Sans: Streaming Anonymized Network Sensing

Ketai Zhao
School of Computer Science
Nanjing University
lttmg@outlook.com

Yuhang Zhou
School of Computer Science
Nanjing University
yuhangzhou@smail.nju.edu.cn

Hongxu Pan
School of Computer Science
Nanjing University
221501029@smail.nju.edu.cn

Zhibin Wang
School of Computer Science
Nanjing University
wzbwangzhibin@gmail.com

Sheng Zhong
School of Computer Science
Nanjing University
sheng.zhong@gmail.com

Chen Tian
School of Computer Science
Nanjing University
tianchen@nju.edu.cn

***Abstract*—Large-scale network sensing is an important task with applications in various domains. Recently, researchers have proposed a network sensing algorithm based on GraphBLAS, which divides the input data into multiple disjoint blocks and constructs a graph for each block containing hypersparse network sensing data. However, this block-based approach may miss some anomalies between two consecutive blocks. In this paper, we aim to address this issue by developing a streaming anonymized network sensing systems, Sans. Specifically, Sans combines the advantages of directly maintaining edges in the hashtable and maintaining the vertices as well as the adjacent edges in the hashtable of list to develop a dynamic, efficient, and compressed data structure for hypersparse network sensing data. Furthermore, we develop an incremental calibration algorithm based on gradient descent by leveraging the previous analysis parameters. We also propose a parallel version of the algorithm, which supports shared-memory lock-based and distributed-memory lock-free designs. We conduct extensive experiments to evaluate the performance of the proposed streaming network sensing algorithm. The results demonstrate that Sans outperforms the static CSR (GraphBLAS) approach by one million times.**

***Index Terms*—Graph, Hashtable, Sliding Window, Network analysis, Parallelization**

I. Introduction

Large-scale network sensing [1] is an important research area with various applications, wireless sensor analysis [2], large-scale statistical cyber characterization of network traffic [3] [4], social recommendation systems [5]. Regarding the importance of large-scale network sensing, the GraphChallenge competition [6] has proposed the Anonymized Network Sensing Graph Challenge, which aims to construct and analyze anonymized traffic matrices from network packet capture (PCAP) data to enable open community-based approaches to protecting networks.

Recently, [1], [3], [4] proposed a network sensing algorithm based on GraphBLAS. Specifically, they first divide the input data into multiple disjoint blocks and construct a graph for each block containing hypersparse network sensing data by using GraphBLAS. Subsequently, the constructed graphs will be analyzed to identify whether there are any anomalies. Unfortunately, this block-based approach may miss some anomalies across two consecutive blocks. For example, the anomaly may happen in the time window starting from the half of the first block and ending at the half of the second block. The core reason is that the block-based approach transforms the *streaming* network sensing data into *static* network sensing data, which may lose some information.

In this paper, we aim to address this issue by developing a streaming network sensing algorithm. Specifically, we consider the analysis of the network sensing data within a sliding time window, e.g., $[0, B]$, $[1, B+1]$, etc. This requires online maintenance of hypersparse network sensor data and incremental analysis targeting dynamically shifting focused areas. Therefore, streaming network sensing faces several challenges as follows.

- **Dynamic, efficient and compressed data structure.** It is critical to carefully design a data structure to handle the hypersparse network sensing data. The data structure should be dynamic to support the streaming data, efficient to handle update and query operations, and compressed to limit memory usage. The existing data structures, e.g., compressed sparse row (CSR) [7], can not directly support the dynamic and streaming network sensing data.
- **Incremental analysis of the focusing and calibration.** Instead of analyzing the network sensing data from scratch for each window, it is necessary to leverage the previous analysis results to accelerate the computation. This requires an incremental analysis algorithm that can update and modify the analysis from the previous window. Notably, the calibration step in network sensing, involving intricate fitting with established distributions, poses challenges for algorithm design.
- **Parallelization.** In the block-based approach, due to the independence of blocks, the analysis of each block can be easily parallelized. However, in the streaming approach, the analysis of two consecutive windows may overlap, inevitably introducing dependencies, and making parallelization more challenging. Furthermore, to support parallel updates, stricter requirements are imposed on the design of the data structure.

To tackle these challenges, we propose a streaming anonymized network sensing systems, Sans. The main contributions of this paper are as follows.

A streaming network sensing framework, which can handle the hypersparse network sensing data in a sliding time window and support incremental analysis (Section III). We first review and analyze five possible data structures for the dynamic hypersparse network sensing data. However, none of them can satisfy all the requirements. CSR and edge lists are targeted for static graphs. Directly maintaining edges in the hashtable (HT for edge) does not support neighbor queries. Hierarchical data structures, e.g., hashtable of hashtable (HT of HT) and hashtable of list (HT of list), can support neighbor query but have either unbounded memory usage (HT of HT) or high complexity for add and remove operations (HT of list). To address these issues, we propose a novel data structure that combines the advantages of the HT for edge and HT of list. Specifically, we maintain the edges in the HT for edge and the vertices as well as the adjacent edges in the HT of the list, ensuring synchronization between them. Furthermore, leveraging the prior analysis parameters, we develop an incremental calibration algorithm based on gradient descent [8].

The parallelization of the streaming network sensing algorithm (Section IV). We consider two main-stream parallel data structure designs: shared-memory lock-based and distributed-memory lock-free. The former handles parallel updates by locking the data structure, e.g., atomic operations. The latter avoids locks by partitioning the data structure as well as the workload. We implement both two parallel versions of the streaming network sensing algorithm and conduct scalability experiments to identify the more efficient one.

Finally, we conduct extensive experiments to evaluate the performance of the proposed streaming network sensing algorithm (Section V). Compared with leveraging the static CSR (GraphBLAS) to handle the streaming network sensing data, Sans achieves a speedup of a million times. Even compared with the original block-based approach, which may miss some anomalies, Sans still outperforms it by $9.9\times$. Moreover, by scaling from 1 to 32 threads, we achieve a speedup of $5.6\times$ for the distributed-memory lock-free design. The results demonstrate the effectiveness and efficiency of the proposed streaming network sensing algorithm.

II. Existing Network sensing

Anomaly detection in network sensing involves the following steps: we need to transform the hypersparse network sensing data into a graph, focus on the active IP addresses, and calibrate the traffic distribution to detect anomalies.

A. Preprocessing and maintenance of network sensing data

The PCAP [9] file is a common format for storing network packet data. It contains a series of packets, each of which includes the source and destination IP addresses. Subsequently, we introduce how to transform the PCAP file into a hypersparse graph and maintain it.

Blocking of streaming network traffic. As indicated in [10], we should consider the traffic within any time window to reduce statistical fluctuations. Accordingly, the traditional way [1] to perform network sensing is to block the traffic into several disjoint blocks and analyze the traffic in each block. Moreover, two adjacent blocks can be merged to form a larger block with a time window of twice the size.

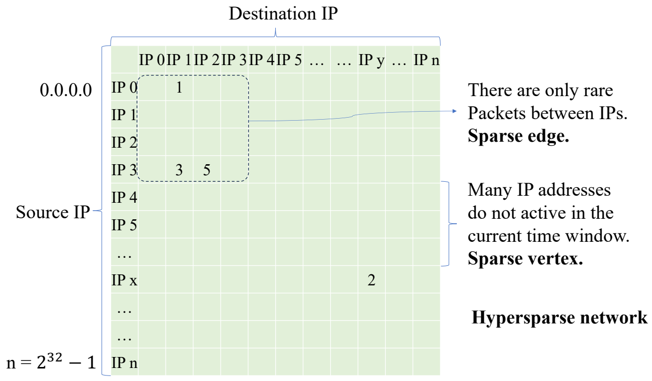


Fig. 1: Hypersparse adjacency matrix.

Unfortunately, *blocking method fails to capture the real-time traffic within a time window across two blocks*. Actually, the blocking method only senses the traffic in the pre-defined block, e.g., the packet data in $[0, B]$, $[B+1, 2B]$, etc., where B is the block size. However, the anomaly traffic may occur at any window, e.g., $[10, B+10]$, which fails to be captured by the blocking method.

Hypersparse network data. Within a given time window, the network sensing data can be elegantly represented as a graph. Here, individual IP addresses serve as vertices, while the transmission of packets from a source IP to a destination IP is conceptualized as a directed edge connecting the respective vertices. Additionally, the number of packets between the same source and destination IP pairs can be expressed as the weight of this edge.

In the real world, graphs are usually *sparse* [11], i.e., the number of edges is much smaller than the possible connections between vertices. Researchers have developed many data structures to store the sparse graph, e.g., adjacency list [12], edge list [13] and compressed sparse row (CSR) [7].

In addition to the sparsity of edges, the vertices also exhibit sparse characteristics within the time window of network sensing. Specifically, there are up to 2^{32} IP addresses, while only a small fraction of them have active packet transmission within this time window. Considering the sparsity of both edges and vertices, we call *hypersparse* [14] network.

Fig. 1 illustrates an example of a hypersparse adjacency matrix. Firstly, we observe that IP 0 to IP 3 have active packet transmission, while there are only 3 edges among them compared to the total possible 4×4 connections, indicating the sparsity of edges. Secondly, there are many IP addresses with no active packet transmission, IP 4 to IP x-1, reflecting the sparsity of vertices. The main reason for the hypersparse nature is there are $2^{32} \times 2^{32}$ possible connections, while only a small fraction of them (e.g., 2^{17}) have active packet transmission in the time window.

Maintenance of Hypersparse network data. Though the blocking method misses several possible anomalies, it provides the convenience of maintaining the network sensing data. Specifically, the data within each block are static, thereby allowing complex transformations to compress the graph into CSR format for further analysis.

B. Analysis

Focusing. The focusing step aims to find a range of IP addresses that are experiencing active packet transmission. In practice, it's common for certain subnets to experience significant communication over a while. For example, in environments such as university campuses, subnets dedicated to research departments may experience increased traffic due to activities such as video conferencing and remote debugging sessions. To identify these high-traffic subnets, an efficient approach is to calculate the total number of data packets exchanged within each subnet. Critical to this process is recognizing the hypersparse nature, where a majority of subnets may have relatively low traffic.

Calibration. The calibration step aims to distinguish whether the traffic within the focused range is normal or abnormal. Researchers observe that the normal transmission follows the heavy-tail distributions [15] (like Zipf-Mandelbrot [16]). By fitting the heavy-tail distributions to the data collected within a specific time window, we can accurately measure the extent of deviation from this expected pattern. This process enables us to pinpoint abnormal traffic that stands out from the norm.

III. Streaming network sensing

In this paper, we target to develop a streaming approach to network sensing, to detect anomalous traffic within arbitrary time windows. Diverging from the blocking method [17], our method is designed to enable sliding time window analysis, exemplified by the intervals $[0, B]$, $[1, B+1]$, and so on. Therefore, we introduce a novel data structure that dynamically maintains hypersparse network sensor data and enables incremental analysis tailored to the evolving focused range.

A. Dynamic hypersparse network data structure

Given the continuous evolution of network sensor data as the time window slides, we require a data structure capable of maintaining the data $A[i, j]$ online and efficiently supporting corresponding modification operations. Before delving into the details of the data structure, we first review the fundamental operations it supports. When sliding to the next window, we need to update the data structure by removing S oldest packets and adding S new packets, where S is the stride of the sliding window.

- **Add.** When a new packet is added, we first check if the source and destination IP addresses are already in the data structure. If not, we insert new IP addresses into the data structure. Similar operations are also performed for the edges, i.e., the link corresponding to the packet. If the link is already in the data structure, the count of the link, i.e., the weight of the edge representing the number of packets between the two IP addresses, is updated.
- **Remove.** After a packet is removed, we will first update the count of its corresponding link which is similar to the add operation. Then, if the count is decreased to zero, the link should be removed from the data structure.

As the network is hypersparse, we will not maintain all vertices (IP addresses) and edges (communication links) in the network. Instead, we will only maintain the active vertices and edges in the current window. In addition, we also have the following observations:

> The number of packets within a sliding window is limited. We can maintain the data structure in a fixed-size memory.

Subsequently, we summarize several possible data structures for the dynamic hypersparse network as well as their pros and cons in Table I.

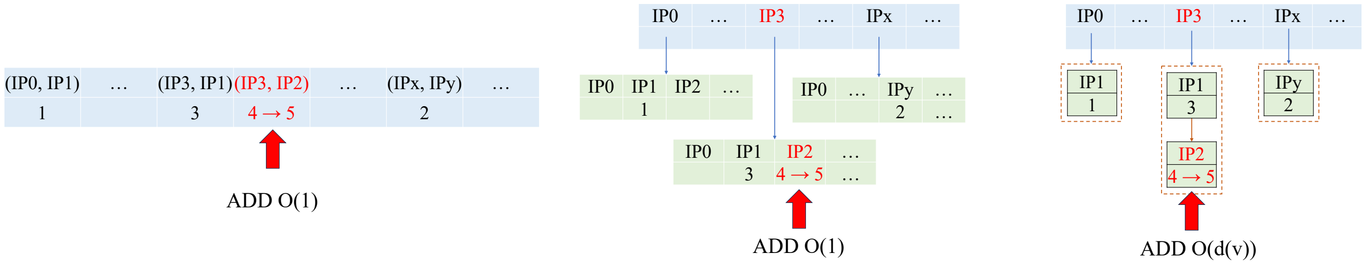
TABLE I: Comparison of data structures, where HT, $d(v)$ and E denotes hashtable, the degree of vertex v and the set of edges, respectively.

Category	structure	type	Add	Remove	Memory
Direct (edge)	Edge list	Static	$O(\lvert E\rvert)$	$O(\lvert E\rvert)$	Bounded
	HT for edge	Dynamic	$O(1)$	$O(1)$	Bounded
Hierarchical (vertex-adjacent)	CSR	Static	$O(\lvert E\rvert)$	$O(\lvert E\rvert)$	Bounded
	HT of HT	Dynamic	$O(1)$	$O(1)$	Unbounded
	HT of list	Dynamic	$O(d(v))$	$O(d(v))$	Bounded

Edge list and CSR (Compressed Sparse Row) are two common data structures for static graphs, which belong to two distinct categories of data structures: direct and hierarchical, respectively. The edge list, as a direct structure, straightforwardly maintains a listing of all edges in the graph. On the other hand, CSR, a hierarchical structure, organizes vertices at the first level and their adjacent edges on the second level. This hierarchical structure streamlines accessing a vertex's neighbors, a pivotal operation in network analysis and graph traversal tasks.

However, they are not suitable for dynamic graphs due to the high complexity of the add and deletion operations [18][19]. In contrast, the hashtable, which is a dynamic data structure with $O(1)$ complexity for add and deletion operations, is more suitable for dynamic graphs. And the three data structures derived from it for the dynamic hypersparse network shown in Fig. 2 have their pros and cons.

- *Hashtable for edge* can efficiently add and remove packets ($O(1)$ in Fig. 2a) but suffers from the inefficiency of accessing the neighbors of a vertex. Moreover, as the number of packets in the sliding window is limited, we can pre-allocate a bounded amount of memory for the hashtable, ensuring efficient use of resources.
- *Hashtable of hashtable* excels at both efficiently managing packet add/remove ($O(1)$ in Fig. 2b) and swiftly accessing a vertex's neighbors. However, the size of the inner hashtable that holds these neighbors is unbounded,



(a) HT for edge

(b) HT of HT

(c) HT of list

Fig. 2: Three data structures for the dynamic hypersparse network.

accommodating the varying number of neighbors a vertex may have across different windows.

- *Hashtable of list* effectively retrieves a vertex's neighbors but struggles with the inefficiency of packet add and remove ($O(d(v))$ in Fig. 2c), requiring a list traversal to locate the specific edge. Notice, that by pre-allocating a list pool and having all lists in the hashtable share this pool's memory, we can ensure that memory usage remains bounded.

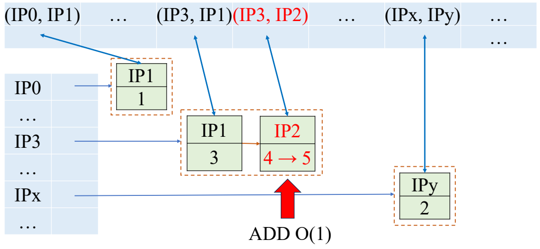


Fig. 3: Sans, combining HT for edge and HT of list.

B. Our solution

Firstly, the hashtable of hashtable is dismissed due to its potential for unbounded memory growth, which conflicts with the compression needs of hypersparse network. Given the pros and cons of the remaining two solutions, a clever approach is to blend them, harnessing the strengths of both to achieve an optimal balance.

Specifically, as shown in Fig. 3, we maintain a hashtable for all edges as well as a hashtable for adjacency lists corresponding to individual vertices. The hashtable for edges is responsible for adding and deleting packets, while concurrently maintaining the weight of each edge. On the other hand, the hashtable of list can be treated as a copy that facilitates access to the neighbors of a vertex. The synchronization between these two data structures is paramount; any modifications made to the hashtable for edges—be it the insertion of a novel edge (there is no corresponding edge for added packets), or the deletion of an edge upon its count dwindling to zero—trigger an automatic update to the corresponding adjacent list within the hashtable of list. Further, the synchronization is seamlessly achieved through pointers interconnecting them, enabling even the hashtable of list to perform addition and deletion operations with a time complexity of $O(1)$.

Constant memory chaining. Noticing the deletion operation is a frequent operation in our case, we choose chaining instead of open addressing to handle the collision in the hashtable. We allocate a fixed-size memory for the data structure, where the chained slots in the hashtable are pre-allocated as there are at most $|B|$ edges in the sliding window. When a link is freed from the data structure, the memory is not freed but will be reused for the next insertion.

C. Dynamic Sliding Window Analysis

When the stride is narrow, leading to numerous windows waiting for analysis, it is impractical to analyze each window from scratch. A more efficient way is to perform dynamic incremental analysis, continuously refining the results based on the previous window's analysis. The incremental focusing is easy to implement by tracking the packet count for each subnet. Therefore, we focus on the incremental calibration in this paper.

In the calibration phase, we aim to fit the weight of the links in the focused area to the Zipf-Mandelbrot distribution [16]:

$$P(d) = \frac{1}{Z}\left(\frac{1}{d+\theta}\right)^{\alpha}, \qquad (1)$$

where d is the weight of the link and P(d) is the probability of the link with weight d [1]. The calibration step determines the rest of the parameters α, θ, and Z to fit the distribution to the data. We adopt the maximum likelihood estimation (MLE) to estimate the parameters. Specifically, we minimize the mean squared error (MSE) between the empirical distribution and the Zipf-Mandelbrot distribution by gradient descent [8].

Instead of fitting the distribution for each window, we can fit the distribution for the first window. For the subsequent windows, we can reuse the parameters from the previous window and update the parameters by gradient descent.

IV. Parallelization

To further accelerate the computation, we develop a parallel version of the algorithm.

[1] Instead of considering the probability for each individual d, we consider the probability for a range of d, as suggested by [1].

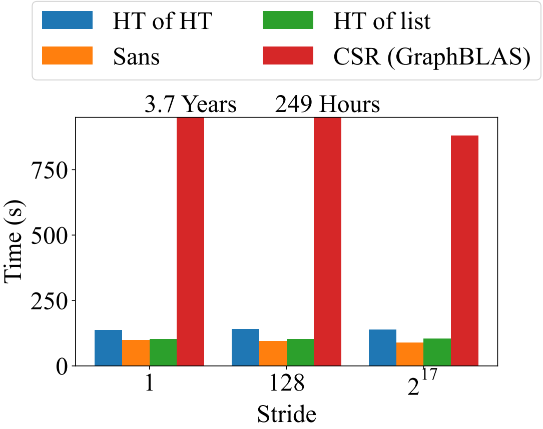


Fig. 4: Time consumption of four data structures.

Parallel Data Structure Design Two mainstream approaches for designing parallel data structures are the *shared-memory lock-based* and the *distributed-memory lock-free* models. The former employs locking mechanisms, such as atomic operations, to efficiently handle parallel updates. The latter, on the other hand, circumvents the need for locks by cleverly partitioning the data structure and workload. To facilitate our discussion, we take the hashtable for edge as an example.

- **Shared-memory lock-based.** The shared-memory lock-based design is straightforward by letting all threads access the data structure concurrently. As we employ the chaining method to handle hash collisions, we can lock the bucket where the edge is located. Subsequently, adding or deleting an edge, i.e., a slot in the bucket, can be done when the bucket is locked. However, the lock-based design may suffer from the contention issue, as multiple threads may access the same bucket concurrently.
- **Distributed-memory lock-free.** Firstly, we partition the hashtable into multiple segments, where each thread is responsible for a segment as well as the corresponding workload in this segment. When a batch of updates is received, we will distribute the updates to the corresponding threads maintaining the corresponding edges. The threads can update the edges concurrently without locks, as the edges are in different segments. However, it may suffer from the load imbalance issue, as the workload in different segments may vary.

To accelerate network analysis, we offer parallel version implementations of our algorithm in Sans using these two different strategies. Through corresponding scalability experiments (Section V-C), we explore the performance advantages and disadvantages of each strategy.

V. Experiments

A. Experimental setting

Sans is implemented with approximately 1000 lines of C++ code, utilizing the std::thread library. We evaluate Sans on

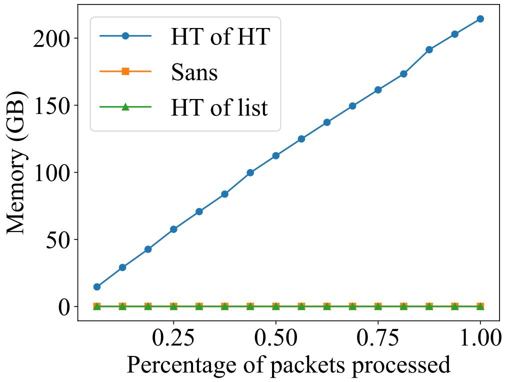


Fig. 5: Memory consumption of three methods with processed packets.

a server with a 56-core 2.0GHz Xeon Gold 6330 CPU, and 512 GB main memory. The dataset provided by GraphChallenge [6] is a 56 GB packet capture (PCAP) file, which consists of 2^{30} synthetic packets using randomly generated data.

In the following experiments, we assume that the packets are already stored in the main memory, meaning the time consumed by reading the PCAP file is ignored. Unless otherwise specified, the programs tested use 4 threads to update the hash tables, with a stride of 1 and a window size of 2^{17} following [1]. Moreover, as evaluated in Section V-C, the distributed-memory lock-free design is more efficient than the shared-memory lock-based design, so we choose the former for the parallel version of Sans. The experiments are repeated 3 times, and the average results are reported.

B. Compare with SOTA

The details of the data structures and algorithms used in the experiments are as follows:

- **CSR (GraphBLAS)**: We extend the blocking method in [1] which uses GraphBLAS to generate the CSR format of the graph to support the streaming sliding window analysis. Specifically, for each window, we generate the CSR format of the graph and analyze it from scratch. For small stride values, this strategy can not be finished in a reasonable time, and we estimate the time consumption by extrapolating the results of larger stride values.
- **HT of list and HT of HT**: We follow Section III to implement these dynamic data structures for the streaming network sensing data. Notice, for the HT of list approach, we pre-allocate the memory for the list to avoid the memory allocation overhead.
- **Sans**: We implement the proposed Sans algorithm which combines the advantages of HT for edge and HT of list to support the streaming network sensing data. Notice, since the HT for edge is the critical part of Sans, i.e., Sans has similar performance with HT for edge, we omit the HT for edge in the evaluation.

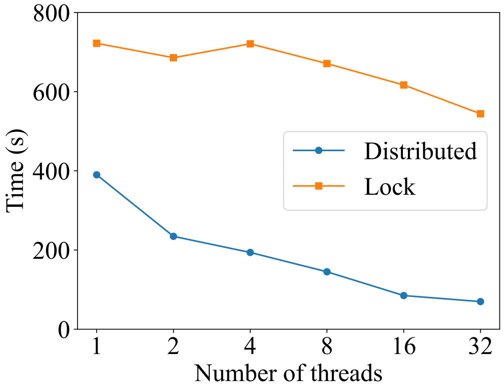


Fig. 6: Scalability.

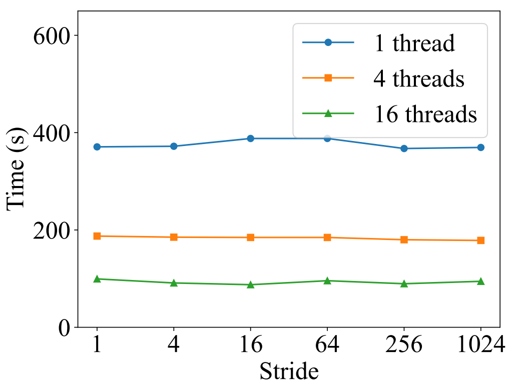


Fig. 7: Performance of Sans with varying strides.

Fig. 4 shows the comparison of the performance of these data structures. Obviously, the CSR (GraphBLAS) approach is significantly slower than the other approaches (a million times with stride=1), which is attributed to the fact that it generates the CSR format of the graph from scratch. However, even with a stride equal to the block size (2^{17}), the CSR (GraphBLAS) approach is still slower than Sans by $9.9\times$.

The HT of list approach is also slower than Sans by $1.1\times$ average due to the high complexity of add and remove operations, whose time complexity is related to the degree of the vertex. In contrast, the HT of HT achieves a similar performance with Sans, which is attributed to the fact that the HT of HT approach has the same time complexity $O(1)$ as Sans.

As mentioned in section III, the memory usage of the HT of HT design is not bounded. Fig. 5 reveals that it uses more than 200 GB of memory to process all the packets, which is significantly larger than the other approaches (125 MB). The memory allocation process also takes time, this is partially why the HT of HT approach has a relatively low performance than the HT for edge approach, despite they have the same theoretical time complexity.

C. Scalability

As mentioned in Section IV, we have two parallel designs, shared-memory lock-based (Lock) and distributed-memory lock-free (Distributed). We evaluate the scalability of the two designs by varying the number of threads from 1 to 32. We notice that even with 1 thread, the shared-memory lock-based design is significantly slower than the distributed-memory lock-free design ($1.85\times$), as it requires locking and unlocking the bucket for each edge when updating the hashtable, where the update operation is only increasing or decreasing the weight of the edge. With 32 threads, the shared-memory lock-based design is $7.8\times$ slower than the distributed-memory lock-free design, which is attributed to the contention issue when multiple threads access the same bucket concurrently.

From 1 to 32 threads, the shared-memory lock-based design has a speedup of $5.6\times$, while the distributed-memory lock-free design has a speedup of $1.3\times$. The relatively low performance of lock-based design is partially owing to the performance overhead of mutex locks when there is only 1 thread. Besides, when there are multiple threads, they may compete for shared resources of the hash table (e.g. the queue to store unused nodes), while the distributed-memory lock-free design does not have this issue.

D. Varying stride

As the stride will impact the number of analyzed sliding windows for anomaly detection, we evaluate the time consumption under different strides. We vary the stride from 1 to 1024 and observe that the time consumption slightly descends as the stride increases. Specifically, stride equals block size is only 5% faster than stride equals 1. This is attributed to our design of the dynamic data structure and incremental calibration algorithm, which avoids redundant computation. However, there still exists additional overhead when the stride decreases, because the parameters' gradients must be calculated at least once (to judge whether the current parameters fit well) for each window.

VI. Conclusion

To conclude, we propose a streaming anonymized network sensing system, Sans, which addresses the issue of missing anomalies between two consecutive blocks in the existing network sensing algorithm. Sans combines the advantages of directly maintaining edges in the hashtable and maintaining the vertices as well as the adjacent edges in the hashtable of list to develop a dynamic, efficient, and compressed data structure for hypersparse network sensing data. We also develop an incremental calibration algorithm based on gradient descent by leveraging the previous analysis parameters. Finally, we extend the algorithm to support parallelization. Extensive experiments are conducted to evaluate the performance of Sans, which demonstrates Sans outperforms the static CSR (GraphBLAS) approach by $1,647,156\times$.

References

[1] J. Kepner, M. Jones, P. Dykstra, C. Byun, T. Davis, H. Jananthan, W. Arcand, D. Bestor, W. Bergeron, V. Gadepally *et al.*, "Focusing and calibration of large scale network sensors using graphblas anonymized hypersparse matrices," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2023, pp. 1–9.

[2] C. F. García-Hernández, P. H. Ibarguengoytia-Gonzalez, J. García-Hernández, and J. A. Pérez-Díaz, "Wireless sensor networks and applications: a survey," *IJCSNS International Journal of Computer Science and Network Security*, vol. 7, no. 3, pp. 264–273, 2007.

[3] I. Kawaminami, A. Estrada, Y. Elsakkary, H. Jananthan, A. Buluc, T. Davis, D. Grant, M. Jones, C. Meiners, A. Morris, S. Pisharody, and J. Kepner, "Large Scale Enrichment and Statistical Cyber Characterization of Network Traffic," pp. 1–8, Sep 2022.

[4] J. Kepner, M. Jones, D. Andersen, A. Buluc, C. Byun, k. claffy, T. Davis, W. Arcand, J. Bernays, D. Bestor, W. Bergeron, V. Gadepally, D. Grant, M. Houle, M. Hubbell, H. Jananthan, A. Klein, C. Meiners, L. Milechin, A. Morris, J. Mullen, S. Pisharody, A. Prout, A. Reuther, A. Rosa, S. Samsi, D. Stetson, C. Yee, and P. Michaleas, "Temporal Correlation of Internet Observatories and Outposts," in *Workshop on Graphs, Architectures, Programming, and Learning (GrAPL)*, May 2022.

[5] W. Zhou, Y. Zhou, J. Li, and M. H. Memon, "Lsrec: Large-scale social recommendation with online update," *Expert Systems with Applications*, vol. 162, p. 113739, 2020.

[6] "Graph Challenge," https://graphchallenge.mit.edu.

[7] A. George and J. W. Liu, *Computer Solution of Large Sparse Positive Definite*. Prentice Hall Professional Technical Reference, 1981.

[8] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[9] Daqscribe, "Introduction to packet capture (pcap)," n.d., accessed: 2024-07-14. [Online]. Available: https://daqscribe.com/wiki/resources/introduction-to-packet-capture-pcap/

[10] J. Kepner, V. Gadepally, L. Milechin, S. Samsi, W. Arcand, D. Bestor, W. Bergeron, C. Byun, M. Hubbell, M. Houle *et al.*, "Streaming 1.9 billion hypersparse network updates per second with d4m," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–6.

[11] D. Eppstein, M. Löffler, and D. Strash, "Listing all maximal cliques in large sparse real-world graphs," *ACM J. Exp. Algorithmics*, vol. 18, nov 2013. [Online]. Available: https://doi.org/10.1145/2543629

[12] H. Singh and R. Sharma, "Role of adjacency matrix & adjacency list in graph theory," *International Journal of Computers & Technology*, vol. 3, no. 1, pp. 179–183, 2012.

[13] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," *ACM Transactions on Storage (TOS)*, vol. 15, no. 4, pp. 1–40, 2020.

[14] M. Jones, J. Kepner, A. Prout, T. Davis, W. Arcand, D. Bestor, W. Bergeron, C. Byun, V. Gadepally, M. Houle, M. Hubbell, H. Jananthan, A. Klein, L. Milechin, G. Morales, J. Mullen, R. Patel, S. Pisharody, A. Reuther, A. Rosa, S. Samsi, C. Yee, and P. Michaleas, "Deployment of real-time network traffic analysis using graphblas hypersparse matrices and d4m associative arrays," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, 2023, pp. 1–8.

[15] M. E. Crovella, M. S. Taqqu, and A. Bestavros, "Heavy-tailed probability distributions in the world wide web," *A practical guide to heavy tails*, vol. 1, pp. 3–26, 1998.

[16] T. M. Łapiński, "Law of large numbers unifying maxwell–boltzmann, bose–einstein and zipf–mandelbort distributions, and related fluctuations," *Physica A: Statistical Mechanics and Its Applications*, vol. 572, p. 125909, 2021.

[17] M. Dohler, Y. Li, B. Vucetic, A. H. Aghvami, M. Arndt, and D. Barthel, "Performance analysis of distributed space-time block-encoded sensor networks," *IEEE Transactions on Vehicular Technology*, vol. 55, no. 6, pp. 1776–1789, 2006.

[18] D. Eppstein, Z. Galil, and G. F. Italiano, "Dynamic graph algorithms," *Algorithms and theory of computation handbook*, vol. 1, pp. 9–1, 1999.

[19] B. Wheatman and H. Xu, "Packed compressed sparse row: A dynamic graph representation," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.