Amoeba: Circumventing ML-supported Network Censorship via Adversarial Reinforcement Learning

HAOYU LIU, The University of Edinburgh, United Kingdom
ALEC F. DIALLO, The University of Edinburgh, United Kingdom
PAUL PATRAS, The University of Edinburgh, United Kingdom

Embedding covert streams into a cover channel is a common approach to circumventing Internet censorship, due to censors' inability to examine encrypted information in otherwise permitted protocols (Skype, HTTPS, etc.). However, recent advances in machine learning (ML) enable detectin g a range of anti-censorship systems by learning distinct statistical patterns hidden in traffic flows. Therefore, designing obfuscation solutions able to generate traffic that is statistically similar to innocuous network activity, in order to deceive ML-based classifiers at line speed, is difficult.

In this paper, we formulate a practical adversarial attack strategy against flow classifiers as a method for circumventing censorship. Specifically, we cast the problem of finding adversarial flows that will be misclassified as a sequence generation task, which we solve with Amoeba, a novel reinforcement learning algorithm that we design. Amoeba works by interacting with censoring classifiers *without any knowledge of their model structure*, but by crafting packets and observing the classifiers' decisions, in order to guide the sequence generation process. Our experiments using data collected from two popular anti-censorship systems demonstrate that Amoeba can effectively shape adversarial flows that have on average 94% attack success rate against a range of ML algorithms. In addition, we show that these adversarial flows are robust in different network environments and possess transferability across various ML models, meaning that once trained against one, our agent can subvert other censoring classifiers without retraining.

CCS Concepts: • **Networks → Network privacy and anonymity**; **Network security**; • **Computing methodologies → Adversarial learning**.

Additional Key Words and Phrases: Censorship Circumvention, Traffic Classification, Reinforcement Learning, Adversarial Attacks

ACM Reference Format:
Haoyu Liu, Alec F. Diallo, and Paul Patras. 2023. Amoeba: Circumventing ML-supported Network Censorship via Adversarial Reinforcement Learning. *Proc. ACM Netw.* Vol. 1, CoNEXT3, Article 9 (December 2023), 25 pages. https://doi.org/10.1145/3629131

1 INTRODUCTION

Governments and control authorities in some countries carry out network traffic censorship routinely to restrict the citizens' access to online information that may be perceived by those regimes as politically, socially, or morally objectable. To maintain censorship effectiveness and combat circumvention, e.g., through traffic mimicry and randomization [54], state actors employ a range of tools including Deep Packet Inspection (DPI), protocol fingerprinting, and active probing.

Authors' addresses: Haoyu Liu, haoyu.liu@ed.ac.uk, The University of Edinburgh, 2 Charles St, Edinburgh, United Kingdom, EH8 9AD; Alec F. Diallo, alec.frenn@ed.ac.uk, The University of Edinburgh, 2 Charles St, Edinburgh, United Kingdom, EH8 9AD; Paul Patras, paul.patras@ed.ac.uk, The University of Edinburgh, 2 Charles St, Edinburgh, United Kingdom, EH8 9AD.

2834-5509/2023/12-ART9 $15.00
https://doi.org/10.1145/3629131

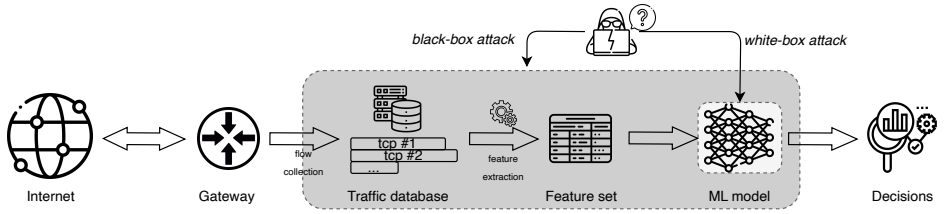

Fig. 1. Typical traffic classification pipeline. An attacker may conduct adversarial attacks against the pipeline with different capabilities/scope. We categorize them into 1) *white-box model* for which the inner workings of the classifier (weights, gradients) are visible; and 2) *black-box model* for which attackers can only craft network flows and observe outputs without extra knowledge, such as feature engineering and model architecture.

In recent years, protocol tunneling has gained traction as a viable means to circumvent censorship. Tunneling leverages existing implementations of innocuous protocols (Skype, WebRTC, TLS, etc.) and embeds covert streams in these protocols to hide destination host identity, payload contents, etc. [3, 4, 16]. As a result, sensitive information becomes encrypted and message exchanges perfectly aligned with the behavior of the tunneling protocol. In turn, observing the tunnels barely unveils any deterministic fingerprints. Censorship is however an arms race, recent studies revealing that Machine Learning (ML) algorithms, which learn statistical features from network flows, can effectively identify 'offending' tunneled traffic, despite not exhibiting deterministic fingerprints [11, 54]. For example, although multiple multi-media tunneling tools claim unobservability, simple ML classifiers such as those based on decision tree and random forest structures can detect tunneled traffic with high confidence [2]. On the other hand, ML is also employed to devise censorship circumvention strategies. For example, Geneva [7] designs a genetic algorithm to discover if existing censorship can be evaded by tampering with canonical TCP implementations, e.g., by corrupting checksums, breaking Transmission Control Blocks (TCBs) (by injecting a RST), or segmenting packets with corrupted ACKs. While this attack targets the incompleteness of network stacks implemented by censors, in this paper we aim to push the boundary of anti-censorship one step further, where we reasonably assume the censor fixes the implementation issues and leverages ML classifiers to detect anti-censorship tools.

Since the inner workings of an ML-based classifier are largely unknown to users seeking to circumvent censorship and the censor can change the underlying neural architecture at any time (black box), the question we aim to answer in this work is: *Instead of perpetually designing new tunneling tools, can we devise adversarial attacks on black-box ML classifiers to consistently subvert ML-supported censorship?* This approach has not been well studied in the network censorship domain. In computer vision, finding adversarial examples, i.e., images that should be recognized as belonging to class A being instead misclassified as of class B, can be achieved by adding adversarial perturbations such that the modified input images remain visually similar to their original versions, but produce erroneous classification results [1, 9, 19].

Conducting adversarial attacks in the networking domain is fundamentally different. A common approach to ML-based network flow classification is to first extract multiple statistical features (packet size distributions, timing information, etc.), then feed these features to a classifier instead of raw flows [2, 28], as illustrated in Figure 1. Censors do not reveal what features they utilize, which poses difficulty in the first step of crafting an adversarial attack. Further, even if users may discover the set of features employed by a censoring classifier and successfully generate adversarial samples, there is no guarantee that such samples can be mapped back to legitimate flows, which renders the entire process unusable in practice. A practical adversarial attack against censoring classifiers requires manipulation at packet level, instead of feature level, and each packet should be transmitted without adding significant delays. Early attempts [31, 53] apply attacks on complete

flows and generate adversarial versions, yet each manipulated packet should be sent before new packets are received. Having a complete view of a flow to perturb is unfortunately unrealistic. The inherent imbalance between what censors can observe (flows) and what users can observe and manipulate (packets) rules out the possibility of applying existing algorithms from other domains to achieve adversarial attacks for censorship circumvention purposes.

In this paper, we formulate the problem of finding adversarial flows against censoring classifiers as a packet sequence generation task. To solve it, we design Amoeba,[1] a novel black-box attack through reinforcement learning, which learns to craft adversarial flows solely based on the classification results of censoring classifiers, without any further knowledge. This leads to the following *contributions*:

(1) We make **no assumption about the underlying model of a censoring classifier**, which may or may not apply feature engineering and may not be differentiable (and hence approximating gradients impractical for generating adversarial flows), but instead treat the problem of finding adversarial flows as a process of generating sequences of packets that, when considered together as flows, will be misclassified.
(2) We propose Amoeba, a black-box attack based on Reinforcement Learning (RL) that **treats the classification results as rewards** and progresses with a policy that maximizes the expected future rewards (return). Our design incorporates a StateEncoder – a dedicated Neural Network (NN) that encodes arbitrarily long network flows into fix-sized hidden representations, to help the RL agent interpret the context of sequence generation at each timestep.
(3) We evaluate Amoeba on datasets collected using two popular anti-censorship systems, Tor and TLS tunneling; our experimental results indicate that the adversarial flows generated by our Amoeba have **~94% attack success rates, regardless of the type of ML classifier** a censor may deploy. We further show empirically that such adversarial flows are transferable across models with similar architectures.
(4) We demonstrate that the Amoeba is **stable** in different network environments, and **robust** when receiving noisy and unclear rewards during training.
(5) We discuss practical aspects and potential limitations of deploying our Amoeba as a transport layer extension, making the case for its adoption for mainstream censorship circumvention.

2 ADVERSARIAL MODELS

As use of ML gains traction in the networking domain, including for Website Fingerprinting (WF) [24, 34, 35, 38, 45, 46] and network intrusion detection [12, 28, 29], censors are increasingly adopting ML-based classifiers to detect unwanted protocols or traffic associated with banned web services. We consider the most common setting where the censor has full control of the network gateway and enough computational power to examine every network flow traversing it. More precisely, the censor may collect network traffic generated by 'forbidden' protocols/web services along with innocuous traffic. A group of statistical features may be extracted from individual flows and fed to a ML classifier for training, as shown in Figure 1. The censor then deploys the ML model on the gateway to block sensitive flows, e.g., by using and maintaining a blacklist of (src_ip, src_port, dst_ip, dst_port, protocol) tuples on the firewall. That said, once a traffic flow is recognized as 'unwanted' by the censor, the pair of sockets used on the source and destination machine cannot communicate to establish a connection. The censor would not block the destination IP entirely, in order to prevent collateral damage, especially as CDNs increasingly serve thousands of service

[1] Our censorship circumvention algorithm's name draws inspiration from the unicellular organism with the same name that is capable of altering its shape. Similarly, our solution alters the shape of network flows.

with the same address [14]. This is a reasonable practical assumption – for instance, the Great Firewall blocks port numbers instead of IP address when censoring Shadowsocks [5].

We consider broad scenarios whereby censors need not use deterministic fingerprints in the decision-making process, such as crypto scheme and SNI in TLS handshakes, since these fingerprints can be eliminated easily by fixing the protocol implementation. Also, a censor would not conduct active probing, which is orthogonal to passive observation and outside the scope of our study.

We define different capabilities of an 'attacker' who attempts to circumvent ML-supported censorship as shown in Figure 1. The most rudimentary setting for adversarial attacks is the *white-box model*: the trained censoring classifier is available to the attacker who leverages weight and gradient information to perturb the inputs to the ML model. Under this setting, the attacker also knows the features extracted by the censor, thus perturbations are conducted directly in the feature space instead of on raw flows/packets. A generated adversarial sample is the set of features of a flow, and converting the features back into a legitimate flow is not of this type of attacks' concern. The Carlini & Wagner (CW) attack [9, 20] uses projected/clipped gradient descent to find minimal perturbations on the inputs, such that the censoring model would misclassify. Generative Adversarial Networks (GAN)-based methods [31, 64] treat the censoring classifier as the discriminator in a GAN and train a generator to produce adversarial samples.

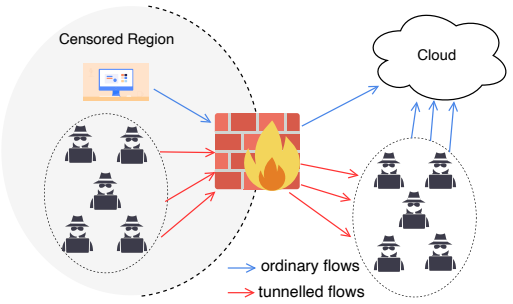

Fig. 2. Strictest adversarial model considered for subverting Internet censorship. 'Attackers' with no knowledge of the censor's tools manipulate packet sizes and inter-packet times based on implicit feedback received (flow permitted or not), to find a tunneled traffic shaping strategy that evades censorship.

However, given that the censor is unlikely to reveal the feature engineering process, the training technique employed and the architecture of ML models, we *define a stricter threat model* for adversarial attacks from a realistic perspective, to which we refer as *black-box attack*, as shown in Figure 1. Assume the attacker has access to a large number of machines with different IP addresses on both sides of the gateway, and can establish connections arbitrarily, as shown in Figure 2. The adversary may finely manipulate every network flow, by controlling packet sizes and packet inter-arrival times. We assume the manipulated network flows would be examined by the censor and the attacker can reliably infer the the censor's decisions. Under this setting:

(1) The attacker does not know which statistical features the censor may extract from each flow;
(2) The attacker does not know the architecture of the classifying ML model;
(3) The ML model may not be built with NNs, but with traditional algorithms, e.g., Support Vector Machine (SVM) or Decision Tree (DT), so gradient information is not guaranteed to exist.

This *black-box* setting gives the attacker very limited guidance on how to generate adversarial samples, while the inherent difference between the networking and other domains (e.g., computer vision) precludes the use of existing adversarial input manipulation techniques such as Square Attack [1], to circumvent censorship.

3 PROBLEM FORMULATION

Adversarial flows that seek to subvert censorship must accommodate the original payloads and be transmissible in real-world network settings. Thus, we first define a set of practical constraints that

adversarial flows must satisfy, then formulate adversarial attacks as a constrained optimization problem, which we solve with a purpose-built RL solution.

Constraints on Adversarial Attacks: We represent a network flow by a tuple $S = (P, \Phi)$, where P is a vector of n packet sizes, and Φ a vector of inter-packet delays, i.e.,

$$P = [p_1^+, p_2^-, ..., p_n^+], \quad \Phi = [\phi_1, \phi_2, ..., \phi_n].$$

Superscript '+' indicates packets transmitted from client to server, and '−' vice versa. An adversarial sample can alter each packet size by padding or truncation, and can delay packets to deceive ML classifiers. However, the attacker must ensure that bidirectional payloads in the original flows are transmitted in the correct order. Denote $\tilde{S} = (\tilde{P}, \tilde{\Phi})$ as the adversarial version of flow S, where

$$\tilde{P} = [\tilde{p}_{1,1}^+, ..., \tilde{p}_{1,k_1}^+, \tilde{p}_{2,1}^-, ..., \tilde{p}_{2,k_2}^-, ..., \tilde{p}_{n,1}^+, ..., \tilde{p}_{n,k_n}^+], \tilde{\Phi} = [\tilde{\phi}_{1,1}, ..., \tilde{\phi}_{1,k_1}, \tilde{\phi}_{2,1}, ..., \tilde{\phi}_{2,k_2}, ..., \tilde{\phi}_{n,1}, ..., \tilde{\phi}_{n,k_n}].$$

The sub-sequence $[\tilde{p}_{i,1}, ..., \tilde{p}_{i,k_i}]$ represents the adversarial manipulation of original packet sizes p_i, with $\{k_1, ..., k_n\}$ denoting the lengths of all sub-sequences. Since we allow for both packet truncation and padding, the length of an adversarial flow can be larger than that of the original, i.e., $|\tilde{P}| \geq |P|$, though the following constraint on packet sizes must be satisfied:

$$\sum_{j=1}^{k_i} \tilde{p}_{i,j} \geq p_i, \ \forall i \in [1, n], \tag{1}$$

which ensures that each original packet can be transmitted without data loss. It is straightforward to derive constraints on timestamps:

$$\tilde{\phi}_{i,1} \geq \phi_i, \tilde{\phi}_{i,j} \geq 0, \ \forall j \in [1, k_i], i \in [1, n]. \tag{2}$$

Finding Adversarial Samples: Let $e(\cdot)$ be a feature extraction function that takes an arbitrary flow S and outputs d-dimensional features. Denote $f : \mathcal{R}^d \rightarrow [0, 1]$ a binary classifier. f can be a neural network using a sigmoid as the activation function in the final layer, so its output $y = f(e(S))$ is a real number between 0 and 1. Alternatively, f can be a traditional ML algorithm (SVM, decision tree, etc.), which directly outputs discrete classification results ($\{0, 1\}$). If using a NN-based classifier, the censor would use a decision function

$$C(y) = \begin{cases} 1, & \text{if } y \geq 0.5; \\ 0, & \text{otherwise.} \end{cases}$$

That said, if the predicted score is smaller than 0.5, the flow is to be blocked. A flow $\tilde{S}$ is regarded as an adversarial version of S if $C(f(e(\tilde{S}))) = 1$. The task of finding $\tilde{S}$ can be rephrased as a constrained optimization problem:

$$\max \ C(f(e(\tilde{S}))) \text{ s.t.} \sum_{j=1}^{k_i} \tilde{p}_{i,j} \geq p_i, \quad \tilde{\phi}_{i,1} \geq \phi_i, \quad \tilde{\phi}_{i,j} \geq 0, \forall j \in [1, k_i], \quad \forall i \in [1, n].$$

Are upper bound constraints necessary? Different from the computer vision domain, we do not impose upper bound constraints on both payload and timing. The adversarial examples $\tilde{x}$ of *an image* x must satisfy an l_p-norm bound, i.e., $||\tilde{x} - x||_p \leq \epsilon$ [1], because $\tilde{x}$ should not tamper with the semantics of the original image x from a human perspective. An image of a panda should still 'look like' a panda after adversarial perturbation. However, in the networking domain, as long as the original payload is transmitted, and the sender and the recipient can interpret messages identically, the semantics remain the same. Therefore, minimizing data overhead and timing delays are not hard constraints for the problem we solve, but optional requirements that users may have in order to prevent performance degradation, for which we also offer a solution in Section 4.2.

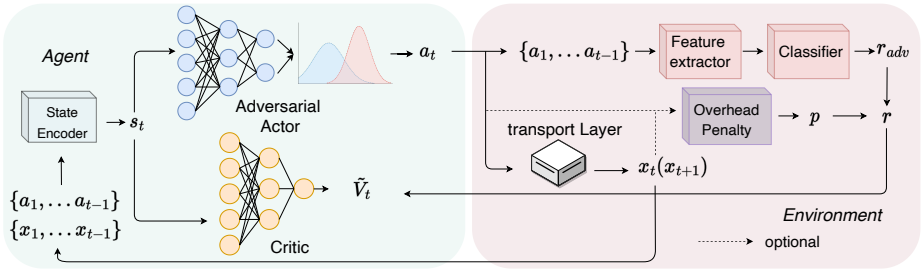

Fig. 3. The architecture of the proposed adversarial reinforcement learning algorithm – Amoeba.

4 AMOEBA DESIGN

Traditional adversarial attacks do not comply with the specifics of network flows, because (1) the length of adversarial samples are variable, according to each flow, and the optimal length is unknown; and (2) one should be able to send adversarial samples packet-by-packet, whereas existing attacks generate the feature set of an entire flow at once, without considering the practicalities of transmission.

Instead, **we regard finding adversarial versions of network flows as a sequence generation process**, which takes an input (a packet and the associated timestamp in the original flow) at each timestep and outputs adversarial manipulations of that input. The adversarial packets can be transmitted in almost real-time, rather than waiting for the entire flow to finish first. Each packet in a flow should be morphed to maximize the chances that the complete flow in the future will be misclassified, which requires an algorithm to look ahead of time and progresses through binary signals received from the censor. Given these requirements, RL is particularly well-suited to our task, where we treat the output of the censoring classifier as reward that guides a RL agent to learn a packet sequence generation policy. We design Amoeba to generate adversarial flows that circumvent censorship. Amoeba models censor decisions as a reward function, and trains an agent to interact with the censor in discrete timesteps. At each step t, the agent receives a packet from the transport layer, takes an adversarial action (effectively modifying the size and timing of the packet), and obtains a reward based on how good that action is. The agent aims to maximize the future rewards when generating adversarial samples. Note that **Amoeba does not change the implementation of any existing protocol** in terms of handshake, error handling and acknowledgment, but simply alters the 'shape' of each packet with payload to deceive ML classifiers. In other words, an adversarial TCP flow is still a legitimate TCP flow. Amoeba comprises four major elements: Network Environment, State Encoder, an Adversarial Actor and a Critic (see Figure 3). Next, we provide a RL primer, before diving into our solution.

4.1 RL Primer

Our algorithm takes a reinforcement learning approach with an agent interacting with the environment in discrete timesteps. At step t, the environment gives an *observation* $x_t = (p_t, \phi_t)$, representing an original packet to send with size p_t and inter-packet delay ϕ_t. For each flow, the actor maintains a vector of previous observations $[x_1, x_2, ..., x_{t-1}]$, as well as a vector of previous actions $[a_1, a_2, ..., a_{t-1}]$, with each *action* $a_i = (\tilde{p}_i, \tilde{\phi}_i)$ representing the manipulation of an original packet x_i. In this paper, we use *actions* and *adversarial packets* interchangeably. The *state* at step t is the history of both the observations and the actions, i.e., $s_t = (x_1, a_1, ..., x_{t-1}, a_{t-1}, x_t)$. Note that $s_t \neq x_t$, because the actor needs a broad understanding of the current environment based on what has been generated up to that point. The actor parameterized by θ maps a state to a probability distribution over the actions $\pi_\theta(s_t)$. An action is randomly sampled $a_t \sim \pi_\theta(s_t)$, and given to the environment, leading to a *reward* $r(s_t, a_t)$ and the next observation x_{t+1}. An *episode* τ indicates the

entire process of generating an adversarial sample given a flow, i.e. $\tau := (s_1, a_1, ..., s_T, a_T)$. The aim of the actor is to select actions at every timestep in a way that maximizes the total future rewards:

$$\max_{\theta} \mathbb{E}_{\tau \sim p_\theta(\tau)} [\sum_{t=1}^{T} r(s_t, a_t)].$$

The above problem can be solved by iteratively updating θ [47]:

$$\theta_{k+1} = \theta_k + \alpha \mathbb{E}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t) Q^\pi(s_t, a_t)\right],$$

where α represents step size, and $Q^\pi(s_t, a_t)$ is known as the *action-value function* that produces the discounted total future reward. Approximating Q values directly suffers from high variance in practice. Thus, a baseline is always subtracted from Q while keeping the objective unbiased [47]:

$$\theta_{k+1} = \theta_k + \alpha \mathbb{E}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t) A^\pi(s_t, a_t)\right] \tag{3}$$

in which *Advantage* $A(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$. Here, the second term is called the *state-value function* $V^\pi(s_t) = \mathbb{E}_{a_t \sim \pi}[Q(s_t, a_t)]$, which represents the expected future reward from step t, and $A(s_t, a_t)$ intuitively indicates how much better the current action a_t is than the average.

4.2 Environment

The network environment offers observations and rewards, given new actions.

Generating Observations. In practice, observations (packets) originate from the buffer in the transport layer. When there is no traffic obfuscation in place, the payload in the buffer would be encapsulated in packets and transmitted immediately. However, to generate adversarial samples, the payload cannot be sent directly but should be passed through the adversarial actor, which decides appropriate packet sizes and timings, such that the Q value can be maximized.

Therefore, as the first step we use a transport layer emulator that reads a payload with p_i bytes from the buffer as the vanilla transport layer does. To adversarially manipulate this packet (observation), $x_t = (p_i, \phi_i)$ is given to the agent, which morphs the packet based on a given policy π, truncating or adding padding to it along with some delays. Both truncation and padding are supported to expand the action space that the agent can explore, and thereby create adversarial flows with more variability. Only supporting either operation may result in the failure of generating adversarial flows. For example, an attack by only padding cannot circumvent censoring models [38, 45] that leverage directional features, since padding only changes the size of each packet but the packet directions in a flow remain the same after morphing; attacks by only truncating may hardly protect protocols with fixed payload unit size such as Tor cells, given that censoring can easily recover by summing the packet sizes in the same direction. Since we allow for truncation, it is possible that the adversarial packet (action) $\tilde{a}_t = (\tilde{p}_i, \tilde{\phi}_i)$ is smaller than the original one, leaving $p_i - \tilde{p}_i$ byte payload to send. In that case, the emulator does not read more payload from the buffer, but generates a second adversarial packet by giving the agent $x_{t+1} = (p_i - \tilde{p}_i, \phi_{i+1})$. Such operation is repeated until the remaining payload is fully sent, and then the emulator reads more payload from the buffer. For example, assume the agent truncates an original packet n times, the list of the observations sent to the agent and the list of actions would be:

$$[(p_i, \phi_i), (p_i - \tilde{p}_{i,1}, \phi_{i+1}), ..., (p_i - \sum_{j=1}^{n-1} \tilde{p}_{i,j}, \phi_{i+n-1})], \text{ and } [(\tilde{p}_{i,1}, \tilde{\phi}_{i,1}), (\tilde{p}_{i,2}, \tilde{\phi}_{i,2}), ..., (\tilde{p}_{i,n}, \tilde{\phi}_{i,n})].$$

Padding occurs if the final adversarial packet is larger than the input size, i.e., $\tilde{p}_{i,n} > p_i - \sum_{j=1}^{n-1} \tilde{p}_{i,j}$. Observe that **the emulator satisfies the constraint on packet sizes (Eq. 1) by design**, so that the adversarial actor does not have to consider it while learning the policy. Also, the observation x_t

and the associated packet size p_i do not share the same subscript because the emulator may read from buffer once, but uses multiple timesteps to send the payload, due to truncation.

Reward Function Design. The reward function evaluates how good an action a_t is under the current state s_t. Since our aim is to find adversarial network flows, the reward should first reflect the judgment of the censor, i.e., $C(f(e(\cdot)))$. There are two standard strategies to assign rewards for each action-state pair. The first is not assigning intermediate rewards while the sequence is being generated, but only assigning a final reward when the episode terminates. One typical example is AlphaGo [44], which assigns either +1 or −1 when a round of go game ends. The other strategy is to give a reward at each timestep, which was adopted for cartpole or Mario game play. The first strategy might seem suitable for our task, since all the intermediate actions should serve the final aim, that is, the adversarial flow as a whole should be misclassified. However, this would imply that the environment knows in advance when a flow will terminate, so it defers a reward until the last packet. In reality, a flow may terminate at an arbitrary timestep due to different communication purposes or network status. Note that in our adversarial model, attackers can control each packet, meaning that they can also terminate a flow at any point. In other words, we consider it possible for the censor to make a classification decision at any timestep, as if this is the last in an episode.

Formally, consider $a_t = (\tilde{p}_{i,n}, \tilde{\phi}_{i,n})$ at timestep t is generated by the attacker given $x_t = (p_i - \sum_{j=1}^{n-1} \tilde{p}_{i,j}, \phi_{i+n-1})$ and sent over the network. The censor already witnesses $\mathbf{a}_{1:t} = [a_1, a_2, ..., a_t]$. Thus, the reward regarding distinguishability is defined as:

$$r(s_t, a_t)_{adv} = C(f(e(\mathbf{a}_{1:t}))).$$

Besides, we also consider extra penalties in terms of data overhead and time delays. One may expect the adversarial sample is as close to the original flow as possible, i.e., introducing the smallest padding and delays, which would do minimal harm to the application performance. We therefore introduce a data overhead penalty and a time overhead penalty:

$$p(s_t, a_t)_{data} = \begin{cases} p_i - \sum_{j=1}^{n} \tilde{p}_{i,j} + \lambda_{split} n, & \text{if } \tilde{p}_{i,n} < p_i - \sum_{j=1}^{n-1} \tilde{p}_{i,j}; \\ \sum_{j=1}^{n} \tilde{p}_{i,j} - p_i, & \text{otherwise.} \end{cases}$$

When the size of the adversarial packet at timestep t is smaller than that of the original packet, the penalty is proportional to the number of truncations n plus the remaining bytes to send. When padding occurs ($\tilde{p}_{i,n} > p_i - \sum_{j=1}^{n-1} \tilde{p}_{i,j}$), the penalty is linear in the extra bytes to send. We do not use symmetric penalties for the two circumstances, because we find empirically that Amoeba is inclined to truncate packets into multiple instances of minimal size. Thus, we discourage this behavior by assigning an extra penalty when packet truncation occurs. The penalty for time delays is straightforward: $p(s_t, a_t)_{time} = \tilde{\phi}_{i,n} - \phi_{i+n-1}$.. The expression of the reward function thus becomes:

$$r(s_t, a_t) = r(s_t, a_t)_{adv} - \lambda_d p(s_t, a_t)_{data} - \lambda_t p(s_t, a_t)_{time},$$

where λ_{split}, λ_d and λ_t are hyperparameters that balance each component.

4.3 Adversarial Actor & Critic

As mentioned in Section 4.1, the state at timestep t is the history of the observations and the actions, meaning that the length of the state would vary as t increases. However, if the agent is built with non-recurrent neural networks, such as Multi-Layer Perceptron (MLP), it requires inputs of fixed size. To overcome this problem, we design StateEncoder, a two-layer, pre-trained Gated Recurrent Unit (GRU) that encodes an arbitrary long network flow to a fixed-size hidden representation. The pretraining and the performance of the StateEncoder are documented Appendix A.2 and A.3.

The adversarial actor aims to pick an optimal action at each timestep, such that the future rewards can be maximized. However, the action space for packet sizes is overwhelmingly large, i.e., 1,448 discrete actions for TCP and 16,384 for TLS, while the action space for time delays is infinite. Thus, we first treat both packet sizes p and time delays ϕ as continuous, and discretize them when the actor makes a choice. For example, for the TCP layer, we let the actor choose an action $(p_i, \phi_i), p_i \in [-1, 1], \phi_i \in [0, 1]$, and then discretize the packet size by $int(p_i \times 1,460)$ $byte$ and the time delay $int(\phi_i * max\_delay)$ ms, where max_delay indicates the maximum allowed delay for a packet. Note that packet sizes can be negative to represent backward traffic.

We follow an actor-critic design where the actor network $\pi_\theta(\cdot)$ approximates the best action given a state, and a critic network $V_c(\cdot)$ estimates the state value. The two networks are parameterized by θ and c respectively. Specifically, the learning objective of the actor is as described by Eq. (3). The critic network aims to approximate the state-value by minimizing the Mean Squared Error between estimated values and the discounted future rewards (R_t):

$$\min_c \mathbb{E}_t[(V_c(s_t) - E_{a\sim\pi}[Q(a_t, s_t)])^2] \approx \mathbb{E}_t[(V_c(s_t) - R_t)^2]. \tag{4}$$

In practice, we set π_θ and V_c as MLPs and find this network structure to be effective in our task. The adversarial actor has two output units: packet size $\tilde{p}$ and inter-packet delay $\tilde{\phi}$. **To satisfy the time constraint on inter-packet delays in Eq. 2**, we let π_θ output a value Δ_ϕ representing how much extra delay should be added to each packet apart from the existing delay ϕ provided by the environment, i.e., $\tilde{\phi} = \phi + \Delta_\phi$.

4.4 Optimization

Optimizing RL algorithms is challenging due to high variance among trajectories and the trade-offs between exploration and exploitation that need to be achieved. A few techniques are widely used to stabilize the training process, speed up convergence, and ensure the networks are differentiable, which we also adopt in training our agent, including (1) surrogate objective function [41]; (2) reparameterization trick, and (3) parallel rollout [41]. Interested readers can refer to Appendix A.1. The full training algorithm is detailed in Algorithm 1.

5 EXPERIMENTS

In this section, we empirically evaluate the effectiveness of applying Amoeba on two popular types of anti-censorship systems, namely Tor network and generic TLS tunneling:

(1) Tor Network is a anonymity system that utilizes relay nodes with onion protocol to conceal user location and prevent network surveillance [48]. The traffic routed inside the Tor network is encrypted by TLS and only the exit node has access to the original traffic, which is forwarded to the real destination. However, Tor is proven to be distinguishable by ML classifiers due to the fixed-size cells of the onion protocol [54].
(2) V2Ray is a generic TLS tunneling tool that tunnels arbitrary TCP/UDP packets inside TLS connections [52]. Users of this type of systems usually do not demand anonymity but only seek to bypass firewalls. Thus, these tools are widely used in countries that employ censorship, such as China. We use V2Ray as the supporting tunneling system rather than its alternatives [17, 26, 49], given that it has the largest community of both maintainers and users, and it is also widely supported by 3rd party clients across multiple platforms [51].

Both system types are vulnerable to ML classifiers due to the fact that the statistical features of the tunneled flows deviate from real TLS/HTTPS traffic.

5.1 Censoring Classifiers

Algorithm 1 The training algorithm of Amoeba

Inputs:
λ_{split} := packet truncation overhead coefficient
λ_d := data overhead coefficient; γ := discount factor
λ_t := time delay coefficient; N := number of environments
T := the length of each rollout in the environment;
Initialisation:
Initialize π_θ and V_c via Xavier initialization [18]
Obtain StateEncoder $\mathcal{E}$ from Algorithm 2
Initialize N *Env*, each of which is provided a feature extractor $e(\cdot)$ and a pretrained classifier $f(\cdot)$
Initialize a rollout buffer with size $N \times T$
while not converged **do**
  Sample $N \times T$ observations by interacting π_θ with N *Env*
  for Each observation x_t, action a_t and reward r_t **do**
    Let $\mathbf{x}_{1:t} := \{x_1, ...x_t\}, \mathbf{a}_{1:t} := \{a_1, ...a_t\}$
    Generate the state representation by $s_t = \mathcal{E}(\mathbf{x}_{1:t})||\mathcal{E}(\mathbf{a}_{1:t})$
    Compute $\hat{A}_t = \sum_{l=0}^{\infty}(\gamma\lambda)^l[r_{t+l} + \gamma V(s_{t+l+1}) - V(s_{t+l})]$
    Compute Return $R_t = \hat{A}_t + V_c(s_t)$
  end for
  Store each $(s_t, a_t, r_t, \hat{A}_t, R_t)$ in the rollout buffer and split them into K mini-batches $\{\mathcal{D}_1, .., \mathcal{D}_K\}$.
  Set $\pi_{\theta_{old}} \leftarrow \pi_\theta$
  for $k = 1, K$ **do**
    Compute $I_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$
    Update θ via $\nabla_\theta \frac{1}{|\mathcal{D}_k|}\sum\left[\min(I_t(\theta)\hat{A}_t, clip(I_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) + H_{\pi_\theta}(a_t)\right]$
    Update c via $-\nabla_c \frac{1}{|\mathcal{D}_k|}\sum (V_c(s_t) - R_t)^2$
  end for
end while
return $\mathcal{E}$

We adopt a range of state-of-the-art traffic analysis models as censoring classifiers:

Deep Fingerprinting (DF) [45] is a state-of-the-art Convolutional Neural Network (CNN)-based deep learning model that automatically extracts features from raw network flows and performs WF.

Stacked Denoising Autoencoder (SDAE) [38] follows a MLP-based encoder-decoder architecture to extract latent features from network flows directly for WF.

Long Short-Term Memory (LSTM) [38] is a multi-layer recurrent neural network that takes arbitrary long network flows as input to perform WF. LSTM is designed to learn long-term dependencies, and therefore can better interpret timeseries data such as consecutive packets.

CUMUL [34] separates different classes of data by using SVM with a radial basis function (RBF) kernel to find the hyperplane that maximizes the margin between classes.

The original versions of DF, SDAE and LSTM are fed with packet directions only (i.e., $(-1, 1)$), and vanilla CUMUL leverages the cumulative representation of network traces without timing features. For consistence, we tailor these classifiers to utilize the flow representation in Sec. 3 as input. That said, these classifiers do not need an external feature extractor.

Tree-based models [2]: Traditional ML models, such as DT and Random Forest (RF), exhibit promising performance in detecting multi-media tunneling protocols . Tree-based approaches possess better interpretability compared to DL models, since the decision-making process can be visualized as a set of tree-like rules. We follow [2] to extract 166 features from each network flow, covering bi-directional packet/timing statistics, burst behaviors, percentile features and flow-level information, and use them to train the DT/RF.

5.2 Adversarial Attack Benchmarks

We choose three advanced *white-box* adversarial attacks as benchmark algorithms for our evaluation:

CW Attack [9] uses projected gradient descent to find minimal perturbations on the inputs, while maximizing the probability of the inputs being misclassified. The CW attack iteratively queries the classifier for a single input, until an adversarial sample is found.

NIDSGAN [64] regards the censoring classifier as the discriminator in a GAN architecture, and trains a generator to learn minimal perturbation patterns needed to fool the discriminator. The generator directly produces adversarial samples given inputs, without needing iterative updates.

Blind Adversarial Perturbation (BAP) [31] also aims at training generator-like NNs, but is more flexible in allowing inserting packets into a given flow, i.e., the length of an adversarial sample is not always identical to the input, posing larger difficulties for censoring classifiers.

We do not consider black-box benchmark algorithms [1, 8, 10], since existing ones are infeasible under our threat model where feature extraction is performed (see Figure 1). We implement the NN-based classifiers, CW attack, NIDSGAN, BAP and Amoeba in Pytorch [37], and import the rest of the classifiers from the scikit-learn package in Python. Detailed hyperparameter selection is documented in Appendix A.4.

5.3 Evaluation Metrics

To evaluate the effectiveness of our solution against ML classifiers, we measure their accuracy and F1 score metrics, which are based on True Positives (TP), False Positives (FP), True Negatives (TN) and False Negatives (FN): $accuracy = (TP+TN)/(TP+TN+FP+FN)$, and $F1 = 2 \times (precision \times recall)/(precision+recall)$, where $precision = TP/(TP+FP)$ and $recall = TP/(TP+FN)$. Accuracy indicates the proportion of samples correctly classified, and F1 score computes the harmonic mean between precision and recall. The former represents how likely an algorithm would give true alarms, and the latter indicates how sensitive an algorithm is towards positive samples.

We use also use another three metrics to evaluate Amoeba, namely Attack Success Rate (ASR), i.e., the percentage of adversarial samples being misclassified, $data\ overhead = padding/(original\ payload + padding)$ and $time\ overhead = delays/(delays + total\ transmission\ \ time)$, in which total transmission time is the time difference between the last and first packet in a flow.

5.4 Data Collection & Training Procedure

We collect two real-world datasets to evaluate our approach. Specifically, we set up a Tor client on a campus machine running Ubuntu 22.04, and a Tor bridge on a Google Cloud E2 instance running Ubuntu 22.04. TCP segmentation offload is disabled on both machines. The same setup is employed for a V2Ray client and V2Ray proxy server. We consider the censor sits between the Tor (or V2Ray) client and the first relay node (or V2Ray server) and distinguishes sensitive flows. To collect a realistic Tor dataset for evaluation, we crawl the landing pages of Alexa top 25,000 websites with and without Tor network respectively (`Tor Dataset`). We use tshark to group packets into TCP flows and extract packet sizes and associated timestamps, where backward packet (server-to-client) sizes are represented with negative numbers to preserve the transmission direction. The same operation is repeated with and without the V2Ray tunnel, named `V2Ray Dataset`. Different from the `Tor Dataset`, we utilize tshark to group packets into TLS flows, and extract TLS record sizes and timestamps. For this dataset, we consider the censor conducts deep packet inspection up to the TLS layer and extracts features from TLS flows instead of TCP flows. The maximal TLS record is 16 KB, i.e., Amoeba is required to explore a much larger action space.

Each dataset is separated into a *clf_train_set* (40%), an *attack_train _set* (40%), a *validation_set* (10%) and a *test_set* (10%). We use the *clf_train_set* to train censoring classifiers, which are then evaluated on the *test_set*. After that, each trained censoring classifier is deployed in the Environment in Figure 3 to generate rewards. We use the *attack_train_set* to train Amoeba instead of using *clf_train_set*, because the attacker may have no access to the dataset owned by the censor in practice. The *validation_set* is utilized to tune the hyperparameters of Amoeba. After training, Amoeba and the benchmark algorithms are evaluated on the *test_set* against the trained censoring classifiers. To facilitate the reproducibility of our results, we make available our data collection configurations, datasets and source code at https://github.com/Mobile-Intelligence-Lab/Amoeba.

Dataset	Attack Threat Model / Censoring Alg.	None *None* F1	None *None* Accuracy	C&W *white-box* ASR (%)	C&W *white-box* DO (%)	C&W *white-box* TO (%)	NIDSGAN *white-box* ASR (%)	NIDSGAN *white-box* DO (%)	NIDSGAN *white-box* TO (%)	BAP *white-box* ASR (%)	BAP *white-box* DO (%)	BAP *white-box* TO (%)	Amoeba *black-box* ASR (%)	Amoeba *black-box* DO (%)	Amoeba *black-box* TO (%)
Tor	SDAE	0.99	0.99	88.34	21.60	0.00	30.75	20.00	4.25	84.72	22.95	21.21	89.0	64.8	8.72
	DF	0.99	0.99	97.88	26.68	23.94	94.13	31.8	7.28	89.46	35.95	12.49	87.5	59.0	7.79
	LSTM	0.99	0.99	90.49	86.64	8.37	97.88	19.09	3.54	93.86	38.88	18.65	98.2	58.1	6.26
	DT	1.00	1.00										96.5	39.0	5.69
	RF	1.00	1.00		N/A			N/A			N/A		92.0	39.1	3.73
	CUMUL	0.99	0.99										93.0	44.5	6.55
V2ray	SDAE	0.99	0.99	99.54	25.24	24.88	26.04	22.99	20.23	79.92	26.76	5.99	93.8	43.2	5.49
	DF	0.99	0.99	84.33	49.31	49.89	95.44	22.9	9.17	62.57	25.13	0.00	96.8	46.1	7.45
	LSTM	0.99	0.99	96.61	16.10	2.51	93.32	38.44	15.23	91.56	16.98	29.78	89.2	7.73	1.46
	DT	1.00	1.00										97.2	40.2	8.44
	RF	1.00	1.00		N/A			N/A			N/A		99.4	53.97	8.30
	CUMUL	0.99	0.99										96.4	51.6	10.48

Table 1. Performance of different classifiers in detecting Tor flows without attack; performance of Amoeba in crafting adversarial flows. For comparison, we also show the Attack Success Rate (ASR) of CW, NIDSGAN, and BAP attacks under different threat models (DO – data overhead; TO – time overhead). The estimated values reported represent the maximal perturbation allowed for data and timing features respectively.

5.5 Evaluation

5.5.1 ***How does Amoeba perform compared to benchmark algorithms?*** Table 1 presents the performance of each classifier detecting Tor and V2Ray traffic respectively, as well as the efficacy of adversarial attacks targeting these classifiers. In the absence of adversarial manipulations, the selected classifiers yield almost perfect accuracy and F1 scores (third column) as expected on the *test_set*, since both anti-censorship systems generate unique statistical patterns during communications. For example, when observed on the TCP layer, Tor traffic mostly consists of packets of (multiples of) 536 bytes, which is the size of an encapsulated onion cell, giving ML classifiers high confidence to detect. V2Ray's TLS-tunneled flows can be differentiated from HTTPS flows, because for HTTPS, once the TLS connection is established, the inner communications are all HTTP requests/responses; while for TLS-tunneled flows, the inner communications may involve a TLS handshake between browser and web server. This TLS-in-TLS pattern would not be witnessed in normal browsing traffic without a tunnel, which gives ML classifiers opportunities to learn the discrepancies based on the statistical features.

On the other hand, the selected white-box adversarial attacks are effective in generating adversarial features of network flows. It is not surprising that the CW attack can reach ~92%ASR on average with ~37% data and ~18% time overhead (fourth column). This attack explores misleading perturbations by leveraging the weights and gradients of the censoring classifiers, and iteratively optimizes an adversarial sample for each input (network flow). However, the practicality of CW is questionable in the networking domain, since it requires 1) a complete flow as input; and 2) multiple rounds of queries to the censoring classifiers until a legitimate adversarial flow is found.

NIDSGAN and BAP overcome the second issue by training a neural network to generate perturbations for arbitrary inputs in advance, and in the deployment stage adversarial flows can be generated in one go. NIDSGAN has however limited flexibility, since the length of adversarial flows must be equal to the length of input flows. If the censoring classifiers are able to learn directional features from sensitive flows, simply adding perturbations to each packet without inserting new packets would not change directional features, potentially leading to the failure of NIDSGAN. BAP utilizes a dedicated NN to learn at which positions in a flow if new packets should be inserted, as an approach to disturb directional features. Based on the results in Table 1, we remark that

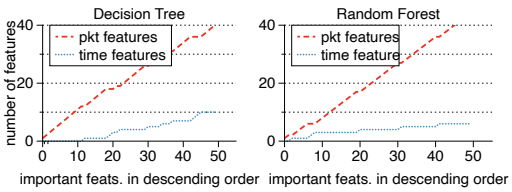

Fig. 4. Difference between packet and timing features among top-50 important ones used by DT/RF on the V2Ray dataset. x-axis arranges features by importance.

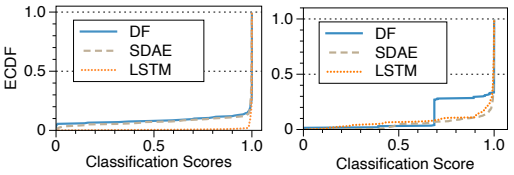

Fig. 5. Empirical Cumulative Distribution Function (ECDF) of classification scores wrt. adversarial flows against different NN-based classifiers. Left plot shows the scores obtained on Tor, right for V2Ray.

NIDSGAN and BAP have their own merits, but can also be unstable when confronting different NN architectures. Since both methods generate perturbations for an entire flow, it would be difficult to learn how the changes of a small number of packets in a flow would impact the final classification results. In contrast, Amoeba is designed to observe the classification result upon every new packet in an adversarial flow, which provides fine-grained information to infer the decision boundary of classifiers.

Table 1 reveals that our proposed **Amoeba reaches ~94% ASR on average against multiple types of classifiers**, being capable of exploring the decision boundary of a classifier even if they are not NN-based (and thus offer no gradient information which is required by existing attacks), including DT, RF and SVM/CUMUL. Compared with white-box methods, Amoeba **follows a much stricter threat model where feature engineering and model architecture are invisible, and it is also more stable against different classifiers**. The data overhead of the adversarial flows are in a similar range, between 43.2–64.8%, except for adversarial samples against DT/RF on Tor Dataset (where it is lower, yet a comparison with gradient-based methods infeasible) and those for LSTM on the V2Ray Dataset. The time overhead of adversarial flows is consistently <10.5%. Appendix A.5 offers an analysis of the actions taken to attack different classifiers.

5.5.2 ***Is Amoeba sensitive to changes in the network environment?*** A shared observation on the results with both datasets is that adversarial flows possess greater data overhead than time overhead. The reason is that censoring classifiers leverage more on packet features than on timing features to make decisions. We visualize important features used by DT and RF in Figure 4, where the x-axis lists top 50 important features in descending order, and the y-axis shows the number of packet and timing features respectively. Observe that packet features in general are overwhelmingly more important than timing features. Practically, network flows may suffer different degree of congestion depending on route and time, while packet/record sizes in a flow are purely determined by the client and the server, thus more reliable for the censoring classifiers. As a result, Amoeba makes more efforts to reshape sizes than timings.

In a more extreme setting where not only network congestion exists but packets are also dropped due to overwhelming volume of traffic in the network, packet retransmission would be needed to tackle data loss. To evaluate the impact of different packet drop rates on the performance of Amoeba, we additionally collect Tor Datasets multiple times where we enforce packet drop rates for bi-directional traffic between 0% and 10%. The same data preprocessing/split convention is followed. We train Amoeba against DF on the *attack_train_set*s collected under different packet drop rates and then evaluate it on different *test_set*s. The results are shown in Table 6, in which the numbers on the first column represent the packet drop rates under which the training sets are collected, and those on the first row indicate the packet drop rates under which the test sets are collected. The ASR [%] of Amoeba trained and tested under the same environment are shown on the diagonal of the table in bold, and the rest of the numbers indicate the performance difference in % when cross-evaluating Amoeba in different environments.

Train/Test Pkt Drop Rate	0%	2.5%	5%	7.5%	10%
0%	**87.5**	-8.2	-8.1	-6.4	-7.4
2.5%	-0	**88.8**	-0	-0.2	-0
5%	-2.0	-1.6	**94.2**	-1.2	-1.2
7.5%	+0.8	-1.8	-1.4	**94.2**	-0.4
10%	-1.2	+0.6	-1.2	-0.8	**92.0**

Fig. 6. The ASR [%] of Amoeba trained and tested under the same environment are shown on the diagonal of the table in bold, and the rest of the numbers indicate the performance difference in % when cross-evaluating Amoeba in different environments.

Amoeba trained with data experiencing 2.5%–10% packet drop rates exhibits particularly robust performance when perturbing network flows collected from other environments (2nd to 5th rows). However, if the training set does not incorporate retransmitted packets, it would be more difficult for Amoeba to perturb network flows collected with non-zero packet drop rates (1st row). This is not surprising since the dataset collected with 0% drop rate is less heterogeneous than that with non-zero drop rates. This set of results reveals that network environment is an important factor when collecting network flows, and if the dataset can reflect the heterogeneity of the network, then Amoeba is less sensitive to changes in the network environment.

5.5.3 ***What is the cost of using Amoeba and can that be reduced?*** Effectiveness against CUMUL/DT/RF aside (where the benchmarks considered don't work), Amoeba's ASR is higher than or on par with that of *white-box* attacks at the cost of a) higher data overhead, and b) 2 to 10 times more interactions with the censoring classifiers (see Fig. 7). The reason is two-fold: 1) Amoeba is a black-box algorithm and therefore requires more queries by nature; 2) Given a flow S, Amoeba is designed to interact with the classifier at least $|S|$ times and observe associated rewards, whereas BAP only needs to interact once in a training epoch.

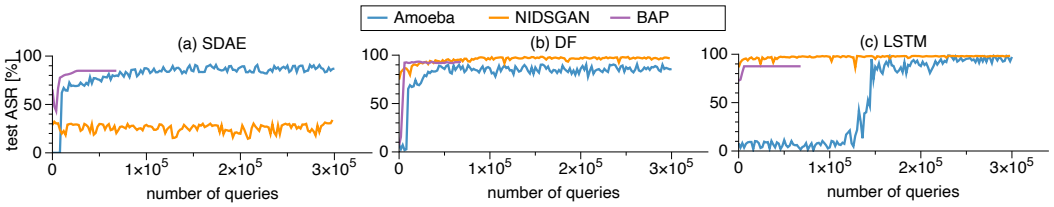

Fig. 7. Convergence curves of NIDSGAN, BAP and Amoeba attacking three classifiers on `Tor Dataset`.

However, in practice it may not be always possible to perform countless queries to censoring classifiers. We therefore attempt to reduce the number of interactions needed by randomly masking the rewards when training Amoeba. In the vanilla version of the training algorithm, Amoeba expects to receive a part of the reward r_{adv} for each subsequence of the generated flows, with 1 denoting good and 0 for sensitive. We mask r_{adv} with a probability p_{mask} from 0% to 90% during training, and the masked r_{adv} is considered to be 0.5 instead, representing unknown feedback. Amoeba is trained with 300,000 timesteps, and the actual number of queries would be $300,000 \times (1 - p_{mask})$. Each experiment is repeated 5 times and Fig. 8 plots the average ASR under each mask rate, with the shaded area representing the $\pm std$ of the results. Amoeba would experience larger variance during training when the reward is randomly masked regardless of the type of censoring classifiers applied. In particular, as the reward mask rate increases from 0% to 90%, the ASR against DF, SDAE, LSTM and CUMUL drops by 16.5% on average, whereas the ASRs against DT and RF only drops by 7% on average. This is because tree-based models utilize flow features for classification [2], and the absence of the reward for a specific adversarial packet is of lesser consequence, provided that the generated packets adhere to the learned adversarial patterns *in the feature space*. On the contrary, other models using the flow representation in Section 3 as inputs can be sensitive to the alterations to *each individual packet*. Lack of accurate rewards at each timestep would challenge Amoeba in

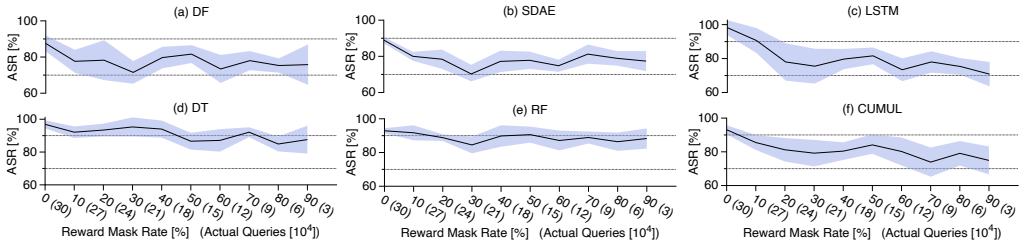

Fig. 8. Impact of reward mask rate on Amoeba's ASR. The reward mask rate is controlled to increase from 0% to 90% and the actual number of queries are in the brackets.

learning a reliable approach to generate adversarial flows (see Fig. 9). However, Amoeba is still robust even if the rewards are highly noisy, considering that the number of queries can be reduce by 10× to 30,000 and the average ASR sustained is 79%.

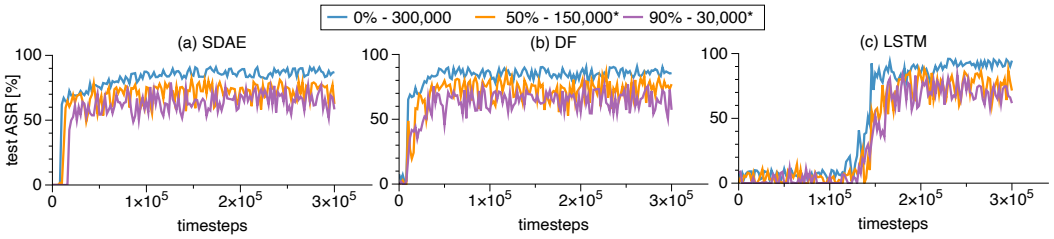

Fig. 9. Convergence curves of Amoeba attacking three classifiers under different reward mask rates, namely 0%, 50% and 90%. The legend represents the mask rate and the number of queries performed at the end of training. * denotes estimated value given that rewards are randomly dropped. Note that the x-axis represents timesteps instead of the number of queries (for orange and purple curves), because at the timesteps when the rewards are dropped, essentially no query is performed.

5.5.4 ***Are adversarial samples transferable?*** Here we investigate whether adversarial flows generated by Amoeba against one classifier can also deceive other models without retraining. To this end, we store all the adversarial samples obtained from each model and feed them to the rest of the classifiers for both datasets. We plot success rates as a heat map in Figure 10, where we find that **adversarial flows targeting similar architectures are transferable with high success rate**, such as SDAE and DF, and DT and RF, meaning that these pairs of classifiers are likely to learn a similar decision boundary.

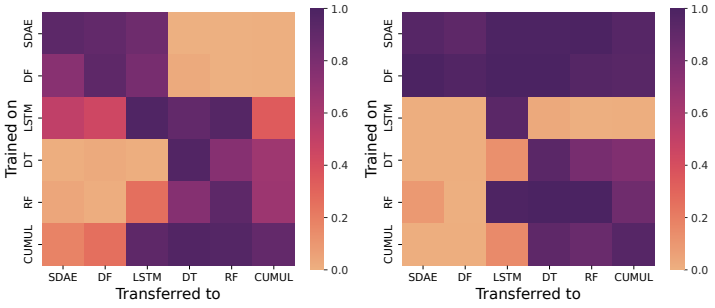

Fig. 10. Transferability of adversarial flows. Adversarial examples generated by Amoeba against each model on the y-axis and tested on other models on the x-axis. Color of each cell represents ASR. The left heatmap is obtained on the `Tor Dataset` and the right one on the `V2Ray Dataset`.

The adversarial samples targeting LSTM on the `V2Ray Dataset` are exceptional with only 7.73% data overhead on average. It is likely that Amoeba uncovers a unique and efficient strategy to attack sequential models on this dataset, but cannot be easily generalized to other censoring classifiers.

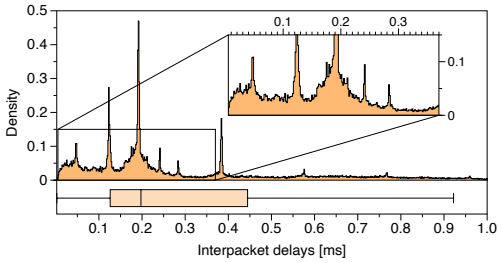

Fig. 11. The distribution (upper) and box-plot (bottom) of the inter-packet delays between every two consecutive packets in the same network direction.

Censoring Classifier	Data Overhead [%]	Time Overhead [%]
SDAE	71.22	50.02
DF	76.37	63.07
LSTM	67.99	43.44
CUMUL	63.22	50.71
DT	64.53	59.68
RF	60.58	38.02

Table 2. Average data overhead and time overhead by embedding tunneled flows into pre-stored adversarial profiles on `Tor Dataset`.

5.5.5 ***How is the quality of adversarial flows?*** When attacking NN-based classifiers, the architecture and the weights are invisible to Amoeba, but our algorithm can still explore the decision boundary effectively and find qualified adversarial flows. Fig. 5 plots the ECDF of the classification scores with respect to adversarial flows against different NN-based classifiers on both datasets, where the majority of the scores are close to 1 (benign) rather than 0.5. This means that during training, Amoeba does not choose actions randomly in the action space, but **can understand where the decision boundary lies in the black box and generates adversarial flows just as innocuous traffic**, from the perspective of ML-based classifiers.

5.6 Discussion

5.6.1 ***Feasibility of deployment.*** Having demonstrated the efficacy of Amoeba, here we discuss the feasibility of deploying our solution in practice. Integrating the algorithm in the transport layer and morphing each packet at line speed is the most ideal way of usage. We run a single-step action inference on a NVIDIA K80 GPU for 10 times and obtain an average inference time of 0.370 ± 0.001 ms, which despite small, may be considered non-negligible in the sequence generation process.

To better understand this, Fig. 11 shows the density and the box plot of the inter-packet delays between *every two consecutive packets in the same network direction* in our dataset, where 67.5% of the inter-packet delays are less than 0.37 ms. The time needed for inference would challenge the deployment of the algorithm in an online manner. However, it is still possible to utilize it once Amoeba is well-trained against a censoring classifier. Specifically, we can generate a number of adversarial flow profiles, which only consist of packet sizes and inter-packet delays without real payloads. The profiles would be saved in a database and synchronized with both client and server proxies. During communications, both parties embed actual payload into flows exactly as the flow profiles instruct. If one end has no payload in the buffer but the profile indicates a packet should be sent, then a packet with dummy payload would be transmitted to align with the pre-generated adversarial flow. Although this approach may further increase data overhead, since the flow profiles are not generated based on the current states, it ensures that ML-supported censorship can be successfully circumvented. To illustrate the overhead involved, we store all the adversarial flows (profiles) in the training set of the `Tor Dataset` that successfully circumvent each classifier, and embed tunneled flows in the test set into the pre-stored, adversarial profiles, as described above. Table 2 lists the data and time overhead against each censoring classifier respectively. Note that the increase in time overhead is much larger than that in data overhead, by comparing columns 1 and 2 in Table 2, since it is common to use multiple adversarial profiles to transmit a single tunneled flow, resulting in extra TCP handshakes to establish connections. More engineering efforts, such as matching optimal adversarial flow profiles with IP addresses, can be explored for better user experience. To fully achieve online deployment of Amoeba, technical advances, such as designing dedicated hardware that embeds NICs with computational processors [61, 62], are needed.

5.6.2 ***Interactions with Censorship Systems.*** Training Amoeba requires hardware advance to accelerate the computation of forward passes. Besides, the training algorithm engages in frequent interactions with the censoring model until a policy is discovered, and in this process, Amoeba may fail to generate adversarial flows, resulting in the blocking of IP addresses or port numbers. Therefore, we assume that attackers can manage a multitude of IP addresses on both sides of the firewall to cope with prompt responses from the censor, as captured by our adversarial model outlined in Section 2. In a practical scenario, censors tend to block the destination IP addresses or specific port numbers [22, 55], but hardly take actions on the source IP. This makes sense since the host initiating connections is often behind NAT, and blocking the source IP would prevent a large group of users from accessing the Internet. On the other hand, Amoeba would require more IP addresses outside the firewall. Tools such as MassBrowser [33] provide a peer-to-peer tunneling approach to circumvent censorship, facilitated by numerous volunteers establishing proxies in unrestricted regions, and a similar design may be utilized to train Amoeba.

Another critical issue when interacting with censorship systems is that observing rewards is not always straightforward, as the censor would not inform the classification results, but the attackers have to infer decisions instead. Although time-consuming, one viable strategy involves iteratively establishing connections and incrementally generating new packets in each connection. At the point when a connection cannot be built due to IP/port blocking, the attacker discerns that the preceding connection triggered an alarm. Otherwise, the rewards can be perceived as 1 (benign). This method is effective against a censorship system that promptly responds to unwanted traffic. For example, it was observed that HTTPS connections with ESNI would be blocked by the Great Firewall within 1 second after the censor observes a TLS Client Hello [6]. A more common scenario may be that the rewards are only observable at a certain timestep (after observing the first n packets [54], after a flow terminates [2], or for client-to-server packets only [59]) rather than upon every adversarial packet being generated. This rationale motivated the experiments conducted in Section 5.5.3, demonstrating that observing only 1/10 of the rewards can still facilitate the progress of Amoeba, albeit with a reduced ASR. The censor, on the other hand, may collect adversarial flows generated by Amoeba, to enrich the dataset of sensitive connections and train the censoring classifier repeatedly. This would nullify the old policy learned by Amoeba and re-training would become necessary. Whether iteratively training the censoring classifier and Amoeba would reach any equilibrium or one model would outperform the other alternatively is yet to be determined. This problem may align with the SeqGAN framework [63], but has not been explored in network traffic generation, which makes it a potential direction for future research.

5.6.3 ***Ethical Implications.*** Although in this work all the data is collected in a controlled environment without real users attempting to evade censorship, certain ethical implications are to be considered, in the sense that the proposed algorithm involves interactions with a censorship system, which may be illegal in restricted regions and may endanger users/attackers. However, given the black-box nature of censorship, interacting with the system is essentially the only way to understand how it works, which is also the methodology followed by prior studies [6, 7, 13, 16, 57, 59].

6 RELATED WORK

Censorship techniques. Internet censorship is carried out in a number of countries in the world, including China, Iran, Russia and India, to block unwanted communications/services.

IP filtering and DNS poisoning is the most straightforward method to prevent users from establishing connections. For example, both DNS resolution and TCP connections to Google Services fail in China, as Google is on the Great Firewall (GFW)'s blacklist [22]. Besides, DPI can inspect application-layer contents for protocol identification. Tor clients use a unique cipher suite

during TLS handshakes, which allows the GFW to narrow down the suspected targets of Tor connections [16]. Active probing involves sending carefully crafted probes to suspicious servers to determine whether they support forbidden protocols, which works against Tor, Shadowsocks, Lantern and obfuscated SSH [5, 16]. In recent years, ML algorithms (Decision Tree-based, SVM, etc.) were adopted to detect sensitive network flows. [2, 54]. ML-supported censorship may appear similar to WF [24, 34, 35, 38, 45, 46], with the difference that the former targets forbidden network protocols and the latter identifies specific websites. Existing traffic analysis models for WF can be easily adopted for network censorship as our results indicate.

Censorship circumvention approaches. SkypeMorph [30] changes the packet distribution of Tor's traffic to look like connections initiated by Skype. ScrambleSuit [58] and obfs4 [60] add random padding to each packet to eliminate the fingerprints of fixed-size onion cells in Tor. Tunneling tools embed covert messages into cover protocols, e.g. Meek [15] tunnels Tor traffic over HTTPS connections. V2ray [52] supports a range of tunnels including HTTP, TLS and Shadowsocks. DeltaShaper [4] transforms covert data into images and transfers them in Skype videocalls. Unfortunately, the aforementioned tools may be vulnerable to ML classifiers [2, 23, 54]. Protozoa [3] hijacks the WebRTC stack in Chromium and transmits hidden messages through real-time video streaming apps. Geneva [7] and SymTCP [56] design automated algorithms to discover the vulnerabilities of stateful DPI system implemented by censors. CDN browsing [65] hosts different web resources on the same set of IP addresses, and provide fake SNI in TLS handshakes to misguide censors. Decoy routing [32] leverages 'friendly' Internet autonomous systems which forward messages to the covert destinations. However, these systems are non-standard compliant.

Adversarial attacks against ML classifiers. The majority of adversarial attacks are confined to computer vision and very few apply to the networking domain. For example, Fast Gradient Sign Method (FGSM) [19] is an effective white-box attack that finds adversarial examples through the gradients of making a wrong prediction. Square attack [1] considers the victim classifier to be a black box and randomly adds perturbations to a small patch of the image, until an adversarial example is found. Another strategy of conducting black-box attacks involves a two-stage approach: 1) substituting model training; 2) adversarial sample crafting, which does not directly infer the decision boundary of ML classifiers, but leverages the transferability of samples obtained from the substitute model [21, 36, 50]. However, as evidenced by our results in Section 5.5.4, adversarial flows are not always transferable if the true architecture is distinct from the substitute model.

Recent research attempts to use ML to obfuscate traffic features. GAN showed ability to generate network flow features that are indistinguishable by ML classifiers [27, 42]. However, only manipulating at feature level is impractical, since mapping features back to a legitimate flow is challenging. iPET [43] and NIDSGAN [64] proposes GAN-based methods to generate perturbations on network traffic directly as an attempt for deceiving ML classifiers. Apart from adding perturbations to existing packets, BAP [31] learns the optimal position in a flow where to insert dummy packets, disturbing directional features.

7 CONCLUSIONS

In this paper we introduced Amoeba, an original black-box attack based on adversarial reinforcement learning for circumventing ML-based network traffic censoring classifiers. We demonstrated empirically that Amoeba can shape user flows of arbitrary length over both Tor and V2ray into sequences of packets that have on average 94% success rates in subverting a broad range of classifiers, and performs stably in different network environments. Amoeba can be trained with considerably noisy rewards and adversarial samples are transferable across similar architectures, proving its robustness and practicality compared with existing attacks. Finally, we provided guidance on how to deploy our solution on real-world systems.

REFERENCES

[1] Maksym Andriushchenko, Francesco Croce, Nicolas Flammarion, and Matthias Hein. 2020. Square Attack: A Query-Efficient Black-Box Adversarial Attack via Random Search. In European Conference on Computer Vision. Springer, 484–501.

[2] Diogo Barradas, Nuno Santos, and Luís Rodrigues. 2018. Effective Detection of Multimedia Protocol Tunneling using Machine Learning. In 27th USENIX Security Symposium (USENIX Security 18). 169–185.

[3] Diogo Barradas, Nuno Santos, Luís Rodrigues, and Vítor Nunes. 2020. Poking a Hole in the Wall: Efficient Censorship-Resistant Internet Communications by Parasitizing on WebRTC. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. 35–48.

[4] Diogo Barradas, Nuno Santos, and Luís ET Rodrigues. 2017. DeltaShaper: Enabling Unobservable Censorship-resistant TCP Tunneling over Videoconferencing Streams. Proc. Priv. Enhancing Technol. 2017, 4 (2017), 5–22.

[5] Jan Beznazwy and Amir Houmansadr. 2020. How China Detects and Blocks Shadowsocks. In Proceedings of the ACM Internet Measurement Conference. 111–124.

[6] Kevin Bock, Pranav Bharadwaj, Jasraj Singh, and Dave Levin. 2021. Your censor is my censor: Weaponizing censorship infrastructure for availability attacks. In 2021 IEEE Security and Privacy Workshops (SPW). IEEE, 398–409.

[7] Kevin Bock, George Hughey, Xiao Qiang, and Dave Levin. 2019. Geneva: Evolving Censorship Evasion Strategies. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2199–2214.

[8] Wieland Brendel, Jonas Rauber, and Matthias Bethge. 2017. Decision-Based Adversarial Attacks: Reliable Attacks against Black-box Machine Learning Models. arXiv preprint arXiv:1712.04248 (2017).

[9] Nicholas Carlini and David Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In 2017 IEEE Symposium on Security and Privacy. IEEE, 39–57.

[10] Jianbo Chen, Michael I Jordan, and Martin J Wainwright. 2020. Hopskipjumpattack: A Query-Efficient Decision-based Attack. In 2020 IEEE Symposium on Security and Privacy. IEEE, 1277–1294.

[11] Ziye Deng, Zihan Liu, Zhouguo Chen, and Yubin Guo. 2017. The Random Forest based Detection of Shadowsock's Traffic. In 2017 9th International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC), Vol. 2. IEEE, 75–78.

[12] Alec F Diallo and Paul Patras. 2021. Adaptive Clustering-based Malicious Traffic Classification at the Network Edge. In IEEE INFOCOM 2021-IEEE Conference on Computer Communications. IEEE, 1–10.

[13] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. 2015. Examining how the Great Firewall discovers hidden circumvention servers. In Proceedings of the 2015 Internet Measurement Conference. 445–458.

[14] Marwan Fayed, Lorenz Bauer, Vasileios Giotsas, Sami Kerola, Marek Majkowski, Pavel Odintsov, Jakub Sitnicki, Taejoong Chung, Dave Levin, Alan Mislove, Christopher A. Wood, and Nick Sullivan. 2021. The Ties That Un-Bind: Decoupling IP from Web Services and Sockets for Robust Addressing Agility at CDN-Scale. In Proc. ACM SIGCOMM.

[15] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. 2015. Blocking-resistant Communication through Domain Fronting. Proc. Priv. Enhancing Technol. 2015, 2 (2015), 46–64.

[16] Sergey Frolov and Eric Wustrow. 2019. The use of TLS in Censorship Circumvention.. In Network and Distributed System Security Symposium (NDSS).

[17] getlantern. 2022. getlantern/tlsmasq: A Library for Servers which Masquerade as other TLS Servers. https://github.com/getlantern/tlsmasq.

[18] Xavier Glorot and Yoshua Bengio. 2010. Understanding the Difficulty of Training DeepFeedforward Neural Networks. In Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics. JMLR Workshop and Conference Proceedings, 249–256.

[19] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and Harnessing Adversarial Examples. arXiv preprint arXiv:1412.6572 (2014).

[20] Alonso Granados, Mohammad Sujan Miah, Anthony Ortiz, and Christopher Kiekintveld. 2020. A Realistic Approach for Network Traffic Obfuscation using Adversarial Machine Learning. In Decision and Game Theory for Security: 11th International Conference (GameSec 2020). Springer, 45–57.

[21] Jie Hang, Keji Han, Hui Chen, and Yun Li. 2020. Ensemble adversarial black-box attacks against deep learning systems. Pattern Recognition 101 (2020), 107184.

[22] Nguyen Phong Hoang, Arian Akhavan Niaki, Jakub Dalek, Jeffrey Knockel, Pellaeon Lin, Bill Marczak, Masashi Crete-Nishihata, Phillipa Gill, and Michalis Polychronakis. 2021. How Great is the Great Firewall? Measuring China's DNS Censorship. In 30th USENIX Security Symposium (USENIX Security 21). 3381–3398.

[23] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. 2013. The parrot is dead: Observing unobservable network communications. In 2013 IEEE Symposium on Security and Privacy. IEEE, 65–79.

[24] Marc Juarez, Sadia Afroz, Gunes Acar, Claudia Diaz, and Rachel Greenstadt. 2014. A critical evaluation of website fingerprinting attacks. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications

Security. 263–274.
[25] Diederik P Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. arXiv preprint arXiv:1412.6980 (2014).
[26] klzgrad. 2022. klzgrad/naiveproxy: Make a Fortune Quietly. https://github.com/klzgrad/naiveproxy.
[27] Jie Li, Lu Zhou, Huaxin Li, Lu Yan, and Haojin Zhu. 2019. Dynamic Traffic Feature Camouflaging via Generative Adversarial Networks. In 2019 IEEE Conference on Communications and Network Security (CNS). IEEE, 268–276.
[28] Haoyu Liu and Paul Patras. 2022. NetSentry: A Deep Learning Approach to Detecting Incipient Large-scale Network Attacks. Computer Communications 191 (2022), 119–132.
[29] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. 2018. Kitsune: an ensemble of autoencoders for online network intrusion detection. In Network and Distributed System Security Symposium (NDSS).
[30] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. 2012. Skypemorph: Protocol Obfuscation for Tor Bridges. In Proceedings of the 2012 ACM conference on Computer and communications security. 97–108.
[31] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. 2021. Defeating DNN-Based Traffic Analysis Systems in Real-Time With Blind Adversarial Perturbations.. In USENIX Security Symposium. 2705–2722.
[32] Milad Nasr, Hadi Zolfaghari, and Amir Houmansadr. 2017. The Waterfall of Liberty: Decoy Routing Circumvention that Resists Routing Attacks. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2037–2052.
[33] Milad Nasr, Hadi Zolfaghari, Amir Houmansadr, and Amirhossein Ghafari. 2020. MassBrowser: Unblocking the Censored Web for the Masses, by the Masses.. In NDSS.
[34] Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. 2016. Website Fingerprinting at Internet Scale.. In NDSS.
[35] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. 2011. Website Fingerprinting in Onion Routing based Anonymization Networks. In Proceedings of the 10th annual ACM workshop on Privacy in the electronic society. 103–114.
[36] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. 2017. Practical black-box attacks against machine learning. In Proceedings of the 2017 ACM on Asia conference on computer and communications security. 506–519.
[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An Imperative Style, High-performance Deep Learning Library. Advances in neural information processing systems 32 (2019).
[38] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. 2018. Automated Website Fingerprinting through Deep Learning. In Proceedings of the 25nd Network and Distributed System Security Symposium (NDSS 2018). Internet Society.
[39] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust Region Policy Optimization. In International Conference on Machine Learning. PMLR, 1889–1897.
[40] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. 2015. High-dimensional Continuous Control using Generalized Advantage Estimation. arXiv preprint arXiv:1506.02438 (2015).
[41] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347 (2017).
[42] Steven Sheffey and Ferrol Aderholdt. 2019. Improving Meek with Adversarial Techniques. In 9th USENIX Workshop on Free and Open Communications on the Internet (FOCI 19).
[43] Akshaye Shenoi, Prasanna Karthik, Kanav Sabharwal, Li Jialin, and Dinil Mon Divakaran. 2023. iPET: Privacy Enhancing Traffic Perturbations for IoT Communications. In Proceedings on Privacy Enhancing Technologies (PoPETs 2023).
[44] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the Game of Go without Human Knowledge. Nature 550, 7676 (2017), 354–359.
[45] Payap Sirinam, Mohsen Imani, Marc Juarez, and Matthew Wright. 2018. Deep Fingerprinting: Undermining Website Fingerprinting Defenses with Deep Learning. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 1928–1943.
[46] Payap Sirinam, Nate Mathews, Mohammad Saidur Rahman, and Matthew Wright. 2019. Triplet fingerprinting: More practical and portable website fingerprinting with n-shot learning. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 1131–1148.
[47] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy gradient methods for reinforcement learning with function approximation. Advances in neural information processing systems 12 (1999).
[48] Tor. 2022. Tor Project | Anonymity Online . https://www.torproject.org/.
[49] trojan-gfw. 2022. An Unidentifiable Mechanism that Helps You Bypass GFW. https://github.com/trojan-gfw/trojan.

[50] Muhammad Usama, Adnan Qayyum, Junaid Qadir, and Ala Al-Fuqaha. 2019. Black-box adversarial machine learning attack on network traffic classification. In 2019 15th International Wireless Communications & Mobile Computing Conference (IWCMC). IEEE, 84–89.
[51] v2fly. 2022. Awesome Tools | V2Fly.org. https://www.v2fly.org/en_US/awesome/tools.html#third-party-gui-clients.
[52] v2fly. 2022. v2fly/v2ray-core: A Platform for Building Proxies to Bypass Network Restrictions. https://github.com/v2fly/v2ray-core.
[53] Gunjan Verma, Ertugrul Ciftcioglu, Ryan Sheatsley, Kevin Chan, and Lisa Scott. 2018. Network Traffic Obfuscation: An Adversarial Machine Learning Approach. In MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM). IEEE, 1–6.
[54] Liang Wang, Kevin P Dyer, Aditya Akella, Thomas Ristenpart, and Thomas Shrimpton. 2015. Seeing through Network-protocol Obfuscation. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. 57–69.
[55] Zhongjie Wang, Yue Cao, Zhiyun Qian, Chengyu Song, and Srikanth V Krishnamurthy. 2017. Your state is not mine: A closer look at evading stateful internet censorship. In Proceedings of the 2017 Internet Measurement Conference. 114–127.
[56] Zhongjie Wang and Shitong Zhu. 2020. SymTCP: Eluding Stateful Deep Packet Inspection with Automated Discrepancy Discovery. In Network and Distributed System Security Symposium (NDSS).
[57] Philipp Winter and Stefan Lindskog. 2012. How the Great Firewall of China is Blocking Tor. In 2nd USENIX Workshop on Free and Open Communications on the Internet (FOCI 12).
[58] Philipp Winter, Tobias Pulls, and Juergen Fuss. 2013. ScrambleSuit: A Polymorphic Network Protocol to Circumvent Censorship. In Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society. 213–224.
[59] Mingshi Wu, Jackson Sippe, Danesh Sivakumar, Jack Burg, Peter Anderson, Xiaokang Wang, Kevin Bock, Amir Houmansadr, Dave Levin, and Eric Wustrow. 2023. How the Great Firewall of China detects and blocks fully encrypted traffic. In 32nd USENIX Security Symposium (USENIX Security 23). 2653–2670.
[60] Yawning. 2022. Yawning/obfs4: The Obfourscator. https://github.com/Yawning/obfs4.
[61] YKEVIN DEIERLING. 2020. What Is a DPU? https://blogs.nvidia.com/blog/2020/05/20/whats-a-dpu-data-processing-unit/.
[62] YKEVIN DEIERLING. 2020. What Is a SmartNIC? https://blogs.nvidia.com/blog/2021/10/29/what-is-a-smartnic/.
[63] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. 2017. SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient. In Proceedings of the AAAI conference on artificial intelligence, Vol. 31.
[64] Bolor-Erdene Zolbayar, Ryan Sheatsley, Patrick McDaniel, Michael J Weisman, Sencun Zhu, Shitong Zhu, and Srikanth Krishnamurthy. 2022. Generating Practical Adversarial Network Traffic Flows using NIDSGAN. arXiv preprint arXiv:2203.06694 (2022).
[65] Hadi Zolfaghari and Amir Houmansadr. 2016. Practical Censorship Evasion Leveraging Content Delivery Networks. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. 1715–1726.

A APPENDIX

A.1 Actor & Critic Optimization

We adopt a few optimization tricks in training our agent, including (1) surrogate objective function [41]; (2) reparameterization trick, and (3) parallel rollout[41]:

(1) Surrogate Objective: Directly optimizing Eq. 3 using a sampled trajectory through multiple steps of gradient ascent may lead to overwhelmingly large, and sometimes worse policy updates. Trust Region Policy Optimization (TRPO) [39] and Proximal Policy Optimization (PPO) [41] propose to use a surrogate objective function which theoretically guarantees policy improvement over stochastic gradient ascent:

$$\max_{\theta} \mathbb{E}_t \left[\frac{\pi_\theta \left(a_t \mid s_t \right)}{\pi_{\theta_{\text{old}}} \left(a_t \mid s_t \right)} A(s_t, a_t) \right],$$

in which θ_{old} represents the parameters of an older version of the actor network in stochastic optimization. The surrogate objective function intuitively encourages the actions with positive advantages $A(a_t, s_t) > 0$ and discourages the opposite. We follow the PPO design to clip the ratio $I_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ (avoiding excessive update steps), and add an entropy term to encourage exploration in the action space in the final version of the objective function:

$$\begin{aligned} \max_{\theta} \mathbb{E}_t [\min I_t(\theta) A(s_t, a_a), clip(I_t(\theta), 1-\epsilon, 1+\epsilon) A(s_t, a_a))] \\ + H_{a_t \sim \pi_\theta}(a_t) \end{aligned} \tag{5}$$

(2) Reparametrization trick: $\pi_\theta(\cdot)$ should approximate the distribution of actions given states but a simple MLP network only generates deterministic outputs. To overcome this issue, we assume that all the actions are sampled from a Gaussian distribution, and make π_θ generate the mean and the standard deviation of actions given states $\bar{a}_t, \sigma = \pi_\theta(s_t)$, as shown in Figure 3. An action then can be sampled by:

$$a_t = \bar{a}_t + \epsilon\sigma, \epsilon \sim \mathcal{N}(0, 1).$$

The trick ensures the actor network is differentiable, as well as generating probabilistic outputs during training.

(3) Parallel Rollout: In order to speed up model convergence, PPO [41] proposes to train the agent with parallel environments (N in total) where trajectories would be sampled from each environment independently with a fixed timestep T, resulting in $N \times T$ observations each time (Algorithm 1 line 4). The advantage at every timestep is estimated via generalized advantage estimation [40]:

$$A_t \approx \sum_{l=0}^{\infty} (\gamma\lambda)^l [r_{t+l} + \gamma V(s_{t+l+1}) - V(s_{t+l})],$$

in which γ is the discount factor and λ balances the bias and the variance of advantage estimation. We set $\gamma = 0.99$ and $\lambda = 0.95$. If one trajectory terminates before step T, the environment starts to generate a new one until reaching T steps and if the trajectory does not terminate after T, the advantage can still be estimated by $A_T \approx r_T + \gamma V(T+1) - V(T)$. $N \times T$ observations along with the actions, the returns and the estimated advantages are then even split into K mini-batches for stochastic optimization (Alg. 1 line 11-13). The full training algorithm is detailed in Algorithm 1.

A.2 StateEncoder

As mentioned in Section 4.1, the state at timestep t is the history of the observations and the actions, meaning that the length of the state would vary as t increases. However, if the agent is built with non-recurrent neural networks, such as MLP or CNN, it requires inputs of fixed size. To overcome this problem, we design StateEncoder, a two-layer, pre-trained GRU that encodes an

arbitrary long network flow to a fixed-size hidden representation. As shown in Figure 12, to ensure that StateEncoder $\mathcal{E}$ can properly encode network flows without nontrivial information loss, we train $\mathcal{E}$ as the encoder part of a Seq2Seq Autoencoder, in which StateDecoder $\mathcal{D}$ shares the same architecture with $\mathcal{E}$. Consider a network flow $S = [s_1, ..., s_T]$ with T packets. $\mathcal{E}$ aims to map S as an representation in the H-dimensional hyperspace, $r_S = \mathcal{E}(S) \in \mathbb{R}^H$, and $\mathcal{D}$ aims to reconstruct the flow from the hidden representation, $\hat{S} = \mathcal{D}(r_S) \in \mathbb{R}^{T\times 2}$. We train the Seq2Seq Autoencoder with a Mean Squared Error (MSE) loss function, i.e.,

$$L(S, \hat{S}) = \frac{1}{T}\sum_{t=1}^{T}(s_t - \hat{s}_t)^2,$$

by the Adam algorithm [25]. The only connection between $\mathcal{E}$ and $\mathcal{D}$ is the hidden representations. Therefore, $\mathcal{E}$ has to encode the input as intact as possible, to ensure that $\mathcal{D}$ can properly reconstruct. Since the StateEncoder is designed to encode heterogeneous network flows effectively, it should be fed with as many distinct flow as possible during training, with a view to acquiring strong generalization abilities. To this end, we create a synthetic, normalized dataset with maximal variability in both packet sizes and time delays, where each packet size p_i and inter-packet delay ϕ_i in the flows are created via:

$$p_i \sim \mathcal{U}(-1, 1); \quad \phi_i \sim \mathcal{U}(0, 1), i \in [1, T],$$

with $\phi_1 = 0$. We assume that all the packet sizes and delays are 0-1 normalized in this dataset. p_i is sampled from $\mathcal{U}(-1, 1)$ because the flow is bidirectional. We create a training set with 12,000 flows and a test set with 3,000 flows. Since a reward is given at each timestep in an episode, the StateEncoder must be able to encode a sequence with a arbitrary length. Thus, the sequence length of each mini-batch during training is randomly sampled from $[1, T]$ to avoid that StateEncoder can only encode fixed-size flows. The complete training algorithm is detailed in Algorithm 2. After training, we only preserve $\mathcal{E}$ to encode states for the adversarial actor and critic.

A.3 Performance of StateEncoder

Algorithm 2 The training algorithm for StateEncoder

Inputs:
$dataset := \{S_1, ..., S_n\}$; $S_i := [s_{i,1}, ..., s_{i,T}]$
Initialisation:
Denote $\mathcal{E}(\cdot)$ a two-layer GRU StateEncoder and
$\mathcal{D}(\cdot)$ as the decoder with the same architecture.
$\mathcal{E}$ and $\mathcal{D}$ initialized via Xavier initialization [18].
while model has not converged **do**
  for S_i sampled from $dataset$ **do**
    $S_i \leftarrow S_i[:t], t \sim \mathcal{U}(1, T)$
    $\hat{S}_i \leftarrow \mathcal{D}(\mathcal{E}(S_i))$
    $\mathcal{L} \leftarrow MAE(S_i, \hat{S}^i)$
    $\mathcal{E}, \mathcal{D} \leftarrow Adam(\mathcal{L}, \mathcal{E}, \mathcal{D})$
  end for
end while
return $\mathcal{E}$

The performance of our StateEncoder impacts the adversarial actor in terms of understanding and interpreting the actual state at each timestep. There is no simple method to evaluate StateEncoder alone, since the hidden representations are in high-dimensional space and information loss during encoding is intractable. Nevertheless, we can obtain an upper bound of the information loss by examining the Noromalized Mean Absolute Errors (NMAE) of the Seq2Seq model (consisting of StateEncoder and StateDecoder):

$$\text{NMAE}(S, \hat{S}) = \frac{1}{T \times N}\sum_{n=1}^{N}\sum_{t=1}^{T}\frac{|s_t^n - \hat{s}_t^n|}{s_t^n}.$$

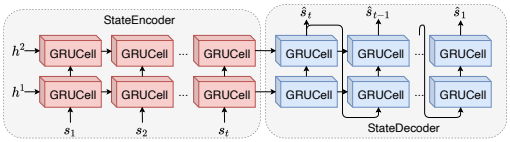

Fig. 12. StateEncoder and associated decoder for sequence-to-sequence training. During training, StateEncoder maps an arbitrary long network flow to a fixed-size hidden representation, which is passed to StateDecoder for reconstruction.

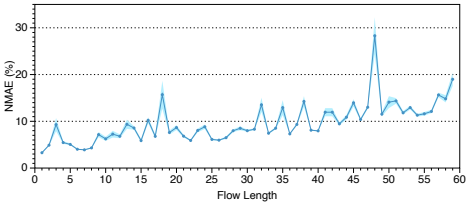

Fig. 13. Normalized reconstruction errors (with error bars) of StateEncoder + StateDecoder.

s_t^n is the packet t in flow n and $\hat{s}_t^n$ the reconstructed packet. We show the NMAE of the Seq2Seq model composed by StateEncoder and StateDecoder in Figure 13, which helps us understand to what extent the encoded hidden representations can sustain the information of the original flows. It can be seen that the NMAE of flow reconstruction would increase as the flow length increases, although this is not obvious when the flows have less than 40 packets and the average NMAE in $[1, 40]$ is around 9%. When the flow length is longer than 40, the NMAE gradually increases from 9% to 19% with an outlier of 28.95% when the flow length equals 48. An intuitive explanation of the NMAE in our case is that, for example, when a flow has 60 packets, each packet size p is encoded as a value between $p \pm 0.19p$ in the hidden representation. Although not perfect, these experiments demonstrate that this level of precision is actually adequate for Amoeba to learn an effective policy. Note that 90.5% of Tor flows in the dataset have less than 60 packets. To ensure that long flows can be encoded properly in practice, an engineering solution is splitting long flows before a pre-set threshold or using deeper networks to encode flows.

Hyperparameter	Search Space	Value
optimizer	Adam, SGD, RMSProp	Adam
learning rate	[0.0001, 0.01]	0.0005
λ_{split}	[0.01, 0.1]	0.05
λ_{time}	[0.1, 2]	0.2
λ_{data} for `Tor`	[0.1, 5]	0.2
λ_{data} for `V2ray`	[0.1, 5]	2
Actor/Critic layer number	[2, 5]	4
Actor/Critic layer dim	[32, 1024]	256⟶ 64⟶ 32 ⟶output
StateEncoder architecture	[GRU, LSTM]	GRU
StateEncoder dim	[256, 1024]	512
StateEncoder layer	[1, 4]	2

Table 3. Hyperparameter selection for Amoeba.

A.4 Hyperparameter Selection

Amoeba is a complex model with a range of hyperparameters and it would be difficult to conduct exhaustive search in the full hyperparameter space. To select hyperparameters for Amoeba, We first choose the search space by our experience and build the model in a block-by-block fashion. StateEncoder requires pretraining and therefore the associated hyperparameters are decided initially, followed by the architecture of actor and critic. λ_{data}, λ_{time} and λ_{split} plays an important role in the

reward function and largely determines the final ASR and overhead rates. We notice that Amoeba is not sensitive to λ_{time} but the results may vary greatly given different λ_{data}. Since `Tor Dataset` and `V2ray Dataset` have different largest transmission units (TCP segement and TLS record), the optimal λ_{data} are 0.2 and 2 respectively. λ_{split} determines how frequently the packet should be truncated. Our experimental results indicate that when $\lambda_{split} > 0.1$, Amoeba would tend not to truncate packets at all. If directional features need to be disturbed, λ_{split} should be set smaller than 0.1. We set $\lambda_{split} = 0.05$ eventually so that packet truncation would occur but is not so frequently that exceeds the capability of StateEncoder. We choose the set of hyperparameters in Table 3 which is good enough to provide high ASR and acceptable overhead rates, but there may exist better selections.

A.5 Action Analysis

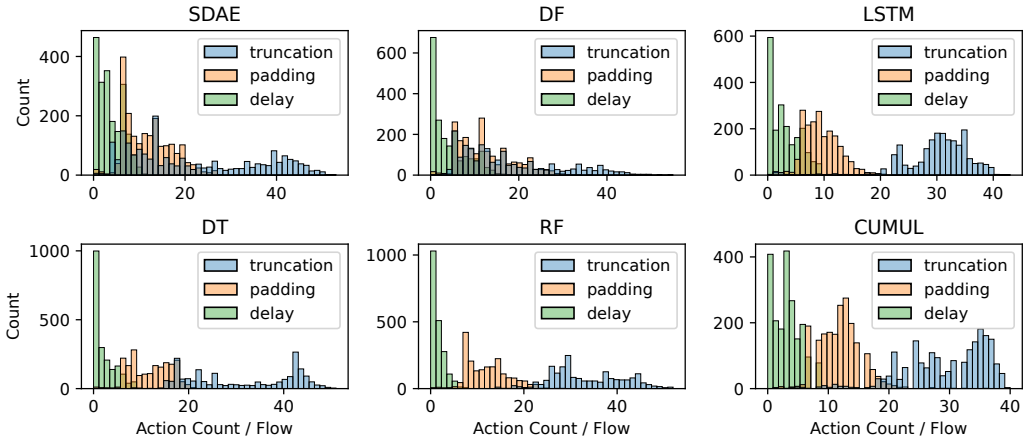

Fig. 14. Histograms of the actions taken per flow (2000 flows in total) to generate adversarial samples against each classifier on `Tor Dataset`.

The time overhead of adversarial flows generated by Amoeba is consistently and significantly lower than the data overhead, as shown in Table 1. Here, we scrutinize the actions selected by Amoeba more closely, namely, truncation, padding, and adding delay. Fig. 14 presents histograms of the number of actions taken per flow (2000 flows in total) to craft adversarial samples against each classifier on the `Tor Dataset`. The average length of the tunneled flows prior to obfuscation is 24.5 packets. It is obvious that when generating adversarial flows, adding delay is the least favored action, irrespective of the backend censoring classifiers, yielding less than 8 instances of added delays for the majority of the adversarial flows. In comparison, truncation is commonly employed, especially when attacking LSTM, DT, RF and CUMUL. Its usage is roughly twice as often as the number of padding instances, which effectively alters the directional features in the original, sensitive traffic.