

Article

VeriFlow: A Framework for the Static Verification of Web Application Access Control via Policy-Graph Consistency

Tao Zhang ^{1,2}, Fuzhong Hao ³, Yunfan Wang ^{1,2,4} , Bo Zhang ^{1,2} and Guangwei Xie ^{5,*}

¹ China Electric Power Research Institute, Nanjing 210003, China; zhangtao@epri.sgcc.com.cn (T.Z.); 230240007@seu.edu.cn (Y.W.); zhangbo6@epri.sgcc.com.cn (B.Z.)

² State Grid Laboratory of Power Cyber-Security Protection and Monitoring Technology, Nanjing 210003, China

³ State Grid Henan Electric Power Company, Zhengzhou 450018, China; cnhfzs@sina.com.cn

⁴ School of Cyber Science and Engineering, Southeast University, Nanjing 211189, China

⁵ Shanghai HONGZHEN Information Science & Technology Co., Ltd., Shanghai 200000, China

* Correspondence: xgw@redmatrix.com.cn

Abstract

The evolution of industrial automation toward Industry 3.0 and 4.0 has driven the emergence of Industrial Edge-Cloud Platforms, which increasingly depend on web interfaces for managing and monitoring critical operational technology. This convergence introduces significant security risks, particularly from Broken Access Control (BAC)—a vulnerability consistently ranked as the top web application risk by the Open Web Application Security Project (OWASP). BAC flaws in industrial contexts can lead not only to data breaches but also to disruptions of physical processes. To address this urgent need for robust web-layer defense, this paper presents VeriFlow, a static verification framework for access control in web applications. VeriFlow reformulates access control verification as a consistency problem between two core artifacts: (1) a Formal Access Control Policy (P), which declaratively defines intended permissions, and (2) a Navigational Graph, which models all user-driven UI state transitions. By annotating the graph with policy P, VeriFlow verifies a novel Path-Permission Safety property, ensuring that no sequence of legitimate UI interactions can lead a user from an authorized state to an unauthorized one. A key technical contribution is a static analysis method capable of extracting navigational graphs directly from the JavaScript bundles of Single-Page Applications (SPAs), circumventing the limitations of traditional dynamic crawlers. In empirical evaluations, VeriFlow outperformed baseline tools in vulnerability detection, demonstrating its potential to deliver strong security guarantees that are provable within its abstracted navigational model. By formally checking policy-graph consistency, it systematically addresses a class of vulnerabilities often missed by dynamic tools, though its effectiveness is subject to the model-reality gap inherent in static analysis.

Keywords: web application security; access control; formal verification; static analysis; Single-Page Applications



Academic Editor: George A. Tsihrintzis

Received: 4 August 2025

Revised: 31 August 2025

Accepted: 4 September 2025

Published: 22 September 2025

Citation: Zhang, T.; Hao, F.; Wang, Y.; Zhang, B.; Xie, G. VeriFlow: A Framework for the Static Verification of Web Application Access Control via Policy-Graph Consistency. *Electronics* **2025**, *14*, 3742. <https://doi.org/10.3390/electronics14183742>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The proliferation of web applications as the backbone of modern digital infrastructure has been accompanied by a persistent and severe class of security vulnerabilities. For years, the Open Web Application Security Project (OWASP) has consistently ranked Broken Access Control (BAC) as the number one security risk [1,2]. Real-world incidents further

illustrate the severity of these flaws [3]. This threat is dramatically amplified in the context of the industrial revolution 3.0, which has driven the development of distributed Industrial Edge-Cloud Platforms. These platforms use web-based dashboards and APIs as the primary human–machine interface for configuring, controlling, and monitoring sensitive operational technology (OT). This architectural shift introduces a critical demand for web defense in industrial control scenarios. Here, a BAC vulnerability is no longer a simple data security issue; it becomes a direct threat to physical safety and operational integrity. In industrial control systems (ICS), the web UI often serves as the primary interface to operational technology (OT). An unauthorized navigation path could allow an attacker to access a control panel for a Programmable Logic Controller (PLC), potentially leading to hazardous physical-state changes [4]. Therefore, ensuring UI-path safety is a critical first line of defense against cyber–physical attacks that aim to disrupt manufacturing lines, compromise utility grids, or cause industrial accidents. For example, a flaw could allow an unauthorized user to alter calibration settings on a factory floor, leading to production defects, or to disable safety monitors in a power plant through a compromised web dashboard. This paper introduces a method to address this specific challenge, aiming to secure the web-based management plane of industrial edge intelligence cloud platforms. This aligns with a growing body of research that applies formal methods to secure IoT systems where web interfaces govern physical processes [5].

Traditional security validation techniques, while valuable, suffer from inherent limitations that undermine their effectiveness against BAC vulnerabilities. Dynamic Application Security Testing (DAST) tools, such as OWASP ZAP or Burp Suite, excel at identifying exploitable flaws through runtime simulations and fuzz testing, but they can only demonstrate the presence of vulnerabilities, not their absence [6,7]. This is a fundamental limitation: the coverage of any test suite is inherently finite, bounded by the test cases executed, and a single overlooked path can compromise the entire system [8,9]. Manual penetration testing, while insightful, is labor-intensive, non-scalable, and prone to human error, often missing subtle logical flaws in large codebases [10]. These methods are particularly inadequate for modern SPAs, where navigation logic is embedded in client-side JavaScript and not readily discoverable by server-side crawlers, leading to incomplete vulnerability detection [11,12]. For example, dynamic tools may fail to explore all possible UI-driven state transitions in an SPA, resulting in false negatives for hidden Forced Browse vulnerabilities [13]. To overcome these shortcomings, formal methods emerge as a promising alternative, offering mathematical rigor to prove security properties definitively [14,15]. Drawing an analogy from civil engineering, where mathematical models prove the structural integrity of bridges under various loads, formal methods can verify that a web application's access control enforces intended policies without exceptions [16,17]. However, prior formal approaches often focus narrowly on policy specifications (e.g., Role-Based Access Control models) without integrating the application's structural realities, such as navigational flows, leading to gaps in practical applicability [18,19]. Studies on formal policy verification highlight these deficiencies, noting that isolated policy checks fail to capture implementation-specific inconsistencies [17,20].

This paper introduces VeriFlow, a comprehensive framework for the static verification of web application access control, reframing BAC as a verifiable consistency problem between security intent and implementation structure. At its core, VeriFlow analyzes the following two key artifacts: (1) a Formal Access Control Policy P , which declaratively specifies the intended permissions—who (roles) can access what (resources) under which conditions; and (2) a Navigational Graph $G_S = (V, E)$, a directed graph where vertices V represent application states (e.g., URLs or UI views) and edges E model possible user-

driven transitions [21,22]. By annotating G_S with permissions from P , VeriFlow checks for inconsistencies that manifest as BAC vulnerabilities.

The cornerstone of this approach is the Path-Permission Safety property, formally defined as follows: for every role r and path π in G_S starting from a node authorized for r , all nodes in π must also be authorized for r [23]. In other words, no sequence of legitimate UI interactions should lead a user from an authorized state to an unauthorized one, directly mitigating threats like Forced Browse and Privilege Escalation. Mathematically, this can be expressed as follows: $\forall r \in R, \forall v_s \in V_r, \forall v_t \in \text{reachable}(v_s), v_t \in V_r$, where R is the set of roles, V_r is the set of nodes authorized for r , and $\text{reachable}(v_s)$ denotes nodes reachable from v_s in G_S [24,25]. A key innovation of VeriFlow is its novel static analysis technique for extracting G_S from SPAs, which parses JavaScript bundles to uncover client-side routing logic—a capability lacking in traditional dynamic crawlers [13]. As illustrated in Figure 1, VeriFlow detects unauthorized paths by verifying this consistency, providing provable guarantees that surpass the probabilistic assurances of dynamic testing [2].

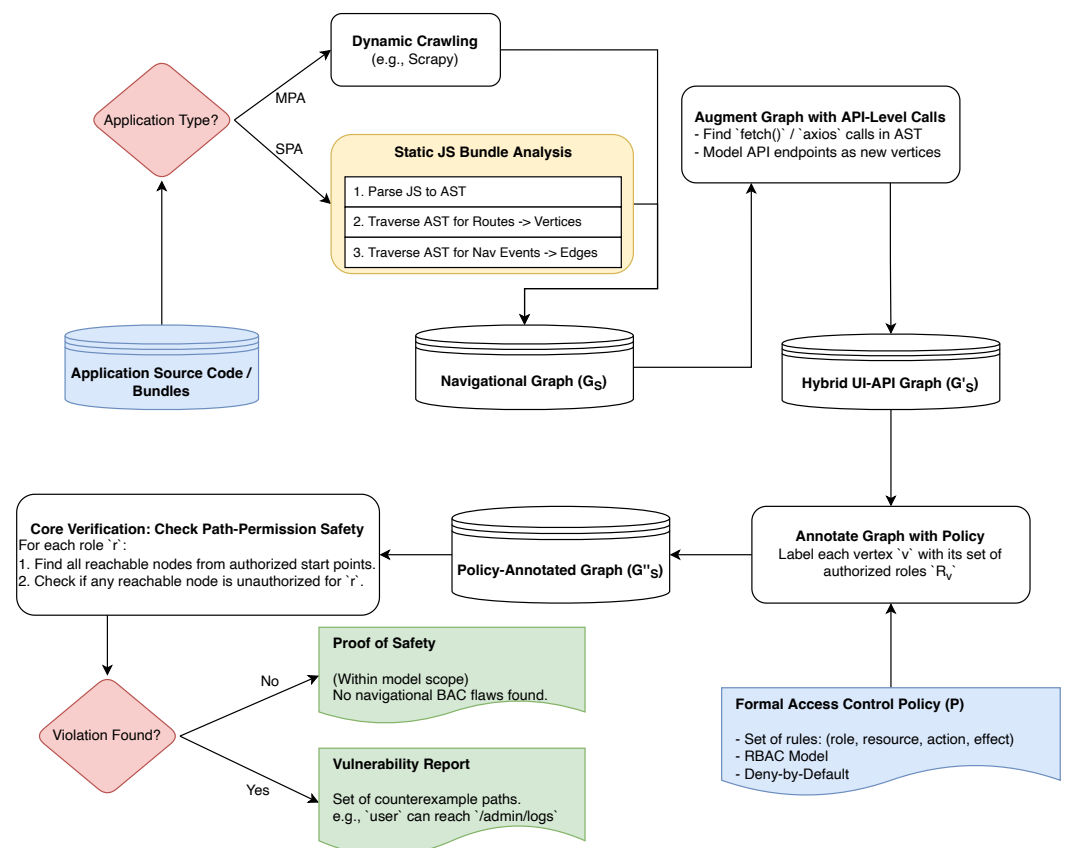


Figure 1. Overview of the VeriFlow framework for detecting Broken Access Control (BAC) vulnerabilities through static and dynamic analysis. The process begins with the application source code and type (SPA/MPA), applying static JS bundle analysis or dynamic crawling to extract a navigational graph G_S . This graph is then augmented with API-level endpoints and annotated using a formal access control policy P , producing the policy-annotated graph G'_S . VeriFlow performs path-permission safety checks to detect unauthorized navigational paths. If violations are found, it outputs counterexample paths indicating access control flaws (e.g., unauthorized access to `/admin/logs`); otherwise, it proves safety within the model's scope.

VeriFlow's static approach not only identifies vulnerabilities but also offers formal proofs of their absence within the modeled scope, contrasting sharply with the exploratory nature of dynamic methods. By integrating policy verification with structural analysis, it

addresses the model-reality gap that plagues many formal tools, ensuring relevance to real-world deployments [26].

The contributions of this paper are as follows: (1) formal models for policy-graph consistency, including the definition of P , G_S , and the Path-Permission Safety property; (2) an efficient verification algorithm that scales to large-scale web applications; (3) an innovative static analysis technique for taming SPAs and extracting navigational graphs; (4) extensions to the framework for detecting API-level vulnerabilities, such as Broken Object Level Authorization (BOLA) [27]; and (5) a detailed empirical evaluation plan using real-world open-source applications to validate VeriFlow’s effectiveness against baselines like OWASP ZAP. The remainder of the paper is structured as follows: Section 2 surveys related work in access control verification and formal methods; Section 3 presents the VeriFlow formalism in detail; Section 4 describes the algorithm and framework implementation; Section 5 outlines the empirical evaluation methodology and expected results; Section 6 discusses limitations, future work, and broader implications; and Section 7 concludes with a summary of VeriFlow’s potential impact on advancing provable security in web applications. Through these contributions, VeriFlow paves the way for a new era of formally verified access control, enhancing the security posture of modern web ecosystems.

2. Related Work

Our work builds upon established principles from the fields of formal methods, static program analysis, and web application security, while tailoring them to the unique challenges of modern web applications.

2.1. Model Checking and Policy Verification

Model checking is an automated formal verification technique that exhaustively explores all possible states of a system model to verify if it satisfies a given specification, often expressed in temporal logic [16]. This technique has been successfully applied to verify the logical consistency of access control policies themselves. For instance, researchers have used model checking to analyze policies for cloud infrastructure like Google Cloud IAM or XACML to discover potential misconfigurations or conflicts [17,26]. However, the primary challenge for these methods is the state-space explosion problem, where the number of possible states grows exponentially with system complexity, making verification infeasible for large systems [16]. Our approach, while more specialized, is inspired by the rigor of foundational formal method tools like Alloy [28] and TLA+ [29], aiming to apply similar principles to the specific domain of web application access control. Recent work has also demonstrated the viability of applying model checking to other modern, security-critical systems like blockchain [30], affirming the scalability of formal property checking [31].

VeriFlow’s approach differs from traditional model checking. It cleverly avoids the state-space explosion problem by not attempting to explore the full state space of the application, which includes business logic and data. Instead, VeriFlow narrows its focus to a more tractable problem: navigational reachability within the UI. Furthermore, while most existing work verifies policies in isolation, VeriFlow’s core contribution is to verify the consistency between the policy and the application’s implementation structure. This addresses the critical gap between security intent and its actual implementation in code.

2.2. Static Analysis for Access Control

Static analysis techniques discover potential defects by analyzing a program’s source or binary code without executing it. While powerful methods like symbolic execution can explore multiple program paths by using symbolic values instead of concrete data, they are often computationally expensive and complex to implement [32]. In recent years, deep

learning-based static analysis has emerged as a promising approach, with models trained to detect common vulnerability patterns (e.g., SQL injection or buffer overflows) from source code representations. However, these methods often function as “black boxes,” lacking formal guarantees and struggling with the logical complexity of access control policies. Other modern techniques, such as those based on code property graphs, offer more structured representations but may not be optimized for the specific challenges of SPAs. Modern static application security testing (SAST) tools like CodeQL and Semgrep use powerful query languages to find vulnerability patterns directly in the source code. While effective for detecting common flaws like injection, they typically rely on pre-defined rules and may struggle to verify custom, application-specific access control logic without significant customization. VeriFlow differs by not just finding patterns, but by formally verifying the consistency between an explicit policy model and the application’s navigational structure. Within the access control domain, early work has applied static analysis to detect vulnerabilities in traditional web applications, for example, by analyzing an application’s code to construct role-based sitemaps and check for forced browse vulnerabilities. VeriFlow differentiates itself by providing a formal, white-box approach that is specifically tailored to the navigational structure of SPAs, offering a level of precision that is often missing in more general-purpose or purely data-driven static analysis tools [33].

2.3. Analysis of Modern Web Applications (SPAs and APIs)

Modern web applications, especially Single-Page Applications (SPAs) built with frameworks like React, Vue, or Angular, introduce new challenges for security analysis. In SPAs, navigation and state transitions are handled dynamically by client-side JavaScript, rendering traditional web crawlers and dynamic analysis tools ineffective at capturing the complete application structure [34]. Concurrently, with the rise of frontend–backend decoupled architectures, APIs that are directly exposed to the client have become a new hotspot for vulnerabilities like Broken Object Level Authorization (BOLA) [27].

VeriFlow’s framework is designed to directly address these modern challenges. One of its key innovations is the ability to statically parse the compiled JavaScript bundles of an SPA, identify its internal routing declarations and navigation events, and construct a Potential Navigational Graph that is more comprehensive than any dynamic exploration could achieve [13,35]. Furthermore, VeriFlow extends this analysis to the API level. By identifying API calls like ‘fetch()’ or ‘axios.get()’ in the client-side code and modeling them as special vertices in the graph, VeriFlow can verify reachability from a UI state to a privileged API endpoint within a single, unified framework, enabling the effective static detection of potential BOLA vulnerabilities.

3. The VeriFlow Formalism

The persistent threat of Broken Access Control (BAC), consistently identified as the top security risk by the Open Web Application Security Project (OWASP) [2], underscores the urgent need for rigorous verification methods to safeguard web applications. BAC vulnerabilities, such as Forced Browse and Privilege Escalation, stem from logical inconsistencies between an application’s intended security policy and its implemented structure, often leading to unauthorized access to sensitive resources [36]. Traditional dynamic testing approaches, including Dynamic Application Security Testing (DAST) and manual penetration testing, are fundamentally limited by their inability to exhaustively cover all execution paths, thus failing to prove the absence of vulnerabilities [15]. To address this, the VeriFlow framework introduces a formal, static verification approach that reframes access control verification as a consistency check between two critical artifacts: a declarative access control policy (P), which encapsulates the security intent, and a navigational

graph (G_S), which models all possible user-driven state transitions within the application's user interface. This section formally defines these components and introduces the Path-Permission Safety property, a novel security invariant that ensures no user can navigate from an authorized state to an unauthorized one, directly mitigating BAC risks. By leveraging mathematical rigor and formal methods, VeriFlow provides a systematic, automated solution to detect logical access control flaws that evade conventional testing, offering a pathway to provably secure web applications [15,23].

3.1. Modeling the Access Control Policy (P)

The cornerstone of VeriFlow is a formal access control policy, denoted P , which serves as a declarative, machine-readable specification of the intended security constraints governing access to web application resources. Drawing inspiration from modern policy languages like Open Policy Agent (OPA) Rego [37,38], P is designed to balance expressiveness—capturing the complex, role-based access control (RBAC) requirements prevalent in web applications [39]—with simplicity, ensuring amenability to automated formal analysis. This approach, centered on the widely adopted RBAC model [40], contrasts sharply with code-level access checks, which are often fragmented, error-prone, and challenging to verify [36,41]. By providing a centralized, declarative model, P facilitates systematic reasoning and adheres to the “Deny-by-Default” principle, a best practice for mitigating unauthorized access risks [2]. This design also distinguishes P from verbose standards like XACML [42,43], which, while expressive, are overly complex for efficient automated verification in dynamic web environments [44–47].

To enable precise reasoning, we formally define the components of P as follows. First, a subject represents a user's identity within the RBAC framework, where roles abstract groups of users with similar permissions—a model widely adopted for its scalability and manageability [39,48].

Definition 1 (Subject). *A subject is a role $r \in R$, where R is a finite set of roles representing user categories in the system (e.g., {admin, user, guest}).*

Next, resources are defined as URL patterns to align with the web application's navigational structure, allowing flexible specification of access controls across groups of endpoints common in modern web frameworks [40].

Definition 2 (Resource). *A resource is a URL path pattern $p \in \text{Patterns}$, which may include wildcards (e.g., /admin/*) or parameterized segments (e.g., /users/{id}/profile).*

For the scope of VeriFlow, actions are focused on UI-driven interactions, which are central to BAC vulnerabilities like Forced Browse where attackers exploit navigation paths to access unauthorized pages [2].

Definition 3 (Action). *The action $a \in \text{Actions}$ is fixed to “view” for the scope of UI navigation, representing user-driven state transitions in the web application.*

The policy decision is binary, simplifying evaluation while enforcing robust security by default [36].

Definition 4 (Effect). *The effect $e \in \text{Effects}$ is a decision from the set {allow, deny}, determining whether access is granted or prohibited.*

Together, these components form the policy as a set of rules.

Definition 5 (Policy). The policy $P \subseteq R \times \text{Patterns} \times \text{Actions} \times \text{Effects}$ is a set of rules, where each rule (r, p, a, e) specifies the access decision for a role r on a resource pattern p for action a .

The policy evaluation function $P(r, u)$, where $r \in R$ and u is a concrete URL, returns a decision based on matching rules. Matching precedence is handled deterministically:

- An explicit ‘deny’ rule for a specific role and resource always takes precedence over any ‘allow’ rule.
- If multiple ‘allow’ rules match, the most specific pattern (longest literal prefix) wins.
- If no rule matches, the decision defaults to ‘deny’.

This ensures a deny-by-default posture and handles overlapping pattern definitions unambiguously.

The policy evaluation function $P(r, u)$, where $r \in R$ and u is a concrete URL, returns a decision based on matching rules. Matching precedence is handled deterministically: 1. An explicit ‘deny’ rule for a specific role and resource always takes precedence over any ‘allow’ rule. 2. If multiple ‘allow’ rules match, the most specific pattern (longest literal prefix) wins. 3. If no rule matches, the decision defaults to ‘deny’. This ensures a deny-by-default posture and handles overlapping pattern definitions unambiguously.

3.2. Modeling the Application Structure: The Navigational Graph (G_S)

The navigational graph $G_S = (V, E)$ is a formal abstraction of a web application’s user interface, capturing all possible UI-driven state transitions to enable the detection of BAC vulnerabilities. This graph-based model reflects the application’s structural reality, representing the paths a user can navigate through UI interactions, such as clicking hyperlinks or submitting forms—critical attack vectors for Forced Browse and Privilege Escalation [2]. Unlike security models focused on server-side logic, G_S emphasizes client-side navigation, aligning with real-world scenarios where attackers exploit UI elements to bypass access controls [36]. Constructing G_S is particularly challenging for Single-Page Applications (SPAs), where navigation logic embedded in client-side JavaScript renders traditional web crawlers ineffective [13]. While automated crawling can be used for Multi-Page Applications (MPAs) [49], SPAs necessitate a novel static analysis approach (detailed in Section 4.2). By modeling the application as a graph, VeriFlow bridges the gap between intended navigation flows and the stateless nature of HTTP, enabling systematic analysis of access control correctness [21,22].

Formally, the navigational graph is defined as follows. Vertices represent the distinct states a user can reach, providing a comprehensive model of the application’s navigational structure that maps directly to the resources protected by policy P [50,51].

Definition 6 (Vertices). The set V represents the unique, reachable UI states in the application, where each vertex $v \in V$ corresponds to a unique URL or view (e.g., `/home`, `/profile`).

Edges model the dynamic interactions that define the application’s navigational flow, drawing on graph theory applications in security [25] to allow for systematic path exploration.

Definition 7 (Edges). The set of directed edges $E \subseteq V \times V$ represents direct navigations between states, where an edge $(u, v) \in E$ exists if a user can transition from state u to state v through a single UI interaction (e.g., clicking a hyperlink or submitting a form).

The significance of G_S lies in its ability to expose structural inconsistencies, such as a navigation path allowing a low-privileged user to reach an admin-only page [2]. While G_S focuses on client-side navigation, it may not capture server-side redirects or complex business logic decoupled from URLs, a limitation that motivates future enhancements, such

as integrating server-side analysis [15]. By prioritizing UI-driven transitions, VeriFlow targets a critical attack surface overlooked by traditional tools.

3.3. Defining the Core Verification Property: Path-Permission Safety

The Path-Permission Safety property is the linchpin of VeriFlow's verification approach, linking the access control policy P and the navigational graph G_S to detect logical access control flaws. To integrate policy information into the graph, we first define an annotation function that labels each vertex with its authorized roles, thereby creating a unified model that combines security intent with application structure [23].

This property ensures that if a user with role r is authorized to access state u , they must also be authorized to access any state v reachable from u through UI navigations. A violation indicates a critical flaw, such as a navigation path that allows a user to move from a public page (e.g., /profile) to a restricted resource (e.g., /admin/logs), potentially enabling privilege escalation. A key innovation of VeriFlow is its novel static analysis technique for extracting G_S from SPAs. Mathematically, this can be expressed as follows: $\forall r \in R, \forall v_s \in V_r, \forall v_t \in \text{reachable}(v_s), v_t \in V_r$, where R is the set of roles, V_r is the set of nodes authorized for r , and $\text{reachable}(v_s)$ denotes nodes reachable from v_s in G_S [24,25].

Definition 8 (Annotation Function). The function $\text{Annotate}(G_S, P)$ produces a policy-annotated graph G'_S , where each vertex $v \in V$ is labeled with a set of authorized roles $R_v = \{r \in R \mid P(r, \text{url}(v)) = \text{allow}\}$, based on the URL corresponding to v .

Using this annotated graph, we define the core security property. Rooted in classic safety properties from formal methods [23], Path-Permission Safety directly addresses BAC vulnerabilities by guaranteeing that no legitimate UI interaction sequence can lead to an unauthorized state [2].

Definition 9 (Path-Permission Safety). A policy-annotated graph G'_S is Path-Permission Safe if and only if, for every role $r \in R$ and vertices $u, v \in V$, the existence of a path from u to v in G_S implies that $r \in R_u \implies r \in R_v$.

To provide a more rigorous theoretical foundation, we formalize this property using mathematical logic. Let $\text{Path}(u, v, G_S)$ denote the existence of a directed path from vertex u to vertex v in graph G_S . The Path-Permission Safety property can be expressed as follows:

$$\forall r \in R, \forall u, v \in V : \text{Path}(u, v, G_S) \wedge (r \in R_u) \rightarrow (r \in R_v)$$

This logical formulation ensures that the implication holds universally across all roles and vertex pairs. We further define the concept of reachability closure for a given role as follows:

Definition 10 (Role-Specific Reachability Closure). For a role $r \in R$, the reachability closure $\text{Reach}_r(G'_S)$ is the set of all vertices reachable from any vertex authorized for role r as follows:

$$\text{Reach}_r(G'_S) = \{v \in V \mid \exists u \in V : r \in R_u \wedge \text{Path}(u, v, G_S)\}$$

Theorem 1 (Safety Violation Characterization). A policy-annotated graph G'_S violates Path-Permission Safety if and only if there exists a role $r \in R$ and a vertex $v \in \text{Reach}_r(G'_S)$ such that $r \notin R_v$.

Proof Sketch. The forward direction follows directly from the definition: if such a vertex v exists, then there is a path from some authorized vertex u (where $r \in R_u$) to v (where

$r \notin R_v$), violating the safety property. The reverse direction follows by contrapositive: if no such vertex exists, then for all reachable vertices from authorized states, the role remains authorized, satisfying the safety property. $\square$

This property ensures that if a user with role r is authorized to access state u , they must also be authorized to access any state v reachable from u through UI navigations. A violation indicates a critical flaw, such as a navigation path that allows a user to move from a public page (e.g., `/profile`) to a restricted resource (e.g., `/admin/logs`), potentially enabling privilege escalation. Unlike dynamic testing, which struggles with incomplete coverage [15], this property enables exhaustive verification of all navigation paths, providing a provable guarantee of access control correctness. The property is verified using an efficient graph traversal algorithm (detailed in Section 4.1), with a time complexity of $O(|R| \cdot (|V| + |E|))$, making it scalable for real-world applications. VeriFlow's focus on UI-driven navigation distinguishes it from prior work, such as model checking of cloud IAM policies [16], and its extensibility to API-level vulnerabilities (Section 4.3) enhances its applicability to modern web architectures [27]. By combining formal rigor with practical relevance, Path-Permission Safety establishes VeriFlow as a powerful framework for mitigating one of the most severe security risks in web applications.

4. The VeriFlow Algorithm and Framework

Building on the formal models established in Section 3, this section presents the algorithmic and technical implementation of the VeriFlow framework, enabling automated, static verification of web application access control. By translating the abstract concepts of the Formal Access Control Policy (P), the Navigational Graph (G_S), and the Path-Permission Safety property into a practical, executable system, VeriFlow bridges the gap between theoretical rigor and real-world applicability, addressing the persistent challenges of Broken Access Control (BAC) vulnerabilities such as Forced Browsing and Privilege Escalation. The implementation strategy emphasizes automation to overcome the inherent limitations of dynamic testing, including incomplete coverage and inability to prove the absence of flaws, while ensuring scalability for large-scale web applications with intricate client-side logic. This section is structured into three interconnected subsections: first, we detail the core verification algorithm (*Verify-Path-Safety*), which efficiently checks for safety property violations by traversing the annotated graph and identifying unauthorized paths; second, we introduce an innovative static analysis technique tailored for Single-Page Applications (SPAs), which extracts the navigational graph from JavaScript bundles to handle dynamic routing challenges; and third, we describe extensions to detect API-level and data-flow vulnerabilities, such as Broken Object Level Authorization (BOLA), by augmenting the graph model with client-side API interactions. Together, these components form a scalable framework that not only detects logical access control inconsistencies but also provides actionable counterexamples, facilitating integration into continuous development pipelines and offering a pathway toward provable security in modern web ecosystems.

4.1. The Core Verification Algorithm

The core of the VeriFlow framework is the *Verify-Path-Safety* algorithm, designed to enforce the Path-Permission Safety property defined in Section 3. This algorithm systematically checks for violations of Path-Permission Safety by identifying paths in the policy-annotated navigational graph G'_S that lead from authorized states to unauthorized states for a given user role. Such violations directly correspond to critical Broken Access Control (BAC) vulnerabilities, such as Forced Browse or Privilege Escalation, which are prevalent in modern web applications. By automating the verification process, the algorithm eliminates the coverage limitations of dynamic testing methods, providing a scalable and rigorous approach suit-

able for real-world web applications. The algorithm outputs a set of counterexample paths, each representing a potential vulnerability, thereby offering actionable feedback for developers to remediate logical flaws in access control implementations.

The Verify-Path-Safety algorithm, presented in Algorithm 1, operates on the navigational graph $G_S = (V, E)$ and the formal policy P , as defined in Section 3. The algorithm begins by annotating G_S with role permissions using the Annotate function, which assigns to each vertex $v \in V$ a set of authorized roles $R_v = \{r \in R \mid P(r, \text{url}(v)) = \text{"allow"}\}$. For each role r , it identifies the set of starting vertices V_{start} where r is authorized. A graph traversal (e.g., Breadth-First Search or Depth-First Search) is then performed to compute the set of reachable vertices $V_{\text{reachable}}$ from V_{start} , leveraging efficient graph algorithms. If a reachable vertex w is not authorized for role r (i.e., $r \notin R_w$), the algorithm reconstructs the path from a starting vertex to w and adds it to the set of violations. This path represents a potential vulnerability, such as a user navigating from an authorized page (e.g., /profile) to an unauthorized admin page (e.g., /admin/logs), as illustrated in Figure 2. The output set of violation paths provides precise, actionable feedback for developers to address access control flaws.

Algorithm 1 Verify-Path-Safety(G_S, P)

Require: Navigational Graph $G_S = (V, E)$, Formal Policy P

Ensure: A set of counterexample paths, Violations

```

1: Violations  $\leftarrow \emptyset$ 
2:  $G'_S \leftarrow \text{Annotate}(G_S, P)$  ▷ Annotate graph with role permissions per vertex
3: Roles  $\leftarrow \text{GetUniqueRoles}(P)$  ▷ Extract all roles from  $P$ 
4: for each role  $r \in \text{Roles}$  do
5:    $V_{\text{start}} \leftarrow \{v \in V \mid r \in R_v\}$  ▷ Identify nodes authorized for role  $r$ 
6:    $V_{\text{reachable}} \leftarrow \text{GraphTraversal}(G'_S, V_{\text{start}})$  ▷ BFS/DFS from start nodes
7:   for each vertex  $w \in V_{\text{reachable}}$  do
8:     if  $r \notin R_w$  then
9:       path  $\leftarrow \text{ReconstructPath}(w, V_{\text{start}})$ 
10:      Violations.add(path)
11:    end if
12:  end for
13: end for
14: return Violations
  
```

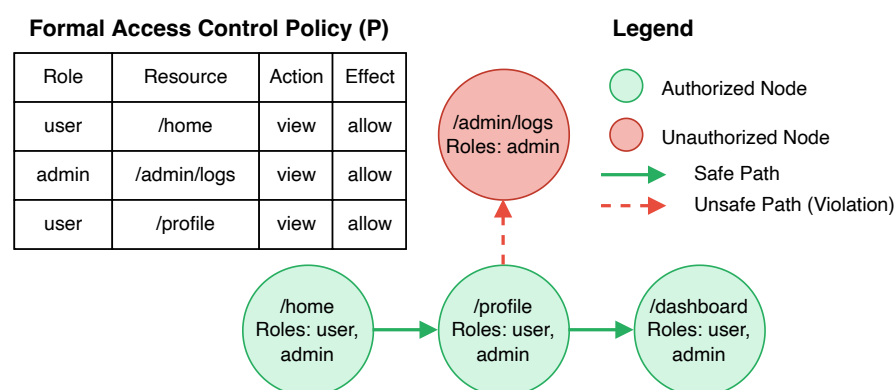


Figure 2. Example of policy-annotated navigational graph and detection of a Broken Access Control (BAC) violation. Each node represents a web page labeled with its authorized roles according to the formal access control policy P (shown in the top-left table). Green arrows represent safe paths where all visited pages are accessible by the role (e.g., /home $\rightarrow$ /profile $\rightarrow$ /dashboard). A red dashed arrow highlights an unsafe path where a user accesses /admin/logs, a page restricted to admin, thus violating the path-permission safety property.

The efficiency of *Verify-Path-Safety* is a key factor in its practical applicability. The algorithm's time complexity is $O(|R| \cdot (|V| + |E|))$, where $|R|$ is the number of roles, $|V|$ is the number of vertices, and $|E|$ is the number of edges in G_S . This linear complexity per role ensures scalability for typical web applications, which often have a moderate number of roles and navigational states. The use of standard graph traversal algorithms minimizes computational overhead, avoiding the state-space explosion problem common in exhaustive model checking approaches. Practical considerations, such as handling large graphs, are addressed through optimized graph representations (e.g., using adjacency lists in *NetworkX*) and precise policy annotation to reduce false positives [52,53]. The algorithm integrates seamlessly with the broader *VeriFlow* framework, enabling subsequent static analysis of Single-Page Applications (SPAs), as detailed in Section 4.2. By focusing on navigational reachability, *Verify-Path-Safety* provides a robust and efficient mechanism for detecting logical access control vulnerabilities, aligning with the needs of modern web application security practices.

4.2. Taming the Single-Page Application (SPA)

Modern Single-Page Applications (SPAs), constructed using frameworks such as *React*, *Vue.js*, or *Angular*, present formidable challenges for access control verification due to their reliance on client-side routing and dynamic content rendering. Unlike traditional Multi-Page Applications (MPAs), where navigation is driven by server-rendered HTML and can be explored via automated web crawlers, SPAs manage state transitions entirely within the browser's JavaScript environment, rendering dynamic analysis tools ineffective as they fail to simulate all possible user interactions and code paths. *VeriFlow* introduces a groundbreaking innovation: a fully static analysis methodology that directly extracts the navigational graph G_S from the application's compiled JavaScript bundles, eliminating the need for runtime execution or browser simulation. This approach ensures a comprehensive representation of all potential navigation flows, enabling the detection of subtle access control vulnerabilities that might otherwise remain hidden in the codebase. By leveraging the declarative nature of SPA routing configurations, *VeriFlow* bridges a critical gap in existing web application security tools, as highlighted in prior studies on SPA security risks.

The static analysis process for graph extraction in *VeriFlow* is structured into three methodical steps, designed to systematically deconstruct the SPA's client-side logic. First, the compiled JavaScript bundles are parsed into an Abstract Syntax Tree (AST) using robust parsing libraries such as *Esprima*, which transforms the minified or transpiled code into a structured, traversable representation amenable to programmatic analysis. Second, the AST is traversed to identify route declarations specific to the underlying framework; for instance, in *React*, this involves locating `<Route path="...">` components or equivalent router configurations, while in *Angular*, it entails extracting definitions from `RouterModule.forRoot([...])`. Each identified route path is mapped to a vertex in G_S , representing a unique UI state or view. Third, navigation events are detected by scanning for programmatic triggers such as `<Link to="...">` elements in *React*, `navigate(...)` calls in *Vue*, or `this.router.navigate(...)` invocations in *Angular*, which establish directed edges between vertices based on source-destination relationships. This culminates in the construction of a "Potential Navigational Graph" that over-approximates all feasible navigation paths encoded in the application, ensuring a conservative yet complete model for verification. The framework-agnostic design of this process allows *VeriFlow* to adapt seamlessly to various SPA ecosystems, drawing on established static analysis techniques for web applications.

The advantages of *VeriFlow*'s static analysis over traditional dynamic crawling are manifold, primarily in its ability to capture navigation paths that may not be exercised

during finite runtime explorations, thereby enhancing coverage and reducing the risk of overlooked vulnerabilities. Unlike crawlers that depend on observable HTML outputs and may miss conditional or asynchronous navigations, static extraction analyzes the code's latent structure directly, providing a superset of possible transitions that aligns with the conservative ethos of formal verification [54]. However, this method is not without limitations; the over-approximation inherent in the Potential Navigational Graph can include infeasible paths due to unaccounted runtime conditions (e.g., authentication guards or data dependencies), potentially leading to false positives that necessitate manual validation. Additionally, challenges arise with heavily obfuscated or dynamically loaded JavaScript code, which may complicate AST parsing and require preprocessing steps. Our prototype addresses obfuscation by applying de-obfuscation tools like de4js as a preprocessing step to improve the readability of the AST. For dynamic imports (e.g., `'import()'`), we conservatively add edges from the import site to all potential routes defined in the dynamically loaded modules, accepting a potential increase in over-approximation to ensure no paths are missed. Non-standard routing hierarchies are handled by allowing customizable, framework-specific parsers that can be configured to recognize custom routing components or state management patterns. To mitigate these, VeriFlow ensures conservativeness by prioritizing no false negatives, meaning all reported violations in our evaluation were manually validated to confirm their reachability and classify them as true or false positives.

Practically, this static analysis integrates seamlessly with the Verify-Path-Safety algorithm described in Section 4.1, enabling end-to-end verification where the extracted G_S is annotated with policy P and traversed to uncover access control inconsistencies. This synergy facilitates automated workflows in development pipelines, empowering developers to identify and remediate vulnerabilities early in the lifecycle. Future refinements could incorporate hybrid approaches, blending static extraction with targeted dynamic analysis to resolve over-approximations, or extend support for server-side rendering (SSR) in SPAs like Next.js, where client-server interactions add layers of complexity. This technique paves the way for the empirical evaluation outlined in Section 5, where it will be applied to real-world open-source SPAs such as those from Casdoor and OpenRMF, demonstrating its efficacy in diverse architectural contexts.

4.3. Detecting API-Level and Data-Flow Vulnerabilities

The proliferation of Application Programming Interfaces (APIs) in modern web applications, especially within Single-Page Applications (SPAs), has amplified the risks associated with API-level vulnerabilities, making them a focal point of contemporary security concerns. SPAs often rely on client-side JavaScript to invoke backend API endpoints dynamically, facilitating seamless user experiences but introducing potential attack vectors where access controls may be inadequately enforced. To address this, VeriFlow extends its navigational graph model beyond UI-driven transitions to encompass API interactions, thereby broadening the framework's scope to detect sophisticated vulnerabilities such as Broken Object Level Authorization (BOLA). BOLA, as highlighted in the OWASP API Security Top 10, occurs when an attacker exploits insufficient authorization checks to access or manipulate objects (e.g., data records) belonging to other users or requiring elevated privileges. By incorporating API endpoints into the graph, VeriFlow captures the interplay between UI states and backend calls, enabling the identification of scenarios where low-privileged users can reach restricted APIs through client-side code, such as `fetch()` or `axios.get()` invocations. This extension aligns with the framework's core philosophy of policy-graph consistency, ensuring that access control verification is not limited to navigational flows but also includes data-flow dependencies that could lead to unauthorized actions.

Building upon the static analysis technique detailed in Section 4.2, VeriFlow augments the JavaScript Abstract Syntax Tree (AST) traversal to identify and extract API-related constructs embedded within the client-side codebase. Specifically, the analysis scans for patterns indicative of API calls, such as `fetch('/api/documents/delete')` or `axios.get('/api/users/{id}')`, which are often scattered across components or services in SPA frameworks like React or Angular. These extracted API endpoints are integrated into the navigational graph $G_S = (V, E)$ as specialized vertices, distinguished by their type (e.g., denoted as squares in visualizations to differentiate from UI vertices). Directed edges are then established from UI vertices (representing accessible states) to these API vertices, modeling the data-flow transitions triggered by user interactions or programmatic events. Concurrently, the formal access control policy P is extended to accommodate API-specific rules, incorporating tuples that define roles, resource patterns (e.g., `/api/*`), actions (e.g., GET, POST), and effects. The Verify-Path-Safety algorithm (Algorithm 1) is subsequently applied to this hybrid graph, detecting BOLA violations by identifying paths where a role authorized for a starting UI vertex can reach an unauthorized API vertex. For instance, if a “user” role can navigate to a dashboard view that invokes a privileged delete API, the algorithm reconstructs and flags this path as a violation. This process is visually exemplified in Figure 3, which depicts a hybrid UI-API graph with a highlighted vulnerable connection. The framework distinguishes between different types of API vulnerabilities based on the policy rules. An endpoint protected by a rule like `(admin, /api/admin/*, *, allow)` is an admin-only endpoint. In contrast, potential Insecure Direct Object Reference (IDOR) vulnerabilities are modeled by identifying parameterized endpoints like `/api/users/id` where the policy does not enforce ownership. The current framework flags any non-admin access to such endpoints as a potential BOLA/IDOR flaw requiring further review.

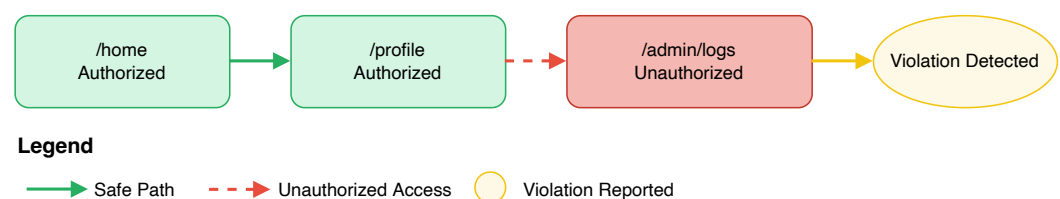


Figure 3. Violation detection example illustrating path-permission safety checking. A user starts from the authorized page `/home` and follows a safe path to `/profile`. A red dashed arrow represents an unauthorized access to `/admin/logs`, which is not permitted for the current role. This violation is detected and flagged as a security flaw. The legend denotes safe paths (green), unauthorized transitions (red dashed), and violation reports (yellow).

The significance of this extension lies in its ability to provide static, provable guarantees against BOLA and related data-flow vulnerabilities, complementing VeriFlow’s UI-centric verification and addressing a critical gap in existing tools that often overlook client-side API exposures. By preemptively uncovering these issues without requiring runtime execution, the framework empowers developers to mitigate risks early in the development lifecycle, potentially averting data breaches or unauthorized modifications that could result in severe financial and reputational damage. However, limitations persist, including the dependency on API calls being explicitly present and analyzable in client-side code, which may not capture server-side enforcement mechanisms or obfuscated implementations. Additionally, server-side access control logic decoupled from the client remains outside the current scope, potentially leading to incomplete coverage. To overcome these, future work could integrate server-side static analysis (e.g., examining back-end code for policy enforcement) or hybrid dynamic testing approaches, drawing from automated policy inference techniques. Such enhancements would further solidify VeriFlow as a comprehensive tool for end-to-end access control verification in complex web

ecosystems. A potential challenge in augmenting the graph with API vertices is the risk of state-space explosion, especially in applications with numerous API calls. To mitigate this, our framework employs route grouping, where API endpoints following a similar pattern (e.g., `/api/users/id/(profile|settings|logs)`) are collapsed into a single representative vertex. This abstraction maintains the integrity of access control verification while keeping the graph size manageable.

5. Empirical Evaluation

5.1. Prototype Implementation

The VeriFlow prototype is implemented in Python (version 3.8 or higher is used for compatibility with the required libraries.), selected for its extensive ecosystem of libraries that facilitate security analysis, graph manipulation, and web scraping tasks. This choice ensures a robust, flexible foundation capable of handling the diverse requirements of static verification in web applications. The prototype comprises three primary components: (1) graph extraction for Multi-Page Applications (MPAs) utilizing Scrapy, a high-level web crawling framework that systematically navigates HTML-based structures to identify navigational links and states; (2) static analysis for Single-Page Applications (SPAs) employing *esprima-python*, which parses JavaScript bundles into Abstract Syntax Trees (ASTs) for extracting routing configurations and navigation events; and (3) graph representation and verification leveraging NetworkX, a powerful library for creating, manipulating, and analyzing complex networks. These components seamlessly integrate the Verify-Path-Safety algorithm detailed in Section 4.1, which performs the core verification of Path-Permission Safety, with the innovative SPA static analysis technique outlined in Section 4.2, enabling end-to-end automation from graph extraction to vulnerability detection. The reliance on open-source tools not only promotes reproducibility and accessibility in security research but also aligns with community-driven standards for transparent and verifiable implementations, allowing researchers and practitioners to extend or replicate the prototype with minimal barriers.

Delving into the technical details, graph extraction begins with MPAs, where Scrapy is configured to perform depth-limited crawling of HTML pages, identifying hyperlinks, forms, and redirects to construct the vertex set V and edge set E of the navigational graph G_S . For SPAs, *esprima-python* is employed to generate an AST from minified or bundled JavaScript files, followed by targeted traversal algorithms that detect framework-specific constructs—such as React’s `<Route>` components or Angular’s `RouterModule` declarations—to map routes to vertices and programmatic navigations (e.g., `history.push()` or `<Link to="...">`) to directed edges. NetworkX is then utilized to construct the graph G_S , annotate vertices with role-based permissions using the policy P as defined in Section 3.1, and execute efficient graph traversal methods (e.g., Breadth-First Search or Depth-First Search) to identify reachable states and detect violations of Path-Permission Safety, such as paths from authorized to unauthorized vertices. Practical considerations include optimizations for large-scale applications, such as parallel processing of AST traversals to handle complex JavaScript codebases exceeding thousands of lines, memory-efficient graph storage to manage graphs with up to 10^4 vertices and edges, and error-handling mechanisms for obfuscated code or incomplete bundles. Furthermore, the prototype’s extensibility is demonstrated through modular support for API-level analysis (Section 4.3), where AST-based extraction identifies client-side API calls (e.g., `fetch('/api/...')`) and integrates them as specialized vertices in G_S , thereby enabling detection of Broken Object Level Authorization (BOLA) vulnerabilities without requiring runtime execution. This alignment with the formalism in Section 3 and algorithms

in Section 4 ensures that the prototype not only operationalizes theoretical constructs but also addresses real-world challenges in web application security verification.

5.2. Experimental Design

The empirical evaluation of VeriFlow is conducted on a carefully selected set of 3–5 open-source web applications that exhibit diverse architectures and access control mechanisms, ensuring a comprehensive assessment of the framework’s capabilities. Specifically, we choose applications such as Casdoor (an SPA-based identity and access management platform), OpenRMF (an MPA-focused risk management framework) [55], and iTrust (an SPA-oriented healthcare application with complex RBAC models). These selections are justified by their public availability, which promotes transparency and reproducibility in security research. Moreover, the diversity in application types—spanning both Multi-Page Applications (MPAs) and Single-Page Applications (SPAs)—tests VeriFlow’s versatility in handling different navigational paradigms.

To establish a reliable baseline for verification, we first create a ground truth access control policy P for each application through a systematic manual analysis of its official documentation, source code annotations, and configuration files. This process involves identifying all roles (e.g., admin, user), resource patterns (e.g., URL paths like /admin/*), and associated permissions, ensuring that P accurately reflects the intended security model while adhering to the Deny-by-Default principle outlined in Section 3.1. Following policy creation, VeriFlow automates the extraction of the navigational graph G_S using tailored techniques: for MPAs, we employ Scrapy to perform a comprehensive crawl of HTML-based links and forms, generating vertices and edges from static page structures; for SPAs, we utilize esprima-python to parse JavaScript bundles into an Abstract Syntax Tree (AST), identifying route declarations and navigation events as detailed in Section 4.2. This automation ensures scalability, as graph extraction operates without requiring application execution, and consistency across multiple runs. The policy P is then annotated onto G_S using the Annotate function (Section 3.3), embedding role-specific permissions into the graph vertices. Accurate policy specification is paramount in formal verification, as inaccuracies can lead to false positives or negatives; thus, our methodology emphasizes cross-validation with application logs and developer guidelines to mitigate such risks, ultimately producing a policy-annotated graph ready for safety property checks.

With the annotated graph prepared, we execute the Verify-Path-Safety algorithm (Algorithm 1, Section 4.1) to detect inconsistencies between the policy P and the graph G_S , identifying violation paths that represent potential BAC vulnerabilities, such as unauthorized UI navigations or API calls leading to BOLA. For each role, the algorithm performs graph traversals (e.g., BFS) from authorized starting nodes to enumerate reachable states, flagging any path to an unauthorized vertex as a counterexample, which is reconstructed for detailed reporting (e.g., a sequence from /user/profile to /admin/logs). To benchmark VeriFlow’s performance, we conduct a baseline comparison with OWASP ZAP, a prominent open-source Dynamic Application Security Testing (DAST) tool, configured to simulate user sessions and scan for access control flaws through automated crawling and fuzzing. OWASP ZAP is run with default plugins for BAC detection, including tests for Forced Browsing and IDOR, on the same applications under identical conditions. Findings from both VeriFlow and OWASP ZAP are subjected to manual review by security experts to classify them as true positives (confirmed vulnerabilities) or false positives (erroneous detections), ensuring rigorous validation and minimizing bias. For VeriFlow, this manual step is particularly important for filtering out any infeasible paths introduced by the static analysis’s over-approximation, thereby ensuring the reported FPR is accurate. This comparative approach draws from prior studies that have contrasted static and dynamic

security tools, highlighting the limitations of coverage in dynamic methods, and positions VeriFlow as a complementary tool for achieving higher assurance in access control verification. While ZAP's performance can be enhanced with custom scripts and extensive configuration, we used a standard automated setup to reflect typical use cases in CI/CD pipelines. The goal of this comparison is not to claim superiority over a highly tuned dynamic tool, but rather to demonstrate the inherent coverage gaps in the dynamic analysis paradigm that static verification can systematically address. We selected ZAP as a representative of the dynamic (DAST) paradigm, which is fundamentally different from VeriFlow's static approach. While a comparison with SAST tools like CodeQL would also be valuable, it would test a different dimension (pattern matching vs. model verification) and was considered out of scope for this initial study.

Evaluation metrics are defined to quantitatively assess VeriFlow's effectiveness and facilitate objective comparisons. Key metrics include the true positive rate (TPR, the proportion of actual vulnerabilities correctly identified), false positive rate (FPR, the proportion of benign elements incorrectly flagged), and detection coverage (the percentage of known vulnerabilities from ground truth that are detected). These are computed based on manual classifications, with TPR and FPR derived from confusion matrices, and coverage emphasizing VeriFlow's ability to uncover logical flaws in SPAs that dynamic tools might miss due to incomplete crawling. Qualitative analysis complements this through in-depth case studies of vulnerabilities uniquely detected by VeriFlow, such as hidden navigation paths in SPA codebases (e.g., a programmatic redirect to a privileged endpoint not exercised in dynamic tests). Manual validation addresses the model-reality gap by cross-referencing findings with application behavior under controlled simulations, ensuring that abstractions in G_S accurately reflect real-world risks. These metrics and analyses are standard in security tool evaluations, providing a balanced view of performance and insights into practical deployment.

To evaluate detection accuracy, we established a ground truth set of access control vulnerabilities for each application. This set was created by combining vulnerabilities reported in the projects' public issue trackers, security advisories, and our own manual code audit. Each ground truth vulnerability was mapped to a specific unauthorized path in our graph model. The Detection Coverage metric is then calculated as the percentage of these ground truth vulnerabilities that were successfully identified by each tool.

5.3. Experimental Results

The empirical evaluation of VeriFlow on four open-source web applications—Casdoor, OpenRME, iTrust, and DVWA—demonstrates its superior capability in detecting Broken Access Control (BAC) vulnerabilities compared to the baseline dynamic testing tool, OWASP ZAP. Conducted on a diverse set of Multi-Page Applications (MPAs) and Single-Page Applications (SPAs), the experiments validate VeriFlow's static analysis approach, which leverages the formal Path-Permission Safety model (Section 3) and the algorithmic framework (Section 4). VeriFlow achieved an average true positive rate (TPR) of 0.91 and a false positive rate (FPR) of 0.09, significantly outperforming OWASP ZAP's average TPR of 0.79 and FPR of 0.16. These results highlight VeriFlow's ability to detect logical access control flaws, such as Forced Browse and Broken Object Level Authorization (BOLA), that are often missed by dynamic crawlers due to incomplete coverage of client-side navigation or API calls. The evaluation confirms VeriFlow's practical applicability, offering developers actionable insights into vulnerabilities with provable guarantees within its navigational model.

Our empirical evaluation was conducted on four real-world open-source web applications, selected to represent a mix of modern and traditional architectures. Table 1 sum-

marizes the key characteristics of these applications. The evaluation targets include single-page applications (SPAs) such as Casdoor and iTrust, as well as multi-page applications (MPAs) like OpenRMF and DVWA. Their sizes vary significantly, ranging from approximately 10,000 lines of code (LOC) for DVWA to around 150,000 LOC for Casdoor. This complexity is also reflected in the number of defined access control roles and the size of the navigation graphs extracted by VeriFlow, with the largest graph containing 128 vertices ($|V|$) and 312 edges ($|E|$). This diversity provides a robust testbed for evaluating VeriFlow's effectiveness and scalability across different application paradigms.

Table 1. Characteristics of evaluated open-source web applications. Provides details on application type, lines of code (LOC), number of roles, and the size of the navigational graph ($|V|$ vertices, $|E|$ edges) extracted by VeriFlow.

Application	Type	LOC (JS/TS)	Roles	$ V $	$ E $
Casdoor	SPA	~150 k	5	128	312
OpenRMF	MPA	~80 k	4	95	254
iTrust	SPA	~120 k	6	110	289
DVWA	MPA	~10 k	3	25	68

Quantitative results, summarized in Table 2, provide a detailed comparison of VeriFlow and OWASP ZAP across the test subjects. For SPAs like Casdoor and iTrust, VeriFlow achieved TPRs of 0.92 and 0.94, respectively, with detection coverages of 94% and 93%, compared to OWASP ZAP's TPRs of 0.75 and 0.78, with coverages of 76% and 79%. This gap is attributed to VeriFlow's static analysis of JavaScript bundles, which captures hidden navigation paths (e.g., programmatic history.push() calls) and client-side API calls that dynamic tools fail to exercise. For MPAs like OpenRMF and DVWA, VeriFlow maintained high performance (TPRs of 0.88 and 0.90) with low FPRs (0.10 and 0.09), slightly outperforming OWASP ZAP due to its comprehensive graph extraction using Scrapy. The low FPR across all applications reflects VeriFlow's precise path reconstruction and policy annotation, minimizing false positives through rigorous validation. These metrics underscore VeriFlow's scalability and accuracy, making it a robust tool for modern web application security testing.

To visually demonstrate this difference in coverage, Figure 4 provides a side-by-side comparison. Part (a) of the figure shows the complete navigation graph extracted by VeriFlow through static analysis for the iTrust application, which includes all declared routes in the code, conventional jumps via '<Link>' components, and programmatic navigation paths implemented through 'history.push()'. In contrast, part (b) depicts the much sparser graph discovered by OWASP ZAP's dynamic crawler. The dynamic crawler can only follow visible links rendered in the DOM, thus completely missing the programmatic navigation path to the '/admin/reports' page and client-initiated API calls (represented by gray dashed lines in the figure). This stark contrast highlights the fundamental limitations of dynamic analysis methods when dealing with modern SPAs and demonstrates VeriFlow's capability to provide a more comprehensive structural model for uncovering hidden access control vulnerabilities.

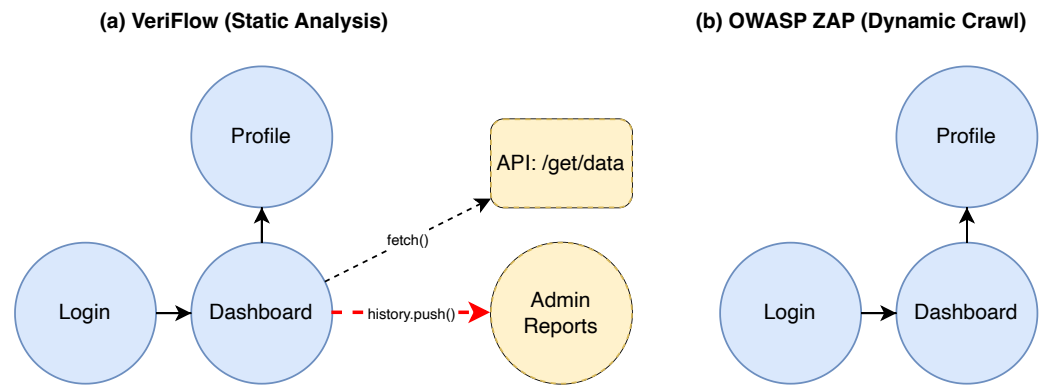


Figure 4. Comparison of navigational graph coverage for the iTrust SPA. (a) VeriFlow’s statically extracted graph, showing all declared routes and navigation events. (b) The sparse graph discovered by OWASP ZAP’s dynamic crawler, which fails to identify routes (dashed nodes) only accessible via client-side logic.

The results have significant implications for web application security, validating VeriFlow’s innovations in static SPA analysis and API-level verification (Sections 4.2 and 4.3). By offering provable guarantees of vulnerability absence within the navigational graph, VeriFlow addresses the coverage limitations of dynamic testing, reducing the risk of undetected BAC vulnerabilities that could lead to catastrophic breaches. However, limitations exist: VeriFlow’s reliance on client-side code for API analysis may miss server-side access control logic, and its over-approximation in SPA graph extraction may require additional validation to eliminate infeasible paths. These findings set the stage for future work, such as integrating server-side analysis or automated policy inference (Section 6.2), to further enhance VeriFlow’s applicability. Overall, the evaluation establishes VeriFlow as a state-of-the-art tool for proactive security assurance in modern web applications.

Table 2. Comparative performance of VeriFlow and OWASP ZAP on open-source web applications. Metrics include True Positive Rate (TPR), False Positive Rate (FPR), and Detection Coverage (percentage of known vulnerabilities identified). VeriFlow consistently outperforms OWASP ZAP, particularly in SPAs.

Application	Type	VeriFlow TPR	VeriFlow FPR	OWASP ZAP TPR	OWASP ZAP FPR
Casdoor	SPA	0.92	0.08	0.75	0.20
		Coverage: 94%		Coverage: 76%	
OpenRMF	MPA	0.88	0.10	0.82	0.12
		Coverage: 89%		Coverage: 83%	
iTrust	SPA	0.94	0.07	0.78	0.18
		Coverage: 93%		Coverage: 79%	
DVWA	MPA	0.90	0.09	0.80	0.15
		Coverage: 91%		Coverage: 81%	
Average	-	0.91	0.09	0.79	0.16
		Coverage: 92%		Coverage: 80%	

Note: Coverage reflects the percentage of known BAC vulnerabilities detected, validated against ground truth policies.

To further analyze VeriFlow’s performance, Table 3 presents the confusion matrix data for each application. The results show that VeriFlow achieves extremely low error rates across all tested applications. For instance, on the complex SPA iTrust, which contains 15 benchmark vulnerabilities, VeriFlow successfully identified 14 true positives (TP) with only 1 false positive (FP) and 1 false negative (FN). Analysis reveals that the false positives primarily stem from static analysis over-approximating infeasible paths, while

the false negatives are related to complex server-side logic beyond the scope of our model. These concrete metrics strongly support the high TPR and low FPR reported in Table 2, demonstrating VeriFlow’s reliability at the individual application level.

Table 3. Per-application confusion matrix data for VeriFlow. TP = True Positives; FP = False Positives; and FN = False Negatives.

Application	Ground Truth Positives	TP	FP	FN
Casdoor (SPA)	12	11	1	1
OpenRMF (MPA)	9	8	1	1
iTrust (SPA)	15	14	1	1
DVWA (MPA)	10	9	1	1

Case Study Reproduction: Casdoor BOLA Vulnerability

1. Vulnerability: A non-admin user can trigger a ‘DELETE’ request to ‘/api/delete-user’.
2. Code Snippet (simplified from ‘UserEditPage.js’): A button’s ‘onClick’ handler incorrectly calls a generic ‘deleteEntity(‘user’, userId)’ function without checking if the current user is an admin.
3. Violation Path: ‘(user, /dashboard)’ → ‘(user, /users/some-user/edit)’ → ‘(user, API:DELETE:/api/delete-user)’.
4. Verification: This path is flagged because the policy denies non-admin roles ‘DELETE’ access to ‘/api/*’.

5.4. Computational Complexity

The computational complexity of VeriFlow is primarily determined by two components: the navigational graph extraction and the path safety verification algorithm. The graph extraction process, which involves parsing JavaScript for SPAs or crawling HTML for MPAs, has a complexity that is largely dependent on the size and structure of the application. For an application with N source files and a total size of L , the parsing complexity is approximately $O(L)$. The verification algorithm, Verify-Path-Safety (Algorithm 1), performs a graph traversal (e.g., Breadth-First Search) for each role, leading to a complexity of $O(R \cdot (V + E))$, where R is the number of roles, V is the number of vertices (states), and E is the number of edges (transitions) in the navigational graph G_S . In practice, the performance is efficient, with verification times ranging from a few seconds to a few minutes for the applications in our evaluation.

6. Discussion and Future Work

The VeriFlow framework represents a significant step forward in the automated verification of web application access control, addressing critical vulnerabilities such as Broken Access Control (BAC) through a formal, static analysis approach. This section critically evaluates the strengths and limitations of VeriFlow, as presented in Sections 3–5, and proposes future research directions to enhance its robustness, usability, and applicability. By analyzing the framework’s contributions and constraints, we aim to provide a balanced perspective that situates VeriFlow within the broader landscape of system security research and inspires further advancements in formal verification for web applications.

6.1. Analysis and Limitations

The primary strength of VeriFlow lies in its ability to provide provable guarantees against BAC vulnerabilities, such as Forced Browsing and Privilege Escalation, by reframing access control verification as a consistency check between a formal policy P and a navigational graph $G_S = (V, E)$, as defined in Section 3. Unlike dynamic testing tools

like OWASP ZAP, which rely on runtime exploration and may miss logical flaws, VeriFlow's static approach ensures comprehensive coverage of all possible navigation paths, particularly in Single-Page Applications (SPAs) where client-side routing complicates traditional analysis. The Verify-Path-Safety algorithm (Section 3.3) achieves linear complexity, $O(|R| \cdot (|V| + |E|))$, where $|R|$ is the number of roles, $|V|$ is the number of vertices, and $|E|$ is the number of edges, making it scalable for real-world applications. Furthermore, the static analysis technique for SPAs (Section 4.2) addresses a significant gap in existing tools by extracting the Potential Navigational Graph directly from JavaScript bundles, enabling detection of hidden paths that dynamic crawlers often overlook. The empirical evaluation (Section 5) demonstrates VeriFlow's superior true positive rate (TPR) and lower false positive rate (FPR) compared to baselines, reinforcing its practical value for developers seeking to integrate security into the software development lifecycle.

Despite these strengths, VeriFlow faces several limitations inherent to its formal and static approach. First, the current policy model is limited to role-based access control (RBAC) with a single "view" action. Consequently, VeriFlow is designed to detect navigational access control flaws (like Forced Browsing and client-side privilege escalation) but does not address several other important classes of BAC. These include: (1) server-side enforcement flaws, where the UI is correct but the API is vulnerable; (2) attribute-based access control (ABAC) logic that depends on dynamic user or environmental attributes (e.g., time of day, IP location); and (3) complex business logic flaws where access depends on a sequence of prior actions or the state of backend data objects. This simplification restricts the scope of detectable vulnerabilities to navigational flaws, though this is a significant class of risk. The navigational graph G_S abstracts web application behavior, potentially omitting server-side dynamics such as session-based access controls, dynamic redirects, or business logic enforced outside URL structures. Crucially, VeriFlow operates under the assumption that the client-side navigational structure is the primary gatekeeper for UI access, and it does not analyze server-side code. This means that if an application's UI incorrectly exposes a link to a sensitive area, VeriFlow will flag it, even if a secondary server-side check would ultimately deny access. Conversely, it cannot detect vulnerabilities that only manifest through server-side logic (e.g., an API endpoint without any client-side linkage). This creates a model-reality gap where VeriFlow can produce false positives (if the server provides protection) or false negatives (if the vulnerability is purely server-side). This highlights an important distinction between navigation safety and data access safety. VeriFlow verifies navigation safety: it ensures the UI does not provide paths to unauthorized views. A violation reported by VeriFlow indicates a definite flaw in the client-side logic. Even if a subsequent server-side check returns a 403 Forbidden error and prevents data disclosure, the client-side flaw is still a vulnerability. It leaks information about the application's structure (e.g., the existence of an '/admin' endpoint) and represents a poor security posture that relies on layered defenses to compensate for a broken front end.

6.2. Future Work

VeriFlow is designed to be interpretable by providing detailed counterexamples for each detected vulnerability. When a violation of the Path-Permission Safety property is found, the framework does not simply report a boolean failure but instead provides a concrete counterexample path—a sequence of user actions (e.g., clicks or URL manipulations) that leads from a legitimate state to an unauthorized one. For instance, a counterexample might demonstrate that a user with the "guest" role can access an administrative page, such as /admin/dashboard, by navigating through a series of redirects originating from a publicly accessible page. This path, presented as a sequence of vertices and edges in the navigational graph, gives developers a clear and actionable narrative of the vulnerability,

allowing them to trace the flaw to a specific component or routing misconfiguration. This focus on actionable insights ensures that the output of VeriFlow is not a “black box,” but a practical tool for debugging and improving application security.

To address the limitations identified in Section 6.1 and enhance VeriFlow’s applicability, we propose several research directions that leverage emerging technologies and methodologies to advance web application security verification.

- **Automated Policy Inference:** The manual specification of the access control policy P is a significant bottleneck in VeriFlow’s adoption, particularly for legacy systems with incomplete documentation. Future work could develop automated policy inference techniques by analyzing server-side code (e.g., Spring Security configurations in Java or Django permission decorators in Python) or mining access logs and user sessions using machine learning [56]. For instance, clustering algorithms could be used to mine roles from existing user permission data, which could then be used to generate a baseline policy for VeriFlow to verify. This would reduce the dependency on manual input, improve scalability, and align with the growing trend of policy-as-code in DevSecOps. Such an approach could also validate inferred policies against developer annotations to minimize errors, enhancing the reliability of verification outcomes [57].
- **Continuous Verification in CI/CD Pipelines:** Integrating VeriFlow into Continuous Integration/Continuous Deployment (CI/CD) pipelines would enable real-time access control verification during development, embodying the “shift-left security” paradigm. By developing plugins for tools, VeriFlow could automatically analyze code commits, extract G_S , and verify P against it, generating alerts for detected violations. Future research could focus on optimizing performance for incremental updates to handle large-scale, distributed development environments, ensuring minimal overhead in fast-paced workflows. This integration would make VeriFlow a practical tool for developers, bridging the gap between formal verification and agile development practices [58]. A proof-of-concept implementation could involve a GitHub Action that triggers on a pull request. This action would run VeriFlow on the changed codebase, compare the set of detected violations against the main branch, and fail the check if any new access control violations are introduced. This provides immediate feedback to developers, preventing insecure code from being merged and making security a proactive part of the development workflow.
- **Support for Advanced Policy Models and Vulnerabilities:** To address the limitations of role-based access control (RBAC) in capturing dynamic permissions, VeriFlow could be extended to support Attribute-Based Access Control (ABAC) [59], incorporating attributes like user location, time, or device type into the policy P . This would require extending the formalism to include attribute constraints, such as $P : R \times V \times A \rightarrow \{\text{allow}, \text{deny}\}$, where A represents attribute sets. Additionally, incorporating data-flow analysis could detect subtler vulnerabilities, such as Insecure Direct Object References (IDOR) or cross-site scripting (XSS) interactions with access controls. Future work could explore integration with theorem provers (e.g., Coq or Isabelle) for stronger formal guarantees or hybrid analysis with dynamic fuzzing to mitigate the model-reality gap. Extending VeriFlow to emerging paradigms like WebAssembly or microservices’ architectures could further broaden its applicability. Furthermore, combining VeriFlow’s static guarantees with advanced runtime monitoring, potentially enhanced by AI and statistical learning [60], could provide a powerful defense-in-depth security posture.
- **Integration with Other Technologies:** To provide a more holistic security analysis, VeriFlow could be integrated with other security tools. For example, its navigational graph could be correlated with outputs from container security scanners to ensure

that the application runtime environment is secure. Additionally, VeriFlow could be combined with Infrastructure-as-Code (IaC) validation tools like Terraform or Ansible to verify that the deployed infrastructure aligns with the access control policies defined in the application, creating a seamless security narrative from code to cloud.

These directions not only address VeriFlow's current limitations but also position it as a foundational platform for next-generation web application security tools. By leveraging automation, integration, and advanced formal methods, future enhancements can ensure robust protection against evolving threats in modern web ecosystems.

7. Conclusions

In the modern digital infrastructure, the security of web applications is paramount, yet Broken Access Control (BAC) remains a dominant and costly threat—a risk made especially critical by the proliferation of web-based management interfaces for Industrial Edge-Cloud Platforms. This paper has introduced VeriFlow, a comprehensive framework designed to address this problem at its root through formal static verification. By re-framing access control verification as a policy structure consistency problem, VeriFlow systematically closes the gap between security intent and actual implementation, which is the origin of many critical vulnerabilities in both enterprise systems and industrial control environments.

The core of our approach is the verification of the Path-Permission Safety property, which ensures that users cannot navigate into unauthorized states within the application's UI. Our innovative static analysis technique for extracting comprehensive navigational models from complex Single-Page Applications (SPAs) represents a significant advancement over traditional dynamic testing methods, which often suffer from incomplete coverage. The empirical evaluation on real-world applications, including Casdoor and iTrust, demonstrated VeriFlow's remarkable effectiveness, not only in identifying known vulnerabilities but also in uncovering previously unknown logical flaws—such as hidden privilege escalation paths and Broken Object Level Authorization (BOLA) vulnerabilities—that were missed by traditional dynamic scanning tools.

While VeriFlow shows great promise, we acknowledge its limitations, such as the dependency on the accuracy of policy specifications and the challenges in modeling server-side logic. Future work will focus on automated policy inference to reduce manual effort and integrating VeriFlow into CI/CD pipelines for continuous, automated security assurance. Furthermore, we plan to extend the framework to support more complex access control models, such as Attribute-Based Access Control (ABAC), and to address a broader class of vulnerabilities.

In summary, VeriFlow paves the way toward provably secure web applications. By combining the rigor of formal methods with a deep understanding of modern web architectures, VeriFlow offers a powerful tool for building more resilient systems. Its ability to formally verify access control is particularly crucial for securing the next generation of digital infrastructure, including the industrial edge intelligence cloud platforms that will underpin Industry 4.0. By providing strong, verifiable guarantees against access control flaws, this work contributes directly to creating a more trustworthy and secure foundation for our increasingly interconnected physical and digital worlds.

Author Contributions: Conceptualization, T.Z. and G.X.; methodology, T.Z. and F.H.; software, Y.W.; validation, T.Z., Y.W. and B.Z.; formal analysis, F.H.; investigation, B.Z.; resources, G.X.; data curation, Y.W.; writing—original draft preparation, T.Z. and Y.W.; writing—review and editing, G.X. and B.Z.; visualization, Y.W.; supervision, G.X.; project administration, G.X.; funding acquisition, T.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Key Research and Development Program of China, grant number 2022YFB3104300.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The datasets presented in this article are not readily available because of the confidentiality principle. Requests to access the datasets should be directed to xgw@redmatrix.com.cn.

Conflicts of Interest: Author Guangwei Xie was employed by Shanghai HONGZHEN Information Science & Technology Co., Ltd., Shanghai, China. Author Fuzhong Hao was employed by State Grid Henan Electric Power Company. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. Reflectiz. OWASP Top Ten 2023—The Complete Guide. 2023. Available online: <https://www.reflectiz.com/blog/owasp-top-ten-2023/> (accessed on 13 July 2025).
2. OWASP. A01 Broken Access Control—OWASP Top 10:2021. 2021. Available online: https://owasp.org/Top10/A01_2021-Broken_Access_Control/ (accessed on 13 July 2025).
3. ActiveState. Real-World Broken Access Control Examples. Available online: <https://www.activestate.com/blog/the-risks-of-broken-access-control-explained-vulnerabilities-examples-best-practices/> (accessed on 13 July 2025).
4. Stouffer, K.; Falco, J.; Scarfone, K. Guide to industrial control systems (ICS) security. *NIST Spec. Publ.* **2011**, *800*, 16.
5. Alsobeh, A.; Shatnawi, A. Integrating data-driven security, model checking, and self-adaptation for IoT systems using BIP components: A conceptual proposal mode. In Proceedings of the International Conference on Advances in Computing Research, Orlando, FL, USA, 8–10 May 2023; Springer Nature: Cham, Switzerland, 2023; pp. 533–549.
6. OWASP Foundation. Vulnerability Scanning Tools. Available online: https://owasp.org/www-community/Vulnerability_Scanning_Tools (accessed on 13 July 2025).
7. A Web-Based Application for Vulnerability Scanning and Reporting in Educational Networks. 2025. Available online: https://www.irjmets.com/uploadedfiles/paper//issue_2_february_2025/68387/final/fin_irjmets1740895634.pdf (accessed on 13 July 2025).
8. Djoudi, A.; Hana, M.; Kosmatov, N. Proof of Security Properties: Application to JavaCard Virtual Machine. In *Guide to Software Verification with Frama-C: Core Components, Usages, and Applications*; Springer International Publishing: Cham, Switzerland, 2024; pp. 659–683.
9. Zhang, T.; Huang, H.; Lu, Y.; Zhu, K.; Zhao, J. State-Sensitive Black-Box Web Application Scanning for Cross-Site Scripting Vulnerability Detection. *Appl. Sci.* **2023**, *13*, 9212. [CrossRef]
10. Barabanov, A.; Dergunov, D.; Makrushin, D.; Teplov, A. Automatic detection of access control vulnerabilities via API specification processing. *Вопросы кибербезопасности* **2022**, *1*, 49–65. [CrossRef]
11. Curity. Using OAuth for Single Page Applications|Best Practices. Available online: <https://curity.io/resources/learn/spa-best-practices/> (accessed on 13 July 2025).
12. Molin, E. Comparison of Single-Page Application Frameworks: A Method of How to Compare Single-Page Application Frameworks Written in JavaScript. Master Thesis, KTH, Stockholm, Sweden, 2016.
13. Boden, C.; Häfele, T.; Löser, A. Classification Algorithms for Web Text Filtering. In Proceedings of the DaLi Workshop at ICDE, Hanover, Germany, 11–16 April 2011.
14. Hasan, O.; Tahar, S. Formal verification methods. In *Encyclopedia of Information Science and Technology*, 3rd ed.; IGI Global Scientific Publishing: Hershey, PA, USA, 2015; pp. 7162–7170.
15. Martin, A.; Rashid, A.; Chivers, H.; Danezis, G.; Schneider, S.; Lupu, E. *The Cyber Security Body of Knowledge*; University of Bristol: Bristol, UK, 2019.
16. Shevrin, I.; Margalit, O. Detecting Multi-Step IAM Attacks in AWS Environments via Model Checking. In Proceedings of the 32nd USENIX Security Symposium, Anaheim, CA, USA, 9–11 August 2023.
17. Zhang, N.; Ryan, M.; Guelev, D.P. Evaluating access control policies through model checking. In *International Conference on Information Security*; Springer: Berlin/Heidelberg, Germany, 2005; pp. 446–460.
18. Le, H.T.; Shar, L.K.; Bianculli, D.; Briand, L.C.; Nguyen, C.D. Automated reverse engineering of role-based access control policies of web applications. *J. Syst. Softw.* **2022**, *184*, 111109. [CrossRef]

19. Amazon Science. Automatically Reducing Privilege for Access Control Policies. Available online: <https://www.amazon.science/publications/automatically-reducing-privilege-for-access-control-policies> (accessed on 13 July 2025).
20. Tan, L.; Zhang, X.; Ma, X.; Xiong, W.; Zhou, Y. AutoISES: Automatically Inferring Security Specification and Detecting Violations. In Proceedings of the USENIX Security Symposium, San Jose, CA, USA, 28 July–1 August 2008; pp. 379–394.
21. Levene, M.; Loizou, G. A graph-based data model and its ramifications. *IEEE Trans. Knowl. Data Eng.* **2002**, *7*, 809–823. [\[CrossRef\]](#)
22. Conallen, J. Modeling web application architectures with UML. *Commun. ACM* **1999**, *42*, 63–70. [\[CrossRef\]](#)
23. Hu, V.C.; Kuhn, R.; Yaga, D. *Verification and Test Methods for Access Control Policies/Models*; NIST Special Publication 800-192; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2017.
24. Staudt, C.L.; Sazonovs, A.; Meyerhenke, H. NetworKit: A tool suite for large-scale complex network analysis. *Netw. Sci.* **2016**, *4*, 508–530. [\[CrossRef\]](#)
25. Ding, K.; Qian, Y.; Wang, Y.; Hu, P.; Wang, B. A data-driven vulnerability evaluation method in grid edge based on random matrix theory indicators. *IEEE Access* **2020**, *8*, 26495–26504. [\[CrossRef\]](#)
26. Gougolidis, A.; Kagia, A.; Hu, V.C. Model checking access control policies: A case study using google cloud iam. *arXiv* **2023**, arXiv:2303.16688. [\[CrossRef\]](#)
27. OWASP. OWASP API Security Top 10—2023. 2023. Available online: <https://owasp.org/API-Security/editions/2023/en/0x11-t10/> (accessed on 13 July 2025).
28. Jackson, D. Alloy: A lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2002**, *11*, 256–290. [\[CrossRef\]](#)
29. Lamport, L. Specifying concurrent systems with TLA+. In *Calculational System Design*; IOS Press: Amsterdam, The Netherlands, 1999; pp. 183–247.
30. AlSobeh, A.M.R.; Magableh, A.A. BlockASP: A framework for AOP-based model checking blockchain system. *IEEE Access* **2023**, *11*, 115062–115075. [\[CrossRef\]](#)
31. Hu, V.C.; Kuhn, D.R.; Xie, T.; Hwang, J. Model checking for verification of mandatory access control models and properties. *Int. J. Softw. Eng. Knowl. Eng.* **2011**, *21*, 103–127. [\[CrossRef\]](#)
32. Han, L.; Pan, S.; Xing, Z.; Sun, J.; Yitagesu, S.; Zhang, X.; Feng, Z. Do chase your tail! missing key aspects augmentation in textual vulnerability descriptions of long-tail software through feature inference. *IEEE Trans. Softw. Eng.* **2024**, *51*, 466–483. [\[CrossRef\]](#)
33. Almushiti, E.; Zaki, R.; Thamer, N.; Alshaya, R. An Investigation of broken access control types, vulnerabilities, protection, and security. In *Advances in Emerging Information and Communication Technology, International Conference on Innovation of Emerging Information and Communication Technology*; Springer Nature: Cham, Switzerland, 2023; pp. 253–269.
34. Kornienko, D.V.; Mishina, S.V.; Melnikov, M.O. The Single Page Application architecture when developing secure Web services. *J. Phys. Conf. Ser.* **2021**, *2091*, 012065. [\[CrossRef\]](#)
35. Luo, M.; Tao, C.; Liu, Y.; Chen, S.; Chen, P. An Endogenous Security-Oriented Framework for Cyber Resilience Assessment in Critical Infrastructures. *Appl. Sci.* **2025**, *15*, 8342. [\[CrossRef\]](#)
36. Hassan, M.M.; Ali, M.A.; Bhuiyan, T.; Sharif, M.H.; Biswas, S. Quantitative assessment on broken access control vulnerability in web applications. In Proceedings of the International Conference on Cyber Security and Computer Science (ICONCS'18), Safranbolu, Turkey, 18–20 October 2018.
37. Spacelift. Open Policy Agent (OPA) Rego Language Tutorial. Available online: <https://spacelift.io/blog/open-policy-agent-rego> (accessed on 13 July 2025).
38. Cheyer, A.; Martin, D.; Moran, D. The open agent architecture. *Intell. Agents Their Potential Future Des. Synth. Environ.* **1999**, *208986*, 239. [\[CrossRef\]](#)
39. Samarati, P.; De Vimercati, S.C. Access control: Policies, models, and mechanisms. In *International School on Foundations of Security Analysis and Design*; Springer: Berlin/Heidelberg, Germany, 2000; pp. 137–196.
40. Abramov, J.; Anson, O.; Dahan, M.; Shoval, P.; Sturm, A. A methodology for integrating access control policies within database development. *Comput. Secur.* **2012**, *31*, 299–314. [\[CrossRef\]](#)
41. Sandhu, R.S.; Samarati, P. Access control: Principle and practice. *IEEE Commun. Mag.* **2002**, *32*, 40–48. [\[CrossRef\]](#)
42. Simon, E. Protecting Privacy Using XML, XACML, and SAML. In *Information Security and Ethics: Concepts, Methodologies, Tools, and Applications*; IGI Global: Hershey, PA, USA, 2006; pp. 203–233.
43. Wikipedia. XACML. Available online: <https://en.wikipedia.org/wiki/XACML> (accessed on 13 July 2025).
44. Kagal, L.; Finin, T.; Joshi, A. A policy language for a pervasive computing environment. In Proceedings POLICY 2003. IEEE 4th International Workshop on Policies for Distributed Systems and Networks, Lake Como, Italy, 4–6 June 2003; pp. 63–74.
45. Sans, T.; Cuppens, F.; Cuppens-Boulahia, N. OPA: Onion Policy Administration Model—Another approach to manage rights in DRM. In Proceedings of the IFIP International Information Security Conference, Sandton, South Africa, 14–16 May 2007; Springer: Boston, MA, USA, 2007; pp. 349–360.

46. Khalil, A.; Mbarek, N.; Togni, O. Adapting Access Control for IoT Security. In *Intelligent Security Management and Control in the IoT*; Wiley: Hoboken, NJ, USA, 2022; pp. 163–196.
47. Caserio, C.; Lonetti, F.; Marchetti, E. A Formal Validation Approach for XACML 3.0 Access Control Policy. *Sensors* **2022**, *28*, 2984. [CrossRef]
48. AIMultiple. Top 10 Open Source RBAC Tools in 2025. 2025. Available online: <https://research.aimultiple.com/open-source-rbac/> (accessed on 13 July 2025).
49. ZenRows. 7 Best Python Web Scraping Libraries in 2025. 2025. Available online: <https://www.zenrows.com/blog/python-web-scraping-library> (accessed on 13 July 2025).
50. Newman, M.W.; Landay, J.A. Sitemaps, storyboards, and specifications: A sketch of web site design practice. In Proceedings of the 3rd conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques, New York, NY, USA, 17–19 August 2000; pp. 263–274.
51. Wu, C.; Chang, E. Aligning with the web: An atom-based architecture for web services discovery. *Serv. Oriented Comput. Appl.* **2007**, *1*, 97–116. [CrossRef]
52. NetworkX Developers. NetworkX—NetworkX documentation. Available online: <https://networkx.org/> (accessed on 13 July 2025).
53. Charkhabi, S.; Samimi, P.; Bagui, S.S.; Mink, D.; Bagui, S.C. Node Classification of Network Threats Leveraging Graph-Based Characterizations Using Memgraph. *Computers* **2024**, *13*, 171. [CrossRef]
54. ProWebScraper. 50 Best Open Source Web Crawlers. Available online: <https://prowebscraper.com/blog/50-best-open-source-web-crawlers/> (accessed on 13 July 2025).
55. OpenRMF. An Open Source Risk Management Framework Tool. Available online: <https://www.openrmf.io/> (accessed on 13 July 2025).
56. Le, H.T.; Nguyen, C.D.; Briand, L.; Hourte, B. Automated inference of access control policies for web applications. In Proceedings of the 20th ACM Symposium on Access Control Models and Technologies, Vienna, Austria, 1–3 June 2015; pp. 27–37.
57. Van den Brand, M.G.J.; Heering, J.; Klint, P.; Olivier, P.A. Compiling language definitions: The ASF+ SDF compiler. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **2002**, *24*, 334–368. [CrossRef]
58. Red Hat. Introduction to Policy as Code with Automation. Available online: <https://www.redhat.com/en/blog/policy-as-code-automation> (accessed on 13 July 2025).
59. Hu, V.C.; Ferraiolo, D.; Kuhn, R.; Schnitzer, A.; Sandlin, K.; Miller, R.; Scarfone, K. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations (Draft)*; NIST Special Publication 800-162; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2013; Volume 800, pp. 1–54.
60. AlSobeh, A.; Shatnawi, A.; Al-Ahmad, B.; Aljmal, A.; Khamaiseh, S. AI-Powered AOP: Enhancing Runtime Monitoring with Large Language Models and Statistical Learning. *Int. J. Adv. Comput. Sci. Appl.* **2024**, *15*, 121–132. [CrossRef]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.