Less is More: Boosting Coverage of Web Crawling through Adversarial Multi-Armed Bandit

Lorenzo Cazzaro*, Stefano Calzavara*, Maksim Kovalkov*, Aleksei Stafeev† and Giancarlo Pellegrino†
*Università Ca' Foscari Venezia, Italy
Email: {lorenzo.cazzaro, stefano.calzavara}@unive.it, 888980@stud.unive.it
†CISPA Helmholtz Center for Information Security, Germany
Email: {aleksei.stafeev, pellegrino}@cispa.de

***Abstract*—Crawlers are critical for ensuring the dependability and security of web applications by maximizing the code coverage of testing tools. Reinforcement learning (RL) has recently emerged as a promising approach to improve crawler exploration. However, existing approaches based on Q-learning face two major limitations: being state-based, they rely on brittle state abstractions that fail to generalize across diverse applications, and they employ rewards that prioritize underused actions but are not necessarily proportional to the improvement in code coverage. In this paper, we first substantiate the limitations of two popular Q-learning-based crawlers. We then propose Multi-Armed Krawler (MAK), a new crawler based on the Adversarial Multi-Armed Bandit problem. MAK is stateless and does not require the definition of brittle state abstractions that do not generalize to new web applications. By modeling the crawling process through a traditional graph abstraction and introducing an extrinsic reward correlated with code coverage, MAK compensates for the loss of expressiveness coming from its stateless nature. Our experimental results on public web applications show that MAK achieves greater coverage and faster convergence than its counterparts.**

I. Introduction

Web applications routinely manage critical operations for industries, such as financial transactions and e-commerce, and therefore must be dependable and secure to protect sensitive assets and maintain company reputations. To achieve this, web applications are typically assessed using *black-box* techniques. These methods focus on crafting and inspecting HTTP traffic rather than analyzing source code, primarily because of the inherent complexity of multi-tier architectures (frontend, backend, and database), the diversity of backend scripting languages, and the frequent unavailability of the source code, e.g., when performing penetration testing [1]. Black-box analysis of a web application is inherently *dynamic*, as it depends on exploring a substantial portion of the testing surface of the web application. This has led to significant interest in *web crawling*, i.e., techniques to explore the functionalities of web applications by actively navigating their pages. Starting with a set of seed URLs, web crawlers interact with web pages to discover new functionalities and URLs for further exploration. Effective web crawling is critical for black-box testing, as inadequate coverage can leave issues undetected.

Seminal work on web crawling has explored strategies and heuristics to enhance the efficiency of web application exploration [2], [3], [4], [5], [6]. More recently, approaches based on reinforcement learning (RL) have been proposed to learn effective crawling policies, like WebExplor [7] and QExplore [8], that exploit Q-Learning [9]. Their design requires to face the challenges of formulating the web crawling problem in terms of RL. For example, traditional RL approaches rely on state transition systems, requiring careful abstraction functions to represent web pages as states [7], [8]. WebExplor addresses this problem by encoding a state as a pair including the URL of the page and the page tags, while QExplore uses the sequence of attribute values of the unique interactable elements of the page. Similarly, RL requires also the definition of reward functions that provide feedback to train the policy. Both crawlers adopt curiosity-driven rewards [10], [11], [12], assigning higher rewards for novel interactions with the web application. This approach compensates for the lack of server-side visibility inherent to black-box testing.

We believe that research on the adoption of RL for web crawling has merits, because it explored the use of a natural and successful learning technique in a prominent research field. Still, our experience in using these RL-based crawlers reveals that the specific design choices adopted by state-of-the-art RL-based crawlers present some problems. First, the adopted heuristics for state abstraction are too brittle and may not perform well on all the web applications. This causes the creation of too many states by the crawler, making it difficult to learn the policy. Second, the curiosity-driven reward mechanism is short-sighted. It prioritizes exploring actions used only a few times without considering their contribution to activating new server-side code. Consequently, the crawler may assign high rewards to unproductive actions while undervaluing fruitful ones.

In this paper, we claim that *less is more*: the more we simplify the application of RL, the better it generalizes to different web applications. Concretely, we propose a novel crawling approach where the crawler learns an optimal navigation policy in a stateless manner by integrating three strategies inspired by the well-established navigation strategies: breadth-first search, depth-first search, and random, traditionally based on a graph abstraction of the web application. It has been shown that one of these three strategies can provide the best performance in code coverage, depending on the specific explored web application [13]. Our approach treats a crawler as an *Adversarial Multi-Armed Bandit* (AdvMAB) [14], [15]

agent, where the agent learns the probability of each action through a reward function that evaluates link coverage, i.e., the new links gathered as a result of the selected action. We choose *adversarial* MAB to allow the crawler to adapt the probability to select a specific navigation strategy through time, since different parts of the web application may have different best exploration strategies. We implement this approach in a crawler called Multi-Armed Krawler (MAK), and we evaluate it on a set of popular web applications, showing that it outperforms other RL-based crawlers in terms of code coverage and convergence speed.

In summary, our contributions are: (1) We criticize the design choices of two existing RL-based solutions for web crawling, providing concrete examples based on real-world applications. (2) We propose a new approach to web crawling based on a simpler RL problem (AdvMAB) and a more informative reward than the ones previously considered. This framework naturally solves many of the challenges faced by the current state of the art. (3) We evaluate our proposal on popular web applications, showing that our approach pays off.

II. Background

A. Reinforcement Learning

RL problems involve an agent learning by interacting with an environment [9]. At each time step t, the agent is in a state $s \in \mathcal{S}$ and has to choose an action $a \in \mathcal{A}$ to interact with the environment. After executing the action, the agent receives a reward $r_t \in \mathbb{R}$ as a consequence of reaching a state $s' \in \mathcal{S}$. The learning objective is to determine the sequence of actions that maximize the cumulative reward over time, i.e., the agent learns a *policy* $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, mapping each state to the probability of selecting each action to maximize the cumulative reward. The policy π is consulted every time an agent enters a given state to identify the best action to perform there, and the agent moves into another (possibly new) state by executing the chosen action. Solving algorithms for RL problems must typically deal with the trade-off of *exploring* the environment and *exploiting* the learned policy to maximize the cumulative reward. Several scenarios have been explored in RL, depending on how the learning process is modeled and the assumptions on the nature of the rewards.

1) Q-Learning: A popular solution for RL problems is the *Q-Learning* algorithm, that iteratively learns a function $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ estimating the quality of each state-action combination. Before learning begins, Q is initialized to a possibly arbitrary fixed value. At time t, the agent in state s executes an action a and enters a new state s', receiving a reward r_t. The value $Q(s, a)$ is then updated using r_t through the Bellman equation [9]. The action chosen at each time step through the policy π is obtained by prioritizing the action with the highest value in Q, which can be implemented through different strategies. Different representations of the Q-function have been proposed in the literature [9].

2) Adversarial Multi-Armed Bandit: A specific type of RL problem called Multi-Armed Bandit (MAB) is formally equivalent to learning a policy where $\mathcal{S}$ contains just a single state. In the case of MAB, the set of actions $\mathcal{A}$ is defined as $\{1, \ldots, K\}$, i.e., it contains K actions, called *arms*, and the policy to learn has the simpler form $\pi : \mathcal{A} \to [0, 1]$. In this paper, we are interested in the AdvMAB formulation [14] where there is no statistical assumption about the process generating the rewards over time, but it is assumed that an adversary sets the reward simultaneously with each action. This means that the distribution of the rewards for each action may change over time. A solution for AdvMAB is the Exp3.1 algorithm [15], shown in Algorithm 1.

Algorithm 1 Exp3.1 algorithm

1: **Input:** Number of arms K and total time budget T
2: $t \leftarrow 0$
3: $\hat{G}_i = 0$ for $i = 1, \ldots, K$
4: **while** $t < T$ **do**
5: **for** $m = 0, 1, 2, \ldots$ **do**
6: $g_m = \frac{K \ln K}{e-1} 4^m$
7: $\gamma_m = \min\left(1, \sqrt{\frac{K \ln K}{(e-1) g_m}}\right)$
8: $w_i = 1$ for $i = 1, \ldots, K$
9: **while** $\max_i \hat{G}_i \leq g_m - K/\gamma_m$ **do**
10: $\pi(i) = (1 - \gamma_m) \frac{w_i}{\sum_{j=1}^{K} w_j} + \frac{\gamma_m}{K}$, for $i = 1, \ldots, K$
11: Choose the action a sampling from policy π
12: Obtain the reward r_t by executing a
13: $\hat{r}_{t,i} = \begin{cases} \frac{r_t}{\pi(i)} & \text{if } i = a \\ 0 & \text{otherwise.} \end{cases}$
14: $w_i = w_i \exp\left(\frac{\gamma_m \hat{r}_{t,i}}{K}\right)$ for $i = 1, \ldots, K$
15: $\hat{G}_i = \hat{G}_i + \hat{r}_{t,i}$ for $i = 1, \ldots, K$
16: $t \leftarrow t + 1$

The algorithm takes as input the number of arms K and the total time budget T. The algorithm starts by initializing the estimated gain $\hat{G}_i$ of each action i to 0 and chooses the next action to perform at the time step t until the total time budget T has been reached. It operates in epochs, i.e., different exploration phases, each with a different estimate g_m and learning rate γ_m, with action weights w_i initially set to 1 (lines 6-8). The adoption of the epochs mechanism ensures that the learning rate is progressively reduced as more information on the gain of the actions is gathered, ensuring proper tradeoff between exploration and exploitation. Each epoch lasts until the maximum estimated gain of the actions exceeds the quantity $g_m - K/\gamma_m$ (line 9). At each time step, the algorithm updates the policy π based on the current weights and learning rate; the policy is then used to sample the next action a to perform at time step t. The corresponding reward r_t is then used to update the estimated rewards $\hat{r}_{t,i}$ of the actions (lines 10-13), that are the reward divided by the probability of each action, to compensate for actions that have a small probability of being selected. Afterward, the weights of the actions and the estimated gains are updated using the estimated rewards and the learning rate before moving to the next time step (lines 14-16).

Algorithm 2 General RL-based web crawling algorithm

```
1: Input: The start page p and the total time budget T
2: π ← INIT()                                          ▷ Initialize the policy
3: t ← 0
4: s ← GET_STATE(p)                                    ▷ Get the initial state
5: while t < T do
6:     A ← GET_ACTIONS(p)                              ▷ Get the set of actions
7:     a ← CHOOSE_ACTION(π, s, A)                      ▷ Check the policy
8:     p' ← EXECUTE(p, a)                              ▷ Navigate the crawler
9:     s' ← GET_STATE(p')                              ▷ Get the new state
10:    r ← GET_REWARD(s, a, s')                        ▷ Get the reward
11:    π ← UPDATE_POLICY(π, r, s, a, s')               ▷ Policy update
12:    p ← p'
13:    s ← s'
14:    t ← t + 1
```

B. Web Crawling

The core principle of a web crawler is an iterative interaction with a target web application. In this work, we assume that the crawler interacts with web applications in a black-box fashion, i.e., without accessing the source code of the web application on the server side. Only the Document Object Model (DOM) of each page of the web application is accessible. This is a widely adopted assumption in the literature [3], [6], [7], [8].

The crawler starts at a seed URL and iteratively extracts interactable elements of web pages (links, buttons, etc...) to schedule them for interaction. Each visited page contributes to the pool of elements, while the scheduling algorithm of the crawler defines the order of interaction with the pool, i.e., the navigation strategy. The most common algorithms are breadth-first (BFS), depth-first (DFS), and random choice [13], that abstract the web application as a graph. BFS traverses a web application by always picking the least recently discovered element, while DFS prioritizes the new elements over the old ones. A random strategy would pick elements from the pool in a randomized fashion. A typical goal of an effective crawler is maximizing *code coverage*, i.e., the number of lines of server-side code activated by crawling the web application. Improving code coverage is useful to ensure that more components of the backend are tested, e.g., for potential security issues.

C. RL for Web Crawling

Among the solutions proposed for performing a dynamic black-box exploration of web applications, RL-based approaches are prominent [7], [8], [16]. Algorithm 2 describes a generic implementation of a RL-based web crawler. The crawler receives as input the start page p of the web application under test and the total time budget T given to perform the exploration. The exploration starts by initializing the policy π to some default value. Then, it iterates by finding a state abstraction of the current web page and scheduling the next action to perform as the best action for the current state, based on the currently learned policy. After executing the action, the crawler gets a reward, which is used to update the policy for the next iteration.

III. Limitations of Existing RL-Based Crawlers

WebExplor [7] and QExplore [8] represent state-of-the-art RL-based solutions for web crawling. They employ Q-Learning to instantiate the main building blocks of Algorithm 2. WebExplor implements the UPDATE_POLICY as defined by the Bellman equation, while QExplore modifies the update to guide the crawler to states with more actions. Their implementation of CHOOSE_ACTION also differ: WebExplor uses the Gumbel-softmax strategy [17] to choose the action, while QExplore privileges a deterministic strategy which always picks the action with the highest Q-value. There are also some small differences in the implementation of the GET_ACTIONS function as well, with the two crawlers supporting different types of operations over web pages, e.g., filling inputs in a sophisticated way.

In this paper though, we want to focus on the definitions of the GET_STATE and GET_REWARD functions, since they present conceptual limitations that affect the effectiveness of these two crawlers in practice. We now discuss them by presenting examples found by testing WebExplor and QExplore on real applications, using the setup described in Section V.

A. State Abstraction

WebExplor and QExplore abstract the state of the explored web application to define the states of the underlying Q-Learning problem. Specifically, they represent states by first applying a *pre-processing function* to the page, that captures the relevant information for the state abstraction, and then using a *similarity function* to establish whether the encountered state is new or an existing one. The two functions are tied since they aim to satisfy two complementary objectives. On the one hand, the pre-processing function should model sufficient information to let the state represent a meaningful part of the interaction of the agent with the environment. On the other hand, the similarity function should limit the number of states, otherwise the agent would observe artificial differences and the RL solving algorithm would not be effective due to the state explosion problem. Ideally, the state abstraction should group the visited web pages such that (i) web pages that share the same business logic (same server-side code) are mapped to the same state and (ii) web pages with different business logic (different server-side code) are mapped to different states. We show below that this may not be the case given the specific design choices of the two crawlers.

The pre-processing function employed by WebExplor abstracts a web page as a pair including the URL of the page and the sequence of its HTML tags. The similarity function, instead, operates by first performing exact URL matching over the existing states. If the URL of the page is not new, then the HTML tags of the page are compared against existing states using a pattern matching algorithm to determine whether to create a new state. If the URL is new, instead, a new state is created. This approach is brittle, because an exact URL matching might lead to the artificial creation of new states. For example, consider HotCRP [18], a popular web application for managing conference review processes (top part of Figure 1).

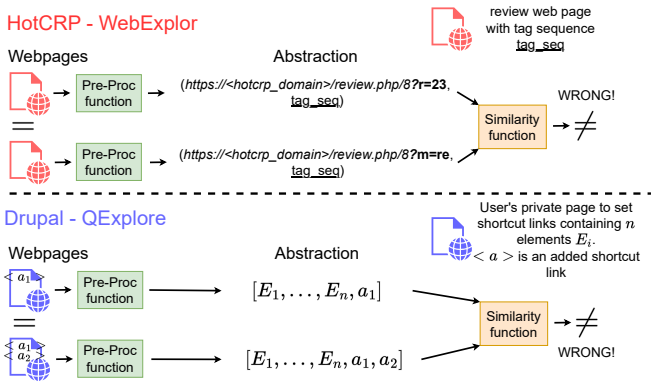

Fig. 1: Examples of the limitations of the state abstractions performed by WebExplor and QExplore.

Suppose a reviewer with ID 23 is logged in and is assigned paper 8, whose review form can be accessed via two different links. The two links contain distinct query parameters, *r* and *m*, in bold in the figure. Although both links lead to the same review form, the similarity function of WebExplor treats the corresponding states as different due to the usage of exact URL matching. It might be tempting to just relax the URL matching algorithm by ignoring the query string entirely. However, this approach would be inappropriate for certain web applications that dynamically generate content based on query parameter values, a common pattern in PHP applications. For instance, Matomo [19], a popular web analytics platform, generates web pages dynamically depending on the value of the query parameter *module*. Different values of the *module* parameter correspond to distinct parts of the application that require testing. For example, setting *module* to *CoreAdminHome* displays the user settings interface, whereas using the value *MultiSites* generates a page listing all the dashboards. Ignoring the query string in such cases would lead to critical parts of the web application being overlooked during testing.

Besides the challenges of URL matching, comparing HTML tags is difficult and error-prone as well, as observed by the authors of QExplore. To mitigate this, QExplore improves upon WebExplor by employing a pre-processing function that abstracts a web page into a sequence of attribute values of the interactable elements of the page. The similarity function then compares the hash of the string representations of the resulting states. Unfortunately, also this approach is far from perfect. We show an example from the popular content management system Drupal [20] (bottom part of Figure 1). Drupal allows users to add shortcut links to their homepage via a shortcut module located on a specific private page of the user. Adding a shortcut link results in the appearance of a new link on this private page, while the other interactable elements of the page remain unchanged. Suppose that QExplore adds a new link when interacting with the page. This link will be an arbitrary string triggering a navigation error, because QExplore has no understanding of the web application semantics and cannot generate valid links. When QExplore processes the private page extended with the new link, the abstraction captures the sequence of attribute values of its n interactable elements E_i and the new link a_1. However, adding a different link a_2 in the same way changes the sequence of elements, forcing QExplore to create a new state for the private page. This can be done an unbounded number of times, yet the new state is unnecessary for effectively crawling the web application, because all the new links trigger navigation errors. This discussion shows that designing an effective state abstraction that generalizes to different web applications is a challenging problem that may not have an optimal solution.

B. Reward Function

Common RL tasks typically expect the agent to reach a clear and visible goal, e.g., winning a computer game, which simplifies the definition of the reward due to the straightforward success or failure signals. Instead, defining a meaningful reward for the task of exploring a web application in a black-box setting is more challenging, because there is no visibility of the explored surface of the web application. To address this, both WebExplor and QExplore leverage an *intrinsic* reward, which depends on the internal state of the agent (the crawler), rather than the external state of the environment (the web application). Although details slightly differ, both WebExplor and QExplore employ a *curiosity*-based reward [11], [12], which encourages the agent to explore unvisited parts of the application. For example, WebExplor uses counters to track the frequency of actions in a given state and assigns higher rewards to less frequently executed actions. While this approach pushes the crawler into getting higher code coverage, it is sub-optimal as it allows the crawler to execute the same action multiple times, irrespective of whether the action triggers a relevant side effect on the server side. For example, WordPress [21] ships a search engine to easily navigate internal pages. Since queries to the search engine are read from the server but do not modify its state, performing the same search action multiple times is generally not useful for discovering new parts of the web application. Although performing the same action multiple times is discouraged by the curiosity mechanism, the crawler does not obtain any useful information from the returned page, i.e., it does not leverage the observation that the search results have not changed. Ideally, we would like to design a reward function that pushes the crawler into taking the same action multiple times only when there is some evidence that this might yield potential benefits in terms of code coverage.

IV. Crawling using Multi-Armed Bandit

We now introduce MAK, our crawler designed to address the limitations of state-of-the-art Q-Learning-based crawlers. MAK leverages the adversarial MAB problem, that allows us to define a simpler yet more effective solution compared to WebExplor and QExplore. In the following sections, we discuss the abstractions employed by MAK and its capabilities, summarized in Table I.

TABLE I: Summary of the components of the reviewed RL-based crawlers and our proposal.

Tool	State Abstraction	Action Definition	Reward	Policy Update	Action Selection
WebExplor	URL + sequence of HTML tags	interactable DOM elements	Curiosity	Q-Learning update	Gumbel-softmax
QExplore	Sequence of attribute values of interactable DOM elements	interactable DOM elements	Curiosity	Modified Q-Learning update	Maximum Q-value
MAK	**Stateless**	**Head, Tail, Random**	**Link coverage**	**Exp3.1**	**Exp3.1**

A. State Abstraction

We already discussed that it is difficult to define a one-size-fits-all state abstraction which generalizes to different web applications, because web developers use different programming conventions. To address this, we adopt a *stateless* representation of the RL problem based on MAB. While this approach may at first seem less expressive compared to the current state-of-the-art, a key claim of this paper is that *less is more*: since state information is so difficult to exploit in the undisciplined setting of web applications, we take a state-agnostic stand by moving away from Q-learning. The expressiveness of our approach lies instead in the careful design of both the action space and the reward.

B. Action Definition

Since our crawler is stateless, we do not keep track of which actions are available depending on the current state of the web application, as done by WebExplor and QExplore. We take a different approach and consider actions inspired by well-known crawling algorithms. In particular, a recent survey on web crawling [13] underlines that the most popular navigation strategies used in web crawling are BFS, DFS, and Randomized strategies (described in Section II-B) that rely on a traditional graph abstraction of the web application. These strategies operate independently of the state of the web application and rely instead on the order in which interactable elements are collected. A major insight of the aforementioned survey is that there is no single strategy which optimally generalizes to different web applications, i.e., different applications benefit from different navigation strategies. Our idea is then to let the crawler learn how to interleave the three different strategies. Concretely, when MAK visits a web page, it first extracts interactable elements and stores them in a deque[1]. The deque is global and shared across different web pages. At each time step, the crawler chooses one of three actions $\mathcal{A} = \{Head, Tail, Random\}$ inspired by the three navigation strategies discussed above: (i) *Head* extracts and interacts with the element at the head of the deque, that corresponds to the least recently discovered interactable element. This action simulates BFS if it is always selected. (ii) *Tail* extracts and interacts with the element at the tail of the deque, that corresponds to the newest action discovered by the crawler. This action simulates DFS if it is always selected. (iii) *Random* extracts and interacts with a random element of the deque. The *Random* action is useful to escape cases where a sequence of *Head* and *Tail* actions does not improve the reward.

[1]A *double-ended queue* (deque) is a generalization of a queue, for which elements can be added to or removed from either the front or back.

Even though the exclusive use of curiosity in WebExplor and QExplore might miss useful information about the coverage of the web application, the rationale behind the curiosity mechanism is sound: encouraging the crawler to prioritize never-seen pages. We consider a different solution while integrating the curiosity principle in our approach by enriching the definition of the action. In particular, we actually implement our deque data structure as a list of deques, each one with an associated level $i \in \mathbb{N}_0$. The deque at level i contains all the interactable elements extracted by the crawler from the pages in the previous interactions with the web application that have already been interacted with by the crawler i times. Then, when one of three actions is executed, the interactable element is extracted from the head, the tail, or randomly from the deque at the lowest level. In this way, we incentivize the crawler to try the least explored actions in general, emulating the curiosity mechanism, and the crawler continues to exploit the three state-agnostic navigation strategies.

Finally, note that the use of a deque does not compromise our stateless abstraction. In previous formulations of web crawling as a RL problem, states and actions encode information about the web application state and the available actions at that state. In contrast, our deque only tracks action availability independently of the state of the web application. This information is not used to model the environment or train the policy, thus no details about the web application are encoded in states of the RL problem formulation.

C. Reward Function

Curiosity is a reasonable reward to encourage the crawler to explore new parts of the web application. Unfortunately, as we explained, curiosity is not necessarily correlated with code coverage, which is the main measure that web crawlers try to optimize [6]. Indeed, curiosity treats two interactable elements used the same number of times as equivalent, even though only one may contribute to increasing the code coverage. For example, consider a shopping webpage with a purchase button for a shopping cart and a link to a frequently asked questions (FAQ) page. Initially, the crawler presses the button with an empty cart, triggering server-side code that leads, for instance, to an error page. Later, it clicks the FAQ link and then adds an item to the cart. Pressing the button again now activates new server-side code related to the purchase, as the cart contains an item, whereas clicking the FAQ link again does not trigger new code. However, the curiosity-driven reward assigns the same value to both interactions, failing to recognize the improved code coverage from pressing the button again. To address this, we propose a different reward mechanism that is not sparse

and effectively approximates the functionalities of the web application discovered through each action.

Since code coverage cannot be directly measured without access to the server-side code of a web application, we use *link coverage* to approximate it. Link coverage is determined by the number of different links gathered during the exploration of the web application and it is positively correlated with code coverage [13]. Hence, we use improvements in link coverage as an easy-to-estimate proxy for changes in code coverage. Of course, an improvement in the number of newly discovered links is more meaningful when compared to the history of previous discoveries. For example, we would penalize a small increment in link coverage if it follows a significant increase over a short period. Conversely, we would not penalize a small increment if the link coverage has stagnated over many steps, because new links (and corresponding server-side code) become increasingly difficult to discover as the exploration proceeds. To take the historical information into account and the variations in link discovery over time, we define the reward to be a *standardized* increment in link coverage relative to previous increments. Formally, given r_t the increment in the link coverage at time step t, and $\bar{r}_t$ and σ_t the mean and standard deviation of all the observed increments up to t, the reward returned by the environment is $\hat{r}_t = (r_t - \bar{r}_t)/\sigma_t$.

D. Policy Update

Finally, we discuss the training of the policy that guides the crawler, which is determined by the specific RL problem formulation. We opt for the *adversarial* formulation of MAB, because it does not rely on statistical assumptions about the reward generation process. Specifically, the reward distribution for each action may not remain fixed even within the same application during exploration; instead, it may change. This is well-suited to web crawling, where modern web applications are often modular, comprising components that act as smaller web applications that benefit from distinct navigation strategies. For example, in platforms like WordPress, the profile page is distinct from the blog pages. This modularity is also enabled by specific features of web development frameworks such as Flask, that offer the *blueprint* abstraction [22] to structure modular web applications.

Having clarified the specific RL problem, we can define how to update the policy. Specifically, we choose to implement the policy update as specified by the Exp3.1 algorithm (see Section II-A), a standard solving algorithm for AdvMAB. We adopt this specific algorithm since it works by periodically resetting the weights associated with each action, used to compute their probabilities, ensuring the stability of the algorithm over time and helping the agent to change selected actions depending on the changes in the reward distributions. Note that this algorithm is also applied in security-related work that exploits the AdvMAB formulation for the same reasons [23].

However, the Exp3.1 algorithm requires the reward to be in $[0, 1]$, while our standardized improvement in link coverage $\hat{r}_t$ ranges in $(-\infty, \infty)$. We use the logistic function $1/(1+e^{-x})$ to normalize $\hat{r}_t$ in $[0, 1]$, as done in [23].

V. Experimental Evaluation

A. Methodology

1) Evaluation Framework: Performing a critical comparison of the effectiveness of MAK, WebExplor, and QExplore is challenging, because their implementations may not just differ in the choice of the RL problem and the associated solving algorithm, but also in other details that may easily introduce bias in the experimental evaluation. In principle, small implementation differences, e.g., support for different HTML elements, may play a role in crawling effectiveness more than the specifics of the RL formulation underlying the crawlers. Our comparison is further complicated by the lack of a public implementation of WebExplor, which forced the authors of QExplore to reimplement their own version of WebExplor to carry out their experimental evaluation [24]. To ensure a fair and insightful comparison of the three crawlers, we implemented a unified framework that enables easy implementation of RL-based crawlers by instantiating the building blocks of Algorithm 2. We implement the considered crawlers within our framework by following the original descriptions in the WebExplor and QExplore papers, using the available code from the QExplore authors as guidance when needed. We make all our code available for transparency [25].

2) Assumptions: To uniform crawler implementations, we enforce the following: (i) We define interactable elements as visible elements (links, buttons, and forms). Extending the crawlers to support additional actions is just a matter of engineering effort. (ii) Actions that lead the navigation to external domains are marked as invalid, as done in previous work [7], [8]. (iii) We do not implement the Deterministic Finite Automaton (DFA) feature of WebExplor since it escapes the RL formulation and guides the exploration of the web application beyond the policy capabilities. This assumption does not overly penalize WebExplor, because the authors show that WebExplor with and without DFA converges to around the same code coverage in 30 minutes [7], which is the time considered in our experiments.

3) Testbed and Evaluation Criteria: We consider eight PHP-based web applications of different complexity used in prior work on crawler evaluations [6], [13]: AddressBook (v8.2.5), Drupal (v8.6.15), HotCRP (v2.102), Matomo (v4.11.0), OsCommerce2 (v2.3.4.1), PhpBB2 (v2.0.23), Vanilla (v2.0.17.10) and WordPress (v5.1.0). We exclude the PHP-based web applications tested in [7] and [8], due to their relatively small size and lower popularity. Moreover, we consider other three web applications built with Node.js and React to diversify the considered frameworks: Actual (v25.2.1), Docmost (v0.8.4), and Retro-board (v5.5.2). Retro-board has been used in [7], while Docmost and Actual have been selected from [26], have more than 10k stars on Github, and are employed on two distinct application domains (documentation reporting and finance accounting) not covered by the other selected applications. We do not consider the other Node.js applications used in [7] and [8] because they are no longer maintained and we were unable to start them.

Finally, we consider *code coverage* of the crawler as the performance metric. We compute it as the lines of code on server side that were executed during the interaction of the crawler with the web application [6], [13]. To retrieve it, we use Xdebug [27] and `coverage-node` [28] for PHP-based and Node.js-based web applications, respectively. Xdebug supports code coverage analysis at any point during the execution of the web application, while `coverage-node` generates code-coverage results only after the application stops, resulting in a coarser-grained analysis for Node.js-based web applications.

4) Configurations: Each experiment consists of running the crawler on a web application for 30 minutes, as done in previous work [7], [8]. We repeat the experiments for each pair of crawlers and web applications for 10 times. Finally, we set the parameters of each baseline as stated in the corresponding papers and implementation, if available.

B. Experimental Results on Code Coverage

First, we compare the mean code coverage achieved over time by MAK, WebExplor, and QExplore. We consider only the PHP-based web applications in this experiment since only Xdebug can provide the code coverage at any time during the execution of the web application. Figure 2 presents the results. MAK consistently outperforms the baselines, particularly on larger applications like Drupal, achieving a mean code coverage of 50,445 lines compared to the 45,761 lines covered by WebExplor (+4,684). On smaller applications like OsCommerce2, MAK also excels with 5,051 lines covered on average, surpassing the 4,924 lines covered by WebExplor (+127). Additionally, MAK generally demonstrates faster convergence than the baselines, reaching the highest coverage on PhpBB2 in under six minutes, whereas the baselines fail to achieve the same code coverage in 30 minutes. This pattern extends to other smaller applications. These results confirm the superior exploration capabilities of MAK due to its stateless nature and its more expressive reward.

Comparing the code coverage over time is useful for evaluating the relative performance of different crawlers, but does not provide an objective evaluation of the crawlers with respect to a ground truth. Unfortunately, calculating the total lines of server-side code for each application is challenging and error-prone, as noted in prior work [6]. `coverage-node` provides the total number of lines of server-side code for Node.js-based web applications, while Xdebug does not offer this feature. To address this, we estimate the total number of lines of server-side code for PHP-based web applications by taking the union of the unique lines of code covered by all crawlers, across all runs, for each application. Using the counted total number of lines of server-side code as the ground truth, we estimate the mean code coverage, i.e., the mean percentage of lines covered by each crawler relative to our ground truth estimate. The results in Table II show that MAK obtains a higher estimated mean code coverage on all the PHP- and Node.js-based web applications. For instance, on HotCRP and Retro-board, MAK covers respectively 87.3% and 51.9% of the ground truth on average, compared to WebExplor 77.2% and 48.9% (+10.1% and +3.0%), despite the latter leveraging state information. MAK also covers over 80% of the observed lines of code on more than half of the considered web applications on average. These results confirm that the approach of MAK enhances exploration and outperforms state-based methods on all the considered PHP- and Node.js-based web applications.

TABLE II: Estimated mean code coverage of the crawlers. The best result for each web application is marked in bold.

Application	MAK	WebExplor	QExplore
AddressBook	**99.3%**	98.5%	96.4%
Drupal	**76.8%**	69.6%	68.7%
HotCRP	**87.3%**	77.2%	71.2%
Matomo	**85.1%**	82.3%	83.5%
OsCommerce2	**80.7%**	78.7%	78.1%
PhpBB2	**89.4%**	83.6%	89.2%
Vanilla	**97.7%**	89.5%	88.7%
WordPress	**50.5%**	48.4%	46.8%
Actual	**64.6%**	64.1%	64.1%
Docmost	**64.7%**	64.0%	64.0%
Retro-board	**51.9%**	48.9%	48.9%

C. Ablation Study

We now assess the importance of the learning component of MAK by performing an ablation study. In particular, we compare MAK against three non-learning crawlers implementing the navigation strategies BFS, DFS, and Random. Note that these strategies can be simulated with MAK by always executing one of its three actions *Head*, *Tail*, and *Random* (see Section IV-B). We define the *regret* of the crawler c on the web application w as the difference between the average number of lines of code covered by the best crawler minus the average number of lines of code covered by c, divided by the total number of lines of code of w. If a crawler exhibits the best estimated mean code coverage on a web application, then the regret for that web application is zero. The *cumulative regret* of a crawler is just the sum of its regrets over the different applications. A lower cumulative regret indicates a crawler that consistently performs closer to the optimal strategy over the web applications. The results confirm the effectiveness of the policy learned by MAK over non-learning crawlers. In particular, MAK achieves the lowest cumulative regret, 14.9. BFS follows with 36.0, while DFS and Random perform significantly worse, with cumulative regrets of 126.7 and 70.2. These results highlight the importance of the learning component of MAK, as dynamically alternating between navigation strategies allows it to outperform static crawlers that can not adapt to different (parts of) web applications.

D. Performance Evaluation

Since all the crawlers were executed for the same amount of time, MAK already proved more efficient than competitors, being able to achieve better code coverage. Still, to better understand the improved efficiency, we compute the mean number of interacted elements during execution by MAK, WebExplor and QExplore over the web applications, which

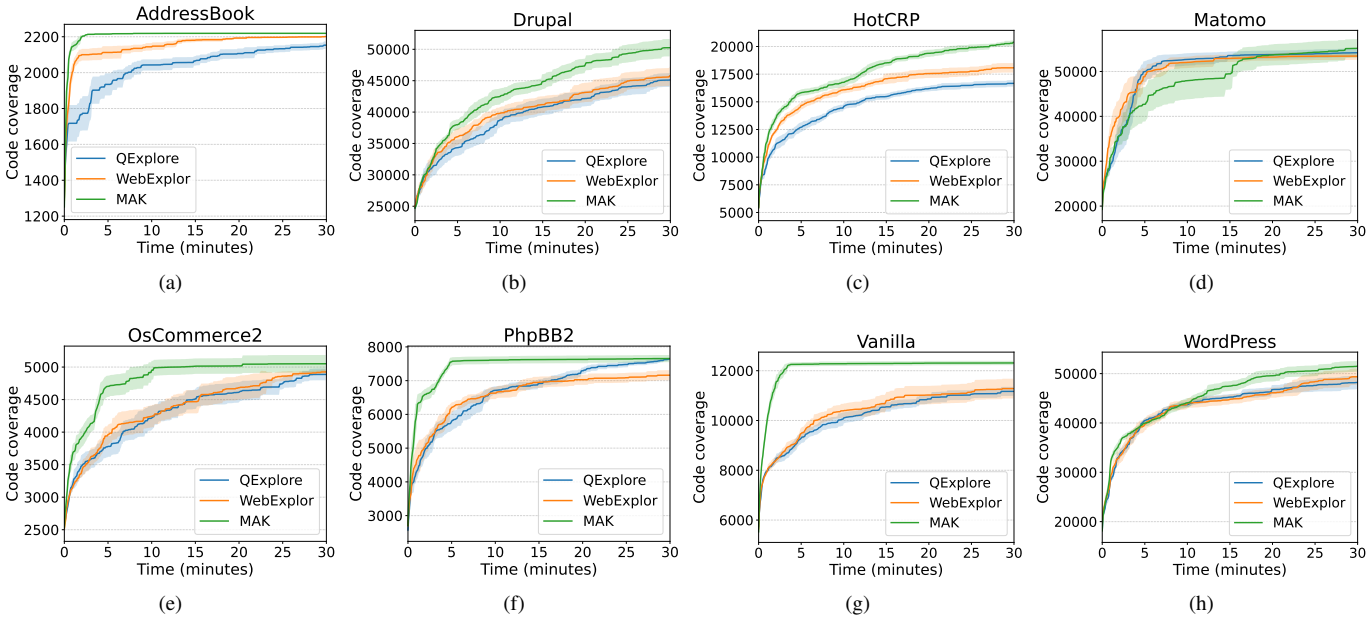

Fig. 2: Mean and standard deviation of the code coverage reached over 30 minutes by QExplore, WebExplor and MAK.

corresponds to the mean number of atomic actions performed by each crawler. The results show that MAK performs a number of interactions comparable to WebExplor and QExplore. Specifically, MAK interacts with an average of 883 elements per run, compared to 854 for WebExplor and 827 for QExplore. These findings indicate that the increase in code coverage achieved by MAK is not merely due to more frequent interactions but rather to a more effective selection of elements to interact with during crawling.

VI. Related Work

Crawlers are widely employed for tasks like web application testing [2], [29], vulnerability detection [6], [30], [31] and web measurements [32], [33], [34]. Therefore, previous work has proposed several web crawlers with different navigation strategies and state abstractions [3], [4], [5], [6], [30], [35]. However, a recent survey [13] has shown that no single navigation strategy or page similarity algorithm performs best across diverse applications. Our work starts from this observation to identify and criticize the limitations of existing RL-based crawlers, providing concrete examples and a possible solution.

RL has gained attention in web crawling [36], [37]. WebExplor [7] and QExplore [8] leverage Q-Learning to learn how to explore web applications. Our work builds on them by proposing a simpler yet effective stateless crawler that addresses their limitations. Other RL-based solutions to explore web applications include the usage of multi-agent reinforcement learning [38] or generic graphical user interface (GUI) testing [39], [40], [41]. In this work, we restrict the scope to single-agent RL-based crawlers specifically tailored for web applications. Our proposal has the potential to improve multi-agent RL-based crawlers as well, because each agent of the ensemble can benefit from our stateless approach.

The work most closely related to ours is [16], that compares Q-Learning-based tools for application testing within a unified framework. However, there are important differences with our work: (i) we not only criticize existing Q-Learning-based solutions, but also propose a novel RL-based crawler to address their limitations; (ii) our experimental evaluation is more realistic, covering eleven production-ready web applications, compared to the two small web applications and a undisclosed commercial portal used in [16]; (iii) the comparison framework in [16] is not publicly available, while we release ours to facilitate integration and evaluation of other RL-based crawlers.

VII. Conclusion

In this paper, we reviewed two state-of-the-art RL-based crawlers, WebExplor and QExplore, showing the limitations of their design choices with concrete examples. We then proposed MAK, a crawler based on the intuition that *less is more*: simplifying the application of RL to web crawling can enhance its exploration capabilities. Our crawler adopts a state-agnostic approach to learn the optimal navigation policy, formulating the crawling problem as an AdvMAB problem and employing an informative reward based on link coverage. Our experimental results show that MAK achieves broader and faster exploration than WebExplor and QExplore.

Future work will focus on assessing other RL-based solutions, such as generic GUI testing approaches, and integrating MAK within web scanners [6] to enhance web application testing and security assessments.

Acknowledgements: This research was supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

REFERENCES

[1] A. Petukhov and D. D. Kozlov, "Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing," in *Computing Systems Lab, Department of Computer Science, Moscow State University*, 2008, pp. 1–120.

[2] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 35–53, 2012. [Online]. Available: https://doi.org/10.1109/TSE.2011.28

[3] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A State-Aware Black-Box web vulnerability scanner," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 523–538. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe

[4] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling ajax-based web applications through dynamic analysis of user interface state changes," *ACM Trans. Web*, vol. 6, no. 1, mar 2012. [Online]. Available: https://doi.org/10.1145/2109205.2109208

[5] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow, "jÄk: Using dynamic analysis to crawl and test modern web applications," in *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan,November 2-4, 2015. Proceedings*, H. Bos, F. Monrose, and G. Blanc, Eds. Cham: Springer International Publishing, 2015, pp. 295–316. [Online]. Available: https://trouge.net/papers/jAEk_raid2015.pdf

[6] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black widow: Blackbox data-driven web scanning," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1125–1142. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00022

[7] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu, "Automatic web testing using curiosity-driven reinforcement learning," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 423–435. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00048

[8] S. Sherin, A. Muqeet, M. U. Khan, and M. Z. Iqbal, "Qexplore: An exploration strategy for dynamic web applications using guided search," *J. Syst. Softw.*, vol. 195, p. 111512, 2023. [Online]. Available: https://doi.org/10.1016/j.jss.2022.111512

[9] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[10] N. Savinov, A. Raichuk, D. Vincent, R. Marinier, M. Pollefeys, T. P. Lillicrap, and S. Gelly, "Episodic curiosity through reachability," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=SkeK3s0qKQ

[11] S. Singh, A. G. Barto, and N. Chentanez, "Intrinsically motivated reinforcement learning," in *Advances in Neural Information Processing Systems 17 [Neural Information Processing Systems, NIPS 2004, December 13-18, 2004, Vancouver, British Columbia, Canada]*, 2004, pp. 1281–1288. [Online]. Available: https://proceedings.neurips.cc/paper/2004/hash/4be5a36cbaca8ab9d2066debfe4e65c1-Abstract.html

[12] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," in *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2017, pp. 488–489. [Online]. Available: https://doi.org/10.1109/CVPRW.2017.70

[13] A. Stafeev and G. Pellegrino, "Sok: State of the krawlers - evaluating the effectiveness of crawling algorithms for web security measurements," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/stafeev

[14] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, "Gambling in a rigged casino: The adversarial multi-armed bandit problem," *Electron. Colloquium Comput. Complex.*, vol. TR00-068, 2000. [Online]. Available: https://eccc.weizmann.ac.il/eccc-reports/2000/TR00-068/index.html

[15] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, "The nonstochastic multiarmed bandit problem," *SIAM Journal on Computing*, vol. 32, no. 1, pp. 48–77, 2002. [Online]. Available: https://doi.org/10.1137/S0097539701398375

[16] Y. Fan, S. Wang, S. Wang, Y. Liu, G. Wen, and Q. Rong, "A comprehensive evaluation of q-learning based automatic web GUI testing," in *10th International Conference on Dependable Systems and Their Applications, DSA 2023, Tokyo, Japan, August 10-11, 2023*. IEEE, 2023, pp. 12–23. [Online]. Available: https://doi.org/10.1109/DSA59317.2023.00013

[17] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=rkE3y85ee

[18] "HotCRP," https://hotcrp.com/ [Accessed: 2/12/2024].

[19] "Matomo," https://matomo.org/ [Accessed: 2/12/2024].

[20] "Drupal," https://www.drupal.org/ [Accessed: 2/12/2024].

[21] "WordPress," https://www.wordpress.com [Accessed: 2/12/2024].

[22] "Blueprint," https://flask.palletsprojects.com/en/3.0.x/blueprints/ [Accessed: 2/12/2024].

[23] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. B. Abu-Ghazaleh, "Syzvegas: Beating kernel fuzzing odds with reinforcement learning," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. D. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 2741–2758. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/wang-daimeng

[24] "Qexplore implementation," https://github.com/salmansherin/QExplore [Accessed: 2/12/2024].

[25] "RL-based crawlers evaluation framework," https://doi.org/10.5281/zenodo.14276533 [Accessed: 23/04/2025].

[26] "Awesome-selfhosted," accessed: 27-Feb-2025. [Online]. Available: https://awesome-selfhosted.net/index.html

[27] "Xdebug," https://xdebug.org/, accessed: 2024-11-22.

[28] "coverage-node," accessed: 27-Feb-2025. [Online]. Available: https://github.com/jaydenseric/coverage-node

[29] B. Yu, L. Ma, and C. Zhang, "Incremental web application testing using page object," in *Third IEEE Workshop on Hot Topics in Web Systems and Technologies, HotWeb 2015, Washington, DC, USA, November 12-13, 2015*. IEEE Computer Society, 2015, pp. 1–6. [Online]. Available: https://doi.org/10.1109/HotWeb.2015.14

[30] S. Khodayari and G. Pellegrino, "JAW: studying client-side CSRF with hybrid property graphs and declarative traversals," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. D. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 2525–2542. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/khodayari

[31] E. Trickel, F. Pagani, C. Zhu, L. Dresel, G. Vigna, C. Kruegel, R. Wang, T. Bao, Y. Shoshitaishvili, and A. Doupé, "Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect SQL and command injection vulnerabilities," in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 2658–2675. [Online]. Available: https://doi.org/10.1109/SP46215.2023.10179317

[32] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 1388–1401. [Online]. Available: https://doi.org/10.1145/2976749.2978313

[33] K. Drakonakis, S. Ioannidis, and J. Polakis, "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*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1953–1970. [Online]. Available: https://doi.org/10.1145/3372297.3417869

[34] M. Steffens, C. Rossow, M. Johns, and B. Stock, "Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/dont-trust-the-locals-investigating-the-prevalence-of-persistent-client-\side-cross-site-scripting-in-the-wild/

[35] A. M. Fard and A. Mesbah, "Feedback-directed exploration of web applications to derive test models," in *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*. IEEE Computer Society, 2013, pp. 278–287. [Online]. Available: https://doi.org/10.1109/ISSRE.2013.6698880

[36] A. Abo-eleneen, A. Palliyali, and C. Catal, "The role of reinforcement learning in software testing," *Inf. Softw. Technol.*, vol. 164, p. 107325, 2023. [Online]. Available: https://doi.org/10.1016/j.infsof.2023.107325

[37] A. Fontes and G. Gay, "The integration of machine learning into automated test generation: A systematic mapping study," *Softw. Test. Verification Reliab.*, vol. 33, no. 4, 2023. [Online]. Available: https://doi.org/10.1002/stvr.1845

[38] Y. Fan, S. Wang, Z. Fei, Y. Qin, H. Li, and Y. Liu, "Can cooperative multi-agent reinforcement learning boost automatic web testing? an exploratory study," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 14–26. [Online]. Available: https://doi.org/10.1145/3691620.3694983

[39] S. Yu, C. Fang, X. Li, Y. Ling, Z. Chen, and Z. Su, "Effective, platform-independent gui testing via image embedding and reinforcement learning," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 7, Sep. 2024. [Online]. Available: https://doi.org/10.1145/3674728

[40] Z. Zhang, Y. Liu, S. Yu, X. Li, Y. Yun, C. Fang, and Z. Chen, "Unirltest: universal platform-independent testing with reinforcement learning via image understanding," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 805–808. [Online]. Available: https://doi.org/10.1145/3533767.3543292

[41] J. Eskonen, J. Kahles, and J. Reijonen, "Automating GUI testing with image-based deep reinforcement learning," in *IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2020, Washington, DC, USA, August 17-21, 2020*. IEEE, 2020, pp. 160–167. [Online]. Available: https://doi.org/10.1109/ACSOS49614.2020.00038