EdgeTorrent: Real-time Temporal Graph Representations for Intrusion Detection

Isaiah J. King
The George Washington University
Washington, DC, USA
iking5@gwu.edu

Xiaokui Shu
IBM Research
Yorktown Heights, NY, USA
xiaokui.shu@ibm.com

Jiyong Jang
IBM Research
Yorktown Heights, NY, USA
jjang@us.ibm.com

Kevin Eykholt
IBM Research
Yorktown Heights, NY, USA
kheykholt@ibm.com

Taesung Lee
IBM Research
Yorktown Heights, NY, USA
TaesungLee@ibm.com

H. Howie Huang
The George Washington University
Washington, DC, USA
howie@gwu.edu

ABSTRACT

Anomaly-based intrusion detection aims to learn the normal behaviors of a system and detect activity that deviates from it. One of the best ways to represent the behavior of a computer network is through provenance graphs: dynamic networks of entity interactions over time. When provenance graphs deviate from their normal behaviors, it could be indicative of a malicious actor attempting to compromise the network. However, efficiently characterizing the normal behavior of large temporal graphs is challenging. To do this, we propose EdgeTorrent, an end-to-end anomaly-based intrusion detection system for provenance graph analysis. EdgeTorrent leverages a novel high-performance message passing neural network for graph embedding over a stream of edges to capture both temporal and topological changes in the system. These embeddings are then processed by a novel adversarially trained sequence analyzer that alerts when a series of graph embeddings changes in an unexpected way. EdgeTorrent preserves temporal ordering during message passing, and its streaming-focused design allows users to conduct out-of-core inference on billion-edge graphs, faster than real-time. We show that our method outperforms state-of-the-art graph-kernel approaches on several host monitoring data sets; notably, it is the first intrusion detection system to perfectly classify the StreamSpot data set. Additionally, we show it is the best-performing method on a real-world, billion-edge data set encompassing 11 days of benign and attack data.

CCS CONCEPTS

• **Security and privacy** → **Intrusion detection systems**; • **Computing methodologies** → *Anomaly detection*; Neural networks.

KEYWORDS

Provenance Graph Analysis, Intrusion Detection System, Graph Kernel

RAID '23, October 16–18, 2023, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-0765-0/23/10…$15.00
https://doi.org/10.1145/3607199.3607201

ACM Reference Format:
Isaiah J. King, Xiaokui Shu, Jiyong Jang, Kevin Eykholt, Taesung Lee, and H. Howie Huang. 2023. EdgeTorrent: Real-time Temporal Graph Representations for Intrusion Detection . In *The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23), October 16–18, 2023, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3607199.3607201

1 INTRODUCTION

Provenance graph analysis has become an increasingly popular way of host monitoring for advanced persistent threats. Because these data structures capture not just individual events, but how those events flow to other entities in a system, they are the logical choice for auditing a compromise [**?**], defining threat signatures [**? ? ? ? ?**], and, more recently, as intrusion detection systems [**? ? ? ?**]. It has been argued that provenance graphs are the ideal data structure for system monitoring because intrusions often manifest as unexpected interactions between entities on hosts and within networks [**?**]. However, quantifying an abnormal interaction is a non-trivial problem.

One approach to this is time-sensitive graph sketching, also called a graph-kernel. This technique aims to find a function that maps a graph at a certain point in time to a vector, called a graph sketch. If two graph sketches are similar, it implies that the graphs used to generate them are also similar. This approach is employed by Unicorn[**?**] and StreamSpot [**?**] in a streaming setting. They read a list of edges for a given provenance graph and use new edges to update the sketch vector over time. This is accomplished by representing each node through a function of its local neighborhood, then aggregating these node representations to produce a full graph sketch. This evolving graph sketch is periodically saved, and the sequence of sketches is used for cluster analysis. Normal behavior is now defined as graph sketches that fit into clusters of previously observed benign sketches.

Compared to other temporal graph embedding approaches [**? ?**], streaming graph-kernel techniques are the most realistic for real-time detection. Because these methods are specifically optimized for a streaming setting, they are suitable to handle the high volumes of data prevalent in this domain. Furthermore, the series of graph sketches they produce can be analyzed as a sequence rather than as singular data points, which provides deeper insight into the evolving state of the system and enables the detection of temporal

anomalies. Unlike snapshotting techniques [**?** **?**], where graph representations through time are isolated from each other and analyzed independently, graph sketches represent the state of the graph from the beginning of time to the present. They do not partition the graph into discrete chunks for analysis; they consider the way new events change the complete structure.

Current state-of-the-art graph-kernel approaches use a hash function to generate node representations. The list of nodes and edges representing a given node's neighborhood is hashed, and these hashes are aggregated to generate sketches. Unfortunately, use of a hash function make the sketch vectors sensitive to minor perturbations and also less semantically expressive. These issues make temporal anomaly detection difficult, as the series of sketches fluctuates to different points in the embedding space even as the graph remains in a similar state over time. The sketches would be more amenable to advanced sequence analysis if the trajectory of sketches moved smoothly until a consequential state change occurred. Another shortcoming of the prior works is their reliance on clustering for anomaly detection. While it is impressive that non-parametric models can extract as much information from these methods as they do, we will show that using more advanced models for anomaly detection enables the detection of more varied attack types across a broader domain of provenance graphs.

To address these shortcomings, we propose EdgeTorrent, a continuous-time approach to streaming temporal graph representations for intrusion detection. Like [**?**] and [**?**] we employ the graph-kernel technique during the sketching phase. However, rather than relying on a hash function, we propose a novel streaming implementation of a message passing neural network (MPNN) to generate node embeddings. Prior works avoided MPNNs due to the intractable size of provenance graphs [**?** **?** **?** **?** **?**]. Those that do use MPNNs either use them on smaller subgraphs [**?** **?**] or are untested on real-world graphs with billions of edges [**?** **?**]. Despite this, MPNNs have a unique advantage: they are a weak form of the WL-isomorphism test [**?**]. This means they have the expressive power to describe graph isomorphisms. While the graph sketching prior works also represent weak forms of this test, they differ in that their representations are not continuous due to the hashing function. By using an MPNN, our method produces smooth transitions between graph states over time, more amenable to powerful series analysis models.

As we will show, on simpler data sets, the MPNN-generated sketches of EdgeTorrent fall into clear, separable clusters. However, on larger, and more complex data, classifying any one data point in isolation is challenging. Instead, it is critical to view sketches in terms of their trajectories, as a sequence over time. Identifying anomalies within a time series is a challenging task, beyond what a clustering algorithm can handle. So, to capture these complex dynamics, we devise a novel, adversarially trained, transformer-based anomaly detection model by extending the work of FenceGAN [**?**]–a framework specifically designed for unsupervised anomaly detection–for use on sequential data. We show that this technique outperforms prior works on smaller data sets, and we demonstrate its utility as an efficient, and precise model on the billion-edge DARPA Transparent Computing (TC) Engagement 5 data set [**?**]. This data set consists of approximately 2TB of host logs spanning 11 days that EdgeTorrent can parse, embed, and classify over the course of a few hours.

In summary, this work's contributions are as follows:

- **End-to-end real-time intrusion detection system**
  We propose a full pipeline, which we call EdgeTorrent, for real-time anomaly detection capable of ingesting workloads typical of a real-world setting. We will show that EdgeTorrent outperforms state-of-the-art anomaly-based intrusion detection systems while running fast enough for use in a streaming setting.
- **Real-time method for streaming MPNN embedding**
  Though other approaches have used MPNNs, ours is the first capable of doing so in a streaming setting upon the complete data of billion-edge graphs. This allows for rich embeddings that make use of all available data, use message passing that follows temporal constraints, and do not require the full graph to be stored in memory. To our knowledge, our approach is the first model to utilize message passing on streams of edges.
- **Novel sequence outlier detection**
  We adversarially train a transformer [**?**] for sequence classification using the FenceGAN [**?**] framework, which allows the model to find a very precise decision boundary between normal behavior and anomalous behavior. By using a transformer for anomaly detection, during inference EdgeTorrent is far more efficient than clustering models and capable of real-time detection.

The remainder of the paper is organized as follows: Section 2 goes over the necessary background knowledge for this paper. Section 3 details our method for streaming graph neural networks. Section 4 describes how the output of the stream is analyzed for anomaly detection. Section 5 summarizes the whole pipeline. Section 6 provides some theory to motivate the use of a GNN for provenance graph encoding. Section 7 presents the experiments we did to evaluate our technique compared to prior works. Section 8 provides a brief overview of related work in this field. Section 9 discusses the limitations of our approach as well as directions for future work. Finally, Section 10 concludes the paper.

2 BACKGROUND

2.1 Provenance-based intrusion detection

Provenance graphs were first introduced by [**?**] to better understand the causes of system compromise. A provenance graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ is defined as a set of vertices, $\mathcal{V}$ representing system entities, and directed edges $\mathcal{E} = \{(u, v) \mid u, v \in \mathcal{V}\}$ representing interactions and causal relations between those entities. Critically, these edges have times associated with them, as well as types to distinguish various ways these entities can interact and when they did. More recently, Han *et al.* [**?**] suggested that these graphs could also be used for preemptive monitoring of a system. They emphasize that when a system is compromised, there will undoubtedly be unexpected interactions. If a baseline for how entities within a system interact is known, we expect malicious interactions to deviate significantly from what is normal.

One way to measure this is through *graph sketching*. This approach aims to stream in the graph in temporal order and maintain

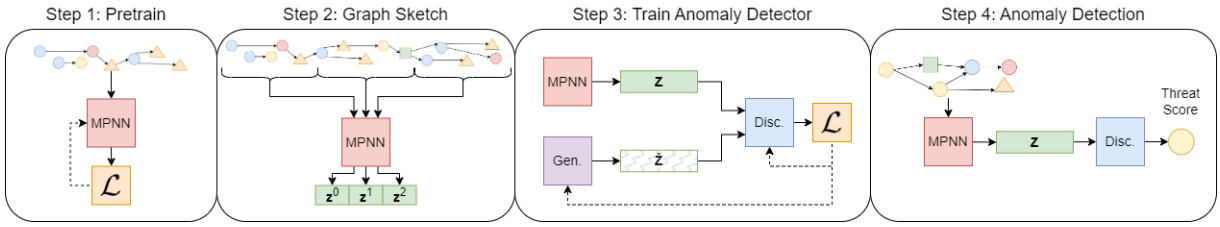


Figure 1: Visualization of the full EdgeTorrent pipeline. Steps 1 and 2 are described in Section 3. Steps 3 and 4 are described in Section 4.

a state vector that changes as new edges and nodes arrive. Formally, it is defined as a function $\Phi(\mathcal{G}, t)$ which takes as input an ordered list of all nodes and edges observed in a graph $\mathcal{G}$ up to some point in time, t, and produces a vector that expresses its state in some way. Critically, this is different from time slice-based approaches, which express a graph's state during a time window $\{t, t + \delta\}$. Whereas, the sketch vector represents all interactions that have occurred since time 0 up to t.

While not as fine-grained as node-level approaches to provenance graph analysis [**?** **?**], graph sketching methods have the advantage in speed and breadth. As they aim to summarize the activity of a system in a holistic manner, they are not subject to the same storage constraints as more fine-grained approaches [**?**]. In other words, at the cost of less informative alerts, graph-level alert systems can analyze more data in real time. We feel this is a fair trade-off, because although anomalous subgraphs detected by our system could potentially be analyzed by node-level approaches or even human experts, it would be very difficult to use a fine-grained approach on a large data set without some form of filtering or compression. We further discuss possible workarounds and solutions to this limitation in Section 9.

2.2 Message passing neural networks

Message passing graph neural networks (MPNNs) have been shown to be powerful in their ability to express complex relationships in non-Euclidean data [**?**]. They work by propagating a feature held by each node to all of its neighbors (sending a message). Next, each node aggregates the messages it received into a single, new feature. When this process is repeated k times, information from every node's k-hop neighborhood is available in a single vector.

Though they are powerful, MPNNs are still flawed. To fully execute a forward pass with traditional implementations of these models, it is necessary to hold the entire graph in memory [**?** **?** **?**], or use a sampling strategy that leaves out some data [**?**]. This is not feasible for graphs with billions of edges, spanning terabytes of data. For provenance graphs specifically, the graph will constantly grow as time passes [**?**] so approaches that use traditional MPNNs such as [**?** **?**] will struggle to continue running over time. Additionally, there are no temporal constraints on traditional message passing. If a node would have only had a certain message at time t, but it has edges that occurred at $t - \delta$, there is no way to enforce that the message is not passed anachronistically to those earlier neighbors. We argue that it is more useful for destination nodes to receive messages about their neighbors' present state in time; passing messages from future neighbors only adds noise to their representations.

To our knowledge, there does not exist a framework to conduct message passing on *streaming data*, nor is there an existing method to *temporally constrain* the messages passed between nodes. To fill these gaps, we present ours.

2.3 Generative Adversarial Networks for Anomaly Detection

Generative Adversarial Networks (GANs) were first introduced by Goodfellow *et al.* [**?**]. They consist of two neural networks trained in parallel: a discriminator and a generator. The discriminator is optimized to determine if a particular data point was sampled from the distribution of real data points, or if it was artificially generated by the generator. They are most often applied to image generation [**?** **?** **?**], but they have also been used for novel drug discovery [**?**] and even stenography [**?**].

The previous approaches take advantage of GANs' generative power, but the discriminator is also a useful tool. Prior works have shown these models to be highly effective at anomaly detection, especially when anomalous data points are unavailable as training data [**?** **?** **?** **?**]. If the generator is optimized to produce samples on the border of the normal distribution of data points, rather than inside the space they reside, it becomes an excellent source of negative samples [**?**]. As a result, the discriminator learns a very precise decision boundary of where the input data points lay in space. Then, when the discriminator processes previously unseen data points, if they deviate from the normal data distribution, they are classified as anomalous.

3 STREAMING GRAPH SKETCHES

In this section, we detail our pipeline to convert from provenance graphs to a series of vectors representing the graph's state as new nodes and edges are streamed in. This section correlates to Steps 1 and 2 from Figure 1. First, we train an MPNN for node representation on a subset of the available graph data in a self-supervised manner; in this work, we use link prediction. Then, when the MPNN is ready to conduct inference, we stream in a list of edges in the temporal order in which they were observed and process them with our streaming MPNN method. This results in a series of graph sketches characterizing the graph's state at a given point in time, and how that state changes as new nodes and edges appear.

The characteristic equation for graph convolutional networks is

$$H^{(\ell)} = \sigma\left(\hat{A}H^{(\ell-1)}W^{(\ell)}\right) \tag{1}$$

where $\hat{A}$ is the normalized adjacency matrix, W is a trainable parameter, $H^{(\ell)}$ is a matrix of activations at layer ℓ, and $\sigma(\cdot)$ is any non-linear activation.

Another way to understand this is on the per-node level. Let $h_v^{(\ell)}$ be the row in $H^{(\ell)}$ corresponding to node $v \in \mathcal{V}$. Then Equation 1 is equivalent to the following, solved for each row of $H^{(\ell)}$:

$$h_v^{(\ell)} = \sigma\left(\frac{\sum_{u \in \mathcal{N}(v)} h_u^{(\ell-1)}}{|\mathcal{N}(v)|} W^{(\ell)}\right) \tag{2}$$

where $\mathcal{N}(v) = \{u \mid (u, v) \in \mathcal{E}\}$. In the first layer, $H^{(0)} = X$, the nodes' features. However, the function is trivially extended to include edge features as well by defining the first layer as

$$h_v^{(1)} = \sigma\left(\frac{\sum_{u \in \mathcal{N}(v)} x_u \| x_{(u,v)}}{|\mathcal{N}(v)|} W^{(1)}\right) \tag{3}$$

where $\|$ denotes the concatenation operator, and $x_{(u,v)}$ is the feature associated with edge (u, v).

Equations 1 and 2 vary slightly between specific MPNN implementations, particularly with respect to the aggregation function [? ? ?] (the fraction in Equation 2), but in general, these methods all follow the above formulae. Nodes propagate a message to their neighbors, and those messages are aggregated in some fashion into a single vector.

3.1 Training

Our method for streaming convolutions is purely an efficient method for inference. Thus, we must first train the model on some subset of the data. This correlates to Step 1 of Figure 1. This training must be inductive so future new nodes can be embedded effectively. For our experiments, we randomly sampled time windows from the graph that were small enough to fit into memory and optimized the model for link prediction. This aims to minimize

$$\begin{aligned} \mathcal{L} &= -\log(\mathrm{sim}(h_u, h_v)) - \log(1 - \mathrm{sim}(h_u h_{v'})) \\ \mathrm{sim}(h_u, h_v) &= \sigma(h_u^T h_v) \end{aligned} \tag{4}$$

for $\{(u, v) \in \hat{\mathcal{E}}\}$ where $\hat{\mathcal{E}} \subset \mathcal{E}$ is the sampled snapshot, and $\{(u, v') \notin \mathcal{E}\}$ are randomly sampled non-edges.

3.2 High-performance Inference

Now that the model is ready for inference, we move to Step 2 of the pipeline shown in Figure 1. In this work, we introduce the notion of streaming convolutions. These can be thought of as incremental updates to each node embedding that is affected by the arrival of a new edge. These embeddings are periodically aggregated to calculate the graph sketch at that point in time. Here, we present an efficient way to continually update this value with as little overhead as possible.

Node Embedding.
Suppose a node $v \in \mathcal{V}$ has a new inbound edge added to it, (u, v). Given the intermediate layer activations of node u, $\{h_u^{(0)}, ..., h_u^{(L-1)}\}$,

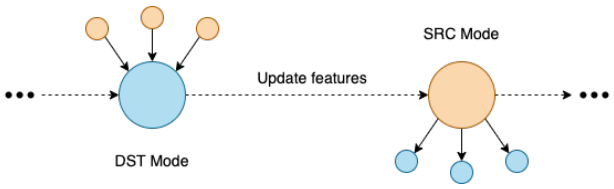


Figure 2: Temporal edges forming a bipartite graph ensuring temporal constraints are followed during stream convolution.

and the number of neighbors of v, c_v, we can update the intermediate layer $h_v^{(\ell)}$ of v as follows:

$$h_v'^{(\ell)} = \sigma\left(\frac{\sigma^{-1}\left(h_v^{(\ell)}\right)c_v + W^{(\ell)}h_u^{(\ell-1)}}{c_v + 1}\right) \tag{5}$$

This follows from the fact that

$$h_v^{(\ell)} = \sigma\left(\frac{W^{(\ell)}(h_{u_1}^{(\ell-1)} + h_{u_2}^{(\ell-1)} + ... + h_{u_{c_v}}^{(\ell-1)})}{c_v}\right) \tag{6}$$

for neighbors of v, $\{u_1, ..., u_{c_v}\}$. That is, this approach simply adds one additional neighbor to the list and recalculates the layer.

We can further optimize the above approach by delaying the computation. Note that the updated node feature is not needed until an outbound edge is added from this node. To this end, as edges stream in, we say that the node is in one of two states: source or destination. If a node is in the "source" state, its encoding is static and is propagated to its neighbors. Otherwise, it is receiving messages from its neighbors that will change its internal state or encoding. But it is not necessary to retroactively update any encodings it may have propagated in the past, as they were representative of that node's state at the time they were sent.

The insight is that the encoding of v only needs to be updated when v undergoes a state transition from destination to source. Based on this insight, if we use MPNN architectures amenable to incremental updates, it is sufficient to passively aggregate new edge data as it comes in, and only perform the matrix multiplication and inverse non-linearity functions when they are needed. For example, if the aggregation function is *mean*, as in Equation 2, let $\mathcal{H}_v$ denote the set of all messages sent to node v before an outbound edge is observed. To fully update h_v, it is assumed that all intermediate activations from $0 \leq \ell < L$ are sent during message passing, much like in a Jumping Knowledge Network [?]. Intermediate features of source nodes connecting to v are then added together forming partial node embedding $h_v^{*(\ell)} = \sum_{h^{(\ell)} \in \mathcal{H}_v} h^{(\ell)}$. We continue to accumulate new edge data until v transitions from destination to source, at which point we calculate $h_v'^{(\ell)}$ with only minor updates to Equation 5. In the numerator, $h_u^{(\ell-1)}$ becomes $h_v^{*(\ell-1)}$, and the denominator becomes $c_v + |\mathcal{H}_v|$. This same principle can be applied to other various aggregation functions such as pooling, but not to models with non-invertible aggregation functions, such as GIN [?].

This state-transition approach to message passing can be thought of as a special kind of edge between nodes: a temporal edge, representing the same entity at different points in time as shown in Figure 2. This enforces the constraint that message passing only flows forward through time while avoiding the memory restrictions that come from storing so many extra nodes. Juxtapose this with traditional MPNNs, which ignore time, and propagate anachronous messages. In prior works, the future state of a node's neighbor affects its encoding during the present, something that is undesirable when attempting to summarize a graph at a particular point in time.

Node Aggregation.
To convert from individual node embeddings to full graph embeddings, we compute a temporally weighted average of each node's embedding after some number of edges are ingested (i.e., the *sketch size* $|S|$). Note that, for soundness, when the weighted average is calculated, it is necessary to update the partial node embeddings for any nodes still in the destination state. The weighted average is the sum of each node embedding weighted with a forgetting parameter e^{-kt_v} where t_v is the period of time since node v was last seen in any interaction, and k is a hyperparameter. Let t denote the $1 \times |\mathcal{V}|$ vector of these times for all nodes.

The naïve way to implement this is with a simple matrix multiplication: $z = \frac{1}{|\mathcal{V}|} H^\intercal e^{-kt}$, where H denotes the final activation. However, this presents two major problems. First, as the graph is streamed for longer periods of time, H will eventually become so large that it is prohibitively expensive to fully load it into memory. Second, using a simple average will gradually reduce the signal generated by new or active nodes and lower the overall average, due to the size of the graph and the presence of long-term inactive nodes.

To solve the first problem, we note that once the average has been calculated in full for a given set of nodes, only nodes that were updated afterward will change it. Other unchanged nodes will simply decay. With a slight abuse of notation, let the superscript n denote a particular variable's value when sketch n was created. For example, z^n denotes the n^{th} sketch, h_v^n denotes v's final activation when sketch n was created, and so on. Then,

$$z^{n+1} = \frac{\sum_{i \in U} h_i^n e^{-k(t_i^n + \delta)} + \sum_{j \in C} h_j^{n+1} e^{-kt_j^{n+1}}}{|U| + |C|} \tag{7}$$

where U and C represent the sets of unchanged and changed node embeddings, respectively, and δ represents how much time has passed since z^n was calculated.

To avoid extraneous memory use, we aim to represent this equation only in terms of the changed nodes, and the previous sketch z^n. To do this, consider the definition of z^n

$$z^n = \frac{1}{|\mathcal{V}^n|} \sum_{v \in \mathcal{V}^n} h_v^n e^{-kt_v^n} \tag{8}$$

If we rearrange the equation as

$$|\mathcal{V}^n| z^n - \sum_{j \in C} h_j^n e^{-kt_j^n} = \sum_{i \in U} h_i^n e^{-kt_i^n} \tag{9}$$

and multiply both sides by the decay parameter $e^{-k\delta}$, we find

$$\sum_{i \in U} h_i^n e^{-k(t_i^n + \delta)} = e^{-k\delta} \left(|\mathcal{V}^n| z^n - \sum_{j \in C} h_j^n e^{-kt_j^n} \right) \tag{10}$$

may replace the first term in the numerator of Equation 7. In other words, all unchanged embeddings that contributed to the previous sketch simply decay in signal strength, and changed nodes are recalculated and added back in. Thus, we can update the graph sketch using only the nodes that were updated rather than the entire graph, which significantly reduces the memory overhead.

We used the python `shelve` module [**?**] to implement this process and ensure only nodes that are being actively updated are cached in memory. The remaining unchanged nodes are stored on the disc in a GDBM database [**?**]. After each graph sketch is calculated, any updated nodes are flushed back to the disc. The set of changed nodes C from Equations 7 and 10 is simply the cache of edited nodes already in memory, managed by `shelve`.

To solve the second problem, we change the denominator in Equation 7 to $\sum_i e^{-kt_i^{(n+1)}}$. Again, we wish to calculate this value using only the updated values present in the database cache. We can again take advantage of the fact that for all the unchanged nodes, their contribution to the denominator is decayed by a fixed amount. Thus, the denominator is calculated as

$$\begin{aligned} d^n &= \sum_i e^{-kt_i^n} := \sum_i d_i^n \\ d^{n+1} &= e^{-k\delta} \sum_{i \in U} d_i^n + \sum_{j \in C} d_j^{n+1} \\ &= e^{-k\delta} \left(d^n - \sum_{i \in C} d_i^n \right) + \sum_{j \in C} d_j^{n+1} \end{aligned} \tag{11}$$

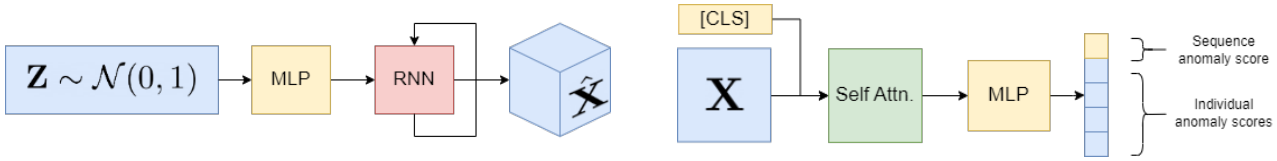
With these optimizations in place, it is possible to stream in data faster than real-time to build these vectors and analyze their changes. The dynamics of these graph sketches, as we will show, hold enough information to detect anomalous activity.

4 ADVERSARIAL ANOMALY DETECTION

In this section, we describe how EdgeTorrent uses the embeddings from Section 3 for anomaly detection. We will first detail the architecture of the generator and discriminator models, then describe the objective functions these models are trained with. This section correlates with Steps 3 and 4 of the pipeline shown in Figure 1. In Step 3 of the pipeline, we train a GAN in the manner described by [**?**] using EdgeTorrent sketches as input data. Then, in Step 4, the discriminator is used as an anomaly detector on new EdgeTorrent sketches; if the discriminator finds a sketch sequence has a high probability of being generated, it is said to be anomalous.

4.1 Model Architecture

We implement the anomaly detection model as a generative adversarial network (GAN) [**?**]. These models work by training two models in tandem: a *generator* and a *discriminator*. The generator takes random noise, Z as input and attempts to transform it into vectors $\hat{X}$ emulating those in the input data distribution. The discriminator takes real data samples, X, and the output of the generator as its inputs then classifies them as real or synthetic.



(a) Sequence generator

(b) Sequence discriminator

Figure 3: The two components of the anomaly detection model. The generator takes noise as input and runs it through a multi-layer perceptron (MLP) and a recurrent neural network (RNN) to simulate a sequence of graph sketches. The discriminator takes a sequence as input, and using a self-attention network, determines the likelihood that the sequence was generated, or real.

The generator is implemented as a recurrent neural network that takes one vector of noise and processes it into a sequence of synthetic graph sketches. Mathematically, the generator function is defined as

$$\begin{aligned} G(Z;\theta_G) &= \hat{x}_0 \| \hat{x}_1 \| ... \| \hat{x}_N \\ \hat{x}_n, h_n &= RNN(\hat{x}_{n-1}, h_{n-1}) \\ \hat{x}_0, h_0 &= RNN(\sigma(W_{G_0} Z + b), 0) \end{aligned} \tag{12}$$

where $\|$ is the concatenation operator, and σ is a non-linearity function. The RNN function denotes any generic recurrent neural network; in this work, we use an LSTM [?]. This process is illustrated in Figure 3a. Note that unlike prior works using GANs for sequence generation such as [? ? ?] we only supply the generator with a single noise data point. This is following the work of [?] which aims to identify anomalous state changes between graph sketches. By supplying a single latent vector, we encourage the generator to learn the way sketches are likely to change over time, given a single random starting position.

The discriminator is implemented as a self-attention network, in a very similar manner to a transformer encoder [?]. It takes a sequence of vectors as input, adds a CLS token as introduced by [?] for classification, and passes this series into a transformer encoder block. This process is illustrated in Figure 3b. For a more detailed description of this process, we direct the reader to [?].

4.2 Objective Functions

In a traditional GAN, the discriminator and generator are optimized over the minimax game,

$$\begin{aligned} \min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_d(x)}\left[\log D(x)\right] \\ + \mathbb{E}_{z\sim p_z}\left[\log(1 - D(G(z))\right] \end{aligned} \tag{13}$$

where G and D are the generator and discriminator models, p_d is the distribution of the input data set, and p_z is the distribution of noise the generator uses as input [?]. Put another way, the loss functions of the generator and discriminator $\mathcal{L}_G$ and $\mathcal{L}_D$ are

$$\mathcal{L}_G = \frac{1}{N}\sum_{i=1}^{N} \log(1 - D(G(z_i))) \tag{14}$$

$$\mathcal{L}_D = -\frac{1}{N}\sum_{i=1}^{N}\left[\log D(x_i) + \log(1 - D(G(z_i))\right] \tag{15}$$

where $Z = \{z_1, ..., z_N\}$ is sampled from p_z and $X = \{x_1, ..., x_N\}$ are real samples from the data set: in this case, a sequence of graph sketches.

This method is very effective at training generators to produce realistic-looking samples, however, for anomaly detection, it must be slightly altered.

Our method takes inspiration from the work of FenceGAN [?] and modifies it for use on sequential data. In this method, the minimax game is changed such that the generator produces samples right on the decision boundary of the discriminator, rather than inside the data distribution. In this way, the generator is trained to produce synthetic anomalies, and the discriminator learns a very precise boundary constraining normal behavior. This is accomplished by changing Equation 14 to

$$\mathcal{L}_G = \mathcal{L}_{G_e} + \beta\mathcal{L}_{G_d} \tag{16}$$

where $\mathcal{L}_{G_e}$ denotes *encirclement loss*, $\mathcal{L}_{G_d}$ denotes *dispersion loss*, and β is a hyperparameter. These two values are defined as

$$\begin{aligned} \mathcal{L}_{G_e} &= \frac{1}{N}\sum_{i=1}^{N} \log\left(|\alpha - D(G(z_i))|\right) \\ \mathcal{L}_{G_d} &= \frac{1}{\frac{1}{N}\sum_{i=1}^{N} \|\mu - G(z_i)[0]\|_2} \end{aligned} \tag{17}$$

where $\alpha \in (0, 1)$ and $\mu = \frac{1}{N}\sum_{i=1}^{N} G(z_i)[0]$.

The encirclement loss ensures that the generator is producing samples that are on the periphery of p_d rather than within it. This ensures the generator is creating samples more similar to anomalies than real data, but not so anomalous that they are implausible. The dispersion loss optimizes the generator to maximize the standard deviation of its samples. This ensures that not only are samples confusing to the discriminator, but they are also varied. Because the random noise only affects the first generated graph sketch, and the remaining sketches are deterministic, we only evaluate the standard deviation of the embeddings for the first generated graph sketch.

The discriminator's loss function is also changed slightly. Instead of prioritizing its ability to discern between real and generated samples equally, the discriminator is biased such that it weights understanding the real samples as more important. This is because while anomalous samples can be near-infinitely varied, we work under the assumption that we have a close to complete understanding of the benign activities of a network. By weighting benign samples as more important to identify, the discriminator better learns where

the distribution of normal activity ends. It is less focused on learning what an anomalous sample looks like, and more focused on learning what they do not look like. Thus, Equation 15 is now

$$\mathcal{L}_D = -\frac{1}{N}\sum_{i=1}^{N}\left[\log D(x_i) + \gamma \log(1 - D(G(z_i))\right] \quad (18)$$

where $\gamma \in (0, 1)$ is a hyperparameter.

4.3 Anomaly Detection

After the full GAN model has been trained, EdgeTorrent assigns threat scores by calculating the likelihood that a new data point was generated. This is of course the output of the discriminator. Because the discriminator was optimized to learn a very tight decision boundary around the benign sequences of graph sketches it was trained with, if a new sequence of sketches is thought to have been generated, it is likely outside of the normal data distribution, and therefore anomalous. Thus, Step 4 of Figure 1 is to pass a sequence of sketches to the discriminator which will assign that sequence an anomaly score. If that score is above a certain threshold, the sequence is classified as anomalous.

5 SYSTEM IMPLEMENTATION

Putting it all together, we will briefly detail a running example of the EdgeTorrent pipeline. In this example, we assume that the MPNN and GAN models have been trained in the manner described in Sections 3 and 4, and simply show the model in action.

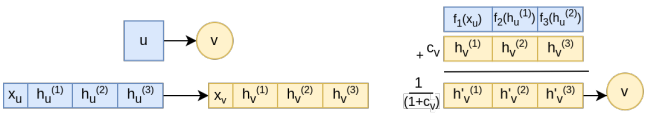


Figure 4: Incremental update as a new edge is streamed in

The system will read logged events from a buffer such that each edge is formatted as <u,v,u_x,v_x,edge> where u and v are unique identifiers for system entities, u_x and v_x are node features associated with entities u and v (usually the type of entity), and edge is the relation type between those two entities. Optionally, an additional `ts` timestamp input may be added if the user wants to use this quantity as the t_i^v variable in Equations 10 and 11, otherwise, the timestamp is the count of events. Each logged event is equivalent to an edge in the provenance graph. For each edge that is read in, the embedding of v as well as v_x is passed to u to update its embedding, as shown in Figure 4. Every |S| edges processed, the weighted average of every node's embedding is stored.

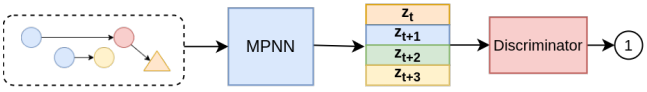
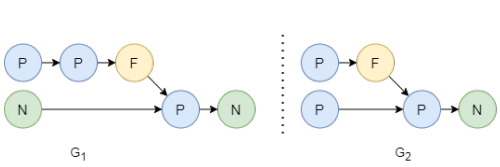


Figure 5: The full EdgeTorrent Pipeline

From here, as is shown in Figure 5, the sequence of stored edges is passed to the trained discriminator module which will output an anomaly score. We will show through experimentation that using as input sequences spanning entire temporal graphs (§ 7.1), fixed time spans (§ 7.2), and fixed sequence lengths (§ 7.3) are all effective ways to process these embeddings.



Type	G_1	G_2
N	1	1
P → F	1	1
P → P	1	1
P → N	1	1
F → P	1	1
N → P	1	

Figure 6: G_1 is an provenance graph showing remote execution of a malicious process followed by exfiltration. G_2 is an provenance graph showing a benign user uploading data to a web server.

6 MOTIVATING EXAMPLE

Graph kernel approaches have been shown to be excellent for capturing the semantics of provenance graphs in prior works. The two that we will compare EdgeTorrent to in this work, StreamSpot [?] and Unicorn [?], both justify their success by showing that their approach is a form of the Weisfeiler-Lehman Isomorphism (WL) test [?], and can thus differentiate between graph states. In the prior works, they achieve this using different hashing algorithms that summarize nodes' local neighborhoods. Similarly, the findings of Xu et *al.* [?] prove that GNNs are a weak form of the WL test. This means that two graphs having the same node embeddings after a pass through a GNN is a necessary (but not sufficient) condition to show that they are isomorphic. Therefore, as in prior works on graph kernels, every graph sketch produced with our method represents a set of unique possible states for the graph to be in. However, our approach produces graph sketches with more semantic information than prior works.

To better illustrate this, consider the example shown in Figure 6. Nodes are labeled as processes (P), files (F), and network events (N). There are two simplified provenance graphs: G_1 shows a malicious process started remotely, exfiltrating data, and G_2 shows a benign process started by the user, uploading data to a web server. The table shows how these graphs would be represented by StreamSpot, which only considers first-order relations. Clearly, the distinction between the two is minor. Our approach, as well as Unicorn, both consider higher-order relations and would capture the important distinction between the two graphs: that one has the path P → P → N, and the other has N → P → N. As our method optimizes for link prediction, if a path like N → P → N is unexpected, we expect each node in that path to have embeddings that are orthogonal. When the embeddings in the graph are averaged, assuming perfect encoding, the sum of the nodes in the abnormal path will have a maximal angle, and create fluctuations in multiple dimensions of the embedding, producing a very strong signal in the aggregated graph sketch. Likewise, an expected path will consist of parallel embeddings whose average will have a low vector basis, thus producing a smaller signal. Unicorn, on the other hand, uses a hashing procedure to represent these small differences. So not only is it unclear if a perturbation this small would create a strong enough signal to

create an alert, but a similarly small but benign change between G_1 and G_2 would create a signal of equal magnitude.

While both our approach and the aforementioned prior works can capture approximations of graph isomorphisms and temporal patterns such as bursts of activity, the prior works are both limited in how they can express minor graph differences. Due to the hashing algorithm used to represent node activity, minor differences in graphs could result in major changes in graph sketches, regardless of the importance of these changes. Our method ensures that major changes in sketches only occur when a graph deviates significantly from an expected pattern. The use of a hashing algorithm makes these two methods insensitive to the magnitude of change in graph structure. As our results will show in Section 7, it makes these methods more susceptible to false positives.

7 EVALUATION

To evaluate the effectiveness of our process, we applied it to several provenance graph datasets and compared it to other provenance-based intrusion detection systems. We evaluate EdgeTorrent on host-level provenance graphs, network-level graphs, and a real-world dataset containing host and network activity. In each experiment, we evaluate EdgeTorrent's ability to precisely detect anomalous graphs given the generated sketch sequences. There are relatively few existing approaches to graph sketch generation and analysis. Many existing approaches for temporal graph analysis are largely not designed for use in a streaming setting [**? ? ? ?**] or are not designed for graph classification [**? ?**]. In our literature review, we identified only two that were comparable to our method: StreamSpot [**?**] and Unicorn [**?**]. Both methods use a hashing algorithm to generate graph sketches.

We will show that EdgeTorrent's approach using MPNN embeddings processed with a GAN outperforms other techniques for anomalous graph detection. Additionally, we show that our method has comparable run times to the state-of-the-art techniques we compare it to and depending on the implementation settings is even faster. Specific hyperparameter details for each experiment can be found in Table 7 in the supplemental material.

7.1 Simplistic Host Data

The StreamSpot dataset [**?**] consists of 600 host log files split into 6 categories of activity, each containing 100 graphs. 5 of these categories are benign behavior (checking email, playing a video game, etc.) and one is the trace of a drive-by-download attack. Because these activities are so well isolated, the results on this dataset are not indicative of how any given IDS would perform in the wild. Nonetheless, it is a popular benchmarking dataset for these types of systems and acts as a proof-of-concept for our technique.

Table 1: StreamSpot dataset metadata

	$\|G\|$	Avg $\|V\|$	Avg $\|E\|$
Benign	500	8,300	173,000
Attack	100	8,900	28,400
All	600	8,400	149,000

As with all datasets we test on, for node and edge features, we use a one-hot representation of their types. These two features are concatenated together to represent the messages passed from source to destination nodes, as defined in Equation 3. There are 8 types of nodes and 26 relation types. Table 1 contains more information about the metadata of this dataset. For all models, we use 128-dimensional vectors for graph sketches and a decay rate of $k = 0.02$.

To evaluate the model, we perform 4-fold validation, as is done by prior works, where 25 graphs are held out from each benign group for testing along with all malicious graphs. On this small dataset, the model quickly converges, requiring only 10 epochs to train. The only inputs to the GAN are sketch sequences built from the benign graphs in the training sets. An additional 25 graphs are held out for validation. These graphs are also used to select the optimal cutoff for classification: the highest anomaly score given to a benign graph is used as the threshold. Any graph scoring higher than this is classified as anomalous. The results of this experiment are shown in Table 2.

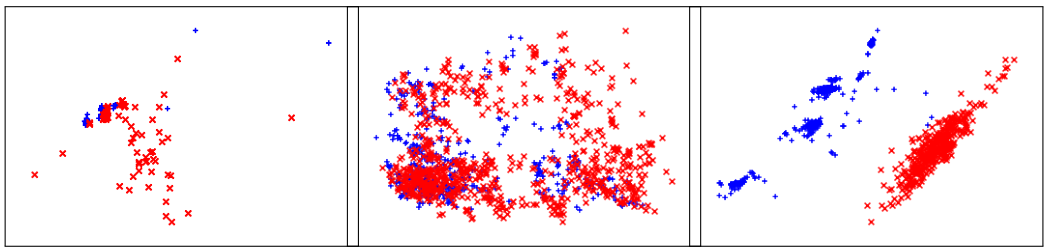
In addition to the two graph-kernel methods, we compare EdgeTorrent to two other popular IDSs: threaTrace [**?**] and PROV-GEM [**?**]. ThreaTrace is a node-level anomaly detection system but was adapted by the authors for full anomalous graph detection on this dataset using the number of anomalous nodes present in the graph to classify it. PROV-GEM is a supervised IDS that is explicitly given a subset of anomalous graphs to learn to classify.

Table 2: Comparison to prior works on the StreamSpot dataset.

	Pr.	Re.	Acc.	F1
StreamSpot	0.7700	0.7400	0.8400	0.7500
Unicorn	0.9800	0.9300	0.9600	0.9400
threaTrace	0.9802	0.9960	0.9920	0.9881
PROV-GEM	**1.0000**	0.9400	0.9700	0.9900
EdgeTorrent, $\|S\| = 2000$	0.8829	0.9722	0.9337	0.9243
EdgeTorrent, $\|S\| = 200$	**1.0000**	**1.0000**	**1.0000**	**1.0000**

The StreamSpot model uses the final form of the graph sketch vector to represent the whole graph, e.g., each graph is represented by a single vector. Unicorn samples the graph sketch as it changes over time and analyzes each sequence in terms of the clusters it forms and the state transitions between those clusters. However, due to the complexity of building a separate clustering model for each benign graph, its runtime is heavily constrained by the number of snapshots. As such, in the original work, they used a sketch size $|S|$ of one sketch per 2,000 events. However, this results in certain graphs being represented by fewer than 10 vectors, which is difficult to build a model from. In a good-faith effort to replicate their experiment, we also use a sketch size of 2,000, however, we also run an experiment with $|S| = 200$. This provides a more fine-grained view of the evolving graph through time and helps to combat the claim made by [**?**] that graph kernel methods cannot detect fine-grained anomalies.

It is evident from these tests that EdgeTorrent is the best performing method, capable of perfectly classifying this dataset. This is despite the fact that it is completely unsupervised, and due to the method we use to determine the classification threshold, it is using less data to train than the other methods. We note that our



(a) StreamSpot (b) Unicorn (c) EdgeTorrent (Ours)

Figure 7: PCA decompositions of the three graph kernel algorithms' embeddings of the UNSW-NB15 dataset. Blue marks indicate sketches from a benign graph. Red marks indicate sketches from a malicious graph.

approach even outperforms the supervised approach PROV-GEM, and the node-level approach threaTrace. We attribute this to our method's consideration of time as an important variable. The latter two methods view the graphs as monolithic structures and ignore the order in which edges appear. Often, to differentiate between a drive-by download attack and a regular downloaded file, for example, the order in which bursts of edges appear is more telling than the edges themselves. This is reflected in our model's perfect recall scores, compared to PROV-GEM's lower scores.

However, as previously stated, this is a simplistic dataset, and our inference model has far more parameters than those we compare it to. It could be argued the GAN model is simply overfitting on simple data. For this reason, we turn to more complex datasets for better evaluation.

7.2 Network Traffic Dataset

The UNSW-NB15 dataset contains web traffic between computers on the University of New South Wales network [**?**]. It is a combination of real benign activity, and simulated attacks over the course of four days–though activity was only recorded for about 8 hours per day. To convert it to a graph, we view hosts as nodes, and connections between them as attributed edges. We split the data into 10-minute increments and build separate graphs for each time span.

Unique to this dataset is its edge-centric nature. As seen in the metadata in Table 3, there are very few unique IP addresses, with only 46 unique hosts across the entire dataset. This presents a very different kind of problem. Often in provenance graph analysis, the limiting factor is the explosion of new nodes representing ephemeral activity like temporary file creation, or short-lived processes. With this dataset, there are very few nodes, but each has an incredibly high degree. For traditional MPNNs this would be a problem, as all edges must be processed at once, but for our method, as well as the other streaming graph kernel methods, this makes computation easier as there are fewer nodes whose embeddings need to be stored. Also unique to this dataset is the data imbalance as a majority of activity is malicious. Data imbalance in anomaly detection datasets is common, but normally a majority of the activity is benign. This presents a challenge for training, due to a the lack of benign activity for these models to learn from. In summary, this is a challenging dataset for any model, and we feel a significant step up in difficulty from the StreamSpot dataset.

In addition to the connectivity information, the data contains a wealth of information about each individual edge. Prior works [**? ?**] use these edge features on their own to train anomaly detection models on the edge level, but this ignores the valuable topological data. As we will show, with minimal data beyond topological information, it is still possible to build a powerful anomaly detector. We extract the top 15 source and destination ports, as well as a token for "other", and use their Cartesian product as the edge feature. Additionally, we tokenize the network prefix from each IP address to use as node features. In full, this results in 7 classes of nodes, and 256 classes of edges.

Table 3: UNSW-NB15 dataset Metadata

	$\|G\|$	Avg $\|V\|$	Avg $\|E\|$
Benign	61	26	14,600
Attack	79	43	20,525
All	140	36	17,963

Before evaluating the anomaly detectors, it is illustrative to simply analyze the raw embeddings produced by the various models. Figure 7 shows the PCA decomposition of sketches generated by each method. More obscure decompositions do not necessarily mean the embeddings are less separable, the data may be too complicated to project into lower dimensions. But if data points are well separated in 2D space, it is fair to assume it is at least as separable in the original embedding space. From the figures, we can see that even without further processing, the sketches produced by EdgeTorrent are almost linearly separable in 2D space. By using a GCN rather than a hashing method, the sketches produced have more meaning. Normal activity falls into several tight clusters, while abnormal activity spans a completely different space, well partitioned from the benign sketches. Additionally, careful examination shows that sketches belonging to the same series move in smooth lines, starting at one end of clusters, and gradually drifting away in smooth lines. We attribute this to the fact that the GCN function used to embed the nodes is differentiable, so as new edges are added, graph sketches change gradually, unlike embeddings from hash-based algorithms.

StreamSpot also produces several tight clusters of benign activity, while sketches for abnormal periods are scattered through the embedding space. However, we note that there is a great deal

of overlap; the clusters are not as well-centered. Unicorn's decomposition is far more scattered with no clear separation. This may be a result of the different sketching approaches followed by these two techniques: Unicorn hashes all activity that has occurred on a single node, producing more unique hashes, and StreamSpot caps the number of events per node to a set number (in our experiments 100) before generating a new node representation. However, the Unicorn method incorporates the state transition between clusters as well as simple cluster membership into its classification strategy, so viewing the embeddings is less telling of how the inference model will perform.

Table 4: Comparison to prior works on the UNSW-NB15 dataset.

	Precision	Recall	Accuracy	F1 Score
StreamSpot	0.9629	0.9797	0.9466	0.9711
Unicorn	0.8254	**1.0000**	0.8573	0.9041
EdgeTorrent	**0.9991**	0.9996	**0.9987**	**0.9993**

Table 4 reports the results of experiments on these methods. The results reported are the average of 5-fold validation tests. All models are trained on 10 hours of clean data with no anomalous activity, with 2 hours of clean data held out for validation, then evaluated on approximately 22 hours of malicious and benign activity. The GAN is trained for 100 epochs.

EdgeTorrent is the best performing model in every respect but recall. Unicorn had perfect recall, but it was at the expense of a low precision score. Even with the threshold for alerting set very high, it was plagued with false positives. However, we feel that the F1 score is the best way to evaluate the balance of true and false positives and score these models. Using this measure, we can say that EdgeTorrent struck the best balance between the two. The next best scoring method was StreamSpot. This is surprising, given that Unicorn outperforms StreamSpot in other tasks, but we attribute this to the low number of nodes. While Unicorn produces a unique hash for each sequence of edges a node receives, StreamSpot caps the number of edges to a user-specified value and hashes each collection separately. This means that in datasets where there are few nodes, each with high degree, StreamSpot's sketches will be composed of more self-similar node embeddings than Unicorn's, as Unicorn's node embeddings are constantly changing randomly with each new hash; StreamSpot's and our own move through space more incrementally.

These results support our trepidation toward hash-based graph kernel methods. If individual node embeddings change in a smoother manner, the sketches built from those node embeddings will too. This produces better sequences for analysis. Finally, by upgrading from simple cluster analysis to a more advanced GAN model, there is the potential to capture more complex temporal patterns. This is evident by EdgeTorrent's ability to improve both in precision, limiting false positives, and also recall, as it can detect abnormal patterns that are more complex than simple point anomalies or abnormal state transitions.

7.3 Large Endpoint Dataset

To demonstrate EdgeTorrent's usefulness in a real-world system, the final experiment is conducted on the DARPA Transparent Computing (TC) Engagement 5 dataset. This dataset consists of 11 days of activity across several host machines collected by multiple teams for analysis [**?**]. Because this data was generated in a real-world setting, not every dataset was clean enough to be useful for training or analysis. As such, we analyze the data collected by 3 teams that provided the data with the fewest discrepancies: CADETS, FiveDirections, and TRACE. The teams captured activity from 1 FreeBSD host, 3 Windows 7 hosts, and 3 Ubuntu hosts, respectively. To our knowledge, this work is the first to implement a graph kernel-based IDS on this dataset, so these results may serve as a benchmark for future works.

In our experiments, nodes may have the following types:

Processes, Threads, Units (a group of related threads, specific to TRACE), Files, Temporary Files, DLLs (specific to Windows machines), Named Pipes, Registry Keys, Local network activity, External network activity, Special network activity, Sockets, and "other" when the type cannot be inferred from the logs.

We categorize files as normal, temporary, or DLLs using hints from their path variables. Similarly, we categorize network events as local if they take place in the local IP range, and special if the address ends with `.255` or starts with `0xff` in the case of IPv6.

Table 5: TC Engagement 5 dataset Metadata

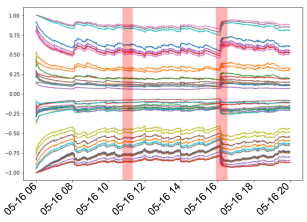
	Hosts		Sequences	Avg $\lvert\mathcal{V}\rvert$	Avg $\lvert\mathcal{E}\rvert$
CADETS	1	Benign	52	329,000	4,370,000
		Attack	4	377,000	5,120,000
		All	58	333,000	4,430,000
FiveDirections	3	Benign	155	713,000	4,540,000
		Attack	7	728,000	5,120,000
		All	162	713,000	4,560,000
TRACE	3	Benign	244	1,600,000	4,830,000
		Attack	3	1,680,000	5,120,000
		All	247	1,600,000	4,830,000

Because the data collection is continuous over the course of the engagement, there is no obvious way to separate out individual graphs. Prior work [**?**], in their analysis of similar data, separated graphs by connected components. But this requires knowledge of what the graph will do in the future–knowledge we cannot presume to have if this were applied as a real-world system. Instead, we consider all activity that occurs on a single host over the course of one day as one graph. Furthermore, because no attacks take place outside of business hours for this dataset, and there is minimal activity at night, the graphs start at 7 am and end at 7 pm. Sketches are generated for all edges that take place during this period. Due to the unwieldy size of these graphs, we set the sketch size to 20,000 edges for CADETS and FiveDirections, and 40,000 for TRACE. The sketches are then broken into sequences of length 256 (approximately 4 hours) for analysis. More details of the datasets are available in Table 5

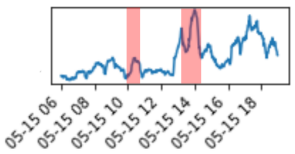
Table 6 shows the scores attained by each technique during evaluation. The results shown are the average of 5-fold validation on benign sketch sequences, where all malicious sequences are reserved for testing after 100 epochs. We find that even though

Table 6: Evaluation of graph kernel methods on different TA1 monitoring systems on the TC Engagement 5 dataset

	TA1	Pr.	Re.	Acc.	F1
StreamSpot	CADETS	0.5066	0.3000	0.4417	0.2883
	5D	0.3740	0.4000	0.6513	0.2410
	TRACE	0.3525	0.2667	**0.8109**	0.2511
Unicorn	CADETS	0.1818	0.5000	0.7027	0.2667
	5D	0.0745	**1.0000**	0.1031	0.1386
	TRACE	0.0219	**1.0000**	0.0290	0.0441
EdgeTorrent	CADETS	**0.7111**	**0.8000**	**0.7242**	**0.6576**
	5D	**0.4944**	0.7429	**0.7710**	**0.5896**
	TRACE	**0.6467**	0.4000	0.7739	**0.4356**



(a) Graph sketch over time for FiveDirections host 1 on 05-16



(b) Anomaly score over time for FiveDirections host 2 on 05-15

Figure 8: The effect of anomalous activity on graph sketches, and anomaly scores. Regions highlighted in red represent periods where the system was under attack.

this task is difficult for all graph-kernel methods, EdgeTorrent outperforms the prior works by quite a large margin. It even attains a perfect score on 2 of CADETS' 5 folds. However, its performance is very contingent on the benign data it trains with, suggesting the benign activity space was far larger than what the models had to train with. The prior works were similarly hindered. Where Unicorn again struggles to find a high enough threshold to avoid false alarms, StreamSpot skews the opposite way, marking most sequences as benign.

It is difficult to properly label this dataset because 1) the ground truth document reports only the time and IP address of the attack and 2) it is unclear when certain attacks end. For example, the attack that occurs on 5/16 on FiveDirections host 1 establishes a C2 connection that remains active throughout the remainder of the day, and through the night. It is unclear if this is disconnected the following day or not, and this may account for the lower precision score on this dataset. Likewise, the attack on CADETS on 5/10 assumes the attacker has stolen credentials. They simply SSH to the machine and use a password they already know. It is unclear if this is truly behavior that should be labeled as anomalous and may account for some of the false negatives EdgeTorrent experiences on certain folds while getting perfect scores on others. Nonetheless, if any attack was made on a host, we noted the time period and labeled any sketches that were made during those times as anomalous.

With results like these, and especially on real-world data sets, it is often more useful to analyze specific instances where the model correctly and incorrectly identified anomalous activity. Figure 8 shows the graph sketch vector, and anomaly score respectively for two hosts, FiveDirections-1 and FiveDirections-2, as they undergo very similar attacks. In both figures, the first red highlighted region is the initial compromise, where the Drakon APT attack injects malware into a Firefox process. In the second red region, the malware activates and begins communicating with a C2 server.

In both instances, the anomalous network activity produces strong signals, while the initial compromise does not. This is likely because the provenance graphs in the benign data set form a very thorough picture of how, and with whom each machine communicates. But, as pointed out by [**?**], individual browsing activity is highly varied, making it difficult to thoroughly summarize what is "normal", so the malicious download goes unnoticed.

Additionally, in both cases, the C2 connection is not severed for several hours, which makes it difficult to decide how to label this region. But we note that the graph sketches in Figure 8a, after the initial perturbation when the C2 connection was opened, remain in their perturbed state until the end of the day. This lines up with the periodic heartbeats it was sending to the C2 server after it was compromised. The anomaly score for the other host, similarly stays high for the rest of the day, for the same reason.

These results are encouraging. We have already shown that with clean datasets, EdgeTorrent is highly effective. We feel that the drop in performance here is due largely to the imprecise labels, and the large scope of benign activities, much of which we could not train with.

7.4 Efficiency Analysis

To further support our claims that EdgeTorrent is capable of real-time analysis of provenance data, we perform an experiment to measure its runtime performance. Our analysis is conducted on a single Intel Xeon E5-2690 v4 CPU [**?**] processing 1 million edges at varying levels of granularity. These results are shown in Figure 9. For comparison, we also plot the results reported by Unicorn of their speed tests using the best settings.

The graph being processed in Figure 9 is a TRACE graph, one of the most edge-dense provenance graphs available. The one-million edges were recorded over the course of 27 real-time minutes. For clarity, we also plot the rate at which new edges are observed in real-time. As is evident, our method, as well as Unicorn fall well

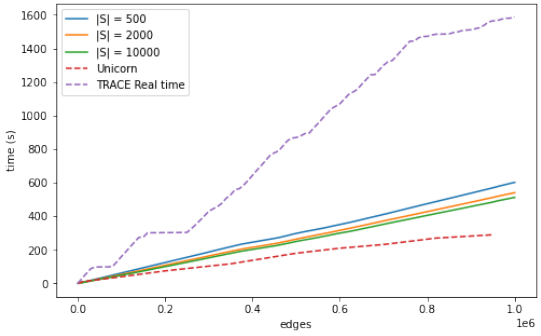


Figure 9: A comparison of runtimes for graph embedding using EdgeTorrent at varying levels of granularity compared to Unicorn. Note that the data provided for Unicorn only includes times for $|\mathcal{E}| < 0.95e6$

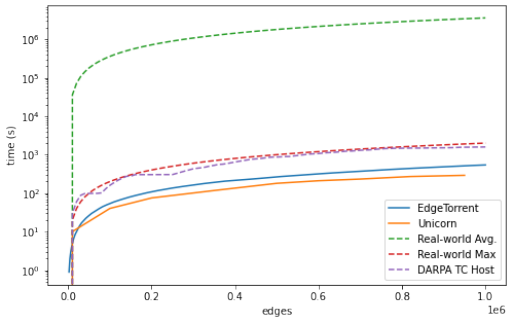


Figure 10: Comparison of real-world edge rates to the processing rates of EdgeTorrent and Unicorn

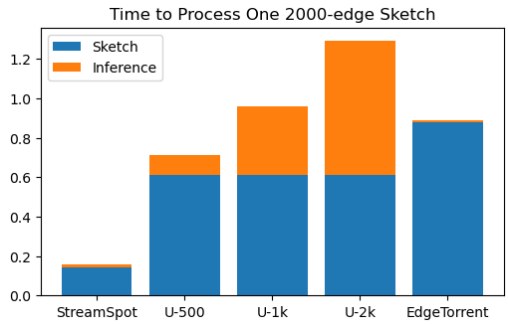


Figure 11: Time to run the full pipeline to ingest and analyze a single new sketch with various graph kernel models. Bars denoted U−∗ represent Unicorn with ∗ trained models.

under the required speed for real-time analysis, with our method slightly slower than Unicorn, but not so slow that it cannot handle real-time processing with quite a large margin for error.

Because a larger sketch size, $|S|$, entails fewer calculations of Equation 7 and fewer write operations, when $|S|$ is set to 10,000, EdgeTorrent runs slightly faster, taking about 500µs per-edge compared to 600µs when $|S| = 500$. We also note that our implementation of EdgeTorrent uses pure Python, and could be further optimized with C++; we leave this as a topic for future work.

Additionally, we obtained real-world endpoint event volumes from hundreds of thousands of monitored servers, workstations, laptops, and network devices within a large multinational technology corporation. This data showed that on average, their endpoint monitoring systems receive 1,000 events per hour with a maximum of 1.8 million per hour. We show these rates compared to the graph kernels' sketch rates in Figure 10, as well as the edge rate for TRACE from the previous figure. As is evident from the figure, both sketching methods operate far faster than an average workload would require. Furthermore, when the real-world system experiences high traffic, both methods are still quite capable of sketching the edges in real-time. In fact, the TRACE dataset, which we have already shown can be sketched in real time, has a higher rate of activity than what is observed in the real world. For these reasons, we feel confident in our assessment that EdgeTorrent is capable of real-time graph sketching.

Generating graph sketches is only the first part of the pipeline, however. So, we also provide a runtime analysis of how long generating sketches, and performing inference takes for each graph kernel method tested. Though Unicorn does provide a more efficient way of generating graph sketches, its method of performing inference is inefficient. Because it generates a new clustering model for each training graph, its time for inference grows in proportion to the number of training samples it was given.

Figure 11 shows the full picture of how long each pipeline takes to fully conduct inference when a new sketch appears. Though Unicorn is moderately faster at sketching compared to EdgeTorrent, after it has seen 500 training samples, it runs slower than EdgeTorrent. This is a realistic number, as we see in the very first experiment. Thus, we can say with confidence that our method is at least as efficient as the state-of-the-art when dealing with a realistic quantity of data. However, we concede, our method is still slower than StreamSpot. But given the incredible improvement between StreamSpot and EdgeTorrent, we feel this is an acceptable trade-off, especially given that typical edge rates are so much slower than all of these methods' processing time.

8 RELATED WORK

Supervised & Signature-based Approaches Prior works [? ? ?] use subgraph matching techniques to identify known malicious signatures within provenance graphs. Prior works [? ?] use expert-designed patterns with semantic meaning to condense large provenance graphs into smaller, easier to analyze ones, which they represent with neighborhood embedding. Supervised learning approaches are also popular, where rather than known signatures, techniques such as [? ?] use known malicious traces within the system to train on. Unfortunately, these works all require expert information, or labeled data to train. Furthermore, these approaches are not robust to zero-day vulnerabilities, which will, by definition, not be a part of the training sets of malicious data these models learn from. To circumvent these issues, it is necessary to use an anomaly-based intrusion detection system.

Anomaly-based approaches Anomaly detection in sequences has long been studied, both from a pure statistical approach [? ? ? ?] and through the lens of machine learning [? ?]. Prior works [?] and [?] both use GANs to analyze network and host events, respectively. But neither incorporate the graph structure present in both. Earlier works on provenance graph anomaly detection [? ?] use counts of individual events to determine the graphs' maliciousness; [?] characterizes system entities by their ego-nets. However, all these works do not look beyond first-order entity relations if at all.

To bridge this gap, ShadeWatcher [?] uses first-order relations to generate knowledge graph embeddings of entity relationships, then enriches these embeddings by further processing them with a graph neural network. However, this method is not inductive, meaning as new entities appear, new knowledge graph embeddings for them and any nodes they interacted with must be generated . Similarly, ProvDetector [?] and prior work [?] use skip-gram encodings of graph paths to generate path and node representations, respectively. But again, these techniques are not inductive and must be retrained as unseen nodes appear. ThreaTrace [?] identifies anomalous nodes individually by using an ensemble of GraphSAGE models to predict

node types given observed edges on subgraphs of the provenance graph. Like Unicorn, as more training data becomes available, it requires more models for inference. Similarly, Sigl [**?**] encodes nodes embedded with word2vec [**?**] with Graph LSTMs [**?**] and attempts to reconstruct their original embeddings. High reconstruction loss indicates likely malicious processes. However, their approach is not suitable for enterprise-wide monitoring. The authors specify that it is meant to monitor the well-defined graphs generated during software installations.

None of these prior methods considered time. Midas [**?**] views streams of edges and learns to identify anomalous bursts of activity over time, but is highly specific in what it can detect. Euler [**?**] uses MPNNs on temporal snapshots and passes the node embeddings through RNNs for link prediction on authentication events; however, its temporal embeddings are not continuous. Graph-kernel methods [**? ?**] analyze representations of the graph in its entirety as it changes over time and identify anomalous graphs as a whole. Time is represented continuously, rather than through arbitrary snapshots. Additionally, these methods are built with streaming in mind, so they are optimized for both speed and precision.

9 DISCUSSION

Limitations of EdgeTorrent. As with any graph-kernel method for IDS, it is difficult to attribute alerts to individual events. Rather, EdgeTorrent detects anomalous periods of time. While this has value, for example, to analyze the data provenience from a single target source node, it is often less granular than a security analyst may want. However, as the sketch size is variable, this problem can be somewhat alleviated if one wishes to isolate the most anomalous sketch. In this way, the problem can be narrowed down to the $|S|$ events that occurred during that period of time. We feel that this is less a limitation of the embedding method, and more a limitation of the experiments we conducted. Rather than compressing each node embedding into a single graph sketch, it is very likely that tracking node embeddings on their own will produce trajectories similar to those produced by the graph sketches. However, on larger graphs, this produces a great deal of data, generating $|\mathcal{V}| \times t \times d$ dimensional matrices. If instead we could identify a small subset of nodes worthy of monitoring and store just their changing embeddings, this approach would be more feasible, but this is a non-trivial problem, so we intend to explore it in future works.

Also true of any anomaly-based IDS, this method assumes that it has enough benign data to learn what normal activity looks like. This assumption may not always be the case, and in the real world, generally is not. However, the cost of retraining the model when new data becomes available is low. Regardless, these models are susceptible to higher rates of false positives than signature-based systems but require no anomalous data to train. Though this trade-off is not unique to this work, it is still something worth considering. We also note that on the large endpoint dataset, EdgeTorrent performed poorly when trying to detect activity on individual machines, but excelled at detecting anomalous network activity. It may be that the broad scope of the activities we monitored resulted in too much noise for the model to properly learn the distribution of benign activities. As [**?**] notes, in anomaly-based IDS, models tend to perform better, and have results that are easier to explain when more minimal feature sets are used. Rebuilding the graphs and omitting all but host-focused or network-focused nodes may provide better results.

Another limitation is the inability to efficiently process longer streams of graph sketches. Due to the requirement of transformers to generate an $n \times n$ matrix for an input of length n, the memory requirements increase rapidly with growing sequence length. We avoid this problem in this work by breaking the sketch sequences into smaller sub-sequences for processing, but this presents problems for detecting long-term activities within the system. Ideally, the global view taken by the sketch generation portion of EdgeTorrent will carry this information forward through time, but this is a limitation. One solution to this is making sketch sizes larger, so as to integrate longer periods of time, but this comes at the cost of some information loss. Alternatively, efficient transformer architectures could be used that are designed for long sequences [**? ?**].

In spite of these limitations, we have shown that EdgeTorrent outperforms the prior works by a significant margin. Because it incorporates both a continuous graph sketching function and a more powerful anomaly detection model, it is able to detect malicious activity that the prior works were unable to, while also alerting on fewer false positives. At the same time, EdgeTorrent remains fast enough to stream in real time.

Future work. Investigating if better alert attribution can be extracted from EdgeTorrent's output is worth further research. We feel that using existing transformer attribution strategies [**? ?**] to identify the most relevant graph sketches, it may be possible to further extract the nodes that contribute most heavily to those alerts. Additionally, we would like to explore the effect of using a different model for sketch analysis. It is possible a simpler machine learning approach such as [**? ? ? ?**] or even a simple statistical model such as [**? ? ? ?**] could yield similar or better results during anomaly detection. Lastly, using a more advanced method to convert from node embeddings to graph sketches is worthy of further exploration. We use a simple weighted average based on time decay, but using methods such as attention instead of temporal decay, or LSTM aggregation as proposed by [**?**] may produce more expressive graph sketches.

10 CONCLUSION

In this work, we presented a novel method of encoding a provenance graph in real-time. The EdgeTorrent pipeline trains an MPNN on a small subset of a much larger graph, then efficiently streams in a large edge list to generate high-quality graph sketches. These graph sketches are then analyzed with an adversarially trained sequence anomaly detector to determine if they were part of a normal or anomalous graph. This work shows that it is possible to embed provenance graphs in real-time with continuous graph sketches, rather than hash-based ones, making them less sensitive to minor perturbations, while retaining the ability to detect unusual activity. We demonstrated through several experiments that this method outperforms state-of-the-art graph-kernel methods for anomaly-based intrusion detection and runs fast enough for real-time monitoring in a realistic setting.

11 ACKNOWLEDGEMENTS

The authors thank the anonymous reviewers and shepherd for their helpful suggestions. This research was developed in part with funding from the Defense Advanced Research Projects Agency (DARPA), and National Science Foundation under grants 1618706, 1717774, and 2127207. The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

A MODEL PARAMETERS

The MPNN used for all EdgeTorrent models in this work was a 3-layer Graph Convolutional Network with 128-dimensional hidden and output layers. The activation used between layers and for the output was hyperbolic tangent. Specific implementation details of the GANs are shown in Table 7.

Table 7: GAN Hyperparameters

Data set	α	β	γ		LR	Hidden	Layers	Latent	Heads
StreamSpot	0.5	10	0.1	Disc	0.001	256	2	–	32
				Gen	0.001	256	2	64	–
UNSW-NB15	0.5	10	0.1	Disc	0.001	512	4	–	32
				Gen	0.01	256	4	32	–
CADETS	0.7	10	0.01	Disc	0.00001	16	4	–	4
				Gen	0.0001	256	2	256	–
5D	0.7	1	1	Disc	0.0001	32	4	–	4
				Gen	0.001	256	2	256	–
TRACE	0.3	0.1	0.01	Disc	0.001	32	4	–	16
				Gen	0.0001	256	2	256	–

To find these hyperparameters, we performed a grid search and evaluated the model on a validation set. The search space for the parameters is shown in Table 8. However, note that some combinations of parameters were not possible due to space constraints on the GPUs used (e.g 512 hidden dimensions with 4 layers and 32 heads), so not all combinations were evaluated, just those that did not produce out-of-memory errors. The results reported in Section 7 use the best hyperparameters on the validation sets, and apply them to the test sets.

Hyperparameter	Search space
α	{0.3, 0.5, 0.7}
β	{0.1, 1, 10}
γ	{0.01, 0.1, 1}
LR	{1e-4, 1e-3, 1e-2}
Hidden	{16,32,64,128,256,512}
Layers	{2, 3, 4}
Latent	{32,64,128,256,512}
Heads	{4,8,16,32}

Table 8: Hyperparameter Search Space