# Theseus: Smart Web Crawling via Resource-Guided Semantic Modeling


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

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

Wenxiao Yao
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China
yaowenxiao25e@ict.ac.cn

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

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

Youlong Chen
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Zhongguancun Laboratory
Beijing, China
chenyoulong20g@ict.ac.cn

Dong Liu
Zhongguancun Laboratory
Beijing, China
liud@zgclab.edu.cn

Mengna Ma
Zhongguancun Laboratory
Beijing, China
mamn@zgclab.edu.cn

Yong Liu
Zhongguancun Laboratory
Qi An Xin Technology Group Inc.
Beijing, China
liuyong03@qianxin.com

Qinfen Hao
Institute of Computing Technology, CAS
Zhongguancun Laboratory
Beijing, China
haoqinfen@ict.ac.cn

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



## Abstract

Crawlers are critical components of web vulnerability scanners, and their effectiveness is vital for discovering new vulnerabilities. However, existing crawlers often perform poorly in two core aspects of crawling: the effective scheduling of user actions and deduplication. This paper addresses these limitations with a smart crawling tool named Theseus, which leverages LLMs to infer the intent behind user actions and models them as resource operations to precisely guide both navigation scheduling and deduplication. Theseus demonstrates significant advantages over existing crawlers through its dependency-aware navigation scheduling and resource-operation-guided deduplication. Experimental results show that, compared to the three state-of-the-art crawlers, BlackWidow, EvoCrawl and YuraScanner, Theseus improves code coverage by 93.5%, 72.1% and 96.8%, respectively. This enhanced coverage enabled the detection of 36 additional XSS vulnerabilities, of which 21 were previously undiscovered and 12 have received CVE identifiers.



## CCS Concepts

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

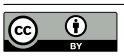
## Keywords

Web Crawling; LLM; Web Security

**ACM Reference Format:**
Yongheng Huang, Chenghang Shi, Wenxiao Yao, Jie Lu, Haofeng Li, Youlong Chen, Dong Liu, Mengna Ma, Yong Liu, Qinfen Hao, and Lian Li. 2026. Theseus: Smart Web Crawling via Resource-Guided Semantic Modeling. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3830454.3832604



*Corresponding author



*CCS '26, The Hague, Netherlands.*



ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832604


## 1 Introduction

The widespread adoption of web applications has made them primary targets for security attacks. According to [2], there were a significant 40,289 web vulnerabilities reported in 2024, recording

a 72% year-over-year increase. Web vulnerability scanners [11, 13, 17, 25, 31, 36, 42] have emerged as essential testing tools to secure web applications. These tools identify vulnerabilities in web applications by automatically crawling web pages to discover the attack surface (e.g., URLs and form fields), then fuzzing or injecting specially crafted input values to trigger vulnerabilities.

The effectiveness of a web vulnerability scanner largely depends on its crawler component [35]: vulnerabilities may remain undetected if the crawler fails to cover a sufficient portion of the application's attack surface during testing. Effective crawling further hinges on two core capabilities: accurately executing actions in a proper order to expose new states (*navigation scheduling*), and determining whether an action on a page (e.g., clicking a button) can lead to a previously unseen page (*deduplication*). The above two functionalities operate synergistically: navigation scheduling ensures thorough exploration of the application's states, while deduplication guarantees performance by eliminating repeated actions, thus avoiding the unnecessary processing of recurring states. The efficacy of these two complementary functions is crucial to the overall performance of web crawlers.

**Existing Work**. However, current web crawlers exhibit significant limitations in both navigation scheduling and deduplication [35]. Existing crawlers typically employ navigation algorithms such as breadth-first search (BFS) [4, 34], depth-first search (DFS) [9, 19], or random exploration [18, 29], which merely determine the order of actions based on their discovery sequence, without considering their logical dependencies or semantic relationships.

This lack of semantic-aware ordering often results in insufficient coverage. For instance, in a blog management system, executing the `Delete` action before the `comment` action on a post can prematurely terminate the crawling process, leaving pages for commenting on a post unexplored, as users cannot comment on deleted posts.

Additionally, the majority of crawlers perform deduplication via URL matching [13, 35], which makes them unable to handle actions–like JavaScript events–that do not trigger new URLs. A few studies [6, 15, 40] attempt to distinguish actions by assessing the structural differences of their corresponding elements in DOM trees, or by comparing the structures of the resulting web pages after the actions are performed. If the edit distance between these elements (or resulting web pages) is less than a manually configured threshold, the actions are regarded as the same. However, this alternative approach is computationally expensive. Moreover, significant variations between web pages often render the manually configured threshold inappropriate, leading to both over- and under-deduplication.

**This Work**. This paper aims to address the limitations of existing crawlers by analyzing the intents behind various actions on web pages. These action intents can precisely guide both navigation scheduling and deduplication. For instance, in a blog management system, actions that edit or comment on a post need to be executed before the `Delete` action that removes the post. Additionally, actions with functionally-equivalent intents (e.g., commenting on distinct posts) can be filtered out. The key challenge is: *how can we understand the drastically different intents behind user actions and accurately model the semantics of these intents in a timely fashion?*

We tackle the above challenge from three aspects, as follows.

- *A resource-guided semantic model.* User actions commonly perform certain operations–specifically the CRUD (`Create`, `Read`, `Update`, `Delete`) types of operations–on application resources. Here, the term *resources* refers to entity objects in web applications, including both data stored in backend storage systems (such as user accounts or blog posts) and intermediate data structures maintained by the frontend (such as form data). Both types of resources play significant roles in application logic. Resources follow a hierarchical structure, where child resources (e.g., comments on a post) belong to their parent resources (e.g., the post).
  By modeling user actions as resource operations, we offer fine-grained guidance for navigation scheduling and deduplication. For navigation scheduling, dependencies between resource operations naturally form a directed acyclic graph (DAG), where operations on the same resource are ordered according to their CRUD types. Operations on child resources follow the `Create` operations on parent resources, and precede the `Delete` operations on parent resources. We demonstrate that this topological ordering ensures comprehensive state exploration. For deduplication, actions performing same operations on identical resources are often semantically equivalent. These actions can be safely discarded, even if they have distinct URLs or DOM structures.
- *LLM-assisted semantic inference.* Based on our semantic model, how can we precisely identify which resource operation is associated with a particular user action? This task has proven difficult because web applications vary widely in their implementation, resource naming conventions, and UI structures. As a result, rule-based systems are often too rigid, and machine learning models are only effective on applications they have been trained on.
  The emergence of large language models (LLMs) [3, 38, 43] provides a promising solution to this challenge. Prior task-driven agents for web and mobile applications, such as YuraScanner [36] and GptDroid [26], demonstrate that pre-trained LLMs can align user actions with semantic intents when supplied with structured context. Following this evidence, we furnish the LLM with rich contextual information—including the application's purpose, page metadata, and relevant DOM elements—enabling it to analyze the page and infer its semantics. Through this context-aware analysis, the LLM identifies not only nuanced operations and their types (for instance, both blog editing and blog categorization are `Update` type of operations on the target post), but also recognizes complex resource dependency relationships (such as comments belonging to blog posts).
- *An efficient implementation.* Efficiency and the ability to act on numerous actions promptly are crucial for web crawlers. However, integrating LLMs can introduce considerable time overhead. Sometimes, an LLM may require tens of seconds to interpret the intent behind an action, and most LLMs enforce tokens-per-minute (TPM) rate limitations. These factors present significant challenges to adopting LLMs in web crawling tasks.
  To mitigate these performance challenges, we have developed a series of optimizations. Our framework is designed

so that the tasks of crawling web pages run in parallel with those invoking LLMs, thereby tolerating LLM-induced delays. Furthermore, we avoid redundant invocations of LLMs by skipping those actions that result in semantically equivalent URLs.

To this end, we develop Theseus[1], a smart crawler that traverses web applications by using inferred resource operations as guiding clues. Theseus introduces two key improvements over traditional crawling: (1) it schedules actions based on the dependence order of their corresponding resource operations, ensuring logical progression through the application and maximizing coverage, and (2) it detects and eliminates functionally equivalent actions based on analyzing their intended operations on resources, preventing both over- and under-deduplication.

We have comprehensively evaluated Theseus on 20 real-world applications. Experimental results demonstrate that Theseus successfully identified 92,104 resource operations and 16,955 inter-resource dependencies, with false positive rates of 19.9% and 28.9%, respectively. These identified elements enable Theseus to achieve significant improvements in code coverage compared to the three state-of-the-art crawlers, namely BlackWidow [13], EvoCrawl [17] and YuraScanner [36], with increases of 93.5%, 72.1% and 96.8%, respectively. Compared to the combined performance of these three crawlers, Theseus achieved an improvement of 29.9% in code coverage. Furthermore, this enhanced coverage facilitated vulnerability scanners in detecting 36 additional XSS vulnerabilities, of which 21 were previously undiscovered and 12 received CVE identifiers.

**Contributions.** In summary, this paper makes the following key contributions:

- We propose a novel approach to model user action intents as resource operations. This model offers fine-grained guidance for the critical functionalities–navigation scheduling and deduplication–in web crawling tasks, thereby improving coverage.
- We develop Theseus, a smart and efficient web crawler that leverages large language models to infer the semantics of resource operations from user actions and, crucially, utilizes these semantics to guide web crawlers for effective navigation scheduling and deduplication.
- We evaluate Theseus on 20 real-world applications, demonstrating significant improvements over existing approaches in code coverage by 29.9%, which enabled the detection of 36 additional XSS vulnerabilities.
- To facilitate future research, we have open-sourced our implementation at https://github.com/Theseus-c/Theseus.

## 2 Background and Motivation

In this section, we first introduce the fundamental concepts of web crawlers. Next, we present an example of a web page with various clickable user actions. This example highlights the limitations of existing crawlers in navigation scheduling and deduplication when they fail to consider the semantics behind different user actions on the web page. Finally, we briefly introduce how our approach addresses these limitations by modeling user actions as resource operations.

[1]The name Theseus is inspired by the Greek mythological hero Theseus, who navigated the labyrinth using a ball of thread (clue) to find his way out.

### 2.1 Web Crawlers

The workflow of a web crawler consists of the following phases.

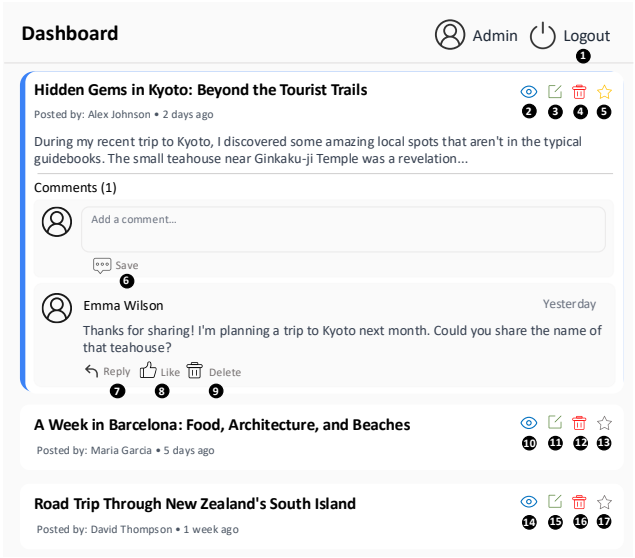
- *Phase I: Initial Configuration.* The process begins with users defining seed URLs as starting points and setting crawling configuration parameters, such as maximum crawling duration and user credentials. During this phase, an initial action queue is created and populated with the seed URLs as the first set of actions to be processed.
- *Phase II: Navigation Scheduling.* This phase selects and executes a target action from the action queue. Note that user actions are not limited to simple URL-triggering actions but also include frontend actions, such as clicks and form filling events. These non-URL actions are also valuable, since they may be necessary to eventually trigger new URLs.
- *Phase III: Action Extraction.* After the target action is executed, this phase extracts new actions from the resulting page, including all clickable actions on the page—such as form submissions, element interactions (e.g., clicks, double-clicks), new URLs, etc. These newly extracted actions are then added to the action queue for further processing.
- *Phase IV: Deduplication.* For efficiency, the crawler implements various algorithms to identify duplicate actions. Modern crawlers [13, 35] compare the DOM structures of web elements associated with user actions, as well as their targeting URLs or resulting pages, to determine whether they would lead to previously explored content. Duplicated actions are then removed from the action queue to avoid revisiting repeated states.

This entire workflow operates as an iterative cycle that continues until the action queue becomes empty, or until the predefined depth thresholds or time limit are reached. Phase II (navigation scheduling) repeatedly selects and executes actions from the action queue, uncovering more actions on the resulting pages (Phase III). Phase IV deduplicates these new actions and retains only previously unvisited actions in the action queue for further processing.

### 2.2 A Motivation Example

Figure 1(a) illustrates a simplified web page from a travel blog website, representing a typical Content Management System (CMS) application. This page enables users to publish travel experiences, view posts created by others, and engage in interactive commenting. The page comprises 17 clickable actions that can be categorized into 8 functional groups: logout (❶), view blog (❷, ❿, ⓮), edit blog (❸, ⓫, ⓯), delete blog (❹, ⓬, ⓰), bookmark blog (❺, ⓭, ⓱), comment on blog (❻), reply to comments (❼), like comments (❽), and delete comments (❾).

An optimal click sequence for this motivation example is depicted in Figure 1(b), which guarantees maximal coverage without redundant actions. This particular sequence highlights implicit dependencies among various user actions. Violating these dependencies results in insufficient state coverage. For instance, actions such as viewing (❷), editing (❸), or bookmarking (❺) a blog should occur before deleting the blog (❹), since these actions may become




(a) A page from a travel blog content management system.


(b) An optimal click sequence.

**Figure 1: A motivating example.**

inaccessible once the blog is deleted. Similarly, actions related to a comment (❻, ❼, ❽, and ❾) need to be executed before deleting the blog associated with that comment (❹). Lastly, the logout action (❶) should always be performed last, as it terminates the session and could render all other actions inaccessible.

Among the 17 clickable actions in Figure 1, several perform equivalent or similar functions. For instance, the actions of viewing (❿, ⓮), editing (⓫, ⓯), bookmarking (⓭, ⓱), and deleting (⓬, ⓰) different blog instances are functionally equivalent to ❷, ❸, ❺, and ❹, respectively. These duplicated actions are excluded from the sequence in Figure 1(b) to ensure maximal performance.

## 2.3 Limitations of Existing Work

*Navigation Scheduling.* Previous studies [35] have demonstrated that Randomized Breadth-First Search (BFS) is the most effective navigation strategy for web crawling. By shuffling actions at each depth level in the discovery sequence, this method enables systematic traversal combined with stochasticity. However, such scheduling easily violates the dependencies among user actions, resulting in subpar coverage. For instance, with a probability of 1/17, the randomized BFS strategy would execute the logout action (❶) first, causing the entire crawling process to terminate prematurely. In fact, only 720 out of the total 362,880 possible (0.19%) sequences ensure maximal coverage without violating any dependencies, highlighting the necessity for dependency-aware scheduling.

*Deduplication.* The state-of-the-art crawlers [13, 17] employ both URL-based and DOM-based matching to identify duplicated user actions. URL-based methods identify duplicated actions by comparing the target URL strings; however, they cannot handle non-URL triggering actions, such as actions ❸-❾ in our example (Figure 1). Furthermore, when matching URL strings, existing URL-based methods often ignore the semantics of different parameters: they either compare the entire string or disregard all parameters altogether, which lead to under- or over-deduplication.

DOM-based methods differentiate actions by examining the DOM tree structures of the elements involved or the web pages generated. These methods require a carefully crafted threshold to recognize subtle differences between actions such as ❸ and ❹, while still identifying equivalent actions like ❸, ⓫ and ⓯. Additionally, the threshold is sensitive to variations in web page structures. As a result, the threshold configured for our example page is often not applicable to other pages, resulting in both under- and over-deduplication.

## 2.4 Our Approach

Figure 2 illustrates how we address existing limitations by modeling user actions as resource operations. Each action (❶ – ⓱) is represented as a triple $\langle r, op, t \rangle$, where $r$ denotes the resource (`post` or `comment`) that undergoes a specific operation $op$ (such as `view` or `bookmark`), and $t$ indicates the type of operation (`C`, `R`, `U`, `D`) as defined by the CRUD model. For example, action ❷ is represented as $\langle$`post`, `view`, `R`$\rangle$. Additionally, we introduce the unknown type `N` for operations whose type cannot be determined (such as `bookmark`). Finally, the block type `B` refers to operations that cause the crawler to be blocked or fail, such as `logout` (❶) in our example.

Existing limitations can then be addressed as follows:

- *Resource-operation-guided Deduplication.* As shown in Figure 2, actions that perform identical operations on the same type of resource are grouped together. For example, actions ❷, ❿, and ⓮ all correspond to the same triple $\langle$`post`, `view`, `R`$\rangle$, indicating that they each perform the `view` operation on the resource `post`. Actions within the same group are typically functionally equivalent and do not expose new states. Leveraging this insight, we perform efficient *deduplication* by processing only one representative action from each group of functionally equivalent actions.
- *Dependency-aware Navigation Scheduling.* We introduce dependencies between user actions based on the resources they operate on and their operation types. In Figure 2, each directed edge $u \rightarrow v$ denotes that operation $u$ must precede operation $v$. There are two types of dependencies: (1) *intra-resource dependency* reflects sequential constraints within the same resource, following the order of `C`, `R`, `U`, `N`, and `D`. We enforce this order because the side effect of a later operation type (e.g., `U` and `N`) may disable the actions of a previous operation type (e.g., `R`); and (2) *inter-resource dependency* arises from structural relationships among resources, such as `post` and `comment`. For instance, the edge $\langle$`comment`, `D`, `delete`$\rangle \rightarrow \langle$`post`, `D`, `delete`$\rangle$ indicates that a post should not be deleted before its associated comments are deleted. Additionally, block-type operations (❶) are executed last. Notably, these dependencies are conservatively approximated by treating all actions with the same operation type as equivalent within a resource.

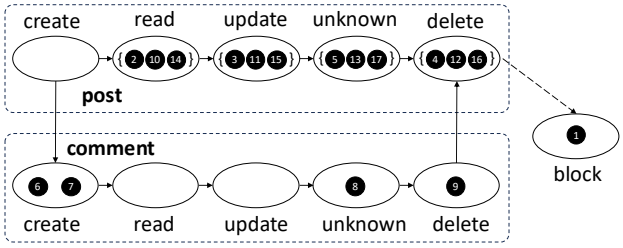



**Figure 2: A resource-guided modeling of user actions for our motivating example.**

*Comparison with YuraScanner.* YuraScanner [36] is a task-driven web crawler that employs an LLM to select the next action for a given task, modeling inter-task dependencies with a simple CRUD model. In contrast, Theseus explicitly infers dependencies between fine-grained actions using a resource-guided semantic model to maximize coverage. Specifically, Theseus explicitly reasons about inter-resource dependencies and avoids destructive or blocking actions until dependent operations are completed, whereas YuraScanner employs coarse-grained CRUD-based task ordering, treating each task as an indivisible action sequence. Notably, the coarse-grained CRUD-based task dependencies in YuraScanner fail to account for destructive tasks (e.g., ❹), inter-resource dependencies (e.g., ❾ and ❹), and repetitive operations (❷ and ❿), leading to subpar performance as evidenced in our empirical results.

## 2.5 Threat Model

In this paper, we focus on web application vulnerabilities that can be exploited by attackers through comprehensive crawling and state exploration. Our threat model assumes that an attacker has network access to the target web application and can systematically browse through its pages and functionalities using both manual navigation and automated crawling tools. The attacker can craft and inject malicious payloads into discovered input vectors to identify potential security vulnerabilities such as XSS or other web application vulnerabilities.

Regarding our approach, Theseus operates under common assumptions for web crawling: the target website is publicly accessible, authentication credentials are available for system login, and no anti-crawling mechanisms are in place. In addition, we do not consider adversarial websites designed to confuse LLMs and render our LLM-assisted semantic inference ineffective. These assumptions are typically valid in controlled penetration testing environments.

# 3 Theseus

We develop Theseus, a smart web crawler that leverages LLMs to identify the semantics of resource operations from user actions, serving as critical guidance for navigation scheduling and deduplication. As shown in Figure 3, Theseus augments existing crawlers with three primary modules that work in concert to enable systematic web crawling. After executing a target action, Theseus conducts **Resource operation inference** (Section 3.1) on newly discovered actions to analyze their semantics and deduce associated resources and operations. Next, for **Deduplication** (Section 3.2),

**Table 1: Types of frontend actions**

| **Action Types** | `get`, `form`, `event`, `iframe`, `ui_form`, `javascript` |
|---|---|

**Table 2: Operations and operation types**

| **Operation Types** | **Concrete Operations** |
|---|---|
| `Create` (C) | `add`, `upload`, `append`, `insert`, … |
| `Read` (R) | `view`, `download`, `search`, `list`, `filter`, … |
| `Update` (U) | `edit`, `rename`, `move`, `change`, `refresh`, … |
| `Delete` (D) | `remove`, `erase`, `clear`, `cancel`, … |
| `Block` (B) | `restrict`, `ban`, `suspend`, `disable`, `freeze`, … |
| `Unknown` (N) | `error`, `undefined`, `miscellaneous`, … |

**Table 3: Summary of LLM prompts used by Theseus. Full templates are available in the online appendix Figs. 5–7 .**

| Task | Goal | Template |
|---|---|---|
| Pre-execution analysis | Infer the target resource, concrete operation, and operation type before an action executes to guide scheduling and deduplication. | Fig. 5 |
| Post-execution refinement | Validate or correct the initial inference and determine execution success after observing runtime effects. | Fig. 6 |
| Inter-resource dependency inference | Determine parent-child relationship between resources to enforce navigation order. | Fig. 7 |

Theseus applies resource operation-guided deduplication to eliminate previously visited actions, thereby preventing redundant exploration. Finally, Theseus develops a dependency-aware **Navigation Scheduling** strategy (section 3.3) for effective exploration.

## 3.1 Resource Operation Inference

Following [13, 17], we categorize actions into six types: `get`, `form`, `event`, `iframe`, `ui_form`, and `javascript`. These action types, summarized in Table 1, capture the typical frontend interactions used to communicate with modern web applications, with `get` actions commonly responsible for loading target URLs. Throughout this paper, we use *action* $a$ to denote a frontend user interaction belonging to one of these types, and its action type is denoted as $T(a)$.

Given an action $a$ as input, the pre-execution analysis models it as a resource operation, denoted as a 3-tuple $\langle r, op, t \rangle$, where $r$ is the accessed resource, $op$ specifies the concrete operation performed (e.g., `view` or `logout`), and $t$ denotes the high-level *operation type* (e.g., `R` or `D`). This two-level abstraction enables both semantic grouping and fine-grained reasoning. Table 2 lists the operation types, including `C`, `R`, `U`, `D`, `B` and `N`, as well as some representative concrete operations for each type. In addition, post-execution refinement appends a boolean annotation indicating whether the execution was successful.

*3.1.1 Pre-execution Analysis.* As in standard practices [13], when a target action is successfully executed, Theseus extracts a set

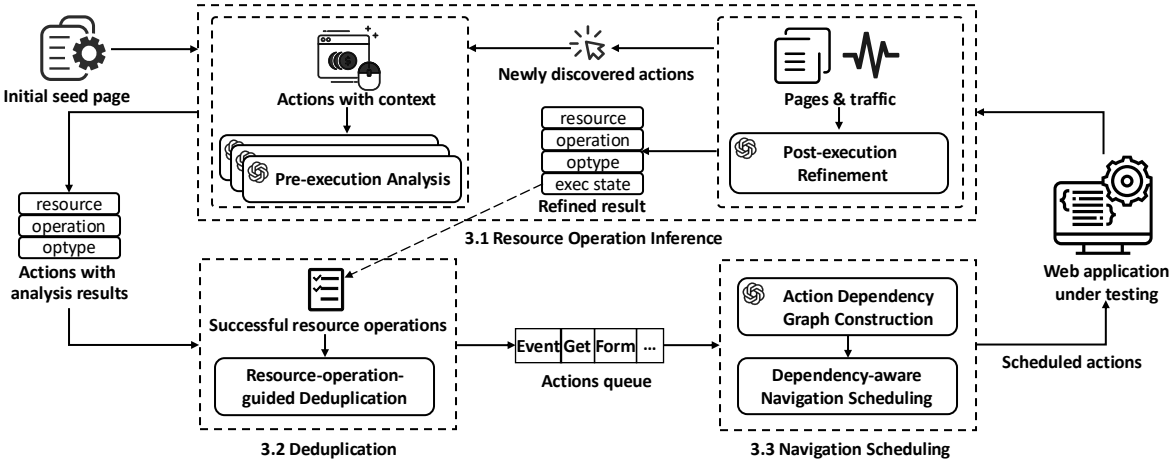



**Figure 3: Overview of Theseus.**

of clickable actions from the resulting web page. These extracted actions–if their target URLs or associated DOM structures do not exactly match those of previously visited actions–are considered potentially new and will be further analyzed by LLMs. To enhance LLMs' understanding of their semantics, we enrich each action with detailed contextual information.

Specifically, the context of an action encompasses three distinct hierarchical levels: (1) the *application level*, which comprises high-level natural language descriptions of the web application's functionality to provide comprehensive understanding of the application; (2) the *page level*, which includes metadata such as page titles and URLs that reflect the current application state; and (3) the *element level*, which consists of fine-grained DOM details such as text content, HTML tags, form types, labels, accessible names, and structural positions within the DOM hierarchy (e.g., parent-child and sibling relationships). Layered contexts of this form have been shown to help LLM agents ground action semantics in prior scanners [26, 36]; accordingly, we adopt the same structure to supply Theseus with evidence for semantic inference.

Hence, after gathering the context of action $a$, we construct prompts to invoke the LLM for pre-execution analysis; the full prompt and a corresponding response are available in the online appendix[2] Fig. 5 . Each prompt consists of a `system prompt`, which defines the LLM's role, analytical objectives, and expected response format, as well as a `user prompt`, which embeds the context of $a$. Additionally, we include representative few-shot examples to guide the LLM's reasoning process. The LLM's response specifies the resource $r$, the concrete operation $op$, and the high-level *operation type* of $op$.

*3.1.2 Post-execution Refinement.* After executing action $a$, Theseus performs a post-execution refinement step to determine whether $a$ was successfully executed, i.e., whether the intended resource operation takes effect. During this step, the action $a$ is reanalyzed in its pre-execution context with additional runtime feedback, such as changes in the DOM structure of web pages before and after execution, as well as network traffic details like HTTP request and response payloads, since HTTP responses alone are insufficient to reliably indicate whether a resource operation has truly taken effect. This additional runtime information enables LLMs to perform a deeper semantic interpretation of the intention behind $a$, thereby refining the pre-execution analysis results of $a$.

With the complete context—including pre- and post-execution page states and network traffic—Theseus invokes the LLM to perform post-execution refinement (row 3 of Table 3), updating or correcting the initial operation inference as needed. This step helps resolve ambiguities, confirm the effects of the action, and improve the semantic analysis based on actual execution outcomes.

## 3.2 Resource-operation-guided Deduplication

Theseus identifies and eliminates duplicated actions by examining their corresponding resource operations–two actions are considered *functionally equivalent* if they share the same action type with an identical triple $\langle r, op, t\rangle$ under resource operation inference. These actions perform the same operation on the same or different instances of the resource $r$ and generally will not uncover any new states, since they are likely to explore the same functionality and thus execute the same code.

Hence, during crawling, Theseus maintains a set $S$ of previously processed resource operations. After pre-execution analysis of a newly discovered resource operation $\langle r, op, t\rangle$ for action $a$, Theseus checks if there exists a tuple $\langle\langle r', op', t'\rangle, T(a)\rangle \in S$ such that $\langle r', op', t'\rangle$ is *equal* to $\langle r, op, t\rangle$. Recall that $T(a)$ denotes the action type of $a$. If such a tuple exists, $a$ will be discarded.

Some resource operations require multiple coordinated steps to complete. For example, editing a post may involve navigating to an edit page (via a `get` or `event` action), modifying a form, and submitting it. In some cases, an LLM may label each step as the same $\langle$`post, edit, U`$\rangle$ operation. Therefore, we define actions as equivalent only if they share the same action type in Table 1, and we retain multiple actions with identical inferred resource-operation semantics when their action types differ.

[2]https://github.com/Theseus-c/Theseus/appendix.pdf.

### 3.3 Dependency-aware Navigation Scheduling

This subsection illustrates in detail our resource-dependency-aware navigation scheduling strategy, which harnesses an on-the-fly action dependency graph for effective exploration. The action dependency graph $G$ serves as the backbone for our navigation scheduling. Essentially, it encodes the dependence between actions based on their corresponding resource operations. Recall that an action is modeled as a tuple $\langle r, op, t\rangle$, as described in Section 3.1.

We adopt a cluster-based abstraction in building $G$, where actions are grouped by their corresponding resources and operation types. In the action dependency graph $G = (V, E)$, each node $v \in V$ is a cluster $\{\langle r, \_, t\rangle\}$, denoting the $t$ type operations on resource $r$. For instance, in our motivating example in Figure 1, action ❻ and action ❼ are grouped in the same cluster ⟨`comment, _, create`⟩ (Figure 2). This abstraction not only reduces the graph's size and complexity, but—more importantly—provides generalization: even if specific actions have not yet been encountered, their corresponding clusters act as placeholders to ensure correct execution order.

Each directed edge $u \rightarrow v \in E$ encodes a dependency relation: actions in cluster $u$ must be considered before executing any action in cluster $v$. There are two primary forms of dependencies, namely intra-resource dependency and inter-resource dependency.

*Intra-resource dependency.* For a given resource $r$, we enforce a canonical ordering of operations based on their types: `C` → `R` → `U` → `N` → `D` → `B`. Intuitively, creation must precede all other operations on the same resource. `R` operations are prioritized over `U` operations, since updating a resource may introduce side effects that could interfere with subsequent reads. In some cases, the LLM may be unable to infer the semantics of an operation type (e.g., `bookmark` operations ❺, ⓭, and ⓱ in Figure 2). To ensure safety, we conservatively place these `N` operations between `U` and `D`-allowing them to execute before deletion while minimizing interference with other known operations.

Block-type actions (e.g., user logout) are deferred and executed only after all other actions have been processed. For example, in Figure 2, the user `logout` action (❶) is scheduled after all other operations. When Theseus cannot determine the resource associated with an action, it conceptually adds the action to $G$ as an independent node with no assumed dependencies. As discussed in Section 2.2, this ordering is conservative, but it does not compromise the effectiveness of Theseus's navigation scheduling strategy in our experiments.

*Inter-resource dependency.* Dependencies may also exist between different resources due to their hierarchical relationships. For instance, a `comment` is associated with a `post`, and deleting the post disables all actions on the associated comments. We therefore enforce a dependency such as ⟨`comment`, _, `D`⟩ → ⟨`post`, _, `D`⟩.

Inferring such hierarchies requires semantic reasoning, a task well-suited to LLMs. To minimize unnecessary computation, we invoke this reasoning task only as needed: specifically, during exploration, when a `D` action is encountered, we prompt the LLM to analyze potential dependencies by pairing the target resource (to be deleted) with each other resource on which some operations have yet to be executed. As such, the number of resource relationship reasoning tasks remains small.

For each pair, the LLM determines whether the second resource depends on the target resource. If such hierarchical dependencies are identified, the graph $G$ is updated accordingly. To ensure high accuracy, we supply rich semantic context for both resources in each pair, leveraging information extracted during the semantic analysis process described in Section 3.1. An example prompt and the LLM's corresponding output are available in the online appendix Fig. 7 .

By maintaining hierarchical relationships between distinct resources, the action dependency graph $G$ remains a directed acyclic graph, facilitating a topological ordering for navigation scheduling: an action in cluster $v$ is scheduled only if all actions in the predecessors of $v$ have been processed. In rare cases, inter-resource dependency relationships may form a cycle, either because the LLM incorrectly treats the same resource entity as different resources, or because it mistakenly introduces a dependency between distinct resources. To resolve such cycles, Theseus prompts the LLM to break the cycle by either merging inter-dependent resources into a single canonical resource, or removing likely invalid edges.

### 3.4 The Overall Algorithm

Algorithm 1 gives a detailed description of our overall algorithm. The algorithm operates over the dynamically constructed dependency graph $G = (V, E)$, as detailed in Section 3.3. It iteratively executes or deduplicates all actions in $G$ until no actions remain. During this process, $G$ is dynamically updated with newly discovered actions, and all executed actions are recorded in $S$.

*Initialization.* For each resource operation $\langle r, op, t\rangle$, a failure counter *FailCnt* is initialized to 0 by default (line 2). In addition, a threshold parameter $MaxCnt = 10$ is introduced to specify the maximum number of attempts before skipping a resource operation. The algorithm starts with processing the web page $P$ retrieved from the seed URL, and $G$ is initialized with all actions in $P$ (lines 3-5).

*Picking an Action.* At each iteration, Theseus invokes the procedure DepGuidedNavig to select the next target action to execute (line 7). In DepGuidedNavig (lines 24-26), an action is selected only if all its dependent actions have been completed–following the topological ordering in $G$. If multiple such actions exist in the same cluster, we select the action with the fewest attempts.

*Executing an Action.* The selected action $a$ is skipped if the number of attempts for its corresponding resource operation has reached the predefined threshold *MaxCnt* (lines 8-10). Subsequently, all equivalent actions of $a$ are removed from $G$ and excluded from further execution (line 10), and the failed resource operation is recorded in $S$ to prevent future attempts. After executing $a$, the execution results are analyzed by PostExecRefine(line 13), yielding a refined tuple $\langle r', op', t', succ\rangle$, where *succ* indicates whether the execution is successful. The successfully executed resource operation $\langle r', op', t'\rangle$ is then recorded in $S$ (line 15). Next, each newly discovered action is first analyzed by PreExecAnalysis and then deduplicated according to its resulting resource operation $\langle r, op, t\rangle$; the non-duplicated actions are subsequently added to $G$ for future processing (lines 17-19). If the execution fails, we increment $FailCnt[\langle\langle r', op', t'\rangle, T(a)\rangle]$ by one and retain the other actions for potential future attempts (lines 21-22).

**Algorithm 1:** Overall algorithm of Theseus

**Input:** Start page $P$ retrieved from the seed URL, failure threshold *MaxCnt*
**Output:** A set of completed resource operations $S$

$S \leftarrow \emptyset$
Assume *FailCnt* has an initial value of 0
**for** *each action a in P* **do**
  $\langle r, op, t\rangle$ = PreExecAnalysis($a$)
  Add $\langle r, op, t\rangle$ to its cluster $\langle r, -, t\rangle$ in $G$
**while** *there exists actions in G* **do**
  $a$, $\langle r, op, t\rangle$ = DepGuidedNavig() //Navigation
  **if** $FailCnt[\langle\langle r, op, t\rangle, T(a)\rangle] \geq MaxCnt$ **then**
    $S \leftarrow S \cup \{\langle\langle r, op, t\rangle, T(a)\rangle\}$
    Remove functionally equivalent actions to $a$ from $G$
    **continue**
  Execute action $a$
  $\langle r', op', t', succ\rangle$ = PostExecRefine($a$)
  **if** *succ* **then**
    $S \leftarrow S \cup \{\langle\langle r', op', t'\rangle, T(a)\rangle\}$
    **for** *each newly discovered action* $a_n$ **do**
      $\langle r_n, op_n, t_n\rangle$ = PreExecAnalysis($a_n$)
      **if** $\langle\langle r_n, op_n, t_n\rangle, T(a_n)\rangle \notin S$ **then**
        add $\langle r_n, op_n, t_n\rangle$ to its cluster $\langle r_n, -, t_n\rangle$ in $G$
    Remove all functionally equivalent actions of $a$ in $G$
  **else**
    $FailCnt[\langle\langle r', op', t'\rangle, T(a)\rangle] \leftarrow FailCnt[\langle\langle r', op', t'\rangle, T(a)\rangle] + 1$
**return** $S$
**Function** `DepGuidedNavig()`:
  Select cluster $v \neq \emptyset$ in $G$ with no non-empty (transitive) predecessors
  Remove action $a = \langle r, op, t\rangle$ with minimal $FailCnt[\langle\langle r, op, t\rangle, T(a)\rangle]$ from $v$
  **return** $a$, $\langle r, op, t\rangle$

# 4 Implementation

We have developed Theseus based on BlackWidow [13] with 4,448 lines of Python code. As a result, Theseus inherits most of BlackWidow's core capabilities, such as DOM traversal and JavaScript event handling. To support the pre-execution analysis (Section 3.1.1) and post-execution refinement (Section 3.1.2), we replaced the original Selenium module with Selenium-Wire [39], which extends Selenium to enable logging of HTTP request and response messages during the crawling process. Additionally, we substituted BlackWidow's navigation and deduplication modules with our own custom implementations, detailed in Section 3.3 and Section 3.2.

For LLM-assisted analysis, we integrated the OpenAI SDK [27] to invoke language models. Our implementation provides flexible model selection through configurable API keys, base URLs, and model names, enabling seamless support for various commercial and open-source LLMs. Furthermore, we implemented request batching to optimize token consumption. Next, we elaborate on two critical optimizations in our implementation.

## 4.1 Parallelized Pre-execution Analysis

Our approach requires LLM analysis for each potential new action. After executing a target action, the crawler may encounter numerous new actions on the resulting page, each of which needs to be analyzed by the LLM during pre-execution analysis. This may incur significant overhead – pre-execution analysis averages 5 seconds, but can take several minutes in worst-case scenarios. Additional delays may also occur due to tokens per minute (TPM) rate limitations. As a result, after executing an action, the crawler may need to wait minutes for the LLM analysis results on new actions, significantly impacting overall efficiency.

To address this limitation, we design Theseus so that pre-execution analysis runs in parallel with the rest of the crawler. Note that pre-execution analysis is also the most time-consuming module, as it is invoked for each potential new action, whereas other modules only invoke LLM as needed, and the number of such LLM invocations is much smaller.

Hence, Theseus utilizes two processes that run in parallel: one is the main crawler process, and the other is dedicated to pre-execution analysis. After executing an action, the crawler process schedules and executes queued actions without waiting. It receives results from the concurrently running pre-execution analysis process in batches, using these results to guide future deduplication and navigation scheduling. Meanwhile, the pre-execution analysis process employs Python's asynchronous capabilities to issue multiple LLM invocations concurrently. This process collects multiple pre-execution prompts into a batch within the LLM's TPM limit, then simultaneously invokes the LLM for all prompts in the batch, thereby enabling concurrent LLM analyses. After firing a batch of prompts, to comply with the LLM's TPM constraints, the process waits for one minute before firing the next batch.

## 4.2 Semantic-aware URL Deduplication

To further reduce the number of LLM invocations during pre-execution analysis, we apply a lightweight URL-based approach that efficiently filters out redundant URL-triggering actions. Our approach significantly outperforms existing URL-based deduplication strategies, by understanding the semantics of different parameters in URLs and recognizing *semantically important* parameters whose modification impacts application logic. For instance, in the URL http://example.com/index.php?action=show&id=1, the parameters `action` and `id` serve fundamentally different purposes-the `action` parameter is *semantically important*, as its modification alters application logic and views, while changing `id` typically only affects the specific data being displayed. However, existing strategies fail to consider the semantic significance of distinct query parameters, often resulting in over- or under-deduplication.

For each URL-triggering action encountered during crawling, we extract all parameters–including both query parameters (e.g., `?id=3`) and path segment parameters (e.g., `/project/3/show`)–paired with their corresponding values from the target URL, in the form `(name, value)`. Unnamed path segments are handled by assigning synthetic names. For example, `/project/3/show` yields

**Table 4: Overview of Evaluated Applications**

| Application | Version | Stars | Category |
|---|---|---|---|
| Leantime | 2.1.7 | 8.7k | PMS |
| Librenms | 22.1.0 | 4.5k | Network monitor |
| Microweber | 1.2.11 | 3.4k | CMS |
| Typo3 | 9.5.11 | 1.2k | CMS |
| Concretecms | 9.0.2 | 817 | CMS |
| Oscommerce | 4.12.56860 | 48 | eCommerce |
| Mautic | 3.3.4 | 8.9k | Marketing |
| Moodle | 4.0.0 | 6.7k | LMS |
| LimeSurvey | 5.6.4 | 3.4k | Polls |
| ITop | 3.0.0 | 1k | IT |
| Kanboard | 1.2.44 | 9.3k | PMS |
| Dokuwiki | 2024-02-06b | 4.5k | Wiki |
| Humhub | 1.17.1 | 6.6k | Social Network |
| Drupal | 11.1.6 | 4.2k | CMS |
| Collabtive | 3.2 | 215 | PMS |
| Opencart | 4.1.0.3 | 8k | eCommerce |
| Phpbb | 3.3.15 | 2k | Forum |
| Smf | 2.1.4 | 683 | Forum |
| Dolibarr | 22.0.3 | 6.7k | CRM |
| Wordpress | 6.8.3 | 20.7k | CMS |

the pair `(PATH_PARAM_1, 3)`, and the URL http://example.com/project/3/show is transformed into the template http://example.com/project/PATH_PARAM_1/show. Two URLs are considered equivalent if they have the same target address with identical values for semantically important parameters.

How do we know which URL parameter is semantically important? Again, we leverage LLMs to analyze the semantics of a URL parameter as needed. Note that here the LLM is queried only for newly encountered URL parameters with distinct values, so the number of LLM invocations is small. The LLM prompt and corresponding response for inferring URL-parameter semantics are available in the online appendix Fig. 8 .

## 5 Evaluation

Our evaluation aims to answer the following research questions:

- **RQ1**: What is the precision of Theseus in inferring resource operations from user actions?
- **RQ2**: How effective is Theseus in improving coverage compared to the state-of-the-art crawlers?
- **RQ3**: How much does Theseus enhance existing vulnerability scanners' ability to detect both known and previously unknown vulnerabilities?
- **RQ4**: What are the contributions of each key system component (ablation study)?

### 5.1 Experiment Setup

**Evaluation Benchmarks.** To evaluate the effectiveness of Theseus, we constructed a benchmark dataset comprising 20 open-source PHP web applications, as summarized in Table 4. Following prior work [13, 17, 36], we selected PHP-based applications which allow code coverage collection through Xdebug [33]. In this paper, we focus on detecting XSS vulnerabilities [28], as in [13]. Our benchmark consists of two categories: ten applications with known XSS vulnerabilities (rows 2–11 of Table 4) and ten up-to-date applications previously evaluated by existing tools (rows 12–21). The first group emphasizes widely used applications with known vulnerabilities, while the second is drawn from applications evaluated by prior crawlers; detailed selection criteria are available in the online appendix A.2 .

**Baseline Crawlers.** We compared Theseus against three state-of-the-art open-source crawlers: BlackWidow [13], EvoCrawl [17] and YuraScanner [36].

**LLM Choice.** We compared four representative LLMs on Leantime in terms of semantic inference effectiveness, latency, cost, and throughput. Detailed results are available in the online appendix Table 12 . Qwen2.5-32B-Instruct [32] achieved the best balance: its fast response time (2.4s per request) and high throughput (1,000,000 TPM) enable Theseus to perform more actions within the same time budget, facilitating broader exploration. We therefore adopt Qwen2.5 as the default model.

**Evaluation Metrics.** For coverage evaluation, following standard practices [13, 17], we used line-level code coverage as our evaluation metric. For vulnerability detection, we assessed the tools' effectiveness in identifying both known and new vulnerabilities. Since we employed dynamic testing, which directly executes exploits against real vulnerabilities, there were no false positives, and only recall was considered.

**Ground Truth.** We established two distinct ground truth datasets for our experimental evaluation: (1) For coverage evaluation, we combined the code coverage achieved by Theseus and the baseline crawlers to form the ground truth; (2) For vulnerability detection, we collected 101 documented XSS vulnerabilities in the applications listed in rows 2–11 of Table 4 and attempted to trigger them using BlackWidow's XSS payloads. We excluded 22 vulnerabilities that could not be reproduced in our deployed versions, leaving 79 reproducible known XSS vulnerabilities. Collection and exclusion details are available in the online appendix A.2.3 .

**Configurations.** All experiments were conducted on a dedicated server equipped with a 64-core Intel Xeon E7-4809 v3 @ 2.00GHz CPU, 1 TB of memory, and running Clear Linux OS version 43410. Each experiment run was conducted in a clean and reproducible dockerized environment. For each target application, we first built a Docker image using the default installation settings, including bundled sample data when available (e.g., Oscommerce), and manually created an administrator account. Given the login URL, administrator credentials, the application name and purpose, and an LLM API key as input, the crawler automatically logged in and began crawling. For each application, the maximum crawling duration was set to eight hours, consistent with the time limits established in prior work [13].

### 5.2 RQ1: LLM-assisted Semantic Inference

Theseus leverages large language models (LLMs) to perform semantic inference over user actions, enabling the identification of resource operations, action execution states, inter-resource dependencies, and semantically important URL parameters. The evaluation results of this LLM-assisted semantic inference are summarized in Table 5.

*Evaluation Methodology.* During the experiment, we logged all prompts issued to the LLMs together with their corresponding inference results. To assess inference correctness, we manually validated whether each LLM response correctly fulfilled its intended semantic inference task under the given prompt. Specifically, two

**Table 5: Results of LLM-assisted semantic inference**

| Application | # RO[1] | | # ES[2] | | # IRD[3] | | # SIP[4] | |
|---|---|---|---|---|---|---|---|---|
| | TP[5] | FP[6] | TP[5] | FP[6] | TP[5] | FP[6] | TP[5] | FP[6] |
| Leantime | 91 | 21 | 36 | 5 | 57 | 24 | 26 | 5 |
| Librenms | 396 | 99 | 79 | 11 | 11 | 6 | 83 | 13 |
| Microweber | 198 | 53 | 67 | 12 | 40 | 15 | 63 | 12 |
| Typo3 | 756 | 166 | 23 | 3 | 30 | 12 | 15 | 4 |
| Concretecms | 435 | 138 | 38 | 8 | 29 | 17 | 73 | 12 |
| Oscommerce | 211 | 79 | 44 | 4 | 6 | 2 | 33 | 7 |
| Mautic | 176 | 41 | 27 | 6 | 125 | 44 | 13 | 2 |
| Moodle | 301 | 118 | 41 | 10 | 89 | 37 | 28 | 5 |
| LimeSurvey | 438 | 137 | 39 | 11 | 99 | 44 | 37 | 10 |
| ITop | 419 | 117 | 27 | 7 | 150 | 67 | 9 | 2 |
| Kanboard | 540 | 88 | 130 | 22 | 21 | 15 | 30 | 4 |
| Dokuwiki | 73 | 15 | 42 | 9 | 16 | 6 | 37 | 9 |
| Humhub | 592 | 89 | 18 | 4 | 3 | 1 | 53 | 15 |
| Drupal | 932 | 248 | 85 | 16 | 66 | 22 | 208 | 17 |
| Collabtive | 58 | 9 | 33 | 2 | 13 | 5 | 38 | 7 |
| Opencart | 334 | 74 | 112 | 29 | 14 | 6 | 68 | 8 |
| Phpbb | 420 | 141 | 121 | 36 | 26 | 11 | 297 | 45 |
| Smf | 374 | 51 | 101 | 39 | 76 | 27 | 176 | 29 |
| Dolibarr | 793 | 188 | 70 | 12 | 168 | 66 | 29 | 6 |
| Wordpress | 430 | 111 | 57 | 19 | 159 | 57 | 73 | 8 |
| **Total** | 7,967 | 1,983 | 1,190 | 265 | 1,198 | 484 | 1,389 | 220 |
| **Precision** | 80.1% | | 81.8% | | 71.1% | | 86.3% | |

[1]**RO**: Resource Operation; [2]**ES**: Execution State; [3]**IRD**: Inter-resource Dependency; [4]**SIP**: Semantically Important Parameter; [5]**TP**: True Positives; [6]**FP**: False Positives.

authors independently annotated the LLM outputs by comparing the inferred semantics with the prompt context, and a third author resolved any disagreements. This manual validation process required approximately three person-months.

Due to the large number of automated analysis results, following prior work that validates such results via sampling [22, 24], we randomly sampled 10.0% of the inferred resource operation (#RO), execution state (#ES), and inter-resource dependency (#IRD) results for validation.

*Overall Results.* As shown in Table 5, Theseus achieves precision rates of 80.1%, 81.8%, 71.1%, and 86.3% for resource operation inference (#RO), action execution state prediction (#ES), inter-resource dependency analysis (#IRD), and semantically important parameter classification (#SIP), respectively. These results indicate that LLMs can effectively capture high-level semantic intents embedded in user actions and application contexts, despite operating in a fully black-box setting.

*False Positive Analysis.* We further conducted a fine-grained analysis of false positives (FPs) to understand their root causes and their impact on the overall system behavior.

For resource operation inference (#RO), each LLM output consists of a triple $\langle r, op, t \rangle$, and any error in the triple is considered a false positive. Among the 1,983 FPs, 35.3% are caused by incorrect resource identification, while the remaining errors are primarily due to misinterpretation of operation semantics, including incorrect operation inference and misclassification of operation types (64.7% in total). This suggests that errors are more likely to arise from fine-grained operation semantics than from coarse-grained resource recognition; for instance, operations triggered by a form submission where the action label is `Save` may be inferred as `Update` instead of the correct `Create` type.

For action execution state prediction (#ES), the LLM outputs a Boolean value indicating whether an action was executed successfully. Among the 265 FPs, 29.8% correspond to failed actions that were incorrectly inferred as successful, mainly due to misleading context. For instance, the LLM may misclassify an HTTP 200 response as successful even though the intended action (adding a picture) does not take effect. Such errors may prevent subsequent semantically equivalent actions from being executed, potentially resulting in missed application states. The remaining 70.2% of FPs correspond to successfully executed actions that were misclassified as failures, primarily because the target URL did not load within a predefined time limit (0.5 s by default) or because the LLM prompt was truncated due to token restrictions. These errors lead to redundant execution of similar actions and thus affect crawling efficiency rather than semantic coverage.

For inter-resource dependency analysis (#IRD), the LLM predicts whether a parent–child relationship exists between two resources. Among the 484 FPs, 75.3% arise from incorrectly inferred dependencies where no actual parent–child relationship exists. These errors are conservative in nature, as deleting either resource does not affect the operability of the other. In contrast, the remaining 24.7% correspond to missed dependencies, which may impact the scheduling strategy by causing premature deletion of parent resources and consequently blocking operations on dependent child resources. These false positives may be caused by misleading or insufficient context, or by inherent LLM limitations such as hallucination.

For semantically important parameter classification (#SIP), the LLM predicts whether a URL parameter contributes to the semantic identity of a resource and therefore should be preserved during deduplication. Among the 220 FPs, 22.2% involve important parameters being incorrectly predicted as unimportant, which may cause distinct operations to be merged and thus potentially result in missed paths. The remaining 77.8% involve non-important parameters being misclassified as important, most commonly filtering parameters such as `category` or `status`. These conservative errors primarily reduce deduplication effectiveness and crawling efficiency, while having only limited impact on semantic coverage.

## 5.3 RQ2: Code Coverage

Figure 4 compares the code coverage achieved by Theseus with three state-of-the-art crawlers: BlackWidow, EvoCrawl, and YuraScanner, across twenty real-world web applications. Overall, Theseus consistently achieves higher code coverage than all baselines. On average, Theseus improves code coverage by 93.5% over BlackWidow, 72.1% over EvoCrawl, and 96.8% over YuraScanner. Moreover, Theseus achieves a higher amount of unique code coverage than the combined baselines, as summarized in Table 9 in the online appendix. Relative to the union of lines reached by all four crawlers, Theseus covers 87.1% on average, compared with 57.3% for BlackWidow, 62.6% for EvoCrawl, and 53.7% for YuraScanner. We use this observed union rather than total source lines because deployed applications can contain installation, obsolete, or otherwise unreachable code. Detailed per-application results are available in the online appendix Table 11 .

The substantial coverage improvements of Theseus are attributable to its more effective scheduling and deduplication. First,

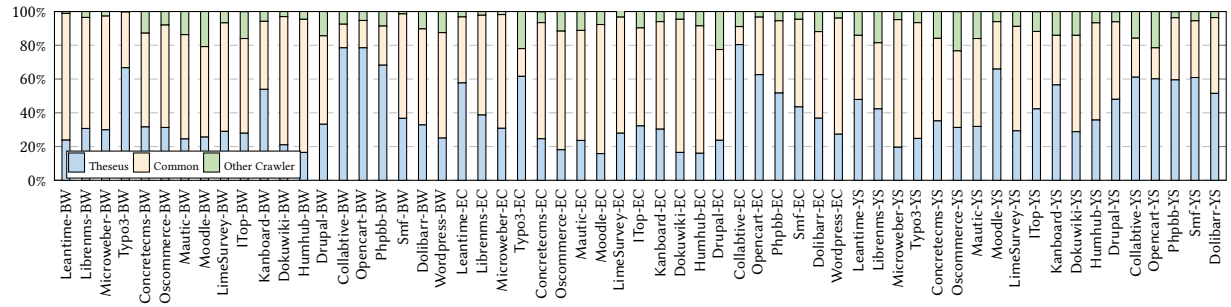



**Figure 4: Code coverage comparison between Theseus and three baseline crawlers across twenty applications. Suffixes "-BW", "-EC", and "-YS" denote comparisons against BlackWidow, EvoCrawl, and YuraScanner, respectively. Each stacked bar shows Theseus unique coverage (blue), common coverage (beige), and baseline unique coverage (green). YS was unavailable on Wordpress, so only BW/EC comparisons are shown. Detailed values are available in the online appendix Table 11 .**

Theseus's dependency-aware navigation scheduling enables actions to be executed in a semantically valid order, preventing state corruption and avoiding the premature execution of blocking actions. For instance, in OpenCart, BlackWidow executes a `logout` action early, which disrupts subsequent interactions, whereas Theseus preserves the session state and continues exploration. In HumHub, YuraScanner schedules the `delete space` action before the `delete space message` action, rendering subsequent `message`-related operations infeasible. Second, Theseus employs resource-operation-guided deduplication to identify semantically equivalent actions and reduce redundant executions. This allows the crawler to allocate more execution budget to unexplored actions and states.

Despite these improvements, Theseus does not guarantee complete code coverage. A small fraction of code lines are exclusively covered by baseline crawlers through randomized exploration. Furthermore, Theseus adopts BlackWidow's form-handling strategy by filling input fields with fixed type-based values, which may violate input constraints and lead to unsuccessful executions, while EvoCrawl partially mitigates this issue by leaving optional fields empty. How to effectively handle complex web forms remains an open challenge and is beyond the scope of this work.

We also observe that YuraScanner uniquely covers a small portion of code that is not exercised by Theseus. This is mainly because YuraScanner is explicitly designed to execute individual tasks in depth. By focusing on completing a specific task and exploring its execution path more thoroughly, YuraScanner can sometimes reach deeper code paths associated with that task, resulting in unique coverage. In contrast, Theseus prioritizes maximizing overall code coverage across the entire application, which may limit the depth of exploration for certain individual tasks.

## 5.4 RQ3: Vulnerability Detection

We investigate the effectiveness of Theseus and the three baseline crawlers in assisting scanners to detect both known and new vulnerabilities. The comparison results are shown in Table 6. Since Theseus significantly improves code coverage, it covers a larger portion of the application's potential attack surface and helps to identify considerably more vulnerabilities.

**Table 6: XSS vulnerabilities detected by different crawlers**

**(a) Known vulnerabilities**

| Application | # BW[2] R/S[1] | # EC[3] R/S[1] | # YS[4] R/S[1] | # TH[5] R/S[1] | # Known[6] R/S[1] | # Uniq.[7] |
|---|---|---|---|---|---|---|
| Leantime | 2/3 | 2/5 | 0/0 | 2/10 | 2/12 | 12 |
| Librenms | 2/3 | 1/4 | 1/3 | 2/7 | 2/11 | 10 |
| Microweber | 1/0 | 1/0 | 1/0 | 1/1 | 3/1 | 2 |
| Typo3 | 0/0 | 0/0 | 0/0 | 0/1 | 0/2 | 1 |
| Concretecms | 0/2 | 0/2 | 0/1 | 0/4 | 0/5 | 4 |
| Oscommerce | 1/3 | 2/4 | 2/8 | 2/10 | 4/15 | 16 |
| Mautic | 0/3 | 0/2 | 0/2 | 1/3 | 1/5 | 4 |
| Moodle | 1/2 | 0/3 | 1/2 | 1/4 | 1/7 | 5 |
| LimeSurvey | 1/1 | 2/1 | 1/0 | 2/1 | 3/1 | 4 |
| ITop | 0/1 | 1/1 | 1/0 | 1/1 | 2/2 | 2 |
| **Total** | 8/18 | 9/22 | 7/16 | 12/42 | 18/61 | 60 |

**(b) New vulnerabilities**

| Application | # BW[2] R/S[1] | # EC[3] R/S[1] | # YS[4] R/S[1] | # TH[5] R/S[1] | # Uniq.[7] |
|---|---|---|---|---|---|
| Leantime | 1/1 | 2/1 | 0/0 | 8/4 | 12 |
| Librenms | 0/0 | 0/0 | 0/0 | 0/1 | 1 |
| Microweber | 0/0 | 0/0 | 0/0 | 0/0 | 0 |
| Typo3 | 0/0 | 0/0 | 0/0 | 0/0 | 0 |
| Concretecms | 0/0 | 0/0 | 0/0 | 0/0 | 0 |
| Oscommerce | 1/1 | 0/3 | 1/2 | 1/3 | 7 |
| Mautic | 0/0 | 0/0 | 0/0 | 0/0 | 0 |
| Moodle | 0/0 | 0/0 | 0/0 | 0/0 | 0 |
| LimeSurvey | 0/0 | 0/0 | 0/0 | 0/0 | 0 |
| ITop | 0/0 | 0/0 | 0/0 | 0/0 | 0 |
| Kanboard | 0/0 | 0/1 | 0/0 | 1/0 | 2 |
| Dokuwiki | 0/0 | 0/0 | 0/0 | 0/0 | 0 |
| Humhub | 0/0 | 0/0 | 0/0 | 0/0 | 0 |
| Drupal | 0/0 | 0/0 | 0/0 | 0/0 | 0 |
| Collabtive | 0/1 | 0/0 | 0/2 | 1/3 | 4 |
| Opencart | 0/0 | 0/0 | 0/1 | 0/2 | 3 |
| Phpbb | 0/0 | 0/0 | 0/0 | 0/0 | 0 |
| Smf | 0/0 | 0/0 | 0/0 | 2/3 | 5 |
| Dolibarr | 0/0 | 0/0 | 0/0 | 0/2 | 2 |
| Wordpress | 0/0 | 0/0 | -/- | 0/0 | 0 |
| **Total** | 2/3 | 2/5 | 1/5 | 13/18 | 36 |

[1]**R/S**: reflected/stored XSS vulnerabilities; [2]**BW**: BlackWidow; [3]**EC**: EvoCrawl; [4]**YS**: YuraScanner; [5]**TH**: Theseus; [6]**Known**: manually collected vulnerabilities; [7]**Uniq.**: total unique vulnerabilities found by all crawlers.

*5.4.1 Known Vulnerabilities.* For previously known vulnerabilities, as shown in Table 6(a), we manually collected a total of 79 known XSS vulnerabilities–18 reflected and 61 stored–across ten vulnerable versions of web applications. The four crawlers collectively helped

to uncover 60 out of these 79 cases, achieving a combined detection rate of 75.9%.

Among the four crawlers, Theseus achieved the best performance, detecting 54 known vulnerabilities, compared to 26 by BlackWidow, 31 by EvoCrawl and 23 by YuraScanner, respectively. This corresponds to detection rates of 68.4% for Theseus, 32.9% for BlackWidow, 39.2% for EvoCrawl and 29.1% for YuraScanner. Notably, Theseus discovered 15 known vulnerabilities that were missed by all three baseline crawlers.

*5.4.2 New Vulnerabilities.* As demonstrated in Table 6(b), the four crawlers collectively identified 36 unique new vulnerabilities. Among them, Theseus exhibited strong discovery capability by identifying 31 new vulnerabilities. In comparison, BlackWidow, EvoCrawl, and YuraScanner detected only 5, 7, and 6 vulnerabilities, respectively. Notably, Theseus discovered 21 new vulnerabilities that were missed by all baseline crawlers.

**Responsible Disclosure.** We have responsibly disclosed all 36 newly discovered vulnerabilities to the respective project maintainers. To date, 22 vulnerabilities have been confirmed and patched across Leantime (12), Librenms (1), Kanboard (2), SMF (5), and Dolibarr (2), resulting in 12 CVE assignments. The remaining vulnerabilities are currently under review. We follow the industry-standard 90-day responsible disclosure timeline [16].

*5.4.3 Additional True Positives.* Theseus uncovered 36 additional true positives—15 known and 21 new—that are missed by all baseline crawlers. We next analyze how Theseus uncovers these additional vulnerabilities in practice.

**Inter-resource Dependency.** Baseline crawlers missed 19 vulnerabilities due to inter-resource dependencies. For instance, in Leantime, 8 vulnerabilities were triggered by operations on child resources that depend on a parent resource, such as a `Project Attachment File` belonging to a `Project`. Without recognizing this dependency, BlackWidow and EvoCrawl deleted the `Project` before executing operations on its attachment files, rendering the vulnerable actions unreachable. In contrast, Theseus preserved these dependencies in its scheduling order and successfully exposed these vulnerabilities.

**Blocking Operations.** There are 6 false negatives caused by baseline crawlers executing blocking operations too early. In OpenCart, BlackWidow executed the `logout` operation at an early stage, which prevented further crawling and caused two vulnerabilities to be missed. Similarly, in Leantime, EvoCrawl and YuraScanner prematurely terminated exploration by executing `deluser`, thereby deleting the only user account in the system. By identifying such blocking operations, Theseus postpones their execution and avoids early aborts, enabling the discovery of these vulnerabilities.

**Deduplication.** Theseus reported 11 additional vulnerabilities because of effective deduplication. In Typo3, BlackWidow repeatedly selected different calendar dates that triggered identical backend logic, wasting substantial execution budget without increasing coverage. Likewise, in Collabtive, the `reply message` action generated a new message identifier upon each execution, leading EvoCrawl to repeatedly exercise semantically equivalent actions. By deduplicating actions based on resource-operation semantics, Theseus avoided redundant executions and explored deeper states, successfully exposing vulnerabilities that were missed by all baselines.

**Table 7: Code coverage improvements of Theseus over a variant implementation. Detailed per-application results are available in the online appendix Table 11 .**

| Baseline | # Th-b | # Th-upq | # Th-rb | # Th-s |
|---|---|---|---|---|
| **Improvements** | 91.6% | 51.9% | 47.1% | 20.5% |

**Table 8: Numbers of succeeded actions and failed attempts for Theseus and Th-rb. Detailed per-application results are available in the online appendix Table 10 .**

| Method | Success | Fail | Total | Success rate |
|---|---|---|---|---|
| Theseus | 12,597 | 2,080 | 14,677 | 85.8% |
| Th-rb | 7,559 | 3,279 | 10,838 | 69.7% |

*5.4.4 False Negatives.* As demonstrated in Table 6, Theseus failed to detect a total of 30 vulnerabilities, including 25 known vulnerabilities, and 5 additional new vulnerabilities.

All vulnerabilities detected by BlackWidow were also identified by Theseus. However, Theseus missed 7 vulnerabilities–3 new and 4 known–reported by EvoCrawl, and 5 vulnerabilities–2 new and 3 known–reported by YuraScanner. Among these, 2 false negatives occurred because Theseus failed to explore pages that were visited by EvoCrawl using its evolutionary search strategy, and 5 false negatives were related to web forms. For example, in Librenms, one vulnerability requires the successful creation of the `rule` resource, which demands a strictly formatted URL field. EvoCrawl bypassed this requirement by leaving the field blank, while Theseus failed to do so with a fixed string value and consequently missed this vulnerability. The remaining 5 false negatives were reported by YuraScanner and are attributable to its task-driven strategy.

All evaluated crawlers showed limitations in handling complex form inputs and navigating deep pages that require strict workflows between web pages, which led to the misdetection of 19 known vulnerabilities. This highlights the necessity of addressing these challenges in future research.

## 5.5 RQ4: Ablation Study

For this research question, we evaluate the contribution of each core component of Theseus: resource-operation-guided deduplication, dependency-aware navigation scheduling, and pre-execution analysis parallelization. To this end, we design four ablated variants:

- Th-b: A BlackWidow variant where the original Random State navigation and URL Equality–based deduplication are replaced by more effective alternatives: randomized BFS navigation scheduling and URL path–query–based deduplication.
- Th-upq: Theseus with its resource-operation-guided deduplication replaced by URL path-query-based deduplication.
- Th-rb: Theseus with its dependency-aware navigation scheduling replaced by randomized BFS scheduling.
- Th-s: Theseus without parallelization.

We evaluate these variants from two perspectives: code coverage and action execution results.

As shown in Table 7, the coverage of Theseus outperforms that of Th-b, Th-upq, Th-rb, and Th-s by 91.6%, 51.9%, 47.1%, and 20.5%, respectively. These results confirm the effectiveness of all three modules—particularly resource-operation-guided deduplication and dependency-aware navigation—which are both underpinned by our semantic modeling of user actions.

To evaluate the impact of dependency-aware navigation scheduling, Table 8 summarizes the overall action execution outcomes of Theseus and Th-rb. Theseus achieves 12,597 successful actions compared to 7,559 for Th-rb (a 66.6% increase), while reducing failures from 3,279 to 2,080 (a 36.6% reduction). These improvements confirm the superiority of our navigation strategy over randomized BFS scheduling. A detailed per-application breakdown is available in the online appendix Table 10 .

## 6 Related Work

**Navigation Scheduling.** Existing web crawlers predominantly utilize basic navigation algorithms such as breadth-first search (BFS) [4, 20, 34], depth-first search (DFS) [9, 19], or random exploration [13, 14, 18, 29]. These methods determine the sequence of actions solely based on the order in which elements are discovered, without considering logical dependencies or semantic relationships, which often results in inadequate coverage. Advanced strategies, including reinforcement learning [44] and evolutionary techniques [17], attempt to optimize navigation sequences but require extensive manual parameter tuning, lack generalizability across diverse applications, and fail to capture the inherent dependencies between web operations. Recent work models crawling as an adversarial multi-armed bandit to avoid brittle state abstractions and improve coverage, but still does not explicitly capture dependencies among web operations [5]. In contrast, our resource-guided approach harnesses natural dependencies among operations to establish a principled topological ordering, ensuring comprehensive exploration of states without the need for application-specific training or manual configuration.

**Deduplication.** Current web crawlers adopt both URL-based [10, 12, 13, 19, 44] and DOM-based [6, 20, 21, 37, 41] deduplication approaches. URL-based methods are ineffective for non-URL-triggering actions and subject to minor parameter changes, prompting researchers to propose partial URL comparison schemes [35] that focus on domain and path components while ignoring query parameters. DOM-based algorithms compute the tree edit distance (RTED) [30] between DOM trees but require carefully crafted similarity thresholds to differentiate functionally distinct pages, often resulting in under- or over-deduplication. In contrast, Theseus conducts deduplication based on the functional equivalence of operations, leveraging LLMs to differentiate actions based on their operational intent, thereby avoiding manual threshold tuning while maintaining accuracy across diverse web applications.

**LLM Integration in Web Testing.** The integration of Large Language Models into web testing frameworks has emerged as a promising method for enhancing automation and coverage. FormNexus [1] utilizes LLMs with semantic inference to generate comprehensive test cases for web forms, achieving 89% coverage by extracting insights from form elements and their relationships. Li et al. [23] conduct an empirical study demonstrating that LLMs can effectively generate web form tests when provided with appropriate contextual information. YuraScanner [36] introduces a task-driven methodology that leverages LLMs to sequentially execute tasks from task lists extracted via shallow crawling or provided by users, enabling deeper exploration and improved vulnerability discovery compared to traditional scanners. Mind2Web [8] and PentestGPT [7] showcase LLM applications in web agent evaluation and interactive penetration testing, respectively.

In this paper, we employ LLMs to perform semantic analysis of action intents across diverse web application components.

## 7 Conclusion

We develop Theseus, a smart web crawling tool that leverages LLMs to infer the semantics of resource operations from user action intents, providing guiding clues for efficient web crawling. We demonstrate that Theseus significantly outperforms existing crawlers in improving code coverage and assisting in the discovery of new vulnerabilities. Our future work includes addressing the limitations of Theseus in handling form inputs and deep pages, as well as testing more types of vulnerabilities.

## Acknowledgments

We thank all reviewers for their valuable feedback. This work is supported by the Young Scientists Fund of the National Natural Science Foundation of China (62402474).

## Ethical Considerations

This research aims to improve web application security by enabling scanners to uncover both known and previously undisclosed XSS vulnerabilities. All evaluations were conducted on self-hosted deployments of open-source applications; no human subjects, user-generated content, or production data were involved. We followed responsible disclosure practices for every newly discovered vulnerability: issues were privately reported to maintainers under a 90-day timeline [16], and exploit details are withheld until patches become available. We align with the ACM CCS Ethics Policy and believe the benefits of strengthening web application defenses outweigh the residual risks, which we mitigated through responsible and staged disclosure.

## Open Science

In accordance with the ACM CCS 2026 Open Science Policy, we have publicly released the source code of Theseus at https://github.com/Theseus-c/Theseus to facilitate future research and reproducibility. The repository also hosts the supplementary appendix referenced in this paper.

## References

[1] Parsa Alian, Noor Nashid, Mobina Shahbandeh, and Ali Mesbah. 2024. Semantic constraint inference for web form test generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 932–944.

[2] Arctic Wolf. 2024. The Most Exploited Vulnerabilities of the Year. https://arcticwolf.com/the-most-exploited-vulnerabilities-of-the-year/.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[4] Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Clara Schneidewind, Marco Squarcina, and Mauro Tempesta. 2018. {WPSE}: Fortifying Web Protocols

via {Browser-Side} Security Monitoring. In *27th USENIX Security Symposium (USENIX Security 18)*. 1493–1510.

[5] Lorenzo Cazzaro, Stefano Calzavara, Maksim Kovalkov, Aleksei Stafeev, and Giancarlo Pellegrino. 2025. Less is More: Boosting Coverage of Web Crawling through Adversarial Multi-Armed Bandit. In *2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 183–192.

[6] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. 2012. CrossCheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 171–180.

[7] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. 2024. {PentestGPT}: Evaluating and harnessing large language models for automated penetration testing. In *33rd USENIX Security Symposium (USENIX Security 24)*. 847–864.

[8] Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems* 36 (2023), 28091–28114.

[9] Vladan Djeric and Ashvin Goel. 2010. Securing {Script-Based} Extensibility in Web Browsers. In *19th USENIX Security Symposium (USENIX Security 10)*.

[10] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the state: A {state-aware}{black-box} web vulnerability scanner. In *21st USENIX Security Symposium (USENIX Security 12)*. 523–538.

[11] Adam Doupé, Marco Cova, and Giovanni Vigna. 2010. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 111–131.

[12] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. 2020. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1953–1970.

[13] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. 2021. Black widow: Blackbox data-driven web scanning. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1125–1142.

[14] Benjamin Eriksson, Amanda Stjerna, Riccardo De Masellis, Philipp Rüemmer, and Andrei Sabelfeld. 2023. Black Ostrich: Web application scanning with string solvers. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 549–563.

[15] Amin Milani Fard and Ali Mesbah. 2013. Feedback-directed Exploration of Web Applications to Derive Test Models. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 278–287.

[16] Google Project Zero Team. 2019. Vulnerability Disclosure FAQ. https://googleprojectzero.blogspot.com/p/vulnerability-disclosure-faq.html. Last updated: November 29, 2021. Detailed explanation of 90-day disclosure policy with grace periods, statistics (96.1% fix rate), and exceptional cases..

[17] Xiangyu Guo, Akshay Kawlay, Eric Liu, and David Lie. 2025. EvoCrawl: Exploring Web Application Code and State using Evolutionary Search. In *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS)*.

[18] Geng Hong, Zhemin Yang, Sen Yang, Lei Zhang, Yuhong Nan, Zhibo Zhang, Min Yang, Yuan Zhang, Zhiyun Qian, and Haixin Duan. 2018. How you get shot in the back: A systematical study about cryptojacking in the real world. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1701–1713.

[19] Jordan Jueckstock, Peter Snyder, Shaown Sarker, Alexandros Kapravelos, and Benjamin Livshits. 2022. Measuring the privacy vs. compatibility trade-off in preventing third-party stateful tracking. In *Proceedings of the ACM Web Conference 2022*. 710–720.

[20] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. 2006. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web*. 247–256.

[21] I Luk Kim, Weihang Wang, Yonghwi Kwon, Yunhui Zheng, Yousra Aafer, Weijie Meng, and Xiangyu Zhang. 2018. Adbudgetkiller: Online advertising budget draining attack. In *Proceedings of the 2018 World Wide Web Conference*. 297–307.

[22] Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. 2025. {LLMxCPG}:{Context-Aware} Vulnerability Detection Through Code Property {Graph-Guided} Large Language Models. In *34th USENIX Security Symposium (USENIX Security 25)*. 489–507.

[23] Tao Li, Chenhui Cui, Rubing Huang, Dave Towey, and Lei Ma. 2024. Large Language Models for Automated Web-Form-Test Generation: An Empirical Study. *ACM Transactions on Software Engineering and Methodology* (2024).

[24] Ziyang Li, Saikat Dutta, and Mayur Naik. 2024. IRIS: LLM-assisted static analysis for detecting security vulnerabilities. *arXiv preprint arXiv:2405.17238* (2024).

[25] Fengyu Liu, Yuan Zhang, Enhao Li, Wei Meng, Youkun Shi, Qianheng Wang, Chenlin Wang, Zihan Lin, and Min Yang. 2025. BACScan: Automatic Black-Box Detection of Broken-Access-Control Vulnerabilities in Web Applications. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. 1320–1333.

[26] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[27] OpenAI. 2023. *OpenAI API Libraries*. https://platform.openai.com/docs/libraries

[28] OWASP Foundation. 2026. Cross-Site Scripting (XSS) Attack. https://owasp.org/www-community/attacks/xss/

[29] Xiang Pan, Yinzhi Cao, and Yan Chen. 2015. I do not know what you visited last summer: Protecting users from third-party web tracking with trackingfree browser. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*.

[30] Mateusz Pawlik and Nikolaus Augsten. 2011. RTED: A Robust Algorithm for the Tree Edit Distance. *Proceedings of the VLDB Endowment* 5, 4 (2011), 334–345.

[31] PortSwigger. 2023. *Burp Suite Professional User Guide*. https://portswigger.net/burp/documentation

[32] Qwen2.5-32B-Instruct. 2024. Qwen2.5-32B-Instruct: Large Language Model with Instruction Tuning. https://modelscope.cn/models/Qwen/Qwen2.5-32B-Instruct.

[33] Derick Rethans. 2026. Xdebug - The PHP Debugger. https://xdebug.org/

[34] Asuman Senol, Gunes Acar, Mathias Humbert, and Frederik Zuiderveen Borgesius. 2022. Leaky forms: A study of email and password exfiltration before form submission. In *31st USENIX Security Symposium (USENIX Security 22)*. 1813–1830.

[35] Aleksei Stafeev and Giancarlo Pellegrino. 2024. {SoK}: State of the Krawlers–Evaluating the Effectiveness of Crawling Algorithms for Web Security Measurements. In *33rd USENIX Security Symposium (USENIX Security 24)*. 719–737.

[36] Aleksei Stafeev, Tim Recktenwald, Gianluca De Stefano, Soheil Khodayari, and Giancarlo Pellegrino. 2025. YURASCANNER: Leveraging LLMs for Task-driven Web App Scanning. (2025).

[37] Karthika Subramani, William Melicher, Oleksii Starov, Phani Vadrevu, and Roberto Perdisci. 2022. PhishInPatterns: measuring elicited user interactions at scale on phishing websites. In *Proceedings of the 22nd ACM Internet Measurement Conference*. 589–604.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[39] Will Wills. 2019. selenium-wire: Extends Selenium's Python bindings to give you access to the underlying requests made by the browser. https://github.com/wkeeling/selenium-wire.

[40] Michal Zalewski. 2025. Skipfish. https://code.google.com/archive/p/skipfish/.

[41] Eric Zeng, Miranda Wei, Theo Gregersen, Tadayoshi Kohno, and Franziska Roesner. 2021. Polls, clickbait, and commemorative $2 bills: problematic political advertising on news and media websites around the 2020 US elections. In *Proceedings of the 21st ACM Internet Measurement Conference*. 507–525.

[42] Wei Zhang and Chen Liu. 2022. OWASP ZAP: Comprehensive Security Testing Framework. In *Proceedings of the International Conference on Web Security*. IEEE, San Francisco, CA, 123–135.

[43] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* 1, 2 (2023).

[44] Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. 2021. Automatic web testing using curiosity-driven reinforcement learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 423–435.