2018 IEEE International Conference on Data Mining

SEDANSPOT: Detecting Anomalies in Edge Streams

Dhivya Eswaran
Carnegie Mellon University, Pittsburgh, USA
deswaran@cs.cmu.edu

Christos Faloutsos
Carnegie Mellon University, Pittsburgh, USA
christos@cs.cmu.edu

***Abstract*—Given a stream of edges from a time-evolving (un)weighted (un)directed graph, we consider the problem of detecting anomalous edges *in near real-time using sublinear memory*. We propose SEDANSPOT, a principled randomized algorithm, which exploits two tell-tale signs of anomalous edges: they tend to (i) occur as bursts of activity and (ii) connect parts of the graph which are sparsely connected. SEDANSPOT has the following desirable properties: *(a) Burst resistance:* It provably downsamples edges from bursty periods of network traffic, *(b) Holistic scoring:* It takes into account the whole (sampled) graph while scoring the anomalousness of an edge, giving diminishing importance to far-away neighbors, *(c) Efficiency:* It supports fast updates and scoring and hence can be efficiently maintained over stream; further, it can detect anomalous edges in sublinear space and constant time per edge. Through experiments on real-world data, we demonstrate that SEDANSPOT is $3\times$ faster and 270% more accurate (in terms of AUC) than the state-of-the-art.**

I. INTRODUCTION

Time-evolving (un)weighted (un)directed graphs, where edges and vertices arrive continuously over time, are becoming increasingly ubiquitous, e.g., phone call, instant messaging, e-mail and IP-IP networks. In these settings, edges are generated in increasing order of timestamps, giving rise to *edge streams*.

We consider the problem of *near real-time anomaly detection* in such edge streams, where the goal is to detect whether an incoming edge is anomalous or not, as soon as it is received. While online graph anomaly detection is a well-explored research area, most methods assume edges have been aggregated into graph snapshots (see Sec. II). In contrast, we seek algorithms which directly process the edge stream to flag anomalies in near real-time, which is crucial in order to curtail the impact of malicious activities and kick-start recovery processes in a timely manner. Moreover, given that the number of vertices is not known a priori and can grow as the stream progresses, the algorithm should operate in memory sublinear in graph size. Informally, the problem we set out to solve is:

Informal Problem 1. *Given an edge stream $\mathfrak{E}=\{e_1, e_2, \ldots\}$ from a/an (un)weighted (un)directed graph, detect whether e_i is anomalous, in near real-time using sublinear memory.*

As the definition of anomaly can be context-dependent, we focus on detecting edges *which connect sparsely-connected parts of graph* (bridge edges). Fig. 1 illustrates this using an edge stream from an unweighted directed graph where the edges received until time t=0 form two clusters of vertices $\{(a_1, \ldots, a_5), (b_1, b_2, b_3)\}$. Thus, edges $a_4 \rightarrow b_2, a_4 \rightarrow b_1$ and $a_4 \rightarrow b_3$ (occurring at t=7) which connect these otherwise disconnected clusters of vertices should be flagged anomalous.

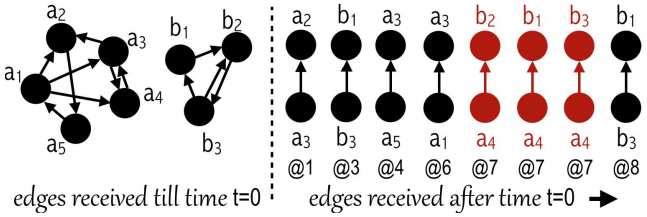


Fig. 1. An edge stream with a burst of three anomalous 'bridge' edges (red).

The *simultaneous occurrence* of these 'red' edges is not coincidental. Prior work has shown that fraudulent or important events in many applications indeed occur as *spikes or bursts of activity* [1] – e.g., network security threats (port-scan, denial-of-service), scams (malicious entities attacking many victims before they get exposed), occasions (festivals producing a burst of longer-than-usual phone calls), etc. Anomaly detection approaches which do not account for this observation ([2], [3]) tend to miss several anomalies, e.g., $a_4 \rightarrow b_3$ being masked as normal by the recent $a_4 \rightarrow \{b_1, b_2\}$. Note that, while anomalous activity tends to occur as bursts, burstiness does not necessarily signify an anomaly: in dynamic situations like network traffic, normal activity can also be bursty. Thus, for reliable detection, we need to combine both *temporal* and *structural* information. As we will see, the proposed SEDANSPOT does precisely this.

Given the running time and memory constraints of Problem 1, SEDANSPOT (Streaming EDge ANomaly SPOTter) maintains an online sample of edges (using SEDANSAMPLER) which is then used to score the anomalousness of any new edge (via SEDANSCORER). It has the following properties. *(a) Burst resistance:* SEDANSAMPLER provably downsamples edges from bursts of activity, *(b) Holistic scoring:* SEDANSCORER scores the anomalousness of edges by taking into account the whole (sampled) graph, giving diminishing importance to far-away neighbors, *(c) Efficiency:* SEDANSPOT supports fast updates and scoring and hence can be efficiently maintained over stream; further, it can detect anomalous edges in sublinear space and constant time per edge.

We use publicly-available datasets and open-source our code at www.github.com/dhivyaeswaran/sedanspot. The supplementary material (with proofs and additional experiments) is at www.cs.cmu.edu/deswaran/papers/icdm18-sedanspot-sup.pdf.

II. RELATED WORK

Graph anomaly detection is a well-studied problem [4], [5]. Here, we review only *online* graph anomaly detection methods.

Graph streams: Many methods assume that the raw edge

978-1-5386-9159-5/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDM.2018.00117

TABLE I
QUALITATIVE COMPARISON WITH CLOSELY-RELATED PRIOR WORK

Property	DC [11], CAD [10]	AH [12]	GOUTLIER [2]	RHSS [3]	SEDANSPOT
Online (operate on edge streams)		✔	✔	✔	✔
Generality (weighted/directed)				✔	✔
Burst resistance	N/A				✔
Holistic scoring	✔				✔
Efficiency (sublinear memory)			?	✔	✔
Efficiency (constant time per edge)	N/A		✔	✔	✔

stream has been processed into a stream of graph snapshots (each containing edges from a given duration) before detecting anomalies. Among many others, these include: [6] to detect the sudden (dis)appearance of dense subgraphs using sketching, [7] and [8] to detect change points using graph partitioning/decomposition, [9] to detect evolutionary community outliers, [10] and [11] to compare consecutive graph snapshots through similarity/distance functions related to random walks.

Edge streams: In contrast, methods operating directly on the edge stream are relatively few. These include: GOUTLIER [2] to score the likelihood of each edge in the stream based on a structural reservoir sample of edges, [12] to detect anomalous nodes using egonet-level Principal Component Analysis and [3] to score edge anomalousness in the stream based on its prior occurrence, preferential attachment and mutual neighbors (homophily). [13] is related, but applies only when *multiple* graphs with *typed* nodes and edges evolve simultaneously.

As such, none of the prior methods have all the desirable properties that SEDANSPOT exhibits, as shown in Table I.

III. BACKGROUND

Reservoir sampling [14] is a classic algorithm to maintain a fixed-size uniform sample of elements in a stream. Weighted reservoir sampling [15] is used when elements are to be sampled with different weights. When the stream contains edges from a graph, several sampling mechanisms are available [16], but none of them downsamples edges from bursty periods.

Random walk with restart (RWR) relevance score of v w.r.t. u is the steady state probability that the surfer will finally remain at v during a random walk from u, with restart probability α [17]. Concretely, if $\bar{\mathbf{A}}$ is the $n \times n$ row-normalized adjacency matrix and $\mathbf{q}_u$ is the n-dimensional binary vector where all but the u^{th} entry are zeros, the vector of RWR relevance scores $\mathbf{r}_u$ of all vertices w.r.t. u is given by:

$$\mathbf{r}_u = (1-\alpha)\bar{\mathbf{A}}^T\mathbf{r}_u + \alpha\mathbf{q}_u \quad (1)$$

As computing Eq. (1) directly can be expensive, approximations such as local random walks (which we use) have been successfully applied, e.g., in link prediction [18] and recommendation [19]. Existing work on RWR relevance score computation for edge streams either assume a single start vertex known ahead of time [20] or maintain all-pair relevance scores [21]; thus they are not applicable to our setting.

IV. PROBLEM FRAMEWORK

Let $\mathfrak{E} = \{e_i\}_{i=1}^{\infty} = \{e_1, e_2, \ldots, e_L, \ldots\}$ be the stream of edges from an underlying time-evolving graph $\mathfrak{G}$. Each element e_i in the stream is 4-tuple (u_i, v_i, w_i, t_i) of its source vertex $u_i \in \mathcal{V}$, its destination vertex $v_i \in \mathcal{V}$, its edge weight w_i and its time of occurrence t_i and represents the addition of this edge to the graph $\mathfrak{G}$. Here, $\mathcal{V}$ is the set of all vertices, which is not known a priori but changes as $\mathfrak{G}$ evolves. However, each vertex is assumed to have a unique identifier, e.g., user ID or IP address, that is fixed over time.

Here, $\mathfrak{G}$ can be a *multigraph*, i.e., two vertices may be connected multiple times with different weights. Many edges could arrive simultaneously, i.e., $t_{i+1} \geq t_i$ (equality allowed). The edges can also be weighted ($w_i = 1\ \forall\ e_i$, if unweighted) and/or have direction (assume a 'fake' (v_i, u_i, w_i, t_i) for every (u_i, v_i, w_i, t_i) when $u_i \neq v_i$, if undirected). We will overload $t(e)$ to denote the timestamp of edge e.

Our goal is to detect anomalous edges by leveraging their temporal and spatial signals, i.e., they tend to (i) occur as bursts of activity and (ii) connect sparsely-connected parts of the graph. To do this quickly using bounded memory, we maintain a fixed-size sample of the edges seen thus far and use it to score the anomalousness of any new edge. Thus, Problem 1 can be subdivided into two subproblems, each incorporating one of the above signals of anomalousness, as follows:

Informal Problem 2 (Edge sampling). *Given an edge stream $\mathfrak{E}$ and $S \in \mathbb{N}$, maintain an online sample $\mathcal{S}$ of S edges while downsampling bursts of activity.*

Informal Problem 3 (Anomaly scoring). *Given a sample of edges $\mathcal{S}$ and a new edge e_i, design an anomaly scoring function $y_i = f(e_i; \mathcal{S})$ to give a higher score to edges connecting parts of the graph which are sparsely connected.*

V. PROPOSED METHOD

Alg. 1 gives the high-level pseudocode of SEDANSPOT. Every edge e_i in the stream is first compared to the current sample of edges via SEDANSCORER (Sec. V-B, addressing Problem 3) to determine its anomaly score. The sample is subsequently updated based on this edge using SEDANSAMPLER (Sec. V-A, addressing Problem 2). We describe the algorithm below assuming *directed edges*. Extensions and theoretical analysis are discussed in Sec. V-C and Sec. VI respectively.

A. Edge sampling using SEDANSAMPLER

Given a sample size $S \in \mathbb{N}$, the core idea of SEDANSAMPLER is to maintain a *rate-adjusted sample* $\mathcal{S}$ of S edges.

Definition 1 (Rate-adjusted sample). *$\mathcal{S}$ is said to be a rate-adjusted sample from a stream $\mathfrak{E}$ iff $\mathbf{Pr}[e \in \mathcal{S}] \propto 1/r(e)\ \forall\ e \in \mathfrak{E}$, where $r(e)$ is the edge rate at the time of occurrence of e.*

Here, $r(\cdot)$ is a measure of *edge rate* such that a larger value signifies a more intense burst of edges. Intuitively, rate-adjusted reservoir sampling ensures that, if a region R of an underlying graph $\mathfrak{G}$ is densely connected *solely* because of attack edges which occurred during bursts of activity, the

Algorithm 1 SEDANSPOT

Input: edge stream $\mathfrak{E} = \{e_i\}_{i=1}^{\infty}$
Output: stream of anomaly scores $\{y_i\}_{i=1}^{\infty}$

SEDANSAMPLER.INITIALIZE()
SEDANSCORER.INITIALIZE()
for edges $\mathcal{E}_t$ received at time t from stream $\mathfrak{E}$ **do**
  for $e_i \in \mathcal{E}_t$ **do**
    $y_i \leftarrow$ SEDANSCORER.ANOMALY_SCORE(e_i)
    $e_{rem}, e_{add} \leftarrow$ SEDANSAMPLER.SAMPLE(e_i)
    SEDANSCORER.ADD(e_{add}) **if** e_{add} is not None
    SEDANSCORER.REMOVE(e_{rem}) **if** e_{rem} is not None
    yield y_i

Algorithm 2 SEDANSAMPLER

Parameter(s): sample size S

procedure INITIALIZE
  $\mathcal{S} \leftarrow$ MinHeap-PriorityQueue(size S)
  ▷ *stores top S edges with the highest priorities; incurs $\mathcal{O}(\log S)$ addition, $\mathcal{O}(1)$ MPE (min. priority element) retrieval costs*
procedure SAMPLE(edge e)
  $x \sim$ Uniform$[0, 1]$
  $p \leftarrow x^{r(e)}$ ▷ *priority of e; $r(e)$ is defined in Eq. (2)*
  if $\mathcal{S}$.is_full() **then**
    $e', p' \leftarrow H$.peek() ▷ *current MPE and its priority*
    if $p' < p$ **then**
      $\mathcal{S}$.pop() ▷ *remove current MPE, i.e., e'*
      $\mathcal{S}$.insert(edge e with priority p)
      return e', e
    else ▷ *leave sample unchanged*
      return None, None
  else ▷ *heap is not full, simply add e*
    $\mathcal{S}$.insert(edge e with priority p)
    return None, e

corresponding region in the sampled graph induced by $\mathcal{S}$ still remains somewhat sparsely connected. This sets the stage to detect a subsequent edge belonging to the same attack and occurring in the same region R as an anomaly w.r.t. the sample.

While other characterizations are possible, we use edge rate $r(e)$ defined below due to its theoretical guarantee (Thm. 1) and its ease of computation in a stream using $\mathcal{O}(1)$ space:

$$r(e) = \left|\mathcal{E}_{t(e)}\right| / \left(t(e) - t_{bef}(e)\right) \tag{2}$$

Here, $t(e)$ is the timestamp of edge e, $\mathcal{E}_{t(e)}$ is the set of edges which arrive at time $t(e)$ (including e) and $t_{bef}(e) = \max_{e' \text{ s.t. } t(e')<t(e)} t(e')$ denotes the timestamp of the most recent edge which arrived strictly before e. The larger the number of edges occurring at $t(e)$ or the smaller the time gap between e and the edge(s) occurring before, the more intense is the burst of edges. Accordingly, Eq. (2) ensures that $r(e)$ is higher. It also assigns the same rate to edges arriving simultaneously. Having now defined $r(\cdot)$, a rate-adjusted sample can be easily maintained using weighted reservoir sampling [15]. The resulting algorithm (Alg. 2) uses a MinHeap-PriorityQueue data structure for efficient $\mathcal{O}(\log S)$ updates.

Algorithm 3 SEDANSCORER

Parameter(s): restart probability α, number of walks N

procedure INITIALIZE
  $A \leftarrow$ Hash table mapping vertices to their LATs
procedure ADD(edge $e = (u, v, w, t)$)
  $A[u]$.increment(v, w)
procedure REMOVE(edge $e = (u, v, w, t)$)
  $A[u]$.decrement(v, w)
procedure SAMPLE_NEIGHBOR(vertex u_*, edge $e = (u, v, w, t)$)
  ▷ *samples neighbor of u_* from $\mathcal{S} \cup \{e\}$ based on edge weight*
  if e is None **or** $u_* \neq u$ **then**
    return $A[u_*]$.random_key()
  else
    $W \leftarrow w +$ out-weight of u_* in $\mathcal{S} \cup \{e\}$
    return v w.p. w/W **else** $A[u_*]$.random_key()
procedure VISIT_FRACTION(vertex u, vertex v, edge e)
  ▷ *outputs an estimator $\hat{s}(v \mid u; \mathcal{S} \cup \{e\})$ for relevance score*
  initialize *num_steps* $\leftarrow 0$, *num_visits* $\leftarrow 0$
  for $i = 1, \ldots, N$ **do**
    walk length $\ell \sim$ Geometric(α)
    num_steps $\leftarrow$ *num_steps* $+ \ell$
    current vertex $a \leftarrow u$
    for $j = 1, \ldots, \ell$ **do**
      num_visits $\leftarrow$ *num_visits* $+ \mathbb{I}(a == v)$
      $a \leftarrow$ SAMPLE_NEIGHBOR(a, e)
      break if a has no outgoing edges in $\mathcal{S} \cup \{e\}$
  return *num_visits*/*num_steps*
procedure ANOMALY_SCORE(edge $e = (u, v, w, t)$)
  visit_frac_before $\leftarrow$ VISIT_FRACTION$(u, v,$ None$)$
  visit_frac_after $\leftarrow$ VISIT_FRACTION(u, v, e)
  return max(0, *visit_frac_after* $-$ *visit_frac_before*)

B. Anomaly scoring via SEDANSCORER

Intuitively, given a sample of edges, a new edge e is more surprising (anomalous) if adding it to the sample produces a larger change in the proximity (distance) between its incident vertices. Thus, SEDANSCORER scores the edge anomalousness as $f(e; \mathcal{S}) = \text{MPI}(e; \mathcal{S})$ where MPI$(\cdot)$ is as defined below:

Definition 2 (Marginal proximity increase). *The marginal proximity increase measure of edge e from source u to destination v w.r.t. a set of edges $\mathcal{S}$ is given by*

$$\text{MPI}(e; \mathcal{S}) = s(v \mid u; \mathcal{S} \cup \{e\}) - s(v \mid u; \mathcal{S}) \tag{3}$$

Here, $s(v \mid u; \mathcal{S})$ is a measure of *directed vertex proximity* between source u and destination v based on edges $\mathcal{S}$, such that greater the number of shorter, heavily weighted paths from u to v in $\mathcal{S}$, the higher is its value. We use the RWR relevance score in Eq. (1) for this purpose, as it is principled (incorporating direct and indirect paths), asymmetric, bounded in $[0, 1]$ and can be estimated fast using local random walks.

Local random walks are a principled way to estimate Eq. (1) in time nearly independent of sample size S. Alg. 3 gives the pseudocode. A is a data structure holding the current sample of

edges. Given parameters N and α, VISIT_FRACTION() outputs an estimate $\hat{s}\left(v \mid u; \mathcal{S} \cup \{e\}\right)$ by performing N local random walks. Each time, a walk length ℓ is sampled based on the restart probability α (line 16). Then, ℓ steps (possibly less if there are dead ends) of a random walk starting from source vertex u are taken, each time sampling a neighbor of the current vertex proportional to edge weight in $\mathcal{S} \cup \{e\}$ (line 17-22). The ratio of the number of times v is visited in this process to the total walk length is returned as the estimate $\hat{s}$.

As the sampling routine in Alg. 2 is used more often than updates, further optimization is possible using Alias method [22]. Given an arbitrary discrete distribution with k outcomes, Alias method can produce a sample in $\mathcal{O}(1)$ time by incurring an $\mathcal{O}(k)$ preprocessing cost upfront. Since neighbor sampling is equivalent to sampling from a discrete distribution, we propose to use a *hash table of Lazy Alias Tables* (LATs), one LAT per vertex, as our data structure A. Assuming the LATs are up-to-date, SAMPLE_NEIGHBOR() takes only $\mathcal{O}(1)$ time. When an edge is added to or removed from the sample, only the LATs of affected vertices have to be updated. Moreover, even these updates can be done in a lazy fashion, i.e., only when we need to sample a neighbor of an affected vertex. Note that, when computing $\hat{s}\left(v \mid u; \mathcal{S} \cup \{e\}\right)$ in Eq. (3), the edge e should *not* actually be inserted into the data structure – this would force unnecessary updates to LATs, incurring a large overhead. See line 12 of Alg. 3 for how to avoid this.

C. Extensions

SEDANSPOT can be extended in many ways: (i) to handle undirected (by symmetrizing the MPI measure in Eq. (3)) and bipartite settings (by allowing forward and backward jumps); (ii) to bias the sample towards recent edges by modifying line 5 of Alg. 2 to incorporate edge recency; (iii) sampling edges proportional to any monotonically decreasing function of their rate, $f(r(e))$, to downsample bursts. Using $f(x)=1/x$ leads to Thm. 1, but other variants may work better in practice.

VI. THEORETICAL ANALYSIS (PROOFS: SUPPLEMENTARY)

We begin by showing that Alg. 2 is indeed correct:

Lemma 1 (Correctness of Alg. 2). *Alg. 2 maintains a rate-adjusted sample $\mathcal{S}$, as defined in Def. 1, from stream $\mathfrak{E}$.*

Importantly, SEDANSAMPLER ensures that the number of sampled edges belonging to a given time interval only depends on its duration and not on the number of edges occurring during it. In the following, a time tick τ is said to be anchored if some edge occurred at time τ, i.e., $\exists$ edge e s.t. $\tau{=}t(e)$.

Theorem 1 (Burst resistance). *Consider time ticks $\tau_0{=}0$ and $\tau_1 \leq \tau_2 \ldots {\leq}\tau_K$ which are anchored. Let $\mathcal{H}_k$ be the set of edges arriving in time interval $I_k := (\tau_{k-1}, \tau_k]$ of duration $\ell_k = \tau_k - \tau_{k-1}$. If $\mathcal{S}$ is the rate-adjusted sample till time τ_K,*

$$\mathbf{Pr}\left[e \in \mathcal{H}_k \mid e \in \mathcal{S}\right] = \ell_k / \sum_{k=1}^{K} \ell_k, \ \forall\ k \tag{4}$$

which is independent of $|\mathcal{H}_k|$.

This is advantageous given the tendency of anomalous edges to occur as bursts of activity: even though many attack edges occur in a small duration, rate-adjusted sampling ensures that only a few of them are stored in the sample. See Example 1.

Example 1. *Consider a 'normal' process generating $1M$ edges over 100 hours followed by an attacker producing $0.5M$ edges in 10 minutes. Using Eq.* (4)*, the expected number of attack edges in the sample is $10/(100 \times 60 + 10) < 0.2\%$. In contrast, the sample $\mathcal{S}'$ produced by Uniform Reservoir sampler, which samples edges uniformly, i.e., $\mathbf{Pr}\left[e \in \mathcal{S}'\right] \propto 1\ \forall e$, has $0.5M/1.5M = 33.3\%$ attack edges in expectation.*

Next, we show that SEDANSPOT meets the sublinear memory and constant time per edge requirements of Problem 1.

Lemma 2 (Constant scoring time per edge). SEDANSPOT *takes at most $\mathcal{O}(N/\alpha)$ time in expectation (usually lesser in practice) to compute anomaly score of an edge (line 5, Alg. 1).*

Lemma 3 (Sublinear memory). SEDANSPOT *takes $\mathcal{O}(S \log |\mathcal{V}|)$ memory to process each edge.*

We also show that updates of SEDANSPOT are fast, amortized over the stream length L, as the updates become less frequent as the stream progresses. Let d_{avg} be an upper bound on the average vertex degree in the sample and let $H_n = \sum_{i=1}^{n} i^{-1}$ be the sum of first n terms in the harmonic series. Also, let the edge rate be bounded in $[r_{\min}, r_{\max}]$. Then,

Lemma 4 (Fast amortized updates per edge). SEDANSPOT *takes at most $\mathcal{O}\left(\frac{\log S + d_{avg}}{L} \cdot \left(S + \frac{r_{\min}}{r_{\max}}(H_L - H_S)\right)\right)$ amortized time in expectation for updates (lines 6-8, Alg. 1).*

As $H_n = \log n + \gamma + \mathcal{O}\left(\frac{1}{n}\right)$ where γ is the Euler-Mascheroni constant, the amortized update time per edge remains small.

VII. EXPERIMENTS

We implement all methods in C++ and run experiments on MacOS High Sierra with 2.7 GHz Intel Core i5 processor and 16 GB main memory.

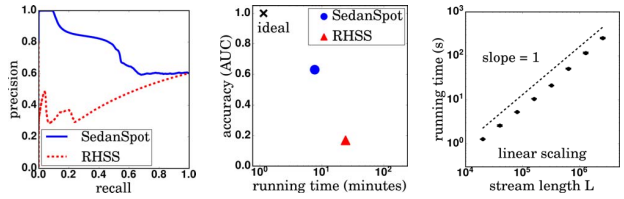
Datasets. We shortlist datasets where the anomalies are verifiable and/or interpretable. These are: **(a) DARPA** [23] <*srcIP, dstIP, 1, time*>: consisting of $4.5M$ directed edges of network traffic from $9.5K$ source IPs to $23.4K$ destination IPs over $87.7K$ minutes. 60% edges are labeled as anomalous (belonging to attacks, which mostly occurred in infrequent bursts). **(b) DBLP** [24] <*author1, author2, 1, pub_year*>: consisting of $55.5K$ authors and $3.7M$ coauthorships (undirected edges) over 20 years. **(c) ENRON** [25] <*sender, receiver, 1, date*>: containing $50K$ e-mails (directed edges) among the 151 people from May 1999 to April 2002. The supplementary material contains a detailed description.

Baseline. We use RHSS [3] (with $\delta{=}\epsilon{=}0.001$), the only edge stream anomaly detector which can be extended to directed and weighted edges (using separate sketches for in- and out- neighborhoods). We use the negative likelihood of edge probabilities as anomaly scores, giving equal importance to sample, preferential attachment and homophily scores.

TABLE II

PRECISION OF SEDANSPOT AND BASELINE (RHSS) AT DIFFERENT CUT-OFF RANKS k. BOLD SIGNIFIES HIGHEST VALUE IN EACH COLUMN AND UNDERLINE SHOWS SIGNIFICANT DIFFERENCES (p-VALUE ≤ 0.01) W.R.T. BASELINE ACCORDING TO A TWO-SIDED MICRO-SIGN TEST [26].

Method	precision@ 200K	400K	600K	800K	1000K	1200K	1400K	1600K	1800K
SEDANSPOT	**1.00** (0.00)	**0.97** (0.02)	**0.93** (0.01)	**0.89** (0.01)	**0.85** (0.01)	**0.83** (0.01)	**0.81** (0.01)	**0.80** (0.01)	**0.79** (0.01)
RHSS	0.49 (0.01)	0.36 (0.00)	0.29 (0.00)	0.29 (0.01)	0.32 (0.01)	0.35 (0.00)	0.36 (0.00)	0.36 (0.00)	0.33 (0.00)



(a) (b) (c)

Fig. 2. SEDANSPOT (a) achieves better recall and precision on DARPA dataset for all cut-off ranks k, (b) outperforms the baseline in terms of both accuracy and speed, (c) scales linearly with the input stream length.

Evaluation metrics. All the methods output an anomaly score per edge (higher is more anomalous). Sorting the edges in descending order of their scores, we count the number of edges c_k flagged correctly as anomalous among the top k edges, for every cut-off rank $k \in \mathbb{N}$. If C is the total number of ground truth anomalies, we compute: *precision@*$k = c_k/k$ and *recall@*$k = c_k/C$. We also report AUC (Area Under ROC).

Experimental design. *Q1) Accuracy:* How well does SEDANSPOT detect anomalies compared to the baseline? What is the trade-off w.r.t. running time? *Q2) Scalability:* How does it scale with input stream length L? *Q3) Discoveries:* Does it lead to interesting discoveries in practice? *Q4) Parameters:* How do accuracy and running time depend on parameters S, N and α? We answer Q1-3 here and Q3-4 in the supplementary.

Q1) Accuracy. We use N=100 walks, α=0.15 restart probability (recommended value [27]) and S=$10K$ sample size.

Precision, recall: Table II tabulates precision@k of RHSS and SEDANSPOT at 9 cut-off ranks $k \in [200K, 1800K]$. Despite the use of randomization, the standard deviations in results (shown in brackets) were low ($\leq$0.02) indicating a fairly consistent performance across multiple runs. We see that SEDANSPOT outperforms RHSS on all considered k values, achieving $100-215\%$ (statistically significant) improvements in precision. Further, a plot of precision vs. recall for all cut-off ranks (Fig. 2a) shows that SEDANSPOT (solid blue) lies completely above the baseline (dashed red), indicating that the performance gains generalize to all cut-off ranks k.

Accuracy vs. running time: Fig. 2b plots accuracy (AUC) vs. running time (in minutes, excluding I/O) averaged over five runs. Error bars are very low and hence omitted. We see that SEDANSPOT achieves a much higher accuracy (=0.63) compared to the baseline (=0.17), while also running faster (8 vs. 24 mins). This is a 270% accuracy improvement in $3\times$ less processing time. The main overhead of RHSS turns out to be the computation of pairwise independent hash functions.

Q2) Scalability. We vary the number of edges L in the input stream in eight logarithmic steps in $[20K, 2.56M]$, setting S=$10K$, N=100 and α=0.15. Fig. 2c, plotting running time vs. L in log-log scales, reveals a line of slope 1.0. This confirms the linear scalability of SEDANSPOT w.r.t. input stream length, thanks to its constant processing time per edge. Note that SEDANSPOT processes $2.56M$ edges in ~ 4 minutes and thus is very fast (speed of about $10.1K$ edges per second).

Q3) Discoveries on DARPA (rest in the supplementary).

To understand the relative contributions of SEDANSAMPLER and SEDANSCORER to the accuracy gain on DARPA dataset, we consider three intermediate versions of algorithms, modifying the baseline RHSS (which originally maintains counts over the *whole* stream, with 'no sampling') by (a) sampling alone using one of UR-SAMPLER (Uniform Reservoir) or SEDANSAMPLER, (b) scoring alone using SEDANSCORER, or (c) both. Fig. 3a summarizes the results.

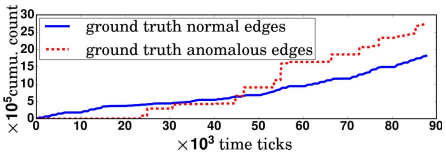
Using RHSS as scorer gives the same AUC with or without Uniform Reservoir sampling. This is because UR-SAMPLER, selecting each edge with equal probability, 'preserves' the fraction of anomalous edges while sampling, thus leading to similar results. Switching to SEDANSAMPLER improves AUC by 165%. Why? Fig. 3b, plotting the cumulative count of normal (solid blue) and anomalous (dashed red) edges over time, contrasts the smooth increase of normal edges to the step-like behavior of red curve which results from the bursty nature of network attacks. SEDANSAMPLER exploits this via rate-adjusted sampling which downsamples edges from bursty time periods and thus significantly decreases the fraction of anomalous edges ('corruption') in the sample, as shown in solid blue curve of Fig. 3c (while the dashed red curve of UR-SAMPLER stabilizes around 0.6, which is exactly the fraction of anomalous edges in this dataset). The decreased 'sample corruption' finally paves the way to better anomaly scoring.

Using SEDANSCORER over the baseline RHSS scoring function always helps, regardless of the sampling algorithm used, as seen from the improved accuracy values in Fig. 3a ($0.17\rightarrow0.57$ with UR-SAMPLER, $0.45\rightarrow0.63$ with SEDANSAMPLER). We attribute this to the holistic edge anomaly scoring by SEDANSCORER based on the *whole* (sampled) graph which is more robust than RHSS which relies only on the local neighborhood of the edge.

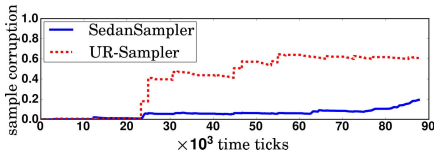
Overall, using SEDANSCORER with SEDANSAMPLER performs the best. Among the top $0.2M$ anomalous edges detected by SEDANSPOT, 82.45% were `smurf(ttl)`, 13.06% were `neptune(ttl)` and 2.36% were `satan` attacks, while only 0.09% were false positives. In contrast, among the top $0.2M$ anomalous edges detected by RHSS, 38.6% were `smurf` and 4.5% were `ipsweep`. Notably, over 53.3% of the flagged edges were false positives, which is unacceptable in critical applications such as network intrusion detection.

Sampler	Scorer	AUC
No sampling	RHSS	0.17
UR-SAMPLER	RHSS	0.17
SEDANSAMPLER	RHSS	0.45
UR-SAMPLER	SEDANSCORER	0.57
SEDANSAMPLER	**SEDANSCORER**	**0.63**

(a)



(b)



(c)

Fig. 3. Anomaly detection in DARPA dataset: (a) contributions of SEDANSAMPLER and SEDANSCORER to AUC gains, (b) steps in red curve show that ground truth anomalous edges occur in bursts, (c) SEDANSAMPLER achieves lower sample 'corruption' due to its rate-adjusted reservoir sampling strategy.

Somewhat surprisingly, RHSS did not detect *any* edge from `neptune` attack – the largest attack in DARPA dataset, consisting of $2.1M$ edges ($=46\%$ of total) and recurring multiple times – as an anomaly. However, this is easily explained: following the first occurrence of `neptune` attack, RHSS increments corresponding edge counts and vertex degrees; thus subsequent occurrences of `neptune` edges, which now have been observed before and connect high degree vertices, are flagged non-anomalous. On the other hand, SEDANSAMPLER gives very low priority to `neptune` edges, which not only decreases their probability of being included in the sample but also ensures that they are easily replaced even if they have been sampled, as stream progresses. Thus, SEDANSPOT successfully detects 13% `neptune` edges among its top $0.2M$.

VIII. CONCLUSION

We considered the problem of anomaly detection given a stream of edges, where anomalies are edges connecting disconnected parts of the graph and possibly occurring in bursts. SEDANSPOT exploited these observations in sublinear memory by (i) performing rate-adjusted sampling which downsamples edges from bursty periods of time and (ii) using a holistic random walk based edge anomaly scoring function to compare an incoming edge with the whole (sampled) graph. Experiments on real-world datasets demonstrated the benefit of our anomaly definition and efficacy of the proposed approach in several scenarios. Future work could explore detecting other anomalies, e.g., slow or periodic attacks, in a streaming setting.

Further links: (a) Code: www.github.com/dhivyaeswaran/sedanspot (b) Supplementary (with proofs, extra experiments): www.cs.cmu.edu/deswaran/papers/icdm18-sedanspot-sup.pdf

Acknowledgments. We thank Travis Dick and Vishwanath Saragadam from Carnegie Mellon University for insightful discussions. This material is based upon work supported by the National Science Foundation under Grants No. CNS-1314632 and IIS-1408924. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding parties. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

REFERENCES

[1] A. Beutel, W. Xu, V. Guruswami, C. Palow, and C. Faloutsos, "Copycatch: stopping group attacks by spotting lockstep behavior in social networks," in *WWW*. ACM, 2013, pp. 119–130.
[2] C. C. Aggarwal, Y. Zhao, and P. S. Yu, "Outlier detection in graph streams," in *ICDE*. IEEE, 2011, pp. 399–409.
[3] S. Ranshous, S. Harenberg, K. Sharma, and N. F. Samatova, "A scalable approach for outlier detection in edge streams using sketch-based approximations," in *SDM*. SIAM, 2016, pp. 189–197.
[4] L. Akoglu, H. Tong, and D. Koutra, "Graph based anomaly detection and description: a survey," *Data Min. Knowl. Discov.*, vol. 29, no. 3, pp. 626–688, 2015.
[5] S. Ranshous, S. Shen, D. Koutra, S. Harenberg, C. Faloutsos, and N. F. Samatova, "Anomaly detection in dynamic networks: a survey," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 7, no. 3, pp. 223–247, 2015.
[6] D. Eswaran, C. Faloutsos, S. Guha, and N. Mishra, "Spotlight: Detecting anomalies in streaming graphs," in *KDD*. ACM, 2018, pp. 1378–1386.
[7] J. Sun, D. Tao, and C. Faloutsos, "Beyond streams and graphs: dynamic tensor analysis," in *KDD*. ACM, 2006, pp. 374–383.
[8] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu, "Graphscope: parameter-free mining of large time-evolving graphs," in *KDD*. ACM, 2007, pp. 687–696.
[9] M. Gupta, J. Gao, Y. Sun, and J. Han, "Integrating community matching and outlier detection for mining evolutionary community outliers," in *KDD*. ACM, 2012, pp. 859–867.
[10] K. Sricharan and K. Das, "Localizing anomalous changes in time-evolving graphs," in *SIGMOD*. ACM, 2014, pp. 1347–1358.
[11] D. Koutra, N. Shah, J. T. Vogelstein, B. Gallagher, and C. Faloutsos, "Deltacon: Principled massive-graph similarity function with attribution," *TKDD*, vol. 10, no. 3, pp. 28:1–28:43, 2016.
[12] W. Yu, C. C. Aggarwal, S. Ma, and H. Wang, "On anomalous hotspot discovery in graph streams," in *ICDM*. IEEE, 2013, pp. 1271–1276.
[13] E. A. Manzoor, S. M. Milajerdi, and L. Akoglu, "Fast memory-efficient anomaly detection in streaming heterogeneous graphs," in *KDD*. ACM, 2016, pp. 1035–1044.
[14] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.
[15] P. S. Efraimidis and P. G. Spirakis, "Weighted random sampling with a reservoir," *Inf. Process. Lett.*, vol. 97, no. 5, pp. 181–185, 2006.
[16] N. K. Ahmed, J. Neville, and R. Kompella, "Network sampling: From static to streaming graphs," *TKDD*, vol. 8, no. 2, p. 7, 2014.
[17] H. Tong, C. Faloutsos, and J. Pan, "Fast random walk with restart and its applications," in *ICDM*. IEEE, 2006, pp. 613–622.
[18] W. Liu and L. Lü, "Link prediction based on local random walk," *EPL (Europhysics Letters)*, vol. 89, no. 5, p. 58007, 2010.
[19] C. Eksombatchai, P. Jindal, J. Z. Liu, Y. Liu, R. Sharma, C. Sugnet, M. Ulrich, and J. Leskovec, "Pixie: A system for recommending 3+ billion items to 200+ million users in real-time," in *WWW*. ACM, 2018, pp. 1775–1784.
[20] M. Yoon, W. Jin, and U. Kang, "Fast and accurate random walk with restart on dynamic graphs with guarantees," in *WWW*. ACM, 2018, pp. 409–418.
[21] W. Yu and J. A. McCann, "Random walk with restart over dynamic graphs," in *ICDM*. IEEE, 2016, pp. 589–598.
[22] M. D. Vose, "A linear algorithm for generating random numbers with a given distribution," *IEEE Transactions on software engineering*, vol. 17, no. 9, pp. 972–975, 1991.
[23] R. Lippmann, R. K. Cunningham, D. J. Fried, I. Graf, K. R. Kendall, S. E. Webster, and M. A. Zissman, "Results of the DARPA 1998 offline intrusion detection evaluation," in *Recent Advances in Intrusion Detection*, 1999.
[24] "Dblp network dataset," http://konect.uni-koblenz.de/networks/dblp_coauthor, 2014.
[25] J. Shetty and J. Adibi, "The enron email dataset database schema and brief statistical report," *Information sciences institute technical report, University of Southern California*, vol. 4, no. 1, pp. 120–128, 2004.
[26] Y. Yang and X. Liu, "A re-examination of text categorization methods," in *SIGIR*. ACM, 1999, pp. 42–49.
[27] A. N. Langville and C. D. Meyer, "Deeper inside pagerank," *Internet Mathematics*, vol. 1, no. 3, pp. 335–380, 2004.