FACTA UNIVERSITATIS
Series: **Electronics and Energetics** Vol. 34, No 2, June 2021, pp. 239 - 257
https://doi.org/10.2298/FUEE2102239C

Original scientific paper

INPUT SPLITS DESIGN TECHNIQUES FOR NETWORK INTRUSION DETECTION ON HADOOP CLUSTER

Vladimir Ćirić, Dušan Cvetković, Nadja Gavrilović, Natalija Stojanović, Ivan Milentijević

University of Niš, Faculty of Electronic Engieering, Niš, Serbia

Abstract. *Intrusion detection system (IDS) is one of the most important components being used to monitor network for possible cyber-attacks. However, the amount of data that should be inspected imposes a great challenge to IDSs. With recent emerge of various big data technologies, there are ways for overcoming the problem of the increased amount of data. Nevertheless, some of this technologies inherit data distribution techniques that can be a problem when splitting a sensitive data such as network data frames across a cluster nodes. The goal of this paper is design and implementation of Hadoop based IDS. In this paper we propose different input split techniques suitable for network data distribution across cloud nodes and test the performances of their Apache Hadoop implementation. Four different data split techniques will be proposed and analysed. The techniques will be described in detail. The system will be evaluated on Apache Hadoop cluster with 17 slave nodes. We will show that processing speed can differ for more than 30% depending on chosen input split design strategy. Additionally, we'll show that malicious level of network traffic can slow down the processing time, in our case, for nearly 20%. The scalability of the system will also be discussed.*

Key words: *Network Intrusion Detection, Cloud Computing, Apache Hadoop.*

Received September 16, 2020; received in revised form October 28, 2020

Corresponding author: Vladimir M. Ćirič
Faculty of Electronic Engineering, Computer Science Department, Aleksandra Medvedeva 14, 18000 Niš, Serbia
E-mail: vladimir.ciric@elfak.ni.ac.rs

1 Introduction

The complexity of the Internet, diversity of available services, and the desire to expand applications of the global network contribute to its increased insecurity. Even with decades of research, and a lot of available security products, the internet has steadily become more and more dangerous [1, 2].

Living in the era when everything is connected to the internet requires a different security strategy. When the attack begins, it is irrelevant how the network is configured or what kind of "boxes" network has, or how many security devices are installed. The only thing that matters is who is defending the network. The only way to stay ahead of new vulnerabilities and attacks is through vivid detection and response [3]. Unfortunately, constant security monitoring is a key component missing in most networks [4, 5].

Intrusion detection system (IDS) is one of the most important components used to detect attacks in monitored network traffic [6]. Intrusion detection is broadly considered to be a classification problem. Based on their classification model IDSs are classified into signature (or pattern) matching and anomaly based IDS. The signature matching IDS monitors the network activity for a known misuse pattern that was previously identified as a malicious attempt [6].

Having in mind typical bandwidths on the network boundaries, the amount of data that need to be analyzed for malicious signatures becomes challenging. There are IDS implementations available that tend to speed up network packet analysis [7–12]. Different approaches to task and data parallelism were exploited [9, 10, 12]. Some implementations use multi-core software development frameworks to parallelize the execution on CPU [11], while some utilize GPUs [8].

The Apache Hadoop is a framework for distributed processing of large amount of data on clusters of computers (nodes) using MapReduce programming model, where each node offers local computation and storage [13]. Hadoop Distributed File System (HDFS) is used for distributed data storage, and it represents a layer above existing file system of every node in cluster used to store input files or parts of them. Large files are split into a group of smaller blocks. Size of these blocks is fixed, so it is easy for Hadoop to index any block within the file [7]. However, this data distribution technique can introduce problems when splitting a sensitive data such as network data frames across a cluster nodes. Due to the fixed size of the block, one part of the network packet can end up on one node, while the other part is on the other, making malicious pattern matching challenging [14, 15].

Several authors already dealt with the problem of IDS implementation on Hadoop. However, for the best of our knowledge there is no solution that implements IDS on Hadoop without support of other software tools. In [16,17] the authors used Hadoop to analyse logs gathered from well-known Snort IDS. In [18] the authors proposed Hadoop as a distributed database manager, but the main processing isn't performed by Hadoop.

The goal of this paper is design and implementation of IDS based on Apache Hadoop, with focus on data splitting and distribution techniques to cluster nodes. In this paper we propose different input split techniques suitable for network data distribution across cloud nodes and test the performances of their Apache Hadoop implementations. Four different data split techniques will be proposed and analysed. The techniques will be described in detail. The IDS will be implemented using Myers pattern search algorithm as a core for signature-based packet analysis and evaluated on Apache Hadoop cluster with 17 slave nodes. We will show that processing speed can differ for more than 30% depending on chosen input split design strategy. Additionally, we'll show that malicious level of network traffic can slow down the processing time, in our case, for nearly 20%. The scalability of the system will also be discussed.

The paper is organized as follows. Section 2 gives a brief introduction to IDS. Section 3 is devoted to the MapReduce framework, as a basis for the proposed Apache Hadoop implementation. Section 4 is the main section and presents the design of the IDS workflow on the Hadoop framework. In this section we will discuss the design of data input split techniques, as well. Section 5 is devoted to the system evaluation, while in Section 6 concluding remarks are given.

2 Intrusion Detection System Background

IDS monitors network traffic and deploys various techniques in order to provide security services. Based on the technique used to assess the network packets as regular or malicious, IDSs are classified into signature (or pattern) matching and anomaly based IDSs [11,19,20]. The signature matching IDS searches the network traffic for a known misuse pattern that was previously identified as a malicious attempt [7,8]. A database with malicious signatures is prepared in advance. This leads to fast and reliable operation, but these IDSs are not able to detect new attacks that have not been seen before. The anomaly based detection IDSs make the decision based on a profile of a normal network behavior, and they are capable of detecting zero day attacks, with a drawback of possible false positives [20,21]. In this paper we will focus on pattern matching based IDS.

The workflow of pattern matching based IDS is shown in Fig. 1 [12,19]. Intrusion detection starts with network monitoring, followed by network packet preparation for efficient pattern matching, which is based on the predefined signature database (Fig. 1). Network monitoring can be performed as packet capture, deep packet inspection and flow-based monitoring. Packet capture intercepts a data packet that is crossing over a specific computer network, but it focuses only on packet headers. Deep packet inspection (DPI) is an advanced method of packet filtering, which inspects at the application layer of the OSI (Open Systems Interconnection) reference model.

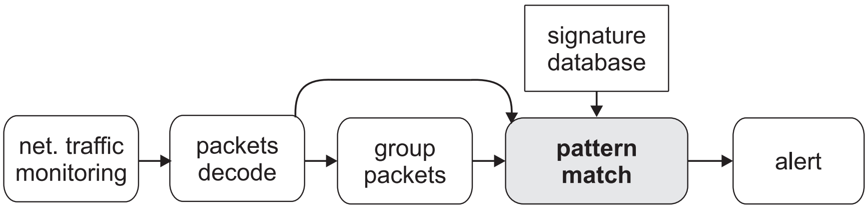


Fig. 1: The typical IDS workflow.

Any signature based IDS checks the presence of a malicious signature in the incoming packet sequence and act as instructed by the corresponding rule. Snort is a widely used open-source IDS based on pattern matching [11].

The pattern matching algorithm must be fast enough in order to support the network link speed. There are various implementations of pattern matching algorithms [7–12]. We will use Myers pattern search algorithm for DPI packets inspection, with rules in Snort syntax as proposed in [12]. In order to speed up pattern matching, in this paper we choose Apache Hadoop distributed environment, with focus on network traffic data distribution across the nodes.

3 Apache Hadoop HDFS and MapReduce

The Apache Hadoop is a framework for distributed computing based on MapReduce programming model, where each computer in a Hadoop cluster (node) offers local computation and storage [13]. The Apache Hadoop cluster consists of one master and many slave nodes. The Apache Hadoop is available in versions 1.x and 2.x. There are two main components of Hadoop 1.x system: Hadoop Distributed File System (HDFS), used for distributed data storage, and MapReduce computing framework for data manipulation. The architecture of Hadoop 2.x adds YARN (Yet Another Resource Negotiator) as an extension for resource management.

The HDFS is an abstraction of all file systems of cluster nodes, which creates an illusion of common data file storage. Large files are split into a group of smaller parts called blocks (default block size is 64MB) [13]. The size of blocks is fixed, due to the simplification of indexing. The HDFS is master-slave architecture, based on the existence of two types of (Linux) deamons: DataNode and NameNode (Fig. 2). NameNode is executed on the master node and it is responsible for managing DataNodes (slaves) [13].

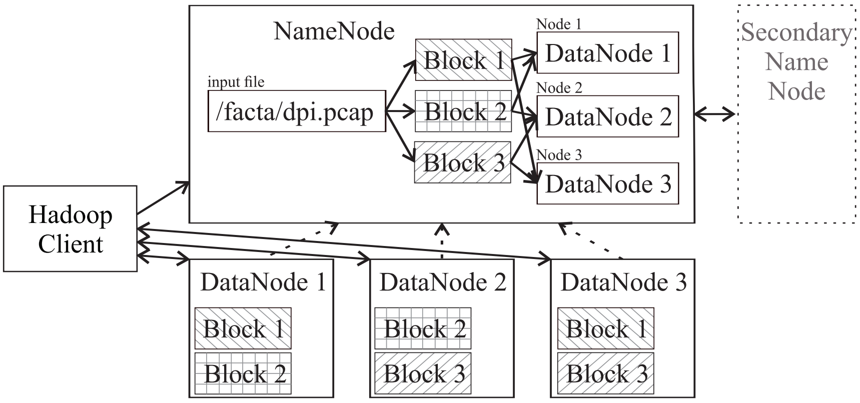


Fig. 2: HDFS components and their communication

The NameNode is also responsible for taking care of the replication factor of data blocks. The replication factor contributes to data fault tolerance by creating a several copies of each block across the cluster. In Fig. 2 the replication factor is 2 (default replication factor is 3). In case of the DataNode failure, the NameNode chooses new DataNodes for new replicas, balances disk usage and manages the communication traffic to the DataNodes [13].

Typical Hadoop workflow has 4 parts: (1) transferring input data from Client host to HDFS, (2) processing data using MapReduce framework on the slave nodes, (3) storing results on HDFS, and (4) reading data by Client host from HDFS.

MapReduce is programming model for distributed data processing, where the map function is applied on every data element in parallel, followed by the reduce function that summarize the collections of intermediate results produced by the map functions. MapReduce paradigm assumes that there are no data dependencies between any given instance of the map functions.

The map and reduce functions are implemented in Hadoop as follows. Before the beginning of execution, the input data files must be added to the HDFS. The beginning of the data processing itself is the determination of the logical units that will be processed - Input Splits. The most common case is that one Input Split corresponds to one block on HDFS, but it is not necessary to be so. In case the data requires it, the partitioning of data to Input Splits can be done differently, through a special implementation of the *InputFormat* class that will create them [14, 15].

All input and output data are given in key-value pairs $< k, v >$. The default behavior is to use *TextInputFormat*, where the key is an offset in bytes from the beginning of file, and the value is the content of one line of the file. The binary files can be used as well. One map task processes one input split (Fig. 3). Each Input Split is divided into records, which are represented as key-value pairs $< k_i, v_j >$. Each pair is processed by a map task with one call of the map function. The map function takes one key-value pair $< k_i, v_j >$ and executes given operations on them. It produces the intermediate results also in the form of key-value pairs $< k_n, v_m >$ (Fig. 3). Those results are then grouped in such manner that all pairs having the same key are sent to the same reducer. Reducer summarizes all data with the same key in order to get the final result (Fig. 3).

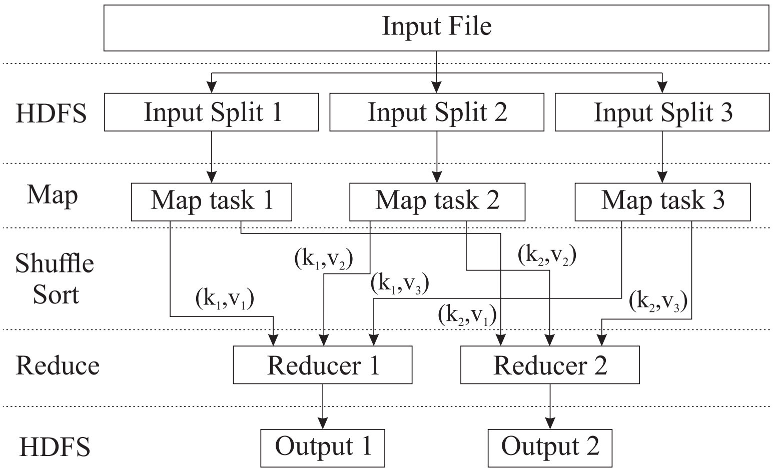


Fig. 3: The MapReduce execution

In order to design an efficient Hadoop based IDS, due to the fixed size nature of HDFS data blocks, in this paper we'll focus on experimenting with different techniques of dividing the input data into Input Splits.

4 Design of Hadoop based IDS

The architecture of the proposed IDS is shown in Fig. 4. The proposed architecture uses available Snort rules database and distributes pattern search across the Hadoop cluster. Due to the default behavior of HDFS to split the data into a fixed size blocks, and the nature of network protocols to have a packets of different sizes, the crucial design decision is how the packets on block boundaries will be handled. Having this in mind, we introduced *pcap_input_format* packet in the architecture from Fig. 4, which will allow us to experiment with different approaches by abstracting the *InputFormat* class mentioned in the previous Section.

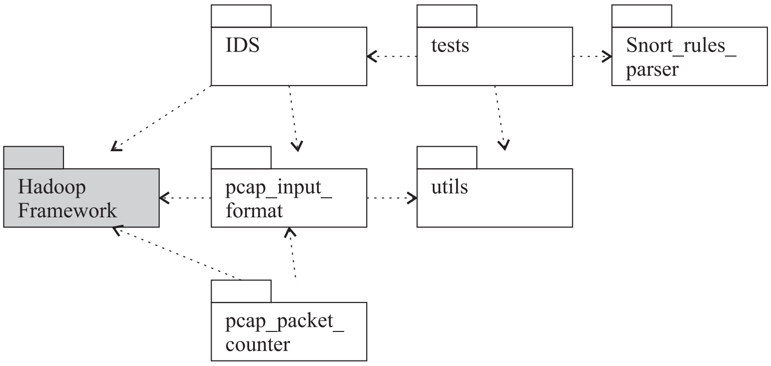


Fig. 4: The architecture of the proposed IDS

The IDS packet from Fig. 4 is a central part that implements MapReduce pattern search through captured network traffic using Myers algorithm [12]. We chose the standard *pcap* format for capturing and storing the network traffic [3]. The architecture's packet *pcap_input_format* from Fig. 4 is specialized for controlling the boundaries of the Input Splits, while *pcap_input_counter* tests the validity of its execution. The *Snort_rules_parser* creates a distributed cache out of Snort rules that will be used as an input in the pattern search algorithm. The pattern search algorithm itself is implemented in the *tests* packet, while *utils* provide *pcap* network traffic decoding functionalities.

4.1 The input file format

De facto standard for network traffic capture and storage is *pcap* file format, which is used by various well known network tools such as Wireshark, tcpdump, libcap, etc. [3]. The internal structure of *pcap* file is given in Fig. 5. The file begins with global header, after which the particular network traffic packets follow. Global header contains, among the others, two important information (Fig. 5): the network protocol of the stored packets (`network`), and the maximum length of the stored packets (`snaplen`). The network protocol is in the most of cases Ethernet protocol, but it can be IP or any other. The maximum length of the stored packets is the feature that enables storage of the beginning of the packets only, for the sake of efficiency, in the cases when only headers are required. In such cases `snaplen` value is less then the value indicating original packet len in the actual header of the packet, showing that only the first `snaplen` bytes of the packet are stored.

Each packet header from Fig. 5 contains the information about the stored network packet and should not be confused with actual network protocol header. The packet header from Fig. 5 contains *pcap* information about the time when the packet is captured (`ts_sec` and `ts_usec`), and its length (`incl_len` and `orig_len`).

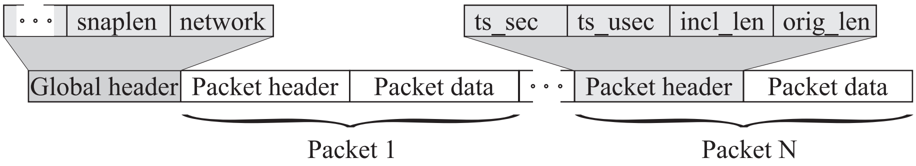


Fig. 5: The internal structure of *pcap* file

As network protocol packets can have variable length (from few bytes to several tens of kB, depending on protocol), and HDFS blocks are of the fixed size, the fields `incl_len` and `orig_len` are of the great importance for the proposed system. The `incl_len` field represents the length of the packet in bytes as it is stored in the pcap file, while the `orig_len` field gives its original length in bytes as seen on the network. For each packet the following relation stands

$$\texttt{incl\_len} \leq \texttt{snap\_len} \leq \texttt{orig\_len}, \qquad (1)$$

where only the first `incl_len` bytes of each packet are captured in the *pcap* file. Here we will demonstrate and compare several techniques for input splits design, having in mind variable nature of network traffic packets.

4.2 Input split techniques

We performed experiments with four different Input Split designs. Within the first design technique we use textual file as an input, while in the next three techniques we use binary file format with different network packet and HDFS data block aligning techniques.

Technique 1 - *tshark* packet pre-decoding

The simplest solution regarding the implementation of the map and reduce functions is to use input file in textual format, and to pre-decode captured network traffic prior to placing the input file on the HDFS. The paper [12] deals with this particular type of implementation.

As malicious attempts can be recognized from their signatures in the form of character or byte arrays, *pcap* file should be decoded in order to obtain data in plain text from all headers of encapsulating network protocols (i.e. Ethernet, IP, and TCP), including data carried by the application layer. In this case it is not necessary to implement custom Hadoop *InputFormat*, but the *TextInputFormat* can be used instead. We use *tshark* Linux command line tool for network traffic decoding. The example of *tshark* tool usage is:

```
tshark -r <pcapfilename> -T fields -E separator=, -e ip.addr -e
_ws.col.Protocol -e tcp.port -e udp.port -e data > output.txt
```

Each line in the *output.txt* contains the information from one fetched network packet, which now represents the input file for HDFS. The input file is divided in Input Splits, and each map task is fed by one Input Split. The map function implemented to support this technique takes one line at the time, and executes the pattern search algorithm. If some of the Snort rules match the malicious network packet, the mapper emits $< key, value >$ pair, where the *key* stands for the attack identification, while the *value* is constant 1. Having the same key, the results from the same malicious flow go to the same reducer, which counts the malicious packets in the flow and outputs the result.

The advantage of this approach is ease of Hadoop implementation, with the drawback that packet decoding have to be done prior the beginning of Hadoop program and pattern search.

Technique 2 - Custom *InputFormat* for *pcap* input

In order to overcome disadvantages of the previous technique, *pcap* file have to be used in the original binary format, without pre-decoding. This can be achieved by implementation of custom Hadoop *InputFormat* that will process *pcap* files. Two considerations should be taken into account: (1) How to divide the input file into Input Splits? (2) How to read records from an Input Split and feed them into the map function?

During the input file processing and creation of each Input Split, in this technique we ensured that the boundary between each two adjacent Input Splits is exactly at the boundary between two packets (Fig. 6). We crawled the *pcap* file package by package until the configured block size limit (Fig. 6). At that point, a new Input Split is created. Since the division is performed at the packet boundaries, the sizes of the obtained Input Splits can differ from each other, but no more than the maximum length of one packet, which is 1536 bytes for Ethernet).

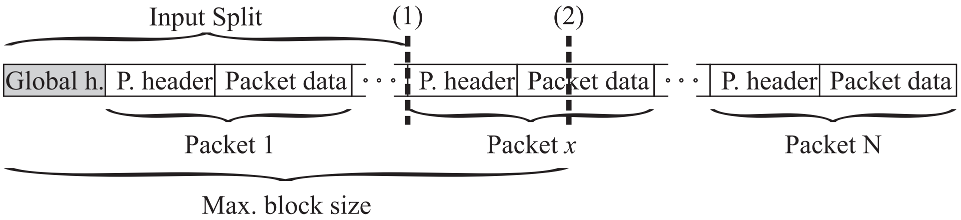


Fig. 6: Dividing the pcap file into Input Splits: (1) correct boundary, (2) incorrect boundary

In order to perform pattern search within the map function, the map function requires both the packet header and the packet data. The output from the mapper is in the form $< key, value >$, where the *key* is the offset of the beginning of the packet header, and the *value* is the whole packet in its original binary format.

In this approach the Mapper itself decodes the binary packet and locates all required data (IP addresses, ports, payload, etc.), making the decoding distributed operation, as well. As Myers algorithm natively works with bytes, Mapper only have to decode the packet up to the application layer (to find IP addresses and port numbers), but not the application layer payload itself, which, compared to *tshark*, reduces the number of operations required for decoding.

Technique 3 - Custom *InputFormat* with probabilistic packet boundary detection

Although the previous technique is much better than *tshark* packets decoding due to its distributed packet decoding, it still crawls from packet to packet through the *pcap* file with aim to align the boundary of the Input Split with the boundary of the packet. In order to do so, it loads each network packet into the memory. This takes time before the start of "useful" distributed processing, and slows down whole processing. In order to avoid this bottleneck we propose the third technique - probabilistic packet boundary detection, where we assume that *pcap* file contains the network packets captured only on the Data Link layer of the OSI reference model, i.e. that each "packet" in the *pcap* file is Ethernet frame.

Let the HDFS block size be z bytes. Here we propose not to load all z bytes into the memory in order to find the boundary between the Input Splits, but rather to skip the first x ($x < z$) bytes (Fig. 7). The question now is whether the boundary between the network packets lies on the chosen offset of x bytes? If so, then for the next package in the *pcap* file Eq. (1) should stand. This practically means that the value on the position of the `orig_len` field should be greater than zero, that the value of the `inc_len` field should be within the valid limits of the Ethernet frame size, and that the value of the `inc_len` field should be less than or equal to the value of the `orig_len` field. Other fields in the package itself must be valid, too. A suitable place for additional check is where the `ethertype` field in the header of the Ethernet frame should be. This value should be compared with the value that indicates the Ethernet protocol.

This is highly probabilistic and fuzzy way of boundary detection, and a few verified fields can mislead us by giving us a false positive answer. Thus, we check the same conditions for the following k packets (up to the packet denoted as $\mathrm{P}_{\mathrm{N}+k}$ in Fig. 7). If all conditions stand for the next k packets, we declare the boundary between the packets found, and create the Input Split. In order to perform mentioned additional checks on the next k packets, after skipping the offset of x bytes we load the following y bytes, as it is shown in Fig. 7. Let us note that $y \ll x$.

This can be used to form a probabilistic algorithm as follows: (1) if the verified conditions stand for the next k packets, we assume that the correct boundary between the Input Splits is on the x-th offset; (2) if the conditions don't stand at least for one of the k packages, the offset x is not the correct limit, and the offset of $x + 1$-st byte should be examined (Fig. 7). In the

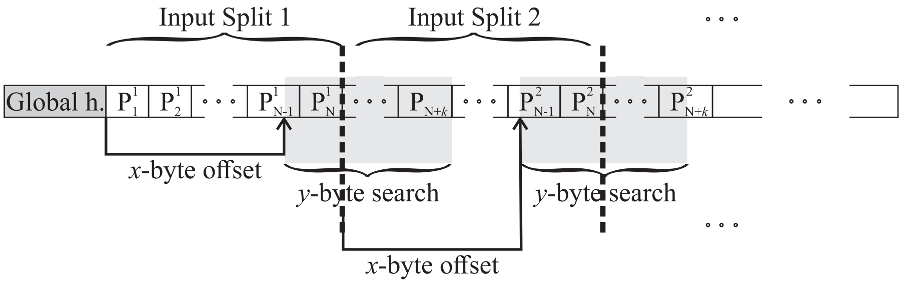


Fig. 7: Probabilistic method for packets boundaries detection

worst case scenario the number of offsets that should be examined is equal to the maximum length of Ethernet frame, which is not a problem having in mind that we already have y bytes from *pcap* loaded and available. Even in this case, it is much less than loading and processing of all z bytes.

With the careful selection of the parameters x, y and k, high degree of certainty of the proposed technique can be reached. In our implementation we selected the following parameters: $z = 128$MB, $x = 123$MB and $y = 5$MB. In the portion of $y = 5$MB there are more than 3.000 Ethernet frames, more then enough for us not to obtain any false positive, while loading only 4% of the *pcap* file.

Technique 4 - Custom *InputFormat* with aligned blocks and Input Splits

The previous technique has an important drawback regarding the way how the HDFS operates in the case of unequal sizes of blocks and Input Splits. In any case, the HDFS block size is constant. If the Input Split size is less than the block size, as it is in the previous technique, the boundary of the input split will not be aligned with the block boundary, forcing the HDFS to fill the remaining space with the next Input Split. That Input Split will be divided having a small portion in one block and a larger portion in the next block. The case when one Input Split resides in two blocks will force the Hadoop to copy both blocks on the node where the mapper who processes the particular Input Split is executed. This can cause large and unnecessary network traffic while copying the blocks.

To overcome this issue and prevent unnecessary blocks copying, we propose the fourth technique where we have a custom *InputFormat* and the exact same sizes of blocks and Input Splits (Fig. 8). Now, the boundaries of Input Splits are not aligned with the boundaries of network packets in most cases, and they split some packets into two parts (Fig. 8). Therefore, we will ignore the split packets as invalid. They intentionally will not be processed further through the MapReduce framework, leaving the small chance of false negative response of our network intrusion detection system for the sake of speed gain by avoiding of unnecessary block copying. In the worst case, the number of packets that will not be processed can be equal to the number of Input Splits, i.e. one invalid packet per Input Split (roughly one Ethernet frame per 100.000 frames will be ignored).

The problem that remains is finding valid beginning of the first packet within the Input Split, and this is a reason for having a custom *InputFormat* within this technique, too. To find the first valid packet in Input Split, we use the same probabilistic algorithm as in the previous technique, and we search through the first y bytes of the Input Split for the valid beginning of Ethernet frame (Fig. 8).

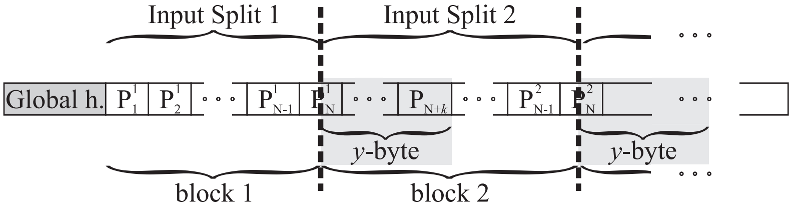


Fig. 8: Aligned blocks and Input Splits

5 Implementation results

The proposed techniques are suitable for implementation in both Hadoop 1.x and 2.x without any restrictions. In order to evaluate the proposed techniques, the IDS is implemented in Apache Hadoop 1.x and 2.x, and tested on a cluster with 18 commodity nodes, where 1 node is a master while the rest 17 nodes are slaves. The nodes are equipped with Intel(R)Core(TM)2 Duo, CPU E4600@2.40GHz, and 1GB of RAM. In order to compare the performances of the proposed Hadoop IDS with the reference Snort IDS, evaluation of the technique 1 is done in single-processor environment, because the Snort IDS doesn't support distributed execution. For that purpose we used an environment with i3 6006U CPU and 8GB RAM. The processor E4600 has 2 cores and operates at 2.4GHz, with Whetstone benchmark results 2.25 FLOPS per core, i.e. 4.50 FLOPS in total. The processor i3 6006U has 4 cores at 2.00GHz, with Whetstone benchmark results 2.08 per core, i.e. 8.31 in total [22].

We used input *pcap* files of variable sizes with 1, 2, 3, 4, and 5GB of network traffic data, having "low", "medium", and "high" number of malicious packets (less than 1%, 30-40%, and more than 70%, respectively). We also varied the number of Snort rules and the number of slave nodes in the cluster.

The proposed Hadoop IDS with technique 1 for Input Splits preparation (*tshark* packet pre-decoding) is evaluated in Hadoop 2.9.0, in pseudo-distributed environment on the previously mentioned single-processor system, along with the Snort IDS. The results are given in Table 1.

	input file size [GB]	preprocessing time [s]	pattern search time [s]	total processing time [s]
Snort IDS	1	0	120	120
Hadoop IDS	1.6	211	94	305

Table 1: Hadoop IDS with *tshark* packet pre-decoding vs. Snort IDS

The used input *pcap* file size is 1GB, and it contained about 2 million network packets with "medium" number of malicious packets. As it can be seen from Table 1, packet pre-decoding increased the file size from 1GB to 1.6GB. Nevertheless, the proposed IDS has 21% faster pattern search time than the Snort (94s vs. 120s). This is due to the fact that the execution of Hadoop IDS takes advantage of multi-core CPU, while the Snort doesn't. However, the time required for *tshark* packet pre-decoding took more than 3 minutes (211 seconds), giving the total processing time for Hadoop 2.5 times slower than the Snort IDS.

For the job execution purposes Hadoop 2.x requests three different kinds of containers from YARN: the application master container, map containers, and reduce containers. Application master itself requires 1.5GB or RAM by default, making Hadoop 2.x suitable for large clusters with a lot of resources. The proposed techniques 2, 3 and 4 are evaluated in Hadoop 1.2.1 environment, due to the lower resource requirements. For the independent variables in the experiment we chose input *pcap* file size, the level of malicious packets, the number of Snort rules in the database, and the number of slave nodes. As dependant variables we obtained the total execution time, the total number of map tasks, as well as the number of data local and rack local[1] map

[1] Data local map task is a map task which has data block already locally available on the node where it executes prior to the execution, while rack local map task needs to fetch the data block from the other slave node.

tasks.

Fig. 9 shows the evaluation results of the techniques 2, 3 and 4 for 1GB input file size with low level of malicious packets, on a cluster with 17 slave nodes, and the Snort database with 1000 rules. The results are as expected: the Hadoop IDS with input split technique 4 has the best execution time (Fig. 9a). In this case it performs 32% faster then the proposed technique 2 (170 vs. 250 in Fig. 9a). The number of DataLocal and RackLocal blocks confirm the design hypothesis about additional block copying (Fig. 9b). The techniques 2 and 4 have the same number of map tasks due to the fact that both techniques force the size of the input split to be exact (Tcq4) or very close to the size of a block (Tcq2), while Tcq3 introduces the greatest deviation between the size of an input split and a block.

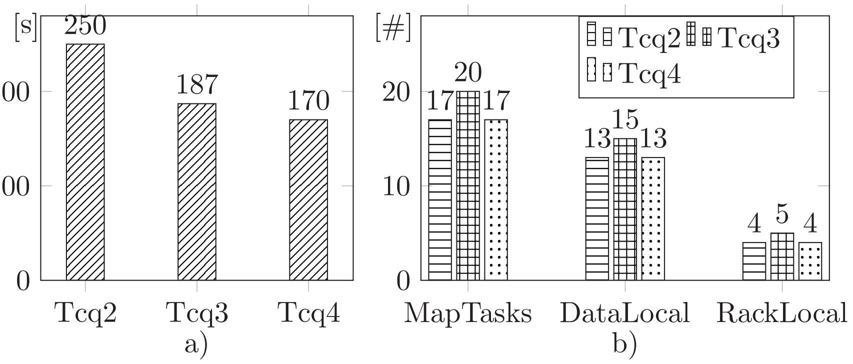
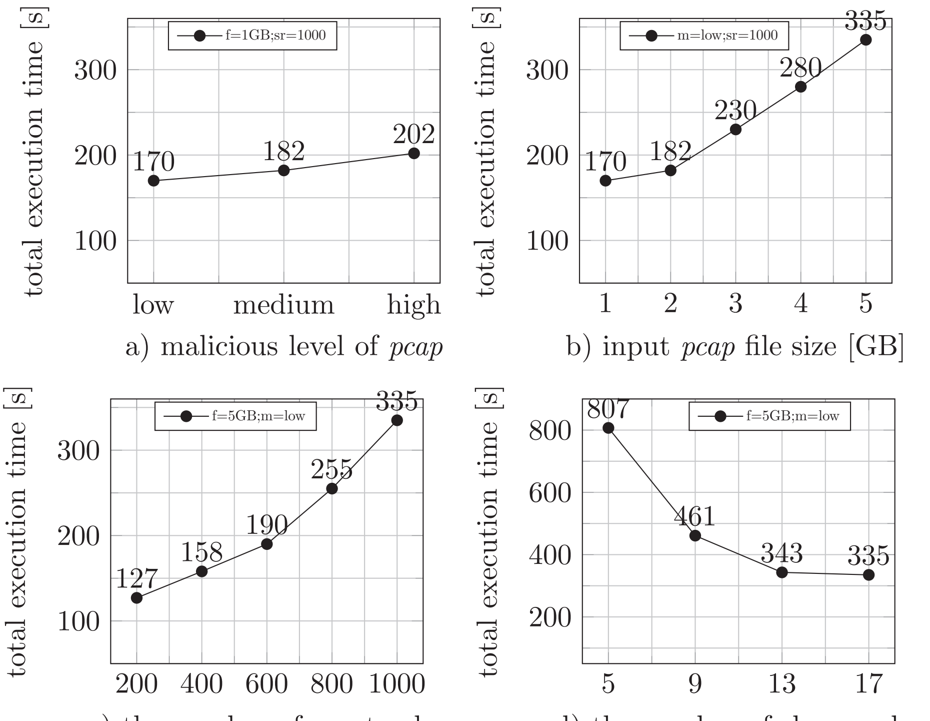


Fig. 9: Evaluation results of different Input Split design techniques: a) total processing time for techniques 2, 3 and 4, b) map tasks and blocks distribution across the cluster

We also evaluated how the technique 4 performs with variable malicious level of input *pcap* file and variable file size, how it performs with variable number of Snort rules in database, and how it performs in the clusters with different number of slave nodes. The evaluation results are given in Fig. 10. For better introspection, we used the same parameters for the starting points of graphics in Figs. 10 a) and b) as in Fig. 9 a): Tcq4, Snort database with 1000 rules, input file size 1GB, and low *pcap* malicious level. Figs. 10 b), c), and d) have one common point, as well.

From Fig. 10 a) it can be seen that the total execution time strongly depends on the number of malicious attempts in the network traffic flow. In this case the total execution time differs for 18% (202 vs. 170 in Fig. 10a). This is not a consequence of the choice of pattern search algorithm, but



a) malicious level of *pcap*

b) input *pcap* file size [GB]

c) the number of snort rules

d) the number of slave nodes

Fig. 10: Evaluation of the Tcq4 with variable parameters: a) variable number of malicious packets in *pcap*, b) variable input *pcap* file size, c) variable number of snort rules, d) variable size of the cluster.

rather the consequence of Snort rules database structure. The used Myers algorithm has a stabile execution time which is not affected by the contents of neither text nor pattern [12]. The Snort database is hierarchically organized, having rules categorized in levels from general to specific. For example, if the protocol is not http, the SQL injection rules are not going to be examined. Thus, for the low malicious *pcap* a lot of packets are simply skipped after ports and protocols check, and the Myers search algorithm is not started for them. If the monitored traffic contains a lot of packets that fall into "suspicious" category, one or more additional pattern searches are going to be performed, depending on the number of specific rules bound to matched general rule. This directly reflects the results from Fig. 10a.

Let us note that IDS process in general can be strongly affected with the choice of the pattern search algorithm, as well as with the specifics of topology and network organization, as well. If anomaly based approach is chosen instead of pattern search, the search performances can be significantly affected also.

From Fig. 10 b) it can be seen that the total execution time linearly depends on the size of the input files. The very slow growth in the beginning of Fig. 10b between execution times for 1GB and 2GB inputs is explained by the fact that the cluster consists of 17 nodes, where each node can execute 2 map tasks on two separate cores in parallel, giving the maximum of 34 simultaneously executed map task on the cluster. A 1GB file is presented on HDFS with 16 blocks of 64MB, while a 2GB file is presented with 32 blocks. This means that all necessary tasks can be run at the same time for both 1GB and 2GB files. The rise in the execution time between these two is due to the increased number of rack local tasks for the larger file. For larger files, more than 34 map tasks are required for processing, which means that not all of them can be started immediately, and they need to wait for the previously started map tasks to finish execution. However, the dependency once the cluster boundary is reached linearly increases (Fig. 10b).

The total processing time depends linearly on the number of Snort rules (Fig. 10c), while it has an asymptotic decline depending on the number of nodes (Fig. 10d). It can be noticed in Fig. 10d that for 17 nodes the graph enters saturation and the processing speed remains slightly under 0.15 Gb/sec. As the number of nodes in the cluster grows, each of them stores a smaller number of blocks on average. This leads to an increase in rack local tasks. Copying of remote blocks during the execution is a limiting factor that leads to the saturation in this case.

6 Conclusion

In this paper the design and implementation of IDS using Apache Hadoop is proposed. Four different input data split techniques are proposed and analysed. The techniques are described in detail. The IDS is implemented using Myers pattern search algorithm as a core for signature-based packet analysis. We showed the suitability of Hadoop environment for the implementation of network IDS and discussed inherited problem from Hadoop that relates to splitting sensitive data across cluster nodes. The system is evaluated on Apache Hadoop cluster with 17 slave nodes. The implementation and evaluation results are given and discussed in detail. We showed that processing speed can differ for more than 30% depending on chosen input split design strategy. Additionally, we showed that malicious level of network traffic can slow down the processing time, in our case, for nearly 20%. The scalability of the system was also discussed. The proposed techniques deal with specific type of input data, i.e. network traffic packets, but they can be easily generalized to deal with any type of sensitive data which need a special attention before it can be split into pieces and scatter onto different nodes in distributed environment.

Acknowledgments

This work was supported by the Serbian Ministry of Education, Science and Technological Development [grant number TR32012].

References

[1] L. A. Maglaras, K.-H. Kim, H. Janicke, M. A. Ferrag, S. Rallis, P. Fragkou, A. Maglaras, and T. J. Cruz, "Cyber security of critical infrastructures," *Ict Express*, vol. 4, no. 1, pp. 42–45, 2018.

[2] M. A. Ferrag, L. Maglaras, S. Moschoyiannis, and H. Janicke, "Deep learning for cyber security intrusion detection: Approaches, datasets, and comparative study," *Journal of Information Security and Applications*, vol. 50, pp. 1–19, 2020.

[3] J. Svoboda, I. Ghafir, V. Prenosil *et al.*, "Network monitoring approaches: An overview," *Int J Adv Comput Netw Secur*, vol. 5, no. 2, pp. 88–93, 2015.

[4] I. Ghafir, V. Prenosil, J. Svoboda, and M. Hammoudeh, "A survey on network security monitoring systems," in *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*. IEEE, 2016, pp. 77–82.

[5] B. Schneier, "Managed security monitoring: Network security for the 21st century," *Computers & Security*, vol. 20, no. 6, pp. 491–503, 2001.

[6] G. Kumar, K. Kumar, and M. Sachdeva, "The use of artificial intelligence based techniques for intrusion detection: a review," *Artificial Intelligence Review*, vol. 34, no. 4, pp. 369–387, 2010.

[7] M. Aldwairi and D. Alansari, "Exscind: Fast pattern matching for intrusion detection using exclusion and inclusion filters," in *2011 7th International Conference on Next Generation Web Services Practices*. IEEE, 2011, pp. 24–30.

[8] D. Xu, H. Zhang, and Y. Fan, "The gpu-based high-performance pattern-matching algorithm for intrusion detection," *Journal of computational information systems*, pp. 3791–3800, 2013.

[9] M. Kharbutli, M. Aldwairi, and A. Mughrabi, "Function and data parallelization of wu-manber pattern matching for intrusion detection systems." *Netw. Protoc. Algorithms*, vol. 4, no. 3, pp. 46–61, 2012.

[10] X. Su, Z. Ji, and X. Lian, "A parallel ac algorithm based on spmd for intrusion detection system," in *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering.* Atlantis Press, 2013.

[11] M. Aldwairi, A. M. Abu-Dalo, and M. Jarrah, "Pattern matching of signature-based ids using myers algorithm under mapreduce framework," *EURASIP Journal on Information Security*, vol. 2017, no. 1, pp. 1–11, 2017.

[12] V. Ciric, D. Cvetkovic, and I. Milentijevic, "Design and implementation of network intrusion detection system on the apache hadoop platform," in *Proceedings on 5th International Conference on Electrical, Electronic, and Computer Engineering (IcETRAN 2018)*, Palic, Serbia, 2018, pp. 1102–1105.

[13] C. Lam, *Hadoop in action.* Manning Publications Co., 2010.

[14] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson, "Cohadoop: flexible data placement and its exploitation in hadoop," *Proceedings of the VLDB Endowment*, vol. 4, no. 9, pp. 575–585, 2011.

[15] A. Sayar *et al.*, "Hadoop optimization for massive image processing: case study face detection," *International Journal of Computers Communications & Control*, vol. 9, no. 6, pp. 664–671, 2014.

[16] J. Cheon and T.-Y. Choe, "Distributed processing of snort alert log using hadoop," *International Journal of Engineering and Technology*, vol. 5, no. 3, pp. 2685–2690, 2013.

[17] P. Prathibha and E. Dileesh, "Design of a hybrid intrusion detection system using snort and hadoop," *International journal of computer applications*, vol. 73, no. 10, 2013.

[18] K. Kato and V. Klyuev, "Development of a network intrusion detection system using apache hadoop and spark," in *2017 IEEE Conference on Dependable and Secure Computing.* IEEE, 2017, pp. 416–423.

[19] C. F. Endorf, E. Schultz, and J. Mellander, *Intrusion detection & prevention.* McGraw-Hill Osborne Media, 2004.

[20] H.-D. J. Jeong, W. Hyun, J. Lim, and I. You, "Anomaly teletraffic intrusion detection systems on hadoop-based platforms: A survey of some problems and solutions," in *2012 15th International Conference on Network-Based Information Systems.* IEEE, 2012, pp. 766–770.

[21] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, "Survey of intrusion detection systems: techniques, datasets and challenges," *Cybersecurity*, vol. 2, no. 1, p. 20, 2019.

[22] U. of Washington, "Cpu performance," https://boinc.bakerlab.org/rosetta/cpu_list.php, accessed: 2020-10-22.