Security event data collection and analysis in large corporate networks

E V Chernova[1], P N Polezhaev[1], A E Shukhman[1], Yu A Ushakov[1], I P Bolodurina[1] and N F Bakhareva[2]

[1]Orenburg State University, Pobedy av., 13, Orenburg, Russia, 460018
[2]Povolzhskiy State University of Telecommunications and Informatics, L. Tolstoy str., 23, Samara, 443010

e-mail: newblackpit@mail.ru, shukhman@gmail.com

Abstract. Every year computer networks become more complex, which directly affects the provision of a high level of information security. Different commercial services, critical systems, and information resources prevailing in such networks are profitable targets for terrorists, cyber-spies, and criminals. The consequences range from the theft of strategic, highly valued intellectual property and direct financial losses to significant damages to a brand and customer trust. Attackers have the advantage in complex computer networks – it is easier to hide their tracks. The detection and identification of security incidents are the most important and difficult tasks. It is required to detect security incidents as soon as possible, to analyze and respond to them correctly, so as not to complicate the work of the enterprise computer network. The difficulty is that different event sources offer different data formats or can duplicate events. In addition, some events do not indicate any problems on their own, but their sequence may indicate the presence of a security incident. All collection processes of security events must be performed in real-time, which means streaming data processing.

CEUR-WS.org/Vol-2416/paper31.pdf

1. Introduction

Recently, computer networks tend to develop rapidly. They become larger and more complex, but they still remain profitable targets for various intruders – criminals, cyber-spies, and even terrorists. Commercial services, critical systems, and information resources are at risk. The consequences can be different, ranging from the theft of strategically important information, highly estimated intellectual property, and direct financial losses to significant damages to a brand and customer trust.

The traditional approach to cybersecurity is based on the idea that it is necessary to create a special trustful environment for networks and data, that is, to organize them in such a way as to reduce access to them from the outside, but not to prevent them from performing their functions correctly. This will help to discover and eliminate vulnerabilities before the intruder will find them. Such an approach is no longer effective in modern computer networks with constantly changing threat scenarios. Attacks can be organized anytime and anywhere, and due to the complexity of networks, it is easier for the attackers to hide their tracks. In theory, the specialists should be ready for all possible variants of attacks, but, in practice, it is impossible. Thus, to protect the systems, it is necessary to collect the data from the entire network, understand how it works, detect and identify threats, and take appropriate

actions as fast as possible. Certainly, there is a huge amount of solutions to perform these tasks, but not all of them are free, able to interact with the most of available sources of security events and to work in the highly distributed networks.

A Network-wide Cyber Situational Awareness (NwCSA) has been introduced in [1] to assist a network security administrator with network security. The challenges include the overload of raw data, low speed of reaction, and a lack of context and unified view on a network. The framework leverages a distributed data stream processing system and methods for real-time big data processing. The paper describes only the concept of such system.

A cyber threat platform for real-time detection and visualization of cyber threats OwlSight is presented in [2]. The platform is composed by several building blocks and it is able to collect huge amounts of data from multiple sources, prepare and analyze the data and present the findings through a set of insightful dashboards. The platform use Cassandra for data storage and Spark for data processing. Authors describe some real usage scenarios, but do not provide results of performance testing of solutions.

In [3] authors introduce an architecture dedicated to security monitoring of network traffic in local enterprise networks. The application domain of the system is network intrusion detection and prevention. Other anomalies are not considered. This architecture integrates two systems, one dedicated to scalable distributed data storage and management and the other dedicated to data exploitation. Several well-known big data framework are compared for data processing. Spark and Shark appear to be the best performers in all tests.

The paper [4] presents a prototype of Security Information and Event Management (SIEM). The system uses a combination of three different approaches for security analysis: misuse detection, query-based analytics and anomaly detection. Authors propose to use anomaly detection in a combination with signatures and queries, applied on the same data, rather than as a full replacement for misuse detection. In this case, the majority of attacks will be captured with misuse detection, while anomaly detection will highlight previously unknown behaviour or attacks. The main drawback of the system is the use of expensive in-memory data storage (SAP HANA). Also anomaly detection methods are tested on obsolete the KDD 1999 dataset.

In [5] authors compare several methods for detecting anomalies on UNSW-NB15 dataset. They test correlation analysis, linear discriminant analysis and seven well known classification algorithms within the bigdata tool Apache Spark. Data collection methods are not covered in the paper.

In this paper, we describe the concept of a system for collection and primary analysis of security events and incidents in large corporate networks.

2. Selection and review of software tools

The main requirements for software tools are the open source, the ability to function in the distributed systems, and the support of streaming data processing. Since a large amount of heterogeneous data is formed in real-time, which is required to be converted to a general form, processed and analyzed, it should be considered that the selected tools must support the work with streaming data. Streaming data mean the data that continuously provided by a variety of sources, from which small batches are formed. The sources of such data are authentication systems, active network devices, IDS/IPS (Intrusion Detection Systems/Intrusion Prevention Systems), event logs of servers, antiviruses, vulnerability scanners, and other security and management systems.

Apache Spark [5] was selected as a framework for processing streaming data. It works much faster than *Hadoop*, supports cluster mode, and it is compatible with other Apache products. It has well-structured documentation, and it is quite popular among developers, which means there are a lot of articles, tutorials, and manuals about it. *Spark* can work in a *Hadoop* environment managed by *YARN*. *Spark* provides API for *Scala*, *Java*, *Python*, *R* languages and supports different distributed storage systems – *HDFS*, *Cassandra*, *OpenStack Swift*, *NoSQL-DBMS*, *Amazon S3*, and others. In addition, *Spark* includes the following components: *Spark SQL* for processing SQL queries, *Spark Streaming* for streaming data processing, *Spark MLib* for machine learning, and *Spark GraphX* for working with graphs.

Apache Spark has an extension to its basic API – *Spark Streaming*. It does not process entire streams, but divides them into small batches. It is called *DStream* (Discretized Stream), which is a sequence of RDDs (Resilient Distributed Datasets). They can be processed in parallel, including computations based on sliding windows. RDD supports two types of operations: transformations and actions. The result of a transformation is a new RDD; the result of an action is a specific value. Transformations are not executed immediately, that is, *Spark* remembers transformations over particular data and executes them, only when an action is called (lazy execution). Such an approach improves the efficiency of *Spark*. In addition to RDD, there is a *DataSet* in *Spark*. It differs by using the optimizer that chooses the most efficient way of computing the result. The optimizer is mainly used for SQL queries.

A *Spark* application represents a set of processes executed in a cluster and controlled by *SparkContext* object, which called a driver, created in the main program [4]. In particular, *SparkContext* can connect to cluster dispatchers of different types that distribute resources between applications. After connection, *Spark* gets available executors on the cluster nodes and sends the code to the executors, and, finally, *SparkContext* assigns tasks to the executors to perform them.

In addition, *Spark* has disadvantages. Firstly, it is the "inheritance" of batch processing – non-constant network loads during data loading and processing. Therefore, it is necessary to set up restrictions on the density of input stream, since *Spark* does not have an efficient tool for tracking it. Secondly, it cannot recover clusters after failures.

Additionally, the *Apache Kafka framework* can be used to solve these problems. It is able to create real-time data pipelines and streaming applications [6]. *Kafka* provides the designing of a distributed server for message queues. Thus, a data stream is distributed over several servers in a cluster providing high scalability. The risk of data loss is reduced due to such replicated and persistent storage.

The main abstraction of *Kafka* is a topic that is a category or feed name used to publish records to it. Topics usually have several subscribers that are consumers, which subscribe to the data written to the topics. *Kafka* provides a partitioned log for each topic. Each partition represents an ordered, immutable sequence of records, which are constantly added to a commit log. Each record in the partition has a unique sequential number that is called offset. It identifies each record in the partition.

The servers of *Kafka* clusters store the distributed commit logs of the partitions. Each server processes requests for shared access to partitions. In order to provide fault-tolerance, partitions are replicated to a configurable number of servers. A partition has a single server that is called leader and zero or more servers which function as followers. Leader processes all requests for reading and writing partitions, and followers passively replicate the leader. One of the followers will be elected as a new leader if the current leader fails. Each *Kafka* server functions as a leader for some of its own partitions, and as a follower for others, therefore, the load in the cluster is well balanced.

Kafka Streams is a client library for parallel processing and analyzing the input and output data stored in *Kafka*. It can be used for computations described as a processing topology.

3. Organization of data transfer and storage

After performing different operations, it is necessary to store the obtained results. *Apache Cassandra* [7] was selected for that as a NoSQL database management system. *Cassandra* has high scalability and throughput for reading and writing operations, and it supports replications. It does not work with SQL, but it has a similar language called CQL (*Cassandra Query Language*). A big advantage of *Cassandra* is the ability to create reliable and fault-tolerant clusters. In addition, joining new *Cassandra* instances to the cluster is very simple. Clients can access any cluster node for reading and writing because all of them are equal, and data are consistent.

Raw data enter the system from different sources like network devices, antivirus programs, firewalls, IDS/IPS. For receiving data from such sources, there is *Kafka Connect* [8], which is a tool for scalable data transfer between *Apache Kafka* and other systems. It can be used to transfer large volumes of data to or from *Kafka*. *Kafka Connect* can get the entire databases or collect data from different application servers and put them into *Kafka* partitions. In addition, the export jobs can transfer data from *Kafka* partitions to the external storages and query systems, or to the batch systems

for offline analysis. Since the sources of data are different, for each of them, the separate *Kafka* partition should be created.

Connector developed by *DataMountaineer* simplifies writing data to a *Cassandra* database [9]. It converts values from *Kafka Connect SinkRecords* to JSON, and then asynchronously inserts records into *Cassandra* tables. The connector can create secure connections over SSL. The selection of fields and partition management are handled by KCQL (*Kafka Connect Query Language*).

The *Spark Cassandra Connector* library [10] was selected to implement a connection between *Spark* and *Cassandra*. For data transferring, it is necessary to create *StreamingContext* and *DStream* that will connect to nodes. The *SparkConf* object is used to configure *StreamingContext*. It is necessary to set up the connection host, user name, and password for a *Cassandra* user. Any node of a *Cassandra* cluster can be a connection host. The driver extracts cluster topology from the connection host and can switch to the closest node in the same data center. Whenever any method is called that requires access to *Cassandra*, the options from the *SparkConf* object are used to create a new connection or to use an already opened one from the pool of connections. In addition, the *Spark Cassandra Connector* can connect to several *Cassandra* clusters.

4. Correlation analysis

Events are consumed by application based on *Spark Streaming*. They have their own place in a hierarchy, depending on the way of receiving events and their features (Figure 1). There are three large classes of events: the events from SNMP, the events from *Syslog*, and others. Classes are divided into subclasses, depending on the common features of events. Subclasses, in turn, can be divided again, etc.

The *Spark Streaming* Application takes each received *Kafka* partition, performs filtering, converts the data into classes of the hierarchy and saves them, after that a correlation analysis is performed.

At the next step, it is necessary to create rules for correlation analysis. There are many ways to organize them. In our system, we use two of them – patterns and requests. A pattern is a set of rules with some fields, which are filled in with the data from events. A request is similar to a pattern, but it works with data streams. It can join, group by, filter, deduplicate, sort data, or pass events through sliding windows. The window saves events to aggregate, join, and match them against particular patterns or subqueries. It defines which subset of events should be saved. For example, the data window saves the last *N* events, and the time window – the events during the last *N* seconds. In addition, in a pattern or a request, the reaction can be defined, that is, the action which should be performed when the data match the conditions. The simplest reaction is a notification of an administrator about suspicious activity, for example, by an e-mail message. However, there are more radical opportunities such as port closing, launching programs or running scripts (for example, to shape the traffic during DDoS attacks).

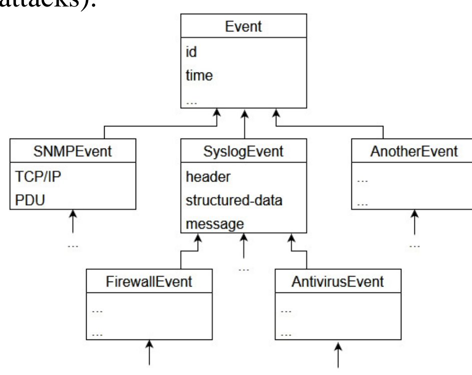

Figure 1. The hierarchy of events.

The pattern fields can be different, for example, such as the event category, the IP address of its source, the type of source device, the port number, the time thresholds, etc. The selection of the fields required for the analysis is based on available data. For example, an SNMP protocol data unit includes IP and UDP headers, protocol version, password, type of unit, request id, error status, error index, and

variable bindings. A *Syslog* message consists of a header, structured-data, and a text. The header contains information on facility, severity, protocol version of *Syslog*, timestamp, host name, application name, system process id, and message type. Knowing how the messages of these protocols are formed, it is possible to extract necessary fields using regular expressions. For each event type, a special regular expression is used, which captures the necessary data from the raw message of the event source. The following method creates a new event.

```
create event ExampleEvent: Event {
severity:  " severitygroup",
datetime:  " datetimegroup",
hostname:  " hostnamegroup",
appname:  " appnamegroup",
type:  " typegroup",
message: " messagegroup",
}
```

Here double quotes indicate the names of groups from the regular expression.

In addition to the usual method of event creation, there are more intended to the events of certain types. It is for the convenience of the users.

```
create event SyslogEvent: Event {
facility:  " facilitygroup",
severity:  " severitygroup",
datetime:  " datetimegroup",
hostname:  " hostnamegroup",
appname:  " appnamegroup",
type:  " typegroup",
message: " messagegroup"
}
```

```
create event SNMPEvent: Event {
destination_address:"destination_addressgroup",
source_address:  " source_addressgroup",
type_of_service:  " type_of_servicegroup",
sourse_port:  " sourse_portgroup",
destination_port:  " destination_portgroup",
pdu-type:  " pdu-typegroup",
error-status:  " error-statusgroup"
}
```

The rules look similar to SQL queries. Firstly, the keyword "*select*" is used in the begging of the rule. Then the selection from the required group of events is specified after the keyword "*from*". According to SQL principles, "*where*" can be used to specify a condition. All logical operations ("*and*", "*or*", "*not*") should be supported. In addition, it is possible to specify the time duration within the events should happen, or the time interval between them ("*timer*"). Then, it is possible to use grouping ("*groupby*") or sorting ("*orderby*"). After that, the reaction is specified ("*then*" with "*msg*" or "*block*").

The following example is the selection of the alert time, the host IP address, the severity and the category of all incidents from the pattern, except for those which source IP address is "192.168.100" or "192.168.1.101", grouped by source IP address:

```
select alert_time, host_ip, severity, category from pattern
[pattern eventA=antivirus -> eventB=scanning_hosts (eventA.src_ip = eventB.host_ip)
where timer within 60]
where src_ip not  "192.168.1.100" or  not "192.168.1.101"
groupby src_ip.
```

The pattern is specified in square brackets. It starts with the keyword "*pattern*", then two events ("*eventA*" and "*eventB*") are specified with their types after the equal sign "=". The sign "->" indicates that the events are sequential. Further, the fields of these events are specified in parentheses. In this example, "*src_ip*" of "*eventA*" should be equal to "*host_ip*" of "*eventB*". This condition should be satisfied within 60 seconds, so the events will be identified as an incident. In addition, it is possible to use a previously saved pattern by specifying its name in the square brackets. The Web interface of the system should has the opportunities for editing and adding correlation rules.

5. Concept of a system for collection and primary analysis of security events and incidents in large corporate networks

Firstly, it is necessary to collect as much data as possible about the security events in computer networks. As it was already mentioned above, data are collected using the *Syslog* and SNMP

protocols. Information about other events that they do not cover can be collected using special tools, for example, IDS Suricata. All data are converted into JSON format. Tools used for data transmission support this format.

The SNMP agents collect information on devices and send it to the SNMP server (Figure 2). In addition, there are two ways for transmission of the collected information: the request-response and the trap. In the second case, the agent unilaterally sends messages to the server.

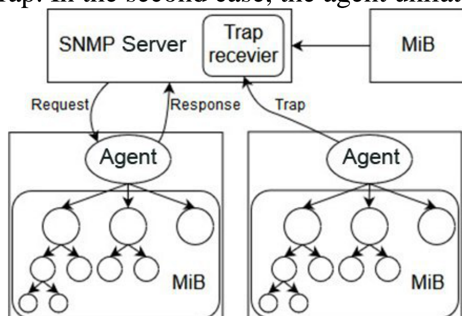

Figure 2. The interaction between the SNMP server and the agents.

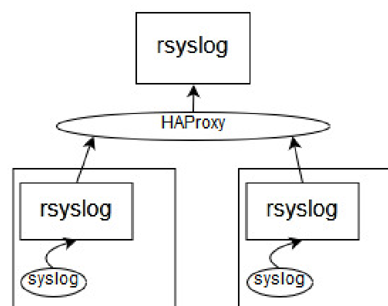

Figure 3. The *Syslog* data collection using *rsyslog* servers.

The data obtained using the *Syslog* protocol are collected by *rsyslog* servers (Figure 3). *HaProxy* is used for load balancing and availability. In order for the rsyslog server to transmit data to the *Kafka* servers, it is necessary to install the special plugin *omkafka* and to change the configuration file "*rsyslog.conf*" by adding the template for converting messages from *Syslog* to JSON format [11].

Information from the sources collected by *Suricata* is transmitted to the database using *Kafka* [12] (Figure 4). For each type of event source, *Kafka* servers should have the separate partition, which processes events of a particular source in parallel with other partitions (in this case, "*sur-topic*" for Suricata, "*rsl-topic*" for *Syslog*, and "*snmp-topic*" for SNMP). All received raw events in JSON format are stored in the database, from which they are extracted by the *Spark* application. It converts, filters, and aggregates events, after that it performs the correlation analysis. The correlation rules are loaded from separate storage. All the results are saved in a separate database. If incidents are detected, the reaction module, according to the rules, alerts the administrator or blocks the malware activity on network devices or hosts. The Web interface of the system should provide many opportunities for users including authorization, generation and viewing of reports, editing and adding correlation rules.

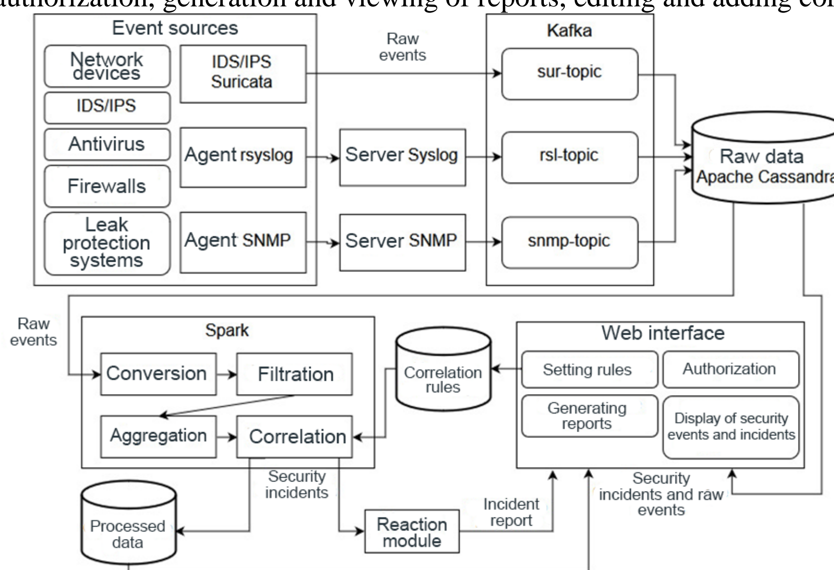

Figure 4. The architecture of the system for the collection and primary analysis of security events and incidents in the large corporate networks.

6. Experiments

The test bed was assembled to check the efficiency of the proposed approach and measure its performance parameters. Figure 5 shows the logical scheme of the test bed and reveals some details about the physical part of the infrastructure and its parameters.

The data were generated by IDS *Suricata* installed on two servers, which were attacked by malicious traffic from the system based on the *Metasploit Framework*. Thus, the accuracy of the IDS responses and the performance of storing data in *Cassandra* were simultaneously checked. For stress-testing, the logging details were chosen as the most verbosity, and the events were described in JSON format in the size of 20-22 Kb.

We used a private cloud based on *OpenNebula* 5.4.6 and 6x Intel Xeon X5670 48 Gb RAM servers for the experiments. Virtual machines were created using KVM. Their virtual disks used the write-back policy of local caching. Physical disks inside the nodes were connected via iSCSI to the SAN based on *OpenMediaVault*, where RAID-Z was configured on the ZFS system with the cache on SSDs. SAN network was 2x10GbE per node (2 nodes), the MTU was set to 9000.

The volume of generated data is 81958 poorly structured events in a mixed text/JSON format (*Syslog*) with the total size of 1703 Gb. The primary data were being saved in a single input table with which *Spark* worked later to structure them. That data were being written back to *Cassandra* to another keyspace. The structure of the data was generated in advance based on a preliminary analysis of the raw data fields.

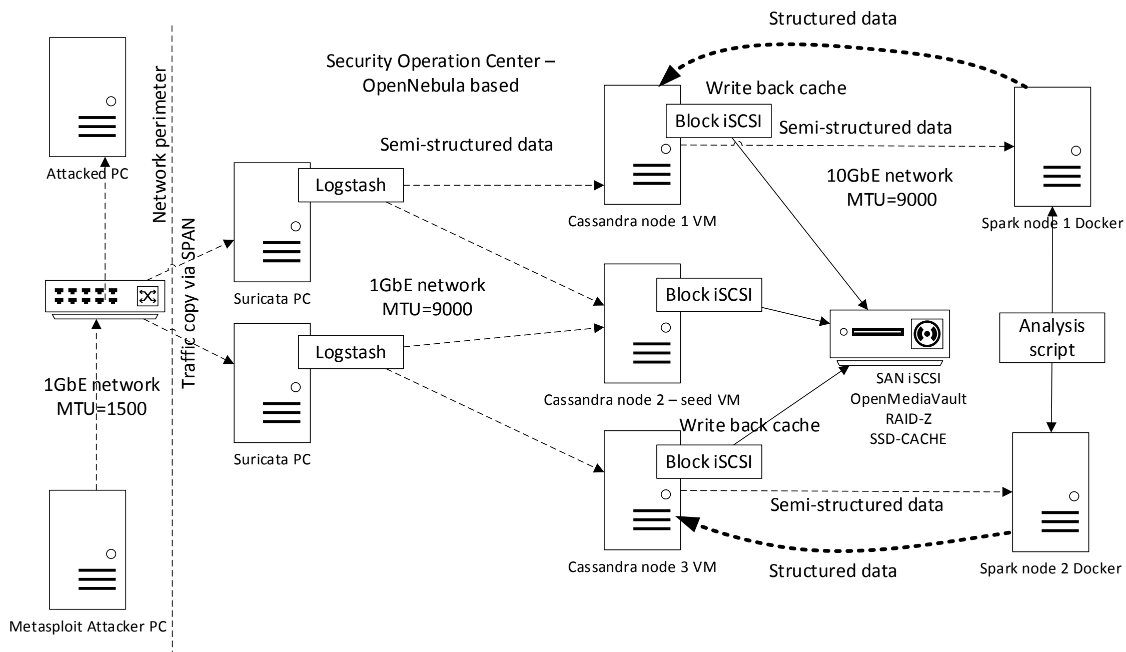

Figure 5. The architecture of the test bed.

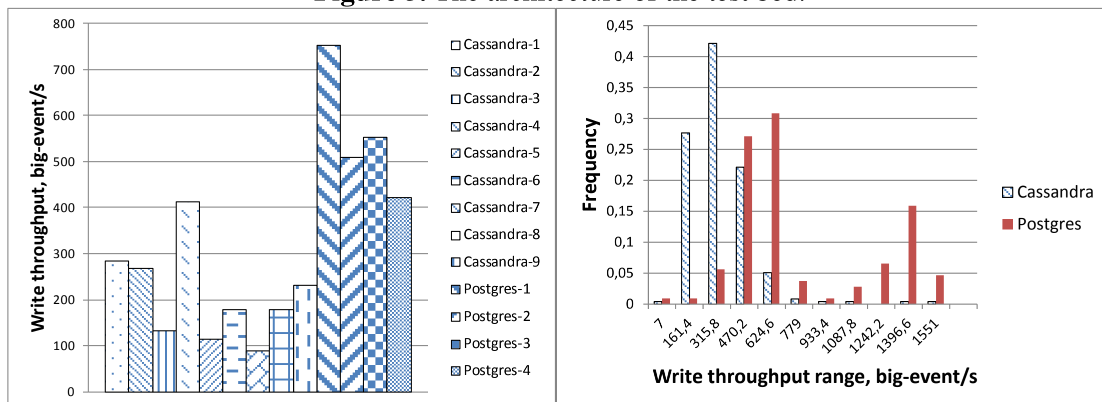

Figure 6. The write throughput for experiments.

Each experiment started with a clean system with removed data and cleared caches. We measured the performance of write throughput of the system with different values of replication factor for keyspace, partition batch size, and chunk size for *Cassandra* connector. Data were saved to volume mounted as "*/var/lib/cassandra*". It was used as a storage, but in some experiments, additionally, we used the directly mounted NFS disks to test common cases. In order to compare the performance with the traditional approach for data storage, the series of experiments were conducted on the same virtual nodes but using *PostgreSQL* as a data storage. The first series of experiments (with *Cassandra*) used the text format for all fields, the second one (with *PostgreSQL*) – JSONB for the data produced in this format. In addition, both volume and NFS disks were used as storage. The results are shown in Table 1 and in Figure 6.

The left graph of Figure 6 shows the average throughput of successful writing requests of large (>20Kb) rows for all experiments from Table 1. The right graph is the distribution histogram for 11 buckets of the write throughput aggregated over all experiments. Aggregated results are presented separately for experiments with *Cassandra* and *PostgreSQL*. According to this graph, there is a noticeable difference in distributions – *PostgreSQL* has a higher rate of requests with high throughput. In Figure 6, we use big-event/s units to distinguish the idea of a classic event for security system (several hundreds of bytes) from the full description of the security event, especially generated by IDS.

Table 1. The results of experiments.

Experiment title	Storage engine	Replication factor	Batch size	Chunk size	Number of parallel processes	Average write throughput, events/s	Average write throughput, Mbit/s
Cassandra-1	Cassandra over volume	2	5	3	2	284	50.7
Cassandra-2	Cassandra over volume	1	5	3	2	286.1	47.9
Cassandra-3	Cassandra over volume	1	5	3	1	133.9	23.9
Cassandra-4	Cassandra over volume	1	5	3	3	413.3	73.8
Cassandra-5	Cassandra over volume	2	5	4	1	115.3	20.6
Cassandra-6	Cassandra over volume	2	5	3	3	178.8	31.9
Cassandra-7	Cassandra over volume	1	3	2	2	88.4	15.8
Cassandra-8	Cassandra over volume	2	3	2	2	100.9	18
Cassandra-9	Cassandra over NFS	1	3	2	2	232	41.4
Postgres-1	PostgreSQL over volume	2	-	-	1	753.7	134.6
Postgres-2	PostgreSQL over NFS	2	-	-	1	510.1	91.1
Postgres-3	PostgreSQL over volume and JSONB fields	2	-	-	1	552.8	98.7
Postgres-4	PostgreSQL over NFS and JSONB fields	2	-	-	1	422.1	75.4

The experiments, which used a volume, showed greater performance due to more optimal caching and file system operation than the experiments with data storage based on NFS on the same server.

The number of replicas for *Cassandra* affects the write throughput less than the number of parallel writing processes. The batch and chunk sizes affect more due to the shorter transactions of the connector. At the same time, *PostgreSQL* showed significantly better performance, even when using JSONB type to store the part of data fields. However, the data integrity and consistency are guaranteed by the relational model of *PostgreSQL*, and replications are configured several times more flexible.

7. Conclusion

Thus, we proposed the concept of a distributed system for collection and primary analysis of security events and incidents in the networks. It is able to extract events from different sources in the entire network, process them, and output results in a convenient format. The system works with real-time streaming data, which is surely its main advantage. Such a solution is suitable for large corporate networks, and due to the high scalability of its components, it can work with big data.

In the future, the functions of the system can be extended by new types of analysis, optimization, and by adding as many event sources as possible.

8. References

[1] Jirsik T and Celeda P 2018 Toward real-time network-wide cyber situational awareness *Network Operations and Management Symposium*
[2] Carvalho V S, Polidoro M J and Magalhães J P 2016 OwlSight: platform for real-time detection and visualization of cyber threats *Big Data Security on Cloud (BigDataSecurity), International Conference on High Performance and Smart Computing (HPSC), and International Conference on Intelligent Data and Security (IDS)* 61-66
[3] Marchal S 2014 A big data architecture for large scale security monitoring *IEEE International Congress on Big Data* 56-63
[4] Sapegin A, Jaeger D, Cheng F and Meinel C 2017 Towards a system for complex analysis of security events in large-scale networks *Computers & Security* **67**16-34
[5] Dahiya P and Srivastava D K 2018 Network intrusion detection in big dataset using Spark *Procedia computer science* **132** 253-262
[6] Kafka 2.0 Documentation URL: http://kafka.apache.org/documentation/
[7] Apache Cassandra Documentation v4.0 URL: http://cassandra.apache.org/doc/latest/
[8] Stevenson A *Connecting to Apache Kafka. The Connect API in Kafka Cassandra Sink: The Perfect Match* URL: https://docs.confluent.io/3.0.0/connect/intro.html
[9] Kafka Connect Cassandra URL: https://www.confluent.io/connector/kafka-connect-cassandra/
[10] Kindling: An Introduction to Spark with Cassandra (Part 1) URL: https:// www.datastax.com/dev/blog/kindling-an-introduction-to-spark-with-cassandra-part-1
[11] Rsyslog Documentation URL: https://www.rsyslog.com/doc/v8-stable/
[12] Suricata User Guide URL: https://suricata.readthedocs.io/en/suricata-4.1.0/

Acknowledgments

The research was supported by the Russian Foundation for Basic Research in the framework of the scientific projects No. 16-29-09639, 18-07-01446, 18-37-00460.