Real-Time Evasion Attacks against Deep Learning-Based Anomaly Detection from Distributed System Logs

J. Dinal Herath, Ping Yang, Guanhua Yan
Department of Computer Science, State University of New York at Binghamton
Binghamton, NY, USA
{jherath1,pyang,ghyan}@binghamton.edu

ABSTRACT

Distributed system logs, which record states and events that occurred during the execution of a distributed system, provide valuable information for troubleshooting and diagnosis of its operational issues. Due to the complexity of such systems, there have been some recent research efforts on automating anomaly detection from distributed system logs using deep learning models. As these anomaly detection models can also be used to detect malicious activities inside distributed systems, it is important to understand their robustness against evasive manipulations in adversarial environments. Although there are various attacks against deep learning models in domains such as natural language processing and image classification, they cannot be applied directly to evade anomaly detection from distributed system logs. In this work, we explore the adversarial robustness of deep learning-based anomaly detection models on distributed system logs. We propose a real-time attack method called LAM (Log Anomaly Mask) to perturb streaming logs with minimal modifications in an online fashion so that the attacks can evade anomaly detection by even the state-of-the-art deep learning models. To overcome the search space complexity challenge, LAM models the perturber as a reinforcement learning agent that operates in a partially observable environment to predict the best perturbation action. We have evaluated the effectiveness of LAM on two log-based anomaly detection systems for distributed systems: DeepLog and an AutoEncoder-based anomaly detection system. Our experimental results show that LAM significantly reduces the true positive rate of these two models while achieving attack imperceptibility and real-time responsiveness.

ACM Reference Format:
J. Dinal Herath, Ping Yang, Guanhua Yan. 2021. Real-Time Evasion Attacks against Deep Learning-Based Anomaly Detection from Distributed System Logs. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy (CODASPY '21), April 26–28, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3422337.3447833

1 INTRODUCTION

Distributed system logs record states and events that occurred during the executions of large distributed systems, such as big data systems, online services, and scientific workflows. Usually printed in predefined formats, these logs help system administrators examine internal system states, identify anomalous behaviors (e.g., due to malicious activities), and troubleshoot root causes. A large body of research efforts have been dedicated to automating anomaly detection from distributed system logs, using machine learning [21] and deep learning particularly [9]. However, although deep learning models have shown superior performance in various application domains, there have been many successful attempts at misleading their predictions by injecting imperceptible modifications to inputs [1, 3, 40, 43]. This naturally raises the concern: *can an attacker perturb distributed system logs with minimal modifications to evade anomaly detection based on deep learning models?* If such perturbations are done when an attacker is performing malicious activities, then he/she does not have to worry about being caught by the anomaly detection system.

Evading anomaly detection from distributed system logs comes with new challenges. Although there are various attacks against deep learning models in domains such as natural language processing (NLP) and image classification [2, 40, 45], none of these techniques can be applied directly to evade system log anomaly detection due to the following reasons. *First*, a key challenge in the generation of adversarial examples is to ensure their imperceptibility, which differs in different application domains. For example, in image classification, an adversarial example should have as few pixels modified as possible. Due to the use of predefined templates, distributed system logs have different syntactical structures, which constrains the attacker's action space in constructing imperceptible adversarial examples. *Second*, as state-of-the-art log-based anomaly detection models for distributed systems such as DeepLog [13] and AutoEncoders [19] are capable of catching misbehavior in an online fashion, successful evasion of these models must respond in real time to the incoming log entries. The streaming nature of the problem dictates that an attacker cannot look ahead for future log entries in the stream. The attacker also cannot perturb a past log entry that has already been processed by the anomaly detection model. Such real-time constraints do not exist in image classification, for which a large body of adversarial machine learning techniques have been developed [1]. *Last but not least*, the temporal correlations inherent in distributed system logs, which are commonly exploited by existing anomaly detection techniques to detect suspicious patterns, significantly complicate real-time evasion attacks because any modification action taken now may alter the anomaly detection model's prediction results for the future log entries, which are not available for attack decision-making at the present moment.

Against this backdrop, in this work we explore the adversarial robustness of deep learning-based anomaly detection from distributed

CODASPY '21, April 26–28, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8143-7/21/04…$15.00
https://doi.org/10.1145/3422337.3447833

system logs, which to the best of our knowledge, has not been investigated previously. We propose a real-time attack method called LAM (Log Anomaly Mask) to perturb streaming logs with minimal modifications in an online fashion so that they can evade anomaly detection by even the state-of-the-art deep learning models. LAM includes two key components, a surrogate model and a perturber. The surrogate model is used to approximate the behavior of the operational deep learning-based anomaly detection model in blackbox or graybox attacks, or simply a duplicate of the operational model in whitebox attacks. The perturber is trained offline to learn the best policy in perturbing streaming logs with minimal changes to evade the detection by the surrogate model. The attacker uses this policy to make immediate decisions when performing a real-time evasion attack against the operational anomaly detection model.

The real-time constraint of LAM requires us to overcome two complexity-related challenges. First, even for a small number of log keywords (shortened as logkeys), the combination of all possible modifications in the attacker's action space can grow exponentially, making it hard to find in real time the optimal one that can evade the operational anomaly detection model while ensuring that the changes should be minimal (*action space complexity*). Secondly, even if an algorithm can avoid exhaustive search of the entire action space, the number of states and actions it uses to find an ideal policy for evasion attacks should be manageable with limited computational resources (*state space complexity*). To overcome the action space complexity challenge, the perturber in LAM is modeled as a Reinforcement Learning (RL) agent that operates in a partially observable environment. Given only the current and some past logkeys in the data stream, the perturber learns an optimal policy to determine which perturbation action should be taken at each time step. Moreover, the perturber addresses the state space complexity challenge by training an LSTM (Long Short-Term Memory) deep learning model to predict the best perturbation action from its observations made in the current environment. In a real attack, the LSTM model trained offline is used to assist the attacker with choosing the best perturbation action for each new logkey encountered. In a nutshell, our contributions can be summarized as follows:

- We propose a real-time attack method called LAM, to perturb streaming logs with minimal modifications in an online fashion. LAM considers three types of attacks – whitebox, graybox and blackbox - depending on the attackers' pre-existing knowledge and access to anomaly detection models.
- We overcome the search space complexity challenge by modeling the perturber in LAM as a reinforcement learning agent that operates in a partially observable environment to predict the best perturbation action.
- We have evaluated the effectiveness of LAM on two state-of-the-art log-based anomaly detection models shown to have superior anomaly detection capability for distributed systems: DeepLog [13] and an AutoEncoder-based anomaly detection system [19]. Our experimental results show that LAM significantly reduces the true positive rate of these two models while achieving attack imperceptibility and real-time responsiveness.
- We have provided thorough discussions on the potential defensive methods against our proposed attack on log-based anomaly detection for distributed systems.

Organization: The rest of the paper is organized as follows. Section 2 provides a brief overview of deep learning based anomaly detection models from distributed system logs. Section 3 formulates the problem of our real-time evasion attack and presents the threat model. Section 4 describes the architecture of LAM. The algorithm details are given in Section 5. Section 6 presents our experimental results. Section 7 discusses potential defensive methods against LAM. Section 8 presents the related work and Section 9 draws the concluding remarks.

2 BACKGROUND

This section provides a brief overview of deep learning-based models used for distributed system log anomaly detection that are targeted in our attack, namely DeepLog [13] and AutoEncoder [19]. Our attack focuses on circumventing anomaly detection at the session level. A session is a collection of system logs grouped together by some predefined criteria (e.g., logs generated from the same virtual machine). Many log based anomaly detection systems that utilize deep learning models as the anomaly detection component (e.g., DeepLog [13], AutoEncoders [4, 19], LogGAN [38], and Desh [11]) operate in two steps – a parsing step and an anomaly detection step.

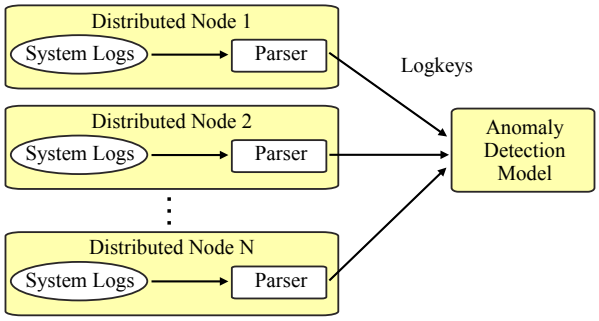


Figure 1: Anomaly Detection on Distributed System Logs with *N* Distributed Nodes.

Logkey	System Logs
1	Adding an already existing block (.*)
2	(.*)Verification succeeded for (.*)
3	(.*) Served block (.*) to (.*)
...	...
29	PendingReplicationMonitor timed out block (.*)

Table 1: Example logkeys for HDFS system logs

Figure 1 shows the flow of the data stream in anomaly detection systems based on distributed system logs. In general, the deep learning based anomaly detection model is deployed in a central location, where the logs are collected and streamed to the anomaly detection system from one or more locations over the network [32].

Examples of such distributed logs are Hadoop File System logs (HDFS) [42] and logs generated by scientific workflow systems [22].

An anomaly detection system as illustrated in Figure 1 involves multiple steps. First, the raw system logs are collected and parsed into numerical values called *logkeys*. This step uses a predefined template that directly maps a given log entry into a numerical value [41]. This template is generated either through automated log parsing tools (e.g., Spell [12], Drain [20]) or is defined by domain experts. An example of the logkeys parsed from the HDFS system log dataset [42] is given in Table 1, where the logkey template has 29 logkey types. In Table 1, the symbols $(.*)$ represent the parameter value positions that are discarded when mapping log entries into logkeys.

In the second step, the logkeys are grouped into sessions and are fed into the anomaly detection model. Formally, we denote a session as a univariate time series $X = \{x_1, x_2, x_3, ..., x_T\}$, where T is the length of the session and x_t $(1 \leq t \leq T)$ is a logkey. An anomaly detection model processes each session using a fixed length sliding window with a step size of 1. At each time instance t, the input to the anomaly detection model is a fixed length sequence $seq_t = \{x_{t-m+1}, x_{t-m+2}, ..., x_t\}$ where m is the sliding window size.

Anomaly detection models are usually trained on benign samples. At the test time, the models detect the presence of anomalies based on the deviation between the input and what it has learnt. The deviation is captured either through the error or the loss of the model (e.g., AutoEncoders [4, 19]), or based on the inability of the model to predict future variations in the time series (e.g., DeepLog [13]). Given a deep learning model F, the entire session is marked as anomalous if an anomaly occurs at any point in the session (i.e., there exists $1 \leq t \leq T$ such that $F(seq_t) = True$).

For the deep learning model F, we consider the following two state-of-the-art models in this work.

Deeplog: Deeplog uses an LSTM model to detect anomalies in system logs. LSTM is a specially designed deep learning architecture that excels in learning temporal variations in data. At each time instance t, the model takes as input a fixed length of sequence seq_t and learns the conditional probability of the next logkey x_{t+1}. DeepLog is trained on a benign set of samples. During the anomaly detection, at each time instance, the model outputs g (a user defined parameter) logkeys that are most likely to arrive next. If the logkey that arrives in reality is not among the g logkeys, then an anomaly flag is raised.

AutoEncoder: AutoEncoder is a Deep Learning model that excels at learning hidden representations of data. AutoEncoder has two main components – an encoder and a decoder – which are generally Feed Forwarding Deep Neural Networks. The encoder learns a hidden representation of an input, which is then fed back into the decoder to reconstruct the input from the hidden representation. To perform system log anomaly detection, AutoEncoder obtains an input sequence of logkeys seq_t at each time step, and tries to reconstruct the sequence. The normalized error associated with this reconstruction is used as an anomaly score. If the error is greater than a fixed threshold, then an anomaly flag is raised. At the training time, AutoEncoder learns to minimize its reconstruction error for a set of benign samples with no anomalies.

3 PROBLEM FORMULATION AND THREAT MODEL

In this work, we consider real-time evasion attacks against deep learning-based anomaly detection systems developed to identify suspicious activities from distributed system logs. Anomaly detection systems are often designed to operate in an online manner [13, 22, 29, 44]. It is important to detect anomalous behaviors in computer systems in a timely and online manner so that system administrators can detect an ongoing attack or address a system performance issue as soon as possible [13, 26]. As the size of the parsed logkeys (numeric values) is much smaller than that of the raw logs, to ensure that the logs are processed in real-time, it is more efficient to send the logkeys than raw logs to the anomaly detection model over the network.

Following adversarial evasion attacks in other domains [3, 40, 43], we generate attacks that aim to perturb the input to anomaly detection models such that anomaly samples would be mistakenly identified as being benign. Similar to many existing evasion attacks, we aim to fool the operational anomaly detection models (i.e., DeepLog and AutoEncoder) without changing any internal parameters such as trained weights. In such an attack, the objective is to identify the modifications to the streaming logkeys such that it can be carried out before the input arrives at the anomaly detection model deployed. We consider imperceptibility to be a negligible percentage of the logkeys in an entire session that need to be modified by a given adversarial attack. Our intuition is that the fewer changes, the more invisible the attack appears to the defender. Formally speaking, our work is aimed at addressing the following question: *given a target anomaly detection model F and an anomalous logkey session X, is it possible to perturb X in real time with minimal modifications so that no anomaly is raised by model F throughout the session?*

Assumptions and threat model: In anomaly detection, an anomaly raised at any location of a session makes the entire session anomalous. When an attacker modifies e.g., the execution of a Virtual Machine (VM), multiple places in the log session may change. An anomaly detection model that can identify any of those changes can successfully detect the malicious activity. Therefore, to successfully fool the anomaly detection model, multiple modifications may be needed in the log stream. We assume that an attacker has the ability to intercept or modify a logkey before the logkey is processed by an anomaly detection model. We also assume that an attacker can use a surrogate anomaly detection model to find the places where anomalies are likely to be raised. We consider the following three types of attacks:

- *Whitebox attack:* In a whitebox attack, the adversary has all the information about the target model, i.e., the surrogate model is an exact replica of the target model. This implicates that any anomaly flag raised by the surrogate model should match exactly the one raised by the target model.
- *Graybox attack:* In a graybox attack, the adversary knows the hyper-parameters and the architecture of the target model, but not its internal parameter values. In deep learning based anomaly detection, models with the same architecture can still have different weights because they are initialized with

different random numbers at the beginning of model training.

- *Blackbox attack:* In a blackbox attack, the adversary has no information about the target model. The surrogate model used by the adversary may be totally different from the target model. Due to the discrepancy in model architectures, the differences in anomalies identified between the surrogate model and the target model should be more significant than the other two types of attacks.

4 DESIGN OF LAM

In this section we present the high-level design goals of LAM. Figure 2 gives the architecture of LAM, which manipulates the parsed logkey stream before it arrives at the anomaly detection model. LAM observes the most recently parsed logkeys in the stream (i.e., the observation window), identifies possible anomaly situations, and determines the logkey to manipulate. Once the logkey is identified, LAM intercepts and perturbs the logkey.

LAM consists of two components: a *surrogate model (SM)* and a *perturber (P)*. The surrogate model plays two roles: to identify attack entry points and to act as a reference model for the perturber. The perturber learns to make adversarial modifications in the logkey stream using the anomaly detection capability of the surrogate model. If an anomaly is identified by the surrogate model, then the perturber identifies the best possible adversarial action to perform (e.g., replacing/dropping a logkey or keeping the logkey as it is), in order to minimize changes to the input stream and thus be able to keep up with the speed of the incoming flow of logkeys. Note that the perturber does not require any knowledge about the internal parameters of the surrogate model, rather it only needs an indication of whether a sequence of logkeys is anomalous or not.

For a typical anomaly detection system, an anomaly flag raised at any point in a session would result in the whole session being marked as anomalous. Therefore, the adversarial modification of any logkey should not adversely affect other correlated future logkey values. Failure to adhere to this criterion may increase the probability of an anomaly being raised in the future within the same session. To address this issue, LAM models the perturber as a reinforcement learning agent leveraging a deep learning algorithm that operates in a partially observable environment to capture the future effect of the current action.

4.1 Why reinforcement learning?

We propose a solution based on reinforcement learning (RL) to overcome the high computational overhead of a brute-force search approach. During the attack, at each time instance t, the RL agent takes one of the two actions: (1) $drop(x_t)$ which drops the last logkey x_t in the observation sequence O; (2) $replace(x_t, x'_t)$ which replaces the last observed logkey x_t in O with x'_t. If x'_t is the same as x_t, then it means that the logkey x_t remains unchanged. Therefore, given L logkey types, there are $L + 1$ possible perturbation actions at each time step.

Although it is possible to try each of the $L + 1$ actions at each time step, this approach is inefficient. As an action taken at time step t may affect the future actions, LAM must be able to forecast how an action carried out in the present affects the future during the attack. Assume that the sliding window used by the operational anomaly detection model is m. As the number of possible actions at each time step is $L + 1$ and an action taken at a time step will affect at least m future time steps, there are $(L + 1)^m$ combinations of actions to be tried for m time steps. The combination of all possible modifications in an attacker's action space can be large even for a small number of logkeys, making it hard to find in real time the optimal action that can evade the operational anomaly detection model. In LAM, the RL agent learns an optimal policy during its offline training phase by taking into consideration the future impact of current actions. During the attack, the RL agent directly identifies an adversarial action without trying all possible actions.

4.2 Why deep reinforcement learning?

One challenge in attacking anomaly detection models is the large state space an RL agent needs to search during the training. The anomaly detection models take a sequence of m logkeys as an input. Similarly, the RL agent uses a sequence of n logkeys as its state. The number of states in the search space that the RL agent needs to explore during the training is L^n, which grows exponentially when n increases. Traditional reinforcement learning approaches use a tabular method to store all the state transitions in order to identify optimal actions resulting in desirable states. In our problem setting, we need to identify actions that can convert an anomalous sequence to a benign sequence. As the search space is large, the tabular method requires a large amount of memory to store the state transition table.

To address this issue, we employ a Deep Neural Network (DNN) to construct the RL agent. We treat the relationship between an input sequence (the state) and a desirable adversarial action as a non-linear mapping where the DNN is used as a blackbox function approximator for the mapping. This removes the need to store the state transitions in a tabular way. The RL agent is also model-free, so the agent does not need to know how the logkeys are generated by the actual system and hence does not need to individually compute the transition probabilities for each state action pair.

4.3 Dealing with a large search space

Using DNN reduces the memory usage of RL agent, but does not circumvent the issue regarding the need to explore a large search space. To address this issue, we train the RL agent in an offline setting to create adversarial perturbations on a sample of system log sessions with known anomalies. We utilize two approaches in our offline training phase to ensure that the agent explores the state space sufficiently while learning an optimal policy. The first method is called *experience replay* [35]. During experience replay, the agent learns to perturb logkeys of anomalous sessions. The result from each perturbing attempt may be different for each training iteration (epoch). All perturbing attempts (successful or not) are valuable experience that is used iteratively when updating parameters of the DNN. At each training epoch, we store the information pertaining to these perturbing attempts in a cyclic buffer called the *replay memory*. When the RL agent updates its internal parameters, the agent trains on a subset of its past experience stored within its replay memory. Therefore in a given epoch, the agent learns from

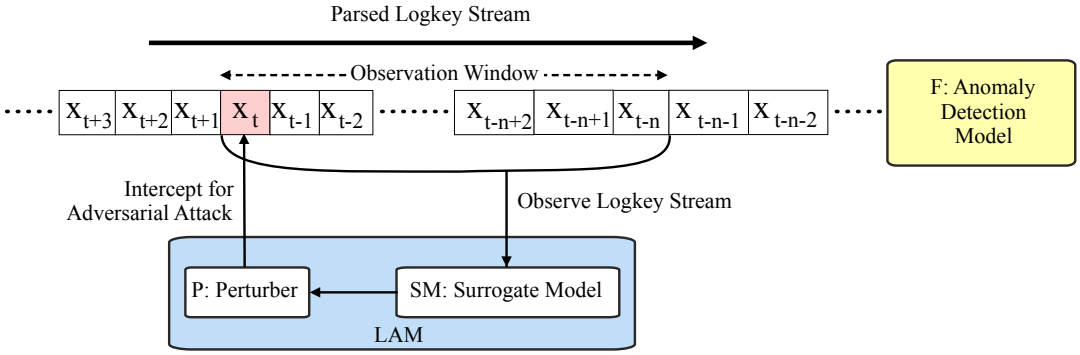


Figure 2: The architecture of LAM operating at time step t

multiple cycles of perturbing attempts made in the past, effectively learning more about the state space.

As the second solution, we use a *decay based approach* that gradually manages the exploration-exploitation in reinforcement learning [36]. The RL agent must explore a sufficient portion of the search space while ensuring exploiting the best actions that result in desirable states. Exploitation without exploration may result in the RL agent learning a sub-optimal local policy that may only work for some anomalous sessions. To address this issue, we use a threshold ($\epsilon \in [0, 1]$) based search during our offline training. If a randomly generated number is greater than the threshold, then a known optimal action will be taken; otherwise a random action will be taken. We define ϵ as a decreasing threshold that starts with a large value (~1) and decreases to a smaller value with each epoch. This is represented as $\epsilon = \epsilon_{end} + (\epsilon_{start} - \epsilon_{end}) \exp\{\frac{-1.epoch}{\epsilon_{decay}}\}$, where $\epsilon_{start}, \epsilon_{end}$, and ϵ_{decay} are user defined parameters that determine the rate at which the threshold decreases and *epoch* is the current iteration of training. In this approach, the RL agent explores the state space more often during the initial epochs of training because ϵ is a larger value. Gradually, as ϵ decreases, the agent exploits more often. Our experimental results show that, the above two approaches when used together lead to fast convergence in training (Figure 4 in Section 6.2.3).

5 ALGORITHM DETAILS

In this section, we use O to denote the observation sequence and n to denote the size of the state of the RL agent. Below, we describe the algorithmic details of LAM.

In LAM, the RL agent is trained offline to learn the best policy in perturbing streaming logs with minimal changes to evade the detection by the surrogate model. The attacker then uses this policy to make immediate decisions when performing a real-time evasion attack against the operational anomaly detection model.

The RL agent operates in a state space, where each state consists of n logkeys, to perturb the logkey stream. Its transition is assisted with an observation sequence O with the past $n + 1$ logkeys in the perturbed stream. Thus, at any time LAM only needs to remember $n + 1$ past logkeys in a given session. During the bootstrapping phase when fewer than $n + 1$ logkeys are observed, the observation sequence O is constructed by appending a filler value -1 prior to all observed logkeys up to a length of $n + 1$. The initial state $s_1 = O[2 : n + 1]$ is given as an input to the DNN model, which contains all logkeys in O except the first one. We use an LSTM model as our DNN because the LSTM model can better capture the temporal correlation in time series data than alternative DNN models such as feed forwarding neural networks and convolutional neural networks [17]. At each state s_t, the LSTM model outputs a score for each possible action (i.e., the action-value). During the attack, the RL agent selects the action that has the highest score without exploring other options.

Moving forward, we use an intermediate state s'_t to compute the reward based on the action taken by the RL agent, which occurs after an action taken by the RL agent but before the next logkey x_{t+1} is observed. If the action is $drop(x_t)$, then $s'_t = O[1 : n]$. If the action is $replace(x_t, x'_t)$, then $s'_t = O[2 : n] \circ x'_t$ where $\circ$ represents the concatenation. Given an action a, the reward at time step t is computed using the surrogate model SM, the logkey observed x_t ($s_t[n]$), and the intermediary state s'_t as follows:

$$r_t = \begin{cases} 1.0 & \text{if } SM(s'_t) = \text{False } \& \; s_t[n] = s'_t[n] \\ 0.5 & \text{if } SM(s'_t) = \text{False } \& \; s_t[n] \neq s'_t[n] \\ -1.0 & \text{if } SM(s'_t) = \text{True} \end{cases} \quad (1)$$

$SM(s'_t)$ is $True$ if the perturbation by the agent causes an anomaly. The goal of the RL agent is to learn an optimal policy that maximizes the expected reward. If the perturbation causes an anomaly, then a negative reward of -1 is given; otherwise, a positive award is given. Additional rewards are given when the RL agent takes no action and no anomaly is flagged, which aims to train the RL agent to make the least possible perturbations to maintain the imperceptibility.

At the next time step $t + 1$, the new state s_{t+1} is computed from the intermediate state s'_t and the new logkey x_{t+1} as $s_{t+1} = s'_t[2 : n] \circ x_{t+1}$. Similarly, the observation sequence O is updated as $O = s'_t \circ x_{t+1}$.

Below, we use an example to explain the RL agent's behavior. Suppose that at time instance $t = 8$, O is $\{-1, -1, -1, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$. Here $n = 10$, $x_1, \ldots, x_8$ are logkeys, and -1 is the filler value. If the RL agent opts to replace x_8 with x'_8, then the intermediate state s'_8 is $\{-1, -1, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x'_8\}$. If the agent drops the logkey x_8, then $s'_8 = \{-1, -1, -1, x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$. When a new logkey x_9 arrives, the agent's observation sequence O is updated as $s'_8 \circ x_9$ and state s_9 is computed as $s'_8[2 : 10] \circ x_9$.

5.1 Offline Training Algorithm

Procedure `OfflineTraining`
  Data: The number of training iterations num_epoch, the anomaly sessions $\mathbf{X}'$, the decay parameters $\epsilon_{start}, \epsilon_{end}$ and ϵ_{decay}, the discount factor γ, the target policy update period M
  Result: The trained model Q_{policy}
1 Randomly initialize the weights of LSTM for Q_{policy}
2 $Q_{target} = Q_{policy}$
3 **for** $epoch \in [1, num\_epoch]$ **do**
4 Randomly pick a session $X' \in \mathbf{X}'$
5 $O = \{-1, \ldots, -1, X'[1]\}$
6 $s_1 = O[2 : n+1]$
7 **for** $t \in [1, |X'|]$ **do**
8 $x_{t+1} = X'[t+1]$ or $null$ if $t == |X'|$
9 $\epsilon = \epsilon_{end} + (\epsilon_{start} - \epsilon_{end}) \exp\{\frac{-1.epoch}{\epsilon_{decay}}\}$
10 **if** *random(0..1)* $> \epsilon$ **then**
  Select an optimal action $a_t = argmax_a(Q_{policy}(s_t, a))$
  else
11 Randomly select an action a_t
  end
12 Compute the intermediate state s'_t
13 Compute the reward r_t based on $SM(s'_t)$
14 $O = s'_t \circ x_{t+1}$
15 $s_{t+1} = s'_t[2 : n] \circ x_{t+1}$
16 Add training sample $< s_t, a_t, s_{t+1}, r_t >$ to replay memory E
  end
17 Randomly pick a subset of training samples $E' \subset E$
18 **for** *each training sample* $< s_t, a_t, s_{t+1}, r_t > \in E'$ **do**
19 Compute action value $y_t = Q_{policy}(s_t, a_t)$
20 Compute next state action value $y_{t+1} =$ $\begin{cases} max_a(Q_{target}(s_{t+1}, a)) & \text{if } s_{t+1} \text{ is a non-terminal state} \\ 0 & \text{otherwise} \end{cases}$
21 Compute the loss between $(y_t, (r_t + \gamma \cdot y_{t+1}))$ according to Eq. (2)
22 Adjust weights in $Q_{policy}(s_t, a_t)$ based on the loss computed
  end
23 **if** $epoch \,\%\, M == 0$ **then**
24 $Q_{target} = Q_{policy}$
  end
25 **return** Q_{policy}

Algorithm 1: Offline training of the RL agent

Algorithm 1 gives the pseudocode for the offline training procedure of the RL agent. In the pseudocode, we use $\mathbf{X}'$ to denote all anomaly sessions, X' a single anomaly session, and $|X'|$ the length of X'.

Lines 1 – 2 in Algorithm 1 initialize the RL agent. In each training epoch (i.e., a loop in Line 3), the algorithm works in two stages. First, the algorithm perturbs a randomly chosen anomaly session (Lines 4 – 16) and then performs one training cycle (Lines 17 – 22).

The purpose of the offline training algorithm is to train an LSTM model that can be used to predict the action value of each state-action pair. During the attack, given a current state s, the attacker always takes the action that results in the largest action value predicted by the LSTM model. The LSTM model outputs a scalar action value [36] associated with each state-action pair (s, a). Algorithm 1 trains two LSTM models, Q_{policy} and Q_{target}. Model Q_{policy} is returned by the algorithm (Line 25) and is used later by the adversary in real-time evasion attacks (see Section 5.2). We next explain the reason for the additional model Q_{target}.

Chattering is a common issue when using DNN models within reinforcement learning [36]. This issue implies that using a deep learning model as a function approximator to identify the mapping between a state and an action may not always result in stable convergence of the RL agent towards learning an optimal policy. A workaround to this problem is to instantiate two LSTM models of the same architecture during the training time, each predicting the action value in a given state. The *target model* Q_{target} acts as a reference or a target for the training objective, while the *policy model* Q_{policy} is updated for each training epoch. Given a state s and an action a, $Q(s, a)$ returns a scalar action-value where Q is either Q_{target} or Q_{policy}. Initially both Q_{target} and Q_{policy} are initialized with the same random weights (Line 2). In each $epoch$, the policy model is trained (Lines 18 – 22) whereas the target model is left untouched. The target model is then updated to be the same as the policy model every M epochs where M is a user-defined variable (Lines 23 – 24).

We next explain the details of each training epoch. The agent initializes its observation sequence O and its initial state s_1 in Lines 5 – 6. For each randomly selected anomaly session X', the algorithm perturbs the logkeys in this session. Given the next logkey x_{t+1}, the agent identifies the adversarial action to take based on the threshold ϵ, where $0 \leq \epsilon \leq 1$ (Lines 9 – 11). With probability $1 - \epsilon$, the agent selects the action a_t associated with the maximum action-value (i.e., $a_t = \text{argmax}_a Q_{policy}(s_t, a)$). With probability ϵ, the agent selects a random perturbation action.

Afterwards, the algorithm computes the intermediate state s'_t (Line 12) followed by calculating the reward as per Equation 1 (Line 13). Then the observation O is updated and the next state s_{t+1} is computed (Lines 14-15). The training sample $\langle s_t, a_t, s_{t+1}, r_t \rangle$ is then added to the replay memory (Line 16). At each training cycle, the RL agent picks a random subset of training samples from its replay memory to train from (Line 17). For each sample, the agent computes the loss and updates the weights of Q_{policy} (Lines 19 – 22). The loss is computed as a smooth L1 loss [15] between the current action value $y_t = Q_{policy}(s_t, a_t)$ and the summation of the immediate reward obtained from the current time step and the discounted best action value for the next time step (i.e., $r_t + \gamma \cdot y_{t+1}$). More specifically the loss function is given in the following equation:

$$loss = \begin{cases} 0.5(y_t - (r_t + \gamma \cdot y_{t+1}))^2 & \text{if } |y_t - (r_t + \gamma \cdot y_{t+1})| < 1 \\ |y_t - (r_t + \gamma \cdot y_{t+1})| - 0.5 & \text{otherwise} \end{cases} \tag{2}$$

We utilize the smooth L1 loss because it is less sensitive to outliers than the mean squared error loss (MSELoss) and in some cases prevents exploding gradients [15]. The action value for the next

time step y_{t+1} is discounted by a factor $\gamma \in [0, 1]$, a user defined parameter. Note that in Line 20, y_{t+1} is computed with respect to the target LSTM model Q_{target} due to the chattering issue. With the loss computed in Line 21, the weights of Q_{policy} are updated using RMSProp [37] as the optimizer in Line 22. Once the training completes, the algorithm returns the policy model Q_{policy}.

5.2 Real-time Evasion Attack

Algorithm 2 shows how LAM performs the real-time attack at time step t. Before any logkey arrives, the observation O' is initialized to be $[-1, -1, ..., -1]$ of length n. For each incoming logkey x_t, the algorithm updates O' and the current state s_t (Line 1). The current state is then probed using the surrogate model SM for a potential anomaly flag (Line 2). If an anomaly flag is not raised (i.e. $SM(s_t) == False$), then O' is updated (Line 3) and the perturber P takes no action (Line 4). If an anomaly is likely to be flagged, an optimal perturbation action a_t is identified to avoid the anomaly (Line 5 – 8).

Procedure `Attack`
Data: incoming logkey x_t in the session, the observation O'
1 $O' = O' \circ x_t$; $s_t = O'[2 : n + 1]$
2 **if** $SM(s_t) == False$ **then**
3 $O' = s_t$
4 **return** ▷ No perturbation action is needed
else
5 Select action $a_t = \text{argmax}_a Q_{policy}(s_t, a)$
6 Compute the intermediate state s'_t based on sequence O'
7 $O' = s'_t$
8 Perform perturbation action a_t
9 **return**
end

Algorithm 2: Real-time evasion attack

6 EVALUATION

In this section, we first describe the datasets used in our experiments and the architectures and hyper-parametric tuning of the anomaly detection models and the RL agent. We then present our experimental results on attack effectiveness, speed, and imperceptibility.

6.1 Datasets and model parameters

We have evaluated LAM using two distributed system log datasets: HDFS [42] and the system logs collected from the DATAVIEW scientific workflow management system [22].

6.1.1 HDFS logs: HDFS is a commonly used benchmark dataset for log based anomaly detection systems [13, 38, 46]. The dataset contains Hadoop file system logs for map-reduce jobs on more than 200 Amazon EC2 Virtual Machines (VMs). The raw log files are grouped into sessions based on the field *block_id*, where each session is labeled for anomaly status by domain experts. The parsed dataset contains 24,396,061 log entries from 29 logkey events amounting to around 974, 762 sessions. We train the anomaly detection models on a random dataset containing 8000 benign sessions.

6.1.2 DATAVIEW logs: DATAVIEW is a scientific workflow management system that runs workflows inside Amazon EC2 VMs [23]. Logs collected from DATAVIEW record the status of scientific workflows executed on EC2 VMs, which includes the VM provisioning status, the communication between a local machine and a EC2 VM, and the task execution status. The system logs contain the interleaved execution traces of three scientific workflows, namely Ligo, WordCount and DiagnosisRecommendation [22]. The logs are grouped into sessions based on the type of workflow executed. The dataset contains synthetic anomalies due to workflow structural changes where the workflow structure is modified to manipulate the final results. The dataset contains 14,362 log sequences generated from 104 logkey events. We train the anomaly detection models on a dataset of 8000 benign samples.

6.1.3 Anomaly detection models: Table 2 gives the architectures and the hyper-parameters used to tune the Deeplog and AutoEncoder anomaly detection systems. The table also contains the true positive rate (TPr) and the false positive rate (FPr) associated with anomaly detection. In DeepLog, the Linear Layer outputs the conditional probabilities for all the logkeys for the next time step. In AutoEncoder, the inputs/outputs are a one hot encoded sequence of values equal to $L \times m$, where L is the number of logkeys and m is the sliding window size.

6.1.4 RL agent architectures: Table 3 gives the architecture and the hyper-parameters of the RL agent tuned for the whitebox attack, in which the target model and the surrogate model are the same. In our experiments, we maintain a replay memory E (Line 16 of Algorithm 1) of 20000 samples. For all experiments, the hyper-parameters $\epsilon_{start}, \epsilon_{end}, M$, and $|E'|$ in Algorithm 1 are kept at constant values of 0.90, 0.05, 150, and 256, respectively. The training datasets for the RL agent contain 50 anomaly sessions (i.e., $|\mathbf{X}'|$), which are randomly sampled without replacement for both HDFS and DATAVIEW datasets.

In the whitebox attack, the attacker has direct access to the anomaly detection models trained. In the graybox attack, we train a surrogate model separately, which has the same architecture and hyper-parameters as the target model, but different training weights (which are randomly initialized). In the blackbox attack, when attacking DeepLog, we use AutoEncoder as the surrogate model. When attacking AutoEncoder, the surrogate model is DeepLog.

6.2 Experimental results

This section presents the experimental results of LAM. All experimental results pertaining to the real-time attack effectiveness, the attack imperceptibility, and the attack speed (shown in Figure 3 and Table 4) were obtained on a dual two-core 3.30 GHz Intel Xeon machine with 8 GB memory. The pre-trained deep learning models were used in all experiments. The deep learning models were trained and tuned on a 2.3-3.7 GHz Intel Xeon Gold 6140 machine with NVIDIA Tesla P100 12GB GPU.

6.2.1 Attack effectiveness: Figure 3 shows the effectiveness of LAM on DeepLog and AutoEncoder. Figure 3(a) gives the true positive rate of LAM on the HDFS dataset. The figure shows that, for the whitebox attack, the true positive rate of DeepLog and AutoEncoder is reduced by approximately 80% and 60%, respectively. For the

Datasets	Models	Architecture	Hyper-Parameters	TPr	FPr
HDFS	DeepLog	LSTM(#weights = 64, #layers = 2), Linear(64×29)	sliding window (m) = 10, learning rate = 0.01, # of candidates (g) = 9	0.9066	0.0023
	AutoEncoder	*Encoder:* { Linear(290×256), Linear(256×128), Linear(128×64), Linear(64×32) } *Decoder:* { Linear(32×64), Linear(64×128), Linear(128×256), Linear(256×290) }	sliding window (m) = 10, learning rate = 0.01, threshold (θ) = 0.1	0.9997	0.0019
DATAVIEW	DeepLog	LSTM(#weights = 128, #layers = 2), Linear(128×104)	sliding window (m) = 10, learning rate = 0.01, # of candidates (g) = 17	1.0000	0.0224
	AutoEncoder	*Encoder:* { Linear(1040×1024), Linear(1024×512) } *Decoder:* { Linear(512×1024), Linear(1024×1040) }	sliding window (m) = 10, learning rate = 0.01, threshold (θ) = 0.2	1.0000	0.0613

Table 2: Anomaly detection model architectures

Datasets & Models	RL agent Architecture	Hyper-Parameters
HDFS		
DeepLog	LSTM(#w = 128, #l = 4), Linear(128×30)	state size (n) = 11 γ = 0.85 ϵ_{decay} = 2000
AutoEncoder	LSTM(#w = 256, #l = 4), Linear(256×30)	state size (n) = 10 γ = 0.85 ϵ_{decay} = 3000
DATAVIEW		
DeepLog	LSTM(#w = 128, #l = 4), Linear(128×105)	state size (n) = 11 γ = 0.85 ϵ_{decay} = 3000
AutoEncoder	LSTM(#w = 128, #l = 4), Linear(128×105)	state size (n) = 10 γ = 0.95 ϵ_{decay} = 3000

Table 3: The architecture and hyper-parameters of tuned RL agent where #w and #l represent the number of weights and layers in the LSTM module, respectively

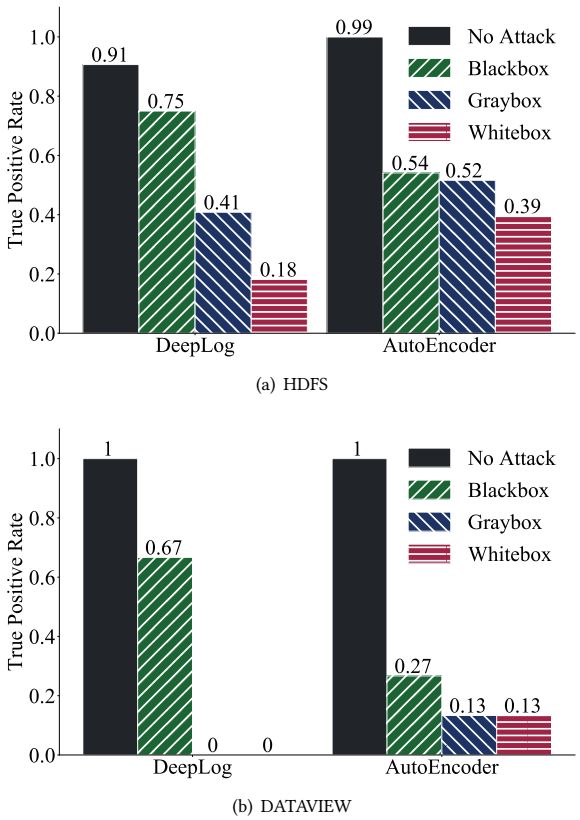


Figure 3: Attack effectiveness of LAM

DATAVIEW dataset (Figure 3(b)), we observe a drop of 100% in the true positive rate with DeepLog and a reduction of around 87% with AutoEncoder. As expected, the whitebox attack demonstrates the greatest damage to the anomaly detection capability of the two models, with an average of around 89.9% for DeepLog and 73.5% for AutoEncoder.

We study the transferability of LAM via graybox and blackbox attack scenarios. Transferability means the likelihood of success for an indirect attack by using a different model as the target. LAM succeeds in evading anomaly detection in the graybox attack, but not to the same degree as the whitebox attack. This is because, even though both the target and the surrogate models have superior anomaly detection capability, their internal model parameters are not the same. As the two models may raise anomalies at different locations in the same session, anomalies detected by the target model may not always be masked by LAM which uses a different surrogate model to decide which logkeys should be perturbed. Figure 3(a) shows that, for the HDFS dataset, the graybox attack reduces the true positive rate of DeepLog from 91% to 41% and AutoEncoder from 99% to 52%. For the DATAVIEW dataset, the graybox attack performs as well as the whitebox attack, as shown in Figure 3(b).

The blackbox attack is the hardest of the three types of attacks, as evidenced by the least decreases in the true positive rates for all the cases in Figure 3. Interestingly, the figure shows that the blackbox attacks targeting the AutoEncoder model with DeepLog as the surrogate model are more transferable than those in the opposite direction. Figure 3(a) shows that the blackbox attack reduces the

Datasets	Attacked Models	Real-Time Responsiveness (Time for a single attack)	Attack Imperceptibility		
			Avg. #Modifications per Session	Avg. Session Length	Modified Percentage
HDFS	DeepLog	0.65ms	1.58	25.28	6.24 %
	AutoEncoder	0.23ms	3.10	25.28	12.27 %
DATAVIEW	DeepLog	0.62ms	3.15	36.85	8.55 %
	AutoEncoder	0.37ms	4.64	36.85	12.59 %

Table 4: Properties of real-time adversarial attack

true positive rate of the AutoEncoder model by 45.9% for the HDFS dataset, whereas the reduction for the DeepLog model is only 17.3%. We see a similar trend in Figure 3(b), which shows a larger drop in the anomaly detection accuracy when targeting the AutoEncoder (73.3%) as opposed to the DeepLog (33.3%). The above results indicate that the attacks generated with DeepLog as the surrogate model are more transferable than those using AutoEncoder.

6.2.2 Attack speed and imperceptibility: Table 4 shows the real-time responsiveness and attack imperceptibility of LAM. We observe from the table that the speed of perturbing one logkey is fast irrespective of the datasets used, which is about 0.63ms for DeepLog and 0.29ms for AutoEncoder. Regarding attack imperceptibility, for a single session in HDFS that contains around 25 logkeys, on average LAM changes 1.5 and 3.1 logkeys when attacking DeepLog and AutoEncoder, respectively. We see a similar trend for the DATAVIEW experiments. Our experimental results show that AutoEncoder is more robust against our attacks. When considering all dataset and model combinations, LAM perturbs around 9.9% logkeys on average to evade anomaly detection.

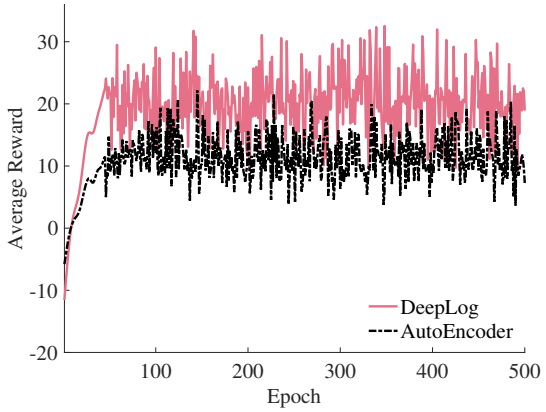
To illustrate the real-time effectiveness of LAM, we experiment with attacking sessions of logkeys using a brute force search algorithm assisted by the surrogate model. For each session, the surrogate model is used to identify the attack entry points. Once an anomaly is identified by the surrogate model, a brute force search is performed to traverse through all possible transitions until an action that results in fooling the surrogate model is observed. Unlike LAM, the brute force search needs to probe the surrogate model at each iteration to ensure that an adversarial action does not result in any anomaly flag raised throughout the entire observed session. In the worst case, the total number of searches required for each iteration can be exponentially large, i.e., L^n where L is the total number of logkeys and n is the size of the state. For example, when DeepLog is used as the surrogate model, the state size for the HDFS dataset is 29^{11} and the state size for the DATAVIEW dataset is 104^{11}. In our experiment, we attacked 20 randomly selected sessions from both the HDFS and the DATAVIEW datasets where the surrogate model is DeepLog. Given that the search time of LAM is bounded by an order of seconds, we consider an upper limit of 24 hours as the total search time for a single session using the brute force search. The brute force search is terminated if the search time exceeds 24 hours. Our experimental results show that the average time for attacking a single session in HDFS and DATAVIEW datasets using the brute force search is 12.87 hours and 17.03 hours respectively. Considering an attack on the whole session, this is approximately 2.8×10^6 times slower than LAM, which illustrates the gains in speed during the attack phase using the RL agent. Overall, the brute force search perturbs approximately 30.4% logkeys. We expect similar results for brute force search when the surrogate model is the AutoEncoder.

6.2.3 Convergence in training: Figure 4 shows the convergence of the average reward per session obtained by the RL agent during the training phase for the first 500 epochs. The average reward obtained per session is plotted with respect to the training progress for the models with parameters mentioned in Table 3.

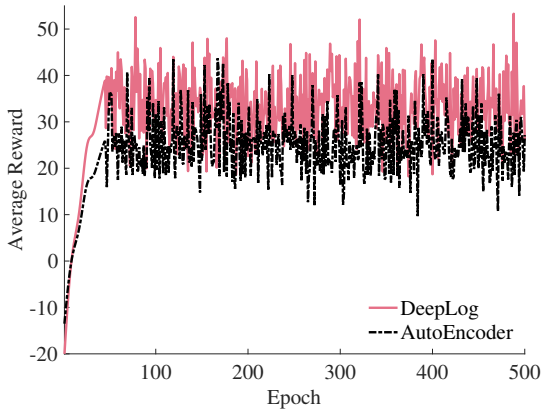
The maximum reward an agent can obtain in each session is the number of logkeys in the session. On average, each session in HDFS and DATAVIEW contains around 25 and 36 logkeys, respectively. In both datasets, the reward is negative at the beginning. This is because, at the beginning, the perturber is not able to create adversarial modifications that fool the surrogate model. However, it quickly rises to a positive value and fluctuates slightly below 25 and 36 as shown in Figures 4(a) and 4(b). The reason behind the sudden rise is that at each epoch, the perturber trains on a batch of training samples stored in its replay memory. Additionally, we observe that the RL agent obtained higher reward for DeepLog than for AutoEncoder. This also implies that the RL agent learns to attack DeepLog more successfully than AutoEncoder.

6.2.4 Hyper-parametric tuning: In this section, we provide insights on hyper-parametric tuning for our real-time adversarial attack method. Due to the space constraint, we present only results from tuning the attack against DeepLog on the HDFS dataset. We observe similar trends for other model-dataset combinations. Figure 5 shows the true positive rate for varied values of γ, the hyper-parameter used for calculating the loss and for varied architectural changes in the LSTM model (see Eq. (2)). As shown in Figure 5, the attack is more successful when γ increases, irrespective of the internal architecture of the LSTM component. This is in line with many standard RL agent tuning tasks where higher values of $\gamma \in [0.8, 1]$ seem to have better results. However, we note that once the RL agent achieves a stable state (#layers = 4, #weights = 128), there is no visible change even when γ increases further.

We make similar observations for ϵ_{decay} (Line 9 in Algorithm 1), where a substantially larger value (i.e., around 2000 or 3000) yields better results. This is also expected as larger ϵ_{decay} enables the RL agent to explore the search space more often, thus circumventing the issue of getting stuck at a local optima to a higher degree. We note an interesting pattern with regards to selecting the optimal architecture for the LSTM component. As seen in Figure 5, models with less depth (i.e., #layers = 2) are not able to successfully make adversarial attacks. As the size of the model increases, either through the number of stacked layers or the number of weights used, we note a greater reduction in the true positive rate. However,



(a) HDFS



(b) DATAVIEW

Figure 4: Convergence in training for the RL agent

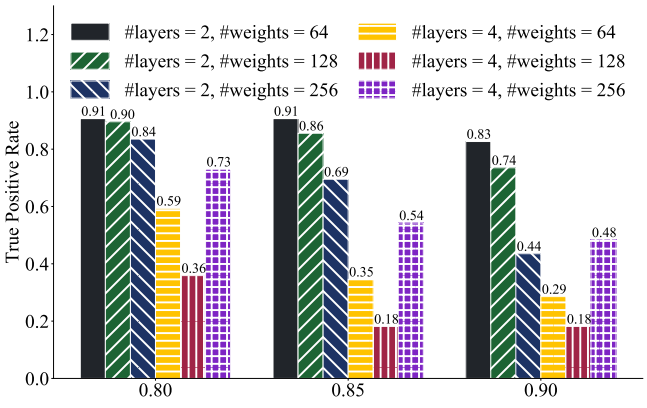


Figure 5: Impact of parametric change on attack performance for $\gamma = [0.80, 0.85, 0.90]$

at a certain point (#layers = 4, #weights = 256 for HDFS + DeepLog), we see a sudden increase in the true positive rate. We see similar trends for the DATAVIEW dataset and the AutoEncoder model, where the true positive rate decreases and then increases with an increasing model size, albeit at different points. Therefore, our rule of thumb for hyper-parameter tuning is to initially start with a smaller model (i.e., #layers = 2, #weights = 64) and then double it at each step until the true positive rate rebounds from a bottom value.

7 DISCUSSIONS

In this section we discuss how to extend LAM for attacking raw log streams directly as well as potential defense schemes.

Extension to attacking raw log streams: Our approach considers logkeys as the input to the anomaly detection model for transmission efficiency. The method can be extended to scenarios where raw log entries are transmitted over the network and fed to the anomaly detection system directly, which parses them into logkeys for further anomaly detection. In the attack, the attacker can intercept raw log entries, convert them to logkeys, use LAM to optimize perturbations, and finally translate the modified logkeys back to raw log entries. As the parameter values are discarded for logkey anomaly detection upon parsing, the attacker can then manipulate the raw logs by intercepting and replacing individual log entries with valid text templates associated with the mapped logkeys.

Potential Defensive Approaches: Below, we discuss potential defense mechanisms against LAM. As LAM relies on interception and modification of the logkey stream fed to the anomaly detection module, it is possible to prevent the attack by ensuring the integrity of the logkeys during their transmissions from their collection sites to the machine where anomaly detection is performed. However, a few technical challenges are involved here. First, a trusted path must be established from the place where the raw logs are generated to the anomaly detection module. For example, a common practice in distributed systems is that data transmissions are secured between two machines (e.g., using the popular `scp` utility). This may not be sufficient because the adversary may run malware on the machine collecting raw logs to modify the logkeys before their transmissions or on the machine executing the anomaly detection model after the logkeys are received. Second, although an application-level end-to-end security protection scheme can be deployed to prevent tampering of logkeys needed by the LAM attack, it introduces extra overhead and complexity to key management in a distributed system where other keys are also needed at the infrastructure level (e.g., Kerberos for Hadoop). The co-existence of multiple sets of unrelated keys in the same distributed system renders it difficult to reason about trust relationships across different components.

Another potential defense strategy is to increase the adversarial robustness by leveraging the disparity in transferability. Our experimental results show that although LAM is able to transfer attacks across different model architectures, the blackbox attack has the least success rate. Instead of utilizing one operational anomaly detection model, it is possible to deploy an ensemble of anomaly detection models within the anomaly detection system. For each session, one of the models within the ensemble is randomly chosen for the purpose of detecting anomalies. If the model chosen is different from the surrogate model, then the attack success rate will be less than an optimal whitebox scenario. From a defensive position, the defender must ensure that all models within the ensemble

are trained adequately such that they all have reasonable anomaly detection capability, which could be computationally intensive.

From an attacker's perspective, LAM can be used to attack even with this defense in place. If the attacker knows that an anomaly model ensemble is used, then the entire ensemble itself can be used as a group of surrogate models during the training phase. For each training iteration, one of the models within the ensemble can be chosen to derive the reward of the RL agent. This would ensure that LAM learns to fool multiple model architectures at the same time. However, this opens a possible question with respect to achieving stable convergence during training, because each training iteration could be anchored on a different surrogate model. This also opens the question of identifying attack entry points within the session, because different models may identify anomalies at different sequential points in the stream. It is worth noting that adversarial attacks and defences form a kind of cat and mouse game, where one party may triumph over the other depending on the information available to the other party. We leave the investigation of this potential defensive strategy and how LAM may fair against them as future work.

8 RELATED WORK

In this section we survey the related works on log-based anomaly detection and adversarmail machine learning attacks.

8.1 Log-based anomaly detection

There have been a number of attempts at identifying malicious behaviors from computer system logs (e.g., [7, 9, 21]). Machine learning based system log anomaly detection can be broadly classified as supervised, unsupervised, and deep learning based methods. The work in [21] provides a comparison between existing supervised and unsupervised machine learning based system log anomaly detection models. Support Vector Machines (SVM) [25], Decision Tree based methods [10], and data mining techniques such as MapReduce [6] have been used as supervised machine learning methods to detect anomalies based on system logs. Unlike supervised methods, unsupervised techniques have the advantage of not needing a labeled dataset at its learning stage. LogCluster [26] is one such unsupervised model that uses clustering based techniques to detect anomalous events by generating clusters for normal/abnormal samples. Other unsupervised machine learning techniques include Statistical methods such as Principal Component Analysis (PCA) [42] that operate by reducing a high dimensional data space to a lower dimension, and techniques such as Invariant Mining [28] that identify common linear relationships in benign log sessions and use them to flag anomalies.

Recently, Deep Learning based models have been used for anomaly detection with great success [9]. DeepLog [13] uses the LSTM model to learn the conditional probabilities of log sequences and raises an anomaly flag whenever newly arrived logs are not predicted by the model. Desh [11] uses LSTM to predict the lead time for future node failure in HPC logs. Researchers have also proposed to combine the LSTM architecture with the attention mechanism [8, 46] to provide an additional level of interpretability for root causes of anomalies identified in system logs. AutoEncoders have also been widely used for anomaly detection (e.g., [4, 5, 19, 34]). In recent years, there have been models constructed by combining AutoEncoder models with LSTM [18], where the AutoEncoder is used to identify a hidden representation of the data, which is then given as the input to an LSTM model to detect anomalies. As anomalies occur sparingly, a common problem plaguing anomaly detection is the imbalance between benign and abnormal training samples. To address this issue, Generative Adversarial Networks (GANs) have been used to improve anomaly detection models [38].

8.2 Adversarial machine learning attacks

Although Deep Learning models have achieved superior performance in many areas, there have been successful attempts at misleading many DNNs at test time by carefully crafting input samples [3]. Some of the existing adversarial attacks are summarized in [40, 43]. Although many attacks were initially proposed in the domain of image classification [1, 27, 40], some recent works have focused on generating adversarial examples targeting text-based NLP domains [2, 45]. Many existing evasion attacks generate adversarial samples in an offline manner [40, 43, 45]. LAM, in contrast, is aimed at real-time evasion of anomaly detection systems. Moreover, attacks targeting NLP applications operate on character/word modifications and do not work well for system logs that have stricter syntactical structure. In addition, newly proposed systems such as LogRobust [46] can successfully parse unstable raw logs, which further reduces the effectiveness of previous attacks.

Recently, there have been some works on creating real-time adversarial attacks. CAG (Content-Aware Adversarial Attack Generator) [30] is one such attack that uses a generative model in the form of U-Net [33] to make adversarial perturbations to images. GAP (Generative Adversarial Perturbations) [31] is another similar image-based generative model that uses a ResNet model within its architecture. Li et al. [24] propose to use an offline-trained fixed discriminator and a generator that is trained to create adversarial perturbations against the discriminator for real-time video classification. The work in [14] introduces an evasion attack against anomaly detection in cyber-physical systems. These previous efforts use generative models to craft adversarial samples, which differ from the reinforcement learning approach developed in our work. Xie et al. [39] propose an attack against speech recognition by generating universal perturbations from repeated playback of fixed-length universal noise.The adversarial attacks proposed by Gong et al. [16] targeting speech recognition models the attack generator as a RL agent. Their approach modifies speech patterns based on imitation of pre-exiting attack trajectories performed by offline attack models. Our approach, in contrast, does not require any pre-known adversarial attacks on anomaly detection models. To the best of our knowledge, there are no adversarial evasion attacks against deep learning based anomaly detection on distributed system logs.

9 CONCLUSION

In this paper, we propose LAM, a real-time evasion attack that perturbs streaming logs in distributed systems with minimal modifications to evade anomaly detection. LAM models the perturber as a reinforcement learning agent that operates in a partially observable environment. The RL agent is trained offline using a surrogate

model to predict the best perturbation action from its observations made in the current environment and then uses this prediction to make immediate decisions during the attack. Our experimental results show that LAM significantly reduces the true positive rate of DeepLog and AutoEncoder while achieving attack imperceptibility and real-time responsiveness. In the future, we plan to extend LAM to consider both logkeys and parameter values and investigate further defense mechanisms against LAM.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under grant OAC-1738929. We also thank the anonymous reviewers for their constructive comments.

REFERENCES

[1] Naveed Akhtar and Ajmal Mian. 2018. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access* 6 (2018), 14410–14430.

[2] Yonatan Belinkov and James Glass. 2019. Analysis methods in neural language processing: A survey. *Transactions of the Association for Computational Linguistics* 7 (2019), 49–72.

[3] Battista Biggio and Fabio Roli. 2018. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition* 84 (2018), 317–331.

[4] Andrea Borghesi, Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. 2019. Anomaly detection using autoencoders in high performance computing systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 9428–9433.

[5] Andrea Borghesi, Antonio Libri, Luca Benini, and Andrea Bartolini. 2019. Online anomaly detection in hpc systems. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 229–233.

[6] Jakub Breier and Jana Branišová. 2015. Anomaly detection from log files using data mining techniques. In *Information Science and Applications*. Springer, 449–457.

[7] Jakub Breier and Jana Branišová. 2017. A dynamic rule creation based anomaly detection method for identifying security breaches in log records. *Wireless Personal Communications* 94, 3 (2017), 497–511.

[8] Andy Brown, Aaron Tuor, Brian Hutchinson, and Nicole Nichols. 2018. Recurrent neural network attention mechanisms for interpretable system log anomaly detection. In *Proceedings of the First Workshop on Machine Learning for Computing Systems*. 1–8.

[9] Raghavendra Chalapathy and Sanjay Chawla. 2019. Deep learning for anomaly detection: A survey. *arXiv preprint arXiv:1901.03407* (2019).

[10] Mike Chen, Alice X Zheng, Jim Lloyd, Michael I Jordan, and Eric Brewer. 2004. Failure diagnosis using decision trees. In *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE, 36–43.

[11] Anwesha Das, Frank Mueller, Charles Siegel, and Abhinav Vishnu. 2018. Desh: deep learning for system health prediction of lead times to failure in hpc. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 40–51.

[12] Min Du and Feifei Li. 2016. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 859–864.

[13] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of ACM Conference on Computer and Communications Security*.

[14] Alessandro Erba, Riccardo Taormina, Stefano Galelli, Marcello Pogliani, Michele Carminati, Stefano Zanero, and Nils Ole Tippenhauer. 2019. Real-time evasion attacks with physical constraints on deep learning-based anomaly detectors in industrial control systems. *arXiv preprint arXiv:1907.07487* (2019).

[15] Ross Girshick. 2015. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*. 1440–1448.

[16] Yuan Gong, Boyang Li, Christian Poellabauer, and Yiyu Shi. 2019. Real-time adversarial attacks. *arXiv preprint arXiv:1905.13399* (2019).

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[18] Aarish Grover. 2018. Anomaly Detection for Application Log Data. (2018).

[19] Simon Hawkins, Hongxing He, Graham Williams, and Rohan Baxter. 2002. Outlier detection using replicator neural networks. In *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 170–180.

[20] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 33–40.

[21] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. 2016. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 207–218.

[22] J Dinal Herath, Changxin Bai, Guanhua Yan, Ping Yang, and Shiyong Lu. 2019. RAMP: Real-Time Anomaly Detection in Scientific Workflows. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 1367–1374.

[23] Andrey Kashlev and Shiyong Lu. 2014. A system architecture for running big data workflows in the cloud. In *2014 IEEE International Conference on Services Computing*. IEEE, 51–58.

[24] Shasha Li, Ajaya Neupane, Sujoy Paul, Chengyu Song, Srikanth V Krishnamurthy, Amit K Roy Chowdhury, and Ananthram Swami. 2018. Adversarial perturbations against real-time video classification systems. *arXiv preprint arXiv:1807.00458* (2018).

[25] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. 2007. Failure prediction in ibm bluegene/l event logs. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*. IEEE, 583–588.

[26] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log clustering based problem identification for online service systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 102–111.

[27] Yen-Chen Lin, Zhang-Wei Hong, Yuan-Hong Liao, Meng-Li Shih, Ming-Yu Liu, and Min Sun. 2017. Tactics of adversarial attack on deep reinforcement learning agents. *arXiv preprint arXiv:1703.06748* (2017).

[28] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. 2010. Mining Invariants from Console Logs for System Problem Detection.. In *USENIX Annual Technical Conference*. 1–14.

[29] Adam J Oliner, Alex Aiken, and Jon Stearley. 2008. Alert detection in system logs. In *2008 Eighth IEEE International Conference on Data Mining*. IEEE, 959–964.

[30] Huy Phan, Yi Xie, Siyu Liao, Jie Chen, and Bo Yuan. 2019. CAG: A Real-time Low-cost Enhanced-robustness High-transferability Content-aware Adversarial Attack Generator. *arXiv preprint arXiv:1912.07742* (2019).

[31] Omid Poursaeed, Isay Katsman, Bicheng Gao, and Serge Belongie. 2018. Generative adversarial perturbations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4422–4431.

[32] Maria A Rodriguez, Ramamohanarao Kotagiri, and Rajkumar Buyya. 2018. Detecting performance anomalies in scientific workflows using hierarchical temporal memory. *Future Generation Computer Systems* 88 (2018), 624–635.

[33] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*. Springer, 234–241.

[34] Mayu Sakurada and Takehisa Yairi. 2014. Anomaly detection using autoencoders with nonlinear dimensionality reduction. In *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*. 4–11.

[35] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).

[36] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[37] Tijmen Tieleman and Geoffrey Hinton. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning* 4, 2 (2012), 26–31.

[38] Bin Xia, Junjie Yin, Jian Xu, and Yun Li. 2019. LogGAN: A Sequence-Based Generative Adversarial Network for Anomaly Detection Based on System Logs. In *International Conference on Science of Cyber Security*. Springer, 61–76.

[39] Yi Xie, Cong Shi, Zhuohang Li, Jian Liu, Yingying Chen, and Bo Yuan. 2020. Real-time, Universal, and Robust Adversarial Attacks Against Speaker Recognition Systems. *arXiv preprint arXiv:2003.02301* (2020).

[40] Han Xu, Yao Ma, Haochen Liu, Debayan Deb, Hui Liu, Jiliang Tang, and Anil Jain. 2019. Adversarial attacks and defenses in images, graphs and text: A review. *arXiv preprint arXiv:1909.08072* (2019).

[41] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. 2009. Online system problem detection by mining patterns of console logs. In *2009 Ninth IEEE International Conference on Data Mining*. IEEE, 588–597.

[42] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 117–132.

[43] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. 2019. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems* 30, 9 (2019), 2805–2824.

[44] Yali Yuan, Sripriya Srikant Adhatarao, Mingkai Lin, Yachao Yuan, Zheli Liu, and Xiaoming Fu. 2020. ADA: Adaptive Deep Log Anomaly Detector. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2449–2458.

[45] Wei Emma Zhang, Quan Z Sheng, and Ahoud Abdulrahmn F Alhazmi. 2019. Generating textual adversarial examples for deep learning models: A survey. *arXiv preprint arXiv:1901.06796* (2019).

[46] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 2019. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 807–817.