Generative Pre-Trained Transformer-Based Reinforcement Learning for Testing Web Application Firewalls

Hongliang Liang, *Member, IEEE*, Xiangyu Li, Da Xiao, Jie Liu, Yanjie Zhou, Aibo Wang, and Jin Li

***Abstract*—Web Application Firewalls (WAFs) are widely deployed to protect key web applications against multiple security threats, so it is important to test WAFs regularly to prevent attackers from bypassing them easily. Machine-learning-based black-box WAF testing is gaining more attention, though existing learning-based approaches have strict requirements on the source and scale of payload data and suffer from the local optimum problem, limiting their effectiveness and practical application. We propose GPTFuzzer, a *practical* and *effective* generation-based approach to test WAFs by generating attack payloads token-by-token. Specifically, we fine-tune a Generative Pre-trained Transformer language model with reinforcement learning to make GPTFuzzer have the least restrictions on payload data and thus more applicable in practice, and we use reward modeling and KL-divergence penalty to improve the effectiveness of our approach and mitigate the local optimum issue. We implement GPTFuzzer and evaluate it on two well-known open-source WAFs against three kinds of common attacks. Experimental results show that GPTFuzzer significantly outperforms state-of-the-art approaches, i.e., ML-Driven and RAT, finding up to 7.8× (3.2× on average) more bypassing payloads within 1,250,000 requests, or finding out all bypassing payloads using up to 8.1× (3.3× on average) fewer requests.**

***Index Terms*—Black-box testing, reinforcement learning, transformer, web application firewall.**

I. Introduction

WEB applications are currently facing more and more frequent attacks. For example, 234 web attacks occur per minute and 4,800+ unique websites are compromised on average every month [1]. Web Application Firewalls (WAFs) are widely deployed to protect key web applications against multiple security threats, and most WAFs (e.g., AWS WAF, Azure WAF, ModSecurity, Naxsi) detect attacks using a set of rules predefined by security experts.

Unfortunately, these rules are growing too complex to update dynamically and manually. Therefore, various types of security testing, e.g., white-box testing, model-based testing, and black-box testing, are proposed to test WAFs automatically, among which black-box testing has fewer limitations in practice [2]. Moreover, machine-learning-based techniques in black-box WAF testing, which learn attack patterns from previous testing attempts and select the next payload likely exposing a vulnerability in WAFs under test, are presented to improve the testing performance [2], [3], [4], [5], [6], [7].

We divide existing efforts on machine-learning-based black-box WAF testing into two categories: 1) mutation-based methods, e.g., XSS Analyzer [3], ML-Driven [2], [4] and WAF-A-MoLE [8], where machine learning models are used to select from a pool of payloads a payload and its part of data for mutation each time. To avoid producing a large number of invalid payloads in the mutation process, an attack grammar is usually used to ensure that the mutated payloads conform to *explicit* grammatical requirements of a type of attack. For example, in ML-Driven E [2], a payload is mapped to a parse tree according to an attack grammar, which guides the mutation. 2) search-based methods, e.g., ART4SQLi [5], XSSART [6], and RAT [7], where the candidate payloads are clustered using machine learning techniques, and those payloads bypassing WAFs are searched out efficiently by exploiting this clustering structure. For instance, RAT [7] uses deep auto-encoders to extract features from payloads for clustering them and uses ϵ-greedy reinforcement learning to search for those clusters containing bypassing payloads.

In mutation-based approaches, the predefined attack grammars (e.g., ML-Driven) or mutation operators (e.g., WAF-A-MoLE) play a vital role. However, they need to be constructed by domain experts manually and are usually outdated without updates in time. Besides, as mutation relies on the limited seeds (initial payloads), the payload space may not be explored sufficiently, causing the local optimum problem.

In search-based approaches, the grammars of discovered payloads are readily satisfied because they are coming from existing valid ones, rather than newly generated ones. However, to obtain bypassing payloads with satisfactory quantity and variety, the payload space must be densely sampled to construct a very large payload set to search from. The extreme case is that the full set of payloads for a type of attack is used, as RAT [7] did, which is prohibitive, if not impossible, in practice. Besides, the ϵ-greedy search strategy in RAT also causes the local optimum problem: it focuses on those clusters containing many bypassing payloads, and thus does not explore the payload set thoroughly.

Manuscript received 12 January 2022; revised 12 February 2023; accepted 27 February 2023. Date of publication 6 March 2023; date of current version 12 January 2024. This work was partially supported by CNKLSTISS. *(Corresponding author: Da Xiao.)*

Hongliang Liang, Xiangyu Li, Da Xiao, Jie Liu, Yanjie Zhou, and Aibo Wang are with the Trusted Software and Intelligent System Lab, Beijing University of Posts and Telecommunications, Beijing 100876, China (e-mail: hliang@bupt.edu.cn; pyterware@bupt.edu.cn; xiaoda99@bupt.edu.cn; liujie_ran@bupt.edu.cn; yanjiezhou@bupt.edu.cn; wangab@bupt.edu.cn).

Jin Li is with the Nation Key Laboratory of Science and Technology on Information System Security, Beijing 100101, China (e-mail: tianyi198012@163.com).

Digital Object Identifier 10.1109/TDSC.2023.3252523

Overall, existing approaches have strict requirements on either payload source (e.g., defined by an explicitly written grammar) or payload scale (e.g., the full set of a type of attack), which limit their applications in practice. They also suffer from the local optimum problem when exploring the payload space, decreasing their effectiveness in discovering bypassing payloads.

To address the above problems, we propose a generation-based WAF testing approach, where rather than mutating or searching within existing payloads, new payloads are generated token-by-token to test the WAFs. On the one hand, instead of relying on an explicit attack grammar or a large set of payloads (e.g., 4 million payloads in RAT), our approach needs only a modest size of payload set (e.g., 20K-200 K). This is achieved by fine-tuning a Generative Pre-trained Transformer language model with reinforcement learning (we thus name our approach GPTFuzzer). Given an attack type (e.g., SQL injection), a language model is pre-trained to learn the attack's *implicit* grammatical requirements and then adapted to different WAFs (e.g., ModSecurity or Naxsi) through reinforcement learning. On the other hand, to mitigate local optimum when exploring payload space, we exploit techniques from reinforcement learning in natural language processing (NLP), i.e., reward modeling and KL-divergence penalty, to improve the effectiveness of GPTFuzzer.

Moreover, we implement and evaluate GPTFuzzer on two well-known open-source WAFs (ModSecurity and Naxsi) against the three most common web attacks, i.e., SQL injection (SQLi), Cross-Site Scripting (XSS), and Remote Command Execution (RCE). The experimental results show that GPTFuzzer is practical and effective; it significantly outperforms state-of-the-art approaches, i.e., ML-Driven and RAT, finding up to 7.8× (3.2× on average) more bypassing payloads within 1,250,000 requests, or finding out all bypassing payloads using up to 8.1× (3.3× on average) fewer requests.

To the best of our knowledge, GPTFuzzer is the first work that combines reinforcement learning with a Transformer model to test WAFs. The main contributions of this paper are:

- We propose a practical generation-based framework for black-box WAF testing by combining the Transformer language model with reinforcement learning.
- We combine several techniques, i.e., language model pre-training, reward modeling, and KL-divergence penalty, to generate attack payloads effectively and mitigate the local optimum issue.
- We implement GPTFuzzer and evaluate it on two well-known WAFs against three types of typical web attacks, i.e., SQLi, XSS, and RCE. Experimental results show that GPTFuzzer outperforms two state-of-the-art approaches, i.e., ML-Driven and RAT.
- We make attack grammars for three typical Web attacks, trained models and datasets publicly available,[1] in the hope of facilitating further research in the web security field.

[1]https://github.com/hongliangliang/gptfuzzer

II. Approach

The overall framework of GPTFuzzer is illustrated in Fig. 1, including five phases: data collection and preprocessing (Section II-A), language model pre-training (Section II-B), reward model training (Section II-C), reinforcement learning (RL, Section II-D), and generation (Section II-E).

In the first phase, a dataset of payloads for an attack type is constructed in one of two ways: 1) manually collecting attack payloads (Section II-A1); 2) constructing a context-free grammar (called attack grammar) and then generating attack payloads automatically from it (Section II-A2). Afterwards, each payload is then preprocessed as a sequence of integers, which is later used as an input to our models in subsequent phases (Section II-A3).

Then in the language model pre-training phase, the preprocessed dataset generated in the previous phase is used to pre-train a language model which embodies the implicit grammatical requirements of an attack type. The language model will be fine-tuned against a target WAF in two ways in the following two phases.

In the reward model training phase, we sample some attack payloads and test each payload against the target WAF, obtaining a label indicating if it can successfully bypass the WAF or not. We then fine-tune the pre-trained language model with this labeled dataset to get a reward model. The reward model can estimate the probability which a payload will bypass the target WAF, and take it as a reward in the reinforcement learning (RL) phase.

In the RL phase, guided by the reward model, we fine-tune the pre-trained language model with RL to get a policy network and adapt it to the specific attack patterns against the target WAF.

In the last phase, the RL fine-tuned model is used to generate attack payloads token-by-token. These payloads are then tested against the WAF and labeled. Afterwards, they can be used to re-train the reward model and re-run reinforcement learning. This iterative fine-tuning process ends with attack payloads that can bypass the WAF with maximum probability, as Section IV-A shows.

A. Data Collection and Preprocessing

Attack payloads are first constructed in one of the following two ways before being preprocessed for model learning:

1) Manual Collection: As a basic way, we manually collect existing datasets for specific web attacks published in research papers e.g., [9], [10], [11], [12], [13], and released by other sources, e.g., OWASP.[2] We observe that both the quantity and the quality of open-source datasets in the area of web attacks, particularly in WAF testing, are inadequate in practice. As a complement, attack payloads can also be generated using fuzzing tools, e.g., SqlMap,[3] XSStrike,[4] Dharma,[5] and Commix,[6] as RAT did [7].

[2]https://owasp.org
[3]https://github.com/sqlmapproject/sqlmap
[4]https://github.com/s0md3v/XSStrike
[5]https://github.com/MozillaSecurity/dharma
[6]https://github.com/commixproject/commix

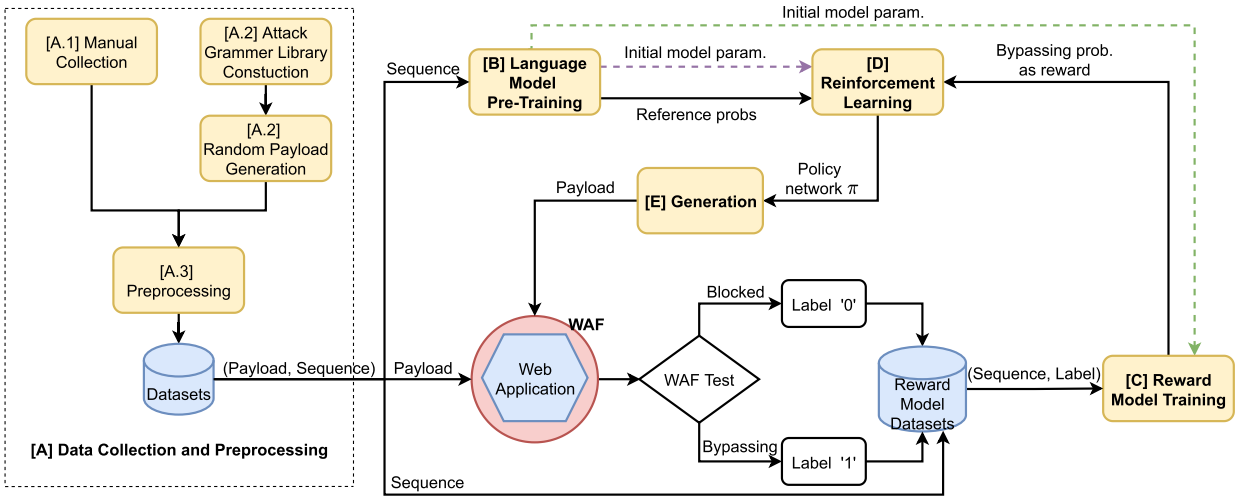


Fig. 1. The overall framework of GPTFuzzer.

2) Generation From Attack Grammar: The second approach to building datasets is to first construct an attack grammar for a specific attack and then generate attack payloads automatically from it. The attack grammar can be constructed either bottom-up, by analyzing existing datasets, extracting common patterns from them, and expressing the patterns in the grammar, or top-down, by studying the underlying mechanisms and principles of the attack and constructing the attack grammar from scratch. Both ways are used in our work. Compared with directly collecting existing datasets, this approach requires more human labor and expertise but can generate datasets with higher quantity and quality.

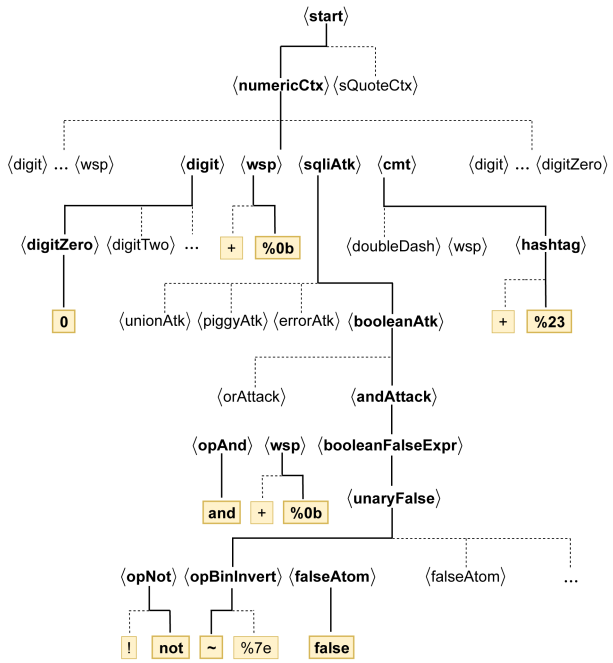
A grammar for a kind of attack is defined in the Extended Backus Normal Form as ML-Driven does. Fig. 2 shows an excerpt of the SQLi grammar, which covers two kinds of contexts (i.e., *numericCtx* and *SQuoteCtx* at lines 2-4), four kinds of SQLi attacks (i.e., *unionAtk, piggyAtk, errorAtk, booleanAtk* at lines 5-12), SQL functions, operators and keywords. In the grammar, $\langle start \rangle$, "::=", "" and "|" denotes the start, production, concatenation, and alternatives symbol, respectively.

We found in preliminary experiments that the attack grammar for SQLi proposed in ML-Driven could hardly generate any effective bypassing payloads for the newer ModSecurity and CRS version than the ones used in ML-Driven. So we improve the SQLi grammar of ML-Driven by modifying the injection contexts, adding more SQL functions (e.g., lines 13-15), and adding more semantically equivalent terminal characters or unicode encodings to the production rules for terminals (e.g., lines 19, 20, 22). Moreover, while ML-driven only targets SQLi attack, we also construct attack grammars for XSS and RCE attacks.

Based on the above grammars, generating random payload sequences (we name the process as Random Fuzzer) is straightforward, as shown in Fig. 3(a). Specifically, we begin from the start symbol $\langle start \rangle$ and randomly select and apply a production rule for each non-terminal. The recursive process ends until all symbols are terminals. As a result (Fig. 3(b)), two kinds of payload sequences can be obtained by either concatenating *terminal* symbols (short sequence for short) or concatenating *all* symbols (long sequence for short), in the pre-order traversal. GPTFuzzer works with short sequences by default and supports long sequences as well. We evaluate their effectiveness and efficiency in Section IV-B. Obviously, we can easily build an

1. $\langle start \rangle$ ::= ⟨numericCtx⟩ | ⟨sQuoteCtx⟩;
Injection Context
2. $\langle numericCtx \rangle$ ::= ⟨digit⟩, ⟨wsp⟩, ⟨booleanAtk⟩, ⟨wsp⟩
| ⟨digit⟩, ⟨wsp⟩, ⟨sqliAtk⟩, ⟨cmt⟩
| ⟨digit⟩, ⟨squote⟩, ⟨wsp⟩, ⟨booleanAtk⟩, ⟨wsp⟩, ⟨opOr⟩, ... ;
3. $\langle sQuoteCtx \rangle$ ::= ⟨digitZero⟩, ⟨squote⟩, ⟨fuzzStr⟩, ⟨wsp⟩,
⟨booleanAtk⟩, ⟨wsp⟩, ⟨opOr⟩, ⟨squote⟩ | ... ;
4. $\langle sqliAtk \rangle$ ::= ⟨unionAtk⟩ | ⟨piggyAtk⟩ | ⟨errorAtk⟩ | ⟨booleanAtk⟩;
Union Attacks
5. $\langle unionAtk \rangle$ ::= ⟨opUni⟩, ⟨wsp⟩, ⟨opSel⟩, ⟨wsp⟩, ⟨cols⟩ | ... ;
Piggy-backed Attacks
6. $\langle piggyAtk \rangle$ ::= ⟨opSem⟩, ⟨opSel⟩, ⟨wsp⟩, ⟨funcSleep⟩;
Error-based Attacks
7. $\langle errorAtk \rangle$ ::= ⟨opAnd⟩, ⟨wsp⟩, ⟨errorFunc⟩, ⟨parOpen⟩, ... ;
Boolean-based Attacks
8. $\langle booleanAtk \rangle$ ::= ⟨orAttack⟩ | ⟨andAttack⟩;
9. $\langle orAtk \rangle$ ::= ⟨opOr⟩, ⟨wsp⟩, ⟨booleanTrueExpr⟩;
10. $\langle andAtk \rangle$ ::= ⟨opAnd⟩, ⟨wsp⟩, ⟨booleanFalseExpr⟩;
11. $\langle booleanFalseExpr \rangle$ ::= ⟨unaryFalse⟩;
12. $\langle unaryFalse \rangle$ ::= ⟨opNot⟩, ⟨opBinInvert⟩, ⟨falseAtom⟩ | ... ;
SQL Functions
13. $\langle funcSleep \rangle$ ::= ⟨opSleep⟩, ⟨parOpen⟩, ⟨digit⟩, ⟨par⟩;
14. $\langle fuzzStr \rangle$ ::= "<@=1" | "<@!=1" | "<@=1." | ... ;
15. $\langle fuzzEqual \rangle$ ::= "1<@" | "@<@" | "!@<@" | ... ;
SQL Operators and Keywords
16. $\langle cmt \rangle$ ::= ⟨doubleDash⟩, ⟨wsp⟩ | ⟨hashtag⟩;
17. $\langle opAnd \rangle$ ::= "and" | "&&";
18. $\langle opNot \rangle$ ::= "!" | "not";
19. $\langle opBinInvert \rangle$ ::= "~" | "%7e";
20. $\langle hashtag \rangle$ ::= "#" | "%23";
21. $\langle falseAtom \rangle$ ::= "false";
Obfuscation
22. $\langle wsp \rangle$ ::= "+" | "%0b";

Fig. 2. Excerpt of the SQL Injection attack grammar.



(a) generation process

Payload sequence w/ all symbols :

(⟨start⟩, ⟨numericCtx⟩, ⟨digit⟩, ⟨digitZero⟩, 0, ⟨wsp⟩, %0b, ⟨sqliAtk⟩, ⟨booleanAtk⟩, ⟨andAttack⟩, ⟨opAnd⟩, and, ⟨wsp⟩, %0b, ⟨booleanFalseExpr⟩, ⟨unaryFalse⟩, ⟨opNot⟩, not, ⟨opBinInvert⟩, ~, ⟨falseAtom⟩, false, ⟨cmt⟩, ⟨hashtag⟩, %23)

Payload sequence w/ terminal symbols : (0, %0b, and, %0b, not, ~, false, %23)

Payload : (0%0band%0bnot~false%23)

(b) generated payload sequences

Fig. 3. An example of random attack payload generation from SQLi attack grammar.

actual payload, which is used for testing WAFs, from a payload sequence with terminals.

3) Preprocessing: We preprocess each payload or payload sequence in the dataset into a sequence of integers before sending them to our models in subsequent phases.

For the manually collected payloads, we first tokenize each payload into tokens, with a tokenizer either learned from the payload dataset or built from a predefined vocabulary. As an example, in SQLi, constitutive tokens of the sample payload "*0%0band%0bnot˜false%23*" may be: *0, %0b, and, %0b, not, ˜, false, %23*. Next, we build a dictionary mapping each token to an integer (i.e., token ID). Finally, we convert each payload to a sequence of token IDs using the dictionary.

For payload sequences generated from the attack grammar, we follow the same steps as above except that the first step (tokenization) is omitted.

B. Language Model Pre-Training

We take attack payloads as "sentences" in some kind of attack "language," and use a language model trained on attack corpus to generate attack payloads, inspired by the studies [14], [15] where a language model is trained on text corpus to generate natural language sentences. However, before the language model can generate "sentences" for certain a purpose, e.g., attack payloads that can successfully bypass a target WAF, it should first learn to generate correct "sentences," e.g., payloads that conform to implicit grammatical requirements of the attack type. This is achieved by pre-training the language model on an attack payload dataset.

Fig. 4(a) depicts the process of the language model pre-training. Given the token ID sequence $\tau = \{x_0, x_1, \ldots, x_T\}$ of a payload from dataset D_p, where x_0 and x_T are IDs for two special tokens $\langle start \rangle$ and $\langle end \rangle$ added to the beginning and end of the sequence respectively, we use a language model ρ to define the probability distribution of the sequence:

$$\rho(x_{0:T}) = \prod_{0<t\leq T} \rho(x_t | x_{0:t-1}) \tag{1}$$

where $x_{0:t-1}$ denotes $x_0, x_1, \ldots, x_{t-1}$. We use a standard language modeling objective to minimize the following loss function:

$$L_1 = -\frac{1}{|D_p|} \sum_{\tau \in D_p} \sum_{t=1}^{T} \log \rho(x_t | x_{0:t-1}; \theta) \tag{2}$$

where $|D_p|$ is the size of training data, $T+1$ is the length of the sequence, and ρ is parameterized with θ.

The language model ρ is a multi-layer Transformer decoder [16]. Its input $X = [x_0, x_1, \ldots, x_T]$, i.e., the vector representation of payload sequence τ, is first converted into token embeddings which are added by position embeddings. The resulted embeddings are then processed by a stack of (n levels) transformer blocks and an output layer (language model head) that produces for each input token ID x_i an output distribution over next token IDs:

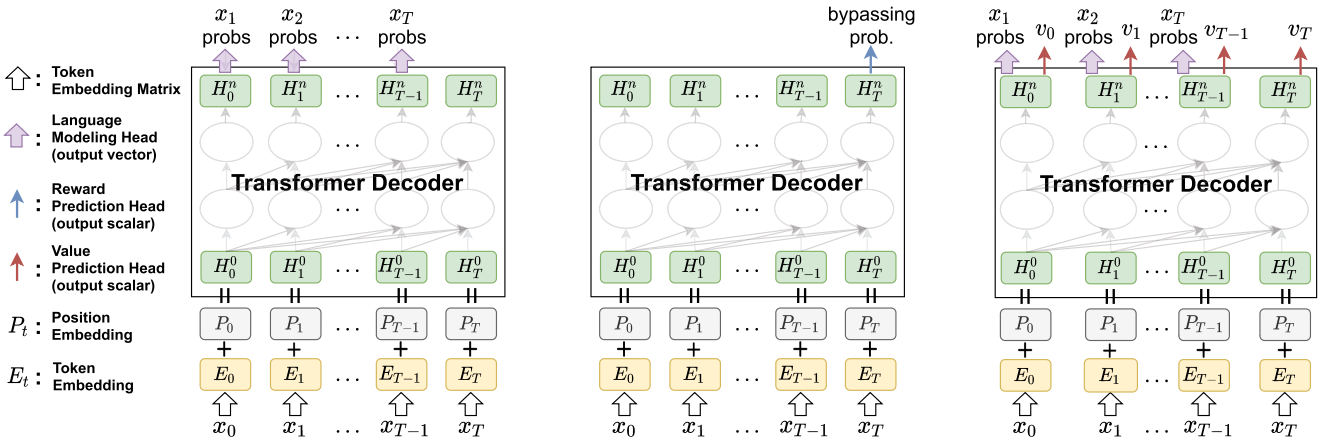
$$\begin{aligned} H^0 &= XW_e + W_p, \\ H^l &= \text{transformer\_block}(H^{l-1}), l = 1, \ldots, n \\ \rho(X) &= \text{softmax}(H^n W_e^T) \end{aligned} \tag{3}$$

where W_e is the token embedding matrix, W_p is the position embedding matrix. Each transformer block contains a multi-head masked self-attention module followed by a position-wise feed-forward neural network since the self-attention modules enable the model to learn long-distance dependencies between input tokens better than RNN or CNN-based sequence models.

C. Reward Model Training

Once upon receiving a payload, the WAF under test can give a *binary-valued* result, i.e., bypassing (1) or blocked (0). It is convenient to directly use the results as rewards for RL, though this simple feedback causes inefficient learning, especially at the early stage of RL when there are a few bypassing payloads, i.e., positive rewards. Therefore, we train a reward model to predict a *real-valued* bypassing probability of a payload, and take it as a reward, providing a richer learning signal for RL.

Specifically, we randomly sample certain numbers (e.g., 2,000 or 4,000 in our evaluation) of attack payloads collected in Section II-A, test them against the target WAF and construct



(a) Language model pre-training (b) Reward model training (c) Reinforcement learning

Fig. 4. Transformer models in different phases. The parameters of the models in (b) and (c) are initialized from the pre-trained model in (a).

a dataset labeled with the test results. Then, we use this dataset to train a reward model to predict the bypassing probability of a payload. Intuitively, the reward model is trained to *simulate* the response of the WAF in the absence of real WAF interaction during RL. Note that after the reward model is trained, interaction with the target WAF is not needed anymore during the RL phase.

To make the reward model inherit the understanding of implicit grammatical requirements from the pre-trained model and thus accelerate its convergence, we initialize the reward model's backbone with the pre-trained language model (Section II-B) and then add on top a reward prediction head, as shown in Fig. 4(b). Given a payload's token ID sequence $\tau = \{x_0, x_1, \ldots, x_T\}$ with a label $y \in \{0, 1\}$ from dataset D_r, the inputs are passed through our pre-trained model to obtain the final step's final transformer block's activation H_T^n, which is then fed into the reward prediction head to predict y:

$$r(\tau) = P(y = 1|x_{0:T}) = \text{sigmoid}(H_T^n W_r + b_r) \quad (4)$$

where the model's scalar output $r(\tau) \in [0, 1]$ is the predicted bypassing probability of the payload, W_r and b_r are learnable parameters of the reward prediction head. We use the binary cross-entropy loss function to train the reward model:

$$L_2 = -\frac{1}{|D_r|} \sum_{\tau \in D_r} \big(y \log(r(\tau)) + (1 - y) \log(1 - r(\tau))\big) \quad (5)$$

where $|D_r|$ is the size of training data. After training, the reward model's output $r(\tau)$ is used as the reward for the payload τ in RL. As shown in Section IV-D, this reward is more effective than the (bypassing or blocked) results obtained from directly testing WAF.

D. Reinforcement Learning

We take black-box WAF testing as an RL process modeled by a Markov Decision Process (MDP), where an agent makes decisions in an environment and earns rewards. Formally, an MDP is a tuple (S, A, P, R), where S is a set of states, A is a set of actions, $P : S \times A \to S$ is the state transition function, and $R : S \times A \to \mathbb{R}$ is the reward function. At each step t, the agent observes the current state s_t and performs an action a_t. It then receives a reward $R_t = R(s_t, a_t)$ and proceeds to another state according to the state transition function: $s_{t+1} = P(s_t, a_t)$. The goal of solving an MDP is to learn a probabilistic policy $\pi_{opt} : S \times A \to [0, 1]$ which maximizes the expected cumulative reward. Afterwards, whenever observing a new state, the agent performs an action sampled from the learned policy according to its probability.

In the RL formulation of WAF testing, the environment is the WAF which provides testing results (i.e., bypassing or blocked) as a reward. In GPTFuzzer, as described in Section II-C, the reward is provided by the reward model r which *simulates* the response of the WAF. So the WAF and the reward model are the environments. The policy π is a neural network, i.e., a Transformer decoder (Fig. 4(c)) which like the reward model, is also initialized by the pre-trained language model ρ. At step t when a payload τ is generated, the state s_t is the already generated sequence $x_{0:t}$ of τ, and the action a_t is to select from all token IDs the next token ID x_{t+1}, which will be appended to the state for the next generation step $t + 1$.

Our RL process is shown in Algorithm 1. We use Proximal Policy Optimization (PPO) [17] to train the policy network. PPO is an on-policy actor-critic RL algorithm that can improve training stability by avoiding those parameter updates that may change the policy too much at a single step. The actor corresponds to the policy π for choosing an action, i.e., emitting next token during payload generation; and the critic corresponds to the value function $V(\cdot)$ (also implemented by a neural network) which estimates the return of the current state. As a common practice in deep reinforcement learning (e.g., AlphaGoZero [18]), the policy network and the value network in GPTFuzzer share a body (i.e., the transformer decoder) and are incarnated with the language modeling head and the value prediction head respectively (Fig. 4(c)).

In Algorithm 1, GPTFuzzer first generates a set of payloads using the policy network (line 4), computes rewards for these

Algorithm 1. RL With Proximal Policy Optimization

Input: pre-trained language model ρ; reward model $r : X \to \mathbb{R}$ (X is payloads space); language model parameterized by θ, acting both as policy π (with language modeling head) and as value function V (with the value prediction head); KL penalty coefficient β; clipping threshold ϵ; stop threshold δ.

1: Initialize θ with parameters of pre-trained language model ρ.
2: $stopCount = 0$
3: **for** $k = 0, 1, 2 \ldots$ **do**
4: Generate a set of payloads $D_k = \{\tau_i\}$ using policy network: $\tau_i = Generate(\pi_{\theta_k})$.
5: **for** each payload $\tau_i \in D_k$ generated in T steps **do**
6: Compute reward for the final step T using the reward model: $R_T^{WAF} = r(\tau_i)$.
7: Compute KL-divergence penalty for all intermediate steps $t \in \{0, \ldots, T-1\}$:

$$KL_t(\pi_{\theta_k}, \rho) = \log \frac{\pi_{\theta_k}(x_{t+1}|x_{0:t})}{\rho(x_{t+1}|x_{0:t})}$$

8: Combine reward and KL-divergence penalty to get the total reward $\{R_t\}$:

$$R_t = \begin{cases} -\beta KL_t(\pi_{\theta_k}, \rho) & \text{for } t \in \{0, \ldots, T-1\} \\ R_T^{WAF} & \text{for } t = T \end{cases}$$

9: Compute advantage estimates $\hat{A}_t$ and return estimates $\hat{R}_t$ based on total reward R_t and the current value function $V_{\theta_k}(\cdot)$.
10: **end for**
11: Compute policy loss (PPO-Clip): $\mathcal{L}_{\theta_k}^{policy} = -\frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \min\Big(r_t(\theta)\hat{A}_t, \text{clip}\big(r_t(\theta), 1-\epsilon, 1+\epsilon\big)\hat{A}_t\Big)$, where $r_t(\theta) = \frac{\pi_\theta(x_{t+1}|x_{0:t})}{\pi_{\theta_k}(x_{t+1}|x_{0:t})}$.
12: Compute value function loss (mean-squared error): $\mathcal{L}_{\theta_k}^{value} = \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \big(V_\theta(x_{0:t}) - \hat{R}_t\big)^2$.
13: Update the parameters θ by minimizing policy loss and value loss via stochastic gradient descent: $\theta_{k+1} = \arg\min_\theta \Big(\mathcal{L}_{\theta_k}^{policy} + \mathcal{L}_{\theta_k}^{value}\Big)$.
14: Compute average reward for this epoch: $R_{avg}^k = \frac{1}{|D_k|} \sum_{\tau \in D_k} R_T^{WAF}$.
15: **if** $k \geq 1$ and $\frac{R_{avg}^k - R_{avg}^{k-1}}{R_{avg}^{k-1}} \leq \delta$ **then** $stopCount = stopCount + 1$.
16: **if** $stopCount == 3$ **then** break.
17: **end for**

payloads (lines 5-8), computes advantage estimates and returns estimates based on the rewards (line 9), and updates the policy and value networks using these data via PPO algorithm (lines 11-13). Next GPTFuzzer interacts with the environment using the updated policy and value networks and the above process repeats. The algorithm terminates when the average rewards for consecutive epochs plateau (lines 14-16).

Algorithm 2. Payload Sequence Generation From the Policy Network

1: **function** Generate$policy\ \pi$
2: $t = 0,\ x_0 = ID_{\langle start \rangle},\ X = [x_0,]$.
3: **while** $x_t \neq ID_{\langle end \rangle}$ **and** $t < T_{max}$ **do**
4: Compute prob. distribution for next ID: $probs_{t+1} = \pi(X)$.
5: Sample from distribution to get next ID: $x_{t+1} \sim probs_{t+1}$.
6: $X = \text{Append}(X, x_{t+1}),\ t = t + 1$.
7: **end while**
8: **return** X
9: **end function**

To mitigate the local optimum issue in optimization, i.e., optimizing too much towards some particular attack patterns while ignoring others, we amend the reward (i.e., the probability of bypassing WAF) by adding a Kullback-Leibler (KL) divergence penalty $KL(\pi_{\theta_k}, \rho)$ (line 7). The KL penalty, which quantifies how much the probability distribution given by the policy π differs from the reference probability distribution given by the pre-trained language model ρ, prevents the policy π from deviating too much from the pre-trained language model ρ. The combined rewards for a payload of length $T + 1$ consist of the WAF-bypassing probability given by the reward model for the final step T and the negative KL-divergence for intermediate steps (line 8). Empirically, this penalty can reduce generating the same payloads repetitively while maintaining a high bypassing rate (Section IV-E).

E. Generation

The RL fine-tuned policy network is used to generate payload sequences token-by-token, which will be tested on the target WAF. The generation process is shown in Algorithm 2. The transformer decoder generates a sequence of token IDs for each payload sequentially from left to right. The decoder first generates x_1 based on x_0, then x_2 based on x_0 and x_1, and so on, until the ID for $\langle end \rangle$ is generated or a predefined maximum length T_{max} is reached (lines 3-7). For each step, the distribution for the next token ID is computed by feeding the already generated sequence to the policy π (line 4), and a token ID is sampled from this distribution (line 5).

III. Empirical Study

A. Research Questions

To explore the performance of GPTFuzzer, we aim to answer the following questions:

RQ1: Is GPTFuzzer effective and efficient?
RQ2: How do attack grammars affect the effectiveness and the efficiency of GPTFuzzer?
RQ3: How does pre-training the language model with different data sizes contribute to GPTFuzzer?
RQ4: What benefits does it bring to guide RL with the reward model instead of WAF testing results?

RQ5: How do two hyper-parameters in RL, i.e., KL-divergence penalty coefficient and size of training data for reward model, contribute to GPTFuzzer?

RQ6: Is GPTFuzzer able to generate bypassing payloads that can not be generated from the attack grammar?

B. Subject WAFs

In our case studies, we evaluate GPTFuzzer against two famous open-source WAFs: ModSecurity[7] and Naxsi.[8]

ModSecurity is a widely deployed WAF that provides real-time protection for web applications by implementing the OWASP core rule set (CRS). The OWASP core rules are maintained by an active community of security experts to defend against various kinds of attacks, e.g., SQLi, XSS, RCE, and Denial of Service. We deployed ModSecurity 3.0.5 and OWASP CRS 3.3.2 with an Apache web server 2.4.29 on a local physical machine.

Naxsi is a third-party Nginx module which uses a small set of rules containing 99% of known patterns of web vulnerabilities such as SQLi, XSS, File Uploads and HTTP header Tracking, etc. We deployed Naxsi 1.3 with an Nginx web server 1.18 in our experiment machine.

C. Procedure

In our experiments, we test two WAFs with the web application under protection. We use as target application DVWA (damn vulnerable web application),[9] a popular web vulnerability demonstration and penetration testing platform. We test payloads using HTTP GET parameter for SQLi and XSS, and POST parameter for RCE. For example, to perform SQLi test against a DVWA page which supports HTTP GET query via a key id, we use an attack payload "*0%0band%0bnot˜false%23*" as the query string for id and send a request like "*http://localhost/DVWA/vulnerabilities/sqli/?id=0%0band%0bnot˜false%23&Submit=Submit#*". A payload is labeled as bypassing if the WAF responds with status code 200 (success); otherwise, the payload is labeled as blocked. Bypassing is used to define our main evaluation metrics in Section III-G (i.e., TP), because it can be automatically obtained for all the three attack types.

Besides bypassing the WAFs, it is also important to know whether the generated payloads are *functional*, i.e., can actually attack the applications. For example, for SQLi, a payload is labeled functional if the response body contains data of the database table queried by the request. We randomly sample 100 bypassing payloads generated by GPTFuzzer for each attack type, and find functional ones by manual inspection. All the experiments are repeated 5 times and the average values are reported. Note that in RAT paper [7] bypassing payloads are measured and their functionality is not considered.

[7]https://github.com/SpiderLabs/ModSecurity
[8]https://github.com/nbs-system/naxsi
[9]https://dvwa.co.uk /

TABLE I
DATASET STATISTICS FOR RAT AND GPTFUZZER. FORM x/y MEANS THAT x PAYLOADS IN TOTAL y ONES ARE USED IN THE EXPERIMENTS

Attack Type	#RAT Payloads	#GPTFuzzer Payloads
SQLi	4,000,000 / 162,109,992	512,000 / 162,109,992
XSS	4,253,696 / 4,253,696	512,000 / 4,253,696
RCE	37,302 / 37,302	37,302 / 37,302

D. Experimental Environment

We implemented GPTFuzzer with Python 3.8.5, Pytorch 1.7.1, and Hugging Face Transformers.[10] We ran all the experiments on a 64-bit server with 24 cores (Intel(R) Xeon(R) CPU E5-2643 v4 @ 3.40 GHz), 256 GB memory and 4 GPUs (24 GB Nvidia GeForce RTX 3090), and Ubuntu 18.04 system. All the experiments are repeated 10 times and the average values are reported.

E. Baseline Approaches

We compare GPTFuzzer with two state-of-the-art methods for testing WAFs: a mutation-based method ML-Driven E [2], and a search-based method RAT [7]. We also compare GPTFuzzer with a simple random payload generation procedure (Random Fuzzer) as a baseline.

1) ML-Driven E: It splits each payload into slices corresponding to subtrees of its parse tree and applies a machine learning method and evolutionary algorithm to select payloads and their specific slices to mutate to generate new payloads. In our experiments, we use ML-Driven E (Enhanced), an adaptive variant of ML-Driven that achieves the best performance among the ML-Driven family. As ML-Driven is not publicly available, we use the open-source implementation by RAT authors.[11]

2) RAT: It tokenizes attack payloads using n-gram, clusters similar ones and then uses an ϵ-greedy reinforcement learning algorithm to search the clusters containing bypassing payloads. We use the open-source implementation by its authors.[12]

3) Random Fuzzer: It randomly samples the payload space defined by the attack grammar and is indeed able to generate attack payloads that can bypass WAF. So we take it as a baseline method.

F. Datasets

We evaluate GPTFuzzer and three baseline approaches on three datasets that correspond to three common web attacks respectively: SQLi, XSS, and RCE. Below we describe how these datasets are constructed. Since ML-Driven needs attack grammar to guide payload mutation, for a fair comparison, we construct the attack grammars for the three attacks by ourselves as described in Section II-A and use the grammars to generate the datasets for RAT and GPTFuzzer. Moreover, for each kind of attack, we attempt to generate the full set of attack payloads (i.e., y in Table I) using the above constructed grammars because RAT requires searching within it.

[10]https://github.com/huggingface/transformers
[11]https://github.com/mhamouei/ml-driven
[12]https://github.com/mhamouei/rat

Table I shows the statistics of our datasets. The second and third columns show the number of payloads used by RAT and GPTFuzzer respectively. These datasets are used by RAT to search bypassing payloads and by GPTFuzzer to pre-train its language model. To facilitate RAT's best performance, we use the full set of payloads for XSS and RCE. For SQLi, we use 4 million payloads, which is twice the number used in RAT [7], because our analysis reveals that the computation cost of RAT's search algorithm scales quadratically with the number of payloads, and our preliminary experiments show that searching the full set of payloads for SQLi would be computationally prohibitive. In contrast to RAT's reliance on very large datasets, GPTFuzzer can outperform three baseline methods even using 0.5 million payloads for XSS and SQLi, as shown in Sections IV-A and IV-C.

G. Evaluation Metrics

1) True Positives (TP): the number of discovered bypassing payloads within a given number of requests, which is the ultimate evaluation metric for a testing method.

2) Time Spent per Request (TSR): the time of generating a payload plus the time of testing it against a WAF.

3) Effective Rate (ER): The proportion of generated bypassing payloads (TP_{gen}) over all the distinct generated payloads ($N_{distinct}$) in the generation phase, i.e., the bypassing rate of the generated attack payloads against a target WAF:

$$ER = TP_{gen}/N_{distinct}$$

4) Non-Repetition Rate (NRR): The proportion of distinctly generated payloads ($N_{distinct}$) over all the generated payloads (N) in the generation phase, i.e., the diversity of the generated attack payloads:

$$NRR = N_{distinct}/N$$

5) Total Effective Rate (TER): The proportion of all bypassing payloads over all tested payloads which are counted during both the reward model training phase ($|D_r|$) and the generation phase ($N_{distinct}$). TER differs from ER by also considering the sample efficiency of the reward model training phase:

$$TER = (TP_{|D_r|} + TP_{gen})/(|D_r| + N_{distinct})$$

TP, as the main metric, is used for comparing GPTFuzzer with the other methods (RQ1) and evaluating its effectiveness (RQ2-4). TSR is used for evaluating GPTFuzzer's efficiency (RQ1-2). ER, NRR and TER are used for studying the effects of hyper-parameters in RL (RQ5).

H. Training Hyper-Parameters

We adopt the recommendation values in the literature (e.g., [14], [17]) for most of the hyper-parameters. For language model pre-training and reward model training, we select the optimal values of the learning rate, batch size, and the number of epochs on the validation sets to minimize validation loss. For RL, we first determine the number of training epochs with Algorithm 1 (lines 15-16) and then select the values of learning rate and batch size at epoch 5 to maximize the average reward obtained at this moment. We evaluate epoch 5 for each attack type because the average reward grows rapidly before it, as shown in Fig. 5.

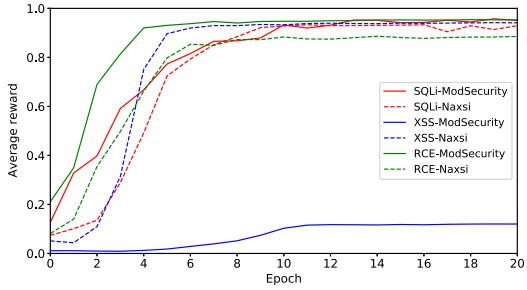


Fig. 5. Average rewards in RL for three types of attack against two WAFs.

1) Language Model Pre-Training: To pre-train the generative language model, we use the gpt-2 model architecture [14] with the *default* parameter settings, which is available in Hugging Face transformers library.[13] It is a decoder-only Transformer with 12 blocks, which has 768 embedding dimensions, 12 attention heads, and 3072 inner layer dimension in its feed-forward network.

We randomly divide the pre-training dataset with proportions 80%, 10%, 10% for training, validation, and testing respectively. We use the Adam optimizer [19] to train the language model. The learning rate is increased linearly from 0 to 1e-4 over the first 1/10 of total update steps and annealed linearly to 0 afterwards. We train the language model for 4 epochs with batch size of 32.

2) Reward Model Training: The reward model training dataset is divided into proportions 70%, 15%, 15% for training, validation, and testing respectively. Adam optimizer is also used, and the learning rate is increased linearly from 0 over the first 1/10 update steps to 2e-5 and then annealed linearly to 0. We set 4 epochs and batch size of 32 to train the reward model.

3) Reinforcement Learning: We use the PPO algorithm with the *default* parameter settings including clipping threshold (i.e., ϵ) 0.2. The learning rate and batch size are set to 1.4e-5 and 256 respectively. As demonstrated in Section IV-E, we set the KL-divergence penalty coefficient (i.e., β) as 0.2 for optimal results.

IV. Experimental Results

In this section, we present experimental results and answer the research questions raised in the previous section.

A. RQ1: Is GPTFuzzer Effective and Efficient?

To evaluate GPTFuzzer's effectiveness, we compared it with ML-Driven E, RAT, and Random Fuzzer on the datasets for three attack types. The reward model of GPTFuzzer was trained by using 4,000, 2,000, and 2,000 labeled payloads for SQLi, XSS, and RCE respectively, which is explained in RQ5 (Section IV-E). We measured the TP values of GPTFuzzer and three baselines over 1,250,000, 35,000, and 35,000 requests for SQLi, XSS and RCE, respectively.

[13] https://huggingface.co/gpt2

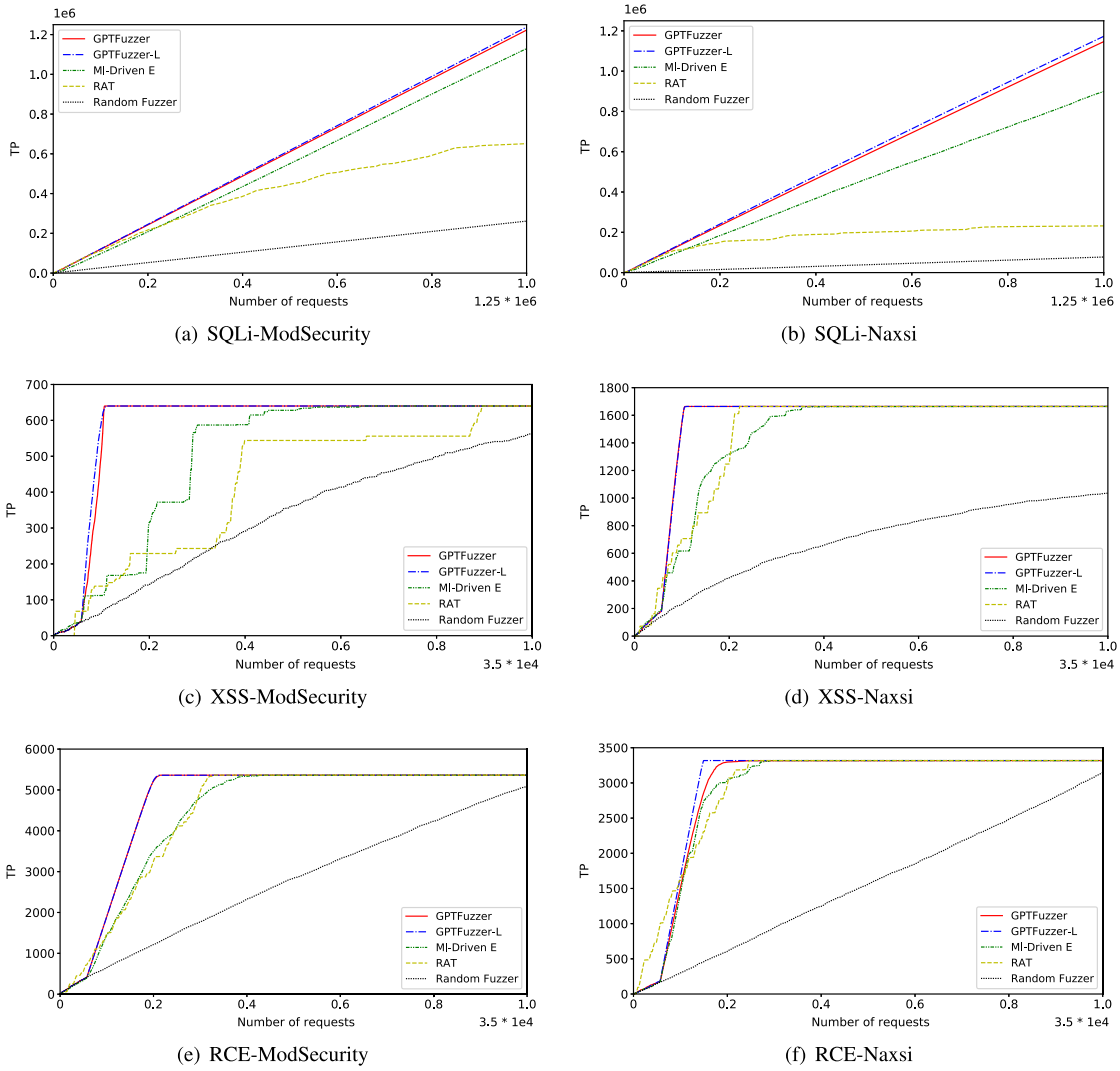


Fig. 6. TPs for three attack types against two WAFs.

Fig. 6 depicts the results. First, all approaches can generate payloads that bypass two target WAFs, while Random Fuzzer performs the worst. Of the bypassing payloads, the percentages of functional ones in GPTFuzzer are 99.0%, 56.8% and 83.3% for SQLi, XSS, and RCE respectively with an average of 79.7%. By comparison, the corresponding values for three baseline methods are much similar because they use the same grammar as GPTFuzzer for each attack type and random sampling operations in all approaches cause the slight difference (within $\pm 4.2\%$).

Second, the sharper a curve increases, the more efficient its corresponding approach is, because an ideal WAF-testing approach aims to get as many effective payloads (TP) as possible with as few requests as possible. GPTFuzzer has higher TP values (i.e., sharpest curves) and significantly outperforms other methods on three attacks against two WAFs. For SQLi, as the payload space is very large and can not be exhausted within an affordable number of requests by all approaches, we measure and compare TPs given a fixed number of requests. GPTFuzzer finds up to 7.8× (3.2× on average) more TPs than ML-Driven E and RAT within 1,250,000 requests. For XSS and RCE, since all approaches except Random Fuzzer can find all bypassing payloads in their limited payload space, we measure and compare the number of requests used to find all bypassing payloads. GPTFuzzer consumes up to 8.1× (3.3× on average) fewer requests than ML-Driven E and RAT.

As described in Section II, three models in GPTFuzzer work together and are trained sequentially. Therefore, we also reported the intermediate evaluation results of these three models, on which the final performance of GPTFuzzer has a dependency.

First, in the language model pre-training phase, we evaluated the capability of the pre-trained language model itself. Table II shows the perplexities of the pre-trained language models on test sets when using 512 K pre-training data (as shown in Table I). Perplexity is defined as the exponentiated average negative log-likelihood of a sequence and is one of the most common metrics for evaluating language models. Intuitively, it measures how well a language model can predict the next token in the test set. A perplexity of n means that the model is as confused on test data as if it had to choose uniformly among

TABLE II
PERPLEXITIES OF PRE-TRAINED LANGUAGE MODELS AND PERCENTAGES OF VALID PAYLOADS GENERATED BY THE MODELS

Attack Type	Perplexity	Percentage of Valid Payloads
SQLi	1.92	99.79%
XSS	1.61	99.89%
RCE	1.46	99.92%

TABLE III
PERFORMANCE OF THE REWARD MODELS IN GPTFUZZER

WAF	SQLi		XSS		RCE	
	F1%	AUC%	F1%	AUC%	F1%	AUC%
ModSecurity	95.15	99.06	90.37	98.68	97.30	99.63
Naxsi	92.35	98.82	97.11	99.58	98.46	99.95

n possibilities for each next token, and thus a lower perplexity indicates a better model. Perplexities of our three pre-trained models are very low, meaning that the next tokens are highly predictable to the models. In other words, the models can get the correct answers by making less than two guesses on average. Table II also reports the percentage of valid (i.e., conforming to the attacks' implicit grammatical requirements) payloads generated by the pre-trained language models using Algorithm 2. It can be seen that the payloads generated by the pre-trained models are almost valid. The capability of the language model is transferred to the reward model and policy network by parameter initialization.

Second, in the reward model training phase, we evaluated the trained reward models using a classification task to explore if they can provide accurate learning signals for RL. In the task, the reward models are used to classify payloads in labeled datasets as bypassing or blocked. A payload is classified as bypassing if the probability given by the reward model is above a threshold. A threshold of 0.5 is used in all settings except XSS-ModSecurity, for which the threshold should be less than 0.5 because its training set is extremely unbalanced with very small proportions of positive (bypassing) examples. By calculating the F1 values under multiple thresholds, e.g., 0.1, 0.2, 0.3, 0.4, we set the threshold to 0.1 with which the reward model achieved the maximum F1 value. The results are shown in Table III. The high F1 and AUC scores indicate that GPTFuzzer's reward models are trained properly and have reliable classification ability. We owe the good performance to the reward models' pre-trained transformer backbone, which excels at capturing long-distance dependency in the payload sequence and can learn the proper representation of payloads for the classification task. It has been widely recognized in the literature that the transformer is more powerful than other machine learning and deep learning models (e.g., Random Forest, CNN/RNN/LSTM) in sequence classification tasks.

Finally, in the RL phase, the policy network is trained to generate the attack payloads to maximize the expected reward, so we reported the average reward for payloads generated per epoch by the policy network for each setting, as shown in Fig. 5. One can observe that for XSS-ModSecurity and the rest settings, with the increase of the training epoch, the average reward gradually approaches 0.1 and 1.0 respectively, indicating that the training of the policy network is effective. The converged average reward for XSS-ModSecurity is lower because the training set of its reward model is extremely unbalanced with a very small proportion of bypassing payloads. As a result, the rewards for the policy network, i.e., the bypassing probabilities given by the reward model are mostly below 0.2.

TABLE IV
AVERAGE PAYLOAD SEQUENCE LENGTH IN TOKENS AND GENERATION TIME WITH GPTFUZZER (SHORT) AND GPTFUZZER-L (LONG)

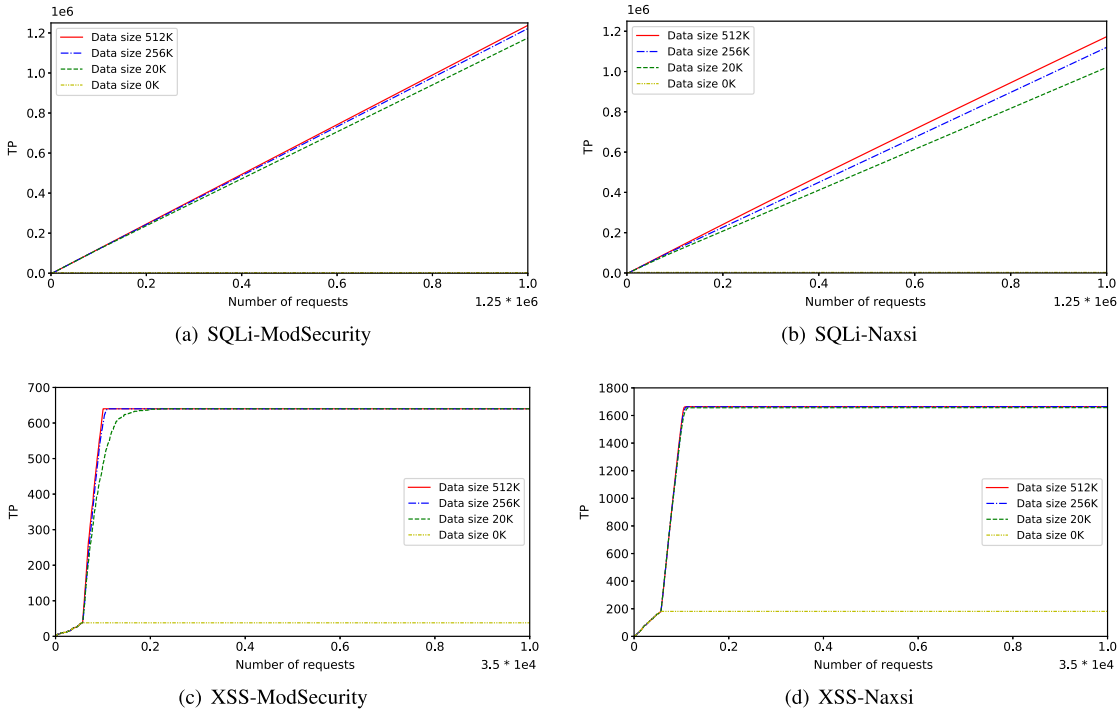
Metrics	SQLi		XSS		RCE	
	Short	Long	Short	Long	Short	Long
Seq. Length	26	61	15	30	20	34
TSR- (ms)	15.44	63.45	6.17	19.07	11.07	26.10

To evaluate GPTFuzzer's efficiency, we measured the average training time of three phases of GPTFuzzer. For SQLi, XSS, and RCE, language model pre-training took 21.38, 16.94, and 18.25 minutes, reward model training took 3.65, 3.32, and 3.48 minutes, and RL took 18.87, 16.52, and 17.64 minutes, respectively. We also measured average TSR, which is 20, 15, and 18 ms for SQLi, XSS, and RCE respectively. The TSR of GPTFuzzer is much less than that of RAT and ML-Driven E (refer to Section V-D in [7]), although the results are not directly comparable: GPTFuzzer uses GPU due to its relatively large policy network, whereas RAT and ML-driven E use CPUs.

B. *RQ2: How Do Attack Grammars Affect the Effectiveness and the Efficiency of GPTFuzzer?*

As mentioned in Section II-A, GPTFuzzer can use short payload sequences with terminal symbols or long payload sequences with all symbols (in this case, we name it GPTFuzzer-L). Note that we use short sequences to mimic manually collected payloads in absence of an attack grammar, and long sequences to include more information in the given attack grammar. To answer RQ2, we trained and tested both GPTFuzzer and GPTFuzzer-L using the same setting as RQ1. The TP curves of GPTFuzzer-L are also shown in Fig. 6. GPTFuzzer-L has slightly higher TPs than GPTFuzzer due to the additional information in the given grammar.

To further evaluate their efficiency, we generated 1,250,000 payloads for three kinds of attacks against the ModSecurity WAF, recorded sequence length and TSR- (including only the generation time) for each payload, and reported the average values in Table IV. The average sequence length of GPTFuzzer-L is about twice that of GPTFuzzer, resulting in 3× more time to generate a payload. This means that with the same time budget, GPTFuzzer can test more payloads against WAFs, though GPTFuzzer-L has slightly higher TP. In a word, there is a trade-off between generation quality and speed for GPTFuzzer to use short or long payload sequences.



(a) SQLi-ModSecurity (b) SQLi-Naxsi

(c) XSS-ModSecurity (d) XSS-Naxsi

Fig. 7. TPs with different pre-training data sizes.

C. RQ3: How Does Pre-Training the Language Model With Different Data Sizes Contribute to GPTFuzzer?

To answer RQ3, we exploited different sizes of pre-training data to evaluate the effectiveness of GPTFuzzer. We measured TPs over 1,250,000 and 35,000 requests on SQli and XSS datasets, respectively. We omitted RCE because its small dataset may lead to a contingent result.

As shown in Fig. 7, GPTFuzzer performs poorly when pre-training is absent (i.e., data size is 0 K). The reason is that without pre-training, the policy network is not properly initialized in the RL phase, and thus has great difficulty in generating valid payloads, i.e., ones that conform to implicit grammatical requirements of the attack type. When using 20 K pre-training data, there are still 0.1% - 1% invalid payloads in all the payloads generated by the policy network, thus decreasing the TP. However, pre-training using (256 K) 512 K data size brings (almost) optimal results. Therefore, we used 512 K pre-training data in the RQ1 experiment and validated the result payloads using a utility program. Overall, these results show that the pre-training language model with sufficient data plays an important role in GPTFuzzer.

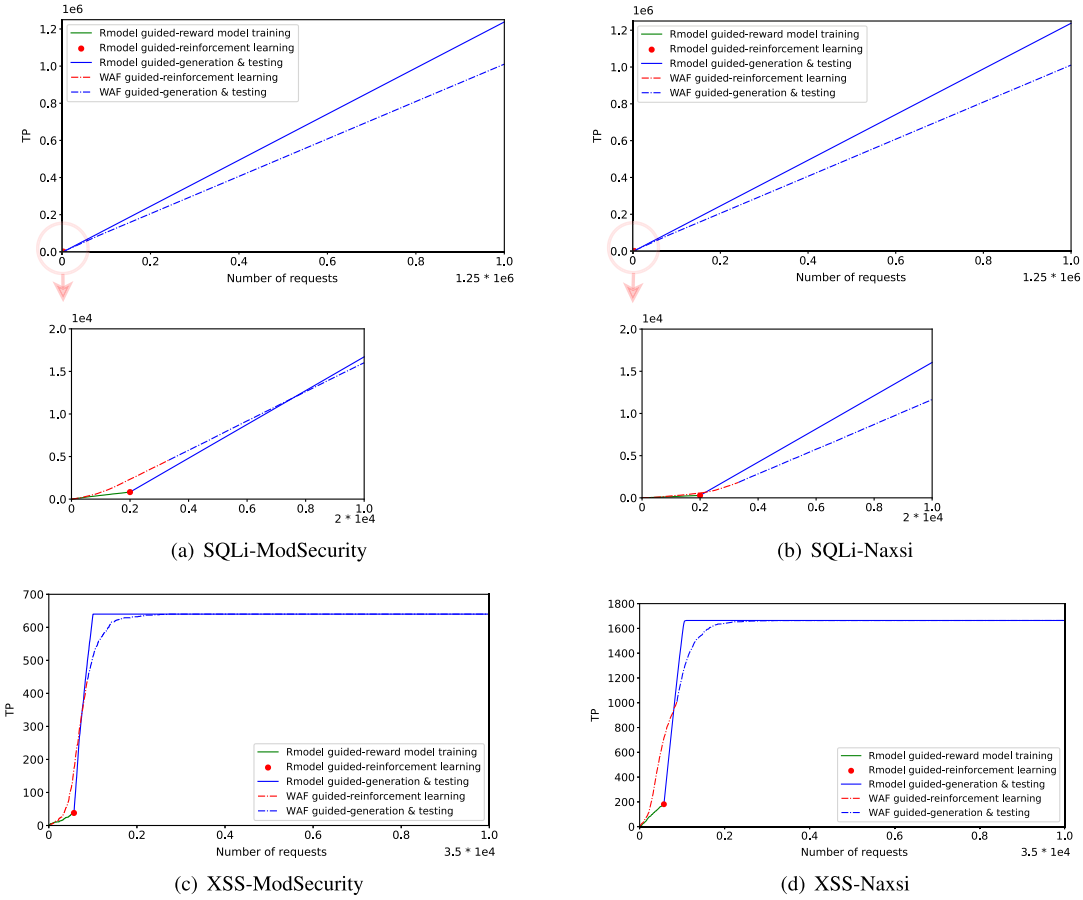
To study further that given only an attack grammar, if we could leverage it with RL to generate bypassing payloads without fine-tuning any language model at all, we carried on a new experiment as follows. For every non-terminal node of the grammar tree, the RL model learns a set of sampling weights for each branch which sum to one. For example, for the grammar in Fig. 3, if the sampling weights for <*numericCtx*> branch and <*sQuoteCtx*> branch at <*start*> node are 0.7 and 0.3 respectively, the <*numericCtx*> branch will be selected with probability 0.7 in the sampling process. These weights are learned using policy gradient with REINFORCE algorithm [20] which is a classical policy-based RL algorithm, with the WAF bypassing results as rewards. The results are shown in Table V. Although much better than Random Fuzzer, grammar-based RL is worse than GPTFuzzer with language models. It can be concluded that language models play an important role in GPTFuzzer by: 1) learning a better policy by capturing long-range dependencies (e.g., between two non-terminal nodes far apart in the grammar tree) via the powerful transformer architecture; 2) enabling GPTFuzzer to work without a predefined attack grammar by learning implicit grammatical requirements from payload datasets, which increases its practicability.

TABLE V
BYPASSING RESULTS FOR ATTACK GRAMMAR-BASED RL WITHOUT FINE-TUNING LANGUAGE MODEL. (GPTFUZZER RESULTS ARE COPIED FROM TABLE VII FOR EASY COMPARISON)

Attack Type-WAF	GPTFuzzer(NRR/ER)	Gram.+RL(NRR/ER)
SQLi-ModSecurity	98.59% / 99.52%	82.04% / 92.91%
SQLi-Naxsi	85.15% / 98.36%	76.12% / 92.23%
XSS-ModSecurity	28.67% / 35.44%	20.19% / 30.67%
XSS-Naxsi	19.46% / 92.73%	21.59% / 84.22%

D. RQ4: What Benefits Does it Bring to Guide RL With Reward Model Instead of WAF Testing Results?

To answer RQ4, we compared GPTFuzzer's reward model guided RL with a naive form of RL that uses direct WAF testing results as a reward (WAF guided RL). We conducted experiments using the same setting as RQ3 and the results are shown in Fig. 8.



(a) SQLi-ModSecurity (b) SQLi-Naxsi

(c) XSS-ModSecurity (d) XSS-Naxsi

Fig. 8. TPs comparison between reward-model (Rmodel) guided RL and WAF guided RL.

WAF guided RL consists of two phases (shown in dotted lines). First, the policy network is trained for the same number of epochs as the reward model guided RL for a fair comparison. Then the trained policy network is used to generate payloads that are then tested against the WAFs.

The Reward model guided RL has three phases (shown in solid lines). In the first phase, a labeled dataset is collected by testing a small amount of randomly generated payloads against the WAFs, and the reward model is trained on this dataset. Its TP (shown in green line) grows slower than that of WAF guided RL (shown in red line) because the payloads are randomly generated. In the second phase, the policy network is trained with reward signals provided by the above reward model, without interacting with any WAFs. In the generation and testing phase, after the policy network is well trained, its TP raises rapidly and surpasses WAF guided RL. These results indicate that the reward model provides richer learning signals for RL than direct WAF testing, bringing in better results.

To further explore why the reward function can derive a better policy network in training, for each attack type, we first sort the generated payloads by the bypassing probability predicted by the reward model, then randomly select three payloads with bypassing probabilities in the low, medium and high range (i.e., [0,t/10), [t/10,t) and [t,1]) respectively, where t is the probability threshold, and finally test these payloads against ModSecurity to confirm whether they can bypass the WAF. The results are shown in Table VI. One can see that the payloads with high bypassing probabilities are also confirmed bypassed by the WAF and that the payloads with high and medium bypassing probabilities share more tokens and n-grams than those with low probabilities, indicating that the reward model can help the policy network distinguish different payloads. Especially, the reward model assigns modest rewards to those blocked payloads which share some features with bypassing ones, telling the policy network that it is in the right direction. This reward signal is very helpful in the early stage of training when there are few bypassing payloads. By contrast, WAF testing only gives two results: bypassing, or blocked, blurring the above difference between different blocked payloads.

E. RQ5: How Do Two Hyper-Parameters in RL, i.e., KL-Divergence Penalty Coefficient and Size of Training Data for Reward Model, Contribute to GPTFuzzer?

To answer RQ5, we evaluated GPTFuzzer with different values of two hyper-parameters: Kl-divergence penalty coefficient β and the size of training data for the reward model. We used ER and NRR as evaluation metrics. For $\beta = 0.2$ settings, which were used to get the main results in Fig. 6, we additionally reported TER.

TABLE VI
EXAMPLES OF GENERATED PAYLOADS BY GPTFUZZER AND THEIR BYPASSING PROBABILITIES GIVEN BY THE REWARD MODELS (BYPASS. PROB.) AND THE TESTING RESULTS AGAINST MODSECURITY (TEST RESULT) RESPECTIVELY

Attck Type	Generated Payload	Bypass. Prob.	Test Result
SQLi	0%27or%0b1<@%0b||%0b%7e+{`if`+1}=1=%0b(~%0b{x+(select+1)}=1)%0bor%27	0.7645	Bypassing
SQLi	0%27||+1<@%0b||%0b~%0b{a+1}=1=%0b(%7e+{`if`%0b1}=1)+||'	0.4953	Blocked
SQLi	2%27%0b||+!%0btrue=%0b(not~false)+or'0	0.0010	Blocked
XSS	<input%0donblur%0a=%0aalert(XSS)%0dx//xss	0.1983	Bypassing
XSS	<input%0donchange%0d=+\u0061lert(XSS)%0dx>xss	0.0916	Blocked
XSS	<form%09tabindex=1%0aonblur%09=+alert(XSS)%0dx><input%0dtype=submit>	0.0062	Blocked
RCE	127.0.0.1&&s$@ort␣/etc/pa$@sswd	0.9990	Bypassing
RCE	127.0.0.1&a=s;b=ort;ab␣/etc/passwd	0.4586	Blocked
RCE	127.0.0.1;system("whi${19}ch␣my${15}sql")	0.0006	Blocked

TABLE VII
NON-REPETITION RATE (NRR) AND EFFECIVE RATE (ER) WITH DIFFERENT KL DIVERGENCE PENALTY COEFFICIENTS β AND SIZES OF TRAINING DATA FOR REWARD MODEL. SETTINGS MARKED WITH STAR ARE THE ONES USED TO GET THE MAIN RESULTS IN FIG. 6

Attack Type-	$\beta = 0$		$\beta = 0.2$(*)			$\beta = 0.5$		$\beta = 0.8$		$\beta = 1.0$	
WAF-Data Size	NRR	ER	NRR	ER	TER	NRR	ER	NRR	ER	NRR	ER
SQLi-ModSecurity-2000	52.26%	99.41%	97.29%	99.64%	98.08%	99.47%	99.24%	99.55%	98.41%	99.61%	93.02%
SQLi-ModSecurity-4000(*)	4.25%	96.61%	98.59%	99.52%	96.44%	98.95%	99.38%	99.44%	98.91%	99.58%	98.15%
SQLi-Naxsi-2000	46.86%	96.01%	96.00%	60.93%	59.83%	99.84%	47.63%	99.87%	17.48%	99.94%	15.55%
SQLi-Naxsi-4000(*)	3.82%	99.35%	85.15%	98.36%	94.26%	97.52%	77.75%	99.01%	77.62%	99.40%	63.20%
XSS-ModSecurity-2000(*)	0.19%	70.93%	28.67%	35.44%	18.01%	30.98%	32.84%	64.72%	13.66%	70.68%	12.52%
XSS-ModSecurity-4000	4.89%	86.81%	31.78%	45.89%	12.45%	37.46%	41.72%	68.94%	24.71%	75.42%	16.89%
XSS-Naxsi-2000(*)	0.25%	98.42%	19.46%	92.73%	36.85%	26.22%	91.67%	28.79%	91.58%	31.45%	91.04%
XSS-Naxsi-4000	3.64%	99.28%	25.02%	92.89%	30.31%	32.64%	92.35%	35.78%	91.79%	40.23%	91.16%

1) KL-Divergence Penalty Coefficient: As discussed in Section II-D, the KL-divergence penalty determines the deviation degree of the policy π that we want to learn from the pre-trained model ρ in the RL phase. We examined the effects of different KL-divergence penalty coefficient $\beta \in [0, 1]$.

Table VII shows the results. It can be seen that $\beta = 0$ makes the policy π not restricted by the pre-trained language model at all and optimizes solely towards getting higher expected rewards: although ER is very high, NRR is too low. The policy π falls into local optimum by repeatedly generating a fixed set of high rewarding payloads; On the other hand, setting β to 1 causes the policy π significantly restricted by the pre-trained language model: NRR is very high though ER is too low. Setting β in [0.2, 0.5] keeps a good balance between ER (bypassing rate) and NRR (diversity). So we use $\beta = 0.2$ in all other experiments.

2) The Size of Training Data for the Reward Model: The size of training data for the reward model determines the reward model's ability to accurately estimate bypassing probability of payloads and provide indicative reward signals for RL. Here we set it to 2,000 and 4,000 respectively to investigate the impact of data size, as shown in Table VII.

We have two observations from the results: 1) For β value 0.2 or 0.5, using 4,000 data can get a better balance between ER and NRR than using 2,000 data; 2) For an attack with smaller payload space and thus fewer total testing requests, e.g., XSS, its TER decreases when the training data for reward model increases, as shown in rows 6-9 in Table VII, because GPTFuzzer consumes a large proportion of total testing requests in the reward model training phase, as Random Fuzzer does. However, this does not hold for attacks with larger payload space and more testing requests (e.g., SQLi). Therefore, we used 4,000 data for SQLi and 2,000 data for XSS and RCE when training the reward models in RQ1 experiments.

TABLE VIII
NEW PAYLOADS GENERATED BY GPTFUZZER

Attack Type-WAF	#Payloads	#New	#New&Bypassing
SQLi-Modsecurity	500K	2818	1080
SQLi-Naxsi	500K	44017	17924

F. RQ6: Is GPTFuzzer Able to Generate Bypassing Payloads That can not be Generated From the Attack Grammar?

Although GPTFuzzer can generate lots of bypassing payloads against two target WAFs, we want to explore whether it can generate bypassing payloads that are *new* with regard to the original grammar, i.e., that can not be generated by the grammar. To this end, we generated 500 K attack payloads using GPTFuzzer for each attack type, recorded the new ones, and tested them against the target WAFs. As shown in Table VIII, for SQLi, GPTFuzzer generated new payloads, about 40% of which can bypass the target WAFs and are functional, showing the potential of GPTFuzzer to transcend the expertise contained in the attack grammar. No new payloads were generated for XSS and RCE because their very small payload spaces lead to a highly deterministic generation process, for example, their full set ratios over SQLi are 2.62% and 0.02% respectively, as shown in Section III-F.

V. Related Work

A. Learning-Based Web Security Testing

A few approaches have been proposed for machine-based black-box web security testing, the pioneering ones of which are mostly mutation-based. Tripp et al. [3] proposed XSS Analyzer, which selects effective payloads by learning from previous attack executions. Following their work, Appelt et al. [2] proposed ML-Driven that applies machine learning models and evolutionary algorithm to improve testing effectiveness. Specifically, payloads are scored and ranked by a decision tree or random forest classifier, and each of the top-ranked payloads is then mutated based on an evolutionary algorithm on the granularity of substrings of a payload. The substrings correspond to subtrees of the payload's parse tree, which represents the derivation steps of the payload from an attack grammar. Demetrio et al. [8] leveraged adversarial machine learning and an evolutionary mutation algorithm to generate payloads that could bypass the WAF. They start from a failing test case and randomly apply their predefined mutation operators to it until getting a successful one.

These mutation-based approaches usually rely on either a predefined attack grammar (e.g., [2]) or a predefined set of mutation operators (e.g., [8]), which need to be constructed by domain experts manually and not always be available. The predefined grammar or operators also limit the payload space, so the testing effectiveness is upper-bounded by their quality and/or quantity. Moreover, they are usually outdated without updates in time.[14] As a generation-based method, GPTFuzzer does not rely on attack grammars or mutation operators to generate payloads. When a grammar is available, GPTFuzzer can leverage it to be more effective and can transcend the expertise contained in the grammar by generating new bypassing payloads with regard to the grammar, as shown in Section IV-F. WAF-A-MoLE [8], as a gray-box testing approach against Machine Learning (ML)-based WAFs, relies on confidence values (similar to bypassing probabilities) assigned by the ML-based WAFs. This information may not be available for rule-based WAFs. In contrast, GPTFuzzer, as a pure black-box approach to rule-based WAFs, only needs the testing results against the WAF, trains a transformer-based reward model, and guides RL by providing the bypassing probability of a payload as the reward.

B. Reinforcement Learning for Network and Web Security

Hsu et al. [21] used deep reinforcement learning (RL) for a network intrusion detection system and evaluated their work with two well-known NIDS benchmark datasets. In the anti-jamming robot communications, Dai et al. [22] developed an RL algorithm for security exploration and used transfer learning to reduce the initial random exploration. To mitigate security attacks in cloud environments, Praise et al. [23] proposed a deep packet inspection-based firewall (RLPM) to prevent malicious attacks by verifying payload signatures. RLPM combines the potential of parallel pattern matching and RL simultaneously. Feng et al. [24] propose an RL-based approach to L7 DDoS attack defense. Closely related to our work, Amouei et al. [7] proposed RAT that uses RL and adaptive testing to discover SQLi and XSS vulnerabilities in WAFs. Hemmati et al. [25] use deep RL to evade WAFs with SQLi payloads. One can refer [26] for a comprehensive survey of RL for cyber security.

These efforts use RL in a coarse-grained way; to get a reward (e.g., produce a payload to test against the WAF), their agents need to perform only one action chosen from a small space (e.g., dozens of payload clusters [7] or several mutation operators [25]). The RL task is relatively simple and the models used are rudimentary, e.g., feed-forward neural networks [24], [25]. Different from these works, RL in GPTFuzzer is more fine-grained: to generate a payload, the policy network needs to perform dozens of actions, each producing a token chosen from hundreds of tokens. On the one hand, it has the potential of getting better results by exploring in a larger space. On the other hand, the RL task becomes harder so more advanced model architecture (e.g., Transformer) and training techniques (e.g., pre-training and reward modeling) are needed.

C. Deep Learning and Reinforcement Learning in NLP

The pre-training fine-tuning paradigm has been well established in Natural Language Processing (NLP) [14], [27]. For example, RL with reward modeling was used for NLP tasks like summarization [28], long-form text generation [29], and dialog response generation [30], showing better results than traditional supervised learning in improving coherence. KL-divergence constraint was proposed for RL-based sequence generation [31] and later pursued RL-based sentiment control and text summarization from human preference [32] and feedback [33]. It is also an ingredient of the TRPO RL algorithm [34] to prevent the new policy from deviating too much from the old policy between updates and thus improve training stability. Motivated by the above studies, we integrate these techniques, i.e., pre-training and fine-tuning, reward modeling, and KL-divergence penalty, in GPTFuzzer to explore whether they are also effective for black-box WAF testing.

VI. Conclusion

Automatic WAF testing is important for revealing and fixing weaknesses in WAFs. In this paper, we propose GPTFuzzer, a generation-based black-box WAF testing approach that generates bypassing payloads token-by-token against a target WAF by fine-tuning a Generative Pre-trained Transformer language model with reinforcement learning. GPTFuzzer is practical, working without reliance on explicit attack grammars or a full set of attack payloads; and effective, mitigating the local optimum problem by combining multiple techniques such as reward modeling and KL-divergence penalty from reinforcement learning for NLP. We evaluate GPTFuzzer on two well-known WAFs (ModSecurity and Naxsi) against three typical web attacks, i.e., SQLi, XSS, and RCE. Evaluation results show that with fewer restrictions, GPTFuzzer significantly outperforms state-of-the-art mutation-based and search-based approaches, i.e., ML-Driven and RAT. We make attack grammars for three

[14] We found in experiments that the attack grammar for SQLi proposed in ML-Driven is useless for the newer ModSecurity and CRS version than ones used in ML-Driven.

typical web attacks and our datasets publicly available in the hope of facilitating further research in the web security field. We believe that GPTFuzzer, as a general and powerful fuzzing approach, can have other potential applications besides WAF testing, which is also our future work.

REFERENCES

[1] Symantec, "2019 internet security threat report," 2020. [Online]. Available: https://docs.broadcom.com/doc/istr-24-executive-summary-en

[2] D. Appelt, C. D. Nguyen, A. Panichella, and L. C. Briand, "A machine-learning-driven evolutionary approach for testing web application firewalls," *IEEE Trans. Rel.*, vol. 67, no. 3, pp. 733–757, Sep. 2018.

[3] O. Tripp, O. Weisman, and L. Guy, "Finding your way in the testing jungle: A learning approach to web security testing," in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 347–357.

[4] D. Appelt, C. D. Nguyen, and L. Briand, "Behind an application firewall, are we safe from SQL injection attacks?," in *Proc. IEEE 8th Int. Conf. Softw. Testing Verification Validation*, 2015, pp. 1–10.

[5] L. Zhang, D. Zhang, C. Wang, J. Zhao, and Z. Zhang, "ART4SQLi: The ART of SQL injection vulnerability discovery," *IEEE Trans. Rel.*, vol. 68, no. 4, pp. 1470–1489, Dec. 2019.

[6] C. Lv, L. Zhang, F. Zeng, and J. Zhang, "Adaptive random testing for XSS vulnerability," in *Proc. 26th Asia-Pacific Softw. Eng. Conf.*, 2019, pp. 63–69.

[7] M. Amouei, M. Rezvani, and M. Fateh, "RAT: Reinforcement-learning-driven and adaptive testing for vulnerability discovery in web application firewalls," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 5, pp. 3371–3386, Sep./Oct. 2022.

[8] L. Demetrio, A. Valenza, G. Costa, and G. Lagorio, "WAF-A-MoLE: Evading web application firewalls through adversarial machine learning," in *Proc. 35th Annu. ACM Symp. Appl. Comput.*, 2020, pp. 1745–1752.

[9] N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira, "Effective detection of SQL/XPath injection vulnerabilities in web services," in *Proc. IEEE Int. Conf. Serv. Comput.*, 2009, pp. 260–267.

[10] D. E. Simos, J. Zivanovic, and M. Leithner, "Automated combinatorial testing for detecting SQL vulnerabilities in web applications," in *Proc. IEEE/ACM 14th Int. Workshop Automat. Softw. Test*, 2019, pp. 55–61.

[11] M. Liu, K. Li, and T. Chen, "DeepSQLi: Deep semantic learning for testing SQL injection," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 286–297.

[12] A. Avancini and M. Ceccato, "Security testing of web applications: A search-based approach for cross-site scripting vulnerabilities," in *Proc. IEEE 11th Int. Work. Conf. Source Code Anal. Manipulation*, 2011, pp. 85–94.

[13] D. E. Simos, K. Kleine, L. S. G. Ghandehari, B. Garn, and Y. Lei, "A combinatorial approach to analyzing cross-site scripting (XSS) vulnerabilities in web application security testing," in *Proc. IFIP Int. Conf. Testing Softw. Syst.*, Springer, 2016, pp. 70–85.

[14] A. Radford et al., "Language models are unsupervised multitask learners," *OpenAI Blog*, vol. 1, no. 8, 2019, Art. no. 9.

[15] T. B. Brown et al., "Language models are few-shot learners," in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2020, Art. no. 159.

[16] A. Vaswani et al., "Attention is all you need," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017, *arXiv:1707.06347*.

[18] D. Silver et al., "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Representations*, 2015, pp. 1–13.

[20] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, no. 3, pp. 229–256, 1992.

[21] Y.-F. Hsu and M. Matsuoka, "A deep reinforcement learning approach for anomaly network intrusion detection system," in *Proc. IEEE 9th Int. Conf. Cloud Netw.*, 2020, pp. 1–6.

[22] C. Dai, L. Xiao, X. Wan, and Y. Chen, "Reinforcement learning with safe exploration for network security," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2019, pp. 3057–3061.

[23] J. J. Praise, R. J. S. Raj, and J. B. Benifa, "Development of reinforcement learning and pattern matching (RLPM) based firewall for secured cloud infrastructure," *Wirel. Pers. Commun.*, vol. 115, no. 2, pp. 993–1018, 2020.

[24] Y. Feng, J. Li, and T. Nguyen, "Application-layer DDoS defense with reinforcement learning," in *Proc. IEEE/ACM 28th Int. Symp. Qual. Serv.*, 2020, pp. 1–10.

[25] M. Hemmati and M. A. Hadavi, "Using deep reinforcement learning to evade web application firewalls," in *Proc. 18th Int. ISC Conf. Inf. Secur. Cryptol.*, 2021, pp. 35–41.

[26] T. T. Nguyen and V. J. Reddi, "Deep reinforcement learning for cyber security," *IEEE Trans. Neural Netw. Learn. Syst.*, early access, Nov. 1, 2021, doi: 10.1109/TNNLS.2021.3121870.

[27] J. D. M.-W. C. Kenton and L. K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2019, pp. 4171–4186.

[28] Y. Wu and B. Hu, "Learning to extract coherent summary via deep reinforcement learning," in *Proc. 32nd AAAI Conf. Artif. Intell.*, 2018, pp. 5602–5609.

[29] W. S. Cho et al., "Towards coherent and cohesive long-form text generation," 2018, *arXiv:1811.00511*.

[30] S. Yi et al., "Towards coherent and engaging spoken dialog response generation using automatic conversation evaluators," in *Proc. 12th Int. Conf. Natural Lang. Gener.*, 2019, pp. 65–75.

[31] N. Jaques, S. Gu, D. Bahdanau, J. M. Hernández-Lobato, R. E. Turner, and D. Eck, "Sequence tutor: Conservative fine-tuning of sequence generation models with KL-control," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 1645–1654.

[32] D. M. Ziegler et al., "Fine-tuning language models from human preferences," 2019, *arXiv:1909.08593*.

[33] N. Stiennon et al., "Learning to summarize from human feedback," 2020, *arXiv:2009.01325*.

[34] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 1889–1897.

Hongliang Liang (Member, IEEE) received the PhD degree in computer science from the University of Chinese Academy of Sciences, China, in 2002. He leads the Trusted Software and Intelligent System Research Group, Beijing University of Posts and Telecommunications, China. His main research interests include system software, trustworthy software, and artificial intelligence. He has published more than 40 articles in high impact journals and blind peer-reviewed conferences. He is a member of the ACM, and serves as a reviewer for some prestigious journals, including the *IEEE Transactions on Software Engineering*, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *IEEE Transactions on Reliability*, *ACM Transactions on Intelligent Systems and Technology*, *Future Generation Computer Systems*, *Computers and Security* and *Journal of Systems and Software*, etc.

Xiangyu Li received the BSc degree in computer science and technology from Shandong Agricultural University, China, in 2020. He is currently working toward the MSc degree with the Beijing University of Posts and Telecommunications, China, working on deep learning and software testing.

Da Xiao received the BS and PhD degrees in computer science and technology from Tsinghua University, China, in 2003 and 2009, respectively. He is currently an assistant professor with the Beijing University of Posts and Telecommunications, China, working on deep learning for software engineering and NLP.

Jie Liu received the MSc degree in computer science from the China University of Petroleum (East China), in 2020. He is currently working toward the PhD degree with the Beijing University of Posts and Telecommunications, China. His research interests include vulnerability discovery and automated exploit generation.

Yanjie Zhou received the MSc degree in computer science from the Beijing University of Posts and Telecommunications, China, in 2021. He is currently a software engineer with Qihoo 360 Technology Co. Ltd, China, working on system and software security.

Aibo Wang received the BSc degree in computer science from QingDao University, China, in 2021. He is currently working toward the MSc degree with the Beijing University of Posts and Telecommunications, China, working on deep learning and mechanistic interpretability of large language models.

Jin Li received the MSc degree in computer science and technology from NUDT, in 2006. He is a senior engineer with the Nation Key Laboratory of Science and Technology on Information System Security. His research interests include information security and network security.