Fast Memory-efficient Anomaly Detection in Streaming Heterogeneous Graphs

Emaad Manzoor*
Stony Brook University
Department of Computer Science
emanzoor@cs.stonybrook.edu

Sadegh M. Milajerdi†
University of Illinois at Chicago
Department of Computer Science
smomen2@uic.edu

Leman Akoglu
Stony Brook University
Department of Computer Science
leman@cs.stonybrook.edu

ABSTRACT

Given a stream of heterogeneous graphs containing different types of nodes and edges, how can we spot anomalous ones in real-time while consuming bounded memory? This problem is motivated by and generalizes from its application in security to host-level advanced persistent threat (APT) detection. We propose StreamSpot, a clustering based anomaly detection approach that addresses challenges in two key fronts: (1) *heterogeneity*, and (2) *streaming nature*. We introduce a new similarity function for heterogeneous graphs that compares two graphs based on their relative frequency of local substructures, represented as short strings. This function lends itself to a vector representation of a graph, which is (a) fast to compute, and (b) amenable to a *sketched* version with bounded size that preserves similarity.

StreamSpot exhibits desirable properties that a streaming application requires—it is (i) fully-streaming; processing the stream one edge at a time as it arrives, (ii) memory-efficient; requiring constant space for the sketches and the clustering, (iii) fast; taking constant time to update the graph sketches and the cluster summaries that can process over $100K$ edges per second, and (iv) online; scoring and flagging anomalies in real time. Experiments on datasets containing simulated system-call flow graphs from normal browser activity and various attack scenarios (ground truth) show that StreamSpot is *high-performance*; achieving above 95% detection accuracy with small delay, as well as competitive time and memory usage.

1. INTRODUCTION

Anomaly detection is a pressing problem for various critical tasks in security, finance, medicine, and so on. In this work, we consider the anomaly detection problem for streaming heterogeneous graphs, which contain different types of nodes and edges. The input is a stream of timestamped and typed edges, where the source and destination nodes are also typed. Moreover, multiple such graphs may be arriving over the stream simultaneously, that is, edges that belong to different graphs may be interleaved. The goal is to accurately and quickly identify the anomalous graphs that are significantly different from what has been observed over the stream thus far, while meeting several important needs of the driving applications including fast real-time detection and bounded memory space usage.

The driving application that motivated our work is the advanced persistent threat (APT) detection problem in security, although the above abstraction can appear in numerous other settings (e.g., software verification). In the APT scenario, we are given a stream of logs capturing the events occurring in the system. These logs are used to construct what is called *information flow graphs*, in which edges depict data or control dependencies. Both the nodes and edges of the flow graphs are typed. Examples of node types are `file`, `process`, etc. and edge types include various system calls such as `read`, `write`, `fork`, etc. as well as other parent-child relations. Within a system, an information flow corresponds to a unit of functionality (e.g., checking email, watching video, software updates, etc.). Moreover, multiple information flows may be occurring in the system simultaneously. The working assumption for APT detection is that the information flows induced by malicious activities in the system are sufficiently different from the normal behavior of the system. Ideally, the detection is to be done in real-time with small computational overhead and delay. As the system-call level events occur rapidly in abundance, it is also crucial to process them in memory while also incurring low space overhead. The problem then can be cast as real-time anomaly detection in streaming heterogeneous graphs with bounded space and time, as stated earlier.

Graph-based anomaly detection has been studied in the past two decades. Most work is for static homogeneous graphs [5]. Those for typed or attributed graphs aim to find deviations from frequent substructures [26, 12, 22], anomalous subgraphs [16, 27], and community outliers [13, 28], all of which are designed for static graphs. For streaming graphs various techniques have been proposed for clustering [3] and connectivity anomalies [4] for plain graphs, which are recently extended to graphs with attributes [35, 24]. (See Sec. 6) Existing approaches are not, at least directly, applicable to our motivating scenario as they do not exhibit all of the desired properties simultaneously; namely, handling heterogeneous graphs, streaming nature, low computational and space overhead, and real-time anomaly detection.

*Primary author.

†This author has contributed with the creation of the datasets used in this work (i.e., collecting and preprocessing system-call traces for the benign/malicious scenarios).

KDD '16, August 13-17, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2939672.2939783

To address the problem for streaming heterogeneous graphs, we introduce a new clustering-based anomaly detection approach called StreamSpot[1] that (*i*) can handle temporal graphs with typed nodes and edges, (*ii*) processes incoming edges fast and consumes bounded memory, as well as (*iii*) dynamically maintains the clustering and detects anomalies in real time. In a nutshell, we propose a new shingling-based similarity function for heterogeneous graphs, which lends itself to graph sketching that uses fixed memory while preserving similarity. We show how to maintain the graph sketches efficiently as new edges arrive. Based on this representation, we employ and dynamically maintain a centroid-based clustering scheme to score and flag anomalous graphs. The main contributions of this work are listed as follows:

- **Novel formulation and graph similarity:** We formulated the host-level APT detection problem as a clustering-based anomaly detection task in streaming heterogeneous graphs. To enable an effective clustering, we designed a new similarity function for timestamped typed graphs, based on *shingling*, which accounts for the frequency of different substructures in a graph. Besides being efficient to compute and effective in capturing similarity between graphs, the proposed function lends itself to comparing two graphs based on their *sketches*, which enables memory-efficiency.
- **Dynamic maintenance:** We introduce efficient techniques to keep the various components of our approach up to date as new edges arrive over the stream. Specifically, we show how to maintain (*a*) the graph sketches, and (*b*) the clustering incrementally.
- **Desirable properties:** Our formulation and proposed techniques are motivated by the requirements and desired properties of the application domain. As such, our approach is (*i*) *fully streaming*, where we perform a continuous, edge-level processing of the stream, rather than taking a snapshot-oriented approach; (*ii*) *time-efficient*, where the processing of each edge is fast with constant complexity to update its graph's sketch and the clustering; (*iii*) *memory-efficient*, where the sketches and cluster summaries consume constant memory that is controlled by user input, and (*iv*) *online*, where we score and flag the anomalies in real-time.

We quantitatively validate the effectiveness and efficiency (time and space) of StreamSpot on simulated datasets containing normal host-level activity as well as abnormal attack scenarios (i.e., ground truth). We also design experiments to study the approximation quality of our sketches, and the behavior of our detection techniques under varying parameters, such as memory size.

2. PROBLEM & OVERVIEW

For host-level APT detection, a host machine is instrumented to collect system logs. These logs essentially capture the events occurring in the system, such as memory accesses, system calls, etc. An example log sequence is illustrated in Figure 1. Based on the control and data dependences, information flow graphs are constructed from the system logs. In the figure, the `tag` column depicts the ID of the information flow (graph) that an event (edge) belongs to.

[1]Code/data at `http://www3.cs.stonybrook.edu/~emanzoor/streamspot/`

time	pid	event	arg/data	tag
100	10639	fork	NULL	1
200	10640	execve	/bin/sh	1
300	10650	read	STDIN	2
400	10640	fstat	0xbfc5598	1
500	10660	sock_wr	0.0.0.0	2
…	…	…	…	…

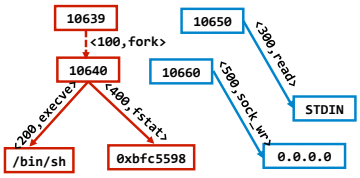


Figure 1: Example stream of system logs, and two resulting information flow graphs (red vs. blue). Both nodes and edges are typed. Edges arriving to different flows may interleave.

The streaming graphs are heterogeneous where edge types correspond to system calls such as `read`, `fork`, `sock_wr`, etc. and node types include `socket`, `file`, `memory`, etc. As such, an edge can be represented in the form of:

`< source-id, source-type` ϕ_s`, dest-id, dest-type` ϕ_d`, timestamp` t`, edge-type` ϕ_e`, flow-tag >`

These edges form dynamically evolving graphs, where the edges sharing the same flow-tag belong to the same graph. Edges arriving to different graphs may be interleaved, that is, multiple graphs may be evolving simultaneously.

Our goal is to detect anomalous graphs at any given time t, i.e., in real time as they occur. To achieve this goal, we follow a clustering-based anomaly detection approach. In a nutshell, our method maintains a small, memory-efficient representation of the evolving graphs in main memory, and uses a new similarity measure that we introduce to cluster the graphs. The clustering is also maintained dynamically as existing graphs evolve or as new ones arrive. Anomalies are flagged in real time through deviations from this clustering model that captures the normal flow patterns.

The main components of StreamSpot are as follows, which are detailed further in the noted (sub)sections:

- **Similarity of heterogeneous graphs:** (§3.1) We introduce a new similarity measure for heterogeneous graphs with typed nodes and edges as well as timestamped edges. Each graph G is represented by a what we call shingle-frequency vector (or shortly *shingle vector*) $\mathbf{z}_G$. Roughly, a k-shingle $s(v,k)$ is a string constructed by traversing edges, in their temporal order, in the k-hop neighborhood of node v. The shingle-vector contains the counts of unique shingles in a graph. Similarity between two graphs is defined as the cosine similarity between their respective shingle vectors. Intuitively, the more the same shingles two graphs contain in common, the more similar they are.
- **Memory-efficient sketches:** (§3.2) The number of unique shingles can be arbitrarily large for heterogeneous graphs with hundreds to thousands of node and edge types. We show how to instead use a *sketch* representation of a graph. Sketches are small, constant-size vector representations of graphs whose pairwise similarities also approximate the (cosine) similarities of the shingle vectors of those graphs.
- **Efficient maintenance of sketches:** (§3.3) As new edges arrive, shingle counts of a graph change. As such, the shingle vector entries need to be updated. Recall that we do not explicitly maintain this vector in memory, but rather its (much smaller) sketch. In this paper, we show how to update the sketch of a graph efficiently, (*i*) in *constant* time and (*ii*) without incurring *any* additional memory overhead.

- **Clustering graphs (dynamically):** (§4) We employ a centroid-based clustering of the graphs to capture normal behavior. We show how to update the clustering as the graphs change and/or as new ones emerge, that again, exhibit small memory footprints.
- **Anomaly detection (in real time):** We score an incoming or updated graph by its distance to the closest centroid in the clustering. Based on a distribution of distances for the corresponding cluster, we quantify the significance of the score to flag anomalies. Distance computation is based on (fixed size) sketches, as such, scoring is fast for real time detection.

3. SKETCHING TYPED GRAPHS

We require a method to represent and compute the similarity between heterogeneous graphs that also captures the temporal order of edges. The graph representation must permit efficient online updates and consume bounded space with new edges arriving in an infinite stream.

Though there has been much work on computing graph similarity efficiently, existing methods fall short of our requirements. Methods that require knowing node correspondence [20] are inapplicable, as are graph kernels that precompute a fixed space of substructures [31] to represent graphs, which is infeasible in a streaming scenario. Methods that rely on *global* graph metrics [7] cannot accommodate edge order and are also inapplicable. Graph-edit-distance-based [9] methods approximate hard computational steps with heuristics that provide no error guarantees, and are hence unsuitable.

We next present a similarity function for heterogenous graphs that captures the local structure and respects temporal order of edges. The underlying graph representation permits efficient updates as new edges arrive in the stream and consumes bounded space, without needing to compute or store the full space of possible graph substructures.

3.1 Graph Similarity by Shingling

Analogous to *shingling* text documents into k-grams [8] to construct their vector representations, we decompose each graph into a set of *k-shingles* and construct a vector of their frequencies. The similarity between two graphs is then defined as the cosine similarity between their k-shingle frequency vectors. We formalize these notions below.

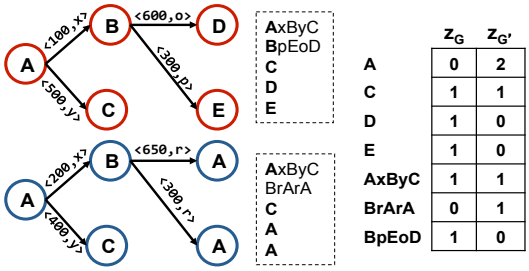
DEFINITION 1 *(k-shingle). Given a graph $G = (V, E)$ and node $v \in V$, the k-shingle $s(v, k)$ is a string constructed via a k-hop breadth-first traversal starting from v as follows:*

1. *Initialize k-shingle as type of node v: $s(v, k) = \phi_v$.*
2. *Traverse the outgoing edges from each node in the order of their timestamps, t.*
3. *For each traversed edge e having destination w, concatenate the types of the edge and the destination node with the k-shingle: $s(v, k) = s(v, k) \oplus \phi_e \oplus \phi_w$.*

We abbreviate the **O**rdered **k**-hop **B**readth **F**irst **T**raversal performed during the k-shingle construction defined above as OKBFT. It is important to note that the k-hop neighborhood constructed by an OKBFT is *directed*.

DEFINITION 2 *(shingle (frequency) vector $\mathbf{z}_G$). Given the k-shingle universe S and a graph $G = (V, E)$, let $S_G = \{s(v, k), \forall v \in V\}$ be the set of k-shingles of G. $\mathbf{z}_G$ is a vector of size $|S|$ wherein each element $\mathbf{z}_G(i)$ is the frequency of shingle $s_i \in S$ in S_G.*

Shingling is illustrated for two example graphs in Figure 2, along with their corresponding shingle vectors.



	$\mathbf{z}_G$	$\mathbf{z}_{G'}$
A	0	2
C	1	1
D	1	0
E	1	0
AxByC	1	1
BrArA	0	1
BpEoD	1	0

Figure 2: Shingling: (left) two example graphs with their shingles listed in dashed boxes (for $k = 1$), (right) corresponding shingle frequency vectors.

The similarity between two graphs G and G' is then the cosine similarity between their shingle vectors $\mathbf{z}_G$ and $\mathbf{z}_{G'}$.

Representing graphs by shingle vectors captures both their local graph structure and edge order. It also permits efficient online updates, since the local nature of k-shingles ensures that only a few shingle vector entries need to be updated for each incoming edge (§3.3). The parameter k controls the trade-off between expressiveness and computational efficiency. A larger k produces more expressive local neighborhoods, whereas a smaller one requires fewer entries to be updated in the shingle vector per incoming edge.

3.2 Graph Sketches by Hashing

With a potentially large number of node and edge types, the universe S of k-shingles may explode combinatorially and render it infeasible to store $|S|$-dimensional shingle vectors for each graph. We now present an alternate *constant*-space graph representation that approximates the shingle count vector via *locality-sensitive hashing* (LSH) [17].

An LSH scheme for a given similarity function enables efficient similarity computation by projecting high-dimensional vectors to a low-dimensional space while preserving their similarity. Examples of such schemes are MINHASH [8] for the Jaccard similarity between sets and SIMHASH [10] for the cosine similarity between real-valued vectors, which we detail further in this section.

3.2.1 SIMHASH

Given input vectors in $\mathbb{R}^d$, SIMHASH is first instantiated with L projection vectors $\mathbf{r}_1, \ldots, \mathbf{r}_L \in \mathbb{R}^d$ chosen uniformly at random from the d-dimensional Gaussian distribution. The LSH $h_{\mathbf{r}_l}(\mathbf{z})$ of an input vector $\mathbf{z}$ for a given random projection vector $\mathbf{r}_l$, $l = 1 \ldots, L$, is defined as follows:

$$h_{\mathbf{r}_l}(\mathbf{z}) = \begin{cases} +1, & \text{if } \mathbf{z} \cdot \mathbf{r}_l \geq 0 \\ -1, & \text{if } \mathbf{z} \cdot \mathbf{r}_l < 0 \end{cases} \quad (1)$$

In other words $h_{\mathbf{r}_l}(\mathbf{z}) = sign(\mathbf{z} \cdot \mathbf{r}_l)$, which obeys the property that the probability (over vectors $\mathbf{r}_1, \ldots, \mathbf{r}_L$) that any two input vectors $\mathbf{z}_G$ and $\mathbf{z}_{G'}$ hash to the same value is proportional to their cosine similarity:

$$P_{l=1\ldots L}[h_{\mathbf{r}_l}(\mathbf{z}_G) = h_{\mathbf{r}_l}(\mathbf{z}_{G'})] = 1 - \frac{cos^{-1}\left(\frac{\mathbf{z}_G \cdot \mathbf{z}_{G'}}{\|\mathbf{z}_G\| \|\mathbf{z}_{G'}\|}\right)}{\pi} \quad (2)$$

Since computing similarity requires only these hash values, each d-dimensional input vector $\mathbf{z}$ can be replaced with an L-dimensional *sketch vector* $\mathbf{x}$ containing its LSH values, i.e.,

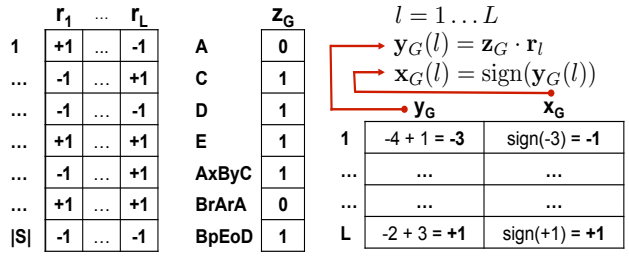


Figure 3: Sketching: (left) L random vectors, (center) shingle vector $\mathbf{z}_G$ of graph G, (right) corresponding projection vector $\mathbf{y}_G$ and sketch vector $\mathbf{x}_G$.

$\mathbf{x} = [h_{\mathbf{r}_1}(\mathbf{z}), \ldots, h_{\mathbf{r}_L}(\mathbf{z})]$. As such, each sketch vector can be represented with just L bits, where each bit corresponds to a value in $\{+1, -1\}$.

The similarity between two input vectors then can be estimated by empirically evaluating the probability in Eq. (2) as the proportion of hash values that the input vectors agree on when hashed with L random vectors. That is,

$$sim(G, G') \propto \frac{|\{l : \mathbf{x}_G(l) = \mathbf{x}_{G'}(l)\}|}{L} \tag{3}$$

In summary, given a target dimensionality $L \ll |S|$, we can represent each graph G with a sketch vector of dimension L, discard the $|S|$-dimensional shingle vectors and compute similarity in this new vector space.

3.2.2 *Simplified* SimHash

Note that we need to perform the *same* set of random projections on the changing or newly emerging shingle vectors, with the arrival of new edges and/or new graphs. As such, we would need to maintain the set of L projection vectors in main memory (for now, later we will show that we do not need them explicitly).

In practice, the random projection vectors $\mathbf{r}_l$'s remain sufficiently random when each of their $|S|$ elements are drawn uniformly from $\{+1, -1\}$ instead of from a $|S|$-dimensional Gaussian distribution [29]. With this simplification, just like the sketches, each projection vector can also be represented using $|S|$ *bits*, rather than $32 \times |S|$ bits (assuming 4 bytes per float), further saving space.

Figure 3 illustrates the idea behind sketching (for now in the static case). Given $|S|$-dimensional random $\mathbf{r}_l$ vectors, $l = 1 \ldots L$, with elements in $\{+1, -1\}$ (left) and a shingle vector $\mathbf{z}_G$ (center), the L-dimensional sketch $\mathbf{x}_G$ is obtained by taking the sign of the dot product of $\mathbf{z}$ with each $\mathbf{r}_l$ (right).

However, three main shortcomings of SimHash remain: (1) it requires explicit projection vectors in memory, (2) $|S|$ can still get prohibitively large, and (3) it requires knowing the size of the complete shingle universe $|S|$ to specify the dimension of each random vector. With new shingles continuously being formed from the new node and edge types arriving in the stream, the complete shingle universe (and hence its size) always remains *unknown*. As such, SimHash as proposed cannot be applied in a streaming setting.

3.2.3 StreamHash

To resolve the issues with SimHash for the streaming setting, we propose StreamHash which, rather than L $|S|$-dimensional random bit vectors (with entries corresponding to $\{+1, -1\}$ as described above), is instead instantiated with L hash functions $h_1, \ldots, h_L$ picked uniformly at random from a family $\mathcal{H}$ of hash functions, mapping shingles to $\{+1, -1\}$. That is, a $h \in \mathcal{H}$ is a *deterministic* function that maps a fixed/given shingle s to either $+1$ or -1.

Properties of $\mathcal{H}$. We require a family that exhibits three key properties; uniformity with respect to shingles and hash functions, and pairwise-independence, as described below.

First, it should be equally probable for a given shingle to hash to $+1$ or -1 over all hash functions in the family:

$$\Pr_{h \in \mathcal{H}}[h(s) = +1] = \Pr_{h \in \mathcal{H}}[h(s) = -1], \ \forall s \in S. \tag{4}$$

Second, to disallow trivially uniform families such as $\mathcal{H} = \{\forall s \in S : h_1(s) = +1, \ h_2(s) = -1\}$, we also require that for a given hash function, hash values in $\{+1, -1\}$ are equiprobable over all shingles in the universe:

$$\Pr_{s \in S}[h(s) = +1] = \Pr_{s \in S}[h(s) = -1], \ \forall h \in \mathcal{H}. \tag{5}$$

To further disallow uniform families with correlated uniform hash functions such as $\mathcal{H} = \{h_1(s), \ h_2(s) = -h_1(s)\}$ (where h_1 is some uniform hash function), we require the hash functions in the family to be pairwise-independent:

$$\forall s, s' \in S \text{ s.t. } s \neq s' \text{ and } \forall t, t' \in \{+1, -1\},$$
$$\Pr_{h \in \mathcal{H}}[h(s') = t' | h(s) = t] = \Pr_{h \in \mathcal{H}}[h(s') = t']. \tag{6}$$

If the shingle universe is fixed and known, picking a hash function h_l at random from $\mathcal{H}$ is equivalent to picking some vector $\mathbf{r}_l$ at random from $\{+1, -1\}^{|S|}$, with each element $\mathbf{r}_l(i)$ equal to the hash value $h_l(s_i)$.

If we overload the "dot-product" operator for an input vector $\mathbf{z}$ and a hash function h_l as follows:

$$\mathbf{y}(l) = \mathbf{z} \cdot h_l = \sum_{i=1,\ldots,|S|} \mathbf{z}(i) h_l(s_i), \ \ l = 1 \ldots L \tag{7}$$

we can define the LSH $g_{h_l}(\mathbf{z})$ of the input vector $\mathbf{z}$ for the given hash function similar to Eq. (1):

$$g_{h_l}(\mathbf{z}) = \begin{cases} +1, & \text{if } \mathbf{z} \cdot h_l \geq 0 \\ -1, & \text{if } \mathbf{z} \cdot h_l < 0 \end{cases} \tag{8}$$

The $\mathbf{y}$ vector is called the *projection vector* of a graph. Each entry $\mathbf{y}(l)$ as given in Eq. (7) essentially holds the sum of the counts of shingles that map to $+1$ by h_l minus the sum of the counts of shingles that map to -1 by h_l.

The L-bit sketch can then be constructed for each input vector $\mathbf{z}$ by $\mathbf{x} = sign(\mathbf{y})$ and used to compute similarity the same way as in SimHash. Unlike SimHash, the sketches in StreamHash can be constructed and maintained *incrementally* (§3.3), as a result of which, we no longer need to know the complete shingle universe S or maintain $|S|$-dimensional random vectors in memory.

Choosing $\mathcal{H}$. A family satisfying the aforementioned three properties is said to be *strongly universal* [34]. We adopt the strongly universal multilinear family for strings [21]. In this family, the input string s (i.e., shingle) is divided into $|s|$ components (i.e., "characters") as $s = c_1 c_2 \ldots c_{|s|}$. A hash function h_l is constructed by first choosing $|s|$ random numbers $m_1^{(l)}, \ldots, m_{|s|}^{(l)}$, and s is then hashed as follows:

$$h_l(s) = 2 \times \left((m_1^{(l)} + \sum_{i=2}^{|s|} m_i^{(l)} \times int(c_i)) \bmod 2 \right) - 1. \tag{9}$$

where $int(c_i)$ is the ASCII value of c_i and $h_l(s) \in \{+1, -1\}$.

Note that the hash value for a shingle s of length $|s|$ can be computed in $\Theta(|s|)$ time.

We represent each hash function by $|s|_{\max}$ random numbers, where $|s|_{\max}$ denotes the maximum possible length of a shingle. These numbers are fixed per hash function h_l, as it is a deterministic function that hashes a fixed/given shingle to the same value each time. In practice, L hash functions can be chosen uniformly at random from this family by generating $L \times |s|_{\max}$ uniformly random 64-bit integers using a pseudorandom number generator.

Merging sketches. Two graphs G and G' will merge if an edge arrives in the stream having its source node in G and destination node in G', resulting in a graph that is their union $G \cup G'$. The centroid of a cluster of graphs is also represented by a function of their union (§4). Both scenarios require constructing the *sketch of the union of graphs*, which we detail in this subsection.

The shingle vector of the union of two graphs G and G' is the sum of their individual shingle vectors:

$$\mathbf{z}_{G\cup G'} = \mathbf{z}_G + \mathbf{z}_{G'}. \tag{10}$$

As we show below, the projection vector of the union of two graphs $\mathbf{y}_{G\cup G'}$ also turns out to be the sum of their individual projection vectors $\mathbf{y}_G$ and $\mathbf{y}_{G'}$; $\forall l = 1, \ldots, L$:

$$\begin{aligned}
\mathbf{y}_{G\cup G'}(l) &= \mathbf{z}_{G\cup G'} \cdot h_l && \text{(by Eq. (7))}\\
&= \sum_{i=1,\ldots,|S|} \left(\mathbf{z}_G(i) + \mathbf{z}_{G'}(i)\right) h_l(s_i) && \text{(by Eq. (10))}\\
&= \sum_{i=1,\ldots,|S|} \mathbf{z}_G(i) h_l(s_i) + \sum_{i=1,\ldots,|S|} \mathbf{z}_{G'}(i) h_l(s_i)\\
&= \mathbf{y}_G(l) + \mathbf{y}_{G'}(l). && (11)
\end{aligned}$$

Hence, the L-bit sketch of $G \cup G'$ can be computed as $\mathbf{x}_{G\cup G'} = sign(\mathbf{y}_G + \mathbf{y}_{G'})$. This can trivially be extended to construct the sketch of the union of any number of graphs.

3.3 Maintaining Sketches Incrementally

We now describe how StreamHash sketches are updated on the arrival of a new edge in the stream. Each new edge being appended to a graph gives rise to a number of *outgoing* shingles, which are removed from the graph, and *incoming* shingles, which are added to the graph. These shingles are constructed by OkBFT traversals from certain nodes of the graph, which we detail further in this section.

Let $e(u, v)$ be a new edge arriving in the stream from node u to node v in some graph G. Let $\mathbf{x}_G$ be the L-bit sketch vector of G. We also associate with each graph a length-L *projection vector* $\mathbf{y}_G$, which contains "dot-products" (Eq. (7)) for the hash functions $h_1, \ldots, h_L$. For an empty graph, $\mathbf{y}_G = \mathbf{0}$ and $\mathbf{x}_G = \mathbf{1}$ since $\mathbf{z}(i)$'s are all zero.

For a given incoming shingle s_i, the corresponding $\mathbf{z}(i)$ implicitly[2] increases by 1. This requires each element of the projection vector $\mathbf{y}_G(l)$ to be updated by simply adding the corresponding hash value $h_l(s_i) \in \{+1, -1\}$ due to the nature of the dot-product in Eq. (7). Updating $\mathbf{y}_G$ for an outgoing shingle proceeds similarly but by subtracting the hash values. For each element $\mathbf{y}_G(l)$ of the projection vector that is updated *and* that changes sign, the corresponding bit of the sketch $\mathbf{x}_G(l)$ is updated using the new sign (Eq. (8)). Updating the sketch for an incoming or outgoing shingle s is formalized by the following update equations. $\forall l = 1, \ldots, L$:

$$\mathbf{y}_G(l) = \begin{cases} \mathbf{y}_G(l) + h_l(s), & \text{if } s \text{ an incoming shingle} \\ \mathbf{y}_G(l) - h_l(s), & \text{if } s \text{ an outgoing shingle} \end{cases} \tag{12}$$

$$\mathbf{x}_G(l) = \text{sign}(\mathbf{y}_G(l)). \tag{13}$$

Now that we can update the sketch (using the updated projection vector) of a graph for both incoming and outgoing shingles, without maintaining any shingle vector explicitly, we need to describe the construction of the incoming and outgoing shingles for a new edge e.

Appending e to the graph updates the shingle for every node that can reach e's destination node v in at most k hops, due to the nature of k-shingle construction by OkBFT. For each node w to be updated, the incoming shingle is constructed by an OkBFT from w that considers e during traversal, and the outgoing shingle is constructed by an OkBFT from w that ignores e during traversal. In practice, both shingles can be constructed by a single modified-OkBFT from w parameterized with the new edge.

Since the incoming shingle for a node may be the outgoing shingle for another, combining and further collapsing the incoming and outgoing shingles from all the updated nodes will enable updating the sketch while minimizing the number of redundant updates.

3.4 Time and Space Complexity

Time. Since sketches are constructed incrementally (§3.3), we evaluate the running time for each new edge arriving in the stream. This depends on the largest directed k-hop neighborhood possible for the nodes in our graphs. Since the maximum length of a shingle $|s|_{\max}$ is proportional to the size of this neighborhood, we specify the time complexity in terms of $|s|_{\max}$.

A new edge triggers an update to $O(|s|_{\max})$ nodes, each of which results in an OkBFT that takes $O(|s|_{\max})$ time. Thus, it takes $O(|s|^2_{\max})$ time to `construct` the $O(|s|_{\max})$ incoming and outgoing shingles for a new edge. Hashing each shingle takes $O(|s|_{\max})$ time (§3.2.3) resulting in a total `hashing` time of $O(|s|^2_{\max})$. Updating the projection vector elements and bits in the sketch takes $O(L)$ time.

This leads to an overall sketch update time of $O(L+|s|^2_{\max})$ per edge. Since L is a constant parameter and $|s|_{\max}$ depends on the value of the parameter k, the per-edge running time can be controlled.

Space. Each graph (of size at most $|G|_{\max}$) with its sketch and projection vectors consumes $O(L+|G|_{\max})$ space.[3] However, the number of graphs in the stream is unbounded, as such the overall space complexity is dominated by storing graphs. Hence, we define a parameter N to limit the maximum number of edges we retain in memory at any instant. Once the total number of edges in memory exceeds N, we evict the oldest edge incident on the least recently *touched* node. The rationale is that nodes exhibit locality of reference by the edges in the stream that touch them (i.e., that have them as a source or destination). With up to N edges, we also assume a *constant* number c of graphs is maintained and processed in memory at any given time.

The total space complexity is then $O(cL + N)$ which can also be controlled. Specifically, we choose N proportional to the available memory size, and L according to the required quality of approximation of graph similarity.

[2] As we do not maintain the shingle vector $\mathbf{z}$'s explicitly.

[3] Note that the projection vector holds L positive and/or negative *integers*, and the sketch is a length-L *bit* vector.

4. ANOMALY DETECTION

Bootstrap Clusters. StreamSpot is first initialized with bootstrap clusters obtained from a training dataset of benign flow-graphs. The training graphs are grouped into K clusters using the K-medoids algorithm, with K chosen to maximize the silhouette coefficient [30] of the resulting clustering. This gives rise to compact clusters that are well-separated from each other. An *anomaly threshold* for each cluster is set to 3 standard deviations greater than the mean distance between the cluster's graphs and medoid. This threshold is derived from Cantelli's inequality [14] with an upper-bound of 10% on the false positive rate.

Provided the bootstrap clusters, StreamSpot constructs StreamHash projection vectors for each training graph, and constructs the projection vector of the centroid of each cluster as the average of the projection vectors of the graphs it contains. In essence, the centroid of a cluster is the "average graph" with shingle vector counts formed by the union (§3.2.3) of the graphs it contains divided by the number of graphs. The sketch of each centroid is then constructed and the bootstrap graphs are discarded from memory.

Streaming Cluster Maintenance. Apart from the cluster centroid sketches and projection vectors, we maintain in memory the number of graphs in each cluster and, for each observed and unevicted graph, its anomaly score and assignment to either one of K clusters or an "attack" class. Each new edge arriving at a graph G updates its sketch $\mathbf{x}_G$ and projection vector $\mathbf{y}_G$ to $\mathbf{x}'_G$ and $\mathbf{y}'_G$ respectively (§3.3). $\mathbf{x}'_G$ is then used to compute the distance of G to each cluster centroid. Let Q be the nearest cluster to G, of size $|Q|$ and with centroid sketch $\mathbf{x}_Q$ and centroid projection vector $\mathbf{y}_Q$.

If G was previously unassigned to any cluster and the distance of G to Q is lesser than its corresponding cluster threshold, then G is assigned to Q and its size and projection vector are updated $\forall l = 1, \ldots, L$ as:

$$\mathbf{y}_Q(l) = \frac{\mathbf{y}_Q(l) \times |Q| + \mathbf{y}'_G(l)}{|Q| + 1}, \quad |Q| = |Q| + 1 \ . \tag{14}$$

If the graph was previously already assigned to Q, its size remains the same and its projection vector is updated as:

$$\mathbf{y}_Q(l) = \mathbf{y}_Q(l) + \frac{\mathbf{y}'_G(l) - \mathbf{y}_G(l)}{|Q|} \ . \tag{15}$$

If the graph was previously assigned to a different cluster $R \neq Q$, Q is updated using Eq. 14 and the size and projection vector of R are updated as:

$$\mathbf{y}_R(l) = \frac{\mathbf{y}_R(l) \times |R| - \mathbf{y}_G(l)}{|R| - 1}, \quad |R| = |R| - 1 \ . \tag{16}$$

If the distance from G to Q is greater than its corresponding cluster threshold, G is removed from its assigned cluster (if any) using Eq. (16) and assigned to the "attack" class. In all cases where the projection vector of Q (or R) is updated, the corresponding sketch is also updated as:

$$\mathbf{x}_Q(l) = sign(\mathbf{y}_Q(l)), \quad \forall l = 1, \ldots, L. \tag{17}$$

Finally, the anomaly score of G is computed as its distance to Q after Q's centroid has been updated.

Time and Space Complexity. With K clusters and L-bit sketches, finding the nearest cluster takes $O(KL)$ time and computing the graph's anomaly score takes $O(L)$ time. Adding a graph to (Eq. (14)), removing a graph from (Eq. (16)) and updating (Eq. (15)) a cluster each take $O(L)$ time, leading to a total time complexity of $O(KL)$ per-edge.

With a maximum of c graphs retained in memory by limiting the maximum number of edges to N (§3.4), storing cluster assignments and anomaly scores each consume $O(c)$ space. The centroid sketches and projection vectors each consume $O(KL)$ space, leading to a total space complexity of $O(c + KL)$ for clustering and anomaly detection.

5. EVALUATION

Datasets. Our datasets consist of flow-graphs derived from 1 attack and 5 benign scenarios. The benign scenarios involve normal browsing activity, specifically watching YouTube, downloading files, browsing `cnn.com`, checking Gmail, and playing a video game. The attack involves a drive-by download triggered by visiting a malicious URL that exploits a Flash vulnerability and gains root access to the visiting host. For each scenario, Selenium RC[4] was used to automate the execution of a 100 tasks on a Linux machine. All system calls on the machine from the start of a task until its termination were traced and used to construct the flow-graph for that task. The flow-graphs were compiled into 3 datasets whose properties are shown in Table 1.

Experiment Settings. We evaluate StreamSpot in the following settings:

(1) *Static*: We use p% of all the benign graphs for training, and the rest of the benign graphs along with the attack graphs for testing. We find an offline clustering of the training graphs and then score and rank the test graphs based on this clustering. Graphs are represented by their shingle vectors and all required data is stored in memory. The goal is to quantify the effectiveness of StreamSpot before introducing approximations to optimize for time and space.

(2) *Streaming*: We use p% of the benign graphs for training to first construct a bootstrap clustering offline. This is provided to initialize StreamSpot, and the test graphs are then streamed in and processed online one edge at a time. Hence, test graphs may be seen only partially at any given time. For each edge, StreamSpot updates the corresponding graph sketch, clusters, cluster assignments and anomaly scores, and a snapshot of the anomaly scores is retained every 10,000 edges for evaluation. StreamSpot is also evaluated under memory constraints by limiting the sketch size and maximum number of stored edges.

5.1 Static Evaluation

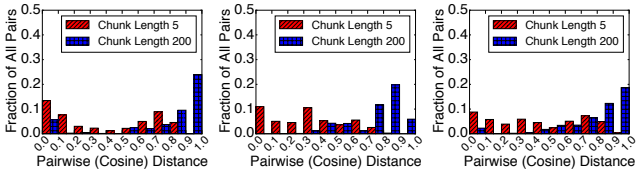
We first cluster the training graphs based on their shingle-vector similarity. Due to the low diameter and large out-degree exhibited by flow-graphs, the shingles obtained tend to be long (even for $k = 1$), and similar pairs of shingles from two graphs differ only by a few characters; this results in most pairs of graphs appearing dissimilar.

To mitigate this, we 'chunk' each shingle by splitting it into fixed-size units. The chunk length parameter C controls the influence of graph structure and node type frequency on the pairwise similarity of graphs. A small C reduces the effect of structure and relies on the frequency of node types, making most pairs of graphs similar. A large C tends to make pairs of graphs more dissimilar. This variation is evident in Figure 4, showing the pairwise-distance distributions for different chunk lengths.

[4] http://www.seleniumhq.org/projects/remote-control/

Table 1: Dataset summary: Training scenarios and test edges (attack + 25% benign graphs).

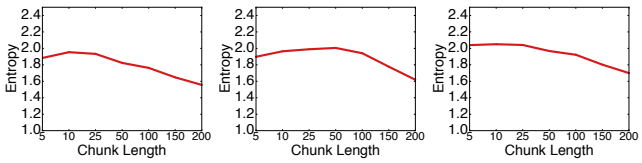
Dataset	Scenarios	# Graphs	Avg. |V|	Avg. |E|	# Test Edges
YDC	YouTube, Download, CNN	300	8705	239648	21,857,899
GFC	GMail, VGame, CNN	300	8151	148414	13,854,229
ALL	YouTube, Download, CNN, GMail, VGame	500	8315	173857	24,826,556



(a) YDC (b) GFC (c) ALL

Figure 4: Distribution of pairwise cosine distances for different values of chunk lengths.

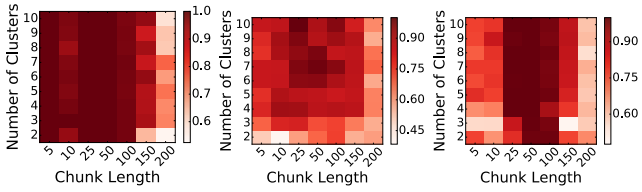
We aim to choose a C that neither makes all pairs of graphs too similar or dissimilar. Figure 5 shows the entropy of pairwise-distances with varying C for each dataset. At the point of maximum entropy, the distances are near-uniformly distributed. A safe region to choose C is near and to the right of this point; intuitively, this C sufficiently differentiates dissimilar pairs of graphs, while not affecting similar ones. For our experiments, we pick $C = 25, 100, 50$ respectively for YDC, GFC, and ALL. After fixing C, we cluster the training graphs with K-medoids and pick K with the maximum silhouette coefficient [30] for the resulting clustering; these are $K = 5, 5, 10$ respectively for YDC, GFC, and ALL.



(a) YDC (b) GFC (c) ALL

Figure 5: Variation in entropy of the pairwise cosine distance distribution. ($p = 75\%$)

To validate our intuition for picking C, Figure 6 shows a heatmap of the average precision obtained after clustering and anomaly-ranking on the test data (the attack and remaining 25% benign graphs), for varying C and K. We can see that for our chosen C and K, StreamSpot achieves near-ideal performance. We also find that the average precision appears robust to the number of clusters when chunk length is chosen in the safe region .



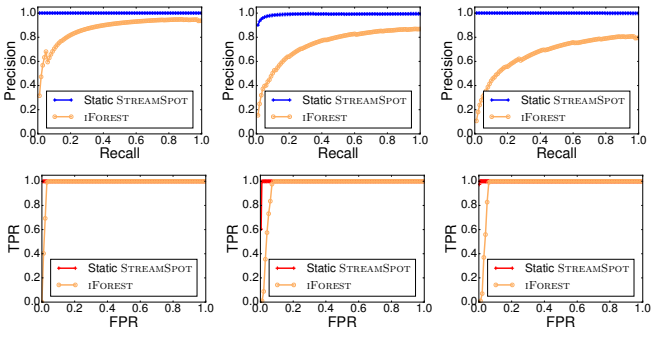
(a) YDC (b) GFC (c) ALL

Figure 6: Average precision for different chunk-length C and number of clusters K. ($p = 75\%$)

To quantify anomaly detection performance in the static setting, we set $p = 25\%$ of the data as training and cluster the training graphs based on their shingle vectors, following the aforementioned steps to choose C and K. We then score each test graph by its distance to the closest centroid in the clustering. This gives us a ranking of the test graphs, based on which we plot the precision-recall (PR) and ROC curves.

As a baseline, we use iForest [23] with 100 trees and 75% subsampling rate with each graph represented by a vector of 10 structural features: the average/maximum degree and distinct-degree[5], the average eccentricity and shortest-path length, and the diameter, density and number of nodes/edges.

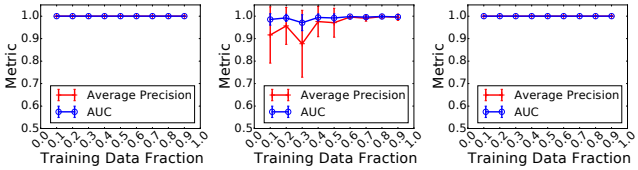
The curves (averaged over 10 independent random samples) for all the datasets are shown in Figure 7. Note that even with 25% of the data, static StreamSpot is effective in correctly ranking the attack graphs and achieves an average precision (AP, area under the PR curve) of more then 0.9 and a near-ideal AUC (area under ROC curve).



(a) YDC (b) GFC (c) ALL

Figure 7: (top) Precision-Recall (PR) and (bottom) ROC curves averaged over 10 samples. ($p = 25\%$)

Finally, in Figure 8 we show how the AP and AUC change as the training data percentage p is varied from $p = 10\%$ to $p = 90\%$. We note that with sufficient training data, the test performance reaches an acceptable level for GFC.



(a) YDC (b) GFC (c) ALL

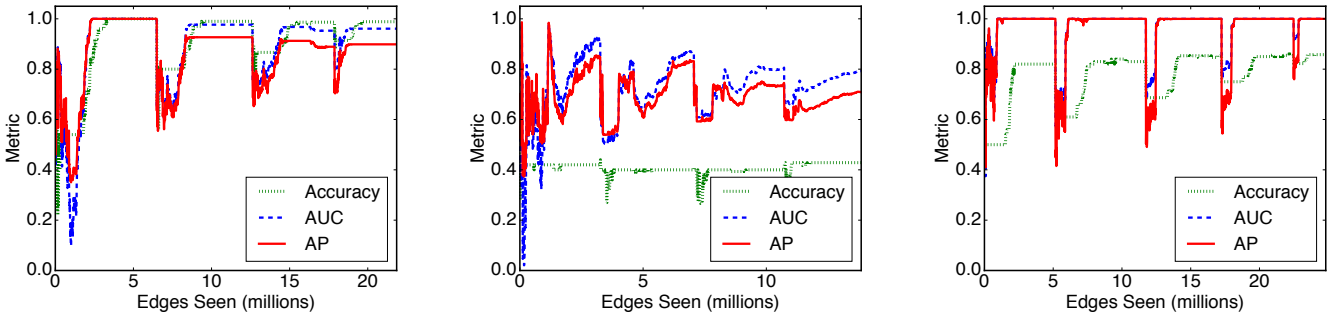
Figure 8: AP and AUC with varying training % p.

The results in this section demonstrate a proof of concept that our proposed method can effectively spot anomalies provided offline data and unbounded memory. We now move on to testing StreamSpot in the streaming setting, for which it was designed.

5.2 Streaming Evaluation

We now show that StreamSpot remains both accurate in detecting anomalies and efficient in processing time and memory usage in a streaming setting. We control the number of graphs that arrive and grow *simultaneously* with parameter B, by creating groups of B graphs at random from the test

[5]Ignoring multi-edges to the same destination node.



(a) YDC (b) GFC (c) ALL

Figure 9: Performance of StreamSpot at different instants of the stream for all datasets ($L = 1000$).

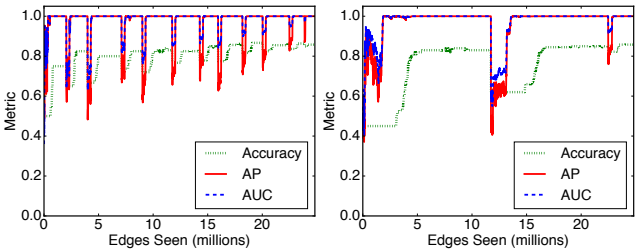


Figure 10: StreamSpot performance on ALL (measured at every 10K edges) when (left) $B = 20$ graphs and (right) $B = 100$ graphs arrive simultaneously.

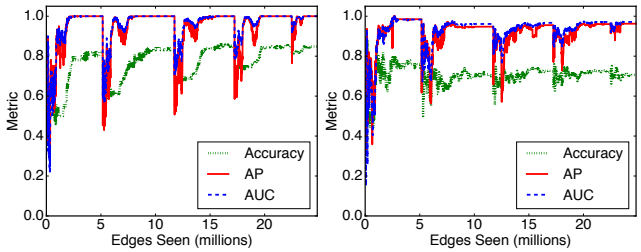
graphs, picking one group at a time and interleaving the edges from graphs within the group to form the stream.

In all experiments, 75% of the benign graphs where used for bootstrap clustering. Performance metrics are computed on the instantaneous anomaly ranking every 10,000 edges.

Detection performance. Fig. 10 shows the anomaly detection performance on the ALL dataset, when $B = 20$ graphs grow in memory simultaneously (left) as compared to $B = 100$ graphs (right). We make the following observations: (*i*) The detection performance follows a trend, with periodic dips followed by recovery. (*ii*) Each dip corresponds to the arrival of a new group of graphs. Initially, only a small portion of the new graphs are available and the detection performance is less accurate. However, performance recovers quickly as the graphs grow; the steep surges in performance in Fig. 10 imply a small anomaly detection delay. (*iii*) The average precision after recovery indicates a near-ideal ranking of attack graphs at the top. (*iv*) The dips become less severe as the clustering becomes more 'mature' with increasing data seen; this is evident in the general upward trend of the accuracy. (*v*) The accuracy loss is not due to the ranking (since both AP and AUC remain high) but due to the chosen anomaly thresholds derived from the bootstrap clustering, where the error is due to false negatives.

Similar results hold for $B = 50$ as shown in Figure 9 (c) for ALL, and (a) and (b) for YDC and GFC respectively.

Sketch size. Figures 9 and 10 show StreamSpot's performance for sketch size $L = 1000$ bits. When compared to the number of unique shingles $|S|$ in each dataset (649,968, 580,909 and 1,106,684 for YDC, GFC and ALL), sketching saves considerable space. Reducing the sketch size saves further space but increases the error of cosine distance approximation. Figure 11 shows StreamSpot's performance on ALL for smaller sketch sizes. Note that it performs equally well for $L = 100$ (compared to Fig. 9(c)), and reasonably well even with sketch size as small as $L = 10$.



(a) $L = 100$ (b) $L = 10$

Figure 11: Performance of StreamSpot on ALL for different values of the sketch size.

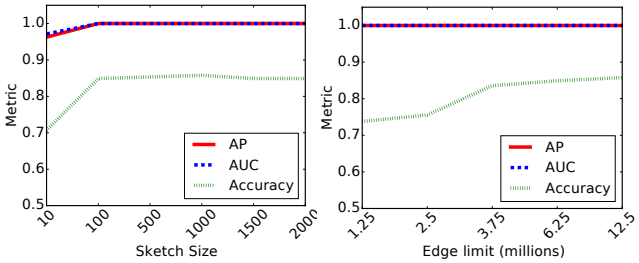


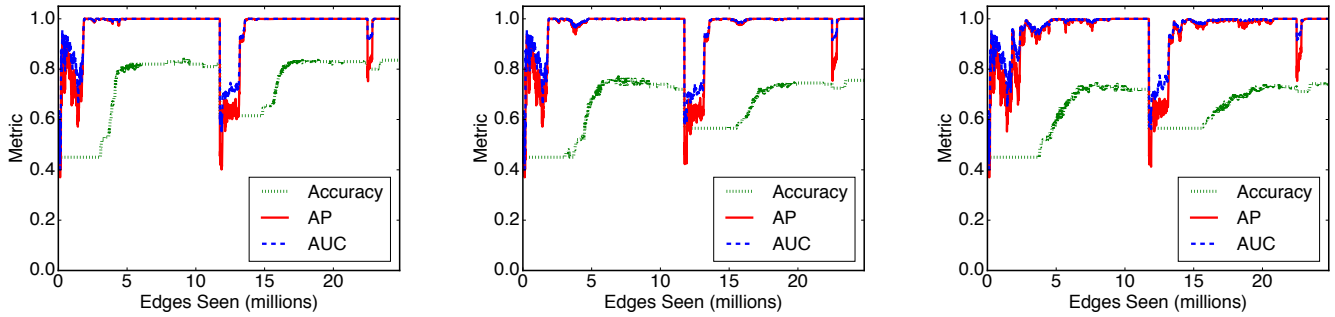
Figure 13: StreamSpot performance on ALL with (left) sketch size L and (right) memory limit N.

Memory limit. Now we investigate StreamSpot's performance by limiting the memory usage using N: the maximum number of allowed edges in memory at any given time. Figures 12 (a)–(c) show the performance when N is limited to 15%, 10%, and 5% of the incoming stream of ∼25M edges, on ALL (with $B = 100$, $L = 1000$). The overall performance decreases only slightly as memory is constrained. The detection delay (or the recovery time) increases, while the speed and extent of recovery decays slowly. This is expected, as with small memory and a large number of graphs growing simultaneously, it takes longer to observe a certain fraction of a graph at which effective detection can be performed.

These results indicate that StreamSpot continues to perform well even with limited memory. Figure 13 shows performance at the end of the stream with (a) increasing sketch size L (N fixed at 12.5M edges) and (b) increasing memory limit N (L fixed at 1000). StreamSpot is robust to both parameters and demonstrates stable performance across a wide range of settings.

Running time[6]**.** To evaluate the scalability of StreamSpot for high-volume streams, we measure its per-

[6] On an Intel Xeon® E7 4830 v3 at 2.1Ghz with 1TB RAM.



(a) Limit = 15% (b) Limit = 10% (c) Limit = 5%

Figure 12: Performance of StreamSpot on ALL ($L = 1000$), for different values of the memory limit N (as a fraction of the number of incoming edges).

edge running time on ALL for sketch sizes $L = 1000, 100$ and 10 averaged over the stream of ∼25M edges in Fig. 14 (left). Each incoming edge triggers four operations: updating the graph adjacency list, constructing shingles, updating the sketch/projection vector and updating the clusters. We observe that the running time is dominated by the sketch-update time (distribution in Fig. 14 (right)), but the total per-edge running time is under $70\mu s$ when $L = 1000$. Thus, StreamSpot can scale to process over 14,000 edges per second. For $L = 100$, which performs comparably (see Fig. 11 (a)), it can further scale to over 100,000 edges per second.

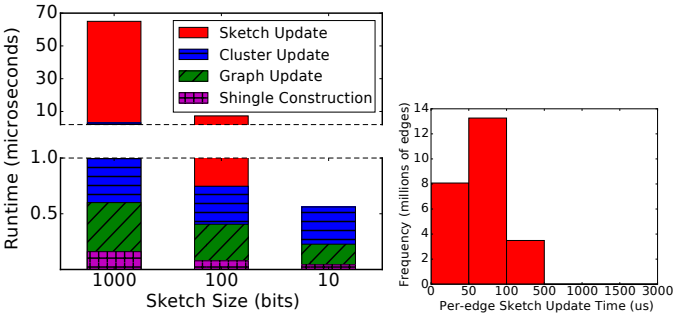


Figure 14: (left) Average runtime of StreamSpot per edge, for various sketch sizes on ALL. (right) Distribution of sketch processing times for ∼25M edges.

Memory usage. Finally, we quantify the memory usage of StreamSpot. Table 2 shows the total memory consumed for ALL by graph edges (M_G) in megabytes and by the projection and sketch vectors (M_Y, M_X) in kilobytes with increasing N (the maximum number of edges stored in memory). We also mention the number of graphs retained memory at the end of the stream. Note that M_G grows in proportion to N, and M_Y is 32 times M_X, since projection vectors are 32-bit integer vectors of the same length as the sketch vectors. Overall, StreamSpot's memory consumption for $N = 12.5$M and $L = 1000$ is as low as 240 megabytes, which is comparable to the memory consumed by an average process on a commodity machine.

In summary, these results demonstrate the effectiveness as well as the time and memory-efficiency of StreamSpot.

6. RELATED WORK

Our work is related to a number of areas of study, including graph similarity, graph sketching and anomaly detection in streaming and typed graphs, which we detail further in this section.

Table 2: StreamSpot memory use on ALL ($L = 1000$). $|G|$: final # graphs in memory. Memory used by M_G: graphs, M_Y: projection vectors, M_X: sketches

N	$\|G\|$	M_G	M_Y	M_X
1.25 M	8	23.84 MB	31.25 KB	0.98 KB
2.5 M	29	47.68 MB	113.28 KB	3.54 KB
3.75 M	42	71.52 MB	164.06 KB	5.13 KB
6.25 M	56	119.20 MB	218.75 KB	6.84 KB
12.5 M	125	238.41 MB	488.28 KB	15.26 KB

Graph similarity. There exists a large body of work on graph similarity, which can be used for various tasks including clustering and anomaly detection. Methods that require knowing the node correspondence between graphs are inapplicable to our scenario, as are methods that compute a vector of *global* metrics [7] for each graph (such as the average properties across all nodes), since they cannot take into account the temporal ordering of edges.

Graph-edit-distance (GED) [9] defines the dissimilarity between two graphs as the minimum total cost of operations required to make one graph isomorphic to the other. However, computing the GED requires finding an inexact matching between the two graphs that has minimum cost. This is known to be NP-hard and frequently addressed by local-search heuristics [19] that provide no error bound.

Graph kernels [33, 31] decompose each graph into a set of local substructures and the similarity between two graphs is a function of the number of substructures they have in common. However, these methods require knowing a fixed universe of the substructures that constitute all the input graphs, which is unavailable in a streaming scenario.

Heterogeneous/typed graphs. An early method [26] used an information-theoretic approach to find anomalous node-typed graphs in a large static database. The method used the SUBDUE [11] system to first discover frequent substructures in the database in an offline fashion. The anomalies are then graphs containing only a few of the frequent substructures. Subsequent work [12] defined anomalies as graphs that were mostly similar to normative ones, but differing in a few GED operations. Frequent typed subgraphs were also leveraged as features to identify non-crashing software bugs from system execution flow graphs [22]. Work also exists studying anomalous communities [16, 27] and community anomalies [13, 28] for attributed graphs. All of these approaches are designed for static graphs.

Streaming graphs. GMicro [3] clustered untyped graph streams using a centroid-based approach and a distance function based on edge frequencies. It was extended to graphs with whole-graph-level attributes [35], and node-level attributes [24] where the goal was to cluster nodes. GOutlier [4] introduced structural reservoir sampling to maintain summaries of untyped, undirected graph streams and detect anomalous graphs as those having unlikely edges. Classy [19] implemented a scalable distributed approach to clustering streams of untyped call graphs by employing simulated annealing to approximate the GED between pairs of graphs, and GED lower bounds to prune away candidate clusters.

There also exist methods that detect changes in graphs that evolve through community evolution, by processing the edge stream to determine expanding and contracting communities [2] and by applying information-theoretic [32] and probabilistic [15] approaches on graph snapshots to find time points of global community-structure change. Methods also exist to find temporal patterns called graph evolution rules [6], which are subgraphs with similar structure, types of nodes, and order of edges. All the aforementioned methods are primarily suited to untyped graphs.

Graph skeletons and sketches. Graph "skeletons" [18] were introduced to approximately solve a number of common graph-theoretic problems such as finding the global min-cut and max-flow with provable error-bounds. Skeletons were also applied [1] to construct compressed representations of disk-resident graphs for efficiently approximating and answering minimum *s-t* cut queries. Work on sketching graphs has primarily focused on constructing specialized sketches that enable approximate solutions to specific graph problems such as finding testing reachability and finding the densest subgraph [25]; the proposed sketches cannot be applied directly to detect graph-based anomalies.

7. CONCLUSION

We have presented StreamSpot to cluster and detect anomalous heterogeneous graphs originating from a stream of typed edges, in which new graphs emerge and existing graphs evolve as the stream progresses. We introduced representing heterogeneous ordered graphs by shingling and devised StreamHash to maintain summaries of these representations online with constant-time updates and bounded memory consumption. Exploiting the mergeability of our summaries, we devised an online centroid-based clustering and anomaly detection scheme to rank incoming graphs by their anomalousness that obtains over 90% average precision for the course of the stream. We showed that performance is sustained even under strict memory constraints, while being able to process over 100,000 edges per second.

While designed to detect APTs from system log streams, StreamSpot is applicable to other scenarios requiring scalable clustering and anomaly-ranking of typed graphs arriving in a stream of edges, for which no method currently exists. It has social media applications in event-detection using streams of sentences represented as syntax trees, or biochemical applications in detecting anomalous entities in streams of chemical compounds or protein structure elements.

Acknowledgments

This research is sponsored by the DARPA Transparent Computing Program under Contract No. FA8650-15-C-7561, NSF CAREER 1452425 and IIS 1408287, an R&D grant from Northrop Grumman, and a faculty gift from Facebook. Any conclusions expressed in this material are of the authors and do not necessarily reflect the views, expressed or implied, of the funding parties.

8. REFERENCES

[1] C. C. Aggarwal, Y. Xie, and P. S. Yu. Gconnect: A connectivity index for massive disk-resident graphs. *PVLDB*, 2009.
[2] C. C. Aggarwal and P. S. Yu. Online analysis of community evolution in data streams. In *SDM*, 2005.
[3] C. C. Aggarwal, Y. Zhao, and P. S. Yu. On clustering graph streams. In *SDM*, 2010.
[4] C. C. Aggarwal, Y. Zhao, and P. S. Yu. Outlier detection in graph streams. In *ICDE*, 2011.
[5] L. Akoglu, H. Tong, and D. Koutra. Graph based anomaly detection and description: a survey. *Dat. Min. Know. Disc.*, 2015.
[6] M. Berlingerio, F. Bonchi, B. Bringmann, and A. Gionis. Mining graph evolution rules. In *ECML/PKDD*, 2009.
[7] M. Berlingerio, D. Koutra, T. Eliassi-Rad, and C. Faloutsos. Network similarity via multiple social theories. In *ASONAM*, 2013.
[8] A. Z. Broder. On the resemblance and containment of documents. In *Compression and Complex. of Sequences*, 1997.
[9] H. Bunke and G. Allermann. Inexact graph matching for structural pattern recognition. *Pattern Recogn. Letters*, 1983.
[10] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.
[11] D. J. Cook and L. B. Holder. Graph-based data mining. *IEEE Intelligent Systems*, 2000.
[12] W. Eberle and L. B. Holder. Discovering structural anomalies in graph-based data. In *ICDMW*, 2007.
[13] J. Gao, F. Liang, W. Fan, C. Wang, Y. Sun, and J. Han. On community outliers and their efficient detection in information networks. In *KDD*, 2010.
[14] G. Grimmett and D. Stirzaker. *Probability and random processes*. Oxford University Press, 2001.
[15] M. Gupta, C. C. Aggarwal, J. Han, and Y. Sun. Evolutionary clustering and anal. of bibliographic networks. *ASONAM*, 2011.
[16] M. Gupta, A. Mallya, S. Roy, J. H. D. Cho, and J. Han. Local learning for mining outlier subgraphs from network datasets. In *SDM*, 2014.
[17] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*, 1998.
[18] D. R. Karger. Random sampling in cut, flow, and network design problems. In *STOC*, 1994.
[19] O. Kostakis. Classy: fast clustering streams of call-graphs. *Data Min. Knowl. Discov.*, 2014.
[20] D. Koutra, J. T. Vogelstein, and C. Faloutsos. Deltacon: A principled massive-graph similarity function. In *SDM*, 2013.
[21] D. Lemire and O. Kaser. Strongly universal string hashing is fast. *The Computer Journal*, 2014.
[22] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu. Mining behavior graphs for "backtrace" of noncrashing bugs. In *SDM*, 2005.
[23] F. T. Liu, K. M. Ting, and Z.-H. Zhou. Isolation forest. In *ICDM*, 2008.
[24] R. McConville, W. Liu, and P. Miller. Vertex clustering of augmented graph streams. In *SDM*, 2015.
[25] A. McGregor. Graph stream algorithms: a survey. *SIGMOD Record*, 2014.
[26] C. C. Noble and D. J. Cook. Graph-based anomaly detection. In *KDD*, 2003.
[27] B. Perozzi and L. Akoglu. Scalable anomaly ranking of attributed neighborhoods. In *SDM*, 2016.
[28] B. Perozzi, L. Akoglu, P. Iglesias Sánchez, and E. Müller. Focused clustering and outlier detection in large attributed graphs. In *KDD*, 2014.
[29] A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2011.
[30] P. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *J. Comp. Appl. Math.*, 1987.
[31] N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-Lehman graph kernels. *JMLR*, 2011.
[32] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu. Graphscope: Parameter-free mining of large time-evolving graphs. In *KDD*, 2007.
[33] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt. Graph kernels. *JMLR*, 2010.
[34] M. N. Wegman and J. L. Carter. New hash functions and their use in authentication and set equality. *J. of Comp. & Sys. Sciences*, 1981.
[35] P. S. Yu and Y. Zhao. On graph stream clustering with side information. In *SDM*, 2013.