to automatically identify credential candidates without any application-specific inputs?* In Java web applications, the presence of thousands of program variables (class fields) presents a significant challenge in automatically discerning which among them could represent user credentials, especially when there are no obvious developer-defined annotations and application-specific inputs. For instance, MACE [12] leverages the characteristics of the PHP language and developers' practices, where user credentials are typically represented by global or super-global variables (e.g., `$_SESSION["role"]`). Hence, MACE requires end-users to manually label these variables related to authentication or authorization in each target application, employing these application-specific annotations to identify user credentials. While MPCHECKER [15] utilizes the format-specific runtime logs of distributed systems as input. These logs capture nearly all credential-related variables, thereby facilitating the inference of credential candidates within the applications.

[2]The empirical data have been open-sourced along with our artifacts

However, providing the aforementioned inputs within Java web applications is notably challenging. First, in PHP web applications, global variables have distinctive declaration keywords, and the types of super-global variables are limited. These characteristics facilitate end-users in swiftly developing annotations manually. In contrast, the Java language does not have global or super-global variables like PHP for finding credential-related variables (often class fields in Java). While a popular application may have thousands of class fields, and there are no obvious differences in their declarations, this makes manually developing annotations a challenging and error-prone task. Second, different from the distributed systems, the runtime logs in Java web applications mainly capture requests leading to application errors, rather than credential-related variables. These obstacles make it challenging to provide application-specific inputs in Java web applications, hindering the effectiveness of employing existing approaches in credentials inference.

Challenge II: *How to accurately identify permission checks among numerous if-statements that validate the credential candidates?* Java web applications employ if-statements for various purposes, such as input validation, functionality checks, and permission checks. Credential candidates may assume different roles in these statements, depending on the context. Disregarding the context of these if-statements can significantly hinder the effectiveness of identification approaches. For example, the design of both existing techniques [12], [15] directly considers if-statements whose conditions validate credential candidates as permission checks. This permission context-unaware design makes them prone to incorrectly classify conditional statements served for other purposes as permission checks (e.g., input validation or functionality check). As shown in Figure 1, existing approaches may incorrectly identify the format validation (lines 7-9) as permission checks because its conditional statement involves the variable `SysUser.Password`, which can also represent user authentication credentials.

III. METHODOLOGY

A. Key Insights

To address these challenges, we leverage two crucial insights from our key observations in §II-B to develop an effective security analysis approach for Java web applications.

❶ *Java web developers often define the crucial variables (i.e., the class fields accommodating user credentials) with meaningful naming semantics, conventions, and program characteristics.* From a broader perspective, the naming conventions of credentials exhibit certain commonalities across various applications. Specifically, to follow the good programming style and improve code readability [23], developers tend to use words with clear authorization or authentication semantics for naming credential candidates, e.g., `"roleId"` to represent user role and `"contentPermissions"` to represent user permission. Meanwhile, some words, due to their widely acknowledged access control-related semantics, are often chosen by developers in different applications. For instance, the word `"role"` is frequently used to indicate a user's level of identity, so

it is commonly adopted by different developers for naming user credentials in various applications, such as `"userRole"`, `"roleId"` and `"roleUid"`.

From a detailed perspective, a portion of credential candidates also exhibit certain data flow dependencies within a single application. For improved code clarity, developers might use distinct class fields to signify the same credentials when creating different functionalities. For instance, the `"roleId"` credential is used for user authorization checks. For consistency and readability, `"memberId"` is employed for similar purposes in membership functionality code. For user convenience, developers avoid recurring logins by assigning the current user's `"roleId"` to `"memberId"`, thereby establishing a data flow dependency between credential candidates.

❷ *Permission check in Java web is conducted regarding the context features associated with the condition and body.* If-statements play an important role in permission-check determination. In general, they can be divided into two parts based on their code structure: the condition and the body. Both of them should be carefully dealt with.

(1) For the condition part, the user credentials validated in permission checks should be user-uncontrollable. The reasoning is that user credentials should be trusted values stored within the application, used to verify against untrusted user access. However, if these credentials are controllable, it allows attackers to evade access control checks, giving rise to potential vulnerabilities like credential forgery. (2) For the body part, we observed two distinct differences in program context features between permission checks and other validations in terms of data flow and control flow. On the one hand, for data flow, actions in the body of other validations typically involve sanitizing malicious input values (e.g., input validations), but this is rare in the body of permission checks. On the other hand, for control flow, interruption actions in permission checks often possess unique permission-denial context semantics, such as redirecting to a forbidden page or throwing specific warning messages like "permission denied", while other validations do not exhibit.

B. Our Main Idea

Drawing on these dual key insights, we propose a novel security analysis approach that 1) leverages an evolvable credential naming model to automatically infer credential candidates, and 2) combines multi-level context analysis to accurately identify permission checks within the target applications. Below we present more details.

1) For credential candidate inference, our approach integrates commonalities in inter-application naming conventions with intra-application data flow dependencies. Taking a set of target applications as input, it operates based on an initialized naming model (which consists of a few root words commonly used in developers' credential naming conventions), and functions through a three-stage iterative process to automatically evolve the naming model and infer credential candidates.

First, our approach conducts naming model-based semantic analysis on all class fields within each application, thereby identifying an initial set of credential candidates. Then, our approach performs credential semantic correlation analysis to further expand the naming model and the identified credential candidates from both intra-application and inter-application aspects. In the inter-application aspect, our approach performs data flow propagation analysis on these identified credential candidates in each application to find more ones with data flow dependencies. In the intra-application aspect, our approach leverages a correlation algorithm to analyze all identified credential candidates, summarizing commonalities in developer naming conventions and extracting commonly used words. Finally, the iteration is continued until a fixed point is reached, i.e., no new credential candidates can be identified. Thus, this process can smartly and effectively expand the naming model and reduce false negatives in referring credential candidates. Benefiting from such a novel design, our approach can automatically infer credential candidates without necessitating application-specific inputs.

2) For permission check identification, we employ a two-pronged approach to analyze the context features of both the condition and the body within the if-statements. Specifically, for the condition context analysis, our approach first filters out conditions that do not contain already identified credential candidates. Then, for the remaining conditions, our approach conducts a credential controllability analysis by examining whether user-controllable parameters at program entry points have data flow dependencies with these validated credentials. Furthermore, our approach identifies the conditions that solely validate user-uncontrollable credential candidates. In the body context analysis, our approach performs data- and control-flow context analysis to hunt privileged operations and permission-denial interruptions crucial for permission check identification.

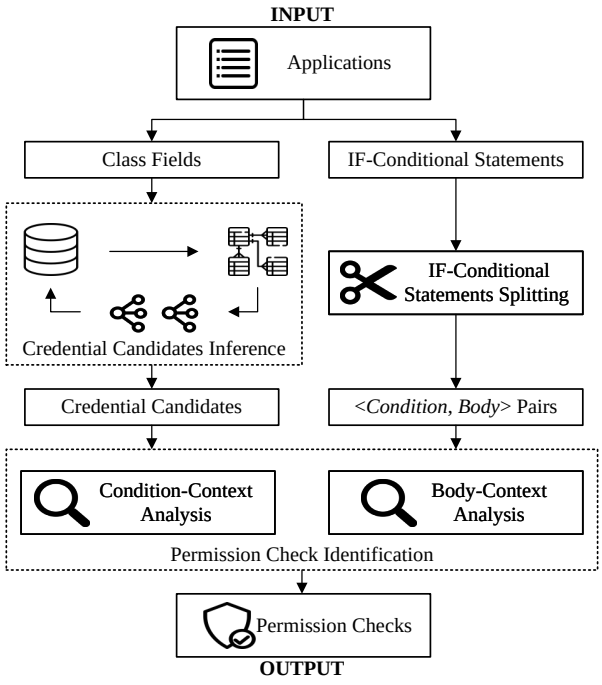


Fig. 2: The Architecture of `PCFinder`.

IV. PCFinder

In this section, we provide a detailed overview of our approach implementation, named `PCFinder`. As shown in Figure 2, `PCFinder` consists of two key modules, each serving the following purposes: *Credential Candidates Inference (§IV-A)* performs multi-level naming semantic analysis and automatically infers the credential candidates within the target applications. *Permission Checks Identification (§IV-B)* leverages a permission-check context analysis and accurately identifies permission checks among numerous if-statements that validate credential candidates.

A. Credential Candidates Inference

This phase infers credential candidates through an evolvable name model. First, `PCFinder` analyzes and collects credential candidates to initialize the subsequent semantic correlation analysis through a naming model. Second, `PCFinder` performs two-pronged semantic correlation analysis based on preliminary credential candidates, to explore additional credentials that have not yet been identified and further expand the naming model. Specifically, following our main idea described in §III-B, `PCFinder` analyzes from two aspects: credential propagation, i.e., the data flow dependencies of user credentials within the application; and naming model expansion, i.e., leverages the commonalities in naming conventions for user credentials between applications. Last, `PCFinder` iteratively performs the previous steps in a loop until one of the following conditions is met: the credential propagation cannot identify new credential candidates, or no new root words can be expanded in the naming model. The workflow is illustrated in Figure 3 and Algorithm 1.

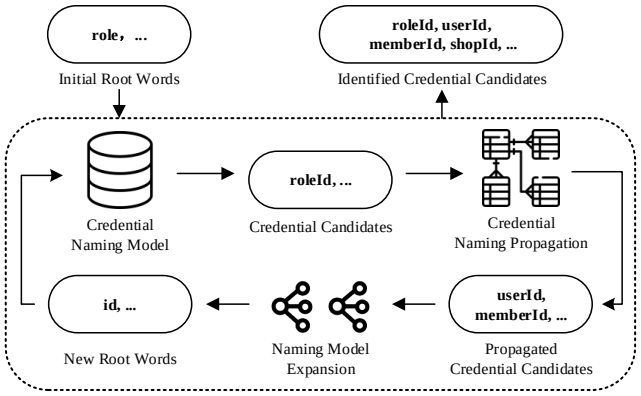


Fig. 3: The workflow of iterative credential candidate inference (e.g., taking the initialized root word `"role"` as an example).

1) Credential Naming Model: The naming model comprises a list of common root words used by developers when naming user credentials. As the sole input for `PCFinder` to initiate the process of inferring credential candidates, we selected three representative words found in credential naming conventions as initial root words, i.e., `"role"`, `"pass"`, and `"permiss"`.

Then, `PCFinder` performs naming model-based semantic analysis on all class fields within the target application based on the naming model, to collect preliminary credential candidates. Considering that relying solely on the naming model for semantic analysis of class fields may introduce false positives (although many of these false positives can be eliminated in subsequent permission check context analysis), `PCFinder`, based on the usage of user credentials, imposes requirements on their possible data types to enhance the precision of the collected results. Specifically, the values of user credentials should generally fall into three categories: numeric, boolean, and character types. For numeric types, they are typically used to represent user permissions or identity levels, for example, `"roleId=1"` or `"permissionLevel=8"`; For boolean types, they are generally used as flags to indicate whether a user meets certain permission conditions, e.g., `"isAdmin=True"`; For character types, they are often used to denote a user's identity or store authentication credentials, like `"role=guest"` or `"password=x...x"`. Therefore, we inspected the official Java tutorial [24] and documentation [25], filtering out seven data types that align with the mentioned three categories, i.e., <`int`,

Algorithm 1 Iterative Credential Candidate Inference

Require: A set of Java applications ($Apps$).
Ensure: A set of identified credential candidates ($Cands$).
$Model \leftarrow$ {"role", "pass", "permiss"};
$Cands \leftarrow \emptyset$;
$updated \leftarrow$ true;
while $updated$ **do**
  $updated \leftarrow$ false;
  $CandsCount \leftarrow |Cands|$;
  $ModelCount \leftarrow |Model|$;
  $Cands \leftarrow$ IdentifyViaModel($Apps, Model, Cands$);
  $Cands \leftarrow$ PropagateCredentials($Cands$);
  $Model \leftarrow$ ExpandModel($Cands, Model$);
  if $|Cands| > CandsCount$ **or** $|Model| > ModelCount$ **then**
    $updated \leftarrow$ true;
  end if
end while
return $Cands$;

function IdentifyViaModel($Apps, Model, Cands$)
  $InitialCands \leftarrow \emptyset$;
  for all class field f in $Apps$ **do**
    if f.name contains a word from $Model$ **and** f.type is qualified **and** $f \notin Cands$ **then**
      $InitialCands \leftarrow InitialCands \cup \{f\}$;
    end if
  end for
  return $Cands \cup InitialCands$;
end function

function PropagateCredentials($Cands$)
  $PropCands \leftarrow \emptyset$;
  for all candidate c in $Cands$ **do**
    $NewFields \leftarrow$ DataFlowAnalysis(c);
    $PropCands \leftarrow PropCands \cup (NewFields \setminus Cands)$;
  end for
  return $Cands \cup PropCands$;
end function

function ExpandModel($Cands, Model$)
  $RWordPairs \leftarrow \emptyset$;
  for all candidate c in $Cands$ **do**
    $RWords \leftarrow$ ExtractRootWords(c.name);
    $RWordPairs \leftarrow RWordPairs \cup \{RWords\}$;
  end for
  $FrequentWords \leftarrow$ Apriori($RWordPairs$);
  return $Model \cup (FrequentWords \setminus Model)$;
end function

short, long, boolean, char, string, enum>. In summary, in this step, PCFinder performs type-sensitive naming model-based analysis on all class fields and considers only those that satisfy two criteria as credential candidates.

2) Credential Semantic Correlation Analysis: In this step, PCFinder conducts credential propagation and naming model expansion to enhance credential candidate inference.

Credential Propagation. For the correlation among user credentials within a single application, PCFinder approaches it through their data flow dependencies and employs credential flow analysis to identify additional credential candidates within the target application. Specifically, PCFinder initiates data flow propagation analysis from preliminary credential candidates. This analysis is performed using the static code analysis engine CodeQL [26]. Considering that class fields in Java applications are generally accessed through getter methods (e.g., "SysUser.userId" is accessed via "SysUser.getUserId()"), PCFinder starts the propagation analysis from all the call sites of those candidates within the application. For the propagation endpoints, PCFinder adopts an approach similar to MACE [12], utilizing the conditions of if-statements as endpoints. During the propagation process, PCFinder only focuses on variables that are of the same type as the source. Overall, this step effectively assists PCFinder in identifying more credential candidates with various names. For instance, as shown in Figure 3, when a user logs into the system, the field "roleId" that represents the authorization credential will be assigned a value. When the user accesses the membership functionality, this value is propagated to another field "memberId". Benefiting from this design, PCFinder can effectively capture the new credential candidate "memberId" whose name was not initially covered by the naming model.

Naming Model Expansion. For the correlation among user credentials across multiple applications, PCFinder delves into the naming conventions established by developers who follow good programming practices. By employing naming commonality analysis, PCFinder infers the common word preferences in the naming of user credentials, thereby further expanding the naming model.

First, to mitigate the impact of stylistic differences in developer naming on the analysis, PCFinder performs splitting and stemming on the names of all class fields. More concretely, given that class fields are generally named following the conventions of camel case naming [27] in Java applications, e.g., "contentPermissions". PCFinder first employs letter-case separation for the name of each class field. For example, separate "contentPermissions" into "content" and "permissions". Then, considering that developers may use different forms of the same word, such as the plural form of "permissions", PCFinder will further transform each separated word into its root word (e.g., "permiss") through the large lexical database, "WordNet" [28], [29]. As a result, the name of the example "contentPermissions" is finally transformed into the root words pair <content, permiss> after this step.

Then, based on the key insight that credentials exhibit commonalities in naming conventions across different applications, PCFinder performs naming commonality analysis on all root word pairs of identified credential candidates. This aims to further identify the words developers prefer when naming credential candidates. To achieve this goal, PCFinder employs the famous association rule mining algorithm, Apriori [30]. For all root word pairs, the Apriori algorithm is applied to discover frequent co-occurrences of words in the naming conventions of credential candidates. Let's illustrate the end-to-end process with an example. Consider a tuple of identified credential candidates <roleId, userId, memberId>. First, PCFinder processes this tuple through splitting and stemming, transforming it into <role, id>, <user, id>, and <member, id>. Then, employing the Apriori algorithm, PCFinder analyzes these pairs and discovers that the word "id" has the highest frequency of co-occurrence among them. This indicates that developers tend to use names with "id" when naming credential candidates.

Finally, PCFinder expands the naming model with the preferred root words that developers tend to use for naming credential candidates. In this step, PCFinder focuses on the top five frequently occurring words in the results of the Apriori algorithm and only adds root words of new ones into the naming model. After these, PCFinder iteratively executes the preceding steps within our naming model in a loop, continuing until no additional new credentials are discovered.

B. Permission Check Identification

Building upon the identified credential candidates, PCFinder further analyzes all if-statements within the target application in this stage to identify permission checks.

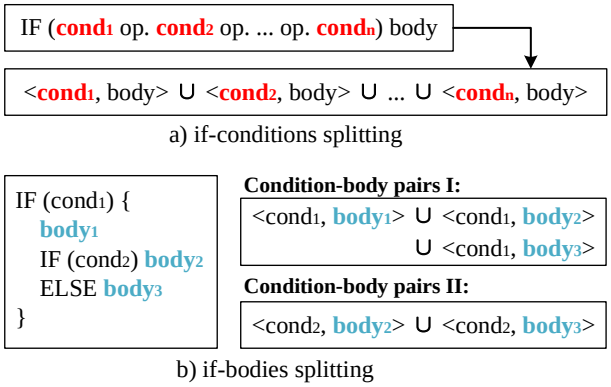


Fig. 4: The splitting of nested if-statements.

1) IF-Statements Splitting: Given the increasing complexity of functionalities in large-scale Java web applications, developers often implement multi-level nested if-statements. This can lead to situations where an if-condition may involve several sub-conditions or an if-body is nested with multiple if-statements. To ensure the accuracy of subsequent analysis, PCFinder first splits all nested if-statements into the atomic level from two aspects: condition splitting and body splitting. Figure 4 illustrates the splitting approach employed by PCFinder.

IF-conditions splitting. Given that an if-condition can involve numerous sub-conditions, which are connected by operators like "||" or "&&". For subsequent fine-grained analysis,

PCFinder splits the nested if-condition based on operators and forms multiple condition-body pairs.

IF-bodies splitting. Likewise, the if-body may also contain diverse if-statements. To perform more accurate context analysis, PCFinder splits the nested body of if-statements into several basic blocks. Then, for each body, PCFinder forms condition-body pairs with their respective conditions for subsequent analysis.

2) Permission-Check Context Analysis: Then, for each condition-body pair, PCFinder conducts permission-check context analysis from two aspects to further determine whether it qualifies as a permission check.

Condition-context analysis. First, PCFinder focuses on the conditions of all if-statements and employs a two-step analysis. For each condition, PCFinder analyzes those that contain credential candidates and filters out those that do not. Furthermore, as user credentials are generally stored as trusted values within the application, PCFinder analyzes the controllability of these credential candidates being validated by the conditions. Leveraging CodeQL's [26] taint-tracking capabilities, this analysis examines whether user-controllable parameters at program entry points have data flow dependencies with the validated credentials. Only when the credential candidates involved in the condition are user-uncontrollable, will PCFinder proceed with subsequent analysis.

Body-context analysis. Second, PCFinder further analyzes the bodies of if-statements whose conditions satisfy requirements. Specifically, PCFinder identifies whether they contain privileged operations or permission-denial interruptions:

❶ *Privileged operation* identification is a technique widely used in related works [11], [12], [13], [14], [15]. In line with these works targeting the web applications [11], [12], PCFinder also considers file handling and database manipulation operations as privileged operations in a similar fashion. Additionally, PCFinder expands its scope to include network request operations, where inadequate permission protection could lead to vulnerabilities, such as the forgery of other users' requests. As a result, we thoroughly review the official Java documentation [25] and relevant sources [31], [32], collecting 42 widely used privileged operations for database manipulation (e.g., `java.sql.Statement.executeQuery`), file handling (e.g., `java.io.FileOutputStream.write`), and network requests (e.g., `java.net.URL`).

❷ *Interruption statements* refer to the statements that restrict program execution from the control flow when the condition validation fails. PCFinder considers the following three types of interruption statements and their permission-denial contexts in Java web applications:

- *Throwing exception statements* are often used after condition validation fails. Developers implement them to interrupt the control flow to protect subsequent privileged operations and notify users of their insufficient or denied access rights. Therefore, PCFinder considers these throw exception statements as interruption statements.
- *Redirect statements* are another common practice used to redirect users with insufficient permissions to a login or forbidden page. Hence, such statements are also categorized as interruption statements by PCFinder.
- *Return statements* are also one of the options developers choose. To enhance code readability and maintainability, some developers choose to encapsulate the aforementioned interruption actions within functions and employ return statements to restrict users with inadequate privileges. For instance, `"return responseUtil.unlogin()"` can be used to restrict access to privileged operations for users who are not logged in. PCFinder thereby also takes return statements as interruption statements.

Permission check determination. Finally, PCFinder will determine if-statements as permission checks if any one of their <`condition, body`> pairs satisfy the following requirements: (1) The condition involves user-uncontrollable credential candidates. (2) the body contains at least one privileged operation or permission-denial interruption statement.

C. Vulnerability Detection

We further integrate two state-of-the-art strategies for detecting broken access control vulnerabilities, i.e., *missing permission check detection* and *inconsistent permission checks detection*. The details are as follows:

Missing Permission Check Detection. PCFinder detects vulnerabilities caused by absent permission checks through the following three steps. First, PCFinder designates the controllers of the application as user entries and identifies operations related to network requesting, database manipulation, and file handling as privileged operations. Then, PCFinder extracts all execution paths by performing a forward analysis from each entry to its corresponding privileged operation. Finally, based on identified permission checks, PCFinder analyzes whether a check exists on each extracted path. In cases where permission checks are absent, PCFinder reports these paths as potential vulnerabilities.

Inconsistent Permission Checks Detection. PCFinder identifies vulnerabilities resulting from inadequate permission checks through the following four steps. First, PCFinder initially collects execution paths based on the application entries and privileged operations. Second, for all identified execution paths, PCFinder groups them according to the privileged operations involved. Specifically, for operations related to file handling and network requests, PCFinder groups the paths based on the method names of their privileged operations. For instance, two execution paths of the same privileged operation would be categorized into one group. For database operations, PCFinder groups paths based on the operation types (i.e., INSERT, SELECT, UPDATE, or DELETE) and the database tables accessed in that operation. For example, if two paths involve database privileged operations that are both of the `"SELECT"` type and are accessing the same table (e.g., `"User"`), then PCFinder would group them together. Third, for each group of paths, with the aid of identified permission checks, PCFinder extracts the permission checks present on each path. After that, for each path, PCFinder extracts the conditions of the permission checks present on them and normalizes them into a tuple. Finally, for paths within the same group, if their tuples are inconsistent, PCFinder will report potential broken access control vulnerabilities.

TABLE I: Breakdown of our evaluation dataset.

Stars	Apps	Fields	IFs	LoCs
High (10,000+)	7	11,735	6,125	291,156
Medium (1,000+)	29	43,561	20,621	877,157
Low (100+)	14	8,038	4,211	159,780
Total	50	63,334	30,957	1,328,093

V. Evaluation

A. Experimental Setup

Prototype. We implemented a prototype of `PCFinder` targeting Java web applications. Specifically, we implement `PCFinder` with two key components: CodeQL [26], which is responsible for collecting code features by several static analysis tasks (e.g., propagation analysis, if-statements analysis), and Python scripts, which parse these features (extracted from CodeQL's results) for algorithms working, e.g., correlation analysis and naming model expansion. In all, the prototype consists of 986 lines of CodeQL code and 2,712 lines of Python code.

Experiments. Our evaluation is organized by answering the following research questions:

- RQ1: How effective is `PCFinder` in inferring credential candidates within target applications? (in §V-B)
- RQ2: How effective is `PCFinder` in identifying permission checks within target applications? (in §V-C)
- RQ3: How do the different components of `PCFinder` contribute to its success? (in §V-D)
- RQ4: How efficient is `PCFinder` in performing the end-to-end analysis? (in §V-E)
- RQ5: What kind of utility could `PCFinder` bring for broken access control vulnerability detection? (in §V-F)
- RQ6: How does `PCFinder` perform in identifying permission checks and vulnerabilities compared to the state-of-the-art work? (in §V-G)

Dataset. We collected 50 widely-used Java web applications from open-source code repositories (e.g., GitHub [22]). The selection process involved three steps. (1) We used a crawler to collect Java applications from open-source code repositories. We focused on applications with over 100 stars and matched them with web-related keywords such as "blog", "mall", and "management". This step collected 802 applications. (2) We then applied a heuristic rule-based pre-scan to analyze the number of if-statements in each application. We retained only those applications with more than 200 if-statements, as `PCFinder` targets in detecting if-implemented permission checks. This step retained 113 projects. (3) Finally, we manually reviewed the remaining applications, excluding any that were not web applications and duplicate entries (e.g., forked projects), leaving a final set of 69 applications. From this final set, we selected 50 representative applications based on their if-statement count. As shown in Table I, among the selected applications are several highly popular ones, with the highest-rated application having over 70k stars. In all, these applications involve 63,334 class fields, 30,957 if-statements, and 1,328,093 lines of code.

Environment. All the experiments in this section are run on a Ubuntu 20.04 machine with an Intel Xeon Gold 6242 processor (64 cores) and 512 GB memory.

B. RQ1: Credential Candidates Inference

We evaluate the effectiveness of `PCFinder` in inferring credential candidates on our datasets. Overall, `PCFinder` effectively identified 1,185 credential candidates with 82 false positives, achieving a precision rate of 93.53%.

False Positive Analysis. Our further analysis showed that false positives stem from two causes. On one hand, 65 false positives are introduced by `PCFinder` during type-sensitive naming model-based analysis. The names of these cases also include high-frequency words used by developers in naming credentials. On the other hand, 17 false positives are introduced by `PCFinder` during credentials propagation due to over-tainting. However, it is important to note that the influence of these false positives on the subsequent identification of permission checks is negligible, given their inability to satisfy the requirements of the following context-aware analysis.

Effectiveness of Naming Model Expansion. In the design of the credential candidate inference module, we introduced a novel iterative algorithm for expanding the naming model, enhancing the inference capability of credential candidates. To demonstrate the effectiveness of the proposed algorithm, we conducted an evaluation experiment on our dataset. First, we used the credential candidates identified by `PCFinder` across the entire dataset as the ground truth. Then, by running `PCFinder` iteratively with gradually increasing numbers of applications as input, we evaluated the CFR (Credentials Found Rate) of `PCFinder` in inferring credential candidates. Specifically, we define $C(n, i)$ as the credential candidates found by `PCFinder` in the ith application among n input applications. The CFR of `PCFinder` in inferring credential candidates under different input application numbers n can be defined as follows:

$$CFR(n) = \frac{\sum_{i=1}^{n} C(n,i)}{\sum_{i=1}^{n} C(50,i)}$$

The evaluation results are shown in Figure 5. Clearly, when only a single application is input, `PCFinder` achieves a CFR of only 33% in identifying credential candidates, missing many true credentials within the applications. This indicates that relying solely on the initialized naming model for single-application analysis has limited effectiveness. As the number of input applications increases, the effectiveness of `PCFinder` initially shows a sharp increase, with CFR gradually converging after around 35 applications. This suggests that there are commonalities in the credential naming conversion, and developers typically use a limited set of root words for credential naming.

C. RQ2: Permission Checks Identification

We evaluate the effectiveness of `PCFinder` in identifying permission checks across our dataset. In all, `PCFinder` effectively identified 486 permission checks throughout the entire dataset with a precision rate of 91.35%.

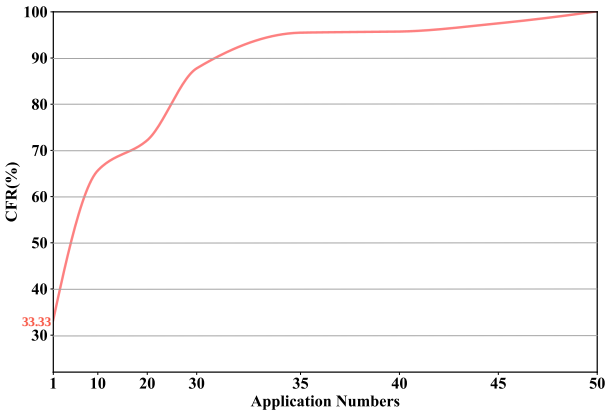


Fig. 5: The effectiveness of naming model expansion.

False Positive Analysis. We meticulously analyzed 46 false positives introduced by `PCFinder` in the permission check identification stage within our dataset. The primary causes can be attributed to three aspects. Firstly, 16 false positives were caused by the erroneously identified credential candidates previously mentioned. Unlike other false positives in credential candidates, these cases also met the requirements of permission check context analysis, thereby leading to false positives in permission checks. Secondly, 26 false positives were introduced in condition-context analysis. When performing controllable analysis, due to the inherent challenges of static analysis, `PCFinder` assumes that variables retrieved from the database are user-uncontrollable. However, in reality, since these variables undergo no additional manipulation from user input to storage in the database and then retrieval, they are actually user-controllable values. This ultimately leads to false positives in the identification of permission checks. Lastly, the remaining 4 false positives in the body-context analysis were attributed to behaviors closely resembling permission-denial interruptions.

Permission-Checking API Identification. Given that permission checks implemented using if-statements have been widely applied as a common security mechanism, they may not necessarily be directly placed before privileged operations but can also be deployed through permission-checking API calls. Therefore, we followed the same idea as existing techniques [15], i.e., permission-checking APIs are recognized via function summary: a method is a permission-checking API if there exists a permission check post-dominating its entry. This enables `PCFinder` to identify such APIs based on the already identified permission checks.

In total, based on previously identified permission checks, `PCFinder` further discovered 65 permission-checking APIs across 23 applications. Taking lamp-boot [33] as an example, its developers prefer using permission-checking APIs for access control. Although the application has only 7 if-implemented permission checks, the wrapper permission-checking APIs are called 59 times within the program.

D. RQ3: Ablation Study

In this experiment, we evaluate the contribution of each component of `PCFinder`.

Ground-Truth. For a comprehensive evaluation involving precision and recall metrics, a dataset with labeled user credentials and permission checks is required. Given the absence of such a publicly available dataset, we need to manually construct a ground truth. Considering the extensive human efforts required to label all credentials and permission checks within an application, we selected two applications with the highest number of if-statements from the dataset as the ground-truth set, i.e., PublicCMS [34] and FlyCMS [35]. To guarantee the quality of labeling the user credentials and permission checks of the ground truth, three authors have participated in the process, each with a minimum of 4 years of expertise in web security. The entire process costs more than 165 human hours. Specifically, the analysts follow the following process to label the ground-truth dataset:

- First, to facilitate the manual labeling of credential candidates and permission checks, the analysts employ `PCFinder` to extract all class fields and if-statements within the applications. In total, the first step extracts 3,357 class fields and 4,157 if-statements. Considering that this step is automated by `PCFinder`, the time required can be negligible.
- Second, for each class field, two analysts independently inspected and labeled whether they were credential candidates. In cases where the field yielded different labeling results, a third analyst would be involved to engage in discussions and determine the appropriate label. As a result, among the 3,357 class fields, the second step successfully labels 71 credential candidates. This step takes about 15 human hours.
- Third, for each if-statement, three analysts followed a similar method in the second step, where two analysts independently inspected the condition and body context of the if-statements, and then labeled whether they were permission checks. For the different labeling results, a third analyst joins the discussion to reach a consensus. In all, the third step successfully labels 99 permission checks from 4,157 if-statements. This step takes about 150 human hours.

As a result, the ground-truth dataset consists of 71 credential candidates and 99 permission checks from 3,357 class fields with 4,157 if-statements.

Variants. Then, we construct two variants of `PCFinder` and compare them with `PCFinder` in our ground-truth dataset. Specifically, we respectively disabled two key components of `PCFinder` to construct the following two variants:

- *PCFinder-OnlyInitialModel*: In this variant, we disabled the key component of the *Credential Candidates Inference* module, namely credential semantic correlation analysis, leading to the variant relying solely on the initial naming model to identify credential candidates in the target applications.
- *PCFinder-ContextUnaware*: In this variant, we disabled the key component for context analysis within the *Permission Check Identification* module, leading to the variant directly identifying if-statements whose conditions involve validating credential candidates as permission checks.

Comparison Results. Table II breaks down the comparison results of `PCFinder` and its two variants against the ground-

TABLE II: The results of our ablation study evaluating credential candidates Inference and permission checks identification for two variants of PCFinder compared with PCFinder in the ground-truth dataset.

Variants	Credential Candidates Inference					Permission Checks Identification				
	TP	FP	FN	Prec(%)	Recall(%)	TP	FP	FN	Prec(%)	Recall(%)
PCFinder-OnlyInitialModel	10	1	63	90.91%	13.70%	36	0	68	100%	34.61%
PCFinder-ContextUnaware	70	2	1	97.22%	98.59%	96	67	3	58.90%	96.97%
PCFinder	70	2	1	97.22%	98.59%	96	6	3	94.12%	96.97%

truth dataset. The results provide strong quantitative evidence supporting the two key insights underlying PCFinder's design. First, by disabling credential semantic correlation analysis (PCFinder-OnlyInitialModel), the recall of credential identification is reduced by nearly 620% compared to the PCFinder (13.70% vs. 98.59%), substantiating our first key insight that a deep semantic analysis is necessary to overcome the challenge of identifying diverse credential candidates. Second, by disabling permission check context analysis (PCFinder-ContextUnaware), the precision of permission check identification fell by approximately 60% (58.90% vs. 94.12%). This confirms our second insight: that context is indispensable for accurately distinguishing true permission checks from the multitude of other if-statements in complex applications, thereby addressing the challenge of accurate identification.

False Negative Analysis. In addition to the evaluation of PCFinder's precision mentioned in the previous part, we also evaluate the recall of PCFinder in inferring credentials and permission checks across the ground truth. The evaluation results show that PCFinder can achieve excellent performance in these two aspects, with 98.59% and 96.97% respectively. As for the false negatives reported by it, the causes are analyzed as follows:

For credential inference, we carefully analyzed the only one false negative reported by PCFinder. The main reason stems from the naming convention of the credential candidate being uncommon across the entire dataset. In PublicCMS, the user class `SysUser` includes a boolean-type field named `ownsAllRight`, which is used to represent whether a user has all permissions within the entire system. However, since its instance does not have data flow dependencies with other credential candidates and its naming convention is also not common in the entire dataset, PCFinder missed it.

For permission check identification, the 3 false negatives were caused by the same reason: as previously mentioned, PCFinder failed to infer the credential candidate `"ownsAllRight"`, so that leading to missing these three permission checks. This also fully reflects the importance of a more comprehensive identification of credentials for locating permission checks.

E. RQ4: Efficiency

We evaluate the performance of PCFinder across our dataset. In total, PCFinder required around 3.18 hours to complete the task of identifying permission checks in our dataset. The efficiency of PCFinder stems from two key factors. Firstly, in its design, PCFinder's analysis is localized, focusing more on the credential candidates and code context around if-statements (typically the controller's parameters → if-condition paths), making it relatively lightweight. Secondly, we observed that many applications include code generated by automatic code generators, such as MyBatis Generator [36]. This generated code comprises more than half of the source code, which is not within the scope of PCFinder's analysis. This significantly reduces the analysis workload of PCFinder. Overall, PCFinder is a lightweight framework tailored for permission check identification.

F. RQ5: Vulnerability Detection

We further evaluate the effectiveness of PCFinder in detecting broken access control vulnerabilities.

Vulnerability Disclosure. In all, PCFinder identified 69 potential vulnerabilities from our dataset, consisting of 46 from missing permission checks detection and 23 from inconsistent permission checks detection. We then manually investigated the identified vulnerabilities to confirm their exploitability. In practice, verifying access control vulnerabilities is less complex than exploiting injection-style flaws, as it does not require the construction of sophisticated payloads. The process typically involves directly accessing privileged functions or making minor modifications to parameter values (e.g., changing a userid from 1 to 2 or a discount parameter from false to true). However, it demands considerable time to configure the test environment. This includes not only deploying the application but also registering users with different roles and setting up various resources (e.g., products and articles) for them. This entire process involved three authors and took approximately 10 person-hours. The bulk of this time (about 8 hours) was dedicated to environment setup, while the remaining 2 hours were spent on the actual verification.

As a result, we manually confirmed 58 0-day vulnerabilities, impacting 14 applications. Specifically, these vulnerabilities pose significant risks, as attackers could exploit them to launch various attacks against other users, including *stealing user privacy information (R1)*, *tampering with user data (R2)*, and even *abusing payment functionalities (R3)*. According to our statistics, there are 11 applications that might be vulnerable to R1, and 5 applications that could be susceptible to R2. This means that the user's phone number, id card information, or address data of these applications can be stolen or tampered with by attackers. Furthermore, there are 4 applications that could potentially be exposed to R3, which could lead to attackers being able to implement zero-cost payments on the websites deployed with these applications.

To address these risks, we promptly reported the vulnerabilities to the developers of the respective vulnerable applications. 12 vulnerability reports received prompt responses from the developers and the vulnerabilities were actively fixed, while the remaining vulnerability reports are still in the process of communication and resolution with the developers. As of now, we have received 30 CVE identifiers (the details are shown in Table III). Note that to prevent unpatched vulnerabilities from posing risks, we follow the developers' requests and ethical considerations, ensuring the anonymization of crucial information related to the vulnerabilities.

TABLE III: Disclosed vulnerabilities found by PCFinder.

# CVE-ID[1]	# Risk Types[2]	# Apps[1]	# Stars
CVE-2023-3**16	R1	m***	High (10,000+)
CVE-2023-3**68	R2	m***	High (10,000+)
CVE-2023-3**18	R3	n***	High (10,000+)
CVE-2023-3**29	R2	x***	Medium (1,000+)
CVE-2023-3**31	R2	x***	Medium (1,000+)
CVE-2023-3**32	R1	x***	Medium (1,000+)
CVE-2023-3**33	R2	x***	Medium (1,000+)
CVE-2023-3**30	R1	x***	Medium (1,000+)
CVE-2023-4**63	R3	x***	Medium (1,000+)
CVE-2024-2**02	R1	s***	Medium (1,000+)
CVE-2024-2**06	R1	p***	Medium (1,000+)
CVE-2024-2**07	R1	p***	Medium (1,000+)
CVE-2024-2**09	R1	s***	Medium (1,000+)
CVE-2024-2**12	R1	s***	Medium (1,000+)
CVE-2024-2**93	R1	b***	Medium (1,000+)
CVE-2023-3**25	R2	y***	Medium (1,000+)
CVE-2023-3**21	R1	n***	Low (100+)
CVE-2023-3**22	R2	n***	Low (100+)
CVE-2023-3**23	R3	n***	Low (100+)
CVE-2023-4**64	R2	n***	Low (100+)
CVE-2023-3**72	R1	d***	Low (100+)
CVE-2023-3**73	R1	m***	Low (100+)
CVE-2023-3**74	R2	m***	Low (100+)
CVE-2023-3**76	R2	m***	Low (100+)
CVE-2023-4**65	R1	s***	Low (100+)
CVE-2023-4**66	R1	s***	Low (100+)
CVE-2023-4**67	R1	s***	Low (100+)
CVE-2023-4**68	R3	t***	Low (100+)
CVE-2023-4**69	R1	t***	Low (100+)
CVE-2023-4**64	R3	t***	Low (100+)

[1]For ethical considerations and the developers request, we anonymized the entire CVE identifiers and names of the vulnerable applications.
[2]R1 represents vulnerabilities that can be exploited to steal user privacy information; R2 represents vulnerabilities that can be used to tamper with user data; R3 represents exploits of abusing payments vulnerabilities.

False Positive Analysis. We have thoroughly analyzed the 11 false positives that were found in the process of access control vulnerability detection, their causes are as follows:

- *Challenging to discern the intentions of developers (8 FPs).* One of the inherent challenges in detecting access control vulnerabilities is understanding the developers' intentions regarding access control. In the context of business logic, there may be certain public resources that can be accessible to any user. For instance, product information in e-commerce applications should be accessible to anyone and thus, does not require access control. However, this can be challenging for program analysis to discern, leading to false positives in vulnerability detection.
- *Protected by post-execution permission checks (3 FPs).* For the other three false positives, we found that they are both protected by post-execution permission checks, thus preventing the exploitation of vulnerabilities. In general, developers deploy permission checks before executing privileged operations to prevent malicious exploitation by attackers. However, in certain cases, these permission checks can also be placed after the privileged operations as a protective measure. For example, an attacker might exploit privileged operations to query sensitive information without authorization. However, after the query, the application employs permission checks to ascertain whether the user is authorized to receive the query results. If not authorized, the results are not returned to the user, thus preventing the attacker's malicious exploitation. Given that these permission checks are implemented as post-protection measures, they fall outside the scope of existing strategies, resulting in false positives.

G. RQ6: Comparison with MACE

Considering that MACE shares our focus on web applications and represents the most state-of-the-art technique in this area, we compared the effectiveness of PCFinder and JMACE (the version of MACE that we ported to Java web applications) in identifying permission checks and access control vulnerabilities on our dataset. The detailed results are presented in Table IV.

JMACE Setup. MACE [12], which requires the identification of permission checks and subsequent detection of access control vulnerabilities in PHP Web applications, is closely related to our work. Unfortunately, it is not open-source. As a result, we re-implemented and adapted it for Java web applications, under the name JMACE. Given that MACE requires end-users to provide manual annotations for each target application, which presents a significant challenge, especially in the context of Java web applications (as detailed in §II-C). Considering our dataset encompasses 50 applications, it's unfeasible to annotate and provide application-specific annotations for each one individually. Therefore, we opted for a compromise - using the manual annotations disclosed in MACE's original paper appendix as input.

Benchmark. Comparing the accuracy of each work requires a comprehensive enumeration of all permission checks and vulnerabilities within our dataset, which is impractical. Therefore, we have established a benchmark by aggregating all permission checks and vulnerabilities identified by both PCFinder and JMACE in our dataset. It is important to note that each permission check and vulnerability included in the benchmark underwent careful scrutiny. This was accomplished by manually analyzing the reports from each work and confirming them as true positives. In total, the benchmark consists of 486 permission checks and 58 vulnerabilities. This indicates that the cases detected by PCFinder cover all that could be discovered by JMACE.

Comparison of Permission Checks Identification. First, we compared the effectiveness of PCFinder and JMACE in identifying permission checks. Table IV presents the detailed results. Benefiting from the evolvable credentials inference strategy, PCFinder successfully locates more permission check candidates compared to JMACE. Coupled with

TABLE IV: Comparison of effectiveness between PCFinder and JMACE in identifying permission checks and vulnerabilities.

Baselines	Permission Check Identification					Vulnerability Detection				
	TP	FP	FN	Prec(%)	Recall(%)	TP	FP	FN	Prec(%)	Recall(%)
JMACE	297	71	189	80.71%	61.11%	26	24	32	52.00%	44.83%
PCFinder	486	46	0	91.35%	100.00%	58	11	0	84.06%	100.00%

PCFinder's finer-grained context analysis, in the permission checks determination phase, PCFinder's precision and recall outperform JMACE by 21.19% and 63.64% respectively. The more precise and abundant identification of permission checks lays a solid foundation for subsequent vulnerability detection.

Given that we have previously analyzed the causes of false positives reported by PCFinder, in this part, we will elaborate on the causes of false positives and false negatives of MACE. For the 71 false positives, we found that their causes originate from two aspects. On one hand, 44 false positives are due to JMACE not considering the behaviors of the if-body statements at the design level, and only introducing permission checks based on the analysis of the if-condition statements. On the other hand, JMACE, like PCFinder, introduced 27 false positives due to incorrect identification of credentials. However, it's worth mentioning that PCFinder only had 16 false positives in this regard. The reason originates from JMACE providing an annotation, i.e., user, that is prone to causing false positives in credential inference. This affected JMACE's identification of permission checks, leading to more false positives. For the 189 false negatives, we found that all the causes for JMACE stem from the same root: *the failure to identify the appropriate credentials.* For instance, the pre-annotated credentials of JMACE missed a crucial credential, namely, tenantId, which consequently led to missing 36 permission checks related to it. On the contrary, with its evolvable and automatic credential inference strategy, PCFinder successfully captured this credential and identified the corresponding permission checks.

Comparison of Vulnerability Detection. Then, based on the identified permission checks, we further compared the effectiveness of PCFinder and JMACE in detecting access control vulnerabilities on our dataset. Table IV presents the detailed results. Thanks to the precision of PCFinder and the identification of an additional 189 permission checks, its effectiveness in vulnerability detection notably surpasses that of JMACE: not only detects all the vulnerabilities found by JMACE but also reveals 32 real vulnerabilities that JMACE failed to uncover. In terms of precision and recall, PCFinder outperforms JMACE by 61.65% and 123.06%, respectively.

We further analyzed the causes of false positives and negatives during the vulnerability detection process. For the 24 false positives reported by JMACE, we found that 11 of them have the same cause as PCFinder, which has been described in detail in §V-F. As for the remaining 13 false positives, we found that they all come from the same reason: *failure to identify the permission checks on the code path.* In particular, JMACE, due to its more limited permission checks identification capability compared to PCFinder, failed to identify the permission checks on the code path. This led to a mistaken belief that there was a lack of access control on protected paths, thus resulting in the reporting of false positives. For the 32 false negatives, we found that the reason JMACE missed these vulnerabilities all originated from *incorrectly taking normal if-statements as permission checks*, thereby mistakenly determining that access control existed on unprotected paths. However, with more precise permission checks identification, PCFinder accurately detected these unprotected paths and reported potential vulnerabilities.

VI. Discussion

Application-General Modeling. In the design and implementation of the PCFinder prototype, we introduced some manual modeling, i.e., naming model construction and widely used privileged operations described in §IV-A. However, it's worth noting that these models only involved minimal human efforts and are application-general, meaning they do not depend on any application-specific features. Therefore, unlike existing approaches, when testing different applications, PCFinder does not require any additional manual annotations or information provided by developers. In addition, application-general modeling is flexible, allowing end-users to modify it according to their preferences or maintain it during long-term use. This flexibility makes PCFinder more customizable for their specific analysis scenarios.

Inadequate Permission Check Analysis. Although PCFinder is capable of detecting vulnerabilities stemming from inadequate permission checks and successfully identified 23 such vulnerabilities during the evaluation phase, we acknowledge that this detection strategy is sound but not complete. It cannot verify the security of unique program entry-privileged operation paths that appear only once within the program. The core challenge here stems from the difficulty of inferring the access control intentions of different functionalities from developers without a pre-defined application access control specification. Consequently, identifying inconsistencies in permission checks across different entry points to the same privileged operation serves as an alternative approach.

Generalization. Broken access control vulnerabilities are prevalent across web applications developed in various programming languages (e.g., PHP or NodeJS), not limited to Java web applications. Therefore, accurately identifying permission checks is a common challenge in detecting vulnerabilities within these applications. Although the prototype of PCFinder is implemented specifically for Java web applications, its high-level design is almost independent of any specific features of the Java language. Therefore, we believe the key idea of PCFinder can be generalized to applications developed in other languages.

Source Code as Analysis Target. Our current implementation of `PCFinder` operates on Java source code, utilizing the CodeQL engine [26] to perform the necessary static analyses, such as data flow and taint tracking. We chose source code as the primary analysis target because our core methodology relies on semantic information, such as the naming conventions of variables (i.e., class fields), to infer credential candidates. This semantic information is readily available in source code but is often obfuscated or entirely lost during compilation into Java bytecode. While powerful bytecode analysis frameworks like Soot [37] and WALA [38] exist, it would require sophisticated techniques to recover variable names and other semantic cues, which is a non-trivial research problem in itself. Therefore, our focus on source code is a deliberate design choice to ensure the high accuracy of our credential naming model.

Future Work. With the rise of Large Language Models (LLM) and their advantages in natural language understanding, leveraging them as a complementary component to enhance `PCFinder` is a promising direction for future work. For example, during the "Credential Candidates Inference" phase, `PCFinder` relies on a structured methodology to comprehend the semantics of variable names. This process involves extracting word stems, utilizing an evolvable naming model, and applying association rule mining algorithms to identify credential candidates based on common developer conventions across multiple applications. While highly effective, this approach could be complemented by an LLM that leverages its superior natural language understanding. For instance, an LLM could be tasked with performing direct semantic analysis on variables that `PCFinder` did not flag as credential candidates. This approach could uncover credentials that are currently missed due to uncommon naming conventions (e.g., the false negative in Section V.D, the credential `ownsAllRight`).

VII. Related Work

Access Control Vulnerability Detection. In addition to the existing work we discussed in §I, there is also a body of work [39], [40], [41], [42], [43], [44], [45], [46], [47] that detects access control vulnerabilities from other aspects. In the realm of static analysis, Sun *et al.* [39] focuses on user roles within the applications. Relying on developers specifying application entry points and role-based states, this approach establishes sitemaps for various user roles in the application and detects access control vulnerabilities by identifying users who violate the sitemaps. FINAD [42] focuses on identifying access control vulnerabilities by assessing the inconsistency between software design documents and their implementation. SPACE [45] focuses on detecting broken access control vulnerabilities in Ruby on Rails applications. Its high-level approach is to identify code within the target program that violates a catalog of pre-defined access control patterns, reporting these violations as potential vulnerabilities. However, this line of work is limited by the need for application-specific inputs or specifications, making it challenging to evaluate on a large scale of applications. More recently, MOCGuard [48] (for Java) and BOLARAY [49] (for PHP) were proposed to detect missing owner check vulnerabilities. Both employ a database-centric methodology, inferring authorization models from database schemas to find vulnerabilities in SQL-implemented checks. In contrast, `PCFinder` is code-centric and identifies the more prevalent permission checks implemented within if statements in Java web applications. It achieves this through code context and semantic analysis, independent of any database structure. As we describe in §IV-B, privileged operations are not limited to database manipulations but also include other types, such as file operations. Consequently, approaches focusing solely on the database would fail to analyze these cases.

Regarding dynamic analysis, several studies have also detected access control vulnerabilities by analyzing changes in responses to testing requests. Block [40] and DetLogic [41] detect access control vulnerabilities by identifying potential state violations. ReACP [46] focuses on reverse-engineering access control policies by systematically generating access requests through combinatorial testing and using machine learning to infer the implemented policies from the results. Rennhard *et al.* [47] present a black-box method that directly detects access control vulnerabilities by crawling an application with different user roles and replaying requests to identify unauthorized access. LogicScope [43] and BATMAN [44] focused on constructing test inputs and identifying potential access control vulnerabilities by leveraging unexpected inputs. However, achieving comprehensive code coverage of the tested application has always been a challenge for dynamic analysis techniques, making it difficult compared to static analysis techniques to ensure that the target application has been thoroughly analyzed.

Web Vulnerabilities Detection. There are numerous efforts dedicated to proposing techniques for automatically detecting vulnerabilities in web applications. Based on their detection strategies, we can categorize them into three main lines: Static analysis [30], [50], [51], [52], [53], [54], [55], [56], [57] is renowned for its high code coverage, but due to inherent challenges caused by program dynamic characteristics, it is prone to reporting false positives; dynamic analysis [58], [59], [60], [61], [62], [63], [64], although more precise in its detection results, often leads to false negatives blamed for its limited code coverage explored; hybrid analysis [65], [66], [67], [68], [69] might attempt to combine the strengths of both approaches, but as it still incorporates white-box components, it also faces the challenges inherent in static analysis. Regardless, we believe that accurate permission check identification results can effectively aid in the application of these techniques for access control vulnerability detection.

VIII. Conclusion

Java web applications are crucial due to their robustness, scalability, and extensive ecosystem. However, in past years, Java web applications often suffered serious access control issues. In this work, we first understood how Java web conducts permission checks. Then, we propose a novel permission check identification solution, named `PCFinder`, utilizing multi-level semantics- and context-analysis to automatically and accurately identify permission checks within Java web applications, without any need for application-specific inputs. By

applying PCFinder to real-world popular web applications, we find access control vulnerabilities are prevalent and may cause serious security consequences. With responsible vulnerability disclosure, we discovered 58 high-risk vulnerabilities in real-world applications, with 30 CVE identifiers assigned.

REFERENCES

[1] "Amazon," https://www.amazon.com/.
[2] "Microsoft," https://www.microsoft.com/.
[3] "eBay," https://www.ebay.com/.
[4] "Alibaba," https://www.alibaba.com/.
[5] "Java Development Framework Spring," https://spring.io/.
[6] M. Chen, T. Tu, H. Zhang, Q. Wen, and W. Wang, "Jasmine: A Static Analysis Framework for Spring Core Technologies," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.
[7] "Apache Struts Framework," https://struts.apache.org/.
[8] "The OWASP Top 10 - 2013," https://owasp.org/www-pdf-archive/OWASP_Top_10_-_2013.pdf, 2013.
[9] "The OWASP Top 10 - 2017," https://owasp.org/www-project-top-ten/2017/Top_10, 2017.
[10] "OWASP Top 10," https://owasp.org/Top10/, 2021.
[11] S. Son, K. S. McKinley, and V. Shmatikov, "Fix Me Up: Repairing Access-Control Bugs in Web Applications." in *NDSS*. Citeseer, 2013.
[12] M. Monshizadeh, P. Naldurg, and V. Venkatakrishnan, "Mace: Detecting Privilege Escalation Vulnerabilities in Web Applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 690–701.
[13] Y. Shao, Q. A. Chen, Z. M. Mao, J. Ott, and Z. Qian, "Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework," in *NDSS*, 2016.
[14] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "Pex: A Permission Check Analysis Framework for Linux Kernel," in *28th USENIX Security Symposium*, 2019.
[15] J. Lu, H. Li, C. Liu, L. Li, and K. Cheng, "Detecting Missing-Permission-Check Vulnerabilities in Distributed Cloud Systems," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2145–2158.
[16] Y. Gu, X. Tan, Y. Zhang, S. Gao, and M. Yang, "Epscan: Automated detection of excessive rbac permissions in kubernetes applications," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 3199–3217.
[17] X. Zhang, Z. Yu, X. Li, C. Zhang, C. Sun, N. Zhang, and R. H. Deng, "Understanding the bad development practices of android custom permissions in the wild," *IEEE Transactions on Dependable and Secure Computing*, 2025.
[18] B. Yuan, Z. Song, Y. Jia, Z. Lu, D. Zou, H. Jin, and L. Xing, "Mqttactic: Security analysis and verification for logic flaws in mqtt implementations," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 2385–2403.
[19] S. Son, K. S. McKinley, and V. Shmatikov, "Rolecast: Finding Missing Security Checks When You Do Not Know What Checks Are," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, 2011, pp. 1069–1084.
[20] "Android Permission Overview," https://developer.android.com/guide/topics/permissions/overview.
[21] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, "AutoISES: Automatically Inferring Security Specification and Detecting Violations." in *USENIX Security Symposium*, 2008, pp. 379–394.
[22] "Github," https://github.com/.
[23] "Java Naming Conventions," https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html.
[24] "Java Tutorials," https://docs.oracle.com/javase/tutorial/java/data/index.html.
[25] "Java Documentation," https://docs.oracle.com/en/java/.
[26] "CodeQL," https://codeql.github.com/.
[27] "Google Java Style Guide," https://google.github.io/styleguide/javaguide.html.
[28] "WordNet," https://wordnet.princeton.edu/.
[29] "The Document of Python Library: WordNet," https://www.nltk.org/howto/wordnet.html.
[30] R. Agrawal, R. Srikant *et al.*, "Fast Algorithms for Mining Association Rules." in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215. Santiago, Chile, 1994, pp. 487–499.
[31] "The Official Document of MyBatis," https://mybatis.org/mybatis-3/.
[32] "Spring Framework," https://spring.io/.
[33] "lamp-boot," https://github.com/zuihou/lamp-boot.
[34] "PublicCMS," https://www.publiccms.com/.
[35] "FlyCMS," https://github.com/sunkaifei/FlyCms.
[36] "MyBatis Generator," https://mybatis.org/generator/.
[37] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, vol. 15, no. 35, 2011.
[38] "WALA," https://github.com/wala/WALA.
[39] F. Sun, L. Xu, and Z. Su, "Static Detection of Access Control Vulnerabilities in Web Applications," in *20th USENIX Security Symposium (USENIX Security 11)*, 2011.
[40] X. Li and Y. Xue, "Block: A Black-box Approach for Detection of State Violation Attacks Towards Web Applications," in *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011, pp. 247–256.
[41] G. Deepa, P. S. Thilagam, A. Praseed, and A. R. Pais, "DetLogic: A Black-box Approach for Detecting Logic Vulnerabilities in Web Applications," *Journal of Network and Computer Applications*, vol. 109, pp. 89–109, 2018.
[42] M. Ghorbanzadeh and H. R. Shahriari, "Detecting Application Logic Vulnerabilities via Finding Incompatibility between Application Design and Implementation," *IET Software*, vol. 14, no. 4, pp. 377–388, 2020.
[43] X. Li and Y. Xue, "LogicScope: Automatic Discovery of Logic Vulnerabilities within Web Applications," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, 2013, pp. 481–486.
[44] X. Li, X. Si, and Y. Xue, "Automated Black-box Detection of Access Control Vulnerabilities in Web Applications," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, 2014, pp. 49–60.
[45] J. P. Near and D. Jackson, "Finding security bugs in web applications using a catalog of access control patterns," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 947–958.
[46] H. T. Le, L. K. Shar, D. Bianculli, L. C. Briand, and C. D. Nguyen, "Automated reverse engineering of role-based access control policies of web applications," *Journal of Systems and Software*, vol. 184, p. 111109, 2022.
[47] M. Rennhard, M. Kushnir, O. Favre, D. Esposito, and V. Zahnd, "Automating the detection of access control vulnerabilities in web applications," *SN Computer Science*, vol. 3, no. 5, p. 376, 2022.
[48] F. Liu, Y. Shi, Y. Zhang, G. Yang, E. Li, and M. Yang, "Mocguard: Automatically detecting missing-owner-check vulnerabilities in java web applications," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 903–919.
[49] Y. Huang, C. Shi, J. Lu, H. Li, H. Meng, and L. Li, "Detecting broken object-level authorization vulnerabilities in database-backed applications," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 2934–2948.
[50] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and Flexible Discovery of PHP Application Vulnerabilities." in *2017 IEEE european symposium on security and privacy (EuroS&P)*. IEEE, 2017, pp. 334–349.
[51] J. Dahse, N. Krein, and T. Holz, "Code Reuse Attacks in PHP: Automated pop chain generation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 42–53.
[52] J. Dahse and T. Holz, "Simulation of Built-in PHP Features for Precise Static Code Analysis." in *NDSS*, vol. 14, 2014, pp. 23–26.
[53] ——, "Static Detection of Second-Order Vulnerabilities in Web Applications." in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 989–1003.
[54] C. Luo, P. Li, and W. Meng, "Tchecker: Precise Static Inter-Procedural Analysis for Detecting Taint-Style Vulnerabilities in PHP Applications," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2175–2188.
[55] B. B. Nielsen, B. Hassanshahi, and F. Gauthier, "Nodest: Feedback-driven Static Analysis of Node.js Applications," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 455–465.

[56] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.

[57] M. Shcherbakov and M. Balliu, "Serialdetector: Principled and Practical Exploration of Object Injection Vulnerabilities for the Web," in *Network and Distributed Systems Security (NDSS) Symposium 202121-24 February 2021*, 2021.

[58] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 523–538.

[59] K. Drakonakis, S. Ioannidis, and J. Polakis, "ReScan: A Middleware Framework for Realistic and Robust Black-box Web Application Scanning." in *NDSS*, 2023.

[60] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black Widow: Blackbox Data-Driven Web Scanning," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1125–1142.

[61] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow, "Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1757–1771.

[62] E. Trickel, F. Pagani, C. Zhu, L. Dresel, G. Vigna, C. Kruegel, R. Wang, T. Bao, Y. Shoshitaishvili, and A. Doupé, "Toss a Fault to Your Witcher: Applying Grey-box Coverage-Guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities," in *2023 IEEE symposium on security and privacy (SP)*. IEEE, 2023, pp. 2658–2675.

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

[64] B. Eriksson, A. Stjerna, R. De Masellis, P. Rüemmer, and A. Sabelfeld, "Black Ostrich: Web Application Scanning with String Solvers," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 549–563.

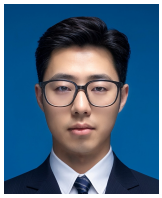
[65] S. Park, D. Kim, S. Jana, and S. Son, "FUGIO: Automatic Exploit Generation for PHP Object Injection Vulnerabilities," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 197–214.

[66] J. Huang, Y. Li, J. Zhang, and R. Dai, "UChecker: Automatically Detecting PHP-based Unrestricted File Upload Vulnerabilities," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 581–592.

[67] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 377–392.

[68] A. Alhuzali, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Chainsaw: Chained Automated Workflow-based Exploit Generation," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 641–652.

[69] P. Li, W. Meng, K. Lu, and C. Luo, "On the Feasibility of Automated Built-in Function Modeling for PHP Symbolic Execution," in *Proceedings of the Web Conference 2021*, 2021, pp. 58–69.

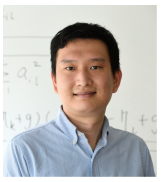


Youkun Shi received the Ph.D. degree in June 2024 from Fudan University, advised by Prof. Yuan Zhang and Prof. Min Yang. He is currently a postdoctoral fellow in the Department of Computing at The Hong Kong Polytechnic University, working under the supervision of Prof. Daniel Xiapu Luo. His research focuses on system security, especially web security.

Fengyu Liu is currently pursuing a Ph.D. degree at the School of Computer Science, Fudan University. His research focus is on Web Security and AI Security.

Guangliang Yang received his PhD degree from Texas A&M University in 2019 and subsequently conducted postdoctoral research at Georgia Tech. He is currently an assistant professor in the School of Computer Science at Fudan University. His research interests primarily focus on computer system security and AI security.

Yuan Zhang received the BEng degree from Nanjing University, Nanjing, China, in 2009, and the PhD degree from Fudan University, Shanghai, China, in 2014, where he is currently a professor with the Software School. His research interests include vulnerability analysis, malware detection, and privacy preservation.



Yinzhi Cao is an Associate Professor in the Department of Computer Science and Technical Director of Information Security Institute at Johns Hopkins University. His research mainly focuses on the security and privacy of the Web, smartphones, and machine learning using program analysis techniques.

Enhao Li is currently pursuing a Master's degree at the School of Computer Science, Fudan University. His research focus is on Web Security.

Xin Tan received the PhD degree in computer science from Fudan University, Shanghai, China, in 2024. He is currently a software analysis researcher at Huawei Technologies Co., Ltd, Shanghai. His research interests include system&software security.

Xiapu Luo is currently a Professor with the Department of Computing, The Hong Kong Polytechnic University. His research focuses on blockchain and smart contracts security, mobile and IoT security, network security and privacy, and software engineering with papers published in top-tier security, software engineering, and networking conferences and journals.

Min Yang received the BSc and PhD degrees in computer science from Fudan University, Shanghai, China, in 2001 and 2006, respectively, where he is currently a professor with the School of Computer Science. His research interests include system security and AI security.

Siyi Chen received the MEng degree from Nanjing University, Nanjing, China, in 2021. She is currently a cyber security algorithm researcher in Alibaba Group. Her research interests include AI-driven software security and AI security.