Real-Time Regex Matching With Apache Spark

Sean Deaton[†], David Brownfield[†], Leonard Kosta[†], Zhaozhong Zhu[†] and Suzanne J. Matthews[*]
Department of Electrical Engineering & Computer Science
United States Military Academy, West Point, NY 10996
[*]Corresponding Author, Email: suzanne.matthews@usma.edu
[†]Email: [sean.m.deaton.mil, david.c.brownfield3.mil, leonard.r.kosta2.mil, zhaozhong.zhu.mil]@mail.mil

***Abstract*—Network Monitoring Systems (NMS) are an important part of protecting Army and enterprise networks. As governments and corporations grow, the amount of traffic data collected by NMS grows proportionally. To protect users against emerging threats, it is common practice for organizations to maintain a series of custom regular expression (regex) patterns to run on NMS data. However, the growth of network traffic makes it increasingly difficult for network administrators to perform this process quickly. In this paper, we describe a novel algorithm that leverages Apache Spark to perform regex matching in parallel. We test our approach on a dataset of 31 million Bro HTTP log events and 569 regular expressions provided by the Army Engineer Research & Development Center (ERDC). Our results indicate that we are able to process 1,250 events in 1.047 seconds, meeting the desired definition of real-time.**

Keywords: Regular Expressions, Parallel Computing, Apache Spark, Bro

I. Introduction

Modern malware authors leverage a variety of techniques to evade detection. One notable strategy is to develop custom string generation and mutation algorithms to create the uniform resource identifiers (URIs) in HTTP requests. Thus, Network Monitoring Systems (NMS) like Bro [1] play a critical role in maintaining situational awareness on government and corporate networks. Bro data is widely used for analysis. A strength of Bro is its ability to export binary packet streams to structured log data separated by application layer. Moreover, the use of regular expressions (regexes) or patterns is ubiquitous in the cybersecurity field for network and host-based detection schemes.

Network defenders continually develop sets of regexes in order to match URI mutations. For example, the pattern `.*<?(java|vb)?script>?.*<.+\/script>?` detects references to JavaScript or VBScript files. The computational resources required to process each pattern is directly correlated to the pattern's complexity. Having multiple wildcards (i.e. `.` and `*`) and length restrictors (i.e. `?` and `+`) can increase the resources required for pattern matching, and hinder overall throughput [2].

Additionally, delays exist between when new malware is found and when researchers have extracted the associated patterns. Lastly, while it is possible to extend Bro with a set of custom patterns through the use of Bro scripts, the process is complex and often infeasible on distributed architecture. In the case where multiple instances of Bro are deployed on a network, the scripts will need to be updated on all Bro instances (which can be difficult depending on the size of the network) and may require the Bro instance to be restarted, causing the NMS to potentially miss malicious traffic. Furthermore, there may be a desire to run new patterns on historical data to ensure that the system is not already compromised. To instill such confidence, defenders typically apply the pattern set against all data that passes through the NMS as a post-processing step.

For large organizations with hundreds of thousands of users, a system like Bro can log gigabytes of network traffic daily, making it difficult to scan files quickly for malicious activity. Thus, a parallel system that performs regex matching in real-time is highly desirable. Our definition of real-time is provided by network administrators at the U.S. Army Engineer Research & Development Center (ERDC), who maintain Army networks and sponsor our research. Real-time is defined as the capability to process approximately 1,250 events per second under normal operations, and 2,500 events during peak. Thus, the immediate goal of our research is to design a parallel system that can be integrated into ERDC's workflow. More broadly, our algorithm and results are of direct interest to any organization that wishes to perform regex matching in parallel on network traffic data.

In this paper, we describe a novel parallel algorithm that leverages the Apache Spark [3] framework and MapReduce to perform regex matching at scale on Bro HTTP log data. We test our approach on an ERDC provided dataset of 31 million real events and 569 sample regexes by performing two rounds of experimentation. In the first round, we test our approach on two randomly selected samples of 1,250 and 2,500 events each. In order to better capture the types of events in the original dataset, our second round of experiments covers 25,600 random samples of 1,250 events and 12,205 random samples of 2,500 events. We test our approach on one node of ERDC's Topaz supercomputer in an effort to consume as few resources as possible.

Our first round of experiments show that our MapReduce algorithm can process 1,250 and 2,500 events in 0.385 and 0.427 seconds respectively on 36 cores. We also show that we can process 12,500 events in 0.837 seconds, and 15,000 events in 1.024 seconds. Our second round of experiments indicate that random samples of 1,250 and 2,500 events require 1.047 and 1.591 seconds on average to process. The results demonstrate that our algorithm is capable of meeting ERDC's real-time requirements.

We believe our work is an important first step in increasing

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

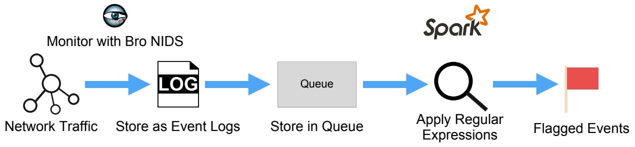
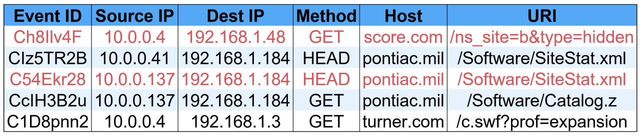


Fig. 1. Anomaly detection workflow.



Event ID	Source IP	Dest IP	Method	Host	URI
Ch8llv4F	10.0.0.4	192.168.1.48	GET	score.com	/ns_site=b&type=hidden
Clz5TR2B	10.0.0.41	192.168.1.184	HEAD	pontiac.mil	/Software/SiteStat.xml
C54Ekr28	10.0.0.137	192.168.1.184	HEAD	pontiac.mil	/Software/SiteStat.xml
CclH3B2u	10.0.0.137	192.168.1.184	GET	pontiac.mil	/Software/Catalog.z
C1D8pnn2	10.0.0.4	192.168.1.3	GET	turner.com	/c.swf?prof=expansion

Fig. 2. Sample log file.

the DOD's ability to perform regex matching at scale on network traffic data. In addition to ERDC, we believe our approach will be of significant interest to network administrators at other organizations who plan to enhance their own workflows with parallel signature detection.

II. Background

Apache Spark [3] is an open source parallel computing framework that is designed to be scalable and easy to use. It provides a platform for implementing parallel algorithms and is widely considered the successor to Apache Hadoop's [4] MapReduce framework. Spark is extremely efficient; for memory-intensive tasks, it is up to 100 times faster than Hadoop [5]. This speedup is primarily due to Spark's use of Resilient Distributed Datasets (RDDs) and lazy evaluation for reducing the cost of operating on large datasets. The framework also supports additional processing paradigms beyond MapReduce. This flexibility and efficiency has prompted several organizations to adopt Spark as their primary cluster computing framework [5].

The concept of utilizing MapReduce frameworks to analyze log data is not entirely novel. Several researchers have utilized Hadoop to try to quickly parse log data. Vernekar *et al.* proposed utilizing a Hadoop MapReduce framework to analyze log files on networks [6]. Events are initially mapped by MAC address. The Reduce Phase uses the MAC address to create separate log files for each machine. Lee *et al.* presented performance analysis on Hadoop analyzing transport-layer traffic on a network [7]. The team proved that Hadoop can be leveraged to analyze terabytes worth of network traffic data.

While Hadoop proves to be a useful framework for log file analysis, using Hadoop often requires transferring log data to a dedicated Hadoop cluster, resulting in significant latency. To remedy this, Logothetis *et al.* proposed a custom MapReduce framework (iMR) to more efficiently process data logs [8].

Other researchers have begun exploring Apache Spark as a way to reduce latency of log file analysis. Karimi *et al.* recently proposed a system leveraging Spark, Spark SQL, and Netmap to detect DDoS attacks in logged data [9]. They tested their framework on the CAIDA dataset [10] and used a six-node Spark cluster to analyze 5 GB of log data in under 3.17 minutes. More recently, Mavridis and Karatza compared the performance of Hadoop and Spark on log file analysis via SQL queries [11]. While Spark consumes more memory than Hadoop, the researchers concluded that Spark is a better MapReduce framework for log file analysis.

Marchal *et al.* [12] compared the performance of several big data frameworks for anomaly detection on 767 MB of network traffic data, including the use of substring matching to detect malicious URIs. Their work concluded that Spark and Apache Shark [13] were the two best performing frameworks for this application. Our work extends the research of Marchal *et al.* by demonstrating the utility of Apache Spark for regex matching on a much larger dataset (5.7 GB).

Shahrivari [14] compared Hadoop's and Spark's performance on regular expressions through Grep [15] on 40 GB of HTML data. While serial Grep required 400 seconds to complete, the Hadoop cluster required only 160 seconds. The Spark cluster finished the task in one second. The work of Shahrivari motivates our work in part, as it suggests that Spark is a superior framework for regex matching at scale.

Böse *et al.* recently proposed a system for real-time anomaly detection in heterogeneous data streams (RADISH) for detecting insider threat [16]. They used Spark on a dataset of 430 million events from the CERT Insider Threat Center at CMU's SEI. Their team was able to process 230 events per second on a single node. Like RADISH, our system also seeks to perform real-time analysis on network traffic data using a single node running Spark. Unlike RADISH, however, our goal is to perform regex matching to determine external threats.

Lastly, we mention the fast uniform resource identifier-specific filter (FARIS) by Takano and Miura [17], a byte-code URL filtering tool. While FARIS is a regex matching tool, it does not leverage Apache Spark or other big data frameworks.

To the best of our knowledge, we are the first to propose a real-time system for regex matching of network traffic log data using Apache Spark. Our unique contributions include: 1.) a MapReduce algorithm for regular expression matching on network log data; 2.) experimentation on a real dataset of 31 million events and 569 regular expressions, both provided by ERDC (big data problem); and 3.) the capability to process $1,250$ network traffic events in approximately one second by leveraging Apache Spark on a single cluster compute node.

III. Our Approach

Currently, Army installations log gigabytes of network traffic each day. Network administrators and security analysts lack the capability to analyze all of the traffic at line speed. In the context of this work, we focus exclusively on Bro HTTP log data. Figure 1 presents an overview of the anomaly detection process. As network traffic data passes through Bro, it is logged for historical record and loaded into a queue. From the queue, the network traffic is analyzed as a series of events, hereafter referred to as a "chunk". These chunk sizes correspond to network traffic targets requested by ERDC at anticipated future "normal" ($1,250$ events) and "peak" ($2,500$

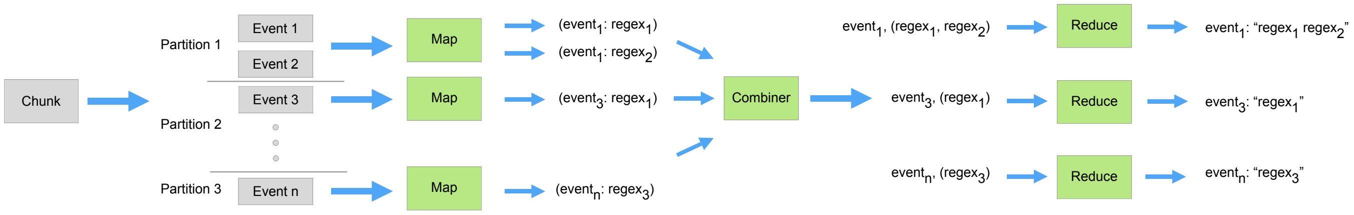


Fig. 3. Overview of MapReduce algorithm.

events) load. To maintain pace with anticipated growths in network traffic, each chunk needs to be processed in approximately one second. On each chunk, we apply our MapReduce algorithm (described in Section III-A), which scans the chunk with the specified set of regexes. If any regex is matched, the associated event is flagged as being suspicious and outputted in JSON format.

ERDC is still in the process of implementing the queue architecture. Therefore we simulate the queue by creating subsets of the Bro HTTP log files provided by ERDC, each representing a separate chunk, and feeding these into Spark's Streaming API. A sample (truncated) log file is shown in Figure 2. The files provided by ERDC are in tab-separated values (TSV) format. Each line in the log file is a single event, which is made up of several fields, including a unique event ID, the origin IP address and port, the destination IP and port, and so forth. The number of fields in the file is dependent on Bro's configuration; in our case, there are 35 total fields per event. In this example, suspicious URIs have been marked in red, indicating that they will be picked up by our approach. In the examples that follow, events 1 and 3 will be labeled as malicious.

Prior to running our MapReduce algorithm, Spark automatically partitions the input log file, and caches the resulting partitions, preparing the data for parallelization. Experiments show that the best performance is derived when the number of partitions is equal to the number of requested cores. Once Spark computes this number, it divides the workload amongst all of the cores on a particular node.

A. Algorithm

Figure 3 depicts our MapReduce algorithm. The input to the algorithm is a.) a log file, representing a single chunk of events, released from the queue; and b.) a list of regexes. In this example, we consider a chunk of n events, 3 regexes, and 3 cores. Events 1, 3, and n will trigger our set of regexes. For simplicity, our figure only depicts event matches. As we discuss below, we inspect matched events further to determine specific matched fields.

In the Map Phase, a chunk is split according the specified number of cores, with each instance of the map function (or mapper) receiving its own partition of events. The map phase processes its set of events and outputs a series of $(key, value)$ pairs where the key is a particular event, and the $value$ indicates the regular expression and field it matched. In our example, each mapper receives $n/3$ events.

Each mapper examines its assigned set of events using the set of regexes. If a regex matches an event, the regex is run on every field of the event to determine a precise match. While we represent a match with regex i in the figure as "$regex_i$" this is in fact a larger structure containing not only the regex, but the set of matching fields.

Returning to our example, event 1 has matches with the first and second regexes in our input list ($regex_1$ and $regex_2$) in mapper 1. This flagged information is passed to the combiner as $(key, value)$ pairs ($event_1$, $regex_1$) and ($event_1$, $regex_2$). In mapper 2, event 3 is matched by the first regex, and is emitted as the $(key, value)$ pair ($event_3$, $regex_1$). In contrast, event 2 is not matched by any of the regular expressions and is ignored. Note that each mapper executes independently and in parallel.

The $(key, value)$ pairs of the flagged events are passed to the Combiner, which aggregates them into $(key, list(value))$ pairs. All regexes that match a particular event are aggregated together in a list. For example, $regex_1$ and $regex_2$ which both matched $event_1$ in the the Map Phase, are combined into the $(key, list(value))$ pair ($event_1$, $list(regex_1, regex_2)$). Note that the Combiner runs concurrently with the Map Phase; however, both the Map Phase and the Combiner must finish executing prior to the start of the Reduce Phase.

In the Reduce Phase, each instance of the reduce function (or reducer) takes a set of $(key, list(value))$ combined mappings and performs a reduction operation on each to construct a final set of $(key, value)$ pairs. In this case, the information associated with each matched regex is converted to a string, and all strings are concatenated together. In our example, reducer 1 receives as input ($event_1$, $list(regex_1, regex_2)$) and processes it to be the final $(key, value)$ pair ($event_1$, "$regex_1, regex_2$"). In contrast, reducer 2 processes $(event_3, list(regex_1))$ to be simply ($event_3$, "$regex_1$"). The $(key, value)$ pairs output from the Reduce phase are output to the user in a JSON dictionary structure, which indicates each flagged event, matched regex and field/URI.

B. Analysis

The run time of our algorithm on a particular chunk is dependent on several variables: the number of events (n), the number of regexes (r), the number of fields associated with each event (f), the worst-case cost (L) to run a regex on an event, and the number of cores (c). In the case that none of the events match the set of regexes, the time required for the

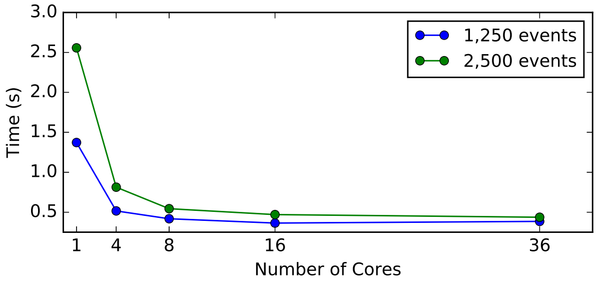


Fig. 4. Processing Time on two randomly selected chunk of $1,250$ and $2,500$ events.

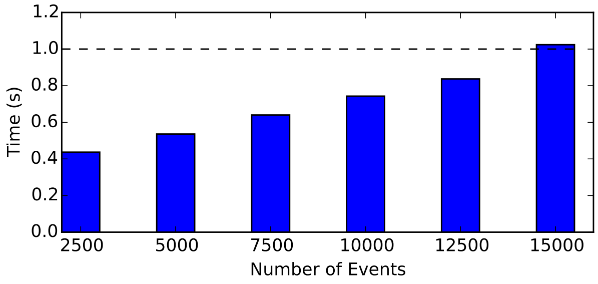


Fig. 5. Chunk size vs. running time.

Map Phase would simply be $O(\frac{n \times r \times L}{c})$. In the worst case, every event matches every regex. For every matched regex, we reapply it to the set of fields for that particular event. In the worst case, all of the fields match the regular expression. Thus, our Map Phase requires at most $O(\frac{n \times r \times f \times L}{c})$ time.

The Reduce Phase is much simpler. Observe that each event has at most $r \times f$ possible matches. Thus, the total number of matches that are passed to the reducer is $n \times r \times f$. It follows that the worst case run time for the Reduce Phase is $O(\frac{n \times r \times f}{c})$, making the total worst case run time of our algorithm $O(\frac{n \times r \times f \times (L+1)}{c})$, or simply $O(\frac{n \times r \times f \times L}{c})$.

Note that the biggest influences on our run time are the number of matching regular expressions, and the cost of the most expensive regex, L. In cases where few regular expressions match the events, the run time will approach $O(\frac{n \times r \times L}{c})$. However, as regular expressions grow complex (higher L) and there are more matches, the run time gets steadily worse, increasing by as much as a factor of f.

We note that there were several key optimizations we applied that are not fully captured in the above analysis. First, we only check the set of fields if a regex matches an event; while this does not improve our worst-case run time, it does improve the average case. Next, all regexes are compiled into regex objects using `re.compile` prior to running our MapReduce approach. This significantly decreases the cost (L) to run a regular expression on a particular event.

IV. Experimental Setup

ERDC provided a dataset consisting of 31 million events (5.7 GB) of HTTP traffic log data collected from Fort Hood, Texas, and a sample of 569 regular expressions. The data was collected over the course of a single day and was broken into log files in TSV format, each spanning fifteen minutes. The size of each log file depends on the amount of traffic within those fifteen minutes. From the data, we estimate that the current rate of network traffic that is collected is between 300 and 400 events per second. Again, our goal is to meet ERDC's anticipated future rate of "real-time" processing, which they define as approximately $1,250$ events per second during normal operations and $2,500$ events per second at peak. Success is measured by the proximity our run times to one second.

We test our approach on a single node of Topaz, a $3,468$-node SGI ICE X supercomputer maintained by the ERDC DOD Supercomputing Research Center. Each node consists of a 36-core Intel Xeon E5-2699v3 Haswell processor at 2.3 GHz, with 117 GB of available RAM. To run our approach on Topaz, jobs were submitted to the cluster using Portable Batch Scripting (PBS) [18].

ERDC requested that we utilize only the 36 cores available on one node to achieve our goal. This is in part due to the large number compute-intensive projects that currently utilize the Topaz cluster, a DOD shared resource. To minimize the impact the regex matching piece has on other projects running on the cluster, it was important that we use as few resources as possible to accomplish our goal. By default, Spark will run on all the cores of a node. To restrict this number for our experiments, we changed parameters in the PBS scripts submitted to Topaz.

Our experimentation is constrained by the set of provided regular expressions and contents of the log files. Topaz was also the only available HPC system that had Apache Spark installed. All together, we consumed roughly $110,000$ hours on the Topaz cluster over the course of this research.

V. Results

We conduct two rounds of experiments to test the efficacy of our approach. In the first set, we measure performance of our algorithm on two random chunks of $1,250$ and $2,500$ events respectively, varying the number of cores from $1 \ldots 36$. From there, we hold the number of cores constant at 36 and increase the number of events in the chunk until we can no longer processes the chunk in less than a second.

In the second round, we move to gather full scale coverage of the Fort Hood log data. In order to accomplish this, we generate several thousand chunks of random events derived from the 31 million event log data. One set contains $25,600$ chunks of $1,250$ random events and the other contains $12,205$ chunks of $2,500$ random events. We compute the average run time and show the percent distribution for each set of chunks.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

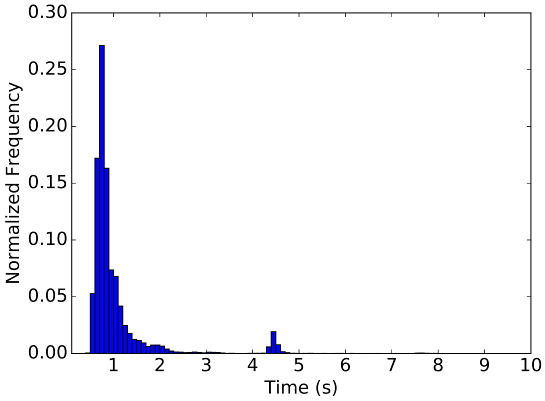


Fig. 6. Normalized times for full-coverage Tests, 1, 250 events.

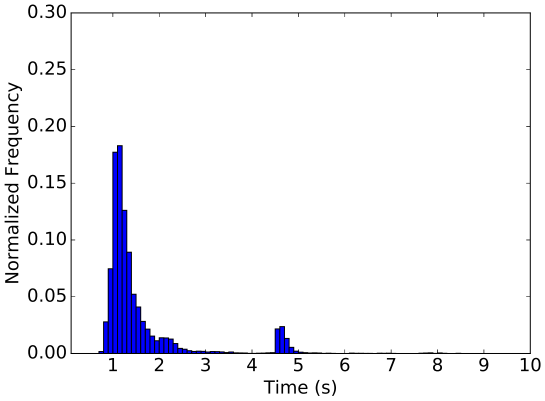


Fig. 7. Normalized times for full-coverage tests, 2, 500 events.

A. *Initial Experimentation*

Figure 4 shows the average time (in seconds) needed to process two randomly selected chunks of 1, 250 and 2, 500 events when we vary the number of cores on one node. On a single core, the algorithm is capable of examining 1, 250 events in 1.372 seconds and 2, 500 events in 2.557 seconds respectively. On 4 cores, the algorithm requires 0.516 seconds and 0.813 seconds to process 1, 250 and 2, 500 events respectively. Our best times are achieved at 36 cores, where 1, 250 events are processed in 0.385 seconds and 2, 500 events are processed in 0.437 seconds. This yields a maximum speedup of 3.77 on 16 cores for the 1, 250 event chunk, and 5.85 on 36 cores for the 2, 500 event chunk.

While we acknowledge that the speedup of the approach leaves much to be desired, we emphasize that our goal for this work was to process chunks of events as quickly as possible and under one second on a single 36-core node, which we are able to do. Motivated by these results, we conduct all further experiments on a single Topaz node, utilizing 36 cores.

Next, we examine how well our algorithm scales as ERDC's data processing requirements grow. In other words, how many events can we process in 1 second? Figure 5 shows the results of our next set of experiments. We create chunks containing between 2, 500 . . . 15, 000 events in increments of 2, 500. For each chunk, we measured the time it takes our algorithm to perform regex matching on 36 cores. The algorithm appears to perform well as the chunk sizes increase. 10, 000 events are processed in 0.743 seconds. 12, 500 events are processed in 0.837 seconds. At 15, 000 events, we surpass the 1 second barrier, requiring 1.024 seconds.

B. *Expanded Experimentation*

As noted in Section III-A, the run time of our method is dependent on the number of matches and the complexity of the regular expressions. Since we are constrained by the set of regexes given to us by ERDC, we decided to expand on the number of chunks we test in order to capture as diverse a selection of events as possible. Each chunk in this set of experiments is composed of randomly selected events (with replacement). Thus, we select 12, 505 chunks of 2, 500 events and 25, 600 chunks of 1, 250 events from the original dataset. For each chunk, we measure the run time required to process it and use binning to capture its frequency. In Figure 6 and Figure 7, each bin represents an interval of 100 milliseconds.

Figure 6 depicts the results of the expanded experimentation on chunks of 1, 250 events. Our results indicate that 73.3% of the chunks take less than a second to complete. There were extreme outliers in the results, with processing times ranging from as little as 0.46 seconds to a maximum of 8 seconds. The average processing time is 1.047 seconds, meeting the stated definition of real-time.

Figure 7 depicts the expanded experimentation using the chunks of 2, 500 events. The average number of results that fell under one second decreases sharply to 5.0%. However, 94.0% of the chunks take less than two seconds to complete. The minimum time and maximum time to process a random chunk increases to 0.72 seconds and 10.69 seconds respectively. The average time required to process 2, 500 events is 1.591 seconds.

The discrepancy between the times in our initial and expanded sets of experiments corresponds with the number of regex matches in each. In the first set of experiments, the number of matches were on the order of a thousand per chunk. In the second set of experiments, however, some chunks had tens of thousands of matches. These results support our analysis that that the run time of our approach is heavily dependent on the number of regex matches in any particular chunk.

VI. Conclusions

In this paper, we described a novel parallel algorithm that leverages Apache Spark for parallel regex matching. The immediate goal of our project was to design a system for the U.S. Army ERDC, which requested the ability to process events at a rate of 1, 250 per second to meet anticipated growth of data at normal operations, and 2, 500 events per second during anticipated peak. ERDC provided us with a dataset of 31 million real events and 569 regular expressions. Our

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

results show that our parallel approach is able to process $1,250$ events in 1.047 seconds and $2,500$ events in 1.591 seconds on average on 36 cores. However, depending on the composition of events and the set of regular expressions, the system is capable of processing up to $15,000$ events in a second.

Our results suggest that our system is capable of meeting the needs of a large organization like the U.S. Army in order to perform regex matching on network traffic at scale. To this end, our results are an important first step to enable the DOD to develop this capability on their networks. We anticipate that our results will also be of interest to other organizations performing regex matching on network traffic data.

We have several recommendations for administrators hoping to implement our system at their organizations. Since the run time of our approach is dependent heavily on the number of regex matches that occur, extreme care should be taken when generating a set of regexes for input. In our specific case, the extent of our analysis was limited by the set of regexes provided to us by ERDC. Some of the regexes we received were redundant, matching the same or very similar strings to other regexes in the set. The generality of some of the provided regexes also led us to believe that there were many false positives (i.e. URIs flagged as suspicious when they were not). Given the expense of regex computations when compared to other pattern matching techniques, less expensive key-word matching should first be used to flag events; only then should one use expensive regexes. We believe this strategy will increase throughput.

Furthermore, if information is provided about a particular regex (i.e., it only matches IP addresses), then we can accelerate our approach by targeting only those specific fields in an event. Our current implementation checks entire events and each event field on a match. Future work can evaluate the potential speed-up of this optimization and extend our approach on other types of application event data (e.g. DNS).

In addition, we would like to explore the effect of parallelizing regexes rather than events. Since regexes can be challenging to load balance effectively, we theorize the First Fit Decreasing Algorithm, an algorithm used to solve Bin Packing problems, could be used to evenly distribute the regexes to each computing core. Extensive testing will need to be performed to validate this hypothesis; we also believe that performance will be heavily dependent on the set of regexes and available architecture at a particular organization. We also hope to cross validate our approach against actual malicious traffic, and perhaps augment our system with machine learning and Mlib [19]. Lastly, we acknowledge both the application and research potential of FPGAs and ASICS for regex matching [2], and hope to explore it as another avenue of future research.

Acknowledgments

This paper summarizes the results of an undergraduate capstone project at the U.S. Military Academy. Funding for this project was provided by the DOD High Performance Computing Modernization Program (HPCMP) and the Army Engineer Research & Development Center (ERDC). We are also extremely grateful to ERDC and HPCMP for providing access to the DOD Topaz system, and providing us experimental data for testing. Special thanks to Dr. Leslie Leonard, Dr. Ben Parsons, and Mr. William Glodek of ERDC for troubleshooting assistance. We are also grateful to Dr. Chris Okasaki and MAJ Benjamin Klimkowski of the U.S. Military Academy for insightful comments and feedback on this research. The opinions in this work are solely of the authors and do not necessarily reflect those of the U.S. Military Academy, the U.S. Army, or the Department of Defense.

References

[1] V. Paxson, “Bro: a system for detecting network intruders in real-time,” *Computer networks*, vol. 31, no. 23, pp. 2435–2463, 1999.

[2] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. H. Katz, “Fast and memory-efficient regular expression matching for deep packet inspection,” in *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*. IEEE, 2006, pp. 93–102.

[3] Apache Foundation, “Apache Spark: Lightning-fast cluster computing,” Internet Website, 2013. [Online]. Available: http://spark.apache.org/

[4] T. White, *Hadoop: The definitive guide*. ” O’Reilly Media, Inc.”, 2012.

[5] Lightning-fast cluster computing. [Online]. Available: http://spark.apache.org

[6] S. S. Vernekar and A. Buchade, “Mapreduce based log file analysis for system threats and problem identification,” in *2013 3rd IEEE International Advance Computing Conference (IACC)*, Feb 2013, pp. 831–835.

[7] Y. Lee and Y. Lee, “Toward scalable internet traffic measurement and analysis with Hadoop,” *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 1, pp. 5–13, 2013.

[8] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum, “In-situ mapreduce for log processing,” in *2011 USENIX Annual Technical Conference (USENIX ATC11)*, 2011, p. 115.

[9] A. M. Karimi, Q. Niyaz, W. Sun, A. Y. Javaid, and V. K. Devabhaktuni, “Distributed network traffic feature extraction for a real-time ids,” in *2016 IEEE International Conference on Electro Information Technology (EIT)*, May 2016, pp. 0522–0526.

[10] The cooperative analysis for internet data analysis. [Online]. Available: http://www.caida.org

[11] I. Mavridis and H. Karatza, “Performance evaluation of cloud-based log file analysis with Apache Hadoop and Apache Spark,” *Journal of Systems and Software*, vol. 125, pp. 133 – 151, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121216302370

[12] S. Marchal, X. Jiang, R. State, and T. Engel, “A big data architecture for large scale security monitoring,” in *2014 IEEE International Congress on Big Data*, June 2014, pp. 56–63.

[13] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, “Shark: fast data analysis using coarse-grained distributed memory,” in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 689–692.

[14] S. Shahrivari, “Beyond batch processing: Towards real-time and streaming big data,” *Computers*, vol. 3, no. 4, pp. 117–129, 2014. [Online]. Available: http://www.mdpi.com/2073-431X/3/4/117

[15] M. Crochemore and W. Rytter, *Text algorithms*. Oxford University Press, 1994.

[16] B. Bse, B. Avasarala, S. Tirthapura, Y. Y. Chung, and D. Steiner, “Detecting insider threats using RADISH: A system for real-time anomaly detection in heterogeneous data streams,” *IEEE Systems Journal*, vol. PP, no. 99, pp. 1–12, 2017.

[17] Y. Takano and R. Miura, “Faris: Fast and memory-efficient url filter by domain specific machine,” in *2016 6th International Conference on IT Convergence and Security (ICITCS)*, Sept 2016, pp. 1–7.

[18] R. L. Henderson, “Job scheduling under the portable batch system,” in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1995, pp. 279–294.

[19] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, “MLlib: Machine learning in Apache Spark,” *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.