DeepAG: Attack Graph Construction and Threats Prediction with Bi-directional Deep Learning

Teng Li[*], **Ya Jiang**[¶], **Chi Lin**[‡], **Mohammad S. Obaidat**[†], **Fellow of IEEE and Fellow of SCS**, **Yulong Shen**[¶], and **Jianfeng Ma**[*]
[*]School of Cyber Engineering, Xidian University, Shaanxi, China
[†]College of Computing and Informatics, University of Sharjah, UAE,
University of Jordan, Jordan, and University of Science and Technology Beijing, China.
[‡]School of Software Technology, Dalian University of Technology, Dalian, China
[¶]School of Computer Science and Technology, Xidian University, Shaanxi, China
Email: litengxidian@gmail.com

Abstract—The complicated multi-step attacks, such as Advanced Persistent Threats (APTs), have brought considerable threats to cybersecurity because they are naturally varied and complex. Therefore, studying the strategies of adversaries and making predictions are still significant challenges for attack prevention. To address these problems, we propose `DeepAG`, a framework utilizing system logs to detect threats and predict the attack paths. `DeepAG` leverages transformer models to novelly detect APT attack sequences by modeling semantic information of system logs. On the other hand, `DeepAG` utilizes Long Short-Term Memory (LSTM) network to propose bi-directional prediction for attack paths, which achieves higher performance than traditional BiLSTM. In addition, with previously detected attack sequences and predicted paths, `DeepAG` constructs the attack graphs that attackers may follow to compromise the network. Furthermore, `DeepAG` offers the mechanisms of Out-Of-Vocabulary (OOV) word processor and online update respectively to adapt new attack patterns that show up during detection and prediction stages. The experiments on open-source data sets show that more than 99% of over 15000 sequences can be detected accurately by `DeepAG`. Moreover, `DeepAG` can improve the baseline by 11.166% of accuracy in terms of prediction.

Index Terms—Attack prediction; Deep learning; Transformer; LSTM; Attack graph

1 Introduction

Unlike traditional attacks, multiple-step attacks, such as advanced persistent threats (APTs) [1], perform several intrusion steps to reach their specific objectives. Such attacks are usually equipped with strong abilities to covert, and generally bypass traditional detection tools which rely on signature-based detection [2] [3]. Besides, APTs are not the one-hit attack, which means the intruders tend to access the target systems for many times. Therefore, once a network is infiltrated, the intruders will remain in the system to attain as much information as possible. In many APT case studies, these actions can last several months or even years as intruders repeat the exfiltration process for many times. Moreover, attackers also frequently use novel techniques to obfuscate their actions. However, many traditional systems for threat prevention [4] [5] like Intrusion Detection Systems (IDS), Intrusion Detection and Prevention Systems (IDPS), Advanced Security Appliances (ASA), are no longer effective because they fail to exploit cyber-threats tactics, or produce high false alarm rates. Therefore, an approach capable of making timely, concrete, and robust detections and predictions for attacks is urgently needed.

Detecting and predicting the actions of attackers qualify as a big data problem, because of the fast booming number of attacks [6]. In that case, machine learning has been widely studied in cybersecurity [7]. Some researchers [8] [9] applied Bayesian Network to reveal the zero-day attack paths or perform the causal analysis. However, it requires the posterior probabilities of attacks occurring at each node, which is not easy to tackle as the prior knowledge for unexpected and sophisticated strategies taken by adversaries is difficult to obtain. $Okutan\ et\ al.$ [10] clustered security domains and achieved high-level accuracy in detecting attacks. Nonetheless, both Bayesian network and clustering are generally offline approaches, which mean that they cannot reflect the real-time behavior of network intrusion. Therefore, researchers can only detect the attacks after happening, making sufficient and effective prevention less possible. Other researchers [11] [12] presented methods based on Hidden Markov Models (HMM) [13] to predict multiple-step attacks, like DDoS attacks. Although HMM is widely used for detection of multi-stage attacks, existing approaches address only a single multi-stage attack instead of considering the problem of interleaving multi-stage attacks, which may compromise the detection performance of such attacks.

The recent breakthrough application of deep learning to machine translation has demonstrated the great potential of neural models' capability of understanding natural languages [14] [15]. From that perspective, a majority of work like Log2Vec [16], LogRobust [17], and LogAnomaly [18] in recent years utilized semantic vector sequences of logs

to detect attacks based on the deep learning framework, and thus fixing semantic gaps. Nevertheless, they can only make a binary prediction of whether the sequence includes attack, instead of where the anomaly points are in the sequences. Besides, Log2Vec, LogAnomaly, and Deeplog [19] all leverage a single LSTM for predictions. However, due to the bias of one-way model that may result in insufficient learning, it cannot perform very well in matching the top 1 prediction with the label, because its last layer (SoftMax layer) will assign negligible probabilities to many other predictions that may have influence on the probabiilities of top predictions.

Many approaches like CloudSeer [20] and Deeplog constructed workflows or attack graphs based on log analysis that is available and valuable for recording system states at various points. However, CloudSeer [20] is limited to single task execution, regardless of the complex relationships among networks. Although Deeplog [19] handled log messages produced by several different threads or concurrently running tasks, we note that it is not suitable when processing more sophisticated and non-linear relationships like multiple branches caused by sophisticated strategies of attackers. In order to construct the attack graph, other approaches like HinDom [21] and NetCycle+ [22] used IP address, domain name, and malware as different node infrastructures, and capture their relationships by defining the meta paths or meta graphs. However, feature engineering in these works is significant but complicated and difficult, as it requires a large amount of domain-specific knowledge to define great rules for the meta paths and meta graphs.

To sum up, we face three key challenges: (1) how to simultaneously detect attacks and locate the attack points based on logs; (2) how to overcome the challenge of insufficient learning caused by bias of one-way model; (3) how to model non-linear dependencies and construct attack graphs to help users master the strategies of intruders.

To cope with the above challenges, we propose `DeepAG`, an online approach capable of simultaneously detecting APT sequences and locating attack phases in the sequences respectively utilizing the log semantic vectors and indexes, and constructing attack graphs according to aforementioned log indexes. When attack sequences are detected, `DeepAG` can locate the abnormal points of the sequences. First, we extract the lexical and semantic information [16] of logs and vectorize them in order to reduce losses of log information. In particular, the log sequence we utilize is comprised of several continuous log sentences, which can help find the abnormal user behavior sequence as well as show the abnormal points. To detect attacks, we then leverage the transformer model [23], which deals with the high-dimensional semantic vectors in parallel and helps reducing the running time. Besides, we propose the bi-directional model to learn the relationships of locations among log index sequences. Based on forward and backward LSTMs, it can generate multiple sequences to provide more information for reliable predictions, unlike traditional BiLSTM [17] which makes single-time-step predictions for the same sequence. Besides, we introduce the mechanisms of OOV word processor and online update to overcome insufficient learning of detection and prediction models respectively. In the end, `DeepAG` constructs the attack graph which models the non-linear dependencies and intuitively demonstrates the attack phases to help users master strategies of adversaries. We evaluate `DeepAG` on the open-source data sets of four different system logs: HDFS, OpenStack, PageRank, and BGL logs. `DeepAG` can efficiently achieve real-time attack detection, which reduces time costs by over 3 times compared to its baselines.

To the best of our knowledge, we are the first to design a framework to simultaneously make the log-entry-level detections for attacks and locate the definite anomaly points in the sequences. We also propose a novel approach to model non-linear dependencies among system logs. In summary, the contributions of this paper can be summarized as follows:

1. We leverage the transformer models to novelly represent log sequences with vectors, which reduce the loss of semantic information and can process log vectors parallelly for attack detection, only costing less than 40% time of that of other state-of-the-art methods.

2. We are the first to propose the bi-directional model including forward and backward LSTM for locating anomaly points over system logs. In that case, it avoids the bias of a single LSTM model, and thus improves the performance, especially *recall* when confirming attack points in the logs.

3. `DeepAG` can model non-linear dependencies among attack sequences through the attack graphs and mark the anomaly points, thus showing the attack paths intuitively for users and helping them make proactive prevention.

4. Experiments on open-source data sets show that `DeepAG` exhibits low runtime overheads (less than 1 second), reducing time by over 3 times compared to four baselines in terms of attack detection. Besides, `DeepAG` can successfully make detections among almost 100% sequences. It also improves accuracy by over 10% when locating the attack points.

TABLE 1: Comparison of approaches

Approaches	[19]	[16]	[18]	[17]	`DeepAG`
Attack prediction	✓	✗	✗	✗	✓
Feedback mechanism	✓	✓	✓	✗	✓
Dependencies modeling	✓	✗	✗	✗	✓
Semantic gap troubleshooting	✗	✓	✓	✓	✓

A comparison of `DeepAG` and prior work has been outlined in table 1. Comparatively, the proposed `DeepAG` can not only predict binary attacks, but also predict concrete attack paths. Moreover, `DeepAG` incorporates unexpected samples through feedback mechanisms. It also models non-linear dependencies through attack graphs according to the attack sequences which is easier for users to analyze the strategies of intruders. Moreover, `DeepAG` reduces the semantic gaps. Comparing with other three semantic-based approaches [16] [17] [18], `DeepAG` performs much better when tested on two different logs with only one model trained on either one type of logs.

The rest of paper is organized as follows. Section II introduces the threat model, our motivation and goals. Section III introduces the system architecture in detail.

Section IV compares DeepAG with its baselines in terms of theoretical analysis. Section V describes the experiments and performance of DeepAG by comparison with the state-of-the-art baselines. Section VI introduces the related work. Section VII discusses challenges of practically deploying deep learning models of cybersecurity. Section VIII summarizes the paper and outlines future work.

2 Threat Model

When hackers attack a computer, they will leave traces of modifications, which will be recorded in the system logs. The IIS (Internet Information Server) log file deployed on the Windows Server has the function of event logging: who visit the site, what content the visitors view, and so on. By checking log files regularly, webmasters can detect which aspects of the servers or sites are vulnerable or facing other security risks, and thus analyze the hackers' strategies and entire attack timeline from logs of the following sources: attacked server and the user operation history.

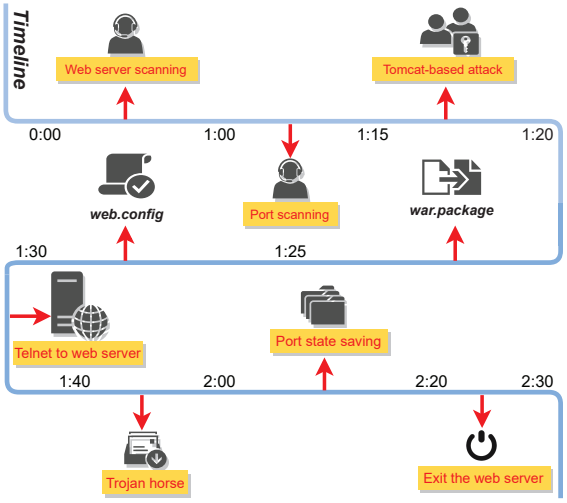


Fig. 1: Infiltration attack shown through the timeline

Fig. 1 depicts nine key phases of a complete infiltration attack through the timeline from $0:00$ to $2:30$. At the beginning, the hacker attacks the external web server and obtains the authority, which is the springboard to access other internal web servers. Then, to collect the information of the target system, the attacker identifies the surviving host IP in the intranet, running port and vulnerability scanning to obtain exploitable vulnerabilities. When the attacker discovers that Tomcat was listening on port 8080, it will greatly facilitate their intranet infiltrations, for the reason that Tomcat tends to be started with the command $nt\ authority\ system$ on the Windows host which will easily cause Tomcat weak password attacks, and thus help attackers easily obtain the password and hash value to fully control the server. Next, attackers exploit file upload vulnerability to upload the war package, which contains all the malicious files they are going to utilize. Then they check the $web.config$ file, which configures all the necessary items to run the web program. Afterward, attackers upload the $explurer.exe$ program, which includes attacks utilizing system vulnerabilities, and then perform local overflow privilege escalation. Next, the hackers remotely log in to the web server, download the Trojan on the remote server, and run it locally. At last, they scan the designated port on the internal network, save the internal network port situation, and exit the server.

Basically, this timeline is similar to a sequence, and they can be shown through analyzing the relationship between system logs and user operation history recorded in the host history file. In this "sequence", some points (i.e. trojan horse) can be anomalous. In that case, we can detect that attack as soon as we find the abnormal points.

However, the well-designed attacks present two key challenges to modern cybersecurity:

More vulnerabilities. The increasingly sophisticated works have brought about more bugs and vulnerabilities, making it hard to maintain the system and detect attacks. For example, office vulnerabilities are still the favorite vulnerabilities of most APT organizations, such as Microsoft Office remote memory corruption vulnerability and Office document attacks that exploit browser zero-day vulnerabilities. That is because office computers are used a lot, which are the best extranet entries and have the most direct effects.

Long duration. APT attacks have strong continuity. Fig. 1 is just the small fraction of the entire compaign. After long-term preparation and planning, attackers usually lurk in the target network for months or even years. Through repeated infiltrations, the attack methods and paths are continuously improved and launched.

Nonetheless, as the timeline shows, many attacks actually have their potential patterns though seem significantly sophisticated. Therefore, their routines can be many sequences through the timeline. In other words, by learning and modeling these "fixed paths" of malicious programs involved in this campaign as much as possible, we can detect the attack sequences and master attack phases of intruders. In the next section, we will discuss our approach based on the previous problems. Besides, we formulate the design goals as follows. Note that we assume DeepAG itself is not attacked, and the data is not tampered by malicious user to confuse the model.

Accurate prediction. To guarantee the effectiveness of the prevention, DeepAG should detect the threats with high accuracy and in a real-time manner.

Adapting new patterns. To overcome the insufficient learning, DeepAG should learn new patterns and achieve good performance when processing unexpected logs.

Graph construction. A single-step prediction is not enough to capture the relationships of attack phases. To obtain more strategies for proactive prevention intuitively, DeepAG should visualize the dependencies into a graph.

3 System Model

Though the multi-step attacks are highly stealthy and complicated, we still find them through analyzing system logs. Inspired by that, we study attack phases from system logs and extract the log templates. Fig. 2 illustrates a high-level overview of DeepAG, which is divided into five parts: text representation, training stage, detection stage, prediction stage, and graph construction. First, to represent the logs, we extract the log templates and then convert them respectively into indexes and vectors. In the training stage, we vectorize the logs to obtain several log vector sequences and input these sequences into the transformer to train

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3143551, IEEE Transactions on Dependable and Secure Computing

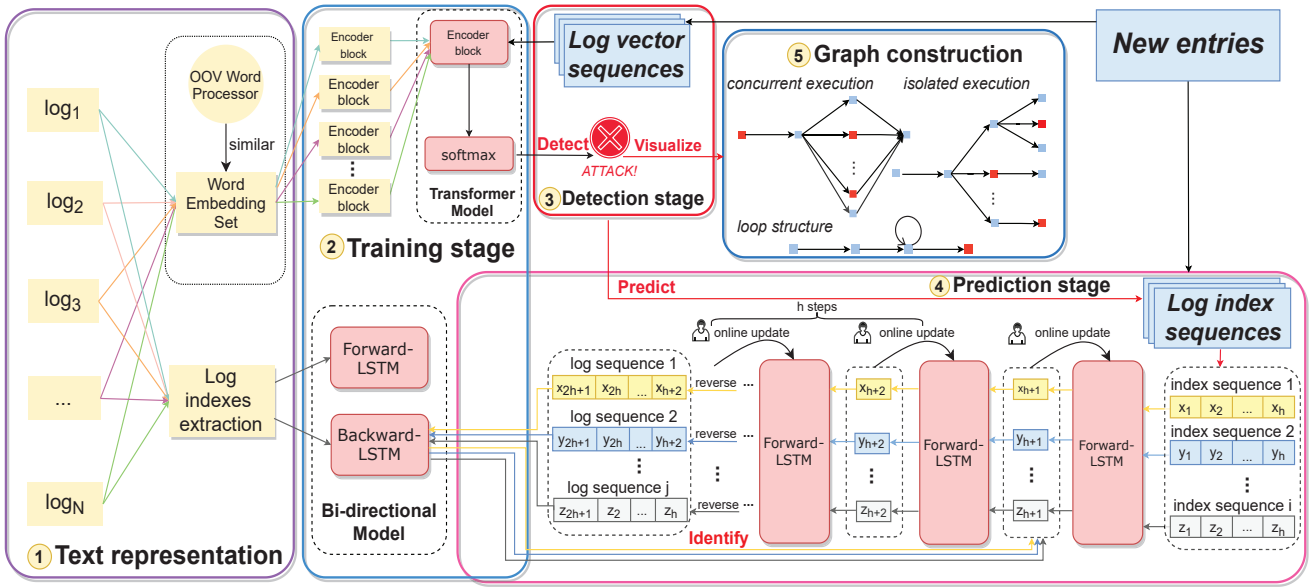


Fig. 2: An overview of DeepAG: **(1) Text representation:** convert logs to log vectors and log indexes respectively; **(2) Training stage:** train the transformer model and bi-directional model utilizing log vector sequences and log index sequences respectively; **(3) Detection stage:** leverage the transformer model to detect if there are attacks; **(4) Prediction stage:** utilize bi-directional model to predict the paths if there are attacks; **(5) Graph construction:** visualize the attack paths recorded in the logs using log indexes sequences

them for APT sequence detection. Moreover, we feed the bi-directional model with a history of recent index sequences and regard the next index following the history as the output. The detection and prediction stages aim to judge the APT sequences and get predicted attack phases and their probability distributions through the bi-directional model. Upon constructing an attack graph, DeepAG offers an approach related to conditional probabilities for the graph generation.

3.1 Preliminaries

Here, we explain the preliminaries and concepts used in DeepAG.

System logs. System logs are the valuable resource and tool to analyze the attack strategies of intruders as they intuitively reflect the activities of systems. They actually encode execution paths ordered by timestamps, which help find a suspicious activity before a major incident occurs.

Log sequence. We utilize the log sequence which is comprised of several continuous log sentences to make the detection and prediction. That is because sequence can help users analyze the correlations of behaviors of intruders, especially when the targets of intruders are not one-hit, which means they tend to spy in the systems to steal the key information and leave anomaly records. Therefore, it can help find the abnormal user behavior sequences as well as show the abnormal points more intuitively.

Transformer model. Transformer model is the state-of-the-art technology in the fields of NLP, utilizing hierarchical encoders and decoders with attention-based structure. Therefore, it models global dependencies between input and output instead of only depending on sequential relationships. Moreover, it highly facilitates parallel input processing, which can be well applied for parallel computing of GPU and thus improve time efficiency. In that case, It can help DeepAG greatly reduce the running time of dealing with the high-dimensional semantic vectors.

LSTM model. LSTM model is the recurrent neural network capable of learning long-term dependencies among sequences. It has achieved success in various tasks such as machine translation, sentiment analysis, and medical self-diagnosis, as it can learn the intricate patterns and be equipped with the sequential nature. The recorded events in the system log tend to be sequential, therefore, we employ the LSTM model to learn the temporal relationships of log sequences and make predictions.

Bi-directional prediction. In the prediction of every step, there may be multiple results with different probabilities caused by possible forks (i.e. concurrency) or insufficient learning. In order to enhance the reliability of every prediction, we propose the bi-directional model including two LSTMs to validate events from both directions. They are respectively trained by forward logs that contain log sequences in the sequential execution and backward logs of reversed log sequences. In the end, for every new entry, after getting the predictions for next time step t from forward LSTM, we further make h-steps predictions based on forward LSTM and obtain several sequences with length of h. Then we reverse these sequences and input them to backward LSTM to get the backward predictions for time step t. Finally, we integrate the predictions of time step t from two directions to make reliable and comprehensive predictions.

Concurrent events. The orders among log sequences provide significant information for the relationships of different attacks, and there might be the log messages with interactive relationships. Concurrent events are several different threads or concurrently running tasks.

Attack graph. Attack graph is an intuitive framework to demonstrate the possible exploited vulnerabilities and attack paths for analysts. It can model the non-linear dependencies including the isolated execution, fork caused by concurrency, and loop structures among system logs.

3.2 Text representation.

As shown in fig. 2, we first convert logs to indexes and vectors. As for log indexes, some logs include an "identifier field", which is the numerical form and represent a certain event. For example, HDFS log and OpenStack log can be grouped into different sessions by the block_id and instance_id respectively. Therefore, we extarct them as log indexes through log indexes extraction in fig. 2. For example, as fig. 3 shows, the log sequence $[log_1, log_2, ..., log_h]$ can be extracted as index sequence $[x_1, x_2, ..., x_h]$; and the log sequence $[log_{h+1}, log_{h+2}, ..., log_{2h}]$ can be extracted as index sequence $[y_1, y_2, ..., y_h]$ and so on and forth.

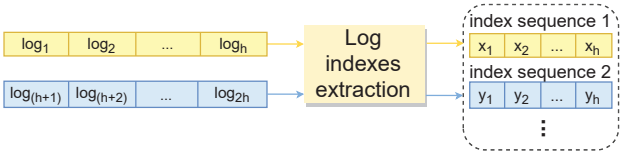


Fig. 3: Extracting log indexes

To vectorize the log sentences, we construct the log-specific word embedding set through the information extracted from logs, and deal with OOV words as well. During the experiments, we extract synonyms, antonyms, and relation triples. Synonyms and antonyms represent lexical information, while relation triples represent relationship information. With the set of vector representation of synonyms and antonyms corresponding to the target word, we define the word vector according to the distance between the tatget word and these synonyms and antonyms. In particular, it should be as close as possible to its synonyms and as far away as possible from its antonyms. Inspired by FastText [24] which points out that vector of a word are often similar to the average vectors of its surrounding words, theoretically, if we have one or more examples of surrounding contexts for the OOV words, then it is conceivable to infer a vector for the target word. Therefore, we utilize surrounding words to predict the target word. Specifically, we apply relation triples (h, r, w_i), in which h is a certain surrounding word, w_i the target words, and r different association relationships with w_i. If the triples are factual information, then $(h + r \approx w_i)$, meaning the corresponding vector of $h + r$ is closer to w_i. Inspired by Log2Vec [16], the objective function is presented as follows, which combines lexical word embeddings and semantic word embeddings. C refers to the corpus used for training, while α and β are constants.

$$V = \sum_{n=1}^{|C|} \lg p(w_i|w_{i-c}^{i+c}) + \alpha \sum_{r \in R_{w_i}} \lg p(w_i|h+r) + \beta\Big(\sum_{u \in SYN_{w_i}} \lg p(w_i|u) - \sum_{u \in ANT_{w_i}} \lg p(w_i|u)\Big) \quad (1)$$

In the end, we represent the words in the log as embeddings, and input the log to the transformer encoder block. Thus, it can output the vector representation of the log. For example, as fig. 4 shows, after removing variables of log_1, we convert every word in log_1, such as $INFO$, $AsyncDispatcher$, and so on, to embeddings according to word embedding set. At last, we input these embeddings to transformer encoder block and obtain the k-dimensional vector representation of log_1. To detect the attacks in the sequence, we input the vector representations of sequence $[log_1, log_2, ..., log_h]$ to transformer model and get its final representation for training and detection.

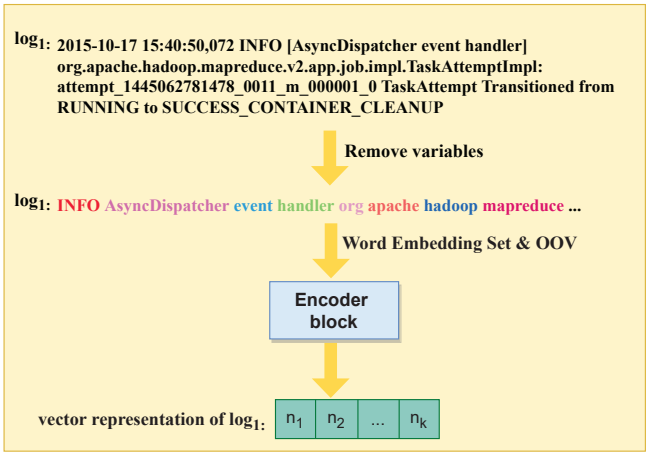


Fig. 4: Removing variables of logs

3.3 Training stage

To train the transformer model used to detect APT sequences, we process the vector sequences of continuous log sentences with encoders, and thus get log sequence representations that are used for binary classification. On the other hand, to train the bi-directional model used to predict attack phases, we train two LSTMs with the index sequences. One is the forward LSTM trained in the sequential order, and the other is the backward LSTM trained in the reversal order. In the end, we can obtain two independent LSTM models. Next, we introduce transformer and bi-directional model respectively.

Transformer model. Transformer model [23] is the state of art in Natural Language Processing (NLP) and can process vectors in parallel, thus making full use of GPU resources and reducing the running time of dealing with the high-dimensional data like semantic vectors. It mainly contains encoders and decoders. The encoders distributedly represent the word vectors, while the decoders aim at decoding the vector representations into sentences. Based on our scenario that extracts the final vector representation of a sequence, `DeepAG` only utilizes the transformer encoding structure to encode log sequences for attack detections. Next, we introduce the structure of a transformer encoder block and demonstrate its design applied in `DeepAG`.

As is shown in Fig. 5, the transformer encoder block is comprised of multi-head attention layer and Feed Forward Network (FFN). First, we input the vector matrix X of a sequence into the block, where $X \in R^{n*d}$, and n is the sequence length and d is the dimension of every log vector. In particular, our transformer uses twelve attention heads,

and we denote the number of attention heads as h. To calculate the attention of every vector X_i ($i \in \{1, 2, 3, ..., n\}$) in the sequence, we first create three vectors (query vector Q, key vector K, and value vector V) for each vector X_i, and will use them for h times respectively. We use $W_Q \in R^{d*d_q}$, $W_K \in R^{d*d_k}$, $W_V \in R^{d*d_v}$ to represent parameter matrices, and d_q, d_k, d_v are the dimensions of query, key, and value vectors. These vectors are computed by multiplying X by three matrices as follows.

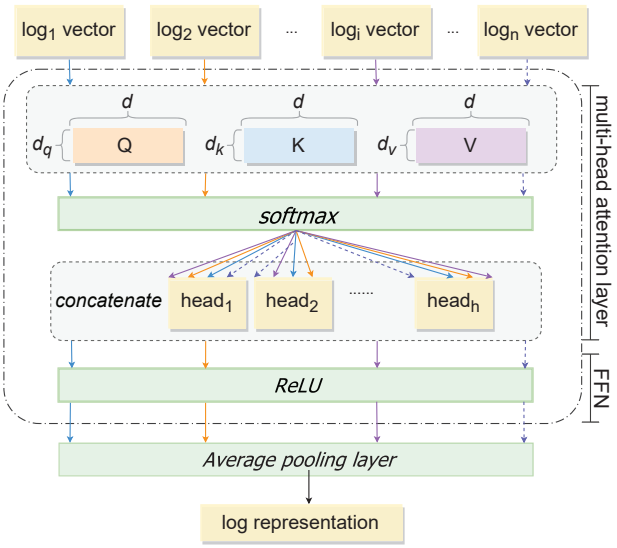


Fig. 5: Design of transformer encoder block

$$Q = X * W_Q$$
$$K = X * W_K \quad (2)$$
$$V = X * W_V$$

For vector X_i ($i \in \{1, 2, 3, ..., n\}$), we then use X_j ($j \neq i$) in the sequence to score X_i through the dot product of K and Q, which determines the degree of importance of X_i. Moreover, we represent the score with Z_i as follows.

$$Z_i = Q_i * K^T \quad (3)$$

To stabilize the gradient, we divide Z_i by $\sqrt{d_k}$ and normalize all scores with $softmax$ function. Due to the multi-headed mechanism, we have multiple sets of the query, key, and value weight matrices. For h_{th} attention head, we multiply z_i^h by $V_h.$, so we concatenate these heads and multiply them with W_O as the output of the multi-head attention layer, where W_O is a trainable parameter for concatenation operation. $head_h$ can be obtained as follows, where Q_{ih} represents i_{th} for h_{th} Q, and K_h^T the h_{th} K.

$$head_h = softmax(\frac{Q_{ih} * K_h^T}{\sqrt{d_k}}) * V_h \quad (4)$$

After multi-head attention layer, we pass the output through the feed-forward neural network, which is a combination of two linear layers using the ReLU activation function. This network is shared at different time steps of each layer but is independent at different layers. The dimension of its hidden layers is d_{ff}, while the input and output are of d dimension. The following is the function and W_1, W_2, b_1, b_2 are trainable parameters in the feed-forward network layer.

$$F_i = max(0, z_i W_1 + b_1) W_2 + b_2 \quad (5)$$

Therefore, the log sequence representations are finally output after the transformer encoder blocks. For example, for a sequence comprised of log sentences $\{l_1, l_2, ..., l_m\}$, we obtain the sequence of log sentence representations $R_L = \{R_{l1}, R_{l2}, ..., R_{lm}\}$, where R_{lm} is the m-th log sentence representation. In the end, with the average pooling layer, we take the average of R_L as final log sequence representation for classification.

Bi-directional LSTM model. LSTM is an effective RNN capable of learning long-term dependencies [25]. Specifically, designed to overcome the vanishing gradient problem, it maintains a constant error, and thus can continue learning over numerous time steps and backpropagate through time and layers. LSTM has been proven a practical and accurate model in various cases when dealing with classification [26]. The recorded events in the system log tend to be sequential, therefore, we employ LSTM for feature learning and the attack events predicting. Fig. 6 demonstrates the structure of an LSTM block. We take the first layer as an example, and the following layers are similar.

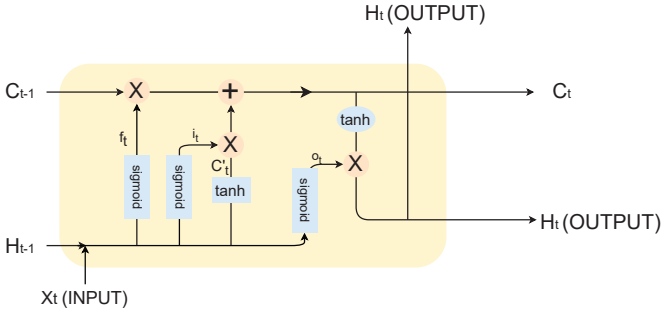


Fig. 6: The first layer of Single LSTM Block

The block processes the input one by one. It passes the previous hidden state $H_i^k (i = t-h, t-(h-1), \cdots, t-1, k = 1, 2, \cdots, n)$ to the next step of the sequence. The hidden state acts as the memory of the neural network, holding information of previous data that the network has seen before. Information is added or removed to the cell state $C_i^k (i = t-h, t-(h-1), \cdots, t-1, k = 1, 2, \cdots, n)$ that transfers relative information down the sequence chain via gates. The gates are different neural networks learning what information will be kept or forgotten on the cell state during training. There are three types of gates: input gate, forget gate, and output gate. In short, input gate determines what information is relevant to add from the current step, and forget gate decides what is relevant to keep from prior steps. Besides, the output gate determines what next hidden state should be. The recurrent memory arrays for the three gates are as follows.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3143551, IEEE Transactions on Dependable and Secure Computing

$$
\begin{aligned}
f_t &= \sigma(W_f * [H_{t-1}, x_t] + b_f) \\
i_t &= \sigma(W_i * [H_{t-1}, x_t] + b_i) \\
C'_t &= tanh(W_C * [H_{t-1}, x_t] + b_C) \\
C_t &= f_t * C_{t-1} + i_t * C'_t \\
o_t &= \sigma(W_o * [H_{t-1}, x_t] + b_o) \\
H_t &= o_t * tanh(C_t)
\end{aligned} \tag{6}
$$

Stacking LSTM blocks will create a multi-layer and feed-forward network at each time step, which means the input to a layer is the output of the previous layer. Therefore, stacking mechanisms automatically create different time scales at different temporal hierarchy, greatly helping learn the temporal order of system logs. As demonstrated in fig. 7, DeepAG stacks blocks and applies the hidden layer, followed by a softmax which leverages a standard multinomial logistic function to output the probability distribution of predicted indexes. For simplicity, we omit an input layer and an output layer constructed by standard encoding-decoding schemes. In this paper, for example, we use structured data (i.e. log indexes) extracted from unstructured log entries to train the LSTM models. Given a sequence of log indexes, the LSTM model is trained to output the probability distributions P=$\{x_t = k \mid x_{t-h}, x_{t-(h-1)}, ..., x_{t-1}\}$. Here, h refers to the length of input sequence and k the next log indexes.

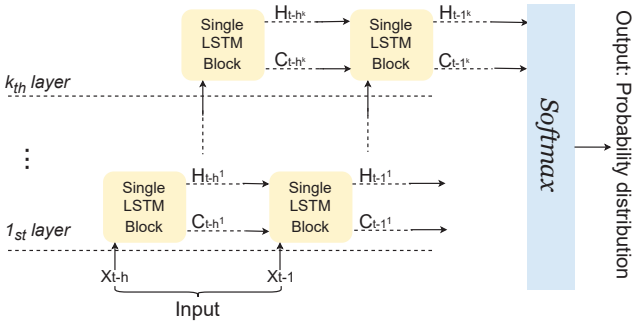


Fig. 7: LSTM model of DeepAG

The word order is a critical factor for modeling sentences because it determines the logic of sentences, providing useful information. However, single LSTM is uni-directional, which can only encode the sentence from front to back or from back to front. From that perspective, BiLSTM is proposed to learn from two directions and better capture the two-way information dependence. BiLSTM consists of two LSTMs, which learn sentence information from front to back and from back to front respectively. It performs single-step prediction on the same input sequence and concatenates the output of last hidden layers of two LSTMs for classification. Taking the log-based attack prediction studied in this paper as an example, fig. 8 shows the application of BiLSTM in this scenario. Assuming that the log sequence is $\{x_1, x_2, x_3\}$, the forward LSTM sequentially inputs x_1, x_2, x_3 to obtain three vectors $\{h_{f0}, h_{f1}, h_{f2}\}$. The backward LSTM sequentially inputs x_3, x_2, x_1 to get three vectors $\{h_{b0}, h_{b1}, h_{b2}\}$. It then concatenates the forward and backward hidden vectors to obtain $[h_{f2}, h_{b2}]$, which contains all the forward and backward information. Finally, It sends the spliced vector to softmax layer for prediction.

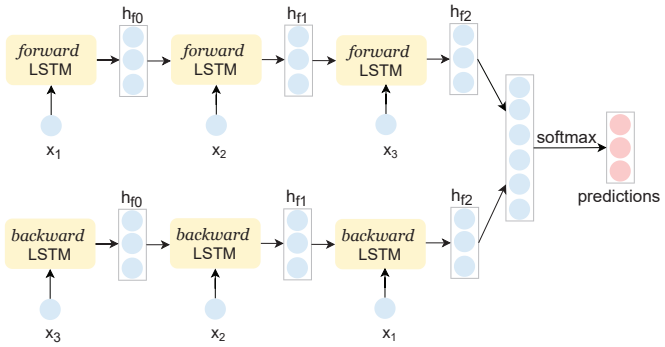


Fig. 8: Application of BiLSTM

However, traditional BiLSTM makes single-time-step predictions for the same input sequence which provides limited information. To collect more information for more reliable predictions, we propose bi-directional model of DeepAG. It generates multiple sequences to provide more information for reliable predictions based on forward and backward LSTMs. Our design is shown in fig. 9. For each log sequence with a length of h in the training set, we train the forward LSTM and backward LSTM respectively with forward and backward training data sets.

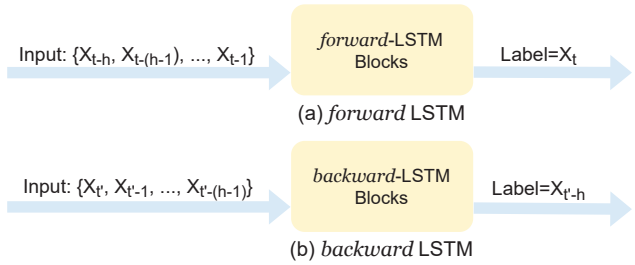


Fig. 9: Bi-directional training models

For example, supposing a small fraction of our data set for forward LSTM is:

$$\{x_1, x_6, x_3, x_5, x_{10}, x_{19}, x_{15}, x_2, x_{26}, x_1, x_5, x_8\} \tag{7}$$

Given a window size h = 10, the input sequence and output label for forward LSTM would be:

$$
\begin{aligned}
&\{x_1, x_6, x_3, x_5, x_{10}, x_{19}, x_{15}, x_2, x_{26}, x_1 \rightarrow x_5\} \\
&\{x_6, x_3, x_5, x_{10}, x_{19}, x_{15}, x_2, x_{26}, x_1, x_5 \rightarrow x_8\}
\end{aligned} \tag{8}
$$

The process for backward LSTM data set is similar.

3.4 Detection stage.

After vectorizing every word with pre-trained word embedding set, we encode every log sentence through the encoder blocks and obtain the vector representation. As fig. 10 shows, vector representations of multiple continuous log sentences (i.e. $[log'_1, log'_2, ..., log'_h]$) form the log vector sequence. Next, we input the log vector sequence into transformer model and get final log representation for that sequence, in order to judge if there are included attacks. In

case of unseen words in the log sentences, DeepAG gets its vectors according to the OOV model. With the strategies of online update and trained OOV processor in the offline stage with MIMICK [27], DeepAG assigns a new embedding vector for unseen words when a new word appears in the online stage, and thus adapts new patterns of capricious intrusion.

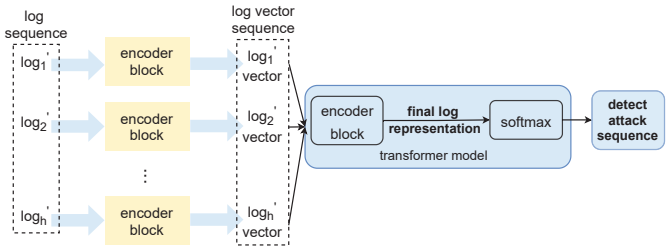


Fig. 10: Detection framework

3.5 Prediction stage.

DeepAG is able to make online predictions utilizing bi-directional model. We first feed the forward LSTM with a new log index sequence. After getting the predicted log indexes and probability distributions, we regard these indexes as the starts of multiple branches. For every one of them, we then make the further predictions for h steps and store every sequence as a branch of the original one. Then we reverse every branch and input them to the backward LSTM and compute the probability of predicted indexes. Finally, our strategy is to sort the average probabilities of predicted indexes that appear in the predictions of forward or backward models and have the probabilities greater than g, where g is threshold for outputting top few predictions that have higher probabilities. We regard the probabilities of indexes that are in the predictions of only one model as 0. After getting the set of predicted indexes intergrated from two LSTMs (i.e. forward and backward LSTMs), DeepAG verifies whether the actual label appears in the set. If not, the prediction will be regarded as wrong. Besides, DeepAG can adapt new patterns with the report from analysts for the wrong prediction.

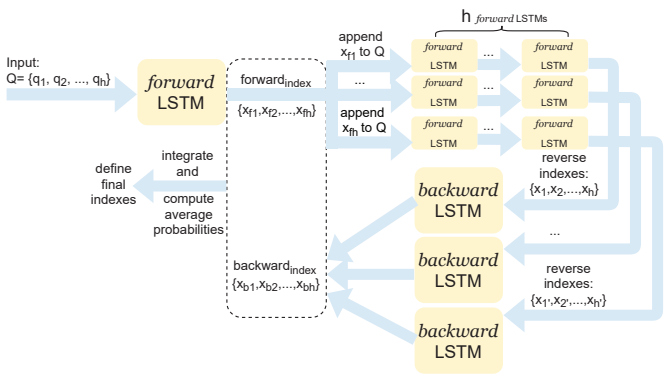


Fig. 11: Process of bi-directional prediction

As demonstrated in fig. 11, we first input a new log index sequence Q = $\{q_1,...,q_h\}$ to the forward LSTM and output the predicted indexes and their probability distributions, where h is the window size of LSTM. We observe that most predicted indexes have an extremely low probability that can even be negligible, for the reason that the attack patterns are finite and what the softmax layer computes is the probability distribution among all indexes.

By investigating the probability of predicted indexes during our experiments, we obtain the following findings: (1) The probabilities of most indexes in a prediction are extremely small, most of which have the magnitudes of only 10^{-6} or even smaller. (2) For our data set which contains 29 classes of attack phases, the number of predicted indexes with the probability greater than 0.1 is usually within 4. (3) If the prediction includes only one index with the probability greater than 0.1, its probability is generally greater than 0.9. (4) If there are more than 3 indexes having the probabilities greater than 0.1 in a prediction, then their probabilities are often from around 0.3 to 0.5.

Therefore, we set up the threshold for prediction as 0.1, in order to avoid the noise of other predicted indexes with negligible probabilities, and thus reduce the training burden. We then obtain the $forward_{indexes}$=$\{x_{f1}, x_{f2}, ..., x_{fn}\}$, which includes the indexes that are predicted from Q and have probabilities greater than g and wait for being verified combining backward LSTM. Their probability distribution in this step predicted by forward LSTM is $forward_{probs}$=$\{p_{f1}, p_{f2}, ..., p_{fn}\}$. Then we append every index in the $forward_{indexes}$ to Q and make a further prediction with the forward LSTM for h steps.

From fig. 11, after h-step prediction of forward LSTM, we obtain the log index sequences $reverse_{indexes} = \{[x_1, x_2, ..., x_h], [x'_1, x'_2, ..., x'_h]\}$ and their probabilities $reverse_{probs} = \{[p_1, p_2, ..., p_h], [p'_1, p'_2, ..., p'_h]\}$ for backward LSTM. With these new index sequences $reverse_{index}$ of length h, we reverse each of them which is fed to backward LSTM model later. Then we get a set of predicted indexs $backward_{indexes}$=$\{x_{b1}, x_{b2}, ..., x_{bn}\}$ and their probabilities $backward_{probs}$=$\{p_{b1}, p_{b2}, ..., p_{bn}\}$. In the end, after integrating the $forward_{indexes}$ and $backward_{indexes}$, we compute the average probability avg of each index. The process of predicting next indexes $forward_{indexes}$ and probabilities $forward_{probs}$, and obtaining final indexes integrated from forward and backward LSTMs is respectively shown in algorithm 1 and algorithm 2.

3.6 Graph construction.

In order to present the possible paths of a attack, we need to perform a high-level abstraction. Inspired by Deeplog [19] which leverages LSTM to predict log key based on log sequence, and proposes the output actually encodes the underlying execution path, we build the attack graph through bi-directional model where the predictions are in the streaming form.

To work with a log sequence that contains multiple tasks or concurrent threads in one task, Deeplog proposes that the main challenge is to find a divergence point to figure out whether the multi-key prediction output is caused by either concurrency in the same task or the start of a different task. Deeplog observes that if the divergence point is caused by concurrency in the same task, a common pattern is that keys with the highest probabilities in the prediction output will appear one after another, and the certainty (measured by higher probabilities for a smaller number of keys) for the

Algorithm 1: Obtaining backward sequences

Input: Original index sequence Q, window size h, an empty list $reverse_{indexes}$ and an empty list $reverse_{probs}$
Output: A list $reverse$ including new index sequences $reverse_{indexes}$ and their probabilities $reverse_{probs}$

Predicting next indexes $forward_{indexes}$ and probabilities $forward_{probs}$;
for $index \in forward_{indexes}$ **do**
  Q' = deepcopy(Q);
  Q'.append($index$);
  $new\_sequence$.append(Q')
for $sequence \in new\_sequence$ **do**
  Further predicting h steps based on forward LSTM and getting several branches $branch_{index}$ and probabilities $branch_{prob}$;
  $reverse_{indexes}$.append($branch_{index}$);
  $reverse_{probs}$.append($branch_{prob}$);
$reverse$.append($reverse_{indexes}$, $reverse_{probs}$);
return $reverse$;

Algorithm 2: Obtaining final indexes integrated from forward and backward LSTMs

Input: Indexes predicted by forward LSTM $forward_{indexes}$, probability distribution $forward_{probs}$, a list of index sequences $reverse_{indexes}$ and a list of their probabilities $reverse_{probs}$, an empty list $reverse$, an empty set $final_{indexes}$, an empty dictionary of final indexes and their average probabilities $indexes_{avg}$.
Output: a dictionary of final indexes and their average probabilities $indexes_{avg}$

$final_{indexes}$.update($forward_{indexes}$) ;
$forward\_dict$ = dict((x, y) for x, y $\in$ zip($forward_{indexes}$, $forward_{probs}$);
$middle$ = [];
for $seq \in reverse_{index}$ **do**
  seq.reverse();
  Further predicting next indexes $backward_{indexes}$=$\{x_1,x_2,\cdots,x_n\}$ and their probabilities $backward_{probs}$ = $\{p_1,p_2,\cdots,p_n\}$ based on seq through backward LSTM;
  $final\_indexes$.update($backward_{indexes}$);
  $middle$.append(dict((x, y) for x, y $\in$ zip($backward_{indexes}$, $backward_{probs}$)));
Constructing a dictionary $backward\_dict$ that includes each index in $backward_{indexes}$ and their computed conditional probability based on $forward_{probs}$, $reverse_{prob}$ and $middle$;
for $index \in final_{indexes}$ **do**
  Computing its average probability avg based on $forward\_dict$ and $backward\_dict$;
  $indexes_{avg}$[$index$] = avg;
return $indexes_{avg}$;

following predictions will increase, as keys for some of the concurrent threads have already appeared. The prediction will eventually become certain after all keys from concurrent threads are included in the history sequence. Thus, it has a parameter g, denoting the number of top g log keys in the predicted output probability distribution function to be considered.

However, Deeplog sets g as the maximum number of branches at all divergence points from the workflows of all tasks, which is compromised by two shortages. On the one hand, it is difficult for us to know the maximum number of branches in advance, because the number of branches may change as the model runs on different sequences. For example, if this time we set g as known number of branches after running the model, then next time g may change due to the changing number of branches. Therefore, it is hard to determine an optimal g. On the other hand, in fact, it is not practical because the prediction output of Deeplog includes probabilities of all 29 classes, in which most keys account for probabilities that are negligible but not zero. From that perspective, Deeplog needs to set g as 29, and thus the noise caused by negligible keys could make judgment of concurrency and isolated tasks so difficult.

To overcome the difficulty to find an optimal g, we set the probability threshold for prediction output, effectively limiting the number of log keys to be considered. We also compare and analyze the impact of different values of probability threshold in Section V.

In the phase of judging concurrency, we utilize the bi-directional model which outputs probability distribution. LSTM tends to make increasingly accurate predictions with the augment of window size, while also brings about complicated computing with the increasing window size due to its intricate inner structure. Therefore, we should determine a proper window size, which can avoid resulting in the segments shared by multiple sequences. For example, the segment $\{x_7 \rightarrow x_3\}$ is shared by $\{x_8 \rightarrow x_7 \rightarrow x_3 \rightarrow x_6\}$ and $\{x_2 \rightarrow x_7 \rightarrow x_3 \rightarrow x_9\}$, may leading to the inaccurate prediction when the window size of LSTM is two.

We define the concurrent relationships among various indexes by their conditional probabilities. Essentially, for a branch point that the next prediction contains more than one event, we should judge whether some of them are concurrent executions or not. We assume that the number of indexes in a prediction is w ($w > 1$). As concurrency means that various tasks are executed by multiple threads, they can appear at least in different orders of factorial w. Besides, due to the linear input form of the LSTM model in our scenario, we should make further predictions of at least w steps for judging concurrency.

We have the following six findings for the judgment of concurrent index: (1) The concurrent indexes belonging to the same sequence must appear in every step of prediction respectively. For example, supposing there are four concurrent indexes, the kinds of orders in that sequence will be at least 24 ($factorial\ 4$). (2) When the predecessor of index is the same as itself, its probability in the prediction of that step should be regarded as 0 for computing conditional probability. (3) Its conditional probability computed in the next prediction should not be smaller than that in the current step. (4) Its predecessor in the previous prediction must be also judged as a concurrent index. (5) The

number of indexes filtered after (1), (2), (3), and (4) must be at least two. (6) These events conforming to the above five steps need converge to the same event. Therefore, if there are branches, we should make a further prediction from the branches respectively combining the original index sequence. We define $initial_{indexes}$ as a list of indexes appearing in both predictions and their probabilities are the lists of $initial_{probs}$. Next, we illustrate the process in detail, and the approach to judging concurrency is demonstrated as algorithm 3.

Algorithm 3: Concurrent indexes judgment

Input: Original index sequence Q
Output: A list of concurrent indexes $concurrent_{indexes}$

```
step = 0;
while initial_indexes.length > 1 and initial_indexes.length > step do
    Making one more prediction as next_indexes;
    for index ∈ next_indexes do
        if index ∉ initial_indexes then
            Remove(index) ;
            Remove(index.edges);
        else
            Calculating conditional probability prob_index;
            if prob_index < previous prob_index then
                initial_indexes.remove(index);
                step = 0;
                break;
            else
                step += 1;
concurrent_indexes = initial_indexes;
return concurrent_indexes;
```

Fig. 12 shows the graph constructed according to a complicated circumstance caused by concurrency, supposing the window size of bi-directional LSTM model is three. In fig. 12 (a), the predicted probability distribution $Prob_d$ of event sequence $\{x_{14} \rightarrow x_8 \rightarrow x_7\}$ is $\{x_{26} : 0.3, x_1 : 0.4, x_3 : 0.2, x_4 : 0.1\}$, which probably contains multiple indexes. Therefore, it is necessary to judge concurrency. We regard $Prob_d$ as the first prediction. As aforementioned, we need at least four further predictions. Therefore, we introduce the following procedures. At the beginning, we combine the original index sequence with the index of $Prob_d$ respectively, to construct the branches and make the second prediction shown as fig. 12 (b). At the second-prediction phase, the index sequence $\{x_8 \rightarrow x_7 \rightarrow x_{26}\}$ leads to the prediction $\{x_1 : 0.5, x_3 : 0.2, x_4 : 0.3\}$, and $\{x_8 \rightarrow x_7 \rightarrow x_1\}$ results in the prediction $\{x_3 : 0.4, x_{26} : 0.6\}$, and $\{x_8 \rightarrow x_7 \rightarrow x_3\}$ brings about the prediction $\{x_4 : 0.3, x_1 : 0.6, x_{26} : 0.1\}$, and the index sequence $\{x_8 \rightarrow x_7 \rightarrow x_4\}$ causes the prediction $\{x_{26} : 0.9, x_5 : 0.1\}$. Noting that the index x_5 does not show up in the previous prediction, so we remove the node x_5 and its edge $\{x_4 \rightarrow x_5\}$. Therefore, the indexes appearing in both two steps are $\{x_{26}, x_1, x_3, x_4\}$. In the second phase, we calculate that their conditional probabilities are respectively $\{0.5, 0.54, 0.314, 0.375\}$, all greater than those in the first phase. So far, we have obtained the initial concurrent indexes as $\{x_{26}, x_1, x_3, x_4\}$. Because there are four initial indexes, we need to make at least two more further predictions from the initial indexes. Then we perform a similar operation and obtain the graph in fig. 12 (c). The probabilities of all events except the index x_1 and x_3 from $\{x_7 \rightarrow x_4 \rightarrow x_{26}\}$ are 1.0 for the reason that they are the only indexes in their prediction. Thus, we predict index sequence $\{x_7 \rightarrow x_4 \rightarrow x_{26}\}$ and its probability distribution is $\{1 : 0.5, 3 : 0.5\}$. It is noteworthy that the prediction for a sequence $\{x_7 \rightarrow x_{26} \rightarrow x_4\}$ is x_7, which does not appear in the previous prediction. Therefore, we remove the node x_7 and its corresponding edge $\{x_{26} \rightarrow x_4 \rightarrow x_7\}$. Next, we calculate the conditional probabilities of the initial indexs as $\{x_{26} : 1, x_1 : 0.678, x_3 : 0.775, x_4 : 0\}$. All probabilities of initial events have increased except the index x_4, which is 0. Thus, in fig. 12 (d), we strip away the sequences $\{x_3 \rightarrow x_4 \rightarrow x_1\}$, $\{x_4 \rightarrow x_{26} \rightarrow x_1\}$ and $\{x_4 \rightarrow x_{26} \rightarrow x_3\}$, which are circled with the red line. Then we make one more prediction and find all the branches converge to the index x_4 as demonstrated in fig. 12 (e). Finally, we know the concurrent indexes are $\{x_{26}, x_1, x_3\}$, and decide the index x_4 as the convergence point, in which the final graph constructed is given in fig. 12 (f).

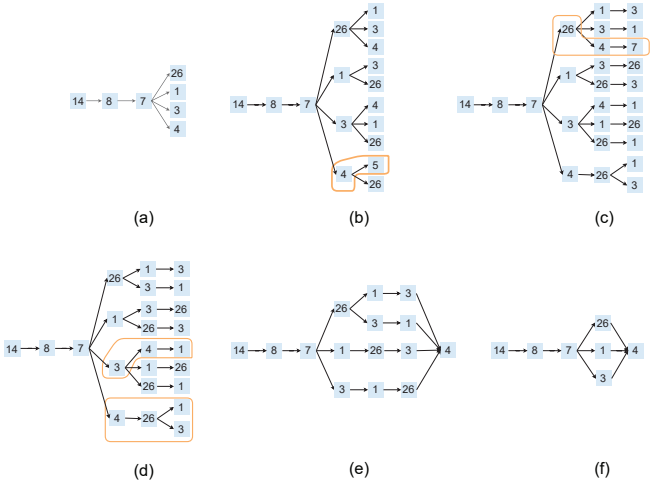


Fig. 12: Process of construction of graph for concurrent indexes

4 Theoretical Analysis

We compare DeepAG with four state-of-the-art models and briefly introduce these baselines as follows.

- Deeplog: Deeplog is a deep neural network framework leveraging LSTM to model a system log as a natural language sequence, and denoting every log entry as a index to make the predictions.
- Log2Vec: Log2Vec converts the log to weighted average of embeddings, and uses LSTM to model the log.
- LogAnomaly: LogAnomaly represents logs as semantic vectors. It uses an attention-based LSTM model as its classification algorithm.
- LogRobust: LogRobust is similar to LogAnomaly, denoting logs with semantic vectors. Moreover, it models the logs utilizing an attention-based Bi-LSTM model.

Specifically, Log2Vec, LogAnomaly, LogRobust, Deeplog, and DeepAG utilize recurrent layers of LSTM to process the temporal relationship of logs, and some of them also leverage self-attention mechanism to strengthen the ability to focus on more useful information. As a counterpart, the multi-head attention mechanism in DeepAG also plays the role in finding key information. Above all, recurrent layer and attention mechanism are the core techniques applied in these models. Therefore, we perform a theoretical analysis for the time complexity of recurrent layer and attention mechanism among DeepAG and its baselines respectively in the attack detection and prediction as shown in table 2, in which n is the sequence length, while d is the dimension of extracted features.

In terms of detection, DeepAG only has the attention mechanisms, and it has nd^2 complexity less than that of LogAmomaly and LogRobust. Comparing to Log2Vec, basically, to represent the semantic information more precisely, d is larger, which is 32 in DeepAG. However, n is 10 in our experiments, which is much less than d. Therefore, it costs the least time in general. On the other hand, for prediction, both Deeplog and DeepAG only have the recurrent layer. Although the time complexity of DeepAG is n times Deeplog, according to our performance analysis of prediction in section 5.3, DeepAG improves the accuracy clearly compared to Deeplog.

TABLE 2: Comparison of time complexity for recurrent layer and attention mechanism among DeepAG and its baselines, in terms of detection and prediction. n is the sequence length, d is the dimension of extracted features.

		Time complexity		Sum
		Recurrent	Attention	
Detection	Log2Vec	$O(nd^2)$	-	$O(nd^2)$
	LogAnomaly	$O(nd^2)$	$O(n^2d)$	$O(nd^2+n^2d)$
	LogRobust	$O(nd^2)$	$O(n^2d)$	$O(nd^2+n^2d)$
	DeepAG	-	$O(n^2d)$	$O(n^2d)$
Prediction	Deeplog	$O(nd^2)$	-	$O(nd^2)$
	DeepAG	$O(n^2d^2)$	-	$O(n^2d^2)$

5 Evaluation

The operating system in our experiment is macOS 10.15.4 with 32GB memory and 2GHz quad-core Intel Core i5 CPU, and Ubuntu16.04 with 48GB memory and GTX 1080Ti GPU. We use Pytorch 1.4.0 and Python3.6 to train the neural network model as a backend. Moreover, we compare DeepAG with four baselines: Deeplog [19], Log2Vec [16], LogAnomaly [18], and LogRobust [17], which have been introduced as section IV. In addition, to investigate the bi-directional mechanism, we compare the performance of DeepAG and BiLSTM.

5.1 Experiment setting

5.1.1 Data sets

Since our objective is to detect and predict the attacks as well as evaluate the performance, which is similar to anomaly detection, we conduct our experiments on HDFS, OpenStack, PageRank, and BGL data sets [1], in which the abnormal logs are all manully labeled by experts. The PageRank data set is gained from Hadoop platform.

We group log entries of HDFS and OpenStack into different sessions respectively by the block ID and instance ID, which are the identifier fields and extarcted as log indexes reprensenting events. Besides, we convert every log in PageRank and BGL data sets to vectors. Particularly, we process the data sets to make the ratio of positive and negetive samples close to 1:1. Table 3 summarizes the data sets we use.

TABLE 3: The statistics of data sets we used

	Number of sequences		Number of log indexes
	Training data	Test data	
HDFS	16560	7520	29
OpenStack	8460	2820	40
PageRank	215524	71840	No need
BGL	120000	40000	No need

To generate data set for backward prediction, we leverage previously attained forward model. First, to avoid the imbalance of data set for backward prediction, we extract all the different types of log sequences in training data for forward prediction, making every log sequence distinct. Then, for every session (including a log sequence with a length of h and its label l), we use forward model to predict log indexes for next h steps and obtain new log sequences with a length of h. Finally, for every log sequence, we reverse its order and use l as its label. Moreover, because a prediction will include most log indexes with extremely low probabilities which can even be ignored, caused by the feature of SoftMax layer, we set the probability threshold of outputting the predicted log indexes as 0.5 to avoid interference. Besides, sufficient learning of bi-directional model and great results in the following experiments can also guarantee the viability of this dataset. Similarly, the test data set is also obtained from test data of forward prediction in this way.

5.1.2 Set up

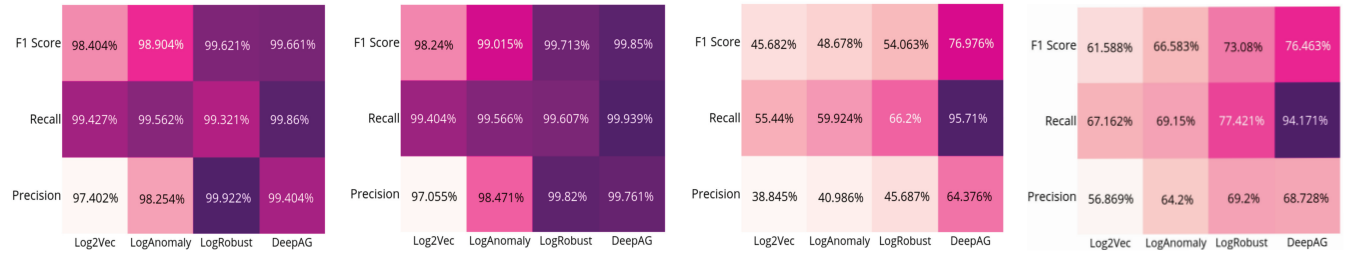
By default, we set the hyperparameters in the experiment as follows: h = 10, L = 2, g = 0.1, and α = 64. h is the window size of the LSTM and transformer model. L and α denote the number of layers of the LSTM and transformer model, and the number of memory units in a single LSTM block respectively. Besides, g is the threshold for outputting top few predictions that have higher probabilities.

5.2 Performance of detection.

Fig. 13 (a) (b) show *precision*, *recall*, and *F1 score* of semantic information-based models trained and tested on PageRank and BGL data sets respectively. DeepAG achieves the highest performance among the four models with an *F1 score* of 99.661% and 99.85% on PageRank and BGL data sets respectively. LogRobust has the highest *precision*, but has the most difference of *recall* on two data sets, where its *recall* on PageRank data set is 0.286% more than that on BGL data set. Therefore, a large number of exceptions have been missed, which may bring great challenges to the stability of software and hardware systems. LogAnomaly has a high

1. https://github.com/logpai/loghub

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3143551, IEEE Transactions on Dependable and Secure Computing



(a) Performance on PageRank data set (b) Performance on BGL data set (c) Performance on PageRank-based model (d) Performance on BGL-based model

Fig. 13: Performance of semantic information-based models

recall, and it also has a great *F1 Score* of 98.904% and 99.015% on PageRank and BGL data sets respectively. In addition, Log2Vec nearly has the lowest *precision*, *recall*, and *F1 Score* on two data sets among four models.

An advantage of representing logs with their semantic information is that we can learn the model from one data set, which is regarded as the source data set, and implement the experiments on another target data set. Thus, it helps relieve the problems of insufficient samples. Fig. 13 (c) (d) show *precision*, *recall*, and *F1 score* of models based on different source data sets respectively. Here, fig. 13 (c) shows the performance of models when training model on PageRank data set and test on BGL data set, while fig. 13 (d) is the presentation of performance of BGL-based model tested on PageRank data set. We find that DeepAG achieves better performance in both settings with *F1 Score* of 76.976% and 79.463% on PageRank source model and BGL source model respectively. It has the *F1 Score* that is over 20% more than other models with PageRank source model. Moreover, only DeepAG achieves the *recall* of over 95%, while other approaches have the *recall* around 60%.

Table 4 compares the test time among DeepAG (Transformer) and its baselines on PageRank and BGL data sets. Due to the parallel way of data processing, we notice that the time comsumed by DeepAG decreases greatly compared with other baselines when testing data set for detection, which is only 1.02 and 0.93 seconds. However, all other models cost 2 times of time more than that of DeepAG.

TABLE 4: Comparison of detection time

Model	Log2Vec	LogAnomaly	LogRobust	DeepAG
PageRank	2.54 s	2.8 s	3.1 s	1.02 s
BGL	1.96 s	2.21 s	2.64 s	0.93s

Inspired by LogRobust [17] which pointed out that log data is usually unstable, specifically meaning that one or several words are inserted into original logs, or removed from them caused by evolving logging statements or inaccurate log parsing, we validate the robustness of transformer model in DeepAG by inserting/removing some words into/from original log sentences, adding or deleting certain log sentences, and shuffling the orders of log sentences. In particular, these changes do not significantly change the semantic meaning of the original ones, thus the labels are not affected. Finally, we inject certain ratios of these changed logs into the original data set.

Fig. 14 compares *F1 Score* of DeepAG and its baselines on PageRank data set. When the ratio is low, we can notice

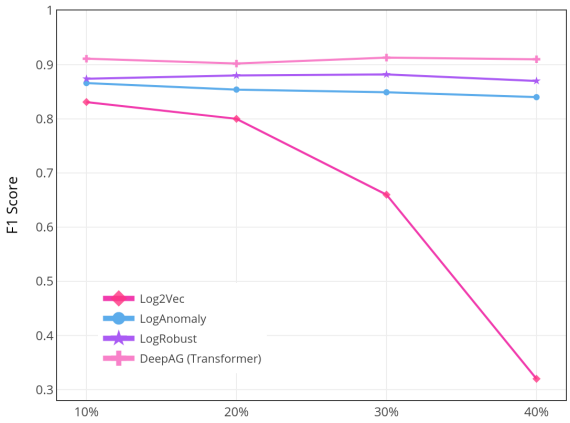


Fig. 14: Performance on unstable log data

the performance of all models are similar to the original performance. However, as the inject ratio increases, only the performance of Log2Vec declines significantly, which decreases by about 50% when the inject ratio ranges from 10% to 40%. In contrast, the performances of LogAnomaly, LogRobust, and DeepAG keep the stable trend though inject ratio increases. Moreover, DeepAG still achieves the best *F1 Score* under different inject ratios.

5.3 Performance of prediction.

Fig. 15 demonstrates the number of sequences in the backward data set generated from various thresholds and the *accuracy* of bi-directional model, respectively on HDFS and OpenStack data sets. We find the threshold has almost no impact on the accuracy, which means the predictions that have top few probabilities can match most labels. However, the smaller the threshold is, the more sequences will be generated, and thus the bigger the backward data set will be. Specifically, when the threshold turns from 0.1 to 0.7, the number of sequences generated from HDFS data set changes from 27882 to 2206, but the *accuracy* varied negligibly. However, the smaller threshold will lead to large data set, making it difficult to adjust parameters for fitting numerous information. In summary, it is reasonable to set up 0.5 as the threshold of generating backward data set for our experiment.

To avoid the impact of noisy indexes that have neglible probabilities as illustrated in section 3.4, we investigate the accuracy of Deeplog, backward model, and DeepAG in matching the predictions on HDFS and OpenStack data

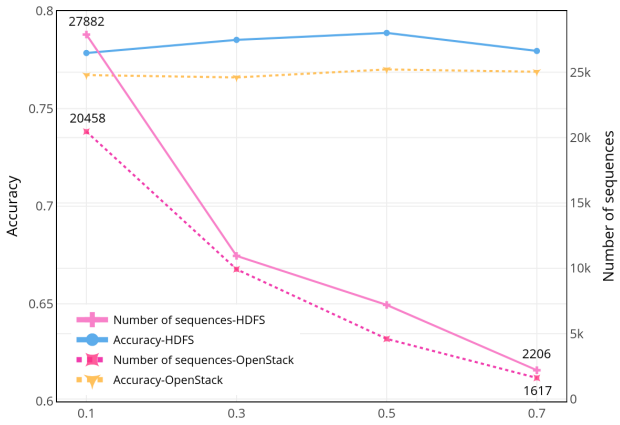
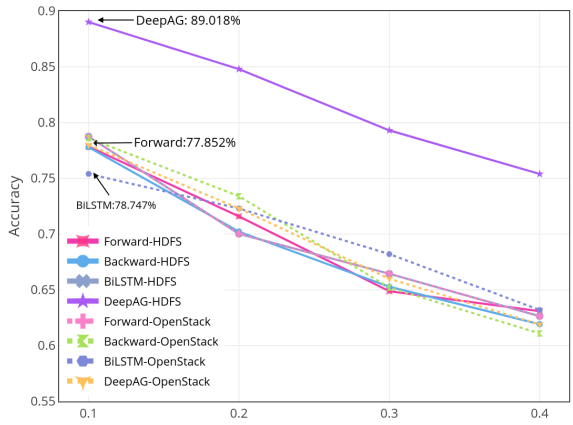
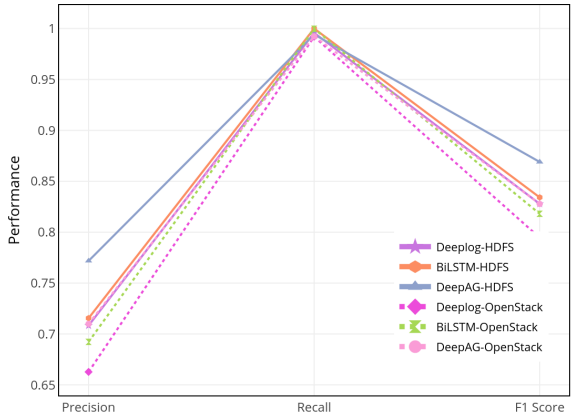


Fig. 15: Performance of different thresholds



(a) Performance of prediction in DeepAG



(b) Performance on two data sets

Fig. 16: Evaluation for index-based models

sets when setting different thresholds for LSTMs, shown in fig. 16 (a). In particular, Deeplog is the forward model. On HDFS data set, when comparing Deeplog and backward model, we find that the backward predictions match the predicted index for 77.8% of sequences when threshold is 0.1, and 70.223%, 65.28%, and 61.92% of seuquences with the thresholds of 0.2, 0.3, 0.4, which are greatly near to Deeplog that has the accuracy of 77.852%, 71.588%, 64.877%, and 63.087% in the counterparts. Therefore, it corroborates the viability of our approach to backward dataset processing.

Besides, DeepAG achieves the *accuracy* of 89.018% with the threshold of 0.1 and improves Deeplog by 11.166%. Mreover, 84.794%, 79.303% and 75.396% of labels are matched by DeepAG with the thresholds of 0.2, 0.3, 0.4 respectively, which are higher than Deeplog by 6.942%, 7.715%, and 10.519% separately. We also compare DeepAG and BiLSTM, both of which have the mechanism of forward and backward LSTM shown in fig. 16 (a). When matching the labels under the setting of threshold of 0.1, the *accuracy* of DeepAG is 10.271% higher than BiLSTM, and it improves BiLSTM by 14.722%, 12.86%, and 12.756% with the thresholds of 0.2, 0.3, 0.4 respectively.

Fig. 16 (b) compares the *precision*, *recall*, and *F1 Score* between Deeplog, BiLSTM, and DeepAG. It is clear that DeepAG achieves better performance than Deeplog, where DeepAG achieves *F1 Score* of 86.902% while *F1 Score* of Deeplog is 82.783%. When compared to BiLSTM that has the similar structure, we observe that both DeepAG and BiLSTM nearly achieve the *recall* of 100%. However, the *precision* of DeepAG is 5.604% higher than that of BiLSTM and DeepAG achieves better *F1 Score* (86.907%). When investigating the results on OpenStack data set, it conveys the similar trend.

5.4 Performance of adapting new patterns

5.4.1 Implementation of OOV word processor

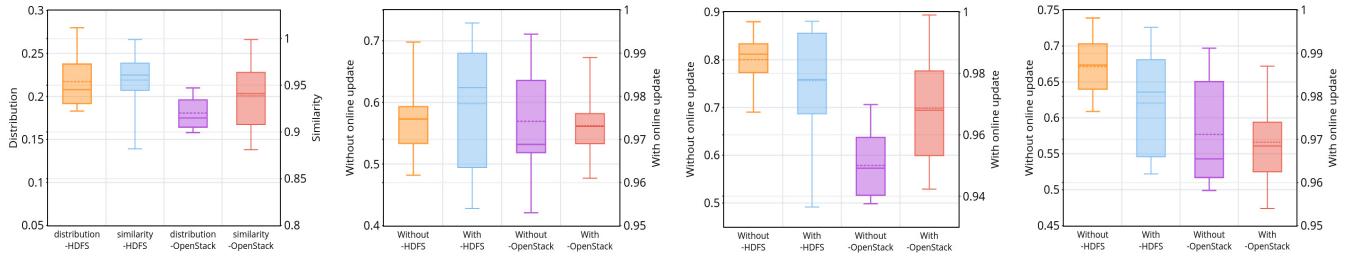
To strengthen the transformer model in DeepAG, we introduce OOV word processor, which can distributedly represent the unexpected words at runtime.

First, we extract several data from original data set ranging from 10% to 90%, in order to train the word embeddings and use the remaining logs to test the word representations. With training data sets, fig. 17 (a) shows the distribution of OOV words of the test data sets. In the HDFS data set, the percentage of OOV words in the test data set ranges from 0.183 to 0.28, while 0.159 to 0.21 in the PageRank data set. Then we randomly select a word in each log and change one of the letters to make the word OOV. Thus, each piece of the log contains OOV words. Next, we test the similarity (Cosine Similarity) between the changed log and the original log. Fig. 17 (a) also shows the similarity of changed logs with original logs. With different sizes of training data set, we obtain various similarities when evaluating the test data set. For the PageRank and BGL data sets, we observe that the both of their original logs achieve more than 0.9 similarities with the changed logs, which verifies the effectiveness of the OOV word processor.

5.4.2 Implementation of online update

Online update is the mechanism of adapting new patterns for bi-directional model in DeepAG. When the model makes predictions that are false positive, the users would incrementally update the model.

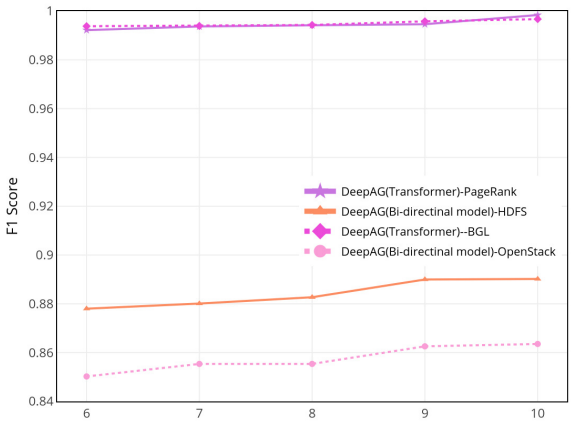
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3143551, IEEE Transactions on Dependable and Secure Computing



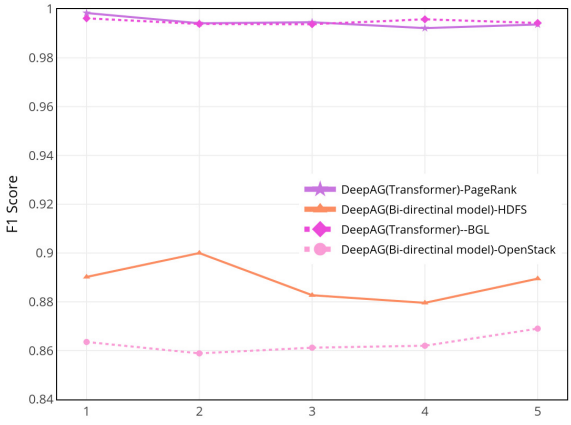
(a) OOV word processor (b) *Precision* of online update (c) *Recall* of online update (d) *F1 Score* of online update

Fig. 17: Performance of adapting new attack patterns

First, we extract several data from original data set ranging from 10% to 90% to train the bi-directional model, and the rest is for testing. We investigate the performance of bi-directional prediction without an online update. Then, after examining the false positive predictions, we add them to the training data set and learn a new pattern. Finally, we compare the performance of models built with or without the mechanism of online update, which are shown in fig. 17 (b) (c) (d). We note that online updating have improved the performance of model significantly. On the HDFS data set, the *F1 Score* without online update is from about 58% to 73% , and the *precision* is even less than 50%. However, after adding false positive predictions, the *precision* is higher than 95% and *F1 Score* achieves about 96.3% to 99.8%, which affirms the robustness of DeepAG. Similarly, on the OpenStack data set, the *precision*, *recall*, and *F1 Score* of DeepAG also improve a lot after being updated online , where the *F1 Score* increases by around 20%.



(a) window size



(b) the number of layers

Fig. 18: *F1 Score* of changing parameters

5.5 Evaluation for parameters

To figure out the influence of different parameters, we compare *F1 score* of DeepAG under different window sizes and number of layers respectively shown in fig. 18 (a) (b). According to fig. 18 (a), we find that increasing h or a number of layers leads to better performance for prediction, which could be caused by more sufficient learning. For example, when window size is 6, *F1 Score* of DeepAG on HDFS data set is 87.8%, which is improved to 89.018% when the window size is 10; while for detection, window size has neglible influence on DeepAG, and its *F1 Score* always keeps at around 99.5%. Having been informed that the performance of models can be influenced by changing h, it is worthy to seek proper patterns of h, in order to get a more satisfactory prediction. It is important because blindly raising h will bring indispensably sophisticated inner computing of network that rather weakens the real-time nature of DeepAG. However, DeepAG achieves the great performance under different parameter settings, which demonstrates that our model is robust.

In addition, fig. 18 (b) demonstrates the performance of models under different number of layers. In general, we observe that DeepAG is not very sensitive to the change of number of layers, and it achieves great performance even when set up to 1 layer, which means DeepAG can be easily deployed and used. That is because a greater number of layers lead to a longer time for training and prediction and more difficulty to fit the model.

5.6 Attack graph construction.

We simulate the concurrent situation utilizing the module *threading* in Python. By instantiating multiple objects of *Thread* that represent threads for concurrent programming and recording the execution paths in the HDFS logs, we determine the concurrent log indexes and obtain some index sequences. Through processing the data set, we obtain 14288 index sequences which include 1600 kinds of

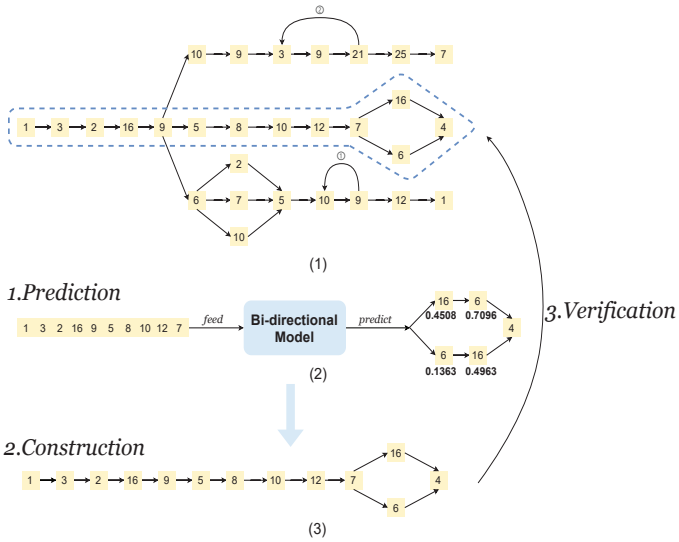


Fig. 19: Verification for part of attack graph

sequences composed of two concurrent indexes, 1000 kinds of sequences comprising three concurrent indexes, and 478 kinds of sequences containing four concurrent indexes. We retrain the bi-directional model and make cross validations. Table 5 shows the performance of retrained model.

TABLE 5: The performance of retrained model

Threshold	0.1	0.2	0.3	0.4
Accuracy	89.84%	83.2%	80.946%	73.171%

Fig. 19 (1) demonstrates a part of the existing attack graph constructed from the data set introduced above (noting that the ① and ② on the arrows denote the loop times of the particular structure). We use the execution path in the blue circle for verification. First, we make the $prediction$, shown in Fig. 19 (2). Fed with the sequence $\{1 \rightarrow 3 \rightarrow 2 \rightarrow 16 \rightarrow 9 \rightarrow 5 \rightarrow 8 \rightarrow 10 \rightarrow 12 \rightarrow 7\}$, bi-directional model outputs two indexes $\{16, 6\}$. We make one more prediction and know that the conditional probabilities of 16 and 6 are both greater than that in the previous prediction. Besides, we find the two sequences are converged to the same index in the further prediction. Then, by $construction$, we can obtain the graph with concurrent structure as shown in fig. 19 (3). Finally, by $verification$, we find it matches the real execution.

6 Related Work

6.1 Log-based detection

Log-based detection has always been emphasized by researchers, especially in the fields of cybersecurity, as it actually encodes the paths of intruders or any anomalies, which helps reveal their strategies.

Most researchers represent system logs with extracted log templates. In order to achieve the goal of automatic log parsing, academia and industry have proposed many methods, including frequent pattern mining [28], calculation of the longest common subsequence (Spell [29]), and parse tree (Drain [30]), etc. Besides, after obtaining log templates, some researchers convert logs to indexes or vectors, thus reaching an advancement of attack detection. Typically, Deeplog [19] converts log templates into indexes. Some researches [16] [18] utilize the log-specific semantic information to represent logs.

With standard log representations, deep learning is widely used by many researchers are able to model the behaviors of attackers and achieve the prediction and detection [31] [32]. From that perspective, Deeplog [19], LogRobust [17], and LogAnomaly [18] utilize LSTM to learn normal system behaviors and report all strange actions. Log2Vec [33] divides the log entries into five attributes and devises fine rules to define relationships among logs within a host, and then vectorizes user's operations, enabling a direct comparison of their similarities to find anomalies. Other approaches [34] [35] focus on analyzing user's logon operations to detect anomalous ones, which are capable of holding the information of interactive relationship among hosts.

6.2 Text vectorization

Text vectorization means that mapping the texts into vectors in a certain corpus. Essentially, it focuses on the extraction and convertion of word vectors, which avoids the loss of semantic information.

N-Gram [36] model is a vectorization method that estimates the probability from relative frequency counts, aiming to predict the next most likely word given a string of words. Another vectorization approach is word embedding [37] [38], including the Continuous Bag of Words(CBOW) model and skip-gram, which are distributed representations of text, based on the prediction through a neural network. It proposes that the semantic information of a word is jointly affected by the surrounding vocabularies. Meng et al. [16] capture words in logs more precisely through defining some pairs of synonyms and antonyms. In that case, domain-specific knowledge improves the performance, especially in the highly-targeted scenes. In order to express semantic information more effectively, many pre-trained word vector models have been proposed, such as GPT3 [39], and other word representations based on global word frequency statistics [40]. Google and Microsoft use a large number of data sets, corpora of various languages, and models with massive parameters to train word vectors that can be directly used by NLP tasks. FastText [24], a variant of Word2Vec published by Facebook, can train word vectors combining n-gram substrings of the full token. Therefore, for languages where such substrings are morphemes hinting at meaning, vectors can be synthesized for OOV words, better-than-random at some tasks especially if the OOV words are variants or misspellings of in-vocabulary words.

6.3 Graph construction

A majority of current work is transforming the behavioral features of users into sequences or graphs, which greatly help the intuitive analysis.

There have been many well-known open-source framework $Yi\ et\ al.$ [41] used to automatically generate attack graphs like MulVAL, TVA, Attack Graph Toolkit, NetSPA and so on. However, their default modeling has some unavoidable shortcomings. For example, MulVAL only gives an attack path instead of generating a complete attack

graph. In addition, their complexity in the worst case may be exponential. Xu. O et al. [42] extend the attack paths to attack graph and achieve the complexity of $O(N^2logN)$. Aiming to fixing the problems that MulVAL cannot represent network protocol vulnerabilities and does not support advanced types of communication and thus cannot model cyber-attacks on networks including IoT devices or industrial components, $Orly\ Stan\ et\ al.$ [43] present an extended network security model for MulVAL considering the short-range communication protocols and vulnerabilities.

Deeplog [19] divides the logs generated by concurrent threads into different sequences, and helps users find anomalies by constructing a workflow model for each individual task. Xiaojun Xu et al. [44] calculate the similarity of binary functions from different platforms to match threats. HOLMES [45] constructs a high-level graph according to the workflow and generates a compact graph by reducing noise and using priority sorting technology. POIROT [46] proposes the query graph, which is similar to the provenance graph. Segugio [47] focuses on who is querying what information and constructs a machine-domain bipartite graph based on DNS traffic between clients and the resolver.

Typically, many researchers apply and improve Node2Vec [48] and other graph embedding algorithms to study efficient techniques of heterogeneous graph learning. In fact, these emerging methods have already been applied in many fields such as recommendation systems, gene prediction, and so on, and also found effective in cybersecurity [21] [49] [50] in recent years. Hindroid [49] is the first work to apply heterogeneous information network (HIN) in the information security field. Wang et al. [50] consider node-level and semantic-level attentions, and propose a heterogeneous graph attention network (HAN) to handle heterogeneous graphs. HinDom [21] combines domain similarity to formulate multiple meta-paths and fully represents the rich semantics contained in DNS-related data.

7 DISCUSSION

While `DeepAG` is capable of making timely, concrete, and robust detections and predictions for attacks, and fixes the challenges as illustrated in section I, it is not complete enough because of lack of deployment, which is significant for deep learning models of cybersecurity from development environments to business operations systems. However, regarding it as our future work, we analyze the main challenges in this process.

Firstly, though there is often no shortage of data in the field of cybersecurity, good data sets are usually proprietary. Security vendors tend to "hide" security-related data, so it is usually impossible to obtain representative and accurate labeled data. Second, due to high professionality of data cleaning and labeling, annotating data sets related to network security detection requires professional security engineers. Thirdly, though DeepAG achieves lower complexity and higher accuracy than that of other states of the art in terms of detection, it still contains a large amount of calculation, posing requirements for the running environment and configuration. However, generally, the complexity of model with a lower error rate cannot be too low. Thus, the choice between efficiency and accuracy is a huge challenge, especially in security monitoring systems, which often require a rapid and real-time response to risks.

Besides, the deep learning framework is usually so complex that it contains hundreds of thousands of codes and many dependencies and almost inevitably known or unknown bugs. Moreover, the rapidly changing nature of cybersecurity makes it extremely difficult to maintain the system, because frequent data and model changes mean that other parts of the system also need to be changed. On the other hand, because the optimization goal of the algorithm is to make a globally optimal solution on the specified data set, the change in the size and distribution of the data set or the increase of black samples during the iteration process may cause the change of decision boundary of the model, making the attacks that system could originally detect cannot be identified after the model is updated.

In addition, our research is based on benign scenario. Specifically, we assume DeepAG is not attacked by malicious users, and attackers do not use the adversarial or tampered sample to interfere with detection. However, to improve the practicality of DeepAG and push its deployment, we are trying to study the anti-attack ability of DeepAG, focusing on the security of deep learning framework.

Finally, in the evaluation of graph construction, due to the complicated calculation brought by our findings of concurrency, it is difficult to test the accuracy of generating attack graph. In fact, this part of the work is under investigation in our lab, and we are actively searching for a solution to designing experiments. Nonetheless, we solve the shortcomings of Deeplog which needs to find an optimal g and is prone to be affected by the keys with negligible probabilities, by setting the probability threshold for prediction output. In that case, our separation methodology also provides useful insights towards the graph construction.

All the above challenges have made the deployment difficult. However, it is not unsolvable. Many researchers have developed practical systems based on their works and achieved great performance [51] [52]. With this confidence, we are fixing this problem and envision that a large spectrum of works and practical deployments can be realized along this road, contributing more power to cybersecurity.

8 CONCLUSION

To help proactive prevention for multi-step attacks, in this paper, we propose `DeepAG` which can be used for detecting attack sequences and predicting the potential threats. Furthermore, `DeepAG` can deal with unexpected patterns through the mechanisms of online upate and OOV word processor. Moreover, it can construct the attack graph intuitively demonstrating the attack path to model a more complicated situation, which will help users reduce the analysis burden and quickly master the strategies of attackers.

In addition, we find DeepAG is potential in providing a framework to achieve attack detection and prediction simultaneously. Though DeepAG performs better than other states of the art, the semantic gaps among different types of logs are difficult to eliminate. For future work, we will try to fix the semantic gaps and improve the perfomance of DeepAG in dealing with more comprehensive types of logs.

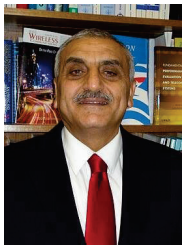
ACKNOWLEDGMENT

This research is funded by National Natural Science Foundation of China (No. 61902291), the Fundamental Research Funds for the Central Universities (XJS211516), Shaanxi University Science and Technology Association Youth Talent Promotion Project (20210120).

REFERENCES

[1] "Aptnotes," https://github.com/kbandla/APTnotes,2019.
[2] Y. Wang and W. M. etc., "A fog-based privacy-preserving approach for distributed signature-based intrusion detection," *Journal of Parallel and Distributed Computing*, 2018.
[3] S. Mohammadi, H. Mirvaziri, M. Ghazizadeh-Ahsaee, and H. Karimipour, "Cyber intrusion detection by combined feature selection algorithm," *Journal of Information Security and Applications*, 2019.
[4] Z. Zohrevand and U. Glsser, "Should i raise the red flag? a comprehensive survey of anomaly scoring methods toward mitigating false alarms," 2020.
[5] S. Anwar and M. Zain, "From intrusion detection to an intrusion response system: Fundamentals, requirements, and future directions," *Algorithms*, vol. 10, 2017.
[6] N. Sun, J. Zhang, P. Rimba, S. Gao, L. Y. Zhang, and Y. Xiang, "Data-driven cybersecurity incident prediction: A survey," *IEEE Communications Surveys Tutorials*, 2019.
[7] Y. Miao, C. Chen, L. Pan, Q.-L. Han, J. Zhang, and Y. Xiang, "Machine learningbased cyber attacks targeting on controlled information: A survey," *ACM Comput. Surv.*, 2021.
[8] X. Sun and J. e. Dai, "Using bayesian networks for probabilistic identification of zero-day attack paths," *IEEE Transactions on Information Forensics and Security*, 2018.
[9] S. Ma, J. Zhai, and F. W. etc., "Mpi: Multiple perspective attack investigation with semantic aware execution partitioning," in *26th USENIX Security Symposium*, 2017.
[10] A. Okutan, S. J. Yang, and K. McConky, "Predicting cyber attacks with bayesian networks using unconventional signals," in *Proceedings of the 12th Annual Conference on Cyber and Information Security Research*, 2017.
[11] P. Holgado, V. A. Villagrá, and L. Vázquez, "Real-time multistep attack prediction based on hidden markov models," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 1, pp. 134–147, 2020.
[12] H. A. Kholidy, A. Erradi, S. Abdelwahed, and A. Azab, "A finite state hidden markov model for predicting multistage attacks in cloud systems," in *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, 2014, pp. 14–19.
[13] D. Ourston, S. Matzner, W. Stump, and B. Hopkins, "Applications of hidden markov models to detecting multi-stage network attacks," in *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*, 2003, pp. 10 pp.–.
[14] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, and Y. Xiang, "A survey of android malware detection with deep neural models," *ACM Comput. Surv.*, 2020.
[15] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1825–1848, 2020.
[16] Y. L. W. Meng and Y. H. etc., "A semantic-aware representation framework for online log analysis," in *the 29th International Conference on Computer Communications and Networks (ICCCN)*, 2020.
[17] X. Zhang and Y. e. Xu, "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*, 2019.
[18] W. Meng and Y. e. Liu, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, 2019.
[19] M. Du and F. e. Li, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.
[20] X. Yu, P. Joshi, and J. Xu, "Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs," *ACM SIGARCH Computer Architecture News*, 2016.
[21] X. Sun, M. Tong, and J. Yang, "Hindom: A robust malicious domain detection system based on heterogeneous information network with transductive classification," 2019.
[22] Y. Xiong and Y. Z. etc., "Netcycle+: A framework for collective evolution inference in dynamic heterogeneous networks," *IEEE Transactions on Knowledge and Data Engineering*, 2018.
[23] A. Vaswani and N. e. Shazeer, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017.
[24] "Aptnotes," https://fasttext.cc/.
[25] Y. Shen, E. Mariconti, and P. A. Vervier, "Tiresias: Predicting security events through deep learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.
[26] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," *arXiv preprint arXiv:1801.06146*, 2018.
[27] Y. Pinter and R. Guthrie, "Mimicking word embeddings using subword RNNs," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2017.
[28] S. Zhang and W. M. etc., "Syslog processing for switch failure diagnosis and prediction in datacenter networks," in *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*, 2017.
[29] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *2016 IEEE 16th International Conference on Data Mining (ICDM)*, 2016.
[30] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *2017 IEEE International Conference on Web Services (ICWS)*.
[31] J. Zhang, L. Pan, Q.-L. Han, C. Chen, S. Wen, and Y. Xiang, "Deep learning based attack detection for cyber-physical system cybersecurity: A survey," *IEEE/CAA Journal of Automatica Sinica*, pp. 1–15, 2021.
[32] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1825–1848, 2020.
[33] F. Liu and Y. e. Wen, "Log2vec: A heterogeneous graph embedding based approach for detecting cyber threats within enterprise," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.
[34] A. Bohara and M. A. N. etc., "An unsupervised multi-detector approach for identifying malicious lateral movement," in *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, 2017.
[35] H. Siadati and N. Memon, "Detecting structurally anomalous logins within enterprise networks," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.
[36] S. Takase, J. Suzuki, and M. Nagata, "Character n-gram embeddings to improve RNN language models," *CoRR*, 2019.
[37] M. Naili, A. H. Chaibi, and H. Hajjami, "Comparative study of word embedding methods in topic segmentation," *Procedia Computer Science*, 2017.
[38] F. Zhang, "A hybrid structured deep neural network with word2vec for construction accident causes classification," *International Journal of Construction Management*, 2019.
[39] T. B. Brown, B. Mann, and N. R. etc., "Gpt-3: Language models are few-shot learners," 2020.
[40] C. Gong and D. e. He, "Frage: Frequency-agnostic word representation," in *Advances in Neural Information Processing Systems*, 2018.
[41] S. Yi, Y. Peng, Q. Xiong, T. Wang, Z. Dai, H. Gao, J. Xu, J. Wang, and L. Xu, "Overview on attack graph generation and visualization technology," in *2013 International Conference on Anti-Counterfeiting, Security and Identification (ASID)*, 2013, pp. 1–6.
[42] X. Ou, W. F. Boyer, and M. A. McQueen, "A scalable approach to attack graph generation," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 336345. [Online]. Available: https://doi.org/10.1145/1180405.1180446
[43] O. Stan, R. Bitton, M. Ezrets, M. Dadon, M. Inokuchi, O. Yoshinobu, Y. Tomohiko, Y. Elovici, and A. Shabtai, "Extending attack graphs to represent cyber-attacks in communication protocols and modern it networks," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2020.
[44] X. Xu and C. e. Liu, "Neural network-based graph embedding for cross-platform binary code similarity detection," *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017.
[45] S. M. Milajerdi and R. G. etc., "Holmes: Real-time apt detection through correlation of suspicious information flows," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.
[46] S. M. Milajerdi and B. e. Eshete, "Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[47] B. Rahbarinia and R. e. Perdisci, "Efficient and accurate behavior-based tracking of malware-control domains in large isp networks," *ACM Trans. Priv. Secur.*, 2016.
[48] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Acm Sigkdd International Conference*, 2016.
[49] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu, "Hindroid: An intelligent android malware detection system based on structured heterogeneous information network," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017.
[50] X. Wang and H. e. Ji, "Heterogeneous graph attention network," in *The World Wide Web Conference*, 2019.
[51] Y. Jia, L. Xing, Y. Mao, D. Zhao, X. Wang, S. Zhao, and Y. Zhang, "Burglars' iot paradise: Understanding and mitigating security risks of general messaging protocols on iot clouds," *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 465–481, 2020.
[52] N. Wang, L. Jiao, P. Wang, W. Li, and K. Zeng, "Machine learning-based spoofing attack detection in mmwave 60ghz ieee 802.11ad networks," in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, 2020, pp. 2579–2588.



Mohammad S. Obaidat received his Ph.D. degree in Computer Engineering with a minor in Computer Science from The Ohio State University, Columbus, USA. He has received extensive research funding and published To Date (2020) about 1000 refereed technical articles-About half of them are journal articles, over 70 books, and over 70 Book Chapters. He is the founding Editor-in Chief of Wiley Security and Privacy Journal. He has chaired over 160 international conferences and has given over 160 keynote speeches worldwide. He has served as ABET/CSAB evaluator and on IEEE CS Fellow Evaluation Committee. He was awarded the IEEE CITS Hall of Fame Distinguished and Eminent Award. He is a Life Fellow of IEEE and a Fellow of SCS. His research interests include Cybersecurity, Wireless Networks Computer Networks and Modeling and Simulation.

Teng Li received the B.S. degree in school of computer science and technology from Xidian University, China in 2013, and Ph. D. degree in school of computer science and technology from Xidian University, China in 2018. He is currently a lecturer at the school of cyber engineering, Xidian University, China. His current research interests include wireless and networks, distributed systems and intelligent terminals with focus on security and privacy issues.

Yulong Shen received the B.S. and M.S. degrees in computer science and PhD degree in cryptography from Xidian University, Xi' an, China, in 2002, 2005, and 2008, respectively. He is currently a Professor with the School of Computer Science and Technology, Xidian University, where he is also an Associate Director of the Shaanxi Key Laboratory of Network and System Security and a member of the State Key Laboratory of Integrated Services Networks. His research interests include wireless network security and cloud computing security. He has also served on the technical program committees of several international conferences, including ICEBE, INCoS, CIS, and SOWN.

Ya Jiang is currently a BSc student at the School of Computer Science and Technology at Xidian University, China. Her current research interests include deep learning applications to cybersecurity, privacy and security of federated learning.

Jianfeng Ma received the B.S. degree in computer science from Shaanxi Normal University in 1982, and M. S. degree in computer science from Xidian University in 1992, and the Ph. D. degree in computer science from Xidian University in 1995. Currently he is the directer of Department of Cyber engineering and a professor in School of Cyber Engineering, Xidian University. He has published over 150 journal and conference papers. His research interests include information security, cryptography, and network security.

Chi Lin received B.E. and Ph.D. from Dalian University of Technology, China, in 2008 and 2013, respectively. He has been an Assistant Professor in School of Software, Dalian University of Technology (DUT), China since 2014. He is currently an Associate Professor with the School of Software, Dalian University of Technology since 2017. Dr. Lin has authored over 50 scientific papers including INFOCOM, SECON, ICDCS, IEEE Trans. on Computing, ACM Trans on Embedded Computing Systems. In 2015, he was awarded ACM Academic Rising Star. His research interests include pervasive computing, cyber physical systems (CPS), and wireless sensor networks