Multilevel Scalability for Effective Detection of Insider Threats

Ajayi Adebowale
Computer Science & Engineering
Babcock University
Ilishan-Remo, Nigeria
deboxyl@gmail.com

Ajayi Olutayo
ICT Resource Centre
Federal University of Agriculture
Abeokuta, Nigeria
olutayoajayi@gmail.com

Idowu Sunday
Computer Science & Engineering
Babcock University
Ilishan-Remo, Nigeria
Saidowu07@gmail.com

Ogbonna A.C
Computer Science & Engineering
Babcock University
Ilishan-Remo, Nigeria
acogbonna@yahoo.com

Otusile Oluwabukola
Computer Science & Engineering
Babcock University
Ilishan-Remo, Nigeria
buhkieotusile@yahoo.com

Ajayi Ibironke A
Department of Computer Science
Federal College of Education
Abeokuta, Nigeria
tayoajayi@unaab.edu.ng

Abel Samuel
Computer Science & Engineering
Babcock University
Ilishan-Remo, Nigeria
abelsammie@yahoo.com

***Abstract*—** Insider threats remain one of the oldest and notorious threats to information security. Early detection remains key to preventing insider attacks on an information system. The vast amount of enterprise data and the little data points pertaining to insider threats calls for techniques to handle the rare class problem. This study conceptualised insider threat data as a streaming data problem. Scalability of insider threat detection systems represents a gap in knowledge in this disposition. Building on existing unsupervised ensemble stream mining techniques, this study proposed an insider threat detection algorithm and evaluated it using Centre for Analysis of Internet Data (CAIDA) Anonymized trace dataset for 2015. CAIDA datasets was used to ascertain the scalability of quantised dictionary construction by applying a distributive approach to graph based anomaly detection (GBAD). Pattern learning anomaly detection system processes GBAD in a streaming approach. Dictionary construction was done using Apache Spark on top of the Hadoop stack. Pattern Learning Anomaly Detection System (PLADS) enhanced GBAD successfully discovered the same anomalous substructure within a streaming approach in a fraction of the time (642 seconds) it took to process the entire graph (59,743 seconds) when applied on the CAIDA Anonymised 2015 dataset. Application of Apache Spark as the distributed computing framework for construction of quantised dictionaries of user command data depicted a reduction in processing time under varying input sizes and number of reducers In conclusion, scalability of Insider Threat Detection systems is essential and a complexity analysis of proposed algorithms showed it scales to increased number of users of the system. The implemented prototype system using Apache Spark scaled to increasing workloads showing its usefulness for early detection of insider threats. This study recommends the use of unsupervised learning ensembles and distributed frameworks for effective detection of insider threats.

Keywords*:* Gbad, Plads, Caida, Quantised Dictionary, Apache Spark, Hadoop

I. INTRODUCTION

Insider threats and problems are critical issues that hinge on overall security. In reference [1] organisations such as RAND and CERT dedicate extensive research to these threats and problems. However, insider definition still lacks consistency in research. Majority of research uses binary approach to define an insider, ascertaining that an insider falls with the employment category parameter. They define an insider as a trusted member with the power to violate security policy. Hence, there exist two broad categories of insider threat: malicious and accidental. The malicious insiders make a conscious decision to intentionally cause harm to the organization. They understand their actions and also recognise the damage it can have on the organization while in contrast, accidental insiders are targeted by adversaries and manipulated to carry out actions that the insider believes to be legitimate but in reality represent a threat to the organization. Such insiders are unaware that their actions are harmful and people in such category might simply be negligent of security practices or through breaches in improper handling of data, systems and networks.

Whether malicious or accidental, insider threats continue to pose a great peril to the integrity, availability and confidentiality of information system in many organizations [2, 3]. Development of detection systems to combat insider threats remains a viable research area in information security.

Data related with insider threat detection and classification are frequently continuous. In such systems, the pattern of the average users and insider threats can slowly evolve over time [4]. A programmer can develop his skills to become an expert over time, therefore an insider can change his actions to imitate legitimate user processes. In any case, the patterns at either end of these developments can look extremely different when compared directly with each other. The traditional static unsupervised methods raise false alarms with these cases because they are unable to handle them when they arise in the system. Learning models must show knowledge in handling evolving concepts and also exhibit high efficiency in building models building models for bulky amounts of data so as to quickly detect threats.

This study conceptualised insider threat problems as a stream mining problem that applies to continuous data stream and proposed an ensemble of unsupervised learning methods for efficiently detecting anomalies in stream data. An evolving ensemble of classifier models was used to cope with concept- drift as the behaviour of valid and invalid agents varies over time.

As breaches continually cause significant damages to organisations, security consciousness is now shifting from the traditional perimeter defence to a holistic understanding of what causes these damages and where organisations are critically exposed. Nevertheless, although a good number of attacks are from external sources, attacks from within cause the most substantial damage [5]. Why insider? It is because the insider has unlimited access to sensitive data as well as the means, motives and methods to access information virtually undetectable. The results of a survey on insider threat [5] shows that organizations are beginning to recognize the importance of protecting themselves against insider threat.

Old security models are oblivious of insider threats. As organizations shore up tools for preventing external invader from gaining access to their network, they operate under the assumption that those with internal access are trustworthy. Even with this assumption, inattention and compromised credentials are also hazardous.

An external attacker only needs to request for credentials via phishing or other social engineering. If this method fails, then they cut their losses which comprised of the time spent composing an email. However, if they succeed, they have all the privileges a legal user and all the access of an insider. On the other hand, a compromised insider has the necessary access privileges. That fact alone heightens the potential for damage especially as most organizations do not monitor their internal network traffic. An inside attacker can progressively and assuredly conduct reconnaissance and collect data at leisure. Once all the target information is packaged in a central location on the network, the invader can then move it out of the network, at that point, the data is gone.

What is apparent from that backdrop is that it is almost impractical to stop an insider threat at the gate. This lays bare the extreme importance of securing data before it is too late to prevent an attack, in this, early detection is significant. Fortunately, an invader is not satisfied with initial breach, as the perpetrator still has to execute a number of steps before their aim can be complete. Stopping the perpetrator becomes the goal. The first thing an organization needs to catch an insider threat is network visibility. Internal network traffic, access logs, policy violations and other similar processes needs to be watched continuously for suspicious and unusual activity.

However, manually monitoring network activity in a very large organisation securely is a humanly impossible task. Vital tracking clues that can point to unusual activities are swiftly drowned out by the plethora of other information. In such a circumstance, data analytics can make a huge difference.

The increasing occurrence of insider attacks and inadequacy of available detection systems in combating the menace is a major motivation for this research.

II. SCALABILITY ISSUES IN GRAPH LEARNING

Stream data continuously flow with great speed and they are very large in size [6]. This agrees to the characteristics of big data. Big data is a data whose scale, diversity and complexity requires new architecture, techniques, algorithms and analytics to manage it and extract value and hidden knowledge from it. Hence, big data researchers are looking for tools to analyse, manage, summarize, visual and discover knowledge from the collected data in a timely manner and scalable fashion.

Adding heterogeneous dataset to the network traffic, represented as graph could provide the basis for discovering interesting structural patterns and anomalies, which may alert a security analyst to the potential threat attempt in the form of a network intrusion, denial-of-service attack or worms. However, computer network traffic is typically voluminous and acquired in real time as stream of information.

In other words, anomalies are majorly associated with illegal activity that tries to imitate normal behaviour. Thus, if some set of data is represented as graph, any nefarious activity would be identifiable by small modifications, insertions or deletions to normative patterns with the graph.

The initial approach graph-based anomaly detection called GBAD [7] uses a based measure to find the normative patterns and also analysed the close matches to find the normative patterns so as to determine if they meet the definition of anomaly.

However, while this approach demonstrated its effectiveness in a variety of domains [8], the issue of scalability has limited this approach when dealing with domains containing millions of nodes and edges. In addition, many graphs of interest are dynamic i.e. changes to the graph are streaming in over time. This further complicates the analysis because a static graph cannot be analysed, snapshots would need to be analysed over a period of time. Moreover, the streaming graph instance offer an opportunity for methods that can update the current set of patterns and anomalies based on the changes to the graph rather than repeated analyses on the large graph snapshots.

III. RELATED WORK

Relevant data to insider threats are typically accumulated over many years of system operations in an organization, and therefore it is best characterised as an unbounded data stream. Such stream can be partitioned into a sequence of discrete chunks; for example, each chunk might comprise a week's worth of data.

In order to mitigate the effects concept drift has on efficient classification of insider threat related data (unbounded continuous stream data). An approach that uses a single MapReduce job for constructing quantised dictionaries [4] was proposed for ensembles of Graph learning models and One Class Support Vector Machine (OCSVM). The work showed that reducing the number of MapReduce jobs reduces the total processing cost. Experiments were carried out to validate the conclusion that mapreduce jobs and processing costs are directly proportional.

Processing costs for a Hadoop job is associated with disk I/O and network transfers.

The following takes place when a job is sent to a Hadoop cluster:

1. The Executable file is moved from client machine to Hadoop JobTracker,
2. The JobTracker determines TaskTrackers that will execute the job,
3. The Executable file is distributed to the TaskTrackers over the network,
4. Map processes initiates reading data from Hadoop Distributed File System (HDFS),
5. Map outputs are written to local discs,
6. Map outputs are read from discs, shuffled (transferred over the network to TaskTrackers which would run Reduce processes), sorted and written to remote discs,
7. Reduce processes initiate reading the input from local discs.
8. Reduce outputs are written to discs.

Algorithm 1 Dictionary construction and compression using single Map-Reduce (1MRJ) [4]

1. Input: *gname* : *groupname*, *cseq* : *command sequences*
2. Output: *Key* : *gname*,*commandpattern*(*css*)
3. *map*(*stringgname*,*stringcseq*)
4. *start* ← 1, *end* ← 2
5. *css* = (*csstart* ⋯*csend*)
6. if *css* 6∈ *dictionary* then
7. *Add css to thedictionary*
8. *emit*(*gname*,*css*)
9. *start* ← *end*
10. *end* ← *end* + 1
11. Else
12. *emit*(*gname*,*css*)
13. *end* ← *end* + 1
14. end if
15. *reduce*(*gname*,(*css*$_1$,*css*$_2$,⋯)
16. *H* ← 0
17. for all *css*$_i$ ∈ ((*css*$_1$,*css*$_2$,⋯) do
18. if *css* ∈6 *H* then
19. *H* ← *H* + (*css*$_i$,1)
20. Else
21. *count* ← *getfrequency*(*H*(*css*$_i$))
22. *count* ← *count* + 1
23. *H* ← *H* + (*css*$_i$,*count*)
24. end if
25. end for
26. *QD* = *QuantizedDictionary*(*H*)

27. for all $css_i \in QD$ do
28. $emit(gname, pair(css_i, count(css_i)))$
29. end for

Mapper emitted user id as key and value as pattern (Algorithm 1) and in mapper partial LZW operation was completed. The same user id arrived at the same reducer since users were the intermediate key. For user id, the reducer had a list of patterns. In the reducer, incomplete LZW was also completed. In addition, full quantization operation was implemented. Parallelization was achieved at the user level (inter-user parallelization) instead of within users (intrauser parallelization). In mapper, parallelization was carried out by dividing large files into a number of chunks and process a certain number of files in parallel.

Algorithm 1 illustrated the idea. The input file consisted of line by line input. Each line had entries namely, gname (userid) and command sequences (cseq). The mapper then takes gname (userid) as key, and values was command sequences for that user. In mapper, they looked for patterns having length 2, 3, etc. The dictionary was checked for existing patterns (line 6), if the pattern does not exist in the dictionary, it then adds it in the dictionary (line 7), and intermediate key value pairs (line 8) are emitted having keys as gname and values as patterns with length 2, 3 and so on. At line 9 and 10, pointer was incremented so that patterns in new command sequences (cseq) can be identified, if the pattern is in the dictionary, the algorithm emits at line 12 and cseq's end pointer is incremented so as to look for super-set command sequence.

At the reducer, each user (gname) was input and list of values was pattern. The compression of patterns was carried for that user and some patterns was pruned using Edit distance. For a user each pattern was stored into Hashmap, H. Each new entry in the H was pattern as key and value as frequency count. For existing pattern in the dictionary, frequency count was updated (line 18). At line 20 dictionary was quantized, and H was updated accordingly. Therefore from the quantized dictionary all distinct patterns from H was emitted as values along with key gname.

IV. APACHE SPARK

Existing distributed processing frameworks have been successfully applied in numerous acyclic data intensive domains. Apache Spark however, is more suited to stream data classification based on its intrinsic design features. A major novelty of the Apache Spark framework is its use of resilient distributed datasets (RDD). RDD represents a collection of read only objects partitioned across a cluster of machines. With Spark, users can cache an RDD in memory across machines in mapreduce-like parallel operations. This is particularly useful in systems that require iterative machine learning algorithms giving real time query results from a single scan of data. Hadoop for instance requires a 10 minute factor latency in its processing. This makes it inherently suitable for batch processing. Apache Spark is built on a cluster operating system called Mesos. Mesos allows multiple parallel applications share a cluster in a fine-grained manner and provides an API for such applications to launch tasks in the cluster. Spark works well with other frameworks such as through its Mesos ports on Hadoop and Message Passaging Interface (MPI).

Distributed Shared Memory

Spark's resilient distributed datasets can be viewed as an abstraction for distributed shared memory (DSM), which has been studied extensively [9]. RDDs differ from DSM interfaces in two ways. First, RDDs provide a much more restricted programming model, but one that lets datasets be rebuilt efficiently if cluster nodes fail. While some DSM systems achieve fault tolerance through check pointing [10], Spark reconstructs lost partitions of RDDs using lineage information captured in the RDD objects. This means that only the lost partitions need to be recomputed, and that they can be recomputed in parallel on different nodes, without requiring the program to revert to a checkpoint. In addition, there is no overhead if no nodes fail. Second, RDDs push computation to the data as in MapReduce [11], rather than letting arbitrary nodes access a global address space. Other systems have also restricted the DSM programming model to improve performance, reliability and programmability.

[12] worked on communication in linda. Linda provides a tuple space programming model that may be implemented in a fault-tolerant fashion. [13] implemented Munin. Munin lets programmers annotate variables with the access pattern they will have so as to choose an optimal consistency protocol for them. [14] worked on safe and efficient sharing of persistent objects in thor. Thor provides an interface to persistent shared objects. Cluster Computing Frameworks: Spark's parallel operations fit into the MapReduce model [11], however they operate on RDDs that can persist across operations.

The need to extend MapReduce to support iterative jobs was also recognized by [15, 16], a MapReduce framework that allows long-lived map tasks to keep static data in memory between jobs. However, Twister does not currently implement fault tolerance. Spark's abstraction of resilient distributed datasets is both fault-tolerant and more general than iterative MapReduce. A Spark program can define multiple RDDs and alternate between running operations on them, whereas a Twister program has only one map function and one reduce function. This also makes Spark useful for interactive

data analysis, where a user can define several datasets and then query them.

Spark's broadcast variables provide a similar facility to Hadoop's distributed cache, which can disseminate a file to all nodes running a particular job. However, broadcast variables can be reused across parallel operations.

Language Integration: Spark's language integration is similar to that of DryadLINQ [17], which uses .NET's support for language integrated queries to capture an expression tree defining a query and run it on a cluster. Unlike DryadLINQ, Spark allows RDDs to persist in memory across parallel operations. In addition, Spark enriches the language integration model by supporting shared variables (broadcast variables and accumulators), implemented using classes with custom serialized. IPython [18] is an interpreter for scientists that lets users launch computations on a cluster using a fault-tolerant task queue interface or low-level message passing interface. Spark provides a similar interactive interface, but focuses on data-intensive computations. This study used a python implementation of Adaboost in creating in its ensemble of unsupervised algorithms and benefitted from the language integration support of Apache Spark.

Lineage: Capturing lineage or provenance information for datasets has long been a research topic in the scientific computing and database fields, for applications such as explaining results, allowing them to be reproduced by others, and re-computing data if a bug is found in a workflow step or if a dataset is lost. Spark provides a restricted parallel programming model where fine-grained lineage is inexpensive to capture, so that this information can be used to re-compute lost dataset elements.

V. CAIDA Dataset

This dataset contains the passive traffic traces that CAIDA took in 2008 and made anonymized versions available for download to academic and government researchers, and CAIDA members.

The CAIDA Anonymized Internet Traces 2015 Dataset

This dataset contains anonymized passive traffic traces from CAIDA's equinix-chicago monitor on high-speed Internet backbone links. This data is useful for research on the characteristics of Internet traffic, including application breakdown, security events, topological distribution, and flow volume and duration. Starting with the 2014 dataset the yearly passive trace datasets will only contain one trace per quarter (previous years contain one trace per month). While we still collect a one-hour trace each month (and add statistics about each trace to the trace statistics page), we are forced by storage limitations to select only one of the three traces for each quarter for inclusion in this yearly collection. Traffic traces in this dataset are anonymized using CryptoPan prefix-preserving anonymization. All traces in this dataset are anonymized with the same key. In addition, the payload has been removed from all packets.

The Endace network cards used to record these traces provide timestamps with nanosecond precision. However, the anonymized traces are stored in pcap format with timestamps truncated to microseconds. The original nanosecond timestamps are provided as separate ascii files alongside the pcap files. The traces can be read with any software that reads the pcap (tcpdump) format, including the CoralReef Software Suite, tcpdump, Wireshark, and many others.

VI. TESTING ANOMALY DETECTION

For a quantized dictionary, it is important to determine the sequence in the data stream which can raise potential threat. To formulate the problem, given a data stream S and Ensemble E where $E = QD_1,QD_2,QD_3,...$ and $QD_i = qd_{i1},qd_{i2},...$, any pattern in the data stream is considered as an anomaly if it deviates from all the patterns qd_{ij} in *E* by more than *X*%.

To find the anomalies, matching patterns are identified and deleted from stream S, so that patterns from the data stream S that is an exact match or α edit distance away from any pattern, qd_{ij} in *E* is then considered as matching pattern. α can be ½, 1/3, or ¼ of the length of the particular pattern in qd_{ij}. The remaining patterns in the stream will then be considered as anomalies.

To identify the non- matching patterns in the stream S,a distance matrix *L* was computed which contained the edit distance between each pattern, qd_{ij} in *E* and the data stream *S*. When there is an exact match the proposed algorithm moves backwards exactly the length of qd_{ij} so as to find the starting point of that pattern in S and delete it from the data stream. But if there is an error in the match which is greater than (>) 0 but less than (<) α, the algorithm traverse either left or diagonal or up within the matrix according to which value is mentioned to find the starting point of that pattern in the data stream and when it is found, the pattern is deleted from the data stream and the remaining pattern will be considered as anomalous.

Complexity Analysis

The time complexity of quantized dictionary construction in order to calculate edit distance between two patterns of length *K* (in worst case maximum length would be *K*), the worst case time complexity would be $O(K^2)$.

The time complexity of finding edit distance between two patterns is $O(K^2)$ and there are total *n* number of distinct patterns, the total time complexity between a

particular pattern and the rest of patterns will be $O(n \times K^2)$. Since a particular pattern is one of the member of n patterns, total time complexity between pair of patterns is $O(n^2 \times K^2)$. This is valid for a single user. If there are u of distinct users, total time complexity across u user is $O(u \times n^2 \times K^2)$.

VII. HADOOP CLUSTER

The hadoop cluster used in this study (phadoop0-phadoop9) is comprised of virtual machines that run in the Computer Science vmware esx cloud - so there are 10 VM's. Each VM is configured as a quad core with 4GB of ram and a 256GB virtual hard drive. The virtual hard drives are stored on the CS SAN (3PAR).

There are three ESX hosts which are Dell Poweredge R720's with 12 cores @2.99GHZ, 128GB of RAM, and fiber to the 3PAR SAN. The VM's are spread across the three ESX hosts in order to balance the load.

"cshadoop0" is configured as the "name node". A "phadoop1" through "phadoop9" are configured as the slave "data nodes". Implementation was done using Java JDK version 1.6.0.39. For MapReduce implementation Hadoop version 2.6.0 was used.

VIII. RESULT OF SCALABILITY EXPERIMENTS ON CAIDA DATASET

The Cooperative Association for Internet Data Analysis (CAIDA) is a publicly available resource for the analysis of IP traffic. Through a variety of workshops, publications, tools, and projects, CAIDA provides a forum for the dissemination of information regarding the interconnections on the internet. One of the core missions of CAIDA is to provide a data repository to the research community that will allow for the analysis of internet traffic and its performance (http://www.caida.org/data/). Using GBAD, the CAIDA AS (Autonomous Systems) data set for normative patterns and possible anomalies. The AS data set represents the topology of the internet as the composition of various Autonomous Systems. Each of the AS units represents routing points through the internet.

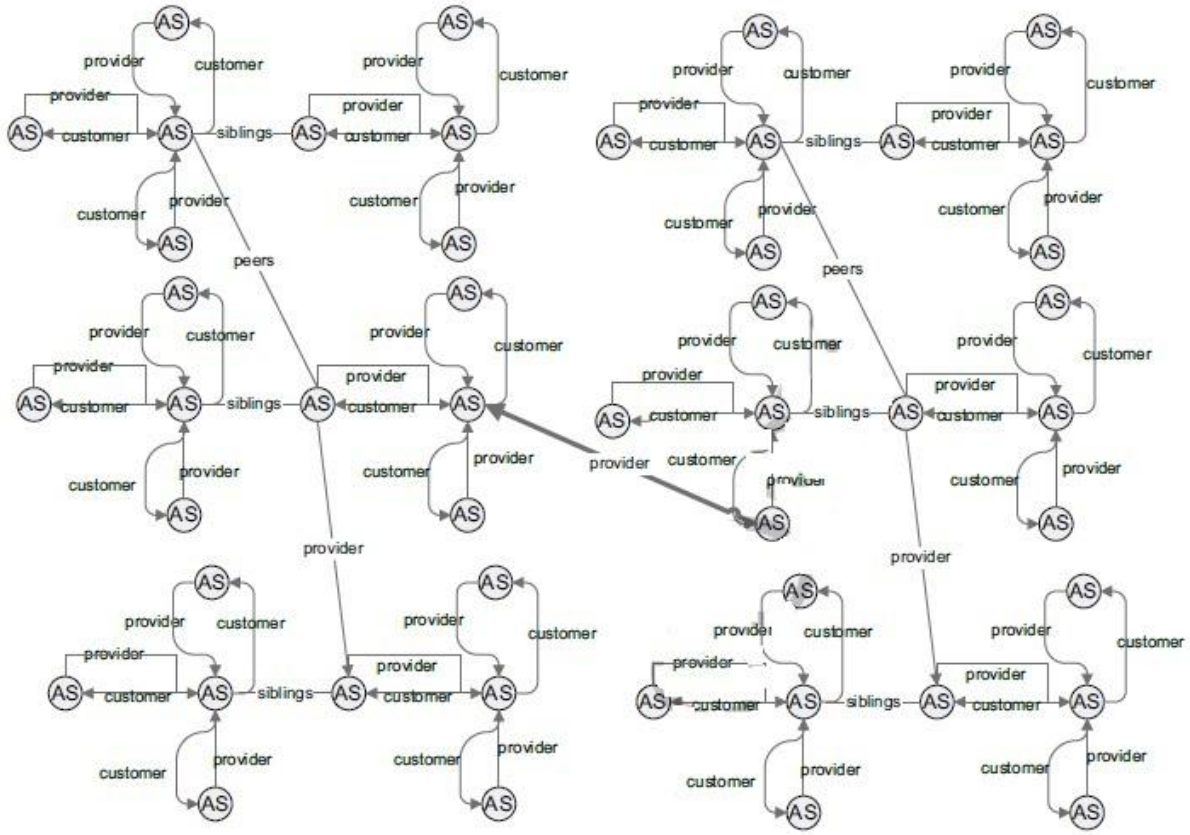

Figure 1: Anomalous pattern discovered in CAIDA dataset

IX. GBAD

For the purposes of analysis, the data was represented as a graph which composes of 24,013 vertices and 98,664 edges, with each AS depicted as a vertex and an edge indicating a peering relationship between the AS nodes. After running GBAD on the AS graph, the anomalous substructure discovered is shown in Fig. 1.

While the data indicates many provider/customer relationships, of which the norm is a particular AS being the provider to three different customers, this single substructure indicates an unusual connection between two AS's. Such an inconspicuous structure would probably be missed by a human analyst, and shows the potential of an approach like GBAD to find these anomalies in network traffic data. However, it took 59,743 seconds to discover the anomalous substructure using a tool like GBAD.

Streaming GBAD (PLADS)

To demonstrate the potential effectiveness of a streaming approach to graph-based anomaly detection, PLADS algorithm was applied to the CAIDA data set as follows.

1. Process N partitions in parallel.

This study arbitrarily chose the initial the first 5 (N) process partitions of the graph running patterns.

2. Determine best normative pattern, P, among NM possibilities.

The partitions' normative patterns was examined, searching for the best normative substructure among them. The result is the normative pattern which is smaller than the normative pattern found when running on the entire graph.

3. Each partition discovers anomalous substructures based upon P.

Based upon the best substructure from among all of the partitions (previous step), search for all anomalous substructures related to that normative pattern was carried out. The result is that 166 substructures are reported as anomalous across all of the partitions, with the longest running partition taking 112 seconds.

4. Evaluate anomalous substructures across partitions and report most anomalous substructure.

All of the reported anomalous substructures across the partitions was examined, and the result showed that 2 substructures are reported as equally anomalous. However, at this point, neither of the substructures are the targeted anomalous substructures. The timing for this step takes less than a second.

5. Process new partition.

The CAIDA data was handled as a stream by removing the oldest partition and processing a new partition. The best substructure was now established on the new partition, so that it can determine the best normative pattern among all of the remaining partitions. The resulted in discovery of the normative pattern in 92 seconds.

Since the normative pattern has changed since the last iteration, each partition re-discover any anomalous substructures based upon the new normative pattern. Examining all of the reported anomalous substructures across the partitions, it was discovered that the most anomalous substructure (found in partition 5) is the one that was identified when GBAD was ran on the entire graph.

At the next iteration (e.g., partition 2 is removed and partition 7 is added), it was discovered in 45 seconds that the normative pattern has not changed (i.e., it is still the best substructure across all of the active partitions). In this case, only the new partition needs to be analyzed for any anomalous substructures, as the anomalies would not change for the already processed partitions. Analysis of the results from the new partition (partition 7) yields (in 58 seconds) no substructures more anomalous than what were already discovered.

Taking this scenario one more iteration (e.g., partition 3 is removed and partition 8 is added), it was discovered that the best normative pattern across all of the partitions is different from the previous iteration. So, similar to two iterations back, all of the active partitions need to be re-evaluated based upon this new best substructure. The result is two new anomalous substructures.

After two more iterations of adding and removing partitions (i.e., processing all of the partitions that represented the single graph), the new normative pattern stays the same, and the anomalousness of reported substructures lessens.

This study is able to implement a graph-based anomaly detection approach on network data that is able to successfully discover the same anomalous substructure within a streaming approach in a fraction of the time (642 seconds) it took to process the entire graph (59,743 seconds). Even the overhead associated with comparing normative patterns and anomalous substructures across partitions is negligible, as the number of substructures to evaluate from each partition is minimal

X. CONCLUSION

This work has provided details of an unsupervised ensemble stream mining based insider threat detection system. The proposed system achieved efficient classification accuracy and showed improvement in the scalability of quantized dictionary attributed to the use of the Pattern Learning Anomaly Detection System (PLADS) and Apache spark's resilient distributed datasets.

ACKNOWLEDGMENT

This work was supported by the Association of African Universities Ph.D. small grants scheme for 2013.

REFERENCES

[1] Information Assurance Technology Analysis Center. (2008). "*The Insider Threat to Information Systems*", State-of-the-Art Report.

[2] Hampton, M., and Levi, M. (1999). Fast spinning into oblivion? Recent developments in money-laundering policies and offshore finance centres, Third World Quarterly 20(3), 645–656.

[3] Salem, M., and Stolfo, S. (2011). Modeling user search behavior for masquerade detection. In Proc. Recent Advances in Intrusion Detection (RAID).

[4] Parveen, P., McDaniel, N., Evans, J., Thuraisingham, B., Hamlen, K., and Khan, L. (2013). Evolving insider threat detection stream mining perspective. International Journal on Artificial Intelligence Tools (World Scientific Publishing) 22 (5), 1360013-1-1360013-24

[5] SANS Institute InfoSec Reading Room (2015). Insider Threats and the Need for Fast and Directed Response

[6] Al-Khateeb, T., Masud, M., Khan, L., and Thuraisingham, B. (2012). Cloud guided stream classification using class-based ensemble. In *IEEE CLOUD*, pp. 694–701

[7] Eberle, W., and Holder, L. (2007). Mining for structural anomalies in graph-based data. In Proc. International Conference on Data Mining (DMIN), pp. 376-389.

[8] Eberle, W., Holder, L., and Graves, J., (2011) Insider threat detection using a graph-based approach, Journal of Applied Security Research 6(1) (January 2011), 32–81.

[9] Nitzberg B. and Lo V., (1991). Distributed shared memory: a survey of issues and algorithms. Computer, 24(8):52 –60.

[10] Kermarrec, A., Cabillic, G., Gefflaut, A., Morin, C., and Puaut, I. (1995). A recoverable distributed shared memory integrating coherence and recoverability. In FTCS '95. IEEE Computer Society.

[11] Dean, J., and Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. Commun. ACM, 51(1):107–113.

[12] Gelernter, D. (1985). Generative communication in linda. ACM Trans. Program. Lang. Syst., 7(1):80–112

[13] Carter, J., Bennett, J., and Zwaenepoel, W. (1991). Implementation and performance of Munin. In SOSP '91, ACM.

[14] Liskov, B., Adya, A., Castro, M., Ghemawat, S., Gruber, R., Maheshwari, U., Myers, A., Day, M., and Shrira, L. (1996). Safe and efficient sharing of persistent objects in thor. In SIGMOD '96, pages 318–329. ACM.

[15] Twister (2015) Iterative MapReduce. Retrieved from http://iterativemapreduce.org

[16] Ekanayake, J., Pallickara, S., and Fox, G. (2008). MapReduce for data intensive scientific analyses. In ESCIENCE '08, pages 277–284, Washington, DC, USA, 2008. IEEE Computer Society.

[17] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P., and Currey, J. (2008). DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In OSDI '08, San Diego, CA.

[18] P´erez, F., and Granger, B. (2007). IPython: a system for interactive scientific computing. Computing. Sci. Eng., 9(3):21–29.