EULER: Detecting Network Lateral Movement via Scalable Temporal Link Prediction

Isaiah J. King and H. Howie Huang
George Washington University
{iking5, howie}@gwu.edu

***Abstract*—Lateral movement is a key stage of system compromise used by advanced persistent threats. Detecting it is no simple task. When network host logs are abstracted into discrete temporal graphs, the problem can be reframed as anomalous edge detection in an evolving network. Research in modern deep graph learning techniques has produced many creative and complicated models for this task. However, as is the case in many machine learning fields, the generality of models is of paramount importance for accuracy and scalability during training and inference. In this paper, we propose a formalized approach to this problem with a framework we call EULER. It consists of a model-agnostic graph neural network stacked upon a model-agnostic sequence encoding layer such as a recurrent neural network. Models built according to the EULER framework can easily distribute their graph convolutional layers across multiple machines for large performance improvements. Additionally, we demonstrate that EULER-based models are competitive, or better than many state-of-the-art approaches to anomalous link detection and prediction. As anomaly-based intrusion detection systems, EULER models can efficiently identify anomalous connections between entities with high precision and outperform other unsupervised techniques for anomalous lateral movement detection.**

I. INTRODUCTION & MOTIVATION

Lateral movement is a key stage of the MITRE ATT&CK framework [1] describing the behavior of advanced persistent threats (APTs). At its core, lateral movement is malware propagating through a network to spread onto new computers in an attempt to find their target. This may involve pivoting through multiple systems and accounts in a network using either legitimate credentials or malware to accomplish the task [2]. As it is one of the final steps in the killchain before complete compromise, detecting it early is of critical importance.

A plethora of machine learning approaches to intrusion detection exist, both signature-based models [3], [4], [5], [6], [7], [8] and anomaly-based [9], [10], [11], [12], [13], [14], [15]. These latter techniques are especially well-suited for lateral movement detection, as APT techniques like "Pass the Ticket" [16], or even just using stolen credentials [17] are very difficult to formalize into signatures for signature-based intrusion detection systems [2].

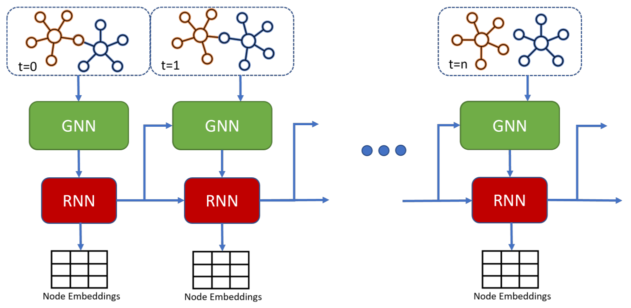
Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24107
www.ndss-symposium.org

The most robust way to detect malware propagation is not to exhaustively list every known malicious signature correlating with it; rather it is to train a model to learn what normal activity looks like, and to alert when it detects behavior that deviates from it. However, detecting anomalous activity in an enterprise network presents unique challenges. The data involved both during training and the implementation of an anomaly-based intrusion detection system is enormous. Often the log files that such a system would require as input are terabytes large. To be useful, a lateral movement detection model must be highly scalable to accommodate such large data. Additionally, when viewed as a classification problem, any such system would have to be highly precise. Millions of events occur in an enterprise network on any given day, and only a fraction of a percent of all interactions are ever anomalous [18]. Therefore, a model must have an extremely low rate of false alerts so as not to overwhelm its users.

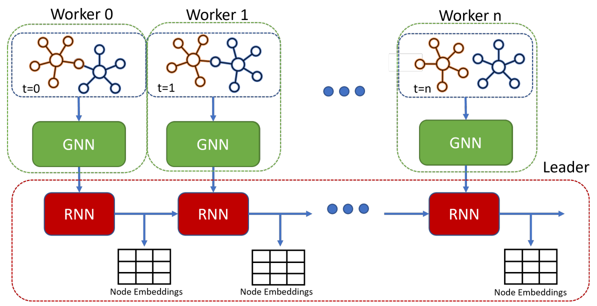
In this work, we formulate anomalous lateral movement detection as a temporal graph link prediction problem. Interactions occurring in discrete units of time on a network can be abstracted into a series of graphs $\mathcal{G}_t = \{\mathcal{V}, \mathcal{E}_t\}$ called snapshots, where $\mathcal{V}$ is the set of entities in the network that had interactions $\mathcal{E}_t = \{(u, v) \in \mathcal{V}\}$ during a set period of time, t. A temporal link prediction model will learn normal patterns of behavior from previous snapshots and assign likelihood scores to edges that occur in the future. Edges with low likelihood scores correlate to anomalous connections within the network. As [9] points out, these anomalous connections are often indicative of lateral movement. As we will later show, this reframing of the problem improves precision over standard anomaly-based intrusion detection techniques.

Recent approaches to temporal link prediction combine a graph neural network (GNN) with a sequence encoder such as a recurrent neural network (RNN) to capture topological and temporal features of an evolving network. However, these approaches are either reliant on RNN output during the GNN stage of embedding [19] or merely incorporate GNNs into the RNN architecture [20], [21], [22]. As Figure 1a illustrates, these models are necessarily sequential, and unfortunately cannot scale to the large datasets that they would need to process to be useful lateral movement detectors.

Proposed solution. To address this problem, we have observed that the most memory-intensive part of existing architectures occurs during the message passing stage within the GNN. Furthermore, there exists an imbalance between the massive size of node input features and the comparatively minuscule topological node embeddings. This means the most work and the most memory usage occurs in the GNN before



(a) Sequential temporal link predictors



(b) EULER parallel framework

Fig. 1: (a) Prior approaches rely on RNN output during the GNN stage of embedding or merely incorporate GNNs into the RNN architecture, which forces models to work in serial, one snapshot at a time. In contrast, (b) the EULER framework can utilize several worker machines to hold consecutive snapshots of a discrete temporal graph. These workers process snapshots in parallel through a replicated GNN shared across each machine. The output of these GNNs is returned to the leader machine which runs them through a recurrent neural network to create temporal node embeddings which may be used for link prediction. The framework is explained in detail in Section IV

the simpler forward pass of the RNN is calculated, necessarily in serial. If several replicated GNNs operate on snapshots independently, they can execute concurrently as shown in Figure 1b. Amdahl's Law [23] would suggest that by distributing such a large portion of work, performance improvements will ensue.

In this work, we have developed EULER[1], a formalized approach for scalable dynamic link prediction and anomalous edge detection. The framework involves stacking a model agnostic GNN upon a sequence encoder such as an RNN. In this way, a network's topology at discrete moments in time is encoded by the GNN, and the dynamic changes in connections are encoded by the RNN. The embeddings produced by this model provide prior probabilities for future states of the network given what is embedded about the past structure. Most importantly, the framework is designed such that GNNs may be replicated across several worker machines, and execute independently, allowing disjoint sets of snapshots to be processed concurrently. When the GNNs' work occurs in parallel for each snapshot, the topological data for an entire series of graphs can theoretically be encoded in the time it takes to encode the snapshot with the most edges. With these immense performance enhancements, detecting anomalous user activity in industry-scale, real-world networks using powerful GNN models becomes tractable.

Experimental evaluation. To prove that the very simple model we propose is adequate for the complex task of anomalous edge detection, and by extension lateral movement detection, we evaluate a model following the EULER framework against several state-of-the-art temporal link prediction models on three datasets. The results of these experiments show that despite its simplicity, the EULER framework performs as well or better than the state-of-the-art, and unlike the other models, can scale to accommodate larger data.

Implementation as an intrusion detection system. We test several models following the EULER framework on the LANL comprehensive multi-source cyber-security events dataset [24]. This dataset includes labeled event data from 58 consecutive days of real-world computer network use interspersed with red team activity. There are approximately 1.6 billion events in total. This is a test not only of the EULER framework's precision, but also its ability to scale. Our tests show that EULER-based models outperform prior works in both precision and compute time.

In summary, our research contributions are:

- We present, to the best of our knowledge, the first use of temporal graph link prediction for anomaly-based intrusion detection. Other research in applying graph analytics to anomaly detection either does not consider the temporal nature of the data or does not use a powerful GNN model. By incorporating both elements into EULER, models built on this framework outperform other unsupervised anomaly-based intrusion detection systems and yield more informative alerts.
- We demonstrate that for temporal link prediction and detection, the simple framework we propose is equally or more accurate and precise than state-of-the-art temporal graph autoencoder models. EULER models attain higher area under the curve and average precision scores by 4% in inductive link detection tests, and about equal metrics to the state-of-the-art in transductive link prediction tests.
- We propose a scalable framework for distributed temporal link prediction for use on big data. The EULER framework is simple and makes message passing lightweight even over large graphs. By breaking edge lists into temporal chunks and processing them in parallel, the computational complexity of the message passing stage of the model is theoretically bound by only the snapshot with the most edges. Other optimizations allow the RNN to operate in parallel with some of the GNN workers, further improving performance.

[1]Source code available at https://github.com/iHeartGraph/Euler

The rest of this work is organized as follows: Section II provides background information about the topic and defines our key terms. Section III gives a motivating example to demonstrate the necessity of temporal graphs in anomaly-based intrusion detection. Section IV explains the EULER framework in detail. Section V details the experiments we conducted with this model compared to other temporal autoencoders. Section VI showcases how we used EULER-based models to build efficient and precise anomaly-based intrusion detection systems. Section VII discusses related work in anomaly-based intrusion detection systems and temporal link prediction. Finally, we provide a brief discussion and suggestions for future work in Section VIII.

II. BACKGROUND

Anomaly-based intrusion detection systems must first define a baseline for normal behavior, then generate alerts when events occur that significantly deviate from this baseline. The definition for normalcy is highly contingent on the abstraction used to represent the system. Our proposed solution represents network activity as discrete temporal graphs. From there, detecting evidence of lateral movement betrayed by anomalous network activity is equivalent to anomalous edge detection, which we accomplish via temporal link prediction.

Discrete Temporal Graphs

A discrete temporal graph[2] $\mathcal{G} = \{\mathcal{G}_1, \mathcal{G}_2, ...\mathcal{G}_T\}$ is defined as a series of graphs $\mathcal{G}_t = \{\mathcal{V}, \mathcal{E}_t, \mathbf{X}_t\}$, which we refer to as snapshots, representing the state of a network at time t. Here, $\mathcal{V}$ denotes a set of all nodes which appear in the network, $\mathcal{E}_t$ denotes relationships between nodes at time t, and $\mathbf{X}_t$ represents any features associated with the nodes at time t. In this work, all graphs are directed, and some have weighed edges, $W : \mathcal{E} \to \mathbb{R}$ representing edge frequencies during the time period each snapshot encompasses.

Let the interactions $\mathcal{I}$ between users and machines in a network at specific times be represented as a multiset of tuples <*src,dst,ts*>. Here, *src* is an entity interacting with entity *dst* at time *ts*. From this multiset, we can build the temporal graph $\mathcal{G} = \{\mathcal{G}_0, ..., G_T\}$ with time window δ. The set of all nodes $\mathcal{V}$, is the set of every *src* and *dst* entity that appears in $\mathcal{I}$. The set of edges at time t, $\mathcal{E}_t$ is constrained such that for all edges $(u, v) \in \mathcal{E}_t$ there exists an interaction $< u, v, w > \in \mathcal{I}$ where $t \leq w < t + \delta$.

Temporal Link Prediction

Temporal link prediction is defined as finding a function that describes the likelihood that an edge exists in a temporal graph at some point in time, given the previously observed snapshots of a network. By representing an enterprise network as a temporal graph, we can further extend this definition to encompass anomaly detection. This follows from the assumption that anomalous edges in a temporal graph will have a low probability of occurring given what is known about the network's behavior in the past.

Lateral movement detection with temporal link prediction is then defined as finding a function learned from the temporal graph $\mathcal{G}$ of network activity that predicts the likelihood of future interactions occurring in unseen snapshots. An observed interaction between entities with a likelihood score below a certain threshold is said to be anomalous. These anomalous edges, in the context of network monitoring, are often indicative of lateral movement [9].

III. MOTIVATION

Historically, there have been two main ways of abstracting data for anomaly detection: frequency-based, and events-based [18]. Frequency-based methods, such as [12], [13], [25] define anomalous behavior as activity which significantly deviates from observed temporal patterns. Events-based methods, such as [10], [14], [15], [26], [27], [28] identify commonalities in features, such as packet count, protocols, etc., associated with individual events within a system. Anomalies are then defined as events with unexpected features.

The problem with these two approaches is that they ignore the inherently structural nature of network data. This is readily apparent by observing that one of the most influential datasets for intrusion detection, [29], [30], has no features for source or destination entities. All that matters with these models are the details of the events themselves in isolation, not the machines between which they occurred. Networks are too complex to be represented as just a series of unrelated vectors, or a multivariate probability distribution of isolated samples. Networks are webs of relational data, fluctuating over time. The most natural way to represent and analyze relational data, is as a graph.

There has been growing interest in static graph-based methods for intrusion detection, such as [9]. Here, normalcy is defined in terms of interactions between entities within systems, but time is not considered. Anomalies are defined as edges with low probability given what is known about the graph's structure.

To demonstrate the advantages and disadvantages of the above intrusion detection systems, consider the example shown in Figure 2. The first two time slices show normal activity in the network: first at $t0$, Alice and Bob authenticate with their computers A and B, then at $t1$ computers A and B make a request to the shared drive. At times $t2$ and $t3$, we see that when Bob does not first authenticate with computer B, it does not communicate with the shared drive. A simple probability distribution is apparent:

$$\begin{aligned} P((C1, SD) \in \mathcal{E}_{t+1} \mid (B, C1) \in \mathcal{E}_t) = 1 \\ P((C1, SD) \in \mathcal{E}_{t+1} \mid (B, C1) \notin \mathcal{E}_t) = 0 \end{aligned} \tag{1}$$

However, in $t4$ and $t5$, something unusual occurs: computer B requests data from the shared drive without Bob authenticating with it first! This could be a sign of an APT using attack techniques (as catalogued by the MITRE ATT&CK framework) T1563, remote service hijacking, or attempting T1080, tainting shared content [2].

In order to detect attacks such as the one in the example, a model would need to consider events with reference to those which occurred previously, and with reference to the other interactions within the network. An event between two entities that happens at one point in time cannot be considered

[2]For simplicity, for the remainder of this work, we refer to these simply as "temporal graphs"

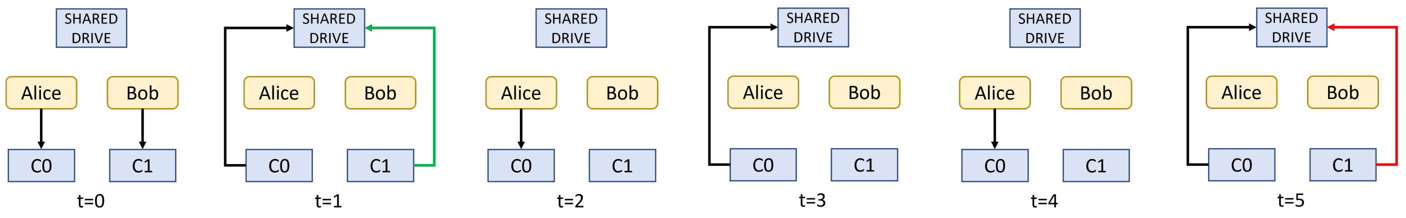


Fig. 2: A simple example of hard to detect temporally anomalous activity in a network with 3 machines and two users. The normal sequence of events is that a user authenticates with a computer, then that computer makes a request to the shared drive. However, at time t5, computer 1 makes a request to the shared drive without Bob first authenticating with it, denoting a perhaps malicious process running on computer 1 acting not on behalf of the machine's primary user.

identical to the same event occurring in the future under a different global context. Unfortunately, existing graph-based approaches, which do not consider time, and many event-based approaches that look at each event in isolation such as [9], [10], [14] would see no difference between $(C1, SD)$ at time $t1$ and $(C1, SD)$ at time $t4$. What makes it stand out as anomalous is something lacking in the previous state of the graph–something that would only be detected by considering prior probabilities for the next state of the system.

Frequency-based and event-based approaches such as [12], [13], [26], [27], [28] would have similar trouble, as they lack the relational data present in the network; Bob's activities a timestep earlier would have little import on the interpretation of the shared drive's activity later on. If the interaction between $(C1, SD)$ at time $t4$ was sufficiently similar to the interaction at time $t1$, these approaches would see no difference between the two. They lack the ability to capture the importance of interactions occurring between other entities in the network, and how they may relate to a separate event.

Our proposed solution, to represent the network as a temporal graph, ensures the global structure of a network at individual points in time is captured without losing the temporal dependencies of the changing connections. We make the more difficult assumption that malicious events can have the same event features as normal ones; if this is the case, traditional event-based approaches will not work. We also assume attackers can make similar connections, or even the same connections as observed in normal activity, meaning available graph-based approaches, and statistical approaches are insufficient. With temporal graphs, assuming they have enough granularity, these problems, as well as those tackled by prior works are solvable.

IV. EULER

In this section we describe our proposed framework, which we call EULER. This framework aims to learn a probability function conditioned on previous states of a temporal graph, to determine the likelihood of an edge occurring at a later state. Furthermore, it is the goal of this work to offer an approach that is not just precise, but also highly scalable. We first describe the basic components of the system, then how they are distributed across multiple machines, and how these components interact. Finally, we describe how different training objectives are implemented on the EULER interface.

A. *The Encoder-Decoder*

The EULER framework is a generic extension of the traditional graph autoencoder model [32] to temporal graphs. It consists of a model-agnostic graph neural network (GNN) stacked upon a model-agnostic recurrent neural network (RNN). Together, these models aim to find an encoding function $f(\cdot)$ and a decoding function $g(\cdot)$. The encoding function maps nodes in a temporal graph with T snapshots to T low-dimensional embedding vectors. The decoding function ensures minimal information is lost during the encoding process and aims to reconstruct the input snapshots from the latent $\mathbf{Z}$ vectors. More formally, we can describe the behavior of the encoder as

$$\begin{aligned}\mathbf{Z} &= f(\{\mathcal{G}_0, \dots, \mathcal{G}_T\}) \\ &= \text{RNN}(\ [\text{GNN}(\mathbf{X}_0, \mathbf{A}_0), \dots, \text{GNN}(\mathbf{X}_T, \mathbf{A}_T)]\ )\end{aligned} \tag{2}$$

where $\mathbf{A}_t$ is the $|\mathcal{V}| \times |\mathcal{V}|$ adjacency matrix representation of the snapshot at time t. This $T \times |\mathcal{V}| \times d$ dimensional tensor $\mathbf{Z}$ is optimized to contain information about both the structure of the graph, and the dynamics of how it changes over time.

This is enforced by a decoder function, $g(\cdot)$ which attempts to reconstruct the original graph structure given the embeddings. More formally,

$$g(\mathbf{Z}_t) = \Pr(\mathbf{A}_{t+n} = 1 \mid \mathbf{Z}_t) \tag{3}$$

where $\mathbf{Z}_t = \mathbf{Z}[t]$ is the embedding of graph $\mathcal{G}_t$ and $n \geq 0$. As was done by [9], [19], [32], [33] we use the inner product decoding as this $g(\cdot)$ function:

$$g(\mathbf{Z}_t) = \sigma(\mathbf{Z}_t \mathbf{Z}_t^\intercal) = \tilde{\mathbf{A}}_{t+n} \tag{4}$$

where $\sigma(\cdot)$ denotes the logistic sigmoid function, and $\tilde{\mathbf{A}}_{t+n}$ represents the reconstructed adjacency matrix at time $t + n$. As a consequence of using inner product decoding, the dot product of vectors $\mathbf{Z}_t[u]$ and $\mathbf{Z}_t[v]$ represents the log-odds that an edge, (u, v) exists at time $t + n$. In this way, the $g(\cdot)$ function is used to detect anomalous edges.

B. *Workflow*

The core of the EULER framework is a simple design. It simply stacks the replicas of a model-agnostic GNN which we refer to as the *topological encoder* upon a model-agnostic *recurrent layer* with a few simple constraints. When fit into the leader/worker paradigm[3] with one recurrent layer as the

[3]Historically called the master/slave paradigm

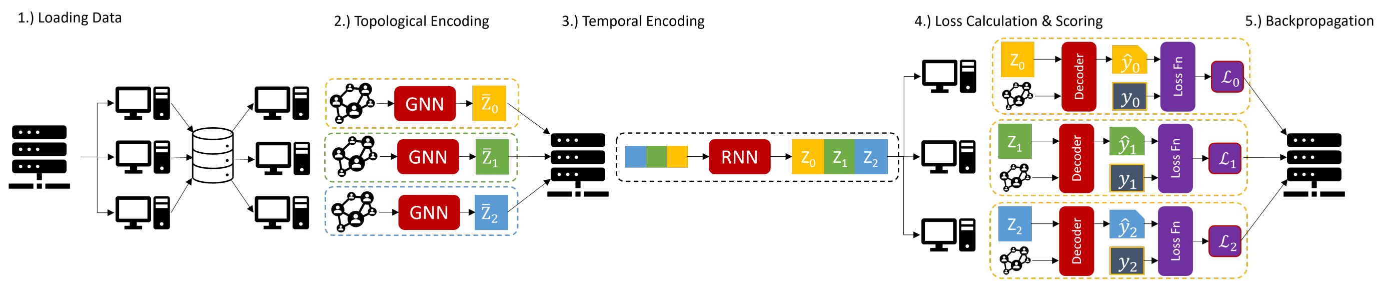


Fig. 3: The complete series of interactions between the leader, and worker machines during one training step. In step 1.), the leader machine initiates each worker, and issues a command for each of them to load a disjoint subset of the snapshots from memory. When this has occurred, the training loop can begin. At step 2.), the leader issues a command to each worker to perform a forward pass on their graphs through their GNNs. As they complete this, workers send the topological embeddings, $\bar{\mathbf{Z}}_t$ to an ordered queue in the leader. Upon receiving the embedding for $t = 0$, step 3.) begins. The leader runs embeddings from the queue through its RNN as they come in to produce the final $\mathbf{Z}$ embeddings. Then, in step 4.), the leader sends the embeddings back to the workers to decode and calculate loss. Finally, 5.), the leader and workers perform a backward pass on the aggregated loss functions according to the `DDP` gradient bucketing algorithm [31]. During evaluation, the steps are nearly identical, but on step 4.) workers return their scores for each edge $\hat{y}_t$ instead of loss.

leader, and multiple topological encoders as workers, it has the potential for massive parallelism.

The overall workflow for EULER is shown in Figure 3. It occurs in 5 basic stages: 1.) The leader spawns the workers and instructs them on which snapshots to load; 2.) the leader initiates the training loop, and workers generate topological embeddings; 3.) as the topological embeddings are received, the leader processes them through an RNN; 4.) the output of the RNN is sent back to the workers to calculate loss or for scoring; 5.) in training mode, loss is returned to the leader to be backpropagated. During evaluation, anomaly scores are returned.

In this section, we describe how these 5 steps are implemented in greater detail. The distributed workers are implemented using the PyTorch `DDP` library [31]. More detailed technical information about the EULER interface is available in the Appendix.

Loading Data

When the program first starts, several processes are spawned. The PyTorch multiprocessing library automatically assigns each spawned process a unique ID from 0 to $k + 1$; for convenience the machine with id:0 becomes the leader, and all others are workers. The leader machine, which holds remote references to all workers in addition to the recurrent layer issues commands to the other processes to spin up all worker instances.

Upon their creation, workers connect to the leader and await instructions. After the initialization of all workers, the leader assigns them contiguous subsets of temporal graph data to load, shown in step 1 of Figure 3. For example, given a set $W = \{w_1, ..., w_k\}$ of k workers, and a temporal graph split into T snapshots, the leader assigns each worker $s = \lfloor \frac{T}{k} \rfloor$ snapshots. Then worker w_i holds $\{\mathcal{G}_{si}, ..., \mathcal{G}_{si+(s-1)}\}$. In the likely case where k does not divide T, extra units of work are assigned to workers holding snapshots later in time. This is because the recurrent layer of the model processes the topological encoders' output in order, as it comes in. Thus, the RNN can perform the necessarily sequential forward pass on earlier embeddings while future snapshots are still being processed by workers. The method for this distribution of work is explained in more detail in Algorithm 1.

Algorithm 1: Distributing $\mathcal{G}$ across workers

```
initialize $k$ workers, and temporal graph $\mathcal{G} = \{\mathcal{G}_0, ..., \mathcal{G}_T\}$;
/* Give each worker an equal amount of work */
int minTasks = $\lfloor \frac{T}{k} \rfloor$;
int tasks = [minTasks] $*$ k;
/* If work is not evenly divided, assign it to
   last workers first */
int remainder = $T$ % $k$;
if remainder then
    for (int i=k-1; i≥k-remainder; i--) do
        tasks[i]++;
    end
end
/* Each worker loads as many contiguous
   snapshots as they were assigned */
int tmp, start=0, end=tasks[0];
for (i=0; i<k; i++) do
    asynchronously call workers[i].load_new_data($\mathcal{G}$[start:end]);
    tmp=start; start += end; end = tmp+tasks[i+1];
end
[w.wait() for w $\in$ workers]
```

After the workers have been assigned their snapshots, they concurrently read them in. The leader waits for each worker to signal that all data is loaded, then moves on to the next phase of the workflow.

Topological Encoding

During the forward pass–shown in step 2 of Figure 3–the recurrent layer issues asynchronous calls to `forward` on each worker machine. To minimize network traffic, the only thing the leader sends to the workers is an enum representing which partition of edges to process, as some are held out for validation and testing. Workers then process every snapshot they hold. Further reducing network traffic, the matrices returned by the workers are far smaller than those used as inputs, as each worker is essentially a graph autoencoder [32].

Algorithm 2: Leader machine forward method

```
def forward(self, workers, partition):
    /* Leader tells each worker to begin
       executing                                  */
    futures = [];
    for w ∈ workers do
        future = asynchronously execute w.forward(partition);
        futures.append(future);

    /* As workers return their embeddings, the
       leader processes them in order, as they
       arrive                                     */
    h=NULL;
    zs = [];
    for f ∈ futures do
        z, h = self.RNN(f.wait(), h);
        zs.append(z);

    return concat(zs)
```

For performance optimization, and to ensure consistency, we impose just one constraint for this stage: topological encoders must not be dependent on any temporal information. They provide a purely spatial encoding of the state of each snapshot they hold, using only features observable at a single point in time. With this constraint satisfied, all encoders can operate in parallel, as no one worker is dependent on the output of another. Theoretically, with as many workers as snapshots, the time complexity of a forward pass is constrained only by the snapshot with the greatest number of edges.

Temporal Encoding
The leader maintains an ordered list of future objects that point to the eventual output of the workers and waits for the future pointing to the first embeddings to finish executing. When the leader receives this tensor, it is immediately processed by its RNN, shown in step 3 of Figure 3. Note that so long as tasks are slightly imbalanced such that workers holding later snapshots contain more work units, the leader's recurrent layer can execute concurrently with at most $k-1$ workers' topological encoders. Workers with earlier snapshots hold fewer work units, and therefore finish executing earlier, so the leader can process their outputs while workers holding greater quantities of snapshots further in time are still processing.

When the recurrent layer has finished processing the output of one worker, the hidden state and outputs from the RNN are saved. The leader waits for the next topological embedding to finish processing, then uses the saved RNN hidden state and the next embedding to repeat the process until all workers have finished executing. This procedure is described in more detail in Algorithm 2.

Decoding
When the leader finishes generating the final embeddings, they are sent back to the workers to decode, and if the model is training, to calculate loss. This process occurs in parallel on the worker machines. In general, graph functions such as those used to find edge likelihoods are more compute and memory intensive, so we endeavor to run them in parallel whenever possible. This stage is shown in step 4 of Figure 3.

During evaluation, instead of returning loss, the workers return edge likelihoods $\hat{y}_t$, and the ground-truth edge labels y_t. The process for decoding the embeddings is the same, however loss is not calculated.

Backpropagation & Evaluation
When loss has been calculated and returned to the leader machine, gradients are calculated via backpropagation, first through the recurrent layer, then components of the loss function generated by each topological encoder are backpropagated in parallel and broadcast between workers in accordance with the bucketing algorithm described in [31]. After the backpropagation step, and collective communication between workers, gradients across all workers' model replicas are equal. Finally, the recurrent layer and the topological encoders all update their parameters using an Adam optimizer [34], and the leader repeats steps 2-5 until convergence.

If the model is in evaluation mode, the leader machine instead uses the $\hat{y}_t$ likelihoods the workers generate, and the known labels y_t, also returned by the workers, to calculate precision and accuracy metrics, which are saved. In a real-world implementation without labels, it would instead raise alerts on observed edges with likelihoods below a certain threshold.

C. Training

There are two modes of training EULER models: as a link *detector*, or a link *predictor*. These two modes are distinguished by which $\mathbf{Z}_t$ embeddings are sent to the workers at step 4 to calculate loss. Link detectors are inductive; they generate $\mathbf{Z}_t$ using partially observed snapshots $\{\hat{\mathcal{G}}_0, ..., \hat{\mathcal{G}}_t\}$ and attempt to reconstruct the full adjacency matrix $\mathbf{A}_t$ with $g(\mathbf{Z}_t)$. In practice, they would be used for forensic tasks, where one is performing an audit to identify anomalous connections that have already occurred.

Link predictors are transductive; they generate $\mathbf{Z}_t$ using snapshots $\{\mathcal{G}_0, ..., \mathcal{G}_t\}$, in order to predict the future state, $\mathbf{A}_{t+n}$ where $n > 0$. In practice, they could be used as a live intrusion detection tool, as predictive models can score edges as they are observed–before they have been processed into full snapshots. For example, when $n = 1$, given what has been observed about the network up until time $t-1$, it is the goal of predictive implementation of EULER to score edges observed at time t. Such a model can use embeddings learned from previous states of the network to process connections as they occur.

To ensure these objectives, the reconstruction loss function aims to minimize the negative log likelihood of Equation 3, where $n = 0$ when training detectors, and $n > 0$ for predictors. For larger graphs, operating over the entire adjacency matrix quickly becomes intractable. Instead, we approximate this value by minimizing binary cross entropy on the likelihood scores for known edges, and a random sample of non-edges at time $t+n$: P_{t+n} and N_{t+n}. Formally, the reconstruction loss function for snapshot $t+n$ is

$$\begin{aligned}\mathcal{L}_t &= -\log(\Pr(\mathbf{A}_{t+n} \mid \mathbf{Z}_t)) \\ &\approx \frac{-1}{|P_{t+n}|}\sum_{p \in P_{t+n}} \log(p) + \frac{-1}{|N_{t+n}|}\sum_{n \in N_{t+1}} \log(1-n)\end{aligned} \quad (5)$$

New negative edges are randomly sampled on each epoch by workers for each snapshot they hold. The leader machine coordinates which slices of the $\mathbf{Z}$ tensor are sent to each worker to generate P_{t+n} and N_{t+n}, and each worker independently calculates loss on the training data they hold.

On predictive models, $\mathbf{Z}_t$ represents the latent space of nodes n snapshots in the future. Consequently, predictive models cannot calculate loss on the first n snapshots of the graph. To account for this, the leader pads $\mathbf{Z}$ with a $n\ |\mathcal{V}| \times d$ zero matrices at the beginning, and the final n matrices are removed before returning the embeddings to the workers. This way, $\mathbf{Z}[t]$ predicts the snapshot indexed at t on all workers except the one holding the initial snapshots, which ignores embeddings that equal the zero matrix. This process is shown more clearly in Algorithm 3.

Algorithm 3: Calculate loss

```
def worker_loss(zs, n_jobs, workers, n=1):
    /* Pad the Z tensor if predictive                */
    zeros = $n \times |\mathcal{V}| \times d$ 0 matrix;
    zs = concat(zeros, zs[:-n]);
    futures = [];
    start=0; end=n_jobs[0]; loss = 0; i=0;
    /* Send the correctly offset embeddings to
       the workers to calculate loss                 */
    for $w \in$ workers do
        f = asynchronously execute w.loss(zs[start:end]);
        futures.append(f);
        tmp=start; start=end; end=start+n_jobs[i++];
    return sum([f.wait() for f in futures]) / len(workers)
```

D. Classification

Though for much of our evaluation, we rely on regression metrics relating to the fitness of scores assigned to edges, it is useful to automate the process of deciding the threshold for what counts as anomalous to obtain classification scores. To this end, when training the model, we hold out one or more full snapshots to act as an extra validation set. Using the final hidden state $\mathbf{h}$ of the RNN from the training snapshots as input for the validation snapshot, a training partition of edges is passed through the model. From there, finding an optimal cutoff threshold for edge likelihood scores becomes a simple optimization problem.

Given a set of scores for edges that exist in the validation snapshots, but were held out of the training set, and a set of scores for non-edges, the optimal cutoff threshold τ is the one which satisfies

$$\underset{\tau}{argmin} \quad \|(1-\lambda)TPR(\tau) - \lambda FPR(\tau)\| \tag{6}$$

where $TPR(\tau)$ and $FPR(\tau)$ refer to the true and false positive rate of classification given cutoff threshold τ, and λ is a hyperparameter in $[0-1]$ biasing the model to optimize for either a high true positive rate, or a low false positive rate. Experiments have shown that for anomaly detection, where low false positive rates are critical, $\lambda = 0.6$ is very effective. For any metric involving classification for the remainder of the paper, this is how classes were determined from the edge likelihood scores, and unless otherwise specified $\lambda = 0.6$.

V. Benchmark Evaluations

Prior works [19] and [22] both assert that simply embedding graph snapshots with a GNN, and running these embeddings through an RNN, as was done by [20], [35] does not adequately capture the shifting distribution of nodes in a dynamic network. Prior work [22] demonstrates that this is the case for inferring complete graph structure several time steps in the future, but [19] does not evaluate their model against the very model they so thoroughly disregard. To remedy this, we present a comparison between several existing temporal autoencoder models, and a simple stacked GCN [36] and GRU [37] following the Euler framework. We select these two layers because of their lack of parameters– they are the most generalized of the models in their respective domains [38], [39].

A. Models Tested & Data Sets

In this section, we implement Euler as a graph convolutional network [36], the most general GNN available [38], stacked upon a GRU, an RNN with very few parameters. Though the model is so simple as to be called the “naive method” by [20], it is also the fastest temporal model tested, and as we will show, quite effective. The topological embedding layer is a two-layer GCN, essentially a graph autoencoder. We include an edge dropout layer [40] before the initial forward pass and feature dropout layer between all layers to prevent overfitting and oversmoothing on the small datasets. Its hidden layer and output are both 32-dimensional. The sequence of GCN outputs is then passed through a tanh activation function before they are processed by a single 32-dimensional GRU, and finally a MLP to project the output into 16-dimensional embeddings.

The other methods evaluated are as follows:

DynGraph2Vec [41]: a model which passes adjacency vectors into an multilayer perceptron (MLP) and an RNN to capture graph dynamics using traditional deep learning techniques. The DynAE variant passes adjacency vectors through a deep autoencoder architecture; the DynRNN variant passes them through a recurrent neural network; and DynAERNN passes the output of deep autoencoders to an RNN to generate node embeddings. This latter model fits into the Euler framework if an MLP were used instead of an GNN. We include this model to show the value of message passing networks.

Evolving GCN [22]: a GCN whose parameters are passed through an LSTM [42] or GRU [37] at each timestep. In the EGCN-O variant, the GCN parameters are the input and output of an LSTM; in the EGCN-H variant, the GCN parameters are used as the hidden states of a GRU, which takes the previous embedding as input. When the models were evaluated for temporal link prediction in the original paper, they were given no subset of ground-truth edges from which to generate embeddings. Instead, they used the predictions from the previous snapshot as the adjacency matrix inputs. This is a far more difficult task than our method of link prediction, where a subset of edges is available as a ground truth training set for every snapshot. But our interest is anomaly detection, where such accommodations are always available. Lastly, we note that the original model uses an MLP that takes two nodes’

TABLE I: Data set metadata

Data Set	Nodes	Edges	Avg. Density	Timestamps
FB	663	23,394	0.00591	9
COLAB	315	5,104	0.01284	10
Enron10	184	4,784	0.00514	11

embeddings as input to calculate likelihood scores, but our experiments found using inner product decoding was more effective.

VGRNN [19]: a GCN stacked upon a GC-LSTM [21]. Their method, however, cannot be fit into the EULER framework; this is because node embeddings from the GCNs are passed into the RNN, whose output is concatenated with node features to be used as input for the next snapshot. This process must happen in serial. The hidden state vectors from the RNN are designed to predict the next state of the graph, in addition to providing information for the GCN. Hence, for predictive tasks, a non-linear transformation of the RNN output is used as the node embeddings rather than the direct output of the GCN.

VGAE [32]: A simple two-layer variational graph autoencoder. This model does not consider time at all. Instead, it views every interaction as one large graph. We include this model to demonstrate the usefulness of the recurrent unit in other models. The VGAE acts as a baseline to compare inductive tests against; as it is a static model, it cannot be used for dynamic (new) link prediction.

We use three data sets for these tests: Facebook [43], Enron10 [44], and COLAB [45]. Table I contains more detailed information about these data sets. None of these data sets contain node features, so the identity matrix is used as the initial feature input for all models. These graphs are all directed, and during evaluation self-loops are not considered.

B. Evaluation Metrics

As we use a regression model to output probabilities, we use the following to measure its effectiveness: area under the ROC curve (AUC) and average precision score (AP). The AUC score is defined as the area under the curve created by plotting the true positive rate (TPR) and false positive rates (FPR) as the threshold for classification changes [46].

The TPR and FPR are defined as

$$TPR = \frac{TP}{TP + FN} \qquad FPR = \frac{FP}{FP + TN} \tag{7}$$

Average precision for regression tasks is defined as

$$AP = \sum_{\tau} (R_\tau - R_{\tau-1}) P_\tau \tag{8}$$

where P_τ and R_τ denote the precision and recall scores at threshold τ. Precision and recall are defined as

$$P = \frac{TP}{TP + FP} \qquad R = \frac{TP}{TP + FN} \tag{9}$$

C. Experimental Setup

As was done by [19], we conduct three different benchmarking tests to compare EULER to other temporal link prediction methods: inductive dynamic link detection, transductive dynamic link prediction, and transductive dynamic new link prediction. Link detection and link prediction are implemented as described in Section IV-C. On link detection tests the objective function is Equation 3 with $n = 0$. On (new) link prediction tests, $n = 1$.

For link detection tests, we randomly remove 5 and 10% of the edges from each snapshot for validation and testing respectively. The reported results are evaluations of final 3 snapshots, which are never used for training; however, a training set of edges from those latter timesteps are withheld and not evaluated, to be used as inputs during the forward pass at evaluation.

For link prediction tests, *all* edges in final three snapshots are considered positive samples in the test set, with an equal number of randomly sampled negative edges. As it is a transductive test, during the forward pass all edges at time t are ingested to predict the state at $t + 1$. However, 5% of edges are held out for validation during decoding.

New link prediction is an extra evaluation of predictive models. This test is identical to link prediction, but the set of true positives only includes edges that were not in the snapshot immediately before:

$$\{(u, v) \mid (u, v) \in \mathcal{E}_{t+1} \wedge (u, v) \notin \mathcal{E}_t\} \tag{10}$$

the purpose of this test is to evaluate predictive models' ability to anticipate new edges given what it has observed about the dynamics of the graph.

Model parameters are updated with an Adam optimizer [34] with a learning rate of either 0.03, 0.02, 0.01, or 0.005 according to a parameter search. Models are trained for 1,500 epochs with early stopping after 100 epochs of no progress on the validation data. Results for the (SI-)VGRNN, the DynGraph2Vec models, and the VGAE are those reported by [19]. We use the source code provided by [22] to evaluate the EGCN models.

D. Results

Here, we report the results of the three tests with some discussion. When comparing results from competing methods, ties occur when the difference between the means of two models' metrics, A and B, are not statistically significant. This is determined via a two-tailed t-test with the null hypothesis H_0 that the two means are the same:

$$t = \frac{0 - (\mu(B) - \mu(A))}{\sqrt{\frac{\text{Var}(B-A)}{N}}} = \frac{\mu(A) - \mu(B)}{\sqrt{\sigma_M(A)^2 + \sigma_M(B)^2}} \tag{11}$$

Where N denotes degrees of freedom, and $\sigma_M(\cdot)$ denotes the standard error. If

$$p = 2\min\{\Pr[T \leq t \mid H_0], \Pr[T \geq t \mid H_0]\} > 0.05 \tag{12}$$

the difference between the two means is deemed insignificant, and the two models are considered equivalent. For ease of

TABLE II: Comparison of EULER to related work on dynamic link detection

Metrics	Methods	Enron	COLAB	Facebook
AUC	VGAE	88.26 ± 1.33	70.49 ± 6.46	80.37 ± 0.12
	DynAE	84.06 ± 3.30	66.83 ± 2.62	60.71 ± 1.05
	DynRNN	77.74 ± 5.31	68.01 ± 5.50	69.77 ± 2.01
	DynAERNN	91.71 ± 0.94	77.38 ± 3.84	81.71 ± 1.51
	EGCN-O	93.07 ± 0.77	**90.77 ± 0.39**	86.91 ± 0.51
	EGCN-H	92.29 ± 0.66	87.47 ± 0.91	85.95 ± 0.95
	VGRNN	94.41 ± 0.73	88.67 ± 1.57	88.00 ± 0.57
	SI-VGRNN	95.03 ± 1.07	89.15 ± 1.31	88.12 ± 0.83
	EULER	**97.34 ± 0.41**	**91.89 ± 0.76**	**92.20 ± 0.56**
AP	VGAE	89.95 ± 1.45	73.08 ± 5.70	79.80 ± 0.22
	DynAE	86.30 ± 2.43	67.92 ± 2.43	60.83 ± 0.94
	DynRNN	81.85 ± 4.44	73.12 ± 3.15	70.63 ± 1.75
	DynAERNN	93.16 ± 0.88	83.02 ± 2.59	83.36 ± 1.83
	EGCN-O	92.56 ± 0.99	**91.41 ± 0.33**	84.88 ± 0.52
	EGCN-H	92.56 ± 0.72	88.00 ± 0.85	82.56 ± 0.91
	VGRNN	95.17 ± 0.41	89.74 ± 1.31	87.32 ± 0.60
	SI-VGRNN	**96.31 ± 0.72**	89.90 ± 1.06	87.69 ± 0.92
	EULER	**97.06 ± 0.48**	**92.85 ± 0.88**	**91.74 ± 0.71**

reading, we highlight only ties between the two models with the highest observed means.

Dynamic Link Detection

As shown in Table II, the simplistic EULER model outperforms the more modern ones in almost every test. In tests where it does not outperform the state-of-the-art methods, it is equivalent, despite having fewer parameters. Compared to the static VGAE, which does not consider time at all, the benefit of the additional RNN layer is clear. We further observe the benefit of graph neural networks over MLPs when EULER and the state-of-the-art methods are compared to the DynGraph2Vec methods. However, the experiments do not support claims that much is gained by the complex models beyond what is afforded simply by using a GNN and RNN. Both highly engineered models, the EGCN and VGRNN variants, do not perform significantly better than the simplistic stacked GCN on a GRU, and in some cases perform worse.

Significantly, the data set where EULER performed better than prior works with $p < 0.05$ was the Facebook data set. This data set contains the most nodes and edges and has the fewest snapshots in the training set. Despite these difficulties, our simple EULER model achieves a 4% improvement over prior work in both AUC and AP, signifying its ability to learn very complex spatio-temporal patterns even on larger data sets. We observe that the model is generalized enough not to become overfit on the smallest data sets, but not so simple it cannot handle larger ones. This supports our claim that this model design, despite its simplicity, is highly precise.

Dynamic (New) Link Prediction

In these cases where temporal data is more significant, the results are less clear. As shown in Tables III and IV, between our method and (SI-)VGRNN models, the results are almost all within each methods' margin of error. We note that on the dynamic new link prediction test for Enron10, though our method's observed mean AUC and AP were lower than both VGRNN methods, a t-test showed that this disparity was not statistically significant. We can conclude that neither method is significantly better than the other. Furthermore, we observe that on the Facebook data set, which has roughly the same edge

TABLE III: Comparison of EULER to related work on dynamic link prediction

Metrics	Methods	Enron	COLAB	Facebook
AUC	DynAE	74.22 ± 0.74	63.14 ± 1.30	56.06 ± 0.29
	DynRNN	86.41 ± 1.36	75.7 ± 1.09	73.18 ± 0.60
	DynAERNN	87.43 ± 1.19	76.06 ± 1.08	76.02 ± 0.88
	EGCN-O	84.28 ± 0.87	78.63 ± 2.14	77.31 ± 0.58
	EGCN-H	88.29 ± 0.87	80.80 ± 0.95	75.88 ± 0.32
	VGRNN	93.10 ± 0.57	**85.95 ± 0.49**	89.47 ± 0.37
	SI-VGRNN	**93.93 ± 1.03**	85.45 ± 0.91	**90.94 ± 0.37**
	EULER	**93.15 ± 0.42**	**86.54 ± 0.20**	**90.88 ± 0.12**
AP	DynAE	76.00 ± 0.77	64.02 ± 1.08	56.04 ± 0.37
	DynRNN	85.61 ± 1.46	78.95 ± 1.55	75.88 ± 0.42
	DynAERNN	89.37 ± 1.17	81.84 ± 0.89	78.55 ± 0.73
	EGCN-O	86.55 ± 1.57	81.43 ± 1.69	76.13 ± 0.52
	EGCN-H	89.33 ± 1.25	83.87 ± 0.83	74.34 ± 0.53
	VGRNN	93.29 ± 0.69	87.77 ± 0.79	89.04 ± 0.33
	SI-VGRNN	**94.44 ± 0.85**	**88.36 ± 0.73**	**90.19 ± 0.27**
	EULER	**94.10 ± 0.32**	**89.03 ± 0.08**	**89.98 ± 0.19**

TABLE IV: Comparison of EULER to related work on dynamic new link prediction

Metrics	Methods	Enron	COLAB	Facebook
AUC	DynAE	66.10 ± 0.71	58.14 ± 1.16	54.62 ± 0.22
	DynRNN	83.20 ± 1.01	71.71± 0.73	73.32 ± 0.60
	DynAERNN	83.77 ± 1.65	71.99 ± 1.04	76.35 ± 0.50
	EGCN-O	84.42 ± 0.82	79.06 ± 1.60	75.95 ± 1.15
	EGCN-H	87.00 ± 0.85	78.47 ± 1.27	74.85 ± 0.98
	VGRNN	**88.43 ± 0.75**	77.09 ± 0.23	87.20 ± 0.43
	SI-VGRNN	**88.60 ± 0.95**	**77.95 ± 0.41**	87.74 ± 0.53
	EULER	87.92 ± 0.64	**78.39 ± 0.68**	**89.02 ± 0.09**
AP	DynAE	66.50 ± 1.12	58.82 ± 1.06	54.57 ± 0.20
	DynRNN	80.96 ± 1.37	75.34 ± 0.67	75.52 ± 0.50
	DynAERNN	85.16 ± 1.04	77.68 ± 0.66	78.70 ± 0.44
	EGCN-O	86.92 ± 0.39	81.36 ± 0.85	73.66 ± 1.25
	EGCN-H	86.46 ± 1.42	79.11 ± 2.26	73.43 ± 1.38
	VGRNN	87.57 ± 0.57	79.63 ± 0.94	86.30 ± 0.29
	SI-VGRNN	**87.88 ± 0.84**	**81.26 ± 0.38**	**86.72 ± 0.54**
	EULER	**88.49 ± 0.55**	**81.34 ± 0.62**	**87.54 ± 0.11**

density as Enron, but 3.5x as many nodes, EULER performs significantly better than other methods in new link prediction. As variance and complexity increases in the data, EULER adapts better than the other methods while retaining precision on simpler data without becoming overfit.

In both tests, models which process graph embeddings using an RNN were significantly better than DynAE which does not. This component enables temporal attributes of the data to be carried over from previous time steps. If a new edge has been seen in the distant past, or a pattern that indicates a new edge is likely to appear has previously been observed, this history is carried over by the RNN into future embeddings. From this, we can again infer that the benefit derived from the (SI-)VGRNN models has more to do with the components of those models, which are also GCNs connected with an RNN. The espoused benefit of the topological embedders ingesting temporal information does not appear to be as great as simply using those components; by removing the GNNs' reliance on temporal information, our models can embed at least the same quality of information in a more efficient manner.

With these data, it is clear that the simplicity of models following the EULER framework is not a hindrance, and in many cases is actually advantageous. The purpose of EULER is chiefly to improve efficiency and scalability, so the fact that it is only a small improvement, or about equal to state-of-the-art models is adequate for our purposes. The real benefit of building models within the EULER framework is their ability to scale. On larger data sets, this advantage is more evident.

VI. LATERAL MOVEMENT DETECTION

In the previous section, all datasets tested were rather small. It is not until a real-world application of the EULER framework is tested that the true performance improvements are evident.

To demonstrate the impressive speedup achieved by this framework when compared to related work we evaluate several EULER models on the LANL 2015 Comprehensive Multi-Source Cyber Security Events dataset [24]. The dataset consists of 57 days of log files from five different sources within the Los Alamos National Laboratory's internal corporate network as it underwent both normal activity and a redteam campaign. Specific details about this dataset are reported in Table V. The edge count in the table represents the number of weighted edges; multiple events between the same entities in the same time period may be compressed into a single weighted edge. Because events from the authentication logs have been labeled as normal or anomalous, this data set has been widely used for cyber security research [9], [12], [13], [15]. The labels make it especially apt for lateral movement detection research. When an APT-level threat is attempting to traverse a system, one possible warning sign will be authentications that should not normally occur, a sign indicative of lateral movement on a network level.

TABLE V: LANL Data Set Metadata

Nodes	17,685
Events	45,871,390
Anomalous Edges	518
Duration (Days)	58

In this section, all distributed models were implemented with 4 worker nodes unless otherwise specified. All experiments are run on a server with two Intel Xeon E5-2683 v3 (2.00 GHz) CPUs, each of which has 14 cores with 28 threads, and 512 GB of memory [47].

We will first present the utility of models following the EULER framework as an anomaly-based intrusion detection system on the LANL dataset, then an analysis of the immense scalability afforded by splitting models in this way.

A. Graph Construction

We construct a weighted, directed graph from the authentication logs by mapping which entities authenticate with one another. As nodes, we use the entities denoted source and destination computers in the LANL documentation. For all authentications that occur from time t to $t + \delta$, an edge is created between the source computer and destination computer. If an edge already exists, a tally keeping track of the number of authentications between the two machines is updated. Experiments have shown that the most effective method to normalize these tallies into usable edge weights is to take the logistic sigmoid of the edges' standardized values. Mathematically, it can be represented as

$$W((u,v) \in \mathcal{E}) = \sigma\Big(\frac{C(u,v) - \mu_{\mathcal{E}}}{\Sigma_{\mathcal{E}}}\Big) \tag{13}$$

where $\sigma(\cdot)$ represents the sigmoid function, $C(u,v)$ represents the frequency of an authentication between u and v in the time window, and $\mu_{\mathcal{E}}$ and $\Sigma_{\mathcal{E}}$ represent the mean and standard deviation of all edge frequencies in the time window. In this way, edges which occur very infrequently are given lower weight during training so as to appear less "normal" and edges which occur with high frequency, such as edges from computers to domain controllers, or ticket granting servers have high weight, and appear routine.

The LANL dataset has no node features by default, however some information can be gleaned from the naming convention used in the log files. Entities have unique, anonymized identifiers that start with either a `U` or a `C` denoting users and computers respectively. There are also nodes with non-anonymized names that have important roles in the system such as `TGT`, the Kerberos key distribution center, `DC`, the domain controller, and so on. To leverage this additional data, we concatenate a 1-hot vector denoting user, computer, or special administrative machine to each node's one-hot ID vector.

For quicker file scanning, and data loading times, the full 69 GB `auth.txt` file is split into chunks, which each hold 10,000 seconds (approximately 3 hours) of logs. Worker machines are issued instructions to read in certain ranges of the log files and build the temporal graphs. Workers accomplish this by spawning several child processes to load multiple snapshots in parallel. The associated edge lists, and edge weight lists from each child process are combined to form the final `TGraph` object, which holds a list of all edge lists, edge weights, node features, and tensor masks to take partitions of each edge list for training.

For all experiments, the training set consists of all snapshots that occur before the first anomalous edge appears in the authentication logs. This allows models to learn what normal activity looks like. From this set, we remove the final 5% of snapshots for tuning the classifier, and mask 5% of edges from each snapshot for validation.

B. Experimental Setup

We test three encoders in conjunction with two recurrent neural networks as well as models with no recurrent layer to measure how much value temporal data adds to the overall embeddings. The encoder models are GCN [36], GAT [48], and GraphSAGE [49]. The recurrent models are GRU [37] and LSTM [42]. The models are trained in the same manner as the link detection and link prediction models in Section V. However, experiments showed that once a local optimum was found and validation scores ceased improving, it rarely improved after further iterations. As such, early stopping occurs after only ten epochs of no improvement.

Experiments showed for every model that using smaller time windows always lead to better results. As such, we only present the output of tests on temporal graphs with time window $\delta = 1800$ seconds (30 minutes).

The GAT encoder uses 3 attention heads, which was found to be optimal via hyperparameter tuning. The SAGE encoder uses maxpooling as its aggregation function, as this was found to be optimal in their paper, and this makes it capable of discerning between certain graphs GCN cannot [50].

Unfortunately, many other works which experiment with the LANL dataset either do not use the data set in full, as is the case with [12], [13] or conduct tests on portions of the data other than purely the authentication logs, as was done by [15], so it would not be fair or meaningful to compare our results to theirs. Prior work [9] does use the full authentication log as its dataset, however it trains on a larger set of data, using all days that contained no anomalous activity as the training set, rather than just the days before the attack campaign. We include their results, nonetheless. We also include the TPR and FPR of a rules-based "Unknown Authentication" (UA) model reported by [9]. This rule simply marks any edge that did not exist in the training data as anomalous.

By default, VGRNN operates on full adjacency matrices, however we modified it to use sparse edge lists for our experiments. This way it was able to scale to the large size of the LANL data set. Unfortunately, the E-GCN and DynGraph2Vec models could not scale to the LANL data set. DynGraph2Vec relies on dense adjacency matrices, and the size of the 1-hot vectors used as inputs was too large for E-GCN to process. As a result, our hardware was unable to fully evaluate these methods and their results are not compared to those of EULER.

All models evaluated use 32-dimensional hidden layers, and 16-dimensional embeddings. All EULER models use a tanh activation function between the encoder and the recurrent layers and an edge dropout layer before the GNNs. They all determine the classification threshold according to Equation 6 with $\lambda = 0.6$, except the GraphSAGE models. For this encoder, experiments showed $\lambda = 0.5$ was more appropriate. All reported results are average scores from 5 independent tests on link detection and link prediction.

C. Anomalous Edge Detection

It is difficult to properly evaluate methods for classifying imbalanced data, especially anomaly detection, where small false positive rates are so critical. For this reason, in addition to the raw true positive and false positive rates, we report precision (P), area under the curve (AUC) and average precision (AP). This latter method is recommended for anomaly detection by [51] as especially adept for imbalanced datasets. The AUC and AP metrics evaluate the overall quality of scores given to edges, as opposed to the quality of classification, and provide better measurements of the model if the anomalous score threshold were to be changed. The precision metric provides further context to the quality of classification at the specific threshold. The average results of 5 experiments are shown in Table VI.

As a baseline, consider the TPR of the UA rules-based approach. This implies that 28% of anomalous connections are those which have occurred before in the network. This supports our claim that temporal information about the context of connections is just as important as the entities that are authenticating. This system is an excellent baseline model to compare to, as any model that has a higher TPR than UA must be using a more advanced metric than simply memorizing every legitimate connection observed in normal activity. If a model has a TPR above 72% with a FPR lower than 4.4%, it must be leveraging topological or temporal context judge the validity of connections.

TABLE VI: Performance of EULER Models on the LANL Dataset when $\delta = 0.5$

Dynamic Link Detection

Encoder	RNN	AUC	AP	TPR	FPR	P
GCN	GRU	0.9912	**0.0523**	86.10	0.5698	0.0054
	LSTM	0.9913	0.0169	89.65	0.5723	0.0056
	None	**0.9916**	0.0116	88.57	**0.4798**	**0.0066**
SAGE	GRU	0.9872	0.0307	84.71	0.6874	0.0044
	LSTM	0.9887	0.0389	83.55	0.6591	0.0045
	None	0.8652	0.0052	79.58	24.5669	0.0001
GAT	GRU	0.9094	0.0076	85.21	21.533	0.0001
	LSTM	0.8713	0.0022	96.83	19.873	0.0002
	None	0.9867	0.0079	**99.88**	23.174	0.0002
UA		–	–	72.00	4.400	0.0010
GL-LV [9]		–	–	67.00	1.200	0.0034
GL-GV [9]		–	–	85.00	0.900	0.0051
VGRNN		0.9315	0.0000	59.69	4.938	0.0000

Dynamic Link Prediction

Encoder	RNN	AUC	AP	TPR	FPR	P
GCN	GRU	**0.9906**	0.0155	85.49	0.6088	0.0050
	LSTM	0.9885	0.0166	78.91	0.5987	0.0047
	None	0.9902	0.0092	86.42	0.5425	**0.0057**
SAGE	GRU	0.9847	0.0200	86.30	1.6542	0.0019
	LSTM	0.9865	**0.0228**	85.29	0.8037	0.0038
	None	0.9284	0.0020	86.23	16.525	0.0002
GAT	GRU	0.8826	0.0020	87.82	21.971	0.0001
	LSTM	0.8383	0.0002	83.42	29.297	0.0001
	None	0.9352	0.0079	**88.83**	20.093	0.0002
VGRNN		0.9503	0.0004	70.00	**0.280**	0.0004

The results show that the GCN is the most effective encoder for link detection, and SAGE is most effective for link prediction; this supports our claim and that of [38] that more generalized models are more effective. The GAT models, which have 3x as many parameters as GCN and SAGE performed quite poorly both in quality of scores, and quality of classification.

Also worth noting is the way using a RNN affects the output. Surprisingly, the best AUC in the link detection tests were from a GCN with no temporal encoder. However, this metric is not a good indicator of model quality on data sets with imbalance as extreme as LANL. The dramatically higher AP score of all models which use RNNs suggests temporal data strongly affects FPR and cannot be ignored. Similarly, the models without an RNN have high precision on the GCN models. However, again, the AP scores would indicate that while at this specific threshold omitting the RNN is beneficial, over all thresholds, models which take time into account perform better.

In the more realistic transductive link prediction tests, though the difference between the two RNNs tested is small in every case, the benefit they add is unquestionable. The best performing encoder, GraphSAGE enjoyed a 10x improvement in AP when used in conjunction with any RNN. The next best performing encoder, GCN achieved a 1.6x improvement in AP. This is evidence that temporal information carries important context for the topological state of the network, particularly for filtering false positives. Where one authentication may appear anomalous in isolation, when viewed in the context of previous authentications, it can be correctly identified as benign.

The GL-LV and GL-GV method reported in [9] does not consider time at all; the network is viewed as a static graph. Here, we again see the benefit of using a sequence encoder in conjunction with a pure topological embedder. The best EULER methods outperform their random walk-based approach in terms of both TPR and FPR. Also worth noting is that because our model uses temporal graphs, the alerts from EULER-based models come with a time stamp, making them more informative and valuable in a real-world scenario. The prior work ranks any duplicate edges, regardless of their temporal context, as equally anomalous.

Like EULER, VGRNN combines a sequence encoder with a temporal one. They claim that by using temporal information as input during the topological encoding phase, complex temporal dependencies are better encoded than without it. However, this method performs no better than the purely statistical UA method. Even still, the false positive rate is excellent in the predictive test, outperforming every EULER model. It is worth noting however, the quality of likelihood scores is very poor with this method, which implies that had the threshold for the EULER models been set lower their FPR would be lower with equivalent TPRs. This is readily apparent in the GCN-based models where the FPR is only a few tenths of a percent larger than the dynamic VGRNN, but their TPRs are almost 10% higher.

Finally, we must concede that while the EULER models do outperform prior works, their FPRs are still too high to be useful as an intrusion detection system on their own. Some of this can be attributed to the data set itself; labeled anomalous events are very coarse-grained. There are likely many events the compromised entities engaged in that should be considered anomalous, and may have even been detected by our models, but that are treated as false positives due to the lack of fine-grained label information. Indeed, the redlog only tracks “compromise events” and not the further malicious activity that ensues [24].

However, even if this is not fully the case, this method has great potential as a filtering device for further analysis tools. The low cost of processing time, that we will demonstrate later on, makes this an efficient way to minimize the number of interactions that need to be analyzed by a signature-based technique for example. As part of future work, we plan to work with analysts to answer the question of whether this approach is best used as a step in a longer pipeline, or on its own. However, we have demonstrated that compared to all other anomaly-based works which analyze this data set, EULER models are the most effective.

D. Parameter Analysis

Time window size, the hyperparameter δ, has significant tradeoffs associated with it. As this value decreases, the number of edges increases, requiring more processing time. As this value increases, more repeat edges are condensed into single, weighted edges; additionally, there will be fewer discrete time steps for the recurrent networks to process, all contributing to faster processing time at the expense of less precise data.

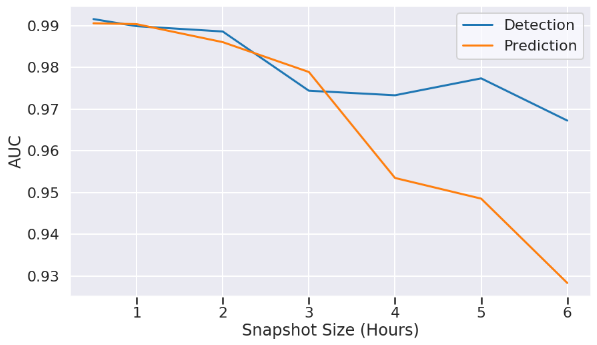


Fig. 4: Change in AUC score as δ increases for link prediction and detection with GCN+GRU models. Scores are the average of five tests on the LANL dataset.

With more edges, and more temporal granularity, models are more capable of learning useful patterns across time. Figure 4 illustrates this phenomenon.

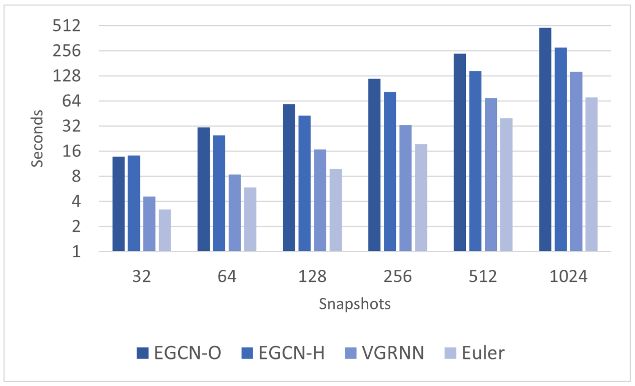
As is evident from the figure, small window sizes are more optimal for quality of scores; however, due to the longer processing time required, and the relatively small performance improvements as window size decreases after a certain point, it may be adequate to leave the snapshot duration a little higher than is optimal for faster training and evaluation. Additionally, we observe that changes in δ seem to affect the link prediction model at a higher rate than the link detection one. We suspect this is caused by the model’s inability to predict short term temporal patterns like the one described in section III as δ grows, and the predictive model is especially apt to detect this type of anomaly.

Nonetheless, in both cases, more granular graphs lead to more informative edge scores. We speculate that at some point, having time slices too granular would have diminishing returns as graphs will have too few edges to be useful. But due to the severe training time as δ decreases, we have never managed to reach this point. We leave finding this boundary as an area for future work.

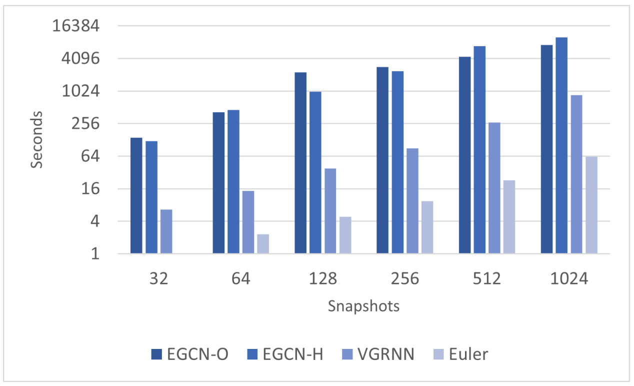
E. Scalability Analysis

The main benefit of using models which fit into the EULER framework is their scalability. While evaluation metrics on benchmarks were generally better than prior work, there were certainly some categories where the advanced models are comparable our simple ones. However, as we will show, distributed topological encoding has tremendous performance benefits.

In Figure 5 we compare the best runtimes of our method to the runtimes of the serial GNN models we used for benchmarking in Section V. The serial methods were allowed 16 threads for intraprocess communication for a fair comparison to our 16 worker processes. However, even with this advantage, these methods are forced to process time steps one at a time; they simply cannot compete with the efficiency of EULER, especially as the size of data increases. Figure 5a shows that as the size of the data being processed increases,



(a) Forward propagation time



(b) Backward propagation time

Fig. 5: Performance comparison between the distributed EULER model and competing methods on varying numbers of snapshots from the LANL dataset. All time windows, δ are 0.5 hours.

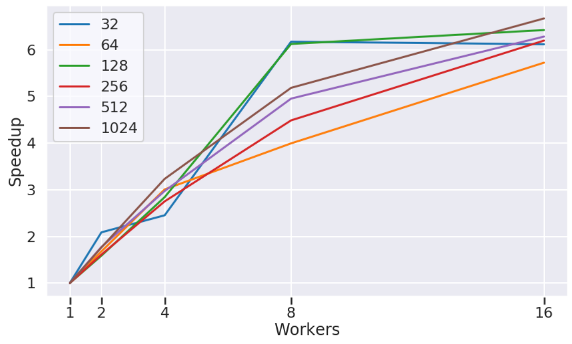


Fig. 6: Performance improvements as more workers are added for varying numbers of snapshots from the LANL data set. The model used was a GCN stacked on a GRU. All time windows, δ are 0.5 hours.

EULER's forward propagation speed is 2x that of the fastest competing algorithm. Additionally, by implementing EULER using `DDP` [31], backpropagation is sped up dramatically. Figure 5b shows EULER has almost a 16x improvement in backpropagation speed, suggesting backpropagation has near linear scaling as workers are added.

Figure 6 shows the speedup of a GCN stacked upon a GRU built within the framework of EULER as more workers are added. For these experiments, we evaluate only workers that are powers of two to ensure each worker holds the same number of snapshots. As each worker requires two threads to run, one for the model replica, and one for collective communication, our equipment can only accommodate up to 16 workers. The framework allows for the number of workers to be a user-defined hyperparameter, so it is effortless to distribute work across a network with enough nodes to support it. It is evident from the chart that performance improvements are immediate. As more workers are added runtime improves rapidly, with negligible improvement after about 8 workers for smaller amounts of data. However, as the amount of data processed increases, performance improvements diminish at a much slower rate. This is because the topological encoding task, the bulk of which must occur on CPU due to the high number of random accesses, scales perfectly with additional workers.

VII. RELATED WORK

We now compare EULER to related work in both intrusion detection, and temporal link prediction generally.

Intrusion Detection Systems

Frequency-based models define normalcy through observed distributions of frequency counts, or other stochastic processes present in the network [12], [13]. Midas [25] does incorporate structural data with frequency counts; however, it can only detect bursts of anomalous events, and would struggle to identify individually anomalous connections. Supervised learning approaches such as [26], [27], [28] analyze only features of events with data mining approaches. Like the EULER approach, many unsupervised systems use deep autoencoders. However, they embed event features rather than network interactions [14], [15]. Kitsune [10] uses temporal patterns in addition to event features; however temporal information is used as an input feature rather than for sequence encoding.

The primary focus of research using graphs for intrusion detection is subgraph matching to detect known malicious patterns in provenance graphs [3], [5], [6], [7], [8]–signature-based approaches. Network-level anomaly-based intrusion detection systems in the field of graph analytics are lacking. Only prior work [9] has proposed a method for detecting anomalous events in host logs that leverages the rich graph structure inherent to the medium. This approach only considers the network as a static graph, so edge embeddings that occurred at different points in time always have the same anomaly score.

Temporal Link Prediction

Temporal link prediction has been used in fraud detection [19], contact tracing [21], and traffic prediction [35]. To our knowledge, however, EULER is the first system to make use of this technique for anomaly-based intrusion detection. Earlier approaches to temporal link prediction used one-hot encodings of node neighbors as features and processed them using MLPs [41], [52], [53], [54], [55]. However, as we and others have shown, these methods are not as powerful as those which leverage GNNs. Many models that incorporate GNNs for temporal graph embedding often do so by replacing the linear layers in RNNs with GNNs [20], [21]; other existing approaches such as VGRNN [19] and E-GCN [22] use highly engineered models, in addition to graph neural network RNNs, and must be run in serial.

VIII. Conclusion

In this work, we presented the Euler framework: a method to exploit the previously untapped potential for distributing the work done to train and execute temporal graph link predictors. When each topological encoder can operate independently, the most compute-heavy task of generating node embeddings can be scaled in a highly efficient manner. Though models following this framework are necessarily simple, we have shown that for anomalous link detection and prediction, models following the Euler framework perform as well, or better than their complex contemporaries. Finally, we showed how this framework can be used to train highly precise anomaly-based intrusion detection systems when network activity is viewed through the abstraction of a temporal graph. These intrusion detection systems are scalable, and more sound than other unsupervised techniques despite being trained with less data.

Future work may include testing other topological or temporal encoders that we did not. Further improvements could be made to the classifier by testing edge decoding techniques more advanced than the simple inner product logistic regression we use. As this framework allows for scalable message passing graph neural networks, future work could even include testing this technique on any large temporal graph data set previously thought intractable for GNNs.

Acknowledgment

The authors thank the anonymous reviewers for their helpful suggestions. This work was supported in part by DARPA under agreement number N66001-18-C-4033 and National Science Foundation grants 1618706, 1717774, and 2127207. The views, opinions, and/or findings expressed in this material are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense, National Science Foundation, or the U.S. Government.

References

[1] B. E. Strom, A. Applebaum, D. P. Miller, K. C. Nickels, A. G. Pennington, and C. B. Thomas, "Mitre att&ck: Design and philosophy," *Technical report*, 2018.

[2] "Lateral movement, tactic TA0008," Oct 2018. [Online]. Available: https://attack.mitre.org/tactics/TA0008/

[3] S. T. King and P. M. Chen, "Backtracking intrusions," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 223–236.

[4] B. Caswell and J. Beale, *Snort 2.1 intrusion detection.* Elsevier, 2004.

[5] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "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, pp. 1795–1812.

[6] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: real-time apt detection through correlation of suspicious information flows," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1137–1152.

[7] S. Wang and S. Y. Philip, "Heterogeneous graph matching networks: Application to unknown malware detection," pp. 5401–5408, 2019.

[8] R. Wei, L. Cai, A. Yu, and D. Meng, "Deephunter: A graph neural network based approach for robust cyber threat hunting," *arXiv preprint arXiv:2104.09806*, 2021.

[9] B. Bowman, C. Laprade, Y. Ji, and H. H. Huang, "Detecting lateral movement in enterprise computer networks with unsupervised graph {AI}," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, 2020, pp. 257–268.

[10] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: An ensemble of autoencoders for online network intrusion detection," *machine learning*, vol. 5, p. 2.

[11] I. H. Sarker, "Cyberlearning: Effectiveness analysis of machine learning security modeling to detect cyber-anomalies and multi-attacks," *Internet of Things*, p. 100393, 2021.

[12] N. Heard and P. Rubin-Delanchy, "Network-wide anomaly detection via the dirichlet process," in *2016 IEEE Conference on Intelligence and Security Informatics (ISI)*. IEEE, 2016, pp. 220–224.

[13] M. Whitehouse, M. Evangelou, and N. M. Adams, "Activity-based temporal anomaly detection in enterprise-cyber security," in *2016 IEEE Conference on Intelligence and Security Informatics (ISI)*. IEEE, 2016, pp. 248–250.

[14] F. Farahnakian and J. Heikkonen, "A deep auto-encoder based approach for intrusion detection system," in *2018 20th International Conference on Advanced Communication Technology (ICACT)*. IEEE, 2018, pp. 178–183.

[15] T. Bai, H. Bian, A. Abou Daya, M. A. Salahuddin, N. Limam, and R. Boutaba, "A machine learning approach for rdp-based lateral movement detection," in *2019 IEEE 44th Conference on Local Computer Networks (LCN)*. IEEE, 2019, pp. 242–245.

[16] "Use alternate authentication material: Pass the ticket," May 2017. [Online]. Available: https://attack.mitre.org/techniques/T1550/003/

[17] "Valid accounts," May 2017. [Online]. Available: https://attack.mitre.org/techniques/T1078/

[18] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Communications surveys & tutorials*, vol. 18, no. 2, pp. 1153–1176, 2015.

[19] E. Hajiramezanali, A. Hasanzadeh, K. Narayanan, N. Duffield, M. Zhou, and X. Qian, "Variational graph recurrent neural networks," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019, pp. 10 701–10 711. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/a6b8deb7798e7532ade2a8934477d3ce-Paper.pdf

[20] Y. Seo, M. Defferrard, P. Vandergheynst, and X. Bresson, "Structured sequence modeling with graph convolutional recurrent networks," in *International Conference on Neural Information Processing*. Springer, 2018, pp. 362–373.

[21] J. Chen, X. Xu, Y. Wu, and H. Zheng, "Gc-lstm: Graph convolution embedded lstm for dynamic link prediction," *arXiv preprint arXiv:1812.04206*, 2018.

[22] A. Pareja, G. Domeniconi, J. Chen, T. Ma, T. Suzumura, H. Kanezashi, T. Kaler, T. Schardl, and C. Leiserson, "Evolvegcn: Evolving graph convolutional networks for dynamic graphs," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, 2020, pp. 5363–5370.

[23] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.

[24] A. D. Kent, "Comprehensive, multi-source cyber-security events data set," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2015.

[25] S. Bhatia, B. Hooi, M. Yoon, K. Shin, and C. Faloutsos, "Midas: Microcluster-based detector of anomalies in edge streams," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, 2020, pp. 3242–3249.

[26] L. Dhanabal and S. Shantharajah, "A study on nsl-kdd dataset for intrusion detection system based on classification algorithms," *International journal of advanced research in computer and communication engineering*, vol. 4, no. 6, pp. 446–452, 2015.

[27] A. Alazab, M. Hobbs, J. Abawajy, and M. Alazab, "Using feature selection for intrusion detection system," in *2012 international symposium on communications and information technologies (ISCIT)*. IEEE, 2012, pp. 296–301.

[28] R. Lohiya and A. Thakkar, "Intrusion detection using deep neural network with antirectifier layer," in *Applied Soft Computing and Communication Networks*. Springer, 2021, pp. 89–105.

[29] "Kdd cup 1999," Oct 1999. [Online]. Available: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

[30] G. Mohi-ud din, "Nsl-kdd," 2018. [Online]. Available: https://dx.doi.org/10.21227/425a-3e55

[31] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *arXiv preprint arXiv:2006.15704*, 2020.

[32] T. N. Kipf and M. Welling, "Variational graph auto-encoders," *arXiv preprint arXiv:1611.07308*, 2016.

[33] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 855–864.

[34] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[35] L. Zhao, Y. Song, C. Zhang, Y. Liu, P. Wang, T. Lin, M. Deng, and H. Li, "T-gcn: A temporal graph convolutional network for traffic prediction," *IEEE Transactions on Intelligent Transportation Systems*, vol. 21, no. 9, pp. 3848–3858, 2019.

[36] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[37] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *arXiv preprint arXiv:1409.1259*, 2014.

[38] O. Shchur, M. Mumme, A. Bojchevski, and S. Günnemann, "Pitfalls of graph neural network evaluation," *arXiv preprint arXiv:1811.05868*, 2018.

[39] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[40] Y. Rong, W. Huang, T. Xu, and J. Huang, "Dropedge: Towards deep graph convolutional networks on node classification," *arXiv preprint arXiv:1907.10903*, 2019.

[41] P. Goyal, S. R. Chhetri, and A. Canedo, "dyngraph2vec: Capturing network dynamics using dynamic graph representation learning," *Knowledge-Based Systems*, vol. 187, p. 104816, 2020.

[42] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[43] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the evolution of user interaction in facebook," in *Proceedings of the 2nd ACM workshop on Online social networks*, 2009, pp. 37–42.

[44] C. E. Priebe, J. M. Conroy, D. J. Marchette, and Y. Park, "Scan statistics on enron graphs," *Computational & Mathematical Organization Theory*, vol. 11, no. 3, pp. 229–247, 2005.

[45] M. Rahman and M. Al Hasan, "Link prediction in dynamic networks using graphlet," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2016, pp. 394–409.

[46] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to data mining*. Pearson Education India, 2016, ch. 5.

[47] "Intel xeon processor e5-2683 v3 (35m cache, 2.00 ghz) product specifications," 2014. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/81055/intel-xeon-processor-e5-2683-v3-35m-cache-2-00-ghz.html

[48] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[49] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *arXiv preprint arXiv:1706.02216*, 2017.

[50] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[51] A. Divakaran and A. Mohan, "Temporal link prediction: a survey," *New Generation Computing*, pp. 1–46, 2019.

[52] L. Zhou, Y. Yang, X. Ren, F. Wu, and Y. Zhuang, "Dynamic network embedding by modeling triadic closure process," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[53] M. Rahman, T. K. Saha, M. A. Hasan, K. S. Xu, and C. K. Reddy, "Dylink2vec: Effective feature representation for link prediction in dynamic networks," *arXiv preprint arXiv:1804.05755*, 2018.

[54] P. Goyal, N. Kamra, X. He, and Y. Liu, "Dyngem: Deep embedding method for dynamic graphs," *arXiv preprint arXiv:1805.11273*, 2018.

[55] T. Li, J. Zhang, S. Y. Philip, Y. Zhang, and Y. Yan, "Deep dynamic network embedding for link prediction," *IEEE Access*, vol. 6, pp. 29 219–29 230, 2018.

TABLE VII: The EULER Interface

Encoding Layer	
`forward(partition: int) -> Tensor`	Takes a subset of edges hosted on the worker machine denoted by the `partition` enum which maps to a mask tensor held on the worker, and passes them through a model; for the purposes of this paper it is assumed to be a GNN. It returns a $T_w \times \|\mathcal{V}\| \times h$ or $\|\mathcal{V}\| \times T_w \times h$ tensor, depending on how one wishes to implement the RNN, where T_w is the number of snapshots held on worker w, and h is a user specified hidden dimension.
`calc_loss(zs: Tensor, partition: int, nratio: float) -> Tensor`	Samples some subset of edges held by the worker, and uses the embeddings, `zs` to calculate loss. In general, we use binary cross entropy loss (BCE) on the training set of known edges, and a random sample of $\|\mathcal{E}\| \times$ `nratio` non-edges at each training step. Returns a 1×1 tensor of the total loss generated from the edges held in the worker
`score_edges(zs: Tensor) -> Sequence[Tensor]`	Returns the likelihood score of each edge held by the worker given the input tensor of node embeddings. This function is used for evaluation to detect anomalous connections, and to calculate AUC scores. It returns a T_w length list of $\|\mathcal{E}_t\| \times 1$ tensors containing likelihood scores.
`load_new_data(fn: Callable[..., TGraph], *args, **kwargs) -> None`	A function to load new data into a worker, called by the constructor by default. The loader should have a return type `TGraph`, a special class to hold temporal graphs.
Recurrent Layer	
`forward(partition: int) -> Tensor`	Issues a command to each worker to run their `forward` method, and waits for their response. As responses come in, in order, it passes output from workers into an RNN. Returns the $T \times \|\mathcal{V}\| \times d$ or $\|\mathcal{V}\| \times T \times d$ tensor, depending on how one wishes to implement the RNN, where T is the total number of snapshots held across all workers and d is the user specified embedding dimension.
`calc_loss(zs: Tensor, partition: int, nratio: float) -> Tensor`	Splits the tensor of embeddings `zs` into contiguous slices such that $z[t]$ is the parameter for the likelihood of edges in snapshot t in the worker it is sent to. It then passes those embeddings to the workers and issues a call for them to calculate loss on them. When the workers have finished calculating loss, the leader aggregates the totals and returns a 1×1 tensor of total loss across all workers.
`score_edges(zs: Tensor) -> Sequence[Tensor]`	Splits the tensor of embeddings `zs` into contiguous slices in the same manner as `calc_loss`, then passes those embeddings to the workers, and issues a call for them to return the likelihood score of each edge they hold. It aggregates each list of edge likelihoods returned by the workers, and returns a T length list of $\|\mathcal{E}_t\| \times 1$ tensors of likelihood scores.

APPENDIX A
IMPLEMENTATION DETAILS

Here we provide the technical details of the EULER framework. Every required method for the leader and follower interfaces is detailed in Table VII.

The Topological Encoder is a replicated GNN across several worker computers. This can be accomplished in a number of ways, but in our implementation we use the PyTorch `DDP` wrapper class around classes which implement the topological encoder interface. This interface requires the following methods: `forward`, `calc_loss`, `score_edges`, and `load_new_data`. Each encoder also has a field containing a `TGraph` object. This object holds a list of edges at each snapshot, the weights of those edges, and node features at each time. When workers are constructed, they load this data from memory using the `load_new_data` method.

The Recurrent Layer is a model-agnostic RNN held on a single leader machine which coordinates the actions of the workers. Users must implement the following methods: `forward`, `calc_loss`, and `score_edges`, which are explained in detail in Table VII.