

MASARYK UNIVERSITY
FACULTY OF INFORMATICS



Real-time Detection of Communication with Blacklisted Network Hosts

BACHELOR'S THESIS

Ondřej Zoder

Brno, Spring 2018

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Ondřej Zoder

Advisor: RNDr. Milan Čermák

Acknowledgements

I would like to thank my advisor, RNDr. Milan Čermák, for giving me invaluable advice and all the information I needed to design and implement the tool that is an outcome of this thesis.

Abstract

The network traffic monitoring is an essential part of the management of computer networks. As the speed of computer networks increases almost exponentially, the demands on the performance of tools processing the network data get higher. With the emergence of new approaches for the processing of big data, the possibility to use these approaches for the network traffic monitoring became apparent. The goal of this thesis was to design and implement a highly scalable tool for the detection of communication with blacklisted network hosts using stream-processing frameworks that would later be part of the Stream4Flow framework. The application that is the outcome was tested on real anonymized data from the Masaryk University network, and results of these tests show that the proposed solution can be used for monitoring of large corporate or university networks.

Keywords

network security, network monitoring, network flow, blacklist, real-time, detection, apache spark, stream4flow, visualization

Contents

1	Introduction	1
2	Network Traffic Monitoring	3
2.1	<i>Simple Network Management Protocol</i>	3
2.2	<i>Deep Packet Inspection</i>	4
2.2.1	Tools Based on DPI	5
2.2.2	Drawbacks of DPI	6
2.3	<i>Flow Monitoring</i>	7
2.3.1	IP Flow Information Export	8
2.3.2	Flow Monitoring Setup and Process	9
2.4	<i>Stream4Flow Framework</i>	10
2.4.1	Stream Processing Framework	11
2.4.2	Apache Spark	11
2.4.3	Analyzed Data Storage	13
2.4.4	Web Interface	14
3	Blacklists	17
3.1	<i>Blacklist Creation Process</i>	17
3.2	<i>Blacklist Distribution Process</i>	19
3.3	<i>Common Use of Blacklists</i>	20
3.3.1	Anti-spam Filters	20
3.4	<i>Established Blacklist Databases</i>	21
3.4.1	Databases Used in This Thesis	22
4	Implementation - Detection	25
4.1	<i>Performance Testing Methodology</i>	25
4.1.1	Dataset Used for Testing	27
4.2	<i>Loading of Blacklists</i>	28
4.2.1	Loading from a Text File	29
4.2.2	Creating Stream from Python Collection	30
4.2.3	Sending Through Kafka	30
4.2.4	Conclusion	31
4.3	<i>Detection of Blacklisted Hosts</i>	32
4.3.1	Intersection	35
4.3.1.1	Joining Streams	35
4.3.1.2	Joining RDDs Using Transform Function	36
4.3.2	Union and Aggregation	36
4.3.2.1	Aggregation Using Group Function	37

4.3.2.2	Aggregation Using Reduce Function	38
4.3.2.3	Aggregation Using Combine Function	38
4.3.3	Conclusion	39
4.4	<i>Visualization</i>	41
4.5	<i>Results</i>	43
5	Conclusion	45
	Bibliography	47
A	Attachments	51

1 Introduction

The network traffic monitoring is an essential part of the management of computer networks ensuring their security. Blacklist as a mechanism used for detection of communication with network hosts having a bad reputation has been an important part of network monitoring for decades. With the rapid increase in the speed of computer networks, the analysis of the network traffic became a challenging task. Another challenge for monitoring tools is to minimize the time between the occurrence of a network event and its detection. The shorter is the time between an event and its detection, the more effectively can the network administrator respond to such event. The batch processing approach that is currently used for processing of network data delivers the results with a delay, which can be crucial when dealing with network threats.

The Stream4Flow framework aims to overcome all of these challenges and provide a highly scalable solution for near real-time monitoring of large computer networks. The network data are collected in the form of network flows and processed using stream-processing frameworks, namely Apache Kafka and Apache Spark. The detected data are stored in an Elasticsearch database and visualized in a web interface. The goal of this thesis was to design and implement a tool for detection of communication with blacklisted network hosts that would later be a part of this framework. All essential parts of the application were thoroughly tested throughout the whole process of implementation to achieve the best performance possible. The detection application, as well as the blacklist parser, were also designed to be easily upgradable.

This thesis consists of five chapters, starting with theoretical concepts behind the Stream4Flow framework and the proposed detection application and following up with a description of the implementation of the detection application. The second chapter focuses on a description of methods used in network traffic monitoring as well as a description of the Stream4Flow framework and its mechanisms. The third chapter describes blacklists and their use for the detection of undesirable communication in computer networks. The fourth chapter describes the implementation of the tool that is an outcome of this thesis and discusses the comparison of different data processing approaches. The conclusion chapter summarizes the achieved results.

2 Network Traffic Monitoring

Network traffic monitoring (the *monitoring*) is the use of a set of tools that monitor communication in a particular network, further analyze acquired data and send the result to the network administrator. Based on the influence the monitoring has on the analyzed network, monitoring can be classified as either **active** or **passive**. *Both are important and should be regarded as complementary. In fact, they can be used in conjunction with one another*, as described by Cottrell [1]. Active monitoring involves injecting test traffic into a network to measure the characteristics of that traffic. All the parameters of that particular traffic are adjustable, which enables the administrator to test a wide variety of network's functionality. This approach may decrease the performance as it adds overhead to the networking hardware. One of the tools that implement active monitoring is ping (Packet InterNet Groper), which tests the reachability of a host on the network by measuring the round-trip time. Passive monitoring collects data from existing traffic using an observation point. This method requires the presence of a device capable of capturing network packets. Passive monitoring can be further sub-categorized into three main types: Simple Network Management Protocol (SNMP), Deep Packet Inspection (DPI) and Flow Monitoring. As this thesis works with tools built upon passive monitoring, these three types will be further described.

2.1 Simple Network Management Protocol

Simple Network Management Protocol (SNMP) is an administration protocol supported by the vast majority of network devices. It has gradually evolved in three different versions sharing the same structure, with SNMPv2 currently being the most supported one. Devices from an SNMP-managed network are logically divided into two groups based on the functionality provided by the integrated software. The network contains at least one managing node with SNMP management applications, called a manager, and several managed nodes that provide remote access to managing nodes, called agents [2]. Network elements supporting SNMP are devices such as cable modems, routers, switches, servers, workstations, printers, etc [3]. The protocol operates in the application layer and transfers the management information between the nodes using User Datagram Protocol (UDP). These messages consist of a version identifier, an SNMP community name and a protocol data unit (PDU). The third version, SNMPv3, specifies seven PDUs. All of them share the following properties:

2. NETWORK TRAFFIC MONITORING

IP header	UDP header	version	community	PDU-type	request-id	error-status	error-index	variable
-----------	------------	---------	-----------	----------	------------	--------------	-------------	----------

The seven PDUs are the following: **GetRequest** is a request sent to an agent by a manager to load the value of one or more variables. The required variables are specified inside the variable bindings property. **SetRequest** is a request sent to an agent by a manager to change the value of one or more variables. New values are specified in the body of the request. **GetNextRequest** is a manager-to-agent request to retrieve available variables. **GetBulkRequest** is an optimized version of GetNextRequest. **InformRequest** was introduced in SNMPv2 and is used for manager-to-manager messages with guaranteed delivery. **Response** is a reply to GetRequest, SetRequest, GetNextRequest, GetBulkRequest or InformRequest requests. **Trap** is a notification by an agent sent to a manager informing about important events.

The main advantage of the SNMP is arguably its simplicity [4]. The configuration of SNMP is highly flexible, and the basic functionality can be further expanded. On the other hand, the first two versions of this protocol were prone to multiple types of attacks, such as *sniffing* (eavesdropping of network traffic), *brute force attacks* or attacks exploiting *default community strings*, where access information of the network can be misused. The first version even sends passwords over the network in plain text. The third version fixes some of these security vulnerabilities, but it still may be subject to brute force attacks.

2.2 Deep Packet Inspection

Deep Packet Inspection (*DPI*) is an approach used for network traffic monitoring that inspects both the payload and the packet header [5]. That means a significant increase in the amount of analyzed information. The additional insight gained from the data part of the packet enables a wide range of new functionality otherwise not feasible. The DPI evolved from Stateful Packet Inspection (SPI), sometimes called Shallow Packet Inspection. SPI operates on the network layer of the OSI (Open Systems Interconnection) model and inspects both incoming and outgoing packets. It secures that all incoming packets are a result of an outbound request. The only inspected part of the packet is the header. That means the packet data part can contain malicious content, as long as the packet header meets the criteria of a legitimate packet. On the other hand, DPI can be invoked in all the layers of the OSI model except the first layer. The packet can be classified using a database of information extracted from the payload. This classification allows packet marking

to ensure Quality of Service (QoS) or to report and block irregular packets. In case the packet is encrypted (e.g. when HTTPS is used), DPI becomes less effective unless the public key used for encryption is known.

2.2.1 Tools Based on DPI

One of the common tools based on DPI is **tcpdump** [6]. It runs under the command line, and by default, it prints the results onto the standard output. It can also save the data into a file by using the **-w** flag. The file can be used for analysis, or with the **-r** flag as a source for the packet capture instead of the network interface. The output starts with a timestamp as hours, minutes, seconds and fractions of a second since midnight. By default, tcpdump keeps capturing packets until interrupted by a SIGINT or a SIGTERM signal. If ran with the **-c** flag, it can additionally be stopped after the capture of a certain amount of packets specified at the start. The results can be filtered by using boolean expressions. For example, the host address can be specified to intercept the communication of a particular user or computer. When finished, tcpdump reports the number of captured packets, i.e. the number of received and processed packets, dropped packets and packets received by a filter. The number of packets received by a filter can differ based on the operating system and its configuration. An example of the output can be seen in Listing 2.1.

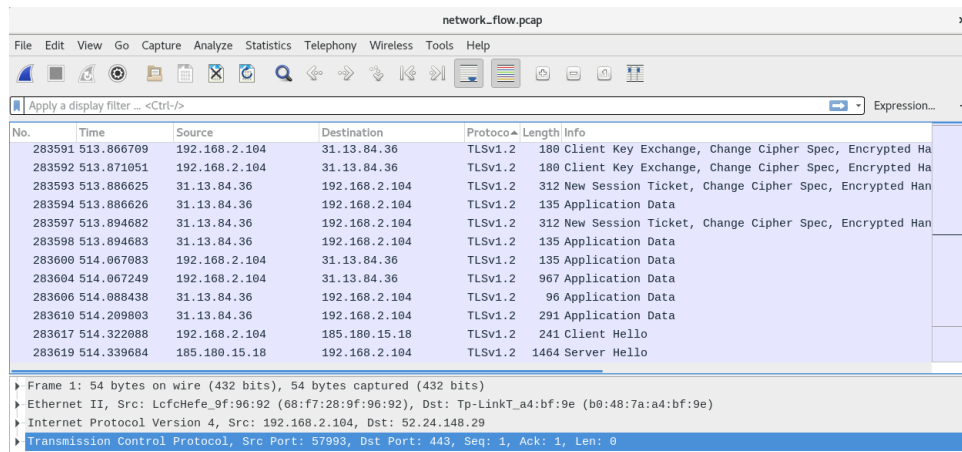
Listing 2.1: Example of tcpdump output (default configuration)

```
22:12:18.605878 IP 151.101.14.49.https > pc-test.47296: Flags [P.], seq
    26923:27111, ack 224, win 63, options [nop,nop,TS val 970823986 ecr
    2196011861], length 188
22:12:18.646818 IP pc-test.47296 > 151.101.14.49.https: Flags [.], ack
    27111, win 1601, options [nop,nop,TS val 2196011921 ecr 970823986],
    length 0
9 packets captured
10 packets received by filter
1 packet dropped by kernel
```

Another widely popular tool is **Wireshark** [7]. This tool is very similar to tcpdump. Wireshark can both capture packets from a network interface and save them to a pcap file or analyze previously saved data. It also has a lot of integrated filtering options. The main difference between these two tools is the presence of a graphical user interface (GUI) in Wireshark, illustrated in Figure 2.1 The GUI enables convenient analysis of captured data by filtering or sorting the packets. Wireshark lets the user put network interface controllers into promiscuous mode, which enables them to capture all the network traffic on those controllers, including broadcast and multi-

2. NETWORK TRAFFIC MONITORING

cast. Nonetheless, when capturing packets on a particular switch port, not all the traffic needs to flow through that port. To gather packets from all the ports, port mirroring can be used. Wireshark also has a terminal-based version called **TShark**, which is a very similar tool to tcpdump. Without any options set, TShark works very much like tcpdump. As shown in Listing 2.2, TShark's output with a default configuration returns similar information, including source and destination addresses, protocol, timestamp and size of the packet.



No.	Time	Source	Destination	Protocol	Length	Info
283591	513.866709	192.168.2.104	31.13.84.36	TLSv1.2	180	Client Key Exchange, Change Cipher Spec, Encrypted Handshake
283592	513.871051	192.168.2.104	31.13.84.36	TLSv1.2	180	Client Key Exchange, Change Cipher Spec, Encrypted Handshake
283593	513.886625	31.13.84.36	192.168.2.104	TLSv1.2	312	New Session Ticket, Change Cipher Spec, Encrypted Handshake
283594	513.886626	31.13.84.36	192.168.2.104	TLSv1.2	135	Application Data
283597	513.894682	31.13.84.36	192.168.2.104	TLSv1.2	312	New Session Ticket, Change Cipher Spec, Encrypted Handshake
283598	513.894683	31.13.84.36	192.168.2.104	TLSv1.2	135	Application Data
283600	514.067083	192.168.2.104	31.13.84.36	TLSv1.2	135	Application Data
283604	514.067249	192.168.2.104	31.13.84.36	TLSv1.2	967	Application Data
283606	514.088438	31.13.84.36	192.168.2.104	TLSv1.2	96	Application Data
283610	514.209803	31.13.84.36	192.168.2.104	TLSv1.2	291	Application Data
283617	514.322088	192.168.2.104	185.180.15.18	TLSv1.2	241	Client Hello
283619	514.339684	185.180.15.18	192.168.2.104	TLSv1.2	1464	Server Hello

Frame 1: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
Ethernet II, Src: LcfChefe_9f:96:92 (68:f7:28:9f:96:92), Dst: Tp-LinkT_a4:bf:9e (b0:48:7a:a4:bf:9e)
Internet Protocol Version 4, Src: 192.168.2.104, Dst: 52.24.148.29
Transmission Control Protocol, Src Port: 57993, Dst Port: 443, Seq: 1, Ack: 1, Len: 0

Figure 2.1: Illustration of the Wireshark's GUI

Listing 2.2: Example of TShark output (default configuration)

```
92 2.512156872 10.0.2.81 -> 74.125.140.155 TCP 66 46870 -> 443 [ACK] Seq=1
Ack=1 Win=289 Len=0 TSval=2336250497 TSecr=3109473457
93 2.536758551 2.21.74.17 -> 10.0.2.81 TCP 66 [TCP ACKed unseen segment] 80
-> 46372 [ACK] Seq=1 Ack=2 Win=1214 Len=0 TSval=4005821563 TSecr
=3259301583
24 packets captured
```

2.2.2 Drawbacks of DPI

Although DPI has been a significant part of network monitoring for years, the invasive form of packet inspection causes discussions about its appropriateness in the modern Internet. Apart from the positive effects of better network security and the ability to regulate unwanted traffic, DPI can also be used for ill-intentioned operations, e.g. Internet surveillance and censorship [8]. A recent change in the US legislation allows Internet providers such as AT&T and Verizon to collect information about users and sell them to a

third party [9]. With the access to the data part of the packet, this increases the possibility of leaks of private personal information or other sensitive data. In some cases, it is even possible to retrieve the entire content of an email just by using DPI [10].

Moreover, the amount of analyzed data increases rapidly when examining whole packets. Combined with the speed of today's networks, this makes DPI very resources and time-consuming. The approach, proposed by Cong et al. [11], suggests that to keep a high performance of DPI platforms, all levels of the platform design, i.e. operating system, applications and hardware, need to take full advantage of parallelism. As the DPI typically analyses many connections simultaneously, the connection based parallelism is considered to be the best way of ensuring the parallelism at the hardware level.

2.3 Flow Monitoring

Flow monitoring has been designed as a scalable solution for monitoring of high-speed networks. It typically inspects only the packet header without the payload itself. This approach overcomes the two major issues of DPI described in Section 2.2.2: privacy and performance. Since the data transferred by the packet are not analyzed, the amount of information that needs to be processed is greatly reduced. Furthermore, the privacy of the payload is not violated, and it is also possible to examine encrypted packets. Instead, the packets are aggregated into a flow, which is the main abstraction of flow monitoring. A flow is defined in [12] as *a set of packets or frames passing an observation point in the network during a certain time interval. All packets belonging to a particular Flow have a set of common properties*. However, data from the application layer can be added to flows, increasing the amount of the gained information about the communication. An example of a network flow is shown in Table 2.1. Flows only contain packets coming in one direction, meaning that communication between two network nodes has to be covered by two flows. Flows were introduced by Cisco Systems company with their NetFlow protocol [13].

Oct	Pack	Prot	SrcIP	DstIP	SrcPort	DstPort	FlowStart	FlowEnd
1045	9	6	127.8.115.27	166.211.112.66	44207	80	1481796997998	1481797013131
5154	32	6	127.8.123.13	44.200.232.22	52558	443	1481796712743	1481797003126
72	1	17	127.8.115.105	244.73.199.233	59616	53	1481797020059	1481797020059

Table 2.1: Example of a typical network flow.

2. NETWORK TRAFFIC MONITORING

Typical use cases of NetFlow are network accounting, network monitoring, and congestion detection. The architecture of the system using NetFlow consists of three main components, as illustrated in Figure 2.2. The flow exporter captures packets entering the interface, transforms them into flows and sends them to a number of collectors. The flow collector, usually a server, does the actual analysis of collected flows. The analysis application then processes flows and detects the specified events. Due to the fact that Cisco is also the main manufacturer of most network devices, the NetFlow protocol is supported by vast majority of network devices.

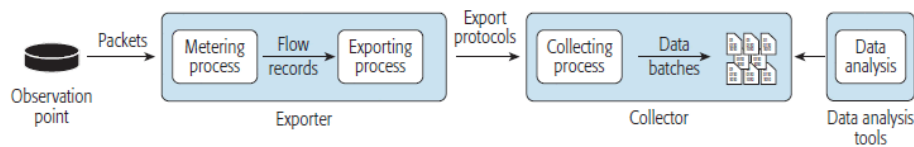


Figure 2.2: Architecture of typical NetFlow setup [14].

2.3.1 IP Flow Information Export

IP Flow Information Export (IPFIX) protocol, defined in RFC 5103 and RFC 7011 through RFC 7015, was created to standardize the export of network flows. It is based on the NetFlow protocol in version 9. The IPFIX protocol defines the process of formatting and transmitting of flows from an exporter to a collector [15]. Each message contains a header, which has the following properties:

Version Number	Length	Export Time	Sequence Number	Observation Domain ID
----------------	--------	-------------	-----------------	-----------------------

The Sequence Number determines the order of messages in the flow. The Observation Domain ID identifies the set of observation points that send the messages. It is unique in the given exporting process.

The IPFIX protocol has been designed to be independent of transport protocol layer [12]. The default transport protocol in IPFIX is the Stream Control Transmission Protocol (SCTP), defined in [16]. SCTP combines features of both TCP and UDP. The packets are managed sequentially, just like in TCP, but the individual packets are separately identifiable as with UDP. SCTP also provides congestion control. The added functionality includes multi-homed paths which improve the error correction and reliability. IPFIX also supports TCP and UDP.

2.3.2 Flow Monitoring Setup and Process

The flow monitoring process can be broken down into four stages [17], first of which is an **Observation Stage**. In this stage, all the packets are captured in an observation point and pre-processed. Packets are typically captured by a Network Interface Card (NIC) and then timestamped. However, most of NICs are not capable of timestamping. Therefore, this task is usually performed by a software. After all the packets are captured and timestamped, they can be truncated or filtered.

The next stage is **Flow Metering & Export**. The Observation and Flow Metering & Export stages are often carried out by a single device called flow exporter. In the Metering process, the packets are aggregated into flows and stored in a flow cache. The flow is terminated under one of the following conditions:

- expiry of a pre-defined time interval, in which no other packets arrive,
- exceeding a maximal flow length,
- observation of a TCP packet with a FIN or RST flag,
- overflow of the flow cache,
- flow cache flush after an unexpected event.

After the flow is terminated, it is encapsulated in a message and sent to a collector, whereby the third stage begins.

The **Data Collection** stage's task is to receive messages from one or more exporters. Messages are further pre-processed and aggregated, and possibly filtered or anonymized. Processed messages are then saved to a file or to a database. Row-oriented databases, such as MySQL or PostgreSQL, are not suitable for flow storage since query answering often needs only a part of each row [17]. On the other hand, column-oriented databases access only given columns of each row, decreasing the overhead of information retrieval. The most notable advantages of file storage are low disk space usage and fast reading and writing capabilities, but the query capabilities are very narrow.

The final stage, **Data Analysis**, is where all the data are gathered to perform the desired analysis. This stage is sometimes integrated into the Data Collection stage and conducted automatically. The three main use cases of data analysis are *flow analysis and reporting*, *threat detection* and *performance monitoring*. Flow analysis and reporting brings a simple overview of the network and provides the ability to browse and filter the data, create statistics

and report about events in the network. If the collected flows are used for threat detection, they can either be used for a simple analysis of the network communication or analysis of the network behavior compared to certain models of threats. Performance monitoring provides an overview of the status of running services, such as delay, jitter or bandwidth usage.

2.4 Stream4Flow Framework

Stream4Flow (*S4F*) is a framework designed for rapid prototyping of IP flow analysis [18, 19]. This project is developed by Institute of Computer Science Masaryk University and supported by the Technology Agency of the Czech Republic. It's functionality consists of 4 main components as illustrated in Figure 2.3. The first component is an **IPFIXcol** collector [20]. IPFIXcol receives flows from IP flow exporters such as IPFIX (by default), NetFlow or sFlow, modifies them and then sends them to another network node. It can also save the result to a file. In *S4F*, IPFIXcol transforms the flows into the JSON format with a possibility to anonymize or filter the flows. The data are then received by the messaging system.

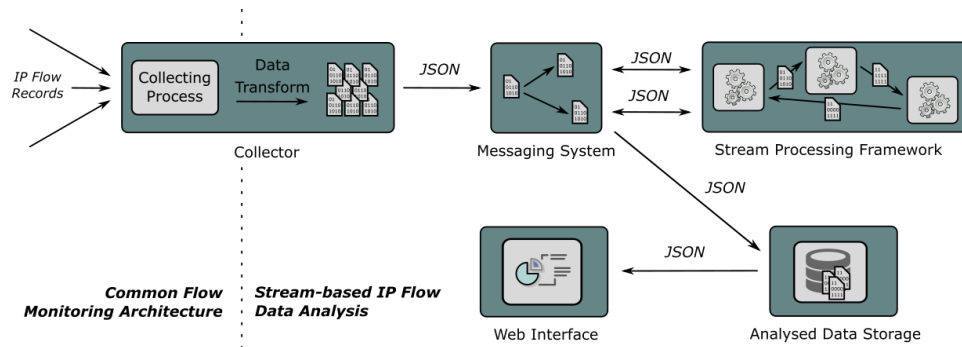


Figure 2.3: Stream4Flow framework architecture [18].

The **Apache Kafka** framework [21] is a distributed publish-subscribe system for message delivery. The data are transformed into streams in a client called Kafka Producer. Streams are divided into categories called *topics*, that consist of one or more partitions. The basic abstraction in Kafka is a message, a 3-tuple consisting of a key, a value, and a timestamp. The streams created by the producer are received by the Kafka Consumer. Each consumer can read just the selected topic, with a possibility of multiple consumers reading a single topic. The data provided by the IPFIXcol collector are transformed into streams and made available for a stream processing framework.

2.4.1 Stream Processing Framework

Stream processing allows for a real-time data manipulation [14], unlike batch processing, that first needs to collect all the data, save it to a database and then produce the desired output. Stream processing requires a different approach to data utilization than the one used in batch processing. The data coming in are available only at that particular moment, and all have to be processed right away. Hence the old data cannot be accessed after they have been transmitted.

As the processing of each entry separately is not as efficient in network monitoring, **sliding windows** can be used. A sliding window sets a time frame on the input data and allows to manipulate all the data sent in that period, including old data. It enables to detect attacks that would otherwise happen in multiple separate chunks of data and thus would not be detected. The windows can be set to move in smaller time intervals to reduce the time between an event, and its detection. For example, the window can be set to 5 minutes with a sliding interval set to 5 seconds, allowing close to real-time detection with a big enough block of processed data to have perspective on ongoing events.

As demonstrated in the article [22], today's stream processing frameworks are capable of analyzing high-speed networks. The Apache Samza [23] framework turned out to have the highest throughput when performing regular operations on network data. However, Samza requires the number of partitions to be set to correspond the number of processor cores. In the case of the S4F framework, Apache Spark has been chosen.

2.4.2 Apache Spark

Apache Spark is an open-source cluster-computing framework [24]. A single block of data in Spark is called a **microbatch**, which is a portion of data received in a small time frame (5 seconds for example). That means data are not processed in real-time in Spark since there is a small delay. However, the delay can be as short as 1 second, which makes it way closer to real-time analysis than batch processing. Spark can be deployed either using software such as Hadoop or in a standalone mode. In the standalone mode, Spark has to be installed on each node and then split into one master node and multiple worker nodes. The master node manages all the running applications and distributes tasks between worker nodes.

The basic abstraction used in Spark is a **resilient distributed dataset** (RDD). RDD is a collection of objects distributed over one or more cluster

nodes and can be recovered in case of a failure. The fault tolerance is accomplished by preserving the sequence of operations that created the RDD. The stream processing functionality is carried out by Spark Streaming library. The data coming into Spark Streaming from one source are formed to a structure called **discretized stream** (DStream). DStream is a continuous series of RDDs. The whole process of stream processing is illustrated in Figure 2.4.



Figure 2.4: Overview of stream processing in Spark [24].

DStreams can be transformed in parallel using **MapReduce** operations. MapReduce is a programming model for processing and generating big data sets with distributed algorithms [25]. Two main methods in MapReduce are *map*, which is used to apply a function to all elements of a sequence, and *reduce*, which performs a summary operation (i.e. aggregation of flows based on IP address). Spark functions can be divided into two groups: **transformations** and **actions**. Transformation functions take an RDD or a DStream, transform it and then return the altered input. Action functions take input data and do a computation over them, such as a sum of all entries having a particular IP address. Transformations use lazy evaluation, e.g. they do not perform the operation until an action function is called.

Spark also has a web interface on the master node. It enables more convenient management of the cluster. The overview lists all worker nodes and running applications with basic information about the setup environment such as its status or usage of the available resources. The main page of the interface is shown in Figure 2.5. The interface also offers detailed information about running applications. Besides tabs with configuration and a list of cached RDDs, the interface enables comprehensive analysis of the application's tasks. The *Jobs* tab lists all of the processed microbatches with a timestamp and the duration of the batch processing. The *Streaming* tab visualizes course of the input rate, processing time and the delay over the time the application was running.

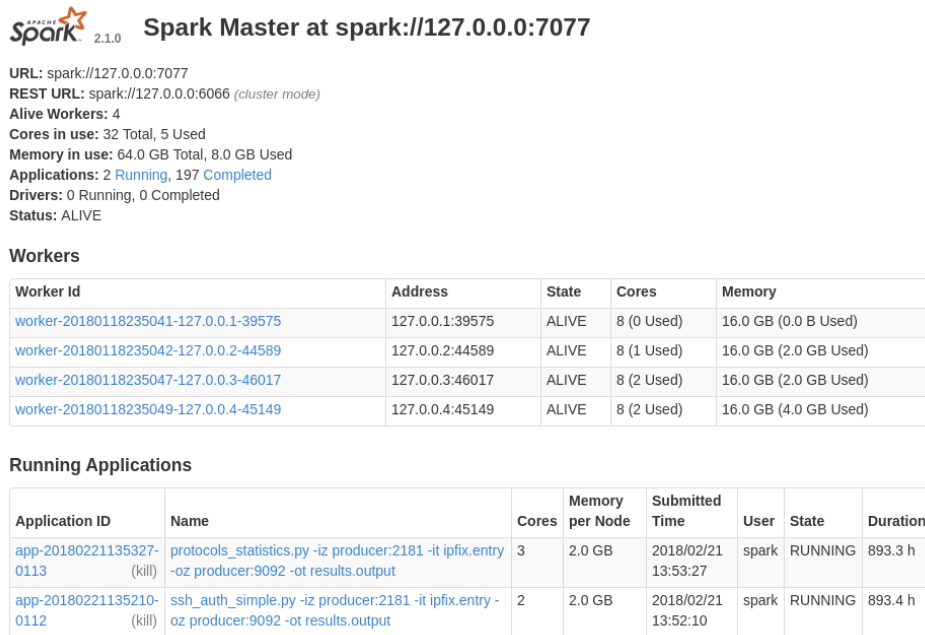


Figure 2.5: Apache Spark web interface.

2.4.3 Analyzed Data Storage

The last component of the Stream4Flow framework is an analyzed data storage. The data processed by the stream processing framework are stored in the JSON format using NoSQL database. The engine used for data manipulation, searching and visualization is **Elastic Stack** [26]. The three main tools in Elastic Stack are Logstash, Elasticsearch, and Kibana. Logstash is a processing pipeline designed to collect, process and forward events and logs. It can receive data from multiple sources simultaneously (with a wide variety of supported formats). The input is transformed into a same format to increase the efficiency of the output reading. Additional information can be added to the data using plug-ins. In S4F, Logstash receives data from Kafka that were processed by Spark and sends them to Elasticsearch.

Elasticsearch is a highly scalable, distributed, full-text, near real-time search engine based on Apache Lucene [27]. Each entry stored by Elasticsearch is represented by a single document. The scalability is achieved by the way the cluster adds new nodes as the amount of handled data increases. When a new node is added to the cluster, the data are redistributed to balance the workload and to maximize the availability. All the incoming JSON

2. NETWORK TRAFFIC MONITORING

entries are automatically parsed and indexed. The engine also determines the data type of all parsed entries. The Elasticsearch itself has only a simple plain-text web interface.

The last component of Elastic Stack is Kibana, a tool enabling visualization of the Elasticsearch data. The graphical outputs include different types of charts, diagrams, tables or geographic maps. These visualizations can be generated from an existing query or by creating a new query and then saved for later use. All the created visualizations can be added to a dashboard, where all graphs can be changed with a change in a single graph. Dashboards can be saved and shared, enabling the administrator to set up the dashboard and share it to people with no knowledge of Kibana. Kibana is basically a web interface extension for Elasticsearch.

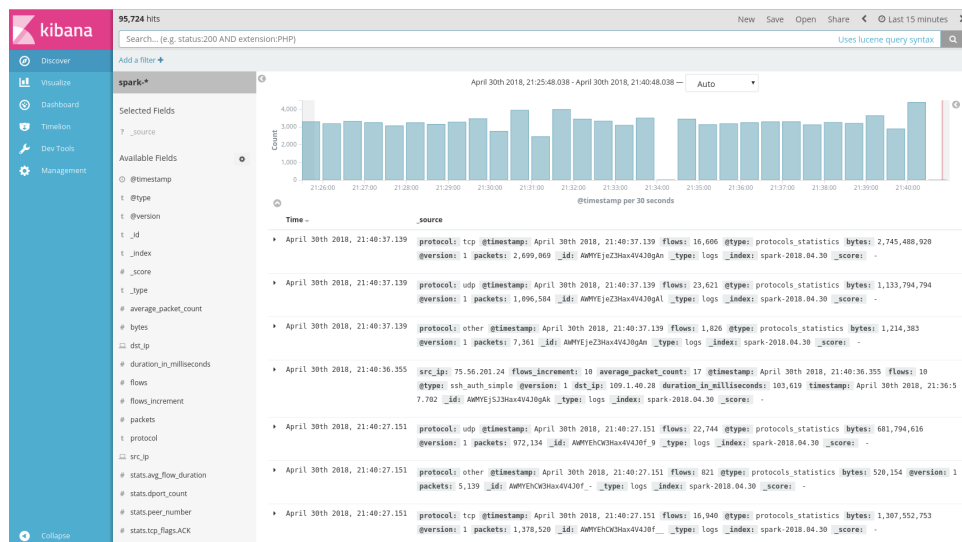


Figure 2.6: Kibana web interface.

2.4.4 Web Interface

The Stream4Flow framework also has its web interface. Besides applications for detection of network attacks, the framework also provides statistical applications, namely Protocol Statistics and Host Statistics. The Protocol Statistics application collects the number of flows, packets and the number of bytes transferred by TCP, UDP and other protocols. The Host Statistics application collects information about the communication of each host in the network. Figure 2.7 shows the visualization of the Protocol Statistics application

2. NETWORK TRAFFIC MONITORING

in the interface. As one of the goals of this thesis is to provide an interface for the application for detection of blacklisted network hosts, it was necessary to understand how the interface is implemented. The framework used for the interface is the **Web2Py** framework [28], which is a full-stack open source framework for a development of web-based applications in Python.

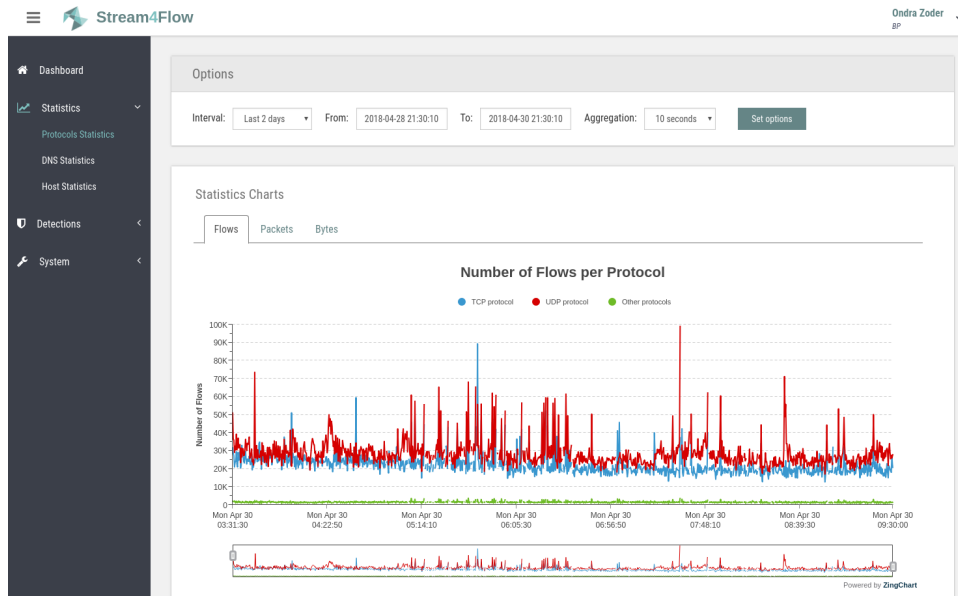


Figure 2.7: Stream4Flow web interface.

3 Blacklists

A blacklist is a registry of elements to be blocked from access to a particular system. It is a common access control mechanism that is used at different points of a system's architecture and can contain various types of elements. Blacklists are commonly used in corporate or university networks as a mechanism for blocking of undesirable websites or in systems with a database of users, where administrators want to prevent users from having offensive nicknames or weak passwords. One of the typical network elements stored in a blacklist is an IP address. When a defense mechanism of some information system detects a suspicious behavior of a particular network host or when this host conducts criminal activities, the host's IP address is stored in a database. This database then helps to preserve the safety of this system by preventing any communication with the listed hosts. The opposite of a blacklist is a whitelist, which allows access only to those elements that are listed in it. A more complex type of blacklist, where all elements are listed only until further steps take place, is called a greylist. For example, when using greylists, an e-mail from an unknown address might get temporarily blocked until the message has been confirmed to be spam-free. These three lists are often combined to increase the efficiency of the access control mechanism.

Access control is defined by Shirey [29], as a *process by which use of system resources is regulated according to a security policy and is permitted only by authorized entities (users, programs, processes, or other systems) according to that policy*. There are multiple approaches to implementing the access control, and most of them are based on whitelists, i.e. only those entities mentioned can grant the access [30]. One of the popular approaches is using Access Control Lists (ACLs). In this approach, each object in the system has an ACL attached to it. The ACL contains a list of access rights for each of the system's entities. When an entity requests access to a particular object, a file, for example, the system checks the ACL attached to the file and goes through all the access rights set for this entity. If the entity has the corresponding right, the system grants the permission for the requested action, e.g. writing in the file.

3.1 Blacklist Creation Process

The whole process of an undesirable communication detection is an important issue in the Internet security and may be implemented in many different ways. Since the aim of this thesis is not to provide a new approach to the

3. BLACKLISTS

detection or a thorough description of existing techniques, only a few progressive ways of detection will be presented. The first approach, proposed by Felegyhazi et al. [31], explores the potential of proactive blacklisting of domain names. The main idea behind this approach is that domains dedicated to Internet scams are often registered together in large quantities and operated using related name servers. The proposed method needs one or more of these domains, called "seeds", to be listed in a basic domain name blacklist. Based on these seeds, all the domains registered together get blacklisted before even getting active in malicious activities. When tested on real Internet data, nearly 73% of domain names detected by this method eventually appeared on blacklists, with an average delay of 2 days. Another 20% of the detected domains were assessed as suspicious by third-party services used for potential misuse predictions.

Another method, proposed by Ma et al. [32], automatically analyzes lexical and host-based properties of website Uniform Resource Locators (URLs). This approach uses machine learning for the automatic selection of the URL's properties. It can select features that would otherwise not be selected in a manual analysis by domain experts. The method proved to be accurate, as 95% of the website URLs classified as malicious eventually appeared on blacklists, with a minimal number of false-positive classifications. The significant advantage of this approach is the low resources requirements since it need not analyze the content of the website itself or the content of the e-mail message that contained the URL. A similar approach, proposed by Chiba et al. [33], analyses features of IP addresses, also by using machine learning techniques. This method is based on two of the characteristics of IP addresses. First, IP addresses are stable compared to DNS and URLs, meaning that it is much more expensive for attackers to change IP addresses than to change URLs and DNS. Second, the IP addresses used by attackers are likely to be in same network address spaces. Using these two characteristics, network devices can get blacklisted proactively.

Besides techniques for the actual detection of addresses to be blocked, there are also many methods to improve the efficiency of the blacklist use. For example, a method, proposed by Yamada and Goto [34], can be used to decrease the amount of analyzed traffic. It suggests that malicious packet can get detected based solely on its time-to-live (TTL) value. Typically, an IP packet needs less than 30 hops (one pass through a network device, e.g. a router, a bridge or a gateway) to travel from a source device to the destination. Packets that are generated by a special software tend to have abnormal TTL values, decreased by more than 30 increments from the initial TTL value. The method assumes that these packets are likely to be malicious. Re-

sults show that a group of packets with abnormal TTL values includes 50 times more malicious packets than the group with normal TTL values. That means that the amount of traffic that needs to be analyzed for malicious content can be reduced.

3.2 Blacklist Distribution Process

Besides the process of the detection of new blacklist entries, an important part of the blacklists usage is also the distribution of the created blacklists and maintenance of the established blacklist databases. One of the popular tools used for blacklist distribution is the **Malware Information Sharing Platform** (MISP) project [35]. MISP is a platform used for storing and sharing of information about network incidents, malware samples, and attackers. The stored elements can be compared to express threat intelligence and connected incidents. The built-in sharing functionality offers different models of distribution that can be adjusted to meet each organization's sharing policy. A similar project developed by the CESNET organization is the **Warden** project [36]. Warden served as the base platform for the **SABU** project [37], which aims to provide a system for distribution and analysis of security incidents. The analysis of incidents could predict the possible progress of attacks and prepare countermeasures against these attacks in participating organizations.

Another widely used technique for the distribution of blacklist is Domain Name System-based Blackhole List (DNSBL). DNSBL is a mechanism for sharing of blacklists over the Domain Name System. A typical DNSBL blacklist usually contains IP addresses of network hosts with a bad reputation. Established DNSBL blacklists vary in the criteria used for listing, but these criteria should always be transparent, easily available and there should be a way to request removal [38]. The blacklists are often split into multiple lists based on the reason for listing so that they can be used both for complete blocking of listed addresses or for simple filtering based on selected criteria. Each entry of a DNSBL storing IP addresses contains the IP address, a corresponding DNS entry and possibly the reason for the listing [39]. The name of the entry is created by a connection of reversed IP address and a domain name. For example, if the IP address is *192.15.120.13* and the domain name is *example.domain.com*, the name would be *13.120.15.192.example.domain.com*. When a connection is established over DNS, the address is transformed into the specified format and compared to existing DNSBLs. When a match is found, it means that the address is listed in a blacklist.

3.3 Common Use of Blacklists

The *blacklist* is a rather broad term, and tools and mechanisms based on its principles have a fairly wide scope of application. A simple example might be its common use in all places where groups of people interact with each other. For example, social networks usually enable users to block other users by adding them to a blacklist. One of the more complex network services using blacklists is Google's **Safe Browsing** [40]. This service collects URL addresses that were detected containing malware or contributing to phishing activities. Many popular web browsers, including Google's Chrome, use these lists of malicious websites to protect users from potential threats. Safe Browsing also notifies internet service providers about the presence of potentially dangerous hosts in their networks. Microsoft offers a similar service called **SmartScreen** [41], which is integrated into Windows 8 and Windows 10 operating systems and Microsoft Edge browser. Both whitelists and blacklists can also be used in antivirus software. The **Kaspersky Antivirus Software** uses blacklisting to collect all types of malware by adding the signature of a new malware that infected a computer connected to the Kaspersky Security Network (KSN) [42]. Kaspersky also combines this technique with whitelisting in a technology called Trusted Applications. This technology tests all the data collected from the KSN and creates a list of trusted applications.

3.3.1 Anti-spam Filters

Another area where blacklisting plays a relatively important role is anti-spam filters in e-mail clients. Many techniques can be used to prevent email spam, two of which employ blacklisting: **DNSBL** and **greylisting**. The DNSBL distribution mechanism, described in Section 3.2, provides easy access to many established blacklists, enabling administrators to use these blacklists in their e-mail anti-spam filters. A DNSBL blacklist usually contains IP addresses of machines sending the spam, but also Internet Services providers (ISPs) hosting the spammers and some DNSBLs contain domain names. As e-mail spam messages often include links to malicious or phishing websites, the body of the spam message is also analyzed, and the links are listed in Uniform Resource Identifier (URI) DNSBLs. Therefore, e-mail spam sent by legitimate addresses that would otherwise not be detected is blocked.

Greylisting is a method where unknown or suspicious sources get a degraded service over a certain period, usually until some further steps take place. The more narrow use of this term refers to a technique used for filter-

ing of incoming messages used in e-mail servers [43]. Messages sent from an unknown or suspicious source are temporarily rejected. If the message is legitimate, the originating source will try to send it again, and if the pre-defined period has elapsed, the receiver will accept the latter message. As defined by Klensin [44], a fully implemented Simple Mail Transfer Protocol (SMTP) maintains a queue of message transmissions that ended in a temporary error and resends these messages. After the sender is proven to be a legitimate source, the receiving server adds its address to a whitelist, so that the following messages from this source are not rechecked. As the aim of a spamware is the volume of the distributed messages rather than successful delivery, tools used for spam distribution usually do not attempt to resend the rejected message. The basic implementation of greylisting only checks whether the IP address of the sender is in the whitelist, however, more comprehensive analysis can be performed on the message's envelope. Most greylisting systems also allow for the use of exceptions. Large e-mail servers usually use many machines to send the messages, and these machines typically use same IP address Classless Inter-Domain Routing (CIDR) blocks. These blocks can be excluded from the greylisting checks, avoiding analysis of each machine in originating server. Single IP addresses, host-names and domain names can also get exceptions.

The main advantage of greylisting is arguably the easy setup. Once the administrator configures the server appropriately, users do not need to take care of anything. Another significant advantage is that from the server's perspective, greylisting does not require many resources since SMTP's temporary errors are quite cheap to handle. The main disadvantage of greylisting is the delayed delivery. Every time a user receives a message from an unrecognized server, the delivery usually takes a few minutes, even more, if the server is incorrectly configured. As the communication via e-mail is usually near-instantaneous, this delay is considerable. One of the possible problems that may occur is also a misinterpretation of the temporary error by the sender. SMTP server implementing an obsolete specification, such as the one in [45], may construe the temporary error as a permanent failure.

3.4 Established Blacklist Databases

There are many well-established blacklist databases on the Internet with different approaches to detecting new entries. Many of these databases are free for non-commercial use. Generally, these databases can be divided into two categories: **IP address databases** and **domain name databases**. One of the well-known IP address databases is the FireHOL IP lists database [46]. It

3. BLACKLISTS

is a collection of multiple IP blacklists with a goal of providing universal IP blacklist that can be used on all kinds of systems. The main prerequisite to achieving its versatility is to eliminate any false positive instances. Every time one of the IP blacklists gets changed, the database updates. One of the blacklists used in this database is the Don't Route Or Peer (DROP) Lists by The Spamhaus Project. The Spamhaus Project [47] is an organization that provides multiple blacklists tracking spam, phishing, botnets, and malware. Besides the DROP blacklist, there is also the Domain Block List (DBL) database of spammer domains or the Exploits Block List (XBL) database of IP addresses of devices infected by third-party exploits.

In the analysis conducted by Spring and Metcalf [48], 86 Internet blacklists are compared to determine the uniqueness of each database and to see if there is any pattern in which same entries are added to different databases. The results show that domain name databases are unique to other databases between 96.16% and 97.37% of the time and IP address databases between 82.46% and 95.24% of the time. These numbers suggest that most of the databases use different detection strategies or collect entries involved in different types of malicious activities. In addition, few databases consistently add entries before others in case of an intersection. This comparison implies that the more blacklists are used in defense mechanisms, the higher is the chance of detecting communication with malicious hosts. However, the analysis conducted by Kühner et al. [49] shows that public malware blacklists include less than 20% of the real-world malware domains.

3.4.1 Databases Used in This Thesis

This thesis focuses on three blacklist databases. The first requirement in the selection process was to have a diversity in the types of entries listed in the databases. Hence the databases selected contain domain names, IP addresses and a combination of IP addresses and ports. Another requirement was to have the possibility to download the blacklist in a text file so that the data can be easily parsed and used. The final selection contains blacklists from following databases:

- malwaredomains.com
- myip.ms
- sslbl.abuse.ch

The malwaredomains.com [50] website is a project that creates and maintains a list of domains known to be involved with malware and spyware. It

is a collection of multiple blacklist databases modified into a uniform file. The default format of the blacklist is a simple text file, where each line contains either a comment (beginning with the # character) or the entry. Entries contain three main values: the domain name, reason for the listing and the database where this entry was originally detected. The basic structure is shown in Table 3.1. This database was chosen because of its size and popularity. The number of entries varies, but it is usually between 15,000 and 20,000 and the database updates approximately once a month.

Domain Name	Type	Original Reference
kalingadentalcare.com	phishing	openphish.com
ziiiraaatbank.com	suspicious	spamhaus.org
secusa.com	virut	private
e2bworld.com	attackpage	safebrowsing.google.com

Table 3.1: Illustration of the malwaredomains.com blacklist structure.

The myip.ms website [51] is a multi-purpose database and search platform for IP addresses. It enables users to find information about a specific IP address, e.g. where it is located, which operational system it is using, etc. Besides that, it collects a real-time blacklist of IP addresses acquired from multiple websites operated by myip.ms. If one of the websites detects a spam bot or a web crawler (bot used for website browsing), the address of this bot is added to the blacklist. Each entry of the blacklist contains the IP address, the date the bot was detected, bot's host address, the country from which it is operating and the type of the bot, represented by a number. These numbers are described in a separate list. This database was chosen mainly because of the option to download it in a text file. The basic structure is shown in Table 3.2. The number of entries is usually around 4,000 and the database updates every time a new bot is detected.

IP Address	Date	Host Address	Country ID	Type
2.93.134.30	2018-02-19	2.93.134.30	RUS	2
91.135.242.203	2018-02-17	91.135.242.203	AZE	2
113.172.64.43	2018-02-16	static.vnpt.vn	VNM	1
195.83.242.250	2018-02-16	195.83.242.250	FRA	18

Table 3.2: Illustration of the myip.ms blacklist structure.

3. BLACKLISTS

The abuse.ch website [52] is a project against malware, operating a Secure Sockets Layer (SSL) blacklist. It provides a list of malicious SSL certificates associated with malware or botnet activities. The detection relies on SHA1 fingerprints of bad SSL certificates and the database is updated every 15 minutes. Besides lists of detected fingerprints, it also offers lists of IP addresses associated with the malicious SSL certificates. The implementation uses an aggressive version of the IP blacklist which contains every IP ever detected, i.e. it might contain a lot of false-positive entries. Each entry contains an IP address, a port and the reason for listing. This blacklist database was chosen both because of the unique combination of an IP address and a port and because of its popularity amongst security companies including the Cern science center. The basic structure of the blacklist is shown in Table 3.2. The blacklist usually contains around 4,000 entries and is updated every 15 minutes.

IP Address	Port	Reason
103.193.4.131	80	Gootkit C&C
161.10.212.151	443	Quakbot C&C
194.68.59.32	2323	Adwind C&C
95.46.99.21	443	Ransomware C&C

Table 3.3: Illustration of the sslbl.abuse.ch blacklist structure.

4 Implementation - Detection

This chapter describes the design and implementation of an application for detection of communication with blacklisted network hosts that would be a part of the Stream4Flow framework, as well as the visualization of the output in the web interface. The two main issues of the application design were the loading of blacklist data and the detection of an occurrence of these data in the received network flows. These two issues are discussed separately in Sections 4.2 and 4.3. In order to understand the results of tests conducted in these two sections, the methodology of testing is described in Section 4.1. The implementation of the visualization is described in Section 4.4. As the network traffic monitoring is an ongoing process and the detection scripts in Stream4Flow are designed to be able to run for long period continuously, one of the main requirements for the application was its stability. That includes the use of sufficient computing resources, as well as the optimization of the application parameters. These parameters were tested to determine the ideal values ensuring a balance between stability and delay of the detection. Results of these tests are presented in Section 4.5. Electronic attachments include all parts of the implementation, as well as the dataset used for testing and a script for test automation.

4.1 Performance Testing Methodology

In order to maximize the efficiency of the implemented application, all proposed approaches were tested on a dataset as well as the real anonymized data from Masaryk University network. The computing resources available for the testing consisted of 32 processing cores and 64GB of primary storage. However, all the performed tests used only a part of these resources, as the number of processed network flows was not as high as this amount of computing resources would handle. The used numbers vary from 4 cores and 2GB of memory to 8 cores and 4GB. As the Stream4Flow framework operates in a distributed environment and uses a stream processing approach for the network flows processing, the testing requires different ways to measure the performance. Both the dataset and the real data measurements were conducted using the Apache Spark web interface. Figure 4.1 illustrates the information the web interface provides about a running application. An important parameter indicating the performance is the **Processing Time** [53]. This parameter shows the average time the application processes each batch. For the application to be stable, the average time should be smaller than the

4. IMPLEMENTATION - DETECTION

length of the microbatch. However, when a window is set on the stream, the average processing time can be longer than the microbatch interval, as long as the whole window of data is processed in time. When the input rate is higher than the application can handle, the **Scheduling Delay** parameter increases. The main goal is always to keep this parameter as low as possible. The efficiency of each approach tested on real data was determined based on the processing time values.

Streaming Statistics

Running batches of 5 seconds for 1 day 1 hour 2 minutes since 2018/04/16 08:11:22 (18028 completed batches, 626726553 records)

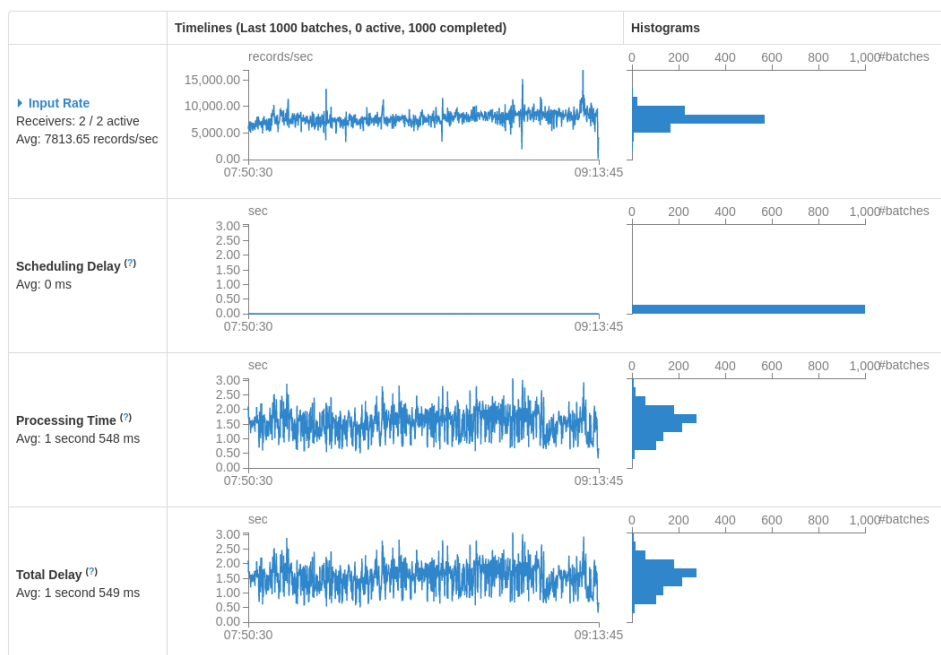


Figure 4.1: Illustration of a streaming tab of a running application in the web interface.

A dataset was used to test the performance as well as the precision of the detection method. Besides measurements of the processing time, the time the application takes to process the whole dataset was also measured. To conduct this measurement, few adjustments needed to be made. In order to determine the end of data processing, the dataset had to be modified, adding a new flow on the end of the dataset. This flow contains an IP address unique to the dataset. The IP address also had to be added to the blacklist so the application can detect it. When the application detects this IP address and sends it to the result stream, the time of processing can be

determined by simple measurement of time between the start of the dataset replay and an occurrence of the last IP address. The lower the time is, the better is the performance of the application. The processing time parameter was also taken into account. Some of the values from the dataset had to be added to the blacklist to determine the precision of the detection method.

Electronic attachments also include a script for automated testing with datasets. The script runs the blacklist parser, replays the dataset, prints the duration of the processing to the console and terminates. Arguments of the script are shown in Table 4.1. The **output_zookeeper** argument specifies an address and a port of the host running Apache Zookeeper. The **output_topic** argument specifies the name of Kafka topic providing the output data. The **blacklist_topic** argument specifies the name of the topic where the blacklist data should be sent. The **last_key** argument specifies the last key that indicates the end of the dataset. The **dataset_path** and **dataset_speed** arguments specify the path to the dataset to be replayed and the speed of the replay, with 1 indicating the real-time speed. The speed is set to 1 by default. The detection application needs to be run separately, with same output and blacklist topics.

Short Flag	Long Flag	Required	Default Value
-oz	-output_zookeeper	True	-
-ot	-output_topic	True	-
-bt	-blacklist_topic	True	-
-lk	-last_key	True	-
-dp	-dataset_path	True	-
-ds	-dataset_speed	False	1 (coefficient)

Table 4.1: Arguments of the testing script.

4.1.1 Dataset Used for Testing

The dataset that was used for testing purposes is the Anonymized Internet Traces 2015 Dataset by Center for Applied Internet Data Analytics (CAIDA) organization [54]. It contains anonymized passive traffic from a backbone link between Chicago and Seattle. Figure 4.2 shows the distribution of flows in the dataset. The input rate is variable, being 2988 records per second on average with a peak rate of around 8000 records per second. The total amount of records in the dataset is 298260. The data were transformed into network

flows and exported into IPFIX format, which is included in electronic attachments. The flow marking the end of the dataset has an IP address set to 240.0.0.3.

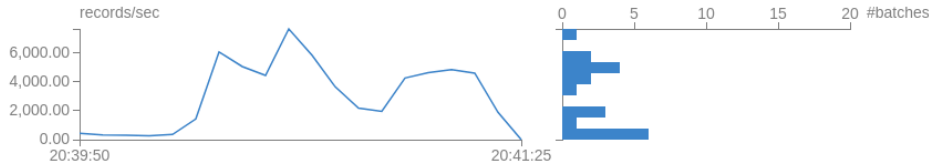


Figure 4.2: Distribution of flows in the dataset.

4.2 Loading of Blacklists

The first issue of the implementation was to determine how to load the blacklist data into Apache Spark data abstraction, either in the form of an RDD or a DStream. Three different approaches were proposed, each described in Sections 4.2.1, 4.2.2 and 4.2.3. All of these three approaches share the same process of the blacklists downloading and parsing and differ in the way the parsed data are loaded into Spark. The blacklists used in the implementation are described in Section 3.4.1.

The designed parser uses a configuration file for the processing. Therefore other blacklists can be added to the application, as long as they can be exported into a text file. An example of blacklist entry in the file is shown in Listing 4.1. The square brackets specify the name of the section. The **URL** parameter specifies the URL address of the blacklist text file. The **Output-File** parameter specifies the path where the blacklist file should be stored. As the blacklist files contain multiple values for each entry separated in some way, the **Separator** parameter specifies the character with which values of each entry are separated. If this parameter is empty, the character is set to space. The parameters **IPAddress**, **URLAddress**, **Port**, **HostAddress** and **ReasonForListing** each specify the index of each blacklist parameter in the separated line. In the case of non-standard formatting of a blacklist file, the **StripChar** parameter specifies a redundant character that should be deleted from the values.

All blacklist entries are parsed into the JSON format with two values: **key** and **description**. The key is determined based on the values each blacklist file contains. If the blacklist contains an URL address, the key is set to this address. If the blacklist contains IP address, the key is set either to the

IP address or to "IPAddress:Port" format if the port is specified. The description contains the ReasonForListing and HostAddress values separated by a semicolon. An example of a parsed blacklist is shown in Listing 4.2

Listing 4.1: Example of a blacklist entry in the config file.

```
[IP_blacklist]
URL = https://myip.ms/files/blacklist/general/latest_blacklist.txt
OutputFile = ./blacklists/download/ip_blacklist.txt
Separator =
IPAddress = 0
URLAddress =
Port =
HostAddress = 3
ReasonForListing = 5
StripChar = ,
```

Listing 4.2: Example of a blacklist parser output.

```
{"@type": "blacklists.entry", "blacklists.key": "insidelocation.ga", "
  blacklists.description": "Reason for listing: phishing"}
{"@type": "blacklists.entry", "blacklists.key": "194.58.96.50:443", "
  blacklists.description": "Reason for listing: KINS C&C"}
{"@type": "blacklists.entry", "blacklists.key": "125.212.205.196:80", "
  blacklists.description": "Reason for listing: Gootkit C&C"}
{"@type": "blacklists.entry", "blacklists.key": "95.218.65.230", "blacklists.
  description": "Host address: 95.218.65.230; Reason for listing: spambot
  "}
```

4.2.1 Loading from a Text File

The first proposed way of loading blacklists into Apache Spark is loading from a text file. In this approach, the blacklist parser processes the blacklist files and saves all the resulted data into a file. There are two options how this file can be loaded: the `textFile()` function, that transforms the data into an RDD and the `textFileStream()` function, that creates a DStream. The `textFileStream` function takes a path to a directory as an argument and monitors this directory to check whether any new files appeared. For the function to repeatedly load the file with blacklist data, the file needs to be different from the last one both in the name and in the content. The `textFile` function has two major drawbacks. The first is that the function only loads the file once and does not register any following changes to the file. The function has to be called every time the file is changed. The second drawback is the fact that the function transforms the data into an RDD rather than a DStream. As the input data are in the form of a DStream, it is more convenient to load the blacklist data as a DStream too. The `textFileStream` function's ma-

major drawback is the need to distribute the blacklist file to all the nodes of the cluster. As the blacklist parser operates on the master node, the file is stored only on this node. The `textFileStream` function is run on all the cluster nodes and needs the file to be stored in the same path on every node. One way to achieve this is to send the file to every node with the Secure Copy Protocol (SCP).

4.2.2 Creating Stream from Python Collection

The second proposed way of blacklist loading is creating it directly from a Python collection. In this approach, the blacklist data are parsed into a list of lines and then, using the `parallelize()` function, this list is transformed into an RDD. The blacklist data can be used in the form of an RDD; however, as stated in Section 4.2.1, the blacklist data should preferably be in the form of a DStream. The `queueStream()` function can be used to create a DStream from an RDD. This function creates a DStream from a queue or a list of RDDs, preserving the order of RDDs. The main advantage of this approach is the fact that the data do not have to be stored or sent anywhere, reducing the overhead. The other advantage is the possibility to accommodate this functionality into the main application, avoiding the need to use two separate applications. However, the main disadvantage that makes this approach unusable is the fact that the `queueStream` function creates the stream when called, but does not update the stream once in each microbatch interval. That means that the function has to be repeatedly called to reflect changes in the blacklist data.

4.2.3 Sending Through Kafka

The third proposed way of blacklist loading is sending it through a Kafka topic. In this approach, the blacklist data are parsed into a string and then sent by a Kafka Producer into a specified Kafka topic. As the Kafka Producer has a limit on the amount of data in one sent message, the parsed data have to be sent in multiple messages. There are two possible ways to split the message. The first approach is to send each line of the blacklist separately. However, this approach is relatively inefficient, as the blacklist may have tens of thousands of lines and the sending function would have to be called each line. The second approach is to send larger portions of blacklist data, e.g. portions of 5000 lines. The problem with this approach is that the sent data are not split by the Kafka Consumer client and have to be additionally split using the `flatMap` function in Spark. The second approach has been

implemented and tested. The main advantage of using Kafka for blacklist loading is the ability to update the stream once every microbatch interval. The other significant advantage is the fact that the stream of network flows is delivered by Kafka as well, which means the streams loading would be uniform.

4.2.4 Conclusion

All of these three approaches were measured in terms of time efficiency to determine which one of them is the most suitable. In every approach, the time when the application starts and when it finished was measured to observe the time it takes to load the blacklist data into Spark. The chart in Figure 4.3 shows the comparison. As the process of the blacklist downloading and parsing was generally the most time-consuming part, the chart differentiates this part from the process of distribution. The measurements show that the approach using Kafka was the most efficient approach performance-wise, but only by a small margin. The approach using text file proved to be the slowest of the three mainly because of the distribution of the final blacklist to all nodes of the cluster using SCP. The important factor in the decision was also the fact that the function `queueStream` used in the second approach does not update the stream once every microbatch as the other two approaches do. Considering this drawback and the poor time efficiency of the first approach, the third approach has been chosen as the most suitable for the blacklist parser implementation.

The final blacklist parser script is part of electronic attachments. Arguments of the script are shown in Table 4.2. The **output_zookeeper** argument specifies an address and a port of the host running Apache Zookeeper [55], which ensures the coordination and synchronization of the distributed applications. The **output_topic** argument specifies the name of the topic, where the data should be sent by Kafka. The **config_file** argument specifies the path to the configuration file used for parsing of blacklists. The default value is set to `./config.txt`.

Short Flag	Long Flag	Required	Default Value
-oz	–output_zookeeper	True	-
-ot	–output_topic	True	-
-cf	–config_file	False	./config.txt

Table 4.2: Arguments of the blacklist parser.

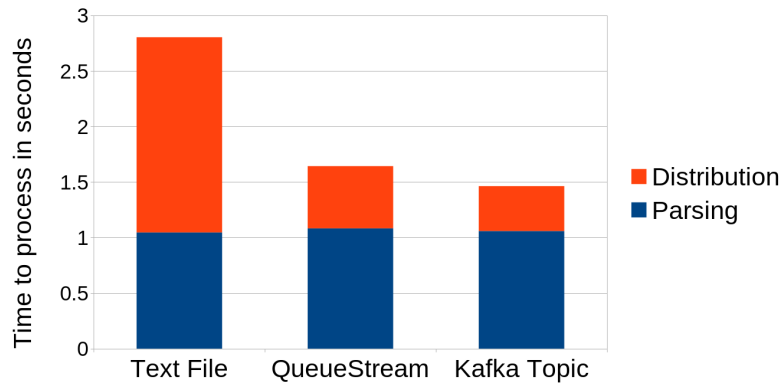


Figure 4.3: Comparison of time efficiency of the three proposed approaches.

4.3 Detection of Blacklisted Hosts

The second issue of the implementation was the detection of network hosts listed in the blacklist. In this case, two approaches were proposed, both with multiple separate approaches having different implementation details. These approaches are described in Sections 4.3.1 and 4.3.2. Both the input and blacklist streams are received by a Kafka Consumer client. These streams are handled by one Spark Streaming Context (SSC), which means they have the same microbatch interval. The SSC updates the state of the stream once every microbatch. Therefore, the blacklist data would need to be sent once a microbatch as well. As the blacklist data are updated in longer time intervals, updating blacklist stream every time the input stream is updated would be fairly inefficient.

For Spark to not update the stream once every microbatch and to save the state of the blacklist stream for a longer time, the **window()** function needs to be used. This function sets a time frame on the stream, collecting all the data received in the given time interval. Besides the length of the window, the **sliding window** interval can also be set. This value specifies the length of the steps the window takes. Both the window length and sliding interval have to be a multiple of the microbatch interval. An illustration is shown in Figure 4.4. For example, if the microbatch is set to 5 seconds, the window length is 15 seconds, and the sliding interval is 10 seconds, the window stores data from 3 consecutive RDDs, i.e. data collected in 15 seconds, and is moving by two RDD. The sliding window is used to make sure the blacklist data are present in the application at all times. The data need to be sent in intervals shorter than the length of the window.

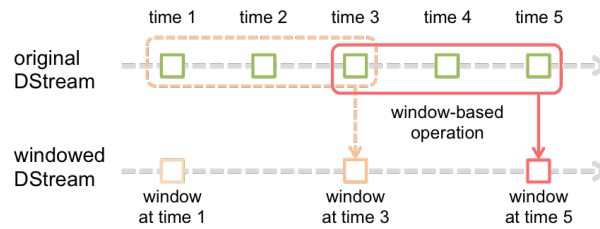


Figure 4.4: Illustration of the sliding window functionality [24].

However, to be able to perform operations on both streams together, Spark requires the input stream to have the same sliding window as the blacklist stream. Having a window set on input stream would unnecessarily increase the amount of data processed in every microbatch. Therefore, the sliding window interval cannot be specified. In the implementation, the window is set only on the blacklist stream with the necessity to send the blacklist data by the parser in shorter intervals than the window interval to have the blacklist data present at all times. The window length should also be shorter than one hour, as longer window intervals proved to be unstable causing the application to crash. The blacklist stream is mapped into 2-tuples containing the key and the description.

The input stream is in the JSON format as well, with variable values in each flow. A typical record in the input stream is shown in Listing 4.3. As described in Section 4.2, the blacklist data consist of three different types of values: IP address, URL address and a pair of an IP address and a port separated by a colon. In order to analyze all these values of a flow, each record in the input stream is split into multiple records with different keys. This is achieved by the `flatMap()` function, which applies a function on each record of the stream, creating zero or more new records from the one record. Specifically, each record from the input stream is split into two to five records based on the values the contained in the record. The five records have the following keys:

- source IP address,
- destination IP address,
- source IP address:source port,
- destination IP address:destination port,
- URL address.

4. IMPLEMENTATION - DETECTION

Every record must contain a source and a destination IP address. Therefore, each record is split into at least two records. Each record is checked for the presence of port values and the URL address value and then split. Both the source and destination IP addresses are used as keys to detect UDP packets, as the UDP communication is not necessarily back and forth. Each parsed record contains a list of multiple variables in the value. After both streams are parsed into the same format, they are processed to find records with matching keys. When a host in the input stream matches a blacklist entry, the corresponding record is further processed into an output format and sent to a Kafka topic. An example of an output entry is shown in Listing 4.4.

Listing 4.3: Example of an input flow in JSON format.

```
{
  "@type": "ipfix.entry",
  "ipfix.octetDeltaCount": 69,
  "ipfix.packetDeltaCount": 1,
  "ipfix.protocolIdentifier": 17,
  "ipfix.sourceTransportPort": 24894,
  "ipfix.sourceIPv4Address": "127.141.85.253",
  "ipfix.ingressInterface": 4,
  "ipfix.destinationTransportPort": 53,
  "ipfix.destinationIPv4Address": "31.251.56.50",
  "ipfix.egressInterface": 0,
  "ipfix.ipVersion": 4,
  "ipfix.flowStartMilliseconds": 1481797015637,
  "ipfix.flowEndMilliseconds": 1481797015637,
  "ipfix.ipTTL": 62
}
```

Listing 4.4: Example of a record in the result stream.

```
{
  "@type": "blacklists_detection",
  "detection_key": "127.8.115.247:53718",
  "src_ip": "127.8.115.247",
  "dst_ip": "0.1.55.234",
  "flows": 15,
  "flows_increment": 3,
  "timestamp": "2018-05-12T08:10:04.050Z",
  "src_port": 53718,
  "dst_port": 80,
  "protocol": 6,
  "bytes": 341,
  "description": "Reason for listing: Malware C&C"
}
```

For the application to report one detected address appearing in multiple flows just once, every address detected for the first time is sent to the output

and saved to a global dictionary. The dictionary stores a timestamp and the number of flows for each detected address. When an address stored in a dictionary is detected again before given period elapses, its number of flows is added to the total number of flows in the dictionary. When the address is detected after the given period elapses, it is sent to the output with the accumulated number of flows. This enables to determine the *flows_increment* in the visualization, specifically the histogram. The application also has a global variable holding the time the dictionary was last cleaned. Every 24 hours, the data stored in the dictionary that are older than 10-times the length of the given time interval are deleted. The output is stored in the Elasticsearch database and can be used for the visualization.

4.3.1 Intersection

The first proposed approach for detection is using functions based on the intersection set operation. Specifically, there are three functions in the core Spark API of this type: **intersection()**, **cogroup()** and **join()**. The intersection function takes two RDDs and returns an RDD containing only elements that were in both RDDs. It removes any duplicate elements, even if the duplicity was in the input RDDs. The main drawback of this function is the fact that it compares the whole elements, i.e. elements must be the same to be considered. As elements from both the blacklist and the input streams contain additional information besides the key used for comparison, this function is inconvenient for the implementation. The cogroup function is not exactly analogy of the intersection operation. It is an equivalent to the SQL operation full outer join, which creates a union of two datasets. However, additional filtering of the stream can create the intersection. The join function is the cogroup function with complemented by the filtering of intersecting elements. This makes the join function more suitable for the implementation than the cogroup function. Join returns an RDD of elements containing the key and a 2-tuple of values from the two matching elements. Join is also part of the Spark Streaming API, expanding the same functionality for DStreams. This function can be used either on the blacklist and the input streams or RDDs of these streams using the transformWith function.

4.3.1.1 Joining Streams

The first way the join function can be used is on the two DStreams. When called on the input stream with the blacklist stream as an argument, this function returns a stream containing all elements with matching keys in

these two streams. Each element contains values from both originating elements in the value as a 2-tuple. Illustration of basic use of the join function on DStreams is shown in Listing 4.5. The simple form of the resultant stream enables for convenient processing of the detected values.

Listing 4.5: Example of a use of the join function on DStreams.

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> xs = ssc.queueStream([x])
>>> y = sc.parallelize([("a", 2), ("a", 3)])
>>> ys = ssc.queueStream([y])
>>> x.join(y).pprint()
('a', (1, 2))
('a', (1, 3))
```

4.3.1.2 Joining RDDs Using Transform Function

The other way the join function can be used is in combination with the **transformWith()** function. This function can be used on a DStream and takes two arguments: a transforming function and a second DStream. TransformWith applies the given function to the cartesian product of the two DStreams, i.e. on each RDD from the first stream combined with each RDD from the second stream. The transforming function argument can be any function taking two RDDs as arguments and producing a single RDD as a return value. An example of a use of the transformWith function in a combination with join function is shown in Listing 4.6. This approach produces the same output as the join function used on the two streams.

Listing 4.6: Example of a use of the transformWith function.

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> xs = ssc.queueStream([x])
>>> y = sc.parallelize([("a", 2), ("a", 3)])
>>> ys = ssc.queueStream([y])
>>> x.transformWith((lambda rdd1, rdd2: rdd1.join(rdd2)), y).pprint()
('a', (1, 2))
('a', (1, 3))
```

4.3.2 Union and Aggregation

The second proposed approach for detection is based on the **union()** function combined with an aggregating function, namely **groupByKey()**, **reduceByKey()** or **combineByKey()**. The union function is a part of the core Spark API and is an analogy to the union operation on sets. It takes two RDDs and creates one RDD containing values from both RDDs. An illustration of basic

use of this function is shown in Listing 4.7. The union is also part of the Spark Streaming API, expanding the same functionality for DStreams. The main idea behind this approach is that if both streams alone contain only records with unique keys, the union of these streams will contain records with duplicate keys only if these records appeared in both streams, thus detecting values from the blacklist in the input stream. In order to remove the duplicity in both streams, the RDDs in these streams need to be aggregated. The aggregating functions group records with same keys into a single record. The united stream is aggregated as well. After the aggregation, the united stream is filtered to contain only the records that contain more than one value. The final stream then only contains blacklisted values that appeared in the input stream. The blacklist stream needs to be aggregated since the blacklist data are sent more than once during one window interval.

Listing 4.7: Example of a use of the groupByKey function.

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2), ("a", 3)])
>>> x.union(y).collect()
[('a', 1), ('b', 4), ('a', 2), ('a', 3)]
```

4.3.2.1 Aggregation Using Group Function

The first function that can be used for the stream aggregation is the **groupByKey()**. This function groups all RDDs of the DStream into a single sequence. The 2-tuples from the original stream are transformed into 2-tuples with the same key, where values are aggregated into an iterable object. An example of a use of the function is shown in Listing 4.8. In the aggregated united stream, all the records have an iterable object of iterable objects as a value, where every record with more than one iterable object in the value is filtered as a blacklisted entry. When applied to a large dataset and processed on multiple computing nodes, this function can be rather ineffective, and memory-consuming as the data are shuffled over the network and aggregated after all are in one place.

Listing 4.8: Example of a use of the groupByKey function.

```
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 2), ("b", 1)])
>>> sorted(rdd.groupByKey().mapValues(len).collect())
[('a', 2), ('b', 1)]
>>> sorted(rdd.groupByKey().mapValues(list).collect())
[('a', [1, 1]), ('b', [1])]
```

4.3.2.2 Aggregation Using Reduce Function

The second function that can be used for the stream aggregation is the **reduceByKey()**. This function aggregates the RDDs one by one using the given function. For example, if the task is to compute the sum for each key, the records are merged one by one, each time increasing the sum in the merged record. This computation is shown in Listing 4.9. In order to aggregate the values in input and blacklist streams, the values are added to a list. The united stream is filtered based on the number of values in the list. As opposed to the **groupByKey** function, **reduceByKey** function aggregates all the records on each worker node, reducing the number of records sent over the network, and then aggregates all the entries in one node. This technique is more effective when processing large dataset on a cluster of multiple worker nodes, and it is designed mainly for calculations of a sum, an average, etc.

Listing 4.9: Example of a use of the **reduceByKey** function.

```
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 2), ("b", 1)])
>>> rdd.reduceByKey(lambda value1, value2: value1 + value2).collect()
[('a', 3), ('b', 2)]
```

4.3.2.3 Aggregation Using Combine Function

The last function that can be used for the stream aggregation is the **combineByKey()**. This function is similar to the **reduceByKey** function, but the values do not need to be explicitly converted to a different type. **CombineByKey** is the most general aggregation function, and other aggregation functions are derived from it. It takes three arguments: an initiating function, a merging function, and a combining function. The initiating function takes an unprocessed value and creates a new value with the combined type. In the implementation, this function creates a single-element list. The merging function takes a value that has already been processed and an unprocessed value and merges these values. In the implementation, this function takes an aggregated list and an unprocessed value and adds this value to the list. As these operations are performed in parallel on different partitions, multiple aggregated values are created. The combining function is used to merge these aggregated values. In the implementation, this function takes two aggregated lists and merges them into one list. The aggregation of values into a list is shown in Listing 4.10.

Listing 4.10: Example of a use of the combineByKey function.

```
>>> x = sc.parallelize([("a", 1), ("b", 1), ("a", 2), ("b", 1)])
>>> x.combineByKey((lambda value: [value]),
                  (lambda value1, value2: value1.append(value2)),
                  (lambda value1, value2: value1.extend(value2))).collect()
[('a', [1, 2]), ('b', [1, 1])]
```

4.3.3 Conclusion

For testing of the performance of these approaches, all of them were implemented in a simplistic way, where the input stream was not split and contained only a 2-tuple with a destination IP address as a key. All the approaches were tested both on a dataset and on real anonymized data from the Masaryk University network. The testing methodology and the dataset used for testing are described in Section 4.1. The computing resources allocated for the measurements on the dataset consisted of 4 worker nodes, altogether using 4 CPU cores and 2GB of memory. Every approach was tested on the dataset five times to eliminate possible swings in the results. The chart in Figure 4.6 shows the comparison of times needed to process the whole dataset. The chart in Figure 4.7 shows the comparison of average processing times while processing the dataset. The real network data were tested using 8 CPU cores and 4GB of memory. Each approach was run for 24 hours. The chart in Figure 4.5 shows the comparison of average processing times while processing the real data. As the input rate varied between 6000 and 8000 records per second, all processing times were proportionally converted to a uniform value of average processing time with an input rate of 10000 records per second. As the results show, approaches using the combination of union and aggregation functions proved to be more efficient than approaches using the join function. The combineByKey function performed the best out of the three aggregation functions in all tests. For this reason, this approach has been chosen for the implementation.

The detection application is part of electronic attachments. Arguments of the application are shown in Table 4.2. The **input_zookeeper** and the **output_zookeeper** arguments specify an address and a port of the host running the Apache Zookeeper. The **input_topic** and **blacklist_topic** arguments specify names of Kafka topics providing the input and the blacklist data respectively. The **output_topic** argument specifies the name of the topic where the processed data should be sent. The **microbatch** argument specifies the size of the microbatch interval and is set to 10 seconds by default. The **blacklist_window** argument specifies the size of the window set on the blacklist

4. IMPLEMENTATION - DETECTION

stream with the default value being 610 seconds. The **dictionary_window** argument specifies the size of the window in which detections with the same key are considered to be part of continuous communication and are sent to the output. This argument is set to 300 seconds by default.

Short Flag	Long Flag	Required	Default Value
-iz	-input_zookeeper	True	-
-it	-input_topic	True	-
-bt	-blacklist_topic	True	-
-oz	-output_zookeeper	True	-
-ot	-output_topic	True	-
-bw	-blacklist_window	False	610 (seconds)
-m	-microbatch	False	10 (seconds)
-dw	-dictionary_window	False	300 (seconds)

Table 4.3: Arguments of the detection application.

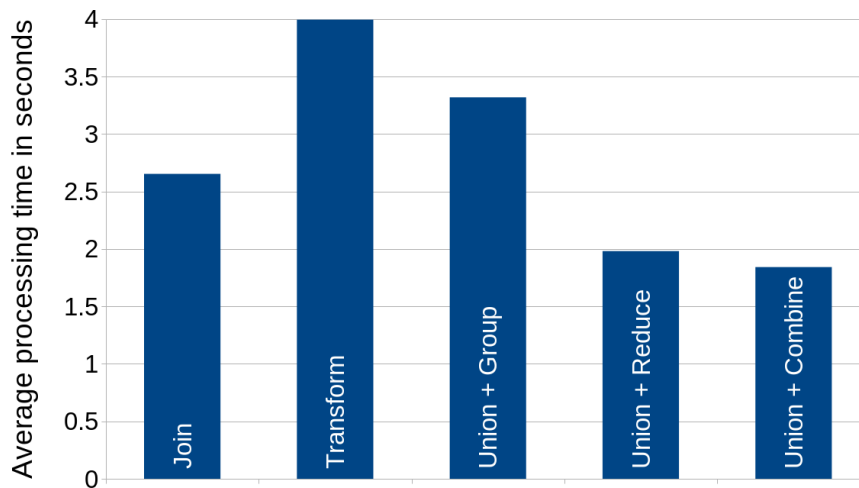


Figure 4.5: Comparison of performance on real data.

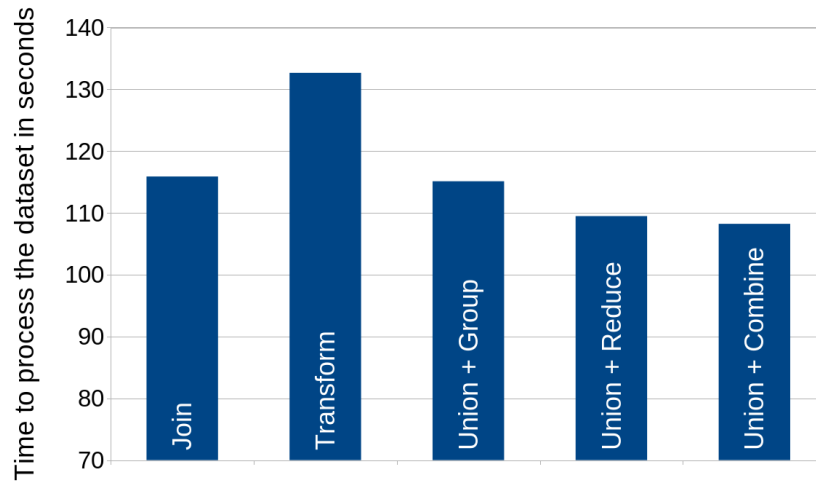


Figure 4.6: Comparison of performance on dataset (time to process).

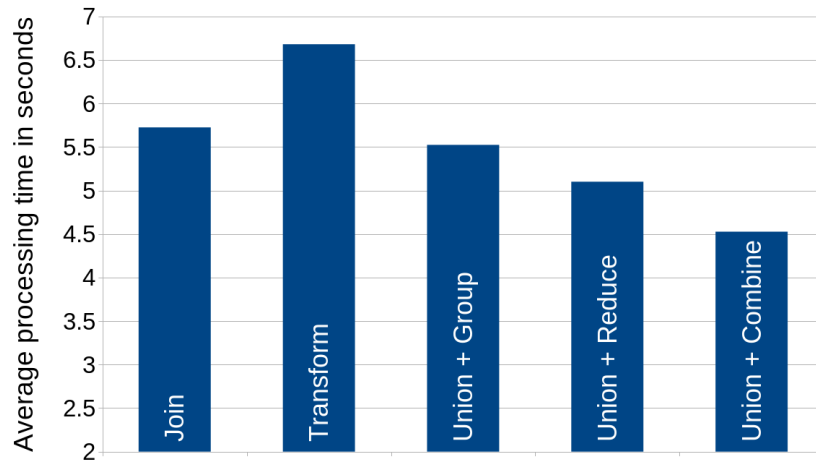


Figure 4.7: Comparison of performance on dataset (average processing time).

4.4 Visualization

Part of the implementation was to extend the web interface of Stream4Flow framework to contain visualization of the implemented detection script. The visualization of detection of blacklisted network hosts is part of the *Detections* tab. The page itself contains four elements. The first one, shown in Figure 4.8, is a settings panel allowing to set different intervals of the de-

4. IMPLEMENTATION - DETECTION

tections, aggregation interval and to filter specific addresses. The second element, shown in Figure 4.9, is a histogram showing the detections over time. The histogram can be set to visualize different time intervals as well. The third element, shown in Figure 4.10, is a Top N chart, which is a pie chart of the most detected addresses, both the blacklisted hosts and the victims in the network. The last element, shown in Figure 4.11, is a table containing detailed information about each detection that occurred in the given time interval. The web interface extension is part of electronic attachments.



Figure 4.8: Illustration of the settings panel.

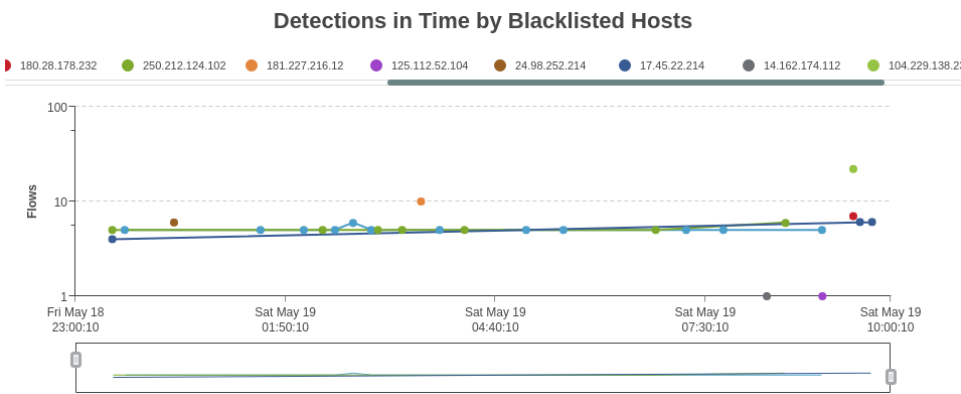


Figure 4.9: Illustration of the histogram of detections.

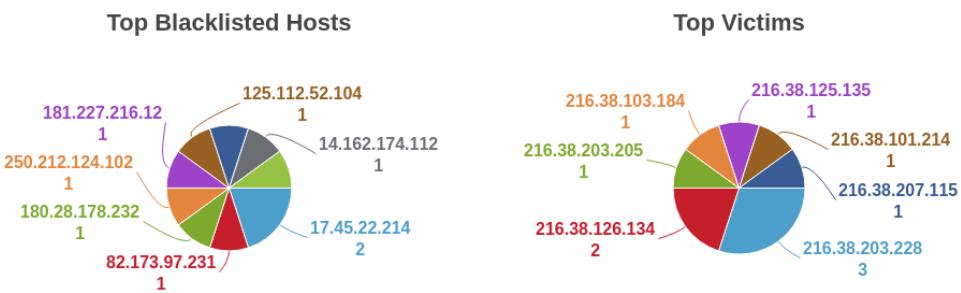


Figure 4.10: Illustration of the Top N chart for the application.

Timestamp	Source host	Destination host	Flows	Description
2018-05-19 07:07:32.917	82.173.97.231	216.38.126.134	61	Reason for listing: emotet
2018-05-19 06:35:06.651	250.212.124.102	216.38.126.134	36	Reason for listing: emotet
2018-05-19 07:45:50.588	17.45.22.214	216.38.203.228	16	Reason for listing: phishing
2018-05-18 21:30:49.061	17.45.22.214	216.38.203.205	4	Reason for listing: phishing
2018-05-19 06:19:18.076	14.162.174.112	216.38.207.115	1	Host address: static.vnpt.vn; Reason for listing: 2
2018-05-18 22:19:22.378	24.98.252.214	216.38.101.214	6	Reason for listing: suspicious
2018-05-19 07:29:43.163	104.229.138.234	216.38.203.228	22	Reason for listing: phishing
2018-05-19 07:05:07.441	125.112.52.104	216.38.103.184	1	Host address: 125.112.52.104; Reason for listing: 1
2018-05-19 07:30:03.125	180.28.178.232	216.38.203.228	7	Reason for listing: phishing
2018-05-19 01:41:32.417	181.227.216.12	216.38.125.135	10	Reason for listing: phishing

Figure 4.11: Illustration of the table with detailed description of detections.

4.5 Results

The finalized application was again tested both on the dataset and on real anonymized data from the Masaryk University network. The dataset was used to test the precision of the detection method. A few of the values from the dataset were added to the blacklist, including IP addresses, URL addresses, and IP addresses with a port. All the added values were detected with zero false-positive detections. The application was also tested on the real anonymized data, running for approximately 48 hours, to test the efficiency and stability. The blacklist data were sent through the Kafka topic every 10 minutes. As the proposed approaches for detection were tested with records in the input stream parsed into simple 2-tuples with destination IP address as a key, the final version of the application, where the input stream was split to up to five times more data, was expected to have higher resource requirements. The application itself was running on 4 worker nodes altogether using 8 processing cores and 4GB of memory. In the first run, the microbatch was set to 5 seconds, but this interval proved to be too low for the cluster to maintain stability. When the microbatch was set to 10 seconds, the application was stable in a long-term, getting delayed only a few times when the input rate was at its peak. The results are shown in Figure 4.12. The input rate was on average 5027 records per second and the average processing time was 6.89 seconds. To completely avoid any delays throughout the use, the amount of used computing resources would need to be higher, as higher microbatch would not lead to better stability.

4. IMPLEMENTATION - DETECTION

Running batches of 10 seconds for 2 days since 2018/05/01 21:34:58 (17280 completed batches, 914541490 records)

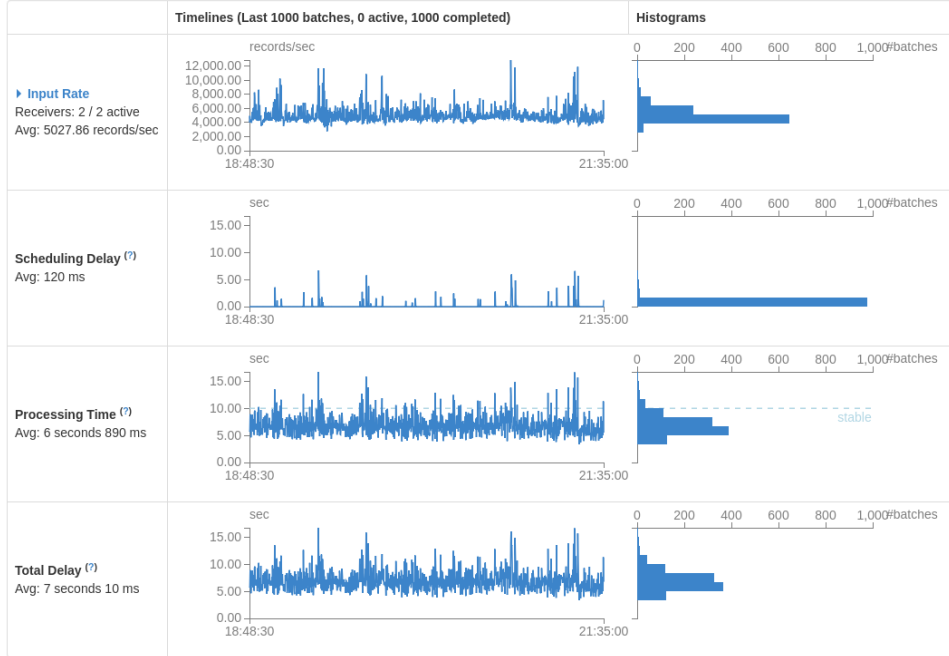


Figure 4.12: The results of the application testing.

5 Conclusion

The main goal of this thesis was to design and implement an application for the detection of communication with blacklisted network hosts that would later be a part of the Stream4Flow framework. All crucial parts of the application were implemented using multiple different approaches and tested on real anonymized data from the Masaryk University network to determine the most efficient approach. Part of the implementation was also to create a web interface for visualization of the application output. Results of the long-term testing of the final application show that the proposed solution can be used for monitoring of large corporate or university networks with a throughput of more than 5000 flows per second using only 8 processing cores and 4 gigabytes of memory. As the whole application was designed to be highly scalable, it can be used for monitoring of computer networks with even higher throughput just by adding more processing cores and memory. All the scripts mentioned in this thesis are part of electronic attachments, which also include the dataset that was used for testing.

The detection application was also designed to be easily upgradable. The detection is based on three different types of values: an IP version 4 address, an IP address with a port and a URL address. It is possible to add other types of values, such as IP address in version 6. The detection of URL addresses can also be enhanced by adding a matching using regular expressions, as URL addresses can be both with the *www* prefix and without it. The blacklist parser uses a config file for the blacklist parsing. Therefore, other blacklists can be added without the need to modify the parser itself. The config file enables the network administrator to establish own collection of trustworthy blacklists. Currently, the blacklist parser can parse only blacklists that are in the form of a text file with an unambiguous format. That can be enhanced by adding the possibility to parse blacklists from tables or directly from web pages, as long as the page contains blacklist entries with the same format as well.

Bibliography

- [1] L. Cottrell. *Passive vs. Active Monitoring*. [online], cit. [2018-1-25]. 2001. URL: <https://www.siac.stanford.edu/comp/net/wan-mon/passive-vs-active.html>.
- [2] J. Case et al. *Introduction and Applicability Statements for Internet Standard Management Framework*. RFC 3410. RFC Editor, 2002, pp. 1–27. URL: <https://tools.ietf.org/html/rfc3410>.
- [3] D. Mauro and K. Schmidt. *Essential SNMP: Help for System and Network Administrators*. O'Reilly Media, 2005. ISBN: 9780596552770. URL: https://books.google.cz/books?id=65_0d25EpB4C.
- [4] J. Malík. “Správa různorodých síťových zařízení pomocí SNMP protokolu”. Diplomová práce. Masarykova univerzita, Fakulta informatiky, Brno, 2011.
- [5] Symantec. *The Perils of Deep Packet Inspection*. [online], cit. [2018-4-19]. 2005. URL: <https://www.symantec.com/connect/articles/perils-deep-packet-inspection>.
- [6] L. M. Garcia. *TCPDUMP/LIBPCAP public repository*. [online], cit. [2017-3-4]. 2010. URL: <http://www.tcpdump.org/>.
- [7] G. Combs. *Wireshark*. [online], cit. [2017-3-4]. 1998. URL: <https://www.wireshark.org/>.
- [8] R. Bendrath and M. Mueller. “The End of the Net as We Know it? Deep Packet Inspection and Internet Governance”. In: *New Media & Society* 13.7 (2011), pp. 1142–1160.
- [9] B. Fung. *What to expect now that Internet providers can collect and sell your Web browser history*. [online], cit. [2018-3-6]. 2017. URL: https://www.washingtonpost.com/news/the-switch/wp/2017/03/29/what-to-expect-now-that-internet-providers-can-collect-and-sell-your-web-browser-history/?utm_term=.7b9872e215f1.
- [10] N. Anderson. *Deep packet inspection under assault over privacy concerns*. [online], cit. [2017-4-18]. 2008. URL: <https://arstechnica.com/2008/05/deep-packet-inspection-under-assault-from-canadian-critics/>.
- [11] W. Cong, J. Morris, and W. Xiaojun. “High performance Deep Packet Inspection on multi-core platform”. In: *2009 2nd IEEE International Conference on Broadband Network Multimedia Technology*. 2009, pp. 619–622.
- [12] B. Claise, B. Trammell, and P. Aitken. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information*. RFC

BIBLIOGRAPHY

7011. RFC Editor, 2013, pp. 1–76. URL: <https://tools.ietf.org/html/rfc7011>.
- [13] Cisco Systems. *Cisco IOS NetFlow*. [online], cit. [2018-3-26]. URL: <https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html>.
- [14] T. Jirsik et al. “Toward Stream-Based IP Flow Analysis”. In: *IEEE Communications Magazine* 55.7 (2017), pp. 70–76. ISSN: 0163-6804. DOI: 10.1109/MCOM.2017.1600972.
- [15] IETF. *Ipfix Status Pages*. [online], cit. [2018-3-26]. URL: <https://tools.ietf.org/wg/ipfix/>.
- [16] R. Stewart. *Stream Control Transmission Protocol*. RFC 4960. RFC Editor, 2007. URL: <http://www.rfc-editor.org/rfc/rfc4960.txt>.
- [17] R. Hofstede et al. “Flow monitoring explained: from packet capture to data analysis with NetFlow and IPFIX”. In: *IEEE Communications Surveys & Tutorials* 16.4 (2014), pp. 2037–2064.
- [18] M. Cermak, T. Jirsik, and D. Tovarnak. *Stream4Flow*. [online], cit. [2018-2-20]. URL: <https://stream4flow.ics.muni.cz>.
- [19] M. Čermák, T. Jirsík, and M. Laštovička. “Real-time analysis of NetFlow data for generating network traffic statistics using Apache Spark”. In: *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*. 2016, pp. 1019–1020. DOI: 10.1109/NOMS.2016.7502952.
- [20] CESNET. *Ipfixcol*. [online], cit. [2018-3-18]. URL: <https://github.com/CESNET/ipfixcol#contact>.
- [21] Apache Software Foundation. *Apache Kafka*. [online], cit. [2018-3-18]. URL: <https://kafka.apache.org/>.
- [22] M. Čermák et al. “A performance benchmark for NetFlow data analysis on distributed stream processing systems”. In: *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*. 2016, pp. 919–924. DOI: 10.1109/NOMS.2016.7502926.
- [23] Apache Software Foundation. *Apache Samza*. [online], cit. [2018-3-18]. URL: <https://samza.apache.org/>.
- [24] Apache Software Foundation. *Apache Spark*. [online], cit. [2018-3-18]. URL: <https://spark.apache.org/>.
- [25] J. Dean and S. Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Commun. ACM* 51.1 (2008), pp. 107–113.
- [26] Elasticsearch. *Elastic Stack*. [online], cit. [2018-3-18]. URL: <https://www.elastic.co/products>.
- [27] Apache Software Foundation. *Apache Lucene*. [online], cit. [2018-3-18]. URL: <https://lucene.apache.org/>.

- [28] M. D. Pierro. *Web2Py Web Framework*. [online], cit. [2018-4-27]. URL: <http://www.web2py.com/>.
- [29] R. Shirey. *Internet Security Glossary, Version 2*. RFC 4949. RFC Editor, 2007. URL: <http://www.rfc-editor.org/rfc/rfc4949.txt>.
- [30] R. S. Sandhu and P. Samarati. "Access control: principle and practice". In: *IEEE Communications Magazine* 32.9 (1994), pp. 40–48. ISSN: 0163-6804.
- [31] M. Felegyhazi, C. Kreibich, and V. Paxson. "On the Potential of Proactive Domain Blacklisting". In: *Proceedings of the 3rd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*. LEET'10. USENIX Association, 2010, pp. 6–6.
- [32] J. Ma et al. "Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious URLs". In: *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2009, pp. 1245–1254.
- [33] D. Chiba et al. "Detecting Malicious Websites by Learning IP Address Features". In: *2012 IEEE/IPSJ 12th International Symposium on Applications and the Internet*. IEEE, 2012, pp. 29–39.
- [34] R. Yamada and S. Goto. "Using abnormal TTL values to detect malicious IP packets". In: *Proceedings of the Asia-Pacific Advanced Network*. 2013, p. 27.
- [35] MISP Project. *Malware Information Sharing Platform*. [online], cit. [2018-5-5]. URL: <http://www.misp-project.org/>.
- [36] CESNET. *Warden*. [online], cit. [2018-5-5]. URL: <https://warden.cesnet.cz/>.
- [37] CESNET. *SABU*. [online], cit. [2018-5-5]. URL: <https://sabu.cesnet.cz/>.
- [38] C. Lewis and M. Sergeant. *Overview of Best Email DNS-Based List (DNSBL) Operational Practices*. RFC 6471. RFC Editor, 2012. URL: <http://www.rfc-editor.org/rfc/rfc6471.txt>.
- [39] J. Levine. *DNS Blacklists and Whitelists*. RFC 5782. RFC Editor, 2010. URL: <http://www.rfc-editor.org/rfc/rfc5782.txt>.
- [40] Google LLC. *Google Safe Browsing*. [online], cit. [2018-4-3]. URL: <https://safebrowsing.google.com/>.
- [41] Microsoft Corporation. *Microsoft SmartScreen Filter*. [online], cit. [2018-4-3]. URL: <https://support.microsoft.com/en-gb/help/17443/windows-internet-explorer-smartscreen-filter-faq>.
- [42] D. Bermingham. *The Wonders of Whitelisting (as Opposed to Blacklisting)*. [online], cit. [2018-4-3]. 2014. URL: <https://www.kaspersky.com/blog/wonders-of-whitelisting/6367/>.

BIBLIOGRAPHY

- [43] M. Kucherawy and D. Crocker. *Email Greylisting: An Applicability Statement for SMTP*. RFC 6647. RFC Editor, 2012. URL: <http://www.rfc-editor.org/rfc/rfc6647.txt>.
- [44] J. Klensin. *Simple Mail Transfer Protocol*. RFC 5321. RFC Editor, 2008. URL: <http://www.rfc-editor.org/rfc/rfc5321.txt>.
- [45] J. B. Postel. *Simple Mail Transfer Protocol*. STD 10. RFC Editor, 1982. URL: <http://www.rfc-editor.org/rfc/rfc821.txt>.
- [46] FireHOL. *FireHOL IP Lists*. [online], cit. [2018-4-8]. URL: <http://iplists.firehol.org>.
- [47] Spamhaus. *The Spamhaus Project*. [online], cit. [2018-4-8]. URL: <https://www.spamhaus.org>.
- [48] L. Metcalf and J. M. Spring. "Blacklist Ecosystem Analysis: Spanning Jan 2012 to Jun 2014". In: *Proceedings of the 2Nd ACM Workshop on Information Sharing and Collaborative Security*. ACM, 2015, pp. 13–22. doi: 10.1145/2808128.2808129.
- [49] M. Kührer, C. Rossow, and T. Holz. "Paint It Black: Evaluating the Effectiveness of Malware Blacklists". In: *Research in Attacks, Intrusions and Defenses. RAID 2014*. Springer International Publishing, 2014, pp. 1–21.
- [50] RiskAnalytics. *Malware Domain Blocklist*. [online], cit. [2018-4-6]. URL: <http://www.malwaredomains.com/>.
- [51] myip.ms. *myip.ms blacklist*. [online], cit. [2018-4-6]. URL: <https://myip.ms/browse/blacklist/>.
- [52] abuse.ch. *SSL Blacklist*. [online], cit. [2018-4-6]. URL: <https://sslbl.abuse.ch/>.
- [53] Apache Software Foundation. *Spark Streaming Programming Guide*. [online], cit. [2018-5-11]. URL: <http://spark.apache.org/docs/latest/streaming-programming-guide.html>.
- [54] CAIDA. *The CAIDA Anonymized Internet Traces 2015 Dataset*. [online], cit. [2018-4-21]. URL: http://www.caida.org/data/passive/passive_2015_dataset.xml.
- [55] Apache Software Foundation. *Apache ZooKeeper*. [online], cit. [2018-4-25]. URL: <https://zookeeper.apache.org/>.

A Attachments

Electronic attachments content

- /dataset/ – directory containing the dataset used for testing
 - ./fbit/ – directory containing records saved in the FastBit database
 - ./ipfix/ – directory containing the ipfix file for replaying of the dataset
 - ./README.md – file describing structure of the directory
- /detection/
 - ./spark/ – directory containing the detection application
 - ./web-interface/ – directory containing the web interface
 - ./README.md – file describing structure of the directory