Parallel and Distributed Normalization of Security Events for Instant Attack Analysis

David Jaeger, Andrey Sapegin, Martin Ussath, Feng Cheng, Christoph Meinel
Hasso Plattner Institute (HPI)
University of Potsdam, 14482, Potsdam, Germany
{david.jaeger, andrey.sapegin, martin.ussath, feng.cheng, meinel}@hpi.de

***Abstract*—When looking at media reports nowadays, major security breaches of big companies and governments seem to be a normal situation. An important step for the investigation or even prevention of these breaches is to normalize and analyze security-related log events from various systems in the target network. However, the number of log events produced in big IT landscapes can grow up to multiple billions per day. Current log management solutions, e.g., Security Information and Event Management (SIEM), cannot even closely normalize such huge amounts of data and therefore disable the tracking of attacks in real-time, which means that the log data remains unusable for attack analysis. In this paper, we present an approach to fully normalize event logs in high-speed by making use of established high-performance inter-thread messaging in conjunction with a hierarchical knowledge-base of log formats and parallel processing on multiple low-end systems. Using our approach, we are able to process more than 250,000 events/sec on relatively low-profile machines and can therefore easily handle more than 20 billion events/day, which is enough to handle average and peek loads of log events from big enterprise networks.**

***Keywords*-network security; event logs; normalization; inter-thread messaging**

I. Introduction

The number and complexity of cyber-attacks has dramatically increased over recent years [1]. Media reports about breached companies and governments can be seen almost on a weekly basis. Whenever such major cyber-attacks happen, security investigators start to analyze security relevant event logs that were produced by the targeted network environment during period of the attacks. Generally, event logs are a valuable source of information and allow understanding and reproducing performed malicious activities. However, there are two major challenges when it comes to the analysis of these event logs. Firstly, event logs are usually not stored in a common format. This means that before investigators can work with these logs, they need to transform them into a common representation. This process is also called event log normalization. The second challenge is the huge amount of log data produced by sensors in networks of big companies and governments. Surveys performed by the SANS Institute [2] show that big enterprises have often more than 10GB of log data per day. Gartner defines a quantity of 25,000 events/sec, which is beyond 2 billion events/day, as large deployment in their Magic Quadrant report [3]. Hewlett-Packard (HP) mentions[4] an amount of 100 billion up to 1 trillion events/day in their infrastructure. HP itself can only handle 3 billion events/day from these. It is obvious that this amount of data can neither be handled manually nor normalized with common SIEMs in reasonable time. Even worse, peeks of incoming events going far beyond the mentioned 25,000 events/sec cannot be handled, but would be very important for finding attacks that cause such peeks. As a result of this situation, analysis of security breaches can take weeks or even months. In fact, the detection of attacks from log data in real-time is not even imaginable at the moment.

In the following of this paper, we present an approach to fully normalize huge quantities of events, i.e. far beyond 25,000 events/sec, by utilizing an established method for lock-free inter-thread communication and parallelizing normalization on multiple low-end systems. With full log normalization, we refer to the extraction of all available information from a log line. In addition to a conceptual description of our approach, we integrate the proposed high-performance normalization into our prototypical Real-Time Event Analysis and Monitoring System (REAMS)[1] SIEM solution. We have structured the paper as follows. Section II first gives a short overview of related work to event normalization and its parallelization. The following Section III gives a short introduction into the basics of event normalization. Then, the concept of high-speed inter-thread communication and its optimized use for event normalization is presented in Section IV and V. Section VI focuses on how normalization speed can be even more increased by using a multi-node architecture. In the end, Section VII concludes our work and gives an outlook to further research.

II. Related Work

A. Event Log Normalization

The normalization of logs relies on the transformation of logs in different formats to a common format and can

[1]REAMS - https://sec.hpi.de/reams

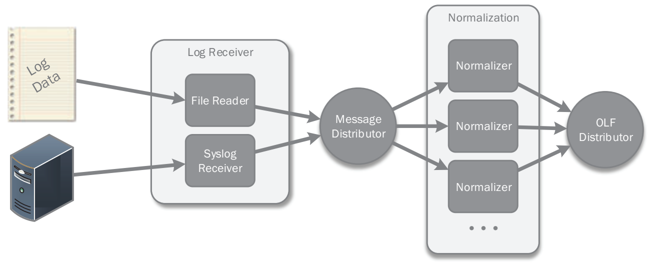


Fig. 1. Normalization Process for Event Logs

be performed with a variety of normalization mechanisms. In our previous papers, we have worked out an overview of different log formats [5], [6] and presented different techniques for the normalization of logs. We have started with normalization using Named-Group Regular Expressions (NGREs) [7], [8] and have presented different indexing mechanisms like Lucene[9] or a hierarchical knowledge base [5]. Because many existing common log formats are either not well structured, such as Common Event Format (CEF), or lack in support of important information fields, like Syslog or Common Event Expression (CEE), we have proposed the object-oriented Object Log Format (OLF) to overcome these limitations in the paper [6].

B. Inter-Thread Communication and Parallel Processing

When processing events with a high throughput using multi-threading, there are limitations on the distribution of tasks to the different threads. A major problem with traditional inter-thread communication is the locking of threads while waiting for new tasks to come in, which is typical for blocking queue implementations. LMAX came up with a concept [10], called LMAX Disruptor, which is supposed to be lock-free and was developed with very high throughput in mind. The Logstash tool by Elastic[2] seems to already employ parts of this disruptor concept, but it is not clear how they use it and whether it is used for event normalization.

Apache Hadoop[3] is currently a famous tool for parallel processing. Bhatt et al. present a framework for processing large amounts of logs via the Hadoop platform in their paper [11]. However, it is unclear whether the framework was implemented and whether there are performance benefits in comparison to existing approaches. Additionally, there are no details given on how deep normalization was performed. Another such system that integrates log collection and analysis into Hadoop has been presented by Han et al. [12] and Therdphapiyanak et al. [13]. Han et al. focus on the persistence of events and Therdphapiyanak et al. focus on clusterization of data for attack analysis. In the end, both solutions do not seem to support full log normalization for arbitrary formats. A concrete solution with Hadoop log management can be found by PetaPylon in form of their *Log Management* solution[4].

We are not convinced about the suitability of the specialized *MapReduce-algorithm* of Hadoop for log normalization. It seems to work for a variety of corner cases for log analysis, but does not seem to work for a comprehensive analysis. We therefore focus on a customized solution that better fits the requirements of log normalization. At the moment, it seems to be more promising to try Hadoop's successor, i.e. Apache Spark[5].

III. Event Normalization

In order to understand the presented improvements in event normalization, we first want to give a short overview of the whole process of normalization and where parallelization can be utilized. Figure 1 pictures the workflow of normalization. At the beginning of the workflow a log file or one or multiple servers produce logs. These logs are collected at a central place by the *Log Receiver* component.

When the log data is received, it is split up into separate log events, which are then forwarded to the *Normalization* component. Within the *Normalization* component, multiple workers then take over the processing of the logs and normalize it into OLF. After the normalization, all logs are forwarded for further processing to the *OLF Distributor*.

From the overview, it is obvious that both distributors are a bottleneck in the processing. Usually, the log receiving can be handled by a few workers, because there is no heavy processing to do. The normalization, however, is very processing intensive and needs as many resources as possible and therefore relies on multi-threading. A major challenge we face in the normalization is to transfer data structures from multiple producers (Log Receiver) to multiple consumers (Normalization). An approach to this challenge is discussed in Section IV.

One *Normalizer* performs normalization with the help of a hierarchical knowledge base. The concept for this kind of normalization has been discussed in the paper [5] and has proven to be highly efficient. The main idea of normalization is to use NGREs to match a log line and then use named groups to extract the properties for the normalized OLF event. Listing 1 shows an exemplary log line from an SSH server (SSHd) that needs to be normalized.

Listing 1. Log event as produced by SSHd

```
Sep 1 08:37:12 target-server sshd: Accepted password for
    john from 192.168.1.1 port 47246 ssh2
```

This concrete event is wrapped into the common Syslog [14] format, which is usually used in UNIX-based operating systems for log exchange. The message part of the Syslog event encapsulates the more concrete SSHd

[2]Elastic Logstash - https://www.elastic.co/products/logstash
[3]Apache Hadoop - https://hadoop.apache.org/

[4]PetaPylon Log Management - http://petapylon.com/solutions/log-normalization/
[5]Apache Spark - http://spark.apache.org/

event information. Our idea with a hierarchical knowledge base is to make use of the typical structuring of logs into wrapper- and sub-formats. We first try to find the wrapper format, which would be Syslog in this case. Listing 2 shows the corresponding NGRE for matching Syslog.

Listing 2. NGRE for Syslog

```
^(?<time>\w{3} \d+ \d+:\d+:\d+) (?<producer_host>\S+) (?<
    producer_appname>\S+): (?<msg>.*)$
```

From this matched regular expression, the `msg` is extracted and then tried to be matched against possible wrapped formats. Listing 3 represents the regular expression that would be matched in this case.

Listing 3. NGRE for SSHd part

```
^Accepted password for (?<user_username>\S+) from (?<
    network_sourceIpv4>\S+) port (?<network_srcPort>\d+)
    (?<application_proto>\S+)$
```

In this paper, we have proposed a method to efficiently find the wrapping and wrapped formats. Once all information is extracted from the NGREs, it is filled into an OLF event, as shown in Listing 4.

IV. Normalization with High-Performance Inter-Thread Communication

Figure 1 shows that normalization is highly dependent on the exchange of event data through the distributors. Taking traditional programming models, this problem of exchange is usually solved with so called blocking queues.

Listing 4. Extracted fields of event from Listing 1 in OLF format

```
{ time: "2015-09-01T08:37:12",
  network: { srcIpv4: "192.168.1.1", srcPort: 47246},
  producer: { host: "target-server", appname: "sshd"},
  user: { username: "john"},
  application: { proto: "ssh2"} }
```

A. Blocking Queue Approach

A blocking queue is a thread safe queue implementation that allows a producer thread to put elements into the queue, while one or multiple consumers wait on the queue for incoming events. Figure 2 shows the event processing architecture based on blocking queues.

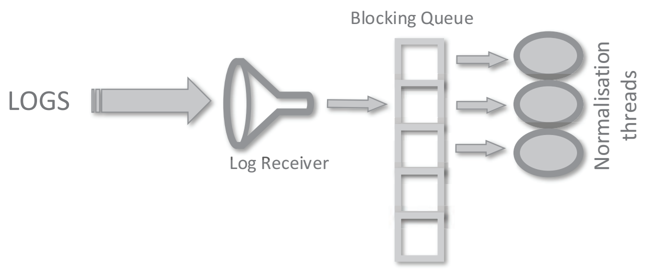


Fig. 2. Event processing approach using blocking queue

In case the queue is empty, the thread blocks and wakes up as soon as the producer puts in a new element. If there are already elements in the queue, the consumer directly gets one of the elements without blocking. While this model is sufficient for exchanging sporadically incoming elements, it does not perform for a high number of incoming elements, because many CPU time is spent for blocking.

We have used blocking queues for our implementation of normalization, so far. The results for this normalization can be found in the evaluation part of the paper [5]. On average, we could reach around 37,000 events/sec with 8 threads[6] , which is already remarkably high, but can be improved.

B. Disruptor Pattern Approach

To increase normalization speed, the so called disruptor pattern[15] can be used, as depicted in Figure 3.

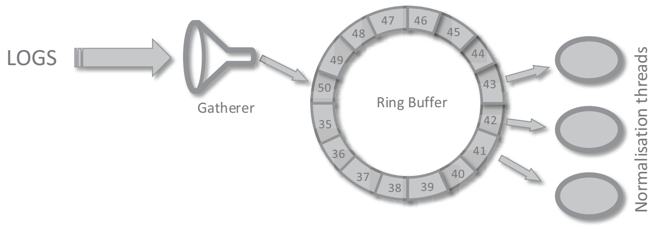
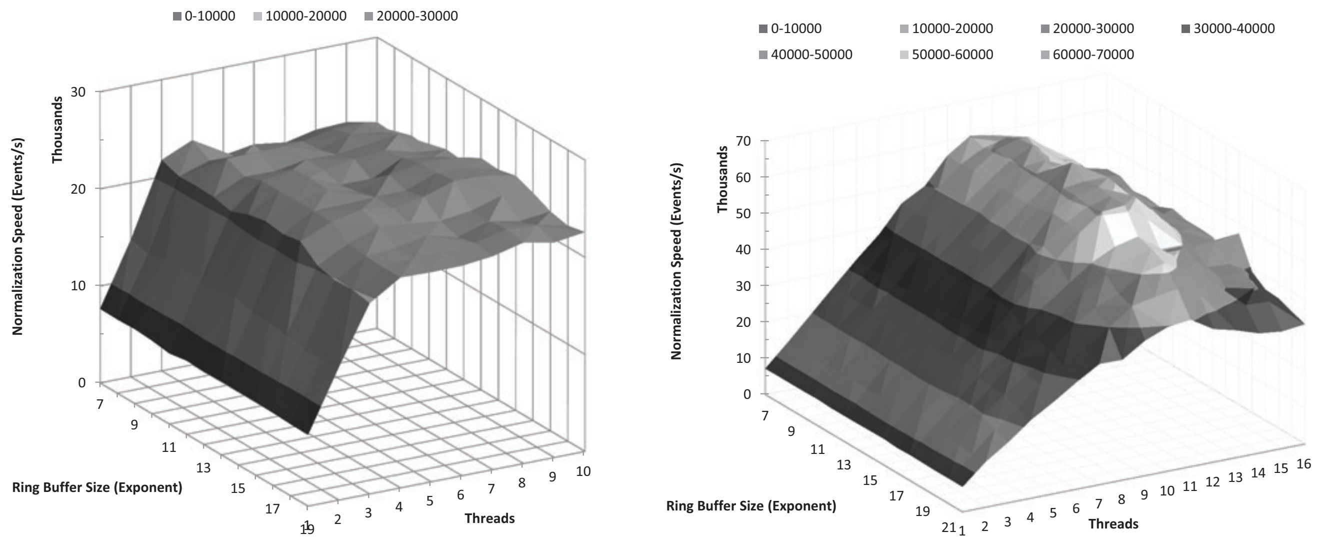


Fig. 3. Event processing approach using blocking queue

The disruptor pattern is based on the exchange of elements through a so called ring buffer. This is a circular buffer where events at the beginning are overwritten when there are more than the buffer's size elements. This structure is efficient, because old elements do not have to be explicitly discarded and no new memory has to be assigned. The disruptor concept helps to prevent locking of threads. In comparison to a blocking queue, new elements are inserted into a free position of the ring buffer and get a sequence assigned to it. For our concrete scenario, the Log Receiver (producer) inserts an element into the buffer and the normalization threads (consumers) store their sequences pointing to the event being processed. All normalization threads can read the sequences of their peers to identify, which events in the ring buffer are available for processing. Therefore, normalization threads could retrieve events from the buffer without a risk of being locked by each other. Since the event is not removed from the buffer during retrieval, there is no need to lock the buffer itself.

Because of the promising performance gain, we have implemented the disruptor pattern in our prototypical REAMS SIEM to achieve an even higher event throughput. However, to achieve best performance, we have to find the optimal configuration of the disruptor's ring buffer size and the number of producers and consumers. According

[6]Virtual Machine (Debian 7.8, 32GB RAM(dedicated), 16 cores(dedicated)) on VMware ESXi host with 256GB RAM and 8x Intel Xeon X7560 CPUs @ 2.27GHz



(a) 4 Core Machine (b) 16 Core Machine

Fig. 4. Normalization with disruptor using different number of threads and ring buffer sizes

to the documentation of the disruptor, the ring buffer size should be a power of 2.

C. Finding an Optimal Disruptor Configuration

To find the optimal configuration, different combinations of ring buffer sizes and normalization threads that consume events from the ring buffer have to be tested. Using the machine from the blocking queue implementation of our normalization, we have 16 cores that could theoretically normalize events in parallel.

We created a test setup, where we normalize 10 million Apache Web Server log lines with different sizes of the ring buffer and different number of normalization threads. We tested a range of buffer size exponents from 7 (128 slots) to 22 (4,194,304 slots) and a thread count from 1 to the number of cores. Because we use the same machine for testing as with the blocking queue implementation, we have a maximum of 16 threads. Additionally, we created a second virtual machine with 4 cores[7] for testing. We performed two tests for each combination of buffer size and thread count and calculated the average of the maximum normalization throughput. Figure 4 shows our results for all tests we performed.

The diagram shows that the number of normalized events per second almost grows linearly up to a certain number of threads and then turns over. For 16 cores, this is with 11 threads, for 4 core it is with 4 threads, respectively. Beginning from this number, the throughput is declining. We would have expected a maximum throughput at around 14-16 normalization threads for 16 cores or 2-3 threads for 4 cores, because in this case each core would exclusively normalize events within one thread. We assumed a number being 2 less than the core number, because 2 additional threads are needed to perform our performance measurement and run the Log Receiver, which reads the log file from disk. We are sure that the reading of logs is not a bottleneck, because we were able to read logs at around 400,000 events/sec with one thread in a previous test. We will show in the next section, why this drop of performance is happening and how it can be eliminated. The size of the

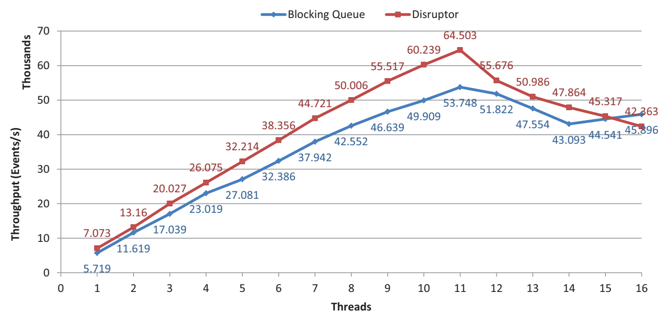


Fig. 5. Comparison of normalization performance with disruptor (buffer size 2^{13}) and blocking queue on a 16-core machine

ring buffer has a rather limited impact on the performance for lower exponents. We see that the performance increases slightly with the buffer size. However, at a size of 18, the performance dramatically decreases. On average, the best throughput is achieved with an exponent of around 13 (8192 slots) to 16 (65,536 slots). We assume that our observed optimal buffer sizes are related to the available memory in our virtual machine and slightly depend on the number of consumers that work on the buffer.

Altogether, we could reach the highest throughput of 64,503 with 11 threads and a buffer size of 13. A comparison of the disruptor performance with the blocking queue

[7]Virtual Machine (Debian 7.8, 4GB RAM(dedicated), 4 cores(dedicated)) on previous VMware ESXi host

performance is shown in Figure 5. We performed both tests with the same log data set with 10 million Apache Web Server events.

From this diagram, it can be deduced that the disruptor performance is generally higher than the blocking queue performance. Both approaches have almost continuously increasing throughput up to 11 threads. At the point of the highest difference, the disruptor has a 10,755 events/sec higher throughput, which estimates to around 20%.

V. Achieving a Lock-Free Implementation

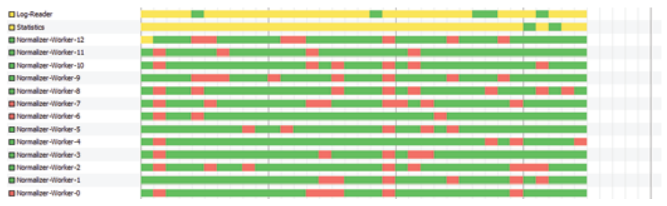


Fig. 6. Locking during normalization with multiple threads (green - thread running, red - thread blocking, yellow - thread waiting)

Already, a performance gain of 20% could be reached by reimplementing our Normalizers to use the disruptor pattern instead of the blocking queue. However, we saw a performance drop at 11 threads. While looking for possible reasons for this behavior, we encountered repeated locking of the normalization threads, although the disruptor is supposed to be lock-free. A screenshot in Figure 6 shows these interruptions in red color in between of the green running state.

By taking stack-traces of the locking threads, we found that one of our used libraries, called Apache commons-beanutils[8] for introspection, is causing locks. We were able to pinpoint and eliminate the calls to these blocking functions and could immediately encounter huge performance improvements for the normalization threads. Figure 7 shows the performance of this now completely lock-free implementation of the normalization in comparison to the previous implementation.

The lock-free implementation allows us to scale up to a throughput of 145,365 events/sec, which is more as double (+125%) the speed as the best throughput of our previous implementation. Additionally, there is now a much more stable growth than before, because now there is no lock contention blocking the processing. The previous limit of 11 threads, which we could even observe on a machine with 24 cores, was because of the locking becoming too inherent. Now, the throughput should grow linearly with the number of threads.

As outlined in Figure 7, the throughput is now even growing up to the number of 16 threads, which is the number of cores in the machine. This is even better than we expected, because the Log Receiver and statistics do not seem to noticeably influence the performance. The throughput seems to remain almost constant for thread numbers beyond 16.

[8]Apache commons-beanutils - http://commons.apache.org/proper/commons-beanutils/

VI. Distributing Event Processing

So far, we have focused on the optimization of the throughput on a single machine. To achieve an even higher throughput, multiple machines can be used to process events in parallel. A single machine cannot be easily scaled without investing a lot of money. It would be cheaper to rather use smaller low-end systems than a single big machine. Therefore, we will now show how log normalization can be performed with a distributed normalization system.

A. Architecture Overview

Figure 8 gives an idea on how normalization can be distributed.

The main idea is to have a central node that is able to receive logs from various sources. We consider this central point important, because all log sources can be configured to point to one node. For example, various existing SIEMs can be pointed to this node's IP for forwarding. Once events are received, they are immediately forwarded to one of multiple normalization nodes. Each of these normalization nodes only has limited processing resources, but can still contribute a part for a higher throughput. All in all, multiple such low-end normalization nodes can together achieve a higher throughput than a single node.

B. Implementation of Network Communication

For the distribution of events to other nodes, the network communication has the highest influence on the overall performance of the processing. While events can be theoretically read at a rate of more than 700,000 events/sec with parallel file readers, our first implementation with Java's simple Socket API could only reach a transmission rate of around 5000 events/sec per connection. This indicates, that even with more than 10 nodes, not more than 50,000 events/sec can be normalized because of a limitation in the transmission speed. Research by Welsh and Culler [16] from 2000 confirms this bad performance of Java's Socket API. They show that native access to C sockets can achieve significantly higher speeds. As a result of this shortcoming, the Java Community came up with an extended library for scalable I/O in 2002, the so called Java NIO specification (JSR51) [17], which was further improved in JSR201 [18]. In this library, the developer works with fast buffers that are directly used to communicate between program and I/O-code.

By changing our original implementation to the Java NIO library, we could immediately achieve a throughput of up to 50,000 events/sec without tuning socket parameters like send/receive buffers. In comparison to the standard Socket API, this enables us to scale our throughput with normalization nodes.

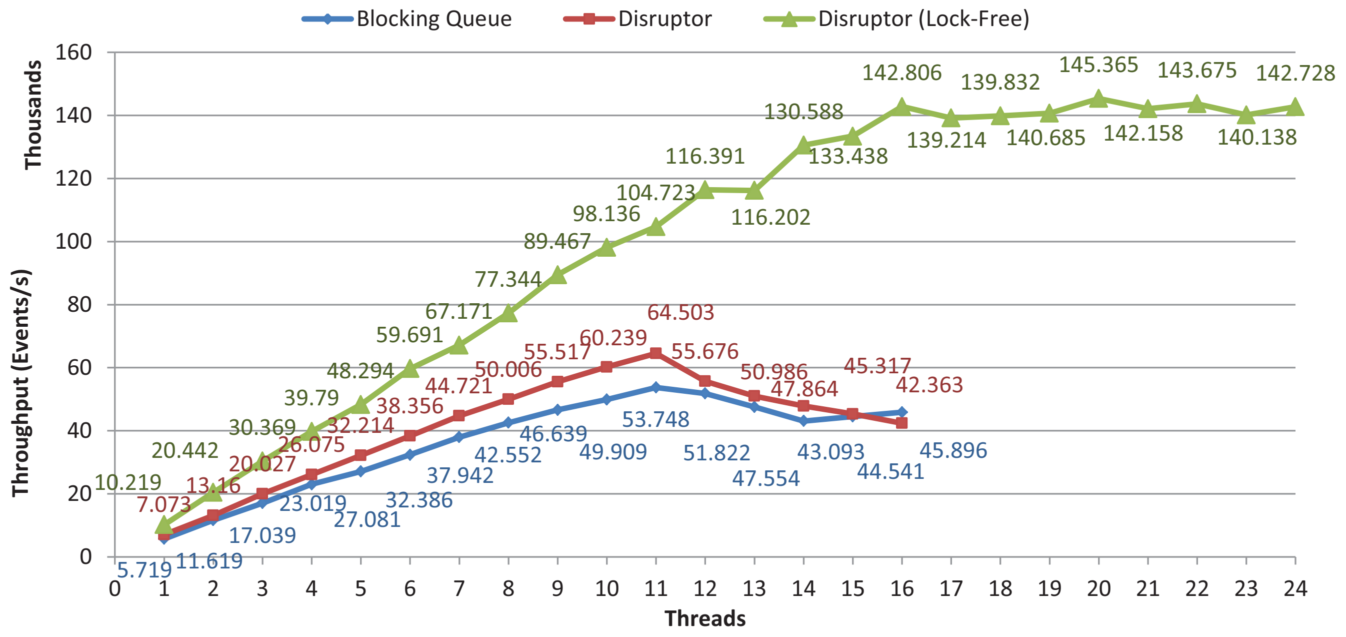


Fig. 7. Performance of fully lock-free normalization threads with 16 cores

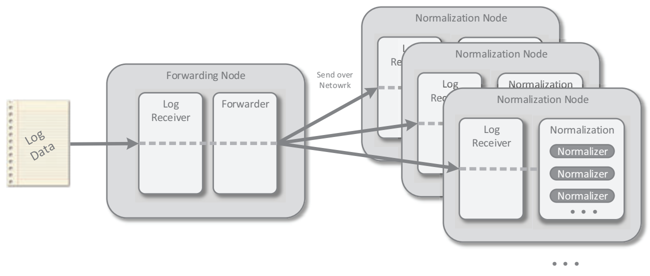


Fig. 8. Distributing the normalization

C. *Efficiently Distributing Events to Nodes*

One challenge when distributing events to multiple nodes is to account for node utilization. The main question is: How can we ensure that each node has an equal share of the load? Only when all nodes are equally busy, the best results for throughput can be achieved. Our first naïve approach to solve this issue is by forwarding one event from the first to the last node and then starting over from the beginning. However, while this pattern ensures the same amount of events for each node, it does not ensure the same workload, because usually events are not equally difficult to normalize. Our second approach is based on the busyness of the nodes, since it takes the fullness of the ring buffer as an indicator for the events that can be processed. If a node can process its events faster, then its buffer will get emptier soon and it will receive more events. The fullness of the buffer can be easily encountered when inserting elements into the buffer, because in the case of a full buffer the inserting thread is blocked until a new slot in the buffer becomes empty. If this inserting thread is at the same time used to receive new events from the network stack, then as long as the thread has space in the buffer, it can receive more events. If it is blocked, then it cannot process the events from the network stack. One of the properties of the TCP protocol is a sliding window of transmitted data, where only further data is sent when the data from the window is received in the application. This means as long as the receiver does not actively take the data from the network, no further data will be sent from the sender. In reality, the sender would be blocked from sending further data. Figure 9 depicts the concept of this transitive blocking.

The forwarding node runs one thread for each connection to a normalization node. This thread takes read events from the ring buffer and then sends it over the network, given that there is no blocking. As soon the ring buffer is full on its corresponding normalization node, the log receiver will block on the ring buffer and therefore will also block the sending of further data over the connection. As soon the ring buffer has space, the Log Receiver will continue and can then take events from the Log Sender. Using this mechanism, each node only gets just as many events as it can process.

We think that this approach of directly using the limiting capabilities of TCP allows immediate reaction to the fullness of the underlying buffer on the normalization nodes. An approach where the receiver would give feedback messages on the fullness of its buffer would probably only deliver outdated messages.

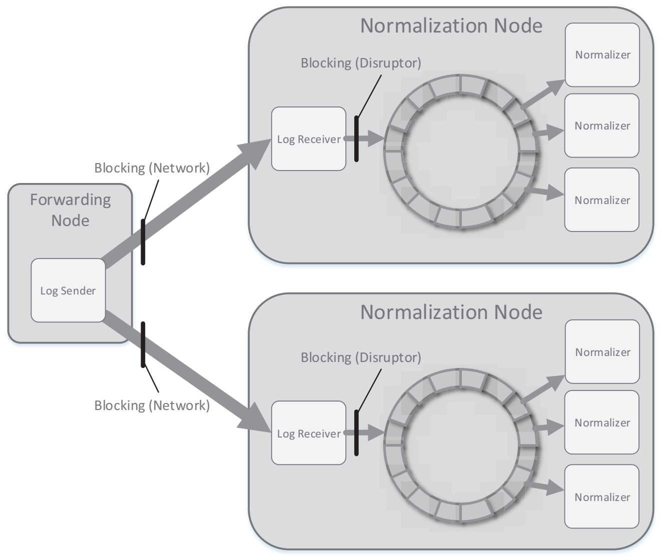


Fig. 9. Transitive blocking to control node utilization

D. *Forwarding Based on Event Properties*

Another way of forwarding messages, different to the plain distribution of messages, is to decide the normalizing node based on its preferences. Similar to the mapping in Hadoop, wrapper for more complex messages could be pre-normalized and then it can be decided from the event properties, where to forward events to. A normalization node can then be specialized in the normalization of certain events, such as Web logs or Intrusion Detection System (IDS) logs, and then could even perform faster for its specialized events. This approach has not been implemented into our REAMS, but promises improvements for logs from heterogeneous sources.

E. *Performance Tests*

We have tested the distribution with a different number of normalization nodes, each having 4 cores and 4GB of main memory. According to our previous performance tests, these machines are best set up with 4 normalization threads and a buffer size of 2^{13}. 3 simultaneous connections are used for each node to transmit raw event logs with the highest speed. Our previous machine with 16 cores was used as the forwarding node. Figure 10 shows the results of our tests.

The diagram makes clear that normalization scales almost linearly with the number of normalization nodes. Each node accounts for around 25,000 − 30,000 events/sec, totaling to around 265,000 events/sec for 10 parallel nodes. With 6 nodes, together having 24 cores, we can already achieve a higher throughput (+18,000 events/sec) than the 16 core machine alone. Thus, a multi-node architecture can outperform a single node, assuming that a significantly larger system is not affordable and multiple low-end systems are at hand.

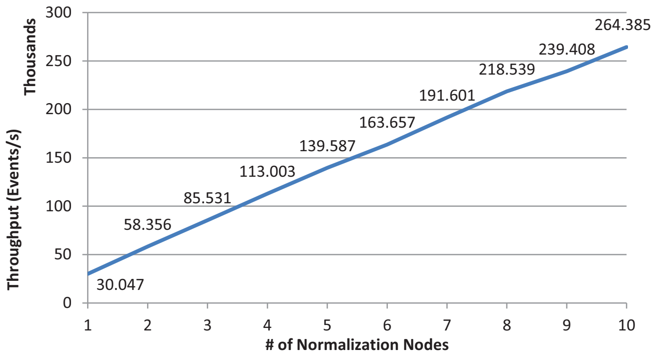


Fig. 10. Performance of distributed normalization with varying number of normalization nodes

Figure 11 shows a screenshot of the performance measurement results for 8 parallel nodes. The first part shows a histogram summary of 32 measurement points after each 30s. The second part is a meter with the current measurement point.

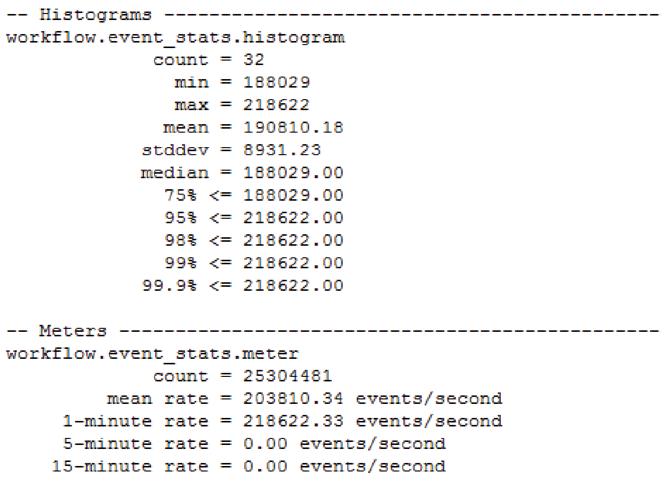
While testing the 10 node configuration, we observed a processor utilization of 300-400% (Full utilization of 3-4 cores) on the forwarding node, indicating that there is still enough space to further scale the throughput. Assuming a linear growth of the throughput with more nodes and a 10 Gb/s Ethernet connection, a theoretical throughput of 1 million events/sec would be imaginable with 40 parallel normalization nodes.

Considering the log quantity of 1 trillion events/day from the HP [4] paper, a throughput of around 11.5 million events/sec would have to be achieved. Taking the fact that HP is able to afford even more expensive machines than what we have at hand, this amount of logs could be handled with 12 independent setups of 20 parallel nodes with 8 cores. These setups could be placed on different continents and different departments of the HP infrastructure.

VII. Conclusion and Future Work

The amount of security-related log data that is relevant for investigating cyber-attacks is rising. Existing tools in the SIEM-domain were not able to handle and especially fully normalize these amounts of events before. In this paper, we have shown how multiple billions of events/day, being typical for big enterprise networks, can be normalized in real-time, making an immediate attack analysis, e.g. via an in-memory database, possible. While Gartner categorizes a throughput of more than 25,000 events/sec as large setup, this still cannot account for peeks in incoming log events for big enterprises. However, especially such peeks can indicate an malicious activity going on. With our results, we can even handle the normalization of security logs in such occasions.

Our performance comparisons have shown, that a single node can already achieve a normalization throughput of



```
-- Histograms ------------------------------------------
workflow.event_stats.histogram
             count = 32
               min = 188029
               max = 218622
              mean = 190810.18
            stddev = 8931.23
            median = 188029.00
              75% <= 188029.00
              95% <= 218622.00
              98% <= 218622.00
              99% <= 218622.00
            99.9% <= 218622.00

-- Meters ----------------------------------------------
workflow.event_stats.meter
             count = 25304481
         mean rate = 203810.34 events/second
     1-minute rate = 218622.33 events/second
     5-minute rate = 0.00 events/second
    15-minute rate = 0.00 events/second
```

Fig. 11. Screenshot of our application while running the test for 8 nodes

up to 145,000 events/sec with a lock-free disruptor implementation. This is a major improvement to our previously achieved throughput of 37,000 events/sec. The distribution of normalization on 10 relatively low-profile nodes can even increase the throughput to 265,000 events/sec, which is already a magnitude more than what is set as a boundary for large environments by Gartner. We even assume that more nodes could scale the normalization speed to a million events per second.

For future research, the processing of events in batches is a promising way to achieve even higher throughput. Various sources show that the disruptor pattern scales with such a batch size. Nevertheless, the normalization overhead of the processing has to be regarded, since it sets the upper bound for the maximum possible throughput.

Another direction of research could be the parallelization with Apache Spark. This large-scale data processing framework is a successor of Hadoop and promises 100x the speed of Hadoop while removing the dependency to the limiting MapReduce algorithm.

One problem that we did not cover so far in our approach is further processing of events, such as anomaly or misuse detection or even the persistence of the events. Researching the possibility of distributed analysis seems promising and has been partly covered in research work already.

VIII. Acknowledgement

We would like to thank the HPI Future SoC Lab[9] for providing us with the latest and powerful computing resources, which make the testing and experiments in the paper possible. Additionally, we would like to thank Marian Gawron for his support and ideas in further improving the speed of our implementation.

[9]HPI Future SoC Lab - http://hpi.de/en/research/future-soc-lab.html

References

[1] US Office of Management and Budget (OMB), "Annual Report to Congress: Federal Information Security Management Act," US Office of Management and Budget, Tech. Rep., Feb. 2015.

[2] J. Shenk, "Ninth Log Management Survey Report," SANS Institute, Tech. Rep., Oct. 2014.

[3] K. M. Kavanagh and O. Rochford, "Magic Quadrant for Security Information and Event Management," Gartner, Tech. Rep., Jul. 2015.

[4] S. Bhatt, P. K. Manadhata, , and L. Zomlot, "The Operational Role of Security Information and Event Management Systems," *IEEE Security & Privacy*, vol. 12, pp. 35–41, Oct. 2014.

[5] D. Jaeger, A. Azodi, F. Cheng, and C. Meinel, "Normalizing Security Events with a Hierarchical Knowledge Base," in *Proceedings of the 9th International Conference on Information Security Theory and Practice (WISTP'15)*, ser. Lecture Notes in Computer Science, R. Akram and S. Jajodia, Eds., vol. 9311, no. 1. Springer International Publishing, 2015, pp. 238–248.

[6] A. Sapegin, D. Jaeger, A. Azodi, M. Gawron, F. Cheng, and C. Meinel, "Hierarchical Object Log Format for Normalisation of Security Events," in *Proceedings of the 9th International Conference on Information Assurance and Security (IAS'13)*, Yassmine Hammamet, Tunisia, Dec. 2013, pp. 25–30.

[7] A. Azodi, D. Jaeger, F. Cheng, and C. Meinel, "Pushing the Limits in Event Normalisation to Improve Attack Detection in IDS/SIEM Systems," in *Proceedings of the First Internation Conference on Advanced Cloud and Big Data (CBD'13)*, Nanjing, China, Dec. 2013.

[8] J. E. F. Friedl, *Mastering Regular Expressions*, 3rd ed., A. Oram, Ed. O'Reilly Media, Aug. 2006.

[9] A. Azodi, D. Jaeger, F. Cheng, and C. Meinel, "A New Approach to Building a Multi-Tier Direct Access Knowledge Base For IDS/SIEM Systems," in *Proceedings of the 11th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC'13)*, Chengdu, China, Dec. 2013.

[10] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart, "Disruptor: High Performance Alternative to Bounded Queues for Exchanging Data Between Concurrent Threads," LMAX, Tech. Rep., May 2011.

[11] P. Bhatt, E. T. Yano, and P. Gustavsson, "Towards a Framework to Detect Multi-stage Advanced Persistent Threats Attacks," in *Proceedings of the 8th IEEE International Symposium on Service Oriented System Engineering*. IEEE Computer Society, 2014, pp. 390–395.

[12] S. Han, M. Kim, and H. Lee, "Design and Implementation of a MongoDB-based Log Processing System in Cloud Computing Environment," in *Proceedings of the 4th International Conference on Internet (ICONI'12)*, 2012.

[13] J. Therdphapiyanak and K. Piromsopa, "Applying Hadoop for Log Analysis toward Distributed IDS," in *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication (ICUIMC '13)*, no. 3, 2013.

[14] R. Gerhards, "The Syslog Protocol," RFC 5424 (Proposed Standard), Internet Engineering Task Force, Mar. 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5424.txt

[15] M. Fowler. (2011, Jul.) The LMAX Architecture. Web Site. [Online]. Available: http://martinfowler.com/articles/lmax.html

[16] M. Welsh and D. Culler, "Jaguar: Enabling Efficient Communication and I/O in Java," *Concurrency: Practice and Experience*, vol. 12, no. 7, pp. 519–538, 2000.

[17] M. Reinhold. (2002, May) JSR 51: New I/O APIs for the Java Platform. Sun Microsystems, Inc. [Online]. Available: https://www.jcp.org/en/jsr/detail?id=51

[18] A. Bateman. (2011, Jul.) JSR 203: More New I/O APIs for the Java Platform ("NIO.2"). Oracle. [Online]. Available: https://jcp.org/en/jsr/detail?id=203