Link: Black-Box Detection of Cross-Site Scripting Vulnerabilities Using Reinforcement Learning

Soyoung Lee
School of Computing, KAIST

Seongil Wi
School of Computing, KAIST

Sooel Son
School of Computing, KAIST

ABSTRACT

Black-box web scanners have been a prevalent means of performing penetration testing to find reflected cross-site scripting (XSS) vulnerabilities. Unfortunately, off-the-shelf black-box web scanners suffer from unscalable testing as well as false negatives that stem from a testing strategy that employs fixed attack payloads, thus disregarding the exploitation of contexts to trigger vulnerabilities. To this end, we propose a novel method of adapting attack payloads to a target reflected XSS vulnerability using reinforcement learning (RL). We present Link, a general RL framework whose states, actions, and a reward function are designed to find reflected XSS vulnerabilities in a black-box and fully automatic manner. Link finds 45, 213, and 60 vulnerabilities with no false positives in Firing-Range, OWASP, and WAVSEP benchmarks, respectively, outperforming state-of-the-art web scanners in terms of finding vulnerabilities and ending testing campaigns earlier. Link also finds 43 vulnerabilities in 12 real-world applications, demonstrating the promising efficacy of using RL in finding reflected XSS vulnerabilities.

CCS CONCEPTS

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

KEYWORDS

reinforcement learning; cross-site scripting; penetration testing;

ACM Reference Format:
Soyoung Lee, Seongil Wi, and Sooel Son. 2022. Link: Black-Box Detection of Cross-Site Scripting Vulnerabilities Using Reinforcement Learning. In *Proceedings of the ACM Web Conference 2022 (WWW '22), April 25–29, 2022, Virtual Event, Lyon, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3485447.3512234

1 INTRODUCTION

Over a decade of research, the identification of reflected cross-site scripting (XSS) vulnerabilities has evolved. Prior research has proposed static methods [27, 32, 33, 39, 73], dynamic detection [21, 51], hybrids of static and dynamic methods [3, 4, 5, 10, 70], symbolic execution [2, 37], and penetration testing [58, 63]. Unfortunately, reflected XSS vulnerabilities remain prevalent on the Web [48].

In practice, black-box web scanners have been heavily used to detect reflected XSS vulnerabilities owing to their operational merits; they require no access to source code nor any changes to the run-time environments of target web applications.

A common testing strategy of such off-the-shelf black-box scanners is to inject a series of payloads in their attack dictionary and then check whether any of the injected payloads indeed trigger XSS vulnerabilities. However, this straightforward strategy suffers from several limitations: (1) penetration testing campaigns take a long time to complete due to the brute-forcing of all payloads in the attack dictionaries; and (2) they disregard the output contexts in which injected input values appear, thus producing false negatives.

Contributions. To overcome these limitations, we present a novel approach of adapting attack payloads to a target reflected XSS vulnerability. We suggest leveraging reinforcement learning (RL) to adapt attack payloads while observing the responses of previous attack attempts. To demonstrate the fidelity of leveraging RL, we design and implement Link, a fully automatic web scanner using an RL agent designed to identify reflected XSS vulnerabilities.

One technical challenge of applying RL is that it is difficult to define states, actions, and a reward function that contribute to the convergence of an RL agent in addressing real-world problems [16]. Another challenge is to train a transferable RL agent when the training and testing environments are different. That is, an RL agent trained on training benchmarks should be effective in finding reflected XSS vulnerabilities in real-world applications.

To address the former challenge, we explore and propose RL algorithms, states, actions, and a reward function for an RL agent, which fit into finding XSS vulnerabilities with high performance. For the latter one, we design states and actions that are general enough to model how an adversary adapts its payloads to a targeted real-world website in a black-box manner. Specifically, we propose a way of encoding the output context around a reflected payload into a state consisting of 47 features. We also suggest seven payload generation and 32 mutation rules for RL actions, enabling the composition of sophisticated payloads. We further present a reward function that enables an RL agent to automatically learn an optimal policy, avoiding implementing vendor-specific heuristics.

We evaluated Link's efficacy in finding reflected XSS vulnerabilities in Firing-Range, OWASP, and WAVSEP benchmarks. Link found 343 vulnerabilities with no false positives, outperforming other existing web scanners, including Burp Suite, Wapiti, ZAP, and Black Widow. The number of requests necessary to identify these vulnerabilities is lower than with other state-of-the-art tools, reducing the overall testing campaign time. We emphasize that Link's ability to adapt attack payloads helps shorten the testing time. When applying Link to finding XSS vulnerabilities in 12 real-world applications, Link reported 43 vulnerabilities.

In summary, we demonstrate that it is feasible to leverage RL to accurately identify reflected XSS vulnerabilities in a black-box manner. We present effective methods of defining the states, actions,

WWW '22, April 25–29, 2022, Virtual Event, Lyon, France
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9096-5/22/04…$15.00
https://doi.org/10.1145/3485447.3512234

```
1 <?php
2 // Exploit Payload:
3 //   '></textarea><scrscriptipt>alert(1);</scrscriptipt>//
4   $value = $_GET["value"];
5   $value = str_replace("script", "", $value);
6   echo "<textarea attribute='" . $value . "'></textarea>"; ?>
```

Figure 1: An XSS vulnerability due to incorrect sanitization.

and a reward function that contribute to making the Link’s RL agent converge with high performance in a fully automatic way.

2 REINFORCEMENT LEARNING

Reinforcement learning is a machine learning algorithm that trains an agent to learn a policy of determining optimal actions in a given environment. The agent’s goal is to learn an optimal policy that maximizes the expected reward [34]. Specifically, an agent performs actions and observes changes in the environment E over a number of discrete time steps [41]. At each time step t, the agent receives a state s_t from E and selects an action a_t from the set of possible actions A according to its policy π. For the selected a_t, the agent computes an expected reward R_t and then derives the next state s_{t+1} via interacting with E.

A reward r_t represents a scalar value that the agent gets at the given time step t. R_t summarizes all rewards r_t from the given time step t to the terminal time step T with discount factor γ: $R_t = \sum_{i=0}^{T} \gamma^i r_{t+i}$. The agent finds an optimal policy π that maximizes R_t when following π for each state s_t. Thus, $\pi(a|s)$ is often a mapping function that maps from a given state s to an action a or a probability distribution over actions across A, given s.

Actor-Critic algorithm. The REINFORCE algorithm parameterizes the policy function with neural networks $\pi_\theta(a|s)$ and updates the networks via conducting gradient ascent on $E_\pi[R_t]$ [67]. However, computed gradients are often noisy and have high variance, causing instability and slow convergence when deriving the optimal policy. A2C [41] addresses this problem by extending the baseline approach [36]. For the objective function that the agent aims to maximize, A2C leverages the difference between $Q(s, a)$ and $V(s)$ that model a quality and value function, respectively. The gradient ascent on this difference tends to produce a low variance, thus enabling fast convergence. While maximizing the objective function, A2C trains actor and critic networks together. The actor networks are trained to select an action at a given state, and the critic networks are trained to evaluate the agent’s action.

3 MOTIVATION

A reflected XSS vulnerability exploits a bug that allows for an adversary to inject malicious JavaScript (JS) code into a target web server’s response (i.e., webpage). The principled approach to preventing this type of vulnerability is to enforce correct sanitizers [29, 74]. Unfortunately, in practice, developers’ mistakes of omitting sanitizers or implementing incorrect ones result in prevalent XSS vulnerabilities on the Web [5, 48]. Existing black-box web scanners have been applied to find reflected XSS vulnerabilities [40, 46, 53, 60, 66, 75]. A common testing strategy of these tools is to inject their payloads and then to determine whether the injected payloads indeed trigger vulnerabilities. However, the existing black-box scanners share two drawbacks: (1) unscalable testing campaigns and (2) context-unaware payload generation.

Unscalable testing campaigns. Most web scanners, including Wapiti [66] and XSSer [75], are agnostic to the innerworking of a target website. They brute-force web attacks by throwing a stream of predefined payloads, expecting one of them exploits inherent vulnerabilities. Unfortunately, this strategy contributes to increasing the number of attack requests, thereby resulting in long testing campaigns. For example, XSSer uses a payload dictionary that consists of 1,293 predefined payloads to find reflected XSS vulnerabilities. When the payload that triggers a target vulnerability is at the end of this dictionary, it would have attempted 1,292 futile payloads. This is not a scalable approach to detecting reflected XSS vulnerabilities.

Context-unaware payload generation. Black-box web scanners do not adapt their payloads to a target vulnerability, producing false negatives, which could have been triggered by adjusting the payloads. Specifically, they ignore the context in which a reflected XSS payload appears in the output HTML webpage, attempting ineffective payloads that do not trigger XSS vulnerabilities.

Figure 1 shows an example of such a reflected XSS vulnerability. This PHP application incorrectly sanitizes user input via `$GET["value"]`. It attempts to remove all script tags by removing the `"script"` string constant in Line (Ln) 5. An attacker is able to bypass this filter by injecting overlapped string values (e.g., `<scrscriptipt>alert(1);<scrscriptipt>`). However, this payload alone cannot trigger the vulnerability since it would appear as a `textarea` attribute in Ln 6, not as a JS script element. Therefore, the attacker should inject both the single quotation mark and the closing `textarea` tag, as the payload in Ln 3 shows, which requires adjusting the original payload of `<script>alert(1);</script>`.

We emphasize that Buyukkayhan *et al.* [7] reported a recent trend of increased sophistication in attack payloads to trigger XSS vulnerabilities. Their findings demonstrate that the surrounding output context in which a reflected input appears determines an attack payload to trigger XSS vulnerabilities, which requires context-aware payload generation.

Motive. The aforementioned limitations have remained as open problems for black-box web scanners. To overcome those, we propose leveraging an RL agent to adapt attack payloads to a target context by analyzing the responses of previous attack attempts. We envision a fully automated RL agent that attacks a target website by adapting attack payloads to discover reflected XSS vulnerabilities.

3.1 Technical Challenges

Finding reflected XSS vulnerabilities using RL entails two challenges: (1) a non-converging RL agent and (2) a transferable RL agent to find XSS vulnerabilities in real-world applications.

Non-convergence problem. Reinforcement learning has been deployed in diverse domains, including arcade games [42], block-chain attacks [30], and robotics [52], demonstrating its effectiveness in various real-world settings. Unfortunately, the application of RL to real-world problems is known to be challenging due to non-converging RL agents [17]. This is primarily due to poorly defined actions, states, and a reward function that model target real-world problems [43, 50]. Therefore, a technical challenge here is to properly model actions, states, and a reward function that contribute to converging a high-performing RL agent that is able to find XSS vulnerabilities [12, 18].

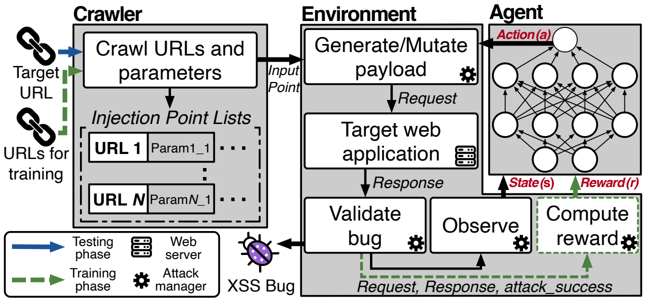

Figure 2: Overview of Link architecture.

There have been a few previous studies of applying RL to find web vulnerabilities, including SQL injection [19] and reflected XSS vulnerabilities [8]. However, these studies have limitations of requiring human involvement in the agent training and testing. Erdodi *et al.* [19] defined a limited set of actions and states for their RL agent to discover SQL injection vulnerabilities, covering a small set of synthetic SQL injection scenarios. Suggester [8] required human expert's involvement in both the RL training and testing steps. The involvement is designed to facilitate the convergence of their RL agent by teaching the agent the expert's decisions. However, this manual involvement hinders automatically finding vulnerabilities and diminish the needs of using RL.

Transferable RL agent. Previous studies of applying RL have in common that the training and testing environments of an RL agent are the same [42, 59]. Mnih *et al.* trained their agent on a specific Atari game and used this agent to play the same game [42]. However, for effective testing campaigns, an RL agent trained on training benchmarks should find XSS vulnerabilities in real-world web applications; we believe that auditors do not want to train their agents from scratch on their websites. In this regard, we argue that states and actions for an RL agent should be general enough to cover various types of real-world reflected XSS vulnerabilities.

Our approach. In this paper, we present Link, an RL agent for detecting reflected XSS vulnerabilities with precision and efficiency. Link tackles the two technical challenges aforementioned. For the former challenge, we define a series of actions and states that are designed to achieve stable convergence without the need for human assistance or the need to assume only a subset of the XSS attack scenarios. For the latter challenge, we demonstrate several ways of observing responses, including the shape of reflected payloads and their surrounding context, which contribute to generalizing the trained agents to operate automatically in real-world applications.

4 OVERVIEW

Link is a penetration testing tool designed to find reflected XSS vulnerabilities. Link has two phases: *training* and *testing*. In the training phase, Link takes in a large number of URLs, each of which refers to a web application with a reflected XSS vulnerability. Using this set of websites, Link trains an RL agent to learn an optimal policy for generating an attack payload to adapt to a given environment. This training phase is a one-time setup procedure. With the guidance of the trained agent, the testing phase conducts black-box penetration testing against a given URL of the target website. Once a testing campaign for the target URL is over, Link reports functional requests that succeeded in exploiting reflected XSS vulnerabilities.

Figure 2 illustrates the architecture of Link. It consists of three components: crawler, environment, and agent. The crawler collects all transitively connected websites, thus identifying subdomain URLs and their input parameters to use in injecting attack payloads. For each pair of an URL and an input parameter, the agent and the environment work in tandem to perform testing (or training) via sending attack requests with payloads. The environment provides interfaces for the agent to observe changes respective to the agent's actions. The agent determines an action to conduct based on observations of the environment.

Crawler. Given URLs, Link starts by crawling their subdomain URLs, which become targets for penetration testing. In particular, Link visits each one of the given URLs, parses its HTML response, and then extracts all link elements for the next URLs to visit. Link iterates this process until there are no remaining URLs to visit.

For each identified URL, Link collects input parameters for the agent to use for injecting payloads. It collects input parameters from GET requests by extracting key attributes in their query strings. Also, when visiting each webpage, the crawler retrieves all attribute values of `input` tags in HTML form elements, which become input parameters for POST requests to perform penetration testing.

Note that computing a sound set of input parameters is beyond the scope of this paper. Link focuses on conducting efficient penetration testing via adaptively changing its attack payloads. We revised Wapiti [66], an open-source web vulnerability scanner, to leverage an RL agent in adapting attack payloads. We extended the crawler of Wapiti to identify target URLs and input parameters; specifically, we included Referer header and URI-based injection for additional input sources.

Environment. The environment consists of two components: web server and attack manager. The web server with a given URL runs the target application. The attack manager is an abstraction layer of the web server that provides interfaces for the RL agent to observe changes due to the agent's actions.

The attack manager is designed to support four major operations: (a) generating or mutating attack payloads based upon the RL agent's actions, (b) checking whether an attack request succeeds in triggering a reflected XSS vulnerability, (c) extracting observation information from the response to an attack request, and (d) computing a reward from observation information.

Agent. The agent conducts the core RL functionality. In the training phase, it takes in configurable parameters, including learning rate and discount factor, as input and learns an optimal policy of generating an attack payload to adapt to a given environment. In the testing phase, the trained agent adaptively selects an action a_t given the current state s_t of the environment.

5 DESIGN

5.1 Link Workflow

Algorithm 1 illustrates the overall training and testing procedures that Link conducts to identify reflected XSS vulnerabilities.

Training. The training algorithm (Ln 1) starts with the target set of URLs and input parameter pairs (*tgt_set*) that the crawler identified (§4). This algorithm then iteratively performs the `Episode` function until *total_step* reaches *MAX_STEPS*. For each iteration, which we call an *episode*, it sequentially selects a URL and an input parameter pair in *tgt_set* (Ln 4) and trains the RL agent (i.e., *agent*) to generate an exploit for this pair via conducting `Episode()`. Note that *MAX_STEPS* is a hyperparameter that determines the number of total steps on which the RL agent should be trained. We set the default value of *MAX_STEPS* to be 3.5M.

In the `Episode` function (Ln 12), the agent performs a series of actions a given number of times (*STEPS_PER_EPISODE*) for the selected URL and input parameter pair. Specifically, for each target pair, Link initiates a training campaign episode. For each episode, Link first checks whether a given initial payload appears in the target webpage. For this, Link sends an HTTP(S) request with a valid input payload of `[Initial Payload]` in Ln 15. `[Initial Payload]` is an alphanumeric string having special characters (e.g., `">/'>injectionhere1234`). It then checks whether this particular string is included in the response in Ln 16. We prepared four initial payloads to perform this testing. If Link cannot confirm the presence of any of the initial payloads, it strongly indicates that attacking the current pair of an input parameter and a target URL cannot trigger a vulnerability. Therefore, Link immediately terminates the current episode and starts a new episode on a different pair of a URL and an injection parameter (Ln 18).

If the payload is present, Link parses its response and checks how this valid input string appears on the webpage. In Ln 19, Link computes injection point features at the first appearance position of the current payload (§5.3). After this initial process of computing injection point features, Link executes a training loop in Lines (Lns) 20–28, which trains the RL agent. This training process iterates multiple steps, and each step becomes an execution unit for which the RL agent performs an action.

For each step, the agent selects an action based on observations in Ln 21. The attack manager then generates or mutates an attack payload corresponding to the selected action a_t (§5.2) in Ln 22. The attack manager sends an attack request with this payload and verifies whether the sent request succeeds in triggering a vulnerability (§5.4) in Lns 23–24. Link then observes this response and computes a feature vector of s_{t+1} in Ln 25 (§5.3). In the training phase, Link also computes a reward (§5.4) and trains the agent to learn an optimal policy that maximizes the expected reward in Lns 26–27. When either (1) the attack is successful or (2) the current step reaches *STEPS_PER_EPISODE*, the agent terminates the current episode.

Testing. Link leverages the `Test` function to find XSS vulnerabilities in a given *tgt_set*. This phase is analogous to the innerworking of the `Train` function except that it inspects every pair in *tgt_set* (Ln 9). For `Episode` in the testing phase, Link does not execute the highlighted code since Link does not need to train the RL agent via computing expected rewards.

RL algorithm. When training *agent* as shown in Ln 27, we leverage Advanced Actor-Critic (A2C) [41]. We train two networks: actor and critic. The actor network maps each state s to an action a, outputting a probability distribution $\pi_\theta(a|s)$. The critic network maps each state s to a quality value by computing $V_\theta(s)$. A2C trains $\pi_\theta(a|s)$ and $V_\theta(s)$ together in the direction of maximizing an advantage function, which denotes the advantage degree of taking a specific action a relative to the quality value of being at state s.

We define our advantage function as $A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$, where $Q(s_t, a_t)$ emits the quality value of taking a_t at s_t, and $V(s_t)$ computes the quality value of being s_t. Considering $Q(s_t, a_t) \simeq r_t + \gamma V(s_{t+1})$, the advantage function can be defined as follows:

$$A(s_t, a_t) = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{1}$$

Here, r_t is a reward for the action a_t that the agent chooses based on the current state s_t. In other words, r_t represents the degree to which a_t is appropriate given s_t to trigger an XSS vulnerability. The trained $V_\theta(s)$ indicates how good s_t is to find target XSS vulnerabilities, and $\pi_\theta(a|s)$ provides a probability distribution of actions, denoting which mutation or generation rule a_t should be favored given s_t to maximize the total reward R_t.

To train the actor and critic networks, A2C uses the objective function J and computes its gradient ascent as follows:

$$J(\theta) = E[\sum_{t=0}^{T} \gamma^t r_t | \pi_\theta],\ \nabla J_\theta = \sum_{t=0}^{T} \nabla_\theta log \pi_\theta(a_t|s_t) A(s_t, a_t) \tag{2}$$

For each networks, we used fully-connected neural networks having three layers, each size of which is 128. We used the root mean square propagation (RMSProp) optimizer to train the networks.

Training automation. Link automates the training procedure without human involvement. Caturano *et al.* [8] used a human-in-the-loop approach in which their system users should analyze attack results and compute the current state. By contrast, Link leverages the bug oracle (§5.4) that checks the reflection of attack payloads and attack success, removing human involvement.

Algorithm 1: Link Workflow (Highlighted codes are only executed during training).

```
1  function Train($tgt\_set$)
2      $total\_step \leftarrow 0$
3      while $total\_step < MAX\_STEPS$ do
4          $url, param \leftarrow$ Select ($tgt\_set$)
5          $found, payload, step \leftarrow$ Episode($url, param$)
6          $total\_step \leftarrow total\_step + step$
7  function Test($tgt\_set$)
8      for $url,\ param \in tgt\_set$ do
9          $found, payload, step \leftarrow$ Episode($url, param$)
10         if $found$ then
11             ReportBug($payload, url, param$)
12 function Episode($url, param$)
13     // Initialize the $payload$ and the current step $t$
14     $payload \leftarrow$ [Initial Payload], $t \leftarrow 0, found \leftarrow false$
15     $response \leftarrow$ SendReqest($url, param, payload$)
16     if !CheckReflected($payload, response$) then
17         // If $response$ does not reflect $payload$, return $false$.
18         return $false,\ payload$
19     $s_t \leftarrow$ Observe($payload, response, found$)
20     while $t < STEPS\_PER\_EPISODE$ and $!found$ do
21         $a_t \leftarrow agent$.GetAction($s_t$) // Get an action $a_t$ given $s_t$.
22         $payload \leftarrow$ MutatePayload($payload, a_t$)
23         $response \leftarrow$ SendReqest($url, param, payload$)
24         $found \leftarrow$ BugOracle($payload, response$)
25         $s_{t+1} \leftarrow$ Observe($payload, response, found$)
26         $r_t \leftarrow$ Reward($t, s_{t+1}, payload, found$)
27         $agent$.Train($s_t, a_t, s_{t+1}, r_t$)
28         $t \leftarrow t + 1$;
29     return $found,\ payload,\ t$
```

5.2 Generating/Mutating Payload

Link supports two action types: *generation* and *mutation*. Table 4 in Appendix 10.2 shows a total of 39 actions that the Link agent can take to generate attack payloads. These actions are categorized into two groups: seven actions of generation and 32 actions of mutation. To define these actions, we investigated attack payloads in

Firing-Range [26], WAVSEP [9] and benchmarks from sanitization bypassing techniques [13, 49, 54, 55]. These benchmarks list known exploitation techniques and common vulnerable logic. We generalized unit actions so that their combinations are able to generate all the exploits in these benchmarks.

Generation. Link has two action groups: basic payload and JS component generations. For the basic payload generation, the attack manager generates a payload using HTML elements that embed JS snippets, such as script tags, media tags, and event attributes. For example, for the second action in this group, Link generates the payload `<img src=‘x’ onerror=‘alert(1);’/>`, injecting the image tag with JS code. Note that only one payload generation rule is randomly selected when there are multiple application rules for a chosen action. For example, when the attack manager performs the second action, one of the four media tags (i.e., `img`, `video`, `audio`, and `svg`) is randomly selected for the payload generation.

For the JS component generation, Link prepares an attack JS snippet without any HTML elements that enable the execution of this JS snippet. This category of generations is designed to address cases in which the injection point is within a script execution environment (e.g., `<script>[payload]</script>`).

Mutation. The attack manager mutates components in the previous attack payload according to a mutation action a_t that the agent selects. We prepared six groups of mutation actions: *prefix*, *suffix*, *tag*, *attribute*, *JS snippet*, and *entire string* changes. Each action is designed to (1) escape the echoing context of a reflected payload (actions 8–20, 21, 22–24) and (2) evade incorrect sanitization (actions other than the aforementioned).

If the previous attack payload does not have components representing the category of the current action a_t, the attack manager does not apply the action to the payload (i.e., Line 22 of Algorithm 1 is skipped). For example, if the agent observes the `<script>alert(1);</script>` payload and then selects the 28th action of the attribute category, the attack manager does not apply this action due to the absence of an attribute and sends a request with the same payload. In this case, the attack manager decreases the reward because the same payload is used repeatedly (§5.4).

We note that actions across different categories are combined over multiple steps. However, actions within the same category are substituted. For example, given the 1st, 22nd, and 23rd actions, Link computes a payload of `<script>alert(1);</script>'` because the 22nd and 23rd actions belong to the same category.

5.3 Observations

We model a state (s) as a feature vector ($f_1, f_2, ..., f_n$), each element of which is a scalar value that represents the respective feature's existence or its type. Table 5 in Appendix 10.2 enlists all the 47 features that the attack manager observes. These features are categorized into five groups: *payload appearance*, *payload repetitiveness*, *reflected payload appearance*, *payload context*, and *attack result*.

Payload appearance. This category of 29 features is designed to check the presence of particular strings in an attempted payload. Each selected string represents a way of injecting a JS snippet; by checking its presence, Link checks which injection method has been used for the RL agent to derive the next state. Specifically, these features check whether a current payload has a certain tag (e.g., `<script>` and `<img>`), contains a JS snippet (e.g., `alert(1)` and `confirm(1)`), or uses capitalized tags. Each feature value is 0 or 1, indicating the presence or absence of the corresponding string.

Payload repetitiveness. This category of features represents the repetitiveness of payloads, which contributes to the agent avoiding using the same attack payload. The 30th feature represents frequency of the current payload has been used in the current episode. When this feature is positive, the attack manager gives a negative reward for a selected action (§5.4). The 31st and 32nd features represent the previous action index at a_{t-1} and the current action index at a_t that the agent has chosen.

Reflected payload appearance. The features in this category describe how the attempted payload is reflected in its response. The 33rd feature indicates whether a script tag (i.e., `<script>`) or media tag (e.g., `<img>`) contained in the current payload appears as the same tag in the response. For this, the attack manager parses the response and then checks for the presence of the same tag at the injection point. Link is able to accurately identify where the injection point is through the episode initialization process in Lns 14–19.

The 34th feature represents whether a JS snippet (e.g., `alert(1)`), JS URL, or JS pseudo protocol in the payload appears in the response. The 35th feature indicates the string similarity between the attempted payload and the reflected payload in the response. For similarity matching, we used a gestalt pattern matching algorithm in difflib [23]. For example, if the generated payload is `<script>alert(1);</script>` and the reflected payload is `<>alert(1);</>`, the 33rd, 34th, and 35th features become -1, 1, and 3.5, respectively. Even if the XSS vulnerability is not triggered via the current action a_t, these features reveal that the reflection of the payload is partially achieved.

Payload context. This category of features represent the context of an injection point in the response, which Link identified in the initialization step in Ln 19 (§5.1). We note that these features are designed to collect context information around an injection point.

The 36th feature indicates the response type among three formats (i.e., HTML, CSS, and JSON). The 37th feature specifies the default value type of the input parameter. To collect this information, the attack manager leverages the crawling results to check whether the parameter has a default value. The 39th to 41st features address the information necessary for pre-escape and post-escape of the injection point. The last feature in this category is the reflection of a valid payload for the decision to terminate the current target pair.

Attack result. This feature indicates whether the vulnerability is triggered by the action a_t by checking the information passed by the bug oracle (§5.4).

5.4 Computing Reward

The attack manager computes a reward r_t for an action a_t that the agent chooses based on the current state s_t. The current action a_t governs the next payload to attempt and affects the next state s_{t+1}. The reward r_t takes into account the next state s_{t+1} to evaluate whether a_t is a good action that maximizes the expected reward.

Algorithm 2 in Appendix 10.3 describes how the attack manager computes an r_t. We designed two types of rewards; (1) a positive reward to indicate the degree to which a testing campaign ends early and (2) a negative reward of indicating the repetitiveness of a chosen action. Specifically, when a_t enables successful exploitation, the reward becomes the difference between the number of maximum steps (*STEPS_PER_EPISODE*) to the current *t*, thereby motivating the agent to find a proper attack payload as early as possible. By contrast, Link decreases r_t by one when a_t is the same as a_{t-1} or when the current payload equals the previous payload, instructing the agent to avoid the same action.

Bug oracle. The `BugOracle` function monitors the reflected payloads in the responses to determine whether the input payloads successfully trigger the reflected XSS vulnerabilities. We designed `BugOracle` to check the presence of actual injected JS snippets, instead of reflected futile HTML tags, thus decreasing false positives. In particular, the validation algorithm differs depending on the last attempted action in the generation category (actions 1–7). For example, when the agent conducts the 1st, 20th, and 25th actions, the oracle uses the validation technique for the 1st action. The reason is that while all mutation actions are designed to either bypass incorrect sanitization logic or escape confined input values, each generation action determines how the injected JS code is executed.

Actions 1–2. The first and second actions are designed to inject script and media tags, respectively. For the first action, the bug oracle checks for the presence of script tags with the JS snippet `alert(1);`. For the second section, it checks the presence of media tags with the attribute-value pair of `onerror=alert(1);`.

Actions 3–4. The third and fourth actions are designed to handle payload reflections in the form of `<[tag][payload]`. The bug oracle identifies which tag is used ahead of the injection point. It then checks whether this tag has an attribute-value pair of which the value has `alert(1);` and attribute is among `onerror`, `onclick`, `onmouseover`, and `onload`. Otherwise, it checks whether the `src` value of this tag is `http://attack.js`.

Actions 5–7. Actions 5–7 are designed to inject JS code in the form of `<[tag][attribute]=[payload]` or `<script>[payload]</script>`. To check for the former, the attack manager fetches the `[tag]` at the injection point. It then checks whether this identified tag has an event attribute (i.e., `onerror`, `onload`, `onclick`, or `onmouseover`) for action 5 or a URL attribute (`src` or `href`) for actions 6–7. It then confirms whether the payload is inserted into the value of this attribute. To validate the latter, the attack manager scans for each script tag across the entire response and then checks if the content of the identified script tag contains the payload.

6 EVALUATION

We evaluate Link on finding reflected XSS vulnerabilities by comparing it against existing scanners (§6.2). We also analyze the capability of Link to find reflected XSS vulnerabilities in 12 real-world PHP applications (§6.3). Appendix 10.4 describes the degree to which hyperparameters and RL algorithms contribute to Link's capability of finding bugs. Finally, we present a case study in Appendix 10.5.

6.1 Experimental Setup

We conducted experiments on one desktop machine running 64-bit Ubuntu 18.04.4 with an Intel i7-8700 CPU (12 cores) and 32 GB of main memory.

Benchmarks. We selected the benchmarks of server-side applications with reflected XSS vulnerabilities from Firing-Range [26], WAVSEP [9], and OWASP benchmarks [47]. Note that these benchmarks have been used to test the capability of web scanners in finding various security bugs. Among the applications in these benchmarks, we selected applications with at least one reflected XSS vulnerability. Our benchmarks include 351 server-side applications in PHP and Java Server Pages (JSP); 45, 60, and 246 applications are from Firing Range, WAVSEP, and OWASP, respectively.

In addition, we prepared the *Filter Evasion* benchmarks, which include 25 of our own applications. Each one is designed to require a sophisticated payload to evade its input sanitization and trigger a reflected XSS vulnerability. We referenced previously known incorrect sanitization filters, evasion techniques [13, 49, 54, 55, 62], and vulnerable applications due to incorrect sanitization, including 4Images (CVE-2009-2131) [1] and UliCMS (CVE-2019-11398) [69].

We also prepared 12 PHP web applications listed in the first column of Table 3 to demonstrate Link's efficacy in finding vulnerabilities in real-world applications (§6.3). We selected these applications from the two sources: (1) open-source web applications with XSS vulnerabilities that a Netsparker scanning engine [44] identified; and (2) evaluation datasets from previous studies [25, 37].

Training and testing. To train the agent, we used Firing-Range, WAVSEP, and Filter Evasion benchmarks. We used Proximal Policy Optimization (PPO) [61], Deep Q-Network (DQN) [42], and A2C [41] training algorithms for comparative evaluation (§10.4) and set up A2C as the baseline for Link. For the remaining evaluations, we trained the model with the hyperparameters set to 3.5M *MAX_STEPS*, 500 *STEPS_PER_EPISODE*, a 0.95 discount factor, and a 0.0005 learning rate, which yielded the best performance (§10.4). The total time required for training the agent was 39.5 hours. In the testing phase, we empirically set the *STEPS_PER_EPISODE* to 30, restricting the agent to try a limited number of attack attempts.

6.2 Comparison to Existing Web Scanners

We compared Link's ability to find reflected XSS vulnerabilities in the Firing-Range, WAVSEP, OWASP, and Filter Evasion benchmarks against four state-of-the-art web scanners: Burp Suite Pro (v2021.6.2) [53], Wapiti (v3.0.5) [66], OWASP ZAP (v2.10.0) [46], and Black Widow (i.e., BW) [20]. We also setup Link with a random policy agent (i.e., Link-R) that selects random actions instead of the agent's actions to demonstrate the agent's efficacy. For a fair comparison, we set the other tools to find only reflected XSS vulnerabilities, preventing them from sending requests for other types of vulnerabilities. Since Link leverages a stochastic policy, for each experimental setup, we conducted five testing campaigns and reported median values.

Table 1 summarizes the experimental results. Link found 343 true positives (TPs) (91.2%) with 33 false negatives (FNs) in all the benchmarks, achieving the best performance compared to all the other tools in terms of maximizing TPs and minimizing FNs.

TPs. Among the 343 TPs, we analyzed 181 TPs that other tools were unable to identify, reporting them as FNs. The reported superior performance of Link stems from three reasons: (1) Link was able to generate appropriate exploit payloads that other tools were unable to compose; (2) Link leveraged the more accurate bug oracle to identify the reflected payloads; (3) the other tools skipped certain types of input parameters due to their own heuristics.

Benchmark	# of Bugs	Tools	TP	FP	FN	Time	# of Attempts
Firing-Range	45	**Link**	**45**	0	0	8s	459
		Link-R	20	0	25	55s	1,351
		Wapiti	37	0	8	8s	447
		Burp	38	0	7	20s	4,344
		ZAP	43	0	2	17s	1,530
		BW	23	0	22	39s	218
WAVSEP	60	**Link**	**60**	0	0	5s	718
		Link-R	28	0	32	55s	1,871
		Wapiti	30	0	30	11s	1,782
		Burp	54	0	6	5s	13,859
		ZAP	52	0	8	11s	4,717
		BW	13	0	47	5m 36s	577
OWASP Benchmark	246	**Link**	**213**	0	33	1m 17s	11,912
		Link-R	178	0	68	5m 47s	16,457
		Wapiti	137	0	109	59s	6,451
		Burp	186	0	60	1m 58s	121,311
		ZAP	186	0	60	1m 30s	29,483
		BW	157	0	89	137m 44s	10,759
Filter Evasion	25	**Link**	**25**	0	0	4s	334
		Link-R	6	0	19	30s	885
		Wapiti	17	0	8	3s	505
		Burp	22	0	3	1s	1,852
		ZAP	6	0	19	5s	857
		BW	12	0	13	8m 4s	602
Total	376	**Link**	**343**	0	**33**	1m 34s	13,423
		Link-R	232	0	144	8m 7s	20,564
		Wapiti	221	0	155	1m 21s	9,185
		Burp	300	0	76	2m 24s	141,366
		ZAP	287	0	89	2m 3s	36,587
		BW	205	0	171	152m 3s	12,156

Table 1: Comparison with state-of-the-art tools.

Table 2 presents the root causes of FNs that other tools reported. We observed that other scanners miss many vulnerabilities due to reason (1) above, which demonstrates that the ability of the RL agent to generate input payloads greatly contributes to finding many vulnerabilities. We also note that Wapiti produced 56 FNs due to reason (3). This result shows that although Link is implemented on top of Wapiti, the Link's input parameter selection algorithm greatly contributes to finding more vulnerabilities.

We emphasize that Link's performance stands out compared to the other scanners regarding Filter Evasion benchmarks. This stems from the effectiveness of the agent, which checks how the input payload is reflected in the response and suggests optimal actions to bypass incorrect sanitization logic. On the other hand, the other scanners exhibit relatively low performance because they do not have functional exploit payloads in their dictionary to identify FNs. One solution is to put these payloads into the dictionary of the existing scanners, but these payloads are uncommon as they are specific to the target benchmarks. This eventually results in an unscalable scanning campaign.

When comparing the performance differences between Link and Link-R, we observed that the RL agent contributed to finding 111 additional vulnerabilities (343 for Link vs. 232 for Link-R) while sending 7,141 fewer requests. These differences clearly demonstrate that the agent learned to generate effective payloads in a context-aware manner. We further describe a case study of describing Link's behaviors to find an XSS vulnerability in Appendix 10.5.

FNs. Link reported 33 FNs. For 33 FNs of these, the corresponding applications used JS snippets to dynamically construct and send requests, enabling Link to miss extracting input parameters to inject. We note that all scanners used in the experiment, including Link, identify input parameters by parsing only HTML responses during the crawling process, thus reporting the same FNs.

Reason	Wapiti	Burp	ZAP	BW	Total
Reason (1): ineffective input payloads	53	29	49	73	204
Reason (2): incomplete bug oracle	13	6	7	0	26
Reason (3): unidentified input parameters	56	8	0	65	129
Total	122	43	56	138	181

Table 2: Vulnerabilities that other scanners report as FNs but Link judges to be TPs (several bugs can be treated as FNs by multiple tools.)

Performance. The seventh and eighth columns of Table 1 present the execution time and the number of attempted requests for each tool, respectively. We observed that Wapiti had the lowest execution cost because it has a small number of predefined payloads and skips certain types of input parameters. These limited payloads and parameters significantly undermine Wapiti's ability to find vulnerabilities compared to other scanners. Black Widow focused on maximizing the crawling coverage rather than generating appropriate payloads. It thus uses a small size of the payload dictionary, which results in reporting a small number of TPs. We note that Link surpassed other techniques in terms of the execution times, number of requests, and number of bugs found. Link required only 13,423 requests to identify 343 TPs, sending much fewer requests than did the second best performant scanner (i.e., Burp). We also observed that Wapiti and Burp sent many attack requests with different payloads for several applications. By contrast, Link generated working payloads within a few attempts. For instance, for a Filter Evasion benchmark, Link, Wapiti, and Burp attempted 9, 37, and 26 different payloads, respectively. For this application, ZAP reported a false negative. This means that Link is capable of generating appropriate payloads without brute-forcing all payloads in the attack dictionary.

Impact of observation features. We conducted evaluations to assess the importance of each observation feature. For each feature, we enforced Link not to use it by fixing its value to the initial value and then measured the number of TPs. We observed that the 32nd, 7th, 41st, and 38th features contributed most to decreasing TPs by 48%, 18%, 12%, and 9%, respectively. The current action, the presence of a JS snippet in the payload, escaping characters, and the injection point type play important roles in finding TPs.

6.3 Discovering Real-World Vulnerabilities

We evaluated the Link's capability of finding vulnerabilities in 12 real-world applications with 49 XSS vulnerabilities. As in Section 6.2, in this experiment, we compared the detection capability of Link against four existing web scanners with a time budget of three hours. Table 3 and Figure 3 show the detection results and the number of requests that each tool attempted, respectively. Link had the second highest number of TPs (i.e., 43 vs. 33 for Wapiti, 46 for Burp, 36 for ZAP, and 25 for Black Widow), while reporting the second lowest number of attack requests (i.e., 4,105 vs. 7,175 for Wapiti, 38,622 for Burp, 147,595 for ZAP, and 879 for Black Widow).

We observed that Link found many real-world vulnerabilities with fewer requests. We emphasize that the Link's agent is trained on the Firing-Range, WAVSEP, and Filter Evasion benchmarks, which are different to the applications under testing; Link still reported the second highest TPs.

Burp found more bugs than Link did because of the Burp's page crawling capabilities, including name guessing and extrapolation

Application (Version)	**Link**			**Wapiti**			**Burp**			**ZAP**			**BW**		
	TP	**FP**	**FN**	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**
Webid (1.2.2)	10	0	0	10	0	0	10	0	0	10	0	0	0	0	10
Monstra (3.0.4)	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
SeoPanel (4.8.0)	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0
FUDForum (3.1.0)	3	0	0	3	0	0	3	0	0	3	1	0	1	0	2
powebform (1.0.3)	13	0	0	12	0	1	13	0	0	12	0	1	12	0	1
chamilo (1.11.14)	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0
Textpattern (4.5.5)	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1
Schoolmate (1.5.4)	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
ImpressCMS (1.3.10)	5	0	3	0	0	8	8	0	0	0	0	8	3	0	5
Ampache (4.4.2)	4	0	2	1	0	5	5	2	1	4	0	2	2	0	4
GeekLog (2.2.1)	3	0	0	2	0	1	3	1	0	2	0	1	2	0	1
osCommerce (2.3.3)	0	0	0	0	0	0	0	0	0	0	18	0	0	0	0
Total	43	0	6	33	0	16	46	3	3	36	19	13	25	0	24

Table 3: The number of vulnerabilities found in real-world applications using four different tools.

from naming conventions [65]. Designing a more advanced crawler to find URLs and input parameters is not a goal that Link addresses. Instead, we focus on adapting payloads using RL. Also, leveraging the Burp crawler was not feasible for Link because Burp is a commercial tool of which source code is not available for revision.

When considering only the input parameters and URLs that Link's crawler found, Link was able to find every vulnerability that Burp found, but with far fewer requests. These experimental results demonstrate that leveraging RL to adapt attack payloads is a promising approach in performing penetration testing.

Zero-day vulnerabilities. We conducted additional testing against the latest versions of applications with CVE reports in the past. Link reported three new vulnerabilities in Geeklog (v2.2.1sr1) and one in PESCMS (v2.3.3) (CVE-2021-44884).

7 LIMITATIONS AND DISCUSSION

We address the problem of adapting input payloads using RL. However, there are several other aspects that determine the performance of black-box web scanners: (1) the coverage issue of crawling target URLs to attack, (2) the identification of all input parameters to inject payloads, and (3) the prioritization of input parameters to attack. Because Link is built on top of Wapiti, the crawler and modules for identifying and prioritizing input parameters are similar. However, as we demonstrated in the evaluation, generating payloads using RL contributed to Link finding 12 additional TPs that all the scanners in the evaluation were unable to find.

We emphasize Link's capability of reducing the number of attack attempts to trigger target vulnerabilities. It is common that testing campaigns of black-box web scanners take very long execution times. Therefore, achieving both objectives of (1) minimizing the number of requests and (2) maximizing true positives are important tasks for these tools. However, existing off-the-shelf tools focus on implementing their own heuristics. We argue that the approach of leveraging RL is new and effective to address this problem while avoiding implementing arbitrary heuristic rules.

We checked the efficacy of leveraging a Chrome headless browser to check the script execution [8]. However, we observed that using a headless browser introduced a 20% increase in training and testing time. For this reason, we chose to parse HTML and JS responses because parsing is required not only to determine the successful exploitation but also to extract observation information. Note that

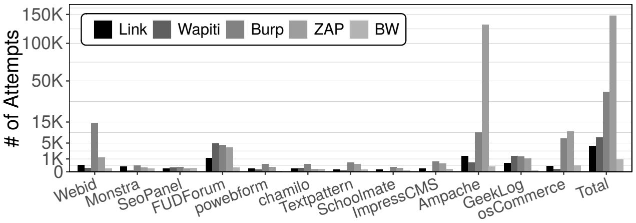

Figure 3: The number of attempted requests by each tool to find vulnerabilities (square root scale).

Link leverages a fully-automatic RL agent in its training and testing. Unlike previous approaches [8], Link requires no human involvement to evaluate chosen actions in given environments.

8 RELATED WORKS

Payload generation with RL. Wang *et al.* [72] applied RL in generating SQL injection payloads to bypass web firewalls. Their RL environments are open-source web firewalls: ModSecurity [64] and WAF-Brain [6]. They encode elements in the SQL injection payloads into states, and actions become mutation rules for each element. The reward function is the value of the confidence interval and whether the attack string passes the firewall or not. Erdödi*et al.* [19] devised a SQL injection vulnerability detection method using Q-Learning. The environment was a web application that implemented a Capture the Flag problem, which is a very specific environment for vulnerability detection.

Caturano *et al.* [8] proposed Suggester, a reflected XSS payload suggestion tool using RL. It suggests mutation rules for penetration testers and requires human involvement, including for analyzing observations, performing actions, and computing rewards. These drawbacks impede Suggester in conducting fully automatic penetration testing, which Link is able to overcome.

Attack detection with RL. Machine learning is often applied to classify malicious payloads. Fang *et al.* [22] present an XSS attack detection model by training adversarial and detection models together. Tariq *et al.* [68] present an XSS attack detection model using genetic algorithms and threat intelligence to detect XSS attacks. They attempted to address the challenge of XSS payloads having various unforeseen patterns.

9 CONCLUSION

To the best of our knowledge, Link is the first black-box web scanner using an RL agent that requires no human involvement. Link uses RL to adapt its attack payloads to vulnerable target contexts, which contributes to avoiding futile attack requests and decreasing false negatives. Link demonstrates its superior efficiency in decreasing the number of attack requests while finding more true positives and fewer false negatives compared to those of its baseline, Wapiti. Our experimental results demonstrate that leveraging RL for identifying reflected XSS vulnerabilities is a promising direction for scalable and accurate penetration testing.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their concrete feedback. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2020R1C1C1009031).

REFERENCES

[1] 4images. 2021. 4images Gallery. https://www.4homepages.de/.

[2] Giovanni Agosta, Alessandro Barenghi, Antonio Parata, and Gerardo Pelosi. 2012. Automated Security Analysis of Dynamic Web Applications through Symbolic Code Execution. In *Proceedings of the International Conference on Information Technology - New Generations*. 189–194.

[3] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2016. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the ACM Conference on Computer and Communications Security*. 641–652.

[4] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. 2018. NAVEX: precise and scalable exploit generation for dynamic web applications. In *Proceedings of the USENIX Security Symposium*. 377–392.

[5] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2008. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*. 387–401.

[6] BBVA. 2019. WAF-Brain: The clever and efficient Firewall for the Web. https://github.com/BBVA/waf-brain.

[7] Ahmet Salih Buyukkayhan, Can Gemicioglu, Tobias Lauinger, Alina Oprea, William Robertson, and Engin Kirda. 2020. What's in an Exploit? An Empirical Analysis of Reflected Server XSS Exploitation Techniques. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses*. 107–120.

[8] Francesco Caturano, Gaetano Perrone, and Simon Pietro Romano. 2021. Discovering reflected cross-site scripting vulnerabilities using a multiobjective reinforcement learning environment. *Computers & Security* 103 (2021), 102204.

[9] Shay Chen. 2014. WAVSEP: The Web Application Vulnerability Scanner Evaluation Project. https://github.com/sectooladdict/wavsep/.

[10] Hyunsang Choi, Seongjin Hong, Sanghyun Cho, and Young-Gab Kim. 2017. HXD: Hybrid XSS detection by using a headless browser. In *Proceedings of the International Conference on Computer Applications and Information Processing Technology*. 1–4.

[11] Tianshu Chu, Jie Wang, Lara Codecà, and Zhaojian Li. 2019. Multi-agent deep reinforcement learning for large-scale traffic signal control. *IEEE Transactions on Intelligent Transportation Systems* 21, 3 (2019), 1086–1095.

[12] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the ACM Conference on Recommender Systems*. 191–198.

[13] Anthony Cozamanis. 2019. XSS Vectors Cheat Sheet. https://gist.github.com/kurobeats/9a613c9ab68914312cbb415134795b45.

[14] Piotr Dabkowski. 2019. pyjsparser. https://github.com/PiotrDabkowski/pyjsparser.

[15] Zoran Djuric. 2013. A black-box testing tool for detecting SQL injection vulnerabilities. In *Proceedings of the International Conference on Informatics and Analytics*. 216–221.

[16] Gabriel Dulac-Arnold, Nir Levine, Daniel J Mankowitz, Jerry Li, Cosmin Paduraru, Sven Gowal, and Todd Hester. 2021. Challenges of real-world reinforcement learning: definitions, benchmarks and analysis. *Machine Learning* (2021), 1–50.

[17] Gabriel Dulac-Arnold, Daniel Mankowitz, and Todd Hester. 2019. Challenges of real-world reinforcement learning. *arXiv preprint arXiv:1904.12901* (2019).

[18] Alvaro Cabrejas Egea, Shaun Howell, Maksis Knutins, and Colm Connaughton. 2020. Assessment of Reward Functions for Reinforcement Learning Traffic Signal Control under Real-World Limitations. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*. 965–972.

[19] Laszlo Erdodi, Åvald Åslaugson Sommervoll, and Fabio Massimo Zennaro. 2021. Simulating SQL Injection Vulnerability Exploitation Using Q-Learning Reinforcement Learning Agents. *arXiv preprint arXiv:2101.03118* (2021).

[20] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. 2021. Black widow: Blackbox data-driven web scanning. In *Proceedings of the IEEE Symposium on Security and Privacy*. 1125–1142.

[21] Olorunjube James Falana, Ife Olalekan Ebo, Carolyn Oreoluwa Tinubu, Olusesi Alaba Adejimi, and Andeson Ntuk. 2020. Detection of Cross-Site Scripting Attacks using Dynamic Analysis and Fuzzy Inference System. In *Proceedings of the International Conference in Mathematics, Computer Engineering and Computer Science*. 1–6.

[22] Yong Fang, Cheng Huang, Yijia Xu, and Yang Li. 2019. RLXSS: Optimizing XSS detection model to defend against adversarial attacks based on reinforcement learning. *Future Internet* 11, 8 (2019), 177.

[23] Python Software Foundation. 2021. difflib: Helpers for computing deltas. https://docs.python.org/3/library/difflib.html.

[24] Vincent François-Lavet, Raphael Fonteneau, and Damien Ernst. 2015. How to discount deep reinforcement learning: Towards new dynamic strategies. *arXiv preprint arXiv:1512.02011* (2015).

[25] Mahmoud Ghorbanzadeh and Hamid Reza Shahriari. 2020. ANOVUL: Detection of logic vulnerabilities in annotated programs via data and control flow analysis. *IET Digital Library* 14, 3 (2020), 352–364.

[26] Google. 2018. Firing Range. https://github.com/google/firing-range.

[27] Mukesh Kumar Gupta, MC Govil, and Girdhari Singh. 2014. Static analysis approaches to detect SQL injection and cross site scripting vulnerabilities in web applications: A survey. In *Proceedings of the International Conference on Recent Advances and Innovations in Engineering*. 1–5.

[28] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, and Anssi Kanervisto. 2021. Stable baselines. https://github.com/hill-a/stable-baselines/.

[29] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. 2011. Fast and Precise Sanitizer Analysis with BEK. In *Proceedings of the USENIX Security Symposium*.

[30] Charlie Hou, Mingxun Zhou, Yan Ji, Phil Daian, Florian Tramer, Giulia Fanti, and Ari Juels. 2019. SquirRL: Automating attack analysis on blockchain incentive mechanisms with deep reinforcement learning. *arXiv preprint arXiv:1912.01798* (2019).

[31] J Stuart Hunter. 1986. The exponentially weighted moving average. *Journal of quality technology* 18, 4 (1986), 203–210.

[32] Martin Johns, Björn Engelmann, and Joachim Posegga. 2008. XSSDS: Server-side detection of cross-site scripting attacks. In *Proceedings of the Annual Computer Security Applications Conference*. 335–344.

[33] Martin Johns and Moritz Jodeit. 2011. Scanstud: a methodology for systematic, fine-grained evaluation of static analysis tools. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*. 523–530.

[34] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* 4 (1996), 237–285.

[35] Martin Kleppe. 2021. JSFuck. http://www.jsfuck.com/.

[36] Vijay R Konda and John N Tsitsiklis. 2000. Actor-critic algorithms. In *Proceedings of the Advances in Neural Information Processing Systems*. 1008–1014.

[37] Penghui Li, Wei Meng, Kangjie Lu, and Changhua Luo. 2021. On the Feasibility of Automated Built-in Function Modeling for PHP Symbolic Execution. In *Proceedings of the Web Conference*. 58–69.

[38] Zhipeng Liang, Hao Chen, Junhao Zhu, Kangkang Jiang, and Yanran Li. 2018. Adversarial deep reinforcement learning in portfolio management. *arXiv preprint arXiv:1808.09940* (2018).

[39] Heloise Maurel, Santiago Vidal, and Tamara Rezk. 2021. Statically Identifying XSS using Deep Learning. In *Proceedings of the International Conference on Security and Cryptography*.

[40] Sebastian Roschke Michal Zalewski, Niels Heinen. 2012. Skipfish - web application security scanner. https://code.google.com/archive/p/skipfish/.

[41] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *Proceedings of the International Conference on Machine Learning*. 1928–1937.

[42] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

[43] Anusha Nagabandi, Ignasi Clavera, Simin Liu, Ronald S Fearing, Pieter Abbeel, Sergey Levine, and Chelsea Finn. 2018. Learning to Adapt in Dynamic, Real-World Environments through Meta-Reinforcement Learning. In *Proceedings of the International Conference on Learning Representations*.

[44] Netsparker. 2021. Web Application Advisories by Netsparker. https://www.netsparker.com/web-applications-advisories/.

[45] OpenAI. 2021. OpenAI Gym. https://gym.openai.com/.

[46] OWASP. 2020. ZAP: The OWASP Zed Attack Proxy. https://www.zaproxy.org/.

[47] OWASP. 2021. OWASP Benchmark. https://owasp.org/www-project-benchmark/.

[48] OWASP. 2021. OWASP Top Ten. https://owasp.org/www-project-top-ten/.

[49] OWASP. 2021. OWASP XSS Fitler Evasion Cheat Sheet. https://owasp.org/www-community/xss-filter-evasion-cheatsheet.

[50] Cosmin Paduraru, Daniel J Mankowitz, Gabriel Dulac-Arnold, Jerry Li, Nir Levine, Sven Gowal, and Todd Hezster. 2021. Challenges of Real-World Reinforcement Learning: Definitions, Benchmarks & Analysis. *Machine Learning Journal* (2021).

[51] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. 2015. jäk: Using dynamic analysis to crawl and test modern web applications. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses*. 295–316.

[52] Jan Peters, Sethu Vijayakumar, and Stefan Schaal. 2003. Reinforcement learning for humanoid robotics. In *Proceedings of the IEEE-RAS International Conference on Humanoid Robots Humanoids*. 1–20.

[53] PortSwigger. 2021. Burp Suite - Cybersecurity Software from PortSwigger. https://portswigger.net/burp.

[54] Portswigger. 2021. Portswigger Research - Cross-Site Scripting. https://portswigger.net/research/cross-site-scripting-research.

[55] PortSwigger. 2022. Cross-site scripting cheet sheet. https://portswigger.net/web-security/cross-site-scripting/cheat-sheet.

[56] Abdul Razzaq, Ali Hur, H Farooq Ahmad, and Muddassar Masood. 2013. Cyber security: Threats, reasons, challenges, methodologies and state of the art solutions for industrial applications. In *Proceedings of the IEEE Eleventh International*

Symposium on Autonomous Decentralized Systems. 1–6.

[57] Leonard Richardson. 2021. Beautiful Soup. https://www.crummy.com/software/BeautifulSoup/.

[58] Marcelo Invert Palma Salas and Eliane Martins. 2014. Security testing methodology for vulnerabilities detection of XSS in web services and ws-security. *Electronic Notes in Theoretical Computer Science* 302 (2014), 133–154.

[59] Ahmad EL Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. 2017. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging* 2017, 19 (2017), 70–76.

[60] Somdev Sangwan. 2019. XSStrike - Advanced XSS Detection Suite. https://github.com/s0md3v/XSStrike.

[61] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[62] Haruyama Seigo. 2011. Vulnerable-Site-Sample. https://github.com/haruyama/Vulnerable-Site-Sample/tree/master/xss.

[63] Prashant S Shinde and Shrikant B Ardhapurkar. 2016. Cyber security analysis using vulnerability assessment and penetration testing. In *Proceedings of the World Conference on Futuristic Trends in Research and Innovation for Social Welfare*. 1–5.

[64] SpiderLabs. 2021. ModSecurity: Open soruce Web Applictaion Firewall. https://github.com/SpiderLabs/ModSecurity.

[65] Dafydd Stuttard. 2009. PortSwigger Blog - Content discovery. https://portswigger.net/blog/v13p-content-discovery.

[66] Nicolas Surribas. 2021. Wapiti. https://wapiti.sourceforge.io/.

[67] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. 2000. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the Advances in Neural Information Processing Systems*. 1057–1063.

[68] Iram Tariq, Muddassar Azam Sindhu, Rabeeh Ayaz Abbasi, Akmal Saeed Khattak, Onaiza Maqbool, and Ghazanfar Farooq Siddiqui. 2021. Resolving cross-site scripting attacks through genetic algorithm and reinforcement learning. *Expert Systems with Applications* 168 (2021), 114386.

[69] UliCMS. 2022. UliCMS - Make Content Management Great Again. https://en.ulicms.de/.

[70] Steven Van Acker, Nick Nikiforakis, Lieven Desmet, Wouter Joosen, and Frank Piessens. 2012. FlashOver: Automated discovery of cross-site scripting vulnerabilities in rich internet applications. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*. 12–13.

[71] Shangxing Wang, Hanpeng Liu, Pedro Henrique Gomes, and Bhaskar Krishnamachari. 2018. Deep reinforcement learning for dynamic multichannel access in wireless networks. *IEEE Transactions on Cognitive Communications and Networking* 4, 2 (2018), 257–265.

[72] Xianbo Wang and Han Hu. 2020. Evading Web Application Firewalls with Reinforcement Learning. https://openreview.net/forum?id=m5AntlhJ7Z5

[73] Gary Wassermann and Zhendong Su. 2008. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the International Conference on Software Engineering*. 171–180.

[74] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. 2011. A systematic analysis of XSS sanitization in web application frameworks. In *Proceedings of the European Symposium on Research in Computer Security*. 150–171.

[75] XSSer. 2020. Cross Site "Scripter" (aka XSSer). https://github.com/epsylon/xsser.

10 APPENDIX

10.1 Implementation

We implemented Link with 7K lines of code (LoC) in Python. We built Link on top of Wapiti [66], a popular open-source web scanner that has been used in many web vulnerability detection studies [15, 56]. We replaced the Wapiti module that applies a set of fixed attack payloads with an RL agent that uses OpenAI gym [45] and Stable baselines [28]. We further used the BeautifulSoup library [57] to parse responses to attack requests in our bug oracle. Also, for accurate detection of JS snippets, we used pyjsparser [14]. To support open science and reproducible research, we release Link at https://github.com/WSP-LAB/Link.

10.2 RL Actions and Observations

Actions. Table 4 describes each of the 39 actions. We describe several actions in detail here. The 20th action prepends the `dummy` string to the payload to evade filters that truncate the prefix of user input. The 21st action prepends a closing tag to escape an opening tag that appears immediately ahead of the injection point. For example, if the agent selects the 21st action when it observes the response `<textarea>[payload]</textarea>`, the attack manager identifies the `<textarea>` tag that is the closest opening tag before [payload] and prepends the closing tag `</textarea>` to the injected payload.

Observations. Table 5 lists 47 features that the attack manager observes. These features are categorized into five groups: payload appearance, payload repetitiveness, reflected payload appearance, payload context information, and attack result. The 30th feature is the number of times that Link uses the current payload. We also use the maximum steps in the training phase for the denominator to provide the same impact in the testing phase as well.

10.3 RL Reward

Algorithm 2 shows how the attack manager computes an r_t. It first checks whether a_t enables the successful exploitation of a reflected XSS vulnerability. If so, the reward becomes the difference between the number of maximum steps (*STEPS_PER_EPISODE*) to the current *t*, thereby motivating the agent to find an appropriate attack payload as early as possible. That is, the agent is rewarded for sending a small number of attack requests to trigger a target vulnerability. On the other hand, Lns 7–10 decrease the reward by one when a_t is the same as a_{t-1} or when the current payload equals the previous payload, instructing the agent to avoid the same action and payload. In a similar vein, the attack manager assigns a negative reward by subtracting the frequency of the current payload (Ln 12), helping the agent learn a way of not generating a payload that the agent has produced before.

10.4 Hyperparameter Optimization

RL algorithms and training steps. We evaluated three RL algorithms for Link agents: PPO, DQN, and A2C. These algorithms were chosen due to their prevalent applications in various domains [11, 38, 42, 71]. We deployed each algorithm for Link and examined the exponential moving average (EMA) [31] of rewards measured over the training step. The higher the EMA is, the better the agent performs in generating optimal actions.

Algorithm 2: Algorithm for computing a reward r_t at a time step t.

```
1  $previous\_payload$ ← Payload produced in the previous step
2  function Reward($t, s_{t+1}, payload, found$)
3      $r_t \leftarrow 0$
4      if $found$ then
5          $r_t \leftarrow STEPS\_PER\_EPISODE - t$
6      // If the current action (#32) is the same with the previouse action
           (#31), give a negative reward.
7      if $s_{t+1}[32]$ == $s_{t+1}[31]$ then
8          $r_t \leftarrow r_t - 1$
9      if $payload$ == $previous\_payload$ then
10         $r_t \leftarrow r_t - 1$
11     // If the input payload is frequently used (#30), give a negative
           reward.
12     $r_t \leftarrow r_t - s_{t+1}[30]$
13     return $r_t$
```

For every agent except DQN, the EMA of rewards increased continuously, becoming almost equal across agents as the number of steps increased. In particular, we observed that the EMA for the A2C agent converges faster than the EMA for the PPO agent. We also noticed that a *total_step* value of 3.5M is the optimal point at which the EMA of rewards for the A2C model starts to plateau. Thus, for the evaluations in this paper, we leveraged the A2C model trained up to a *MAX_STEPS* value of 3.5M.

Learning rate. We compared the EMA of rewards for each A2C model trained with different learning rates of 0.001, 0.0005, and 0.00075. Since there were no significant differences in EMA values according to varying learning rates, we chose 0.0005, which converges slightly faster than the others.

Discount factor. A discount factor plays a role in the stability and convergence of deep reinforcement learning algorithms [24]. We considered three discount factors: 0.8, 0.95, and 0.99. We observed that the EMA of rewards for the A2C model converges faster when the discount factor is 0.8 or 0.95 than when the discount factor is 0.99. We chose 0.95 rather than 0.8 to give more weight to future rewards.

```
1  <form action="./example.php" method="get">
2    <?php
3       $input = htmlentities($_GET["go"]);
4       echo urldecode($input); ?>
5    <input type="text" name="go" value="">
6    ...
7  </form>
```

Figure 4: Vulnerable snippet of incorrect input sanitization that causes a reflected XSS vulnerability.

10.5 Case Study

We investigate the findings of Link in the experiments presented in Section 6.2 and how our RL agent contributed to uncovering the reflected XSS bugs.

Figure 4 shows a vulnerable code snippet of incorrect input sanitization, causing a reflected XSS vulnerability. This PHP application incorrectly sanitizes user input via `$_GET["go"]`. It attempts to remove all script tags by invoking `htmlentities()` in Ln 3, which converts special characters to their corresponding HTML entities.

Type	Category	Action #	Description
Generation	Basic Payload	1	Generate the script tag payload (`<script>alert(1);</script>`)
		2	Generate a payload with a media tag (`img`, `video`, `audio`, `svg`) (e.g., `<img src=‘x’ onerror=‘alert(1);’/>`)
		3	Generate a payload in an event attribute (`onerror`, `onload`, `onclick`, `onmouseover`) (e.g., `onerror=alert(1);`)
		4	Generate a palyload in an src attribute (e.g., `src=‘http://attack.js’`)
	JS Component	5–7	Generate the JS snippet (`alert(1)`), the JS URL (`http://attack.js`), the JS pseudo protocol (`javascript:alert(1)`)
Mutation	Prefix	8–20	Prepend a symbol from `/>` , `"` , `'` , `">` , `'>` , `>` , `-->` , `;}` , `*/` , `';` , `";` , enter , or dummy
		21	†Prepend a closing tag for an opening tag that appears immediately ahead of the injection point
	Suffix	22–24	Append `<!--` , `'` , `"` to the end of the payload
	Tag	25–27	Replace the tag with other script or media tag / Capitalize the tag / Overlap the tag string (e.g., `scrscriptipt`, etc.)
	Attribute	28	Capitalize the attribute
	JS snippet	29	Replace the JS snippet with one of the other JS snippets (`alert(1)`, `confirm(1)`, `prompt(1)`)
		30	Split the JS snippet into several JS pieces (e.g., `alert(1)` ⇒ `var A="al"+"er"+"t(1);";eval(A);`)
		31–32	Prepend `"+` / `'+` and append `+"` / `+'` to the JS snippet (e.g., `alert(1)` ⇒ `"+alert(1)+"` / `'+alert(1)+'`)
		33	Put the JS snippet in `onerror`, and force an error with throw (e.g., `alert(1)` ⇒ `onerror=alert;throw 1`)
	Entire string	34–36	Apply percent encoding / octet encoding / obfuscation with JSFuck [35]
		37–39	Replace a white space with a slash / a single quote with a back quote / a parenthesis (i.e., “(” or “)”) with a back quote

† It can be incrementally combined with actions 8–20.

Table 4: Action details.

Type	Category	Feature #	Features	Range
Input Payload	Payload appearance	1–4	The presence of a “script” string / an “alert” string / a “(” or “)” string / a “dummy” string	0–1
		5–6	The presence of an event attribute (`onerror`, `onload`, `onclick`, `onmouseover`) / a URL attribute (`href`, `src`)	0–1
		7–9	The presence of a JS snippet (`alert(1)`, `confirm(1)`, `prompt(1)`) / a HTML comment / a JS comment	0–1
		10–11	The presence of JS pseudo protocol (`javascript:alert(1)`) / a JS URL (`http://attack.js`)	0–1
		12–14	The presence of an HTML tag / a script tag (`<script>`) / a media tag (`<img>`, `<video>`, `<audio>`, `<svg>`)	0–1
		15–21	The presence of a single quote / a double quote / a back quote / a back slash / a bracket / enter / a closing tag	0–1
		22–25	Whether it is `alert(1)`, `confirm(1)`, or `prompt(1)` / percent encoded / octet encoded / obfuscated	0–1
		26–28	The presence of a capitalized tag / a capitalized attribute / an overlapped tag string (e.g., `scrscriptipt`, etc.)	0–1
		29	The presence of a white space (e.g., `<img/src=‘x’/onerror=‘alert(1)’/>`)	0–1
	Payload repetitiveness	30	Payload frequency = *(# of times that the current payload is used in the current episode - 1)* / *TRAINING_EPISODE_SIZE)*	[0, 1]
		31-32	Action number used in the previous step / current step	Action #
Response	Reflected payload appearance	33	The presence of a script tag or a media tag from a payload in the response	-1–1
		34	The presence of a JS snippet, a JS URL, or a JS pseudo protocol from a payload in the response	-1–1
		35	String similarity between a payload and a reflected payload in the response	[0, 5]
	Payload† context information	36	Content type of a target page (HTML, CSS, JSON)	0–3
		37	The input parameter type from its default value (number, string)	0–2
		38	Where the payload is injected (tag, attribute name, attribute value, URL, comment, enclosed tag content, event value)	0–7
		39–40	A character symbol immediately before / after the injection point (`'`, `"`, < or >, =, `:`, `;`)	0–6
		41	Among the characters (`"`, `'`, `//`, `/*`, <, >, `-`), the character placed closest before the injection point	0–7
		42	HTML tag type immediately before the point where the payload is injected (link, media)	0–2
		43–46	Reflection of initial payload	0–1
Attack	Attack Result	47	The success of an attempt	0–1

† 0 of the feature values refers to the default value that is assigned if no corresponding value is observed.

Table 5: Observation details.

Step	Action #	Type	Payload
1	1	Request	`<script>alert(1);</script>`
		Response	`&lt;script&gt;alert(1);&lt;/script&gt;`
2	34	Request	`%3Cscript%3Ealert%281%29%3B%3C/script%3E`
		Response	`<script>alert(1);</script>`

Table 6: Running sequence of RL agent on one Filter Evasion benchmark.

Unfortunately, an attacker is able to inject a URL-encoded JS payload into an argument of the `urldecode()` function in Ln 4, thereby eliminating the need for injecting special escape characters into an attack payload.

Table 6 shows the sequence of actions that Link conducts against the vulnerable snippet in Figure 4. Link starts by sending the basic attack payload generated by action 1. Recognizing that special characters are filtered or encoded, Link performs a percent-encoding mutation, successfully triggering the vulnerability. ZAP, Wapiti, and BW reported false negatives for this vulnerability because the exploit payload is not included in their payload dictionaries. We observed that Burp Suite triggers this vulnerability through 13 requests. On the other hand, Link achieved the same with only five requests including three requests that exploit initial payloads with the help of its RL agent choosing optimal actions.