SILESIAN UNIVERSITY OF TECHNOLOGY
FACULTY OF AUTOMATIC CONTROL, ELECTRONICS AND COMPUTER SCIENCE

Master thesis

Statistical, real-time classification of IP traffic in Linux operating system

Author: Paweł Foremski
Supervisor: dr inż. Arkadiusz Biernacki

Gliwice, September 2011

Table of Contents

1. INTRODUCTION

The Internet has been constantly evolving since its inception. For more than a decade it has been growing in capacity and versatility with a great pace, often requiring the Internet Service Providers to update and extend their infrastructure in a timely manner.

These changes are connected with the inventions of new kinds of computer software, which in turn generate new types of network traffic. However, the fundamental protocol of the Internet – the IP protocol – does not provide a robust and universal mean to differentiate one traffic type from another. Thus, identification of a particular application in Internet transmissions is not a trivial task, yet it is very important.

For instance, a typical Internet end-user demands a safe and fast Internet access. An Internet Service Provider which is to fulfil such a requirement must be able to monitor the traffic for potential threats and to impose a proper prioritization on the traffic. Moreover, there are political and research organizations which monitor the global Internet. Observing the share of P2P traffic in Internet transmissions of a particular country could reveal trends in its society. Work in these areas cannot be done without a reliable source of information. A fundamental question remains: given an Internet transmission, what is the name of application that produced it? This is the problem of traffic classification.

This thesis proposes a practical implementation of a possible solution to this problem and presents its performance evaluation results.

This chapter introduces the problem of traffic classification (section 1.1), reviews its existing solutions (section 1.3), and formulates the thesis goals (section 1.2).

1.1. The problem of Internet traffic classification

Communication in the Internet follows the *Internet Protocol* (IP) [RFC791]. Basically, on top of IP, there are *User Datagram Protocol* (UDP) [RFC768] and *Transmission Control Protocol* (TCP) [RFC793], both of which being *Transmission Protocols* (TP). A single IP transmission connects two application *processes*, often running on two distant *hosts* (general concept presented on Fig. 1.1). Between these two hosts there is the Internet, comprised of intermediary hosts, called *routers*.

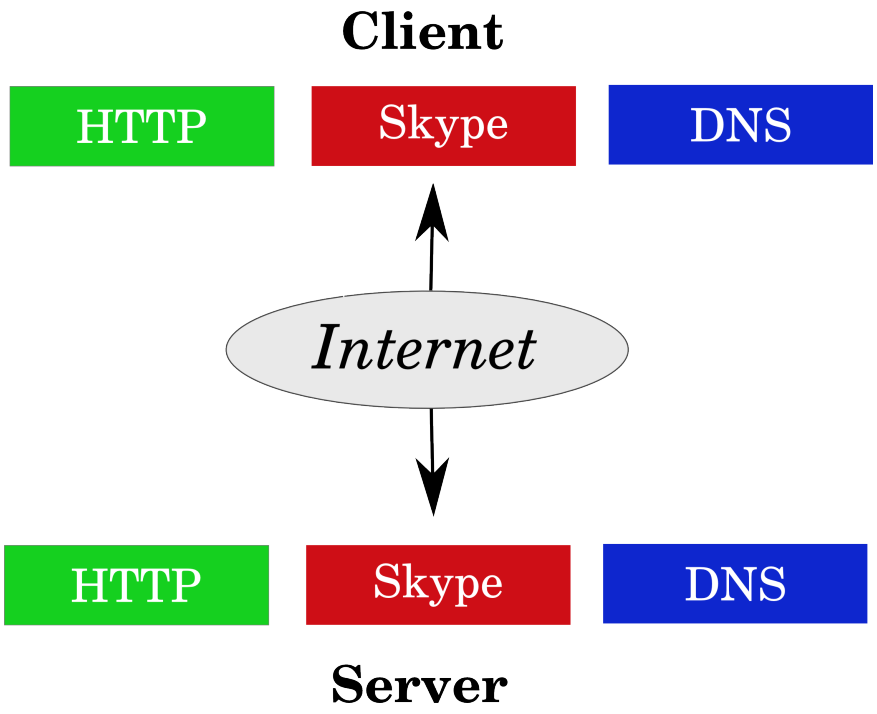

Fig. 1.1: ***General concept of Internet communication.*** *Three processes (HTTP, Skype and DNS) running on "Client" host contact their respective counterparts running on remote "Server" host using the Internet.*

The host *Operating System* (OS) provides an *Application Programming Interface* (API), through which a process can send (and receive) data in a simple manner, basically of arbitrary length and format.

This API facility is thus a kind of a proxy between the process and the Internet. It handles the operation of putting the data into well-formed IP packets, which often means dividing the stream of application data into many IP packets. This operation has the following implications, depicted on Fig. 1.2:

1. Application identity is lost.
2. Packets can go different routes.

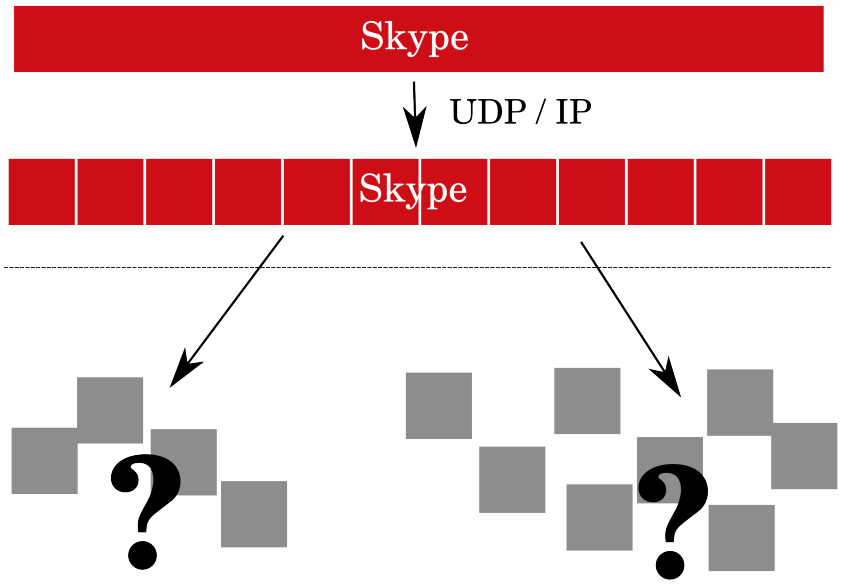

Fig. 1.2: ***Process data leaving the host boundary.*** *Data of a single Skype process is divided into IP packets and enters the Internet using two different routes.*

Moreover, at a single time instant, a router has direct access only to the contents of a single packet. This all leaves Internet operators with very little information on the traffic passing through their infrastructure. For this reason, a useful approach is to look at the traffic from a more distant perspective.

The IP protocol defines a *header* part in its packet, having several fields, including:

1. source IP address,
2. destination IP address,
3. desired transport protocol.

Similarly, Transport Protocol header carries:

4. source port number,
5. destination port number.

A group of packets having the same tuple of fields {*1, 2, 3, 4, 5*} is called a *flow*. It has a useful property that each packet in a flow belongs to the same transmission and thus was generated by the same application (and by the same two processes). Analysis of flows is a convenient way of looking at IP traffic, which allows to gather some statistical characteristics, like the average packet size, flow duration, bit rate, etc. However, application names are still unknown, as presented on Fig. 1.3.

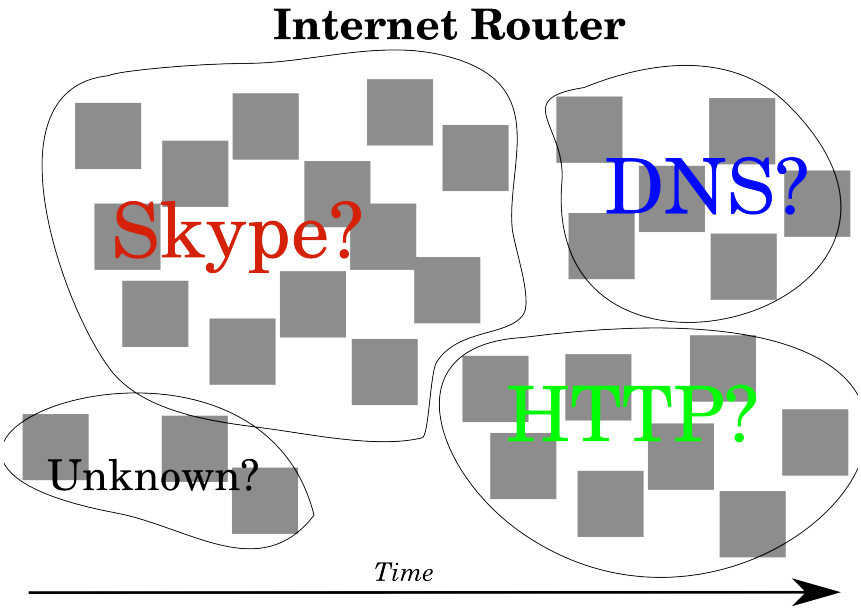

Fig. 1.3: ***Groups of packets collected in flows.*** *The identity of applications is still unknown.*

Finally, the problem of Internet traffic classification can be formulated by the question: *given IP packets, what is the identity of application that sent them?*

An answer to this question is called a *classification verdict*. Usually, it is issued for a whole flow, which indirectly leads to classification of single IP packets. However, limited solutions to direct classification of single packets also exist (see 1.3.1). The concept of *application identity* is quite broad. It can span from rough application type (e.g.

"streaming") to detailed program name and version (e.g. "Skype v. 2.5.3"), and it is directly connected with information on characteristics of the traffic that it represents. The term *application* – apart from typical computer programs – can also include viruses, misconfigured or misbehaving software, etc. Indeed, traffic classification is a process of associating groups of IP packets with application identities.

1.2. Thesis goals

The aim of this work is to provide a practical implementation of a traffic classifier. For the main algorithm, two novel, complementary methods will be used, published in:

- *KISS: Stochastic Packet Inspection Classifier for UDP Traffic [Fin09]*
- *Stochastic Packet Inspection for TCP Traffic [Fin10]*

Minor modifications and extensions will be made to these methods (see 2.1.3) in order to make the resultant system more suitable for practical usage as a *firewall* element and to increase its performance. Application of the final system will be limited to about 5% of Internet endpoints, but carrying more than 98% of bytes (see 2.1).

Result of the thesis is a computer program, with the following characteristics:

- Support for the TCP and UDP protocols.
- Written in C language.
- Operation under the GNU/Linux environment.
- Simultaneous offline and real-time classification.
- Simultaneous training, classification, and performance testing.
- Classification through *Support Vector Machines*.
- Support for the popular "PCAP" format of IP traffic trace files.

For instance, using the thesis program, operator of a Linux router will be able to classify the network traffic passing through his infrastructure. First, the program will be trained with traffic samples of known applications, and then the system will monitor chosen network interfaces in real-time. Finally, each time a trustworthy classification is made, the program will emit an adequate message.

1.3. Review of existing solutions

1.3.1. Simple methods

An Internet application has some standard ways of putting a kind of a label on its packets. Such label can be used to determine the application identity and to make assumptions and predictions on the transmission characteristics.

These standard methods are:

1. Usage of a well-known TP port number. Application of a port-protocol association database enables classification, e.g. destination port 80 could mean an HTTP browser.

2. Usage of the *Type of Service* (ToS) field of the IP header. Its value can tell the transmission characteristics, e.g. value of 00010000_2 is a low delay traffic, like a VoIP transmission. The ToS field was updated and replaced by the *Differentiated Services Code Point* (DSCP) in [RFC2474].

Both methods are limited. Port number has space of 16 bits, what gives an upper limit of well below 10^5. Similarly, length of the ToS field bounds the number of traffic types well below 10^3. By contrast, there are more than 2×10^9 users of the Internet worldwide, as of March 2011 [IUS]. Each user has his favourite set of applications and own usage patterns.

Moreover, these methods base on knowledge which is local to the host and its close neighbourhood. For example, an application requesting a low-delay transmission for a BitTorrent download may not deserve such prioritization compared to the needs and resources of the global Internet. What is more, a frequent practice is to establish interpretations of the DSCP field for the whole *Autonomous System* (AS) or *Internet Exchange* point (IX). Thus, the meaning of a DSCP value changes as IP packet traverses the Internet. This value may be interpreted differently in different parts of the world.

Finally, neither usage of a well-known port nor setting the DSCP value is mandatory. All these deficiencies rendered both methods obsolete. They must not be used as a source of reliable, accurate, and Internet-wide source of information [Kar04].

Still, these simple methods, especially the port-based classification, have some limited application areas and can produce meaningful results under some conditions. Its greatest advantage is that it is stateless and can immediately classify each single IP packet.

1.3.2. Deep Packet Inspection

The growth of *Peer-to-Peer* traffic (P2P), streaming media applications and viruses unveiled inadequacy of relying on simple methods for traffic classification. The Internet industry had to invent better techniques.

The most straight-forward solution is to inspect the packet payload for well-known patterns. A router having a database of patterns would then search the contents of each packet for a match against the database. Such approach is called *Deep Packet Inspection* (DPI). Real-world solutions using this technique exist both in the commercial world [Ellacoya] and in the open source community [L7], [ODPI].

DPI has a serious drawback – it is effectively useless if the application uses encryption. Moreover, analysis of the packet contents brings user privacy issues. Search for a complicated pattern in each packet can be computationally expensive. The database of patterns needs to be constantly kept up-to-date, and each new protocol signature has to be manually analysed and extracted by a skilled engineer before it can be used.

However, DPI methods can be very accurate and seem to be the current industry standard. For researchers, they are often a basis for comparison of new methods [Kar05].

1.3.3. Modern approaches

Another approach pursued nowadays is classification based on some characteristic features – often statistical – of the traffic as a whole.

Such system could, for example, observe the average packet size and inter-packet time gaps. Then, an inference engine would make a decision basing on the values of these statistics – either by comparing them with thresholds or by using machine learning techniques. Research efforts in this field were started with a fundamental work dating 1994 [Cla94], with practical classification systems appearing a decade later [Rou04], [Kar05], [Moo05], [Zan05]. They all bring promising results.

Basically, each of these methods has two discriminating elements: observed traffic features and the classification algorithm.

1.3.3.1 Traffic features

IP packets have many properties. Apart of the ones already introduced, there are also: total packet size, time of arrival, sequence number, and many more. One can aggregate packets in a flow, but by choosing different tuple elements, many kinds of such traffic aggregations are possible. Another common aggregation is host-level, e.g. by tuple of {*source IP*, *TP*}. Then, having a set of packets grouped together according to a common criterion, several traffic features can be computed.

On the lowest packet level, features like mean packet size, observed TCP header flags, inter-packet time gaps, etc. can be computed. By looking at flow-level characteristics, one could analyse mean flow duration, mean data volume per flow, and the variance of these metrics [Rou04]. Another interesting view is a social-level view [Kar05], in which – for example – the number of remote locations a host contacts, and average volume of data exchange for each of them, could be analysed.

Finally, in the classification method used in this thesis, traffic features are extracted from statistical analysis of the packet payload. In [Fin09], randomness of the first few bytes in UDP packet payloads is measured (see 2.1) and used to form a *signature*. In a recent work on fine-grained classification [Byu11], a similar approach adopting a document retrieval technique is presented. Signatures are created from frequencies of keyword occurrences in packet payloads.

1.3.3.2 Classification algorithm

Modern traffic classifiers often use machine learning techniques. In a typical scenario, the classifier must be trained before it is able to distinguish one application from another. In order to do that, traffic samples of a particular computer program are carefully collected. Traffic features are extracted and the data is used for training the classifier.

The number of feature vectors required for training depends on their length and the machine learning technique used, 5-10 x 10^3 being a rough estimation in [Kim08]. Research efforts tend to use different classification algorithms - Naïve Bayes, Bayesian

Networks, Decision Trees, k-Nearest Neighbors (k-NN), Neural Networks and Support Vector Machines (SVM), with the latter giving the best results.

The research community lacks a common set of traffic samples [Sal07], hence comparison of different classification algorithms is hard. Even obtaining a single suitable training set can be a difficult task. Researchers must fall back on DPI solutions in order to annotate the data sets with "ground truth", i.e. labels with application names [Kar05].

1.4. Thesis contents

Chapter 1 discusses the problem of traffic classification and describes the thesis goals: an introduction to the subject, basic definitions, problem statement and a review of possible solutions in the existing literature; thesis goals and description of results.

Chapter 2 describes the resultant system in terms of algorithms and design: summary of the main KISS algorithm with modifications, general system architecture and methods for performance evaluation.

Chapter 3 gives information on software implementation: design decisions, external software libraries and descriptions of the *libspi* library and the *spid* program.

Chapter 4 presents performance evaluation results.

Chapter 5 concludes the thesis with statements about resultant program quality.

Chapter 6 is a summary on feasibility of practical traffic classification – in general, and using the thesis software.

Chapter 7 is an appendix holding implementation details – documentation of: data structure members, Application Programming Interface, and the user interface of the *spid* program.

Chapter 8 gives thesis bibliography.

Chapter 9 contains thesis subject and summary in Polish and English.

2. SYSTEM DESCRIPTION

The goal of this thesis is implementation of a statistical traffic classification system capable of working in real-time.

In order to achieve this, an already existing classification method will be used as the main algorithm. It will be extended by several other, original elements required to make it a practical real-time system.

For the main algorithm, two complementary methods will be applied, covering TCP ([Fin10]) and UDP ([Fin09]) protocols, jointly referred to as “the KISS algorithm”. They both employ a statistical test similar to the Pearson's Chi-Square test, hence the name of “KISS”, standing for *Chi-Square Signatures*.

The system elements introduced in the thesis cover: obtaining IP traffic, training the system, handling the SVM library in an optimal way, and presenting the results.

This chapter describes the whole system from design point of view. Section 2.1 characterize the KISS algorithm with modifications, section 2.2 gives a more practical view on the system architecture, while the section 2.3 presents an adequate method for performance evaluation of the whole system.

2.1. Main algorithm

The KISS algorithm is a traffic classifier which observes randomness in first bytes of IP packet payloads.

As the input it takes a set of IP packets. As the output, a set of {*endpoint*[1], *label*} pairs is provided, where label can be used to uniquely determine application identity. Internally, the algorithm consists of two stages – feature extraction and decision process – depicted on Fig. 2.1.

KISS authors claim very good results of average 99.6% True Positives[2] with less than 1% of False Positives[3]. However, as it requires at least 80 packets, it is limited to about 5% of Internet endpoints, but carrying more than 98.6% of bytes[4].

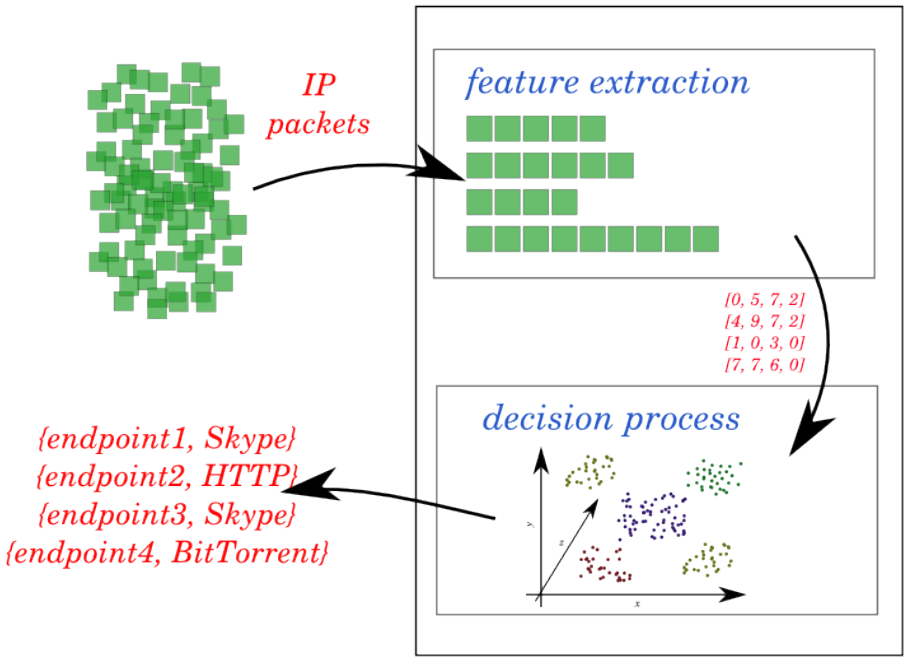

Fig. 2.1: ***The KISS algorithm.*** *IP packets enter feature extractor, which feeds the decision process, which produce the output.*

1 Non-standard mode of operation for UDP. See 2.1.3.

2 See 2.3 for definition.

3 For UDP (sect. VII in [Fin09]), with similar but a bit worse results for TCP (sect. V in [Fin10]).

4 Sect. V-H in [Fin09].

2.1.1. Feature extraction

Feature extraction begins with queueing packets in endpoints, that is, groups of packets having the same tuple of {*IP address, TP protocol, TP port*}. For TCP, all packets except the first P=5 data segments in TCP sessions are dropped[1].

For each queued packet, all data except the first N=12 bytes of packet payload (past the TP header) is removed. Each of these N bytes is divided into two halves, forming G=24 *groups*, each of bit-length equal b=4. Once C=80 packets are gathered in an endpoint, they form a *signature window,* a matrix with C rows and G columns, which is further processed.

For each possible cell value – which is 0 (0000_2) till 15 (1111_2) – the number of its occurrences in each column is counted. Result is denoted with $O_i^{(g)}$, where *g* is the column number and *i* is the value.

Next, each column is summarized with a single value X_g, which is a statistical test similar to Pearson's Chi-Square (χ^2) test:

$$X_g = \sum_{i=0}^{2^b-1} \frac{\left(O_i^{(g)} - E\right)^2}{E} \qquad (2.1)$$

The motivation for this is to measure the distance between distribution of observed values and the uniform distribution. Its expected value E is chosen as if all possible group values were equally probable:

$$E = \frac{C}{2^b} = 5 \qquad (2.2)$$

Indeed, a small X_g value means roughly that all out of the 2^b possible group values are equally probable in the column *g*, hence its randomness across C packets is high. For instance, it may carry a unique query identifier chosen in a random way for each packet. On the other hand, columns carrying constant values – like protocol version numbers – will result in high X_g values.

See Sec. III-A in [Fin09] for a detailed discussion.

1 See Sect. III-B in [Fin10].

Finally, a *signature* (or a *feature vector*) is constructed by collecting X_g values from all columns:

$$\bar{X} = [X_1, X_2, \ldots, X_G] \qquad (2.3)$$

This 24-dimensional vector is a very accurate application fingerprint that can be used for further classification.

2.1.2. Decision process

The task of the decision process is to reveal the application identity behind a signature.

For this task, KISS uses *Support Vector Machines* (SVM) [Cor95], which is a set of machine learning techniques that can be used for classification and regression. The general idea behind SVM is to rearrange the problem space so that the training samples can be easily separated by hyper-planes. This is realized by means of translating the input N-dimensional space into a possibly infinite-dimensional space, maximizing the distance between hyper-planes and the training sample points.

Before it can be used, SVM needs to be trained with several signatures annotated with the target class. As a result of such training process, a *model* is generated, which is later used for prediction of the proper signature class, which is the output of the KISS decision process.

2.1.3. Modifications

A few modifications and extensions were made to the KISS algorithm, for the needs of the thesis.

2.1.3.1 Classification objects

Section IV-A of [Fin09] defines two classification entities – a flow and an endpoint. In the thesis, such a modification is made, that packets are grouped in endpoints without regard to packet direction. This is contrary to Sec. III-B in [Fin10].

Motivation for this is to weaken the limits mentioned in 2.1. With this modification, the requirement of C=80 packets in a signature window can be attained earlier.

2.1.3.2 Signature extension

Feature vectors can be optionally extended with 4 coordinates:

1. Average packet size.
2. Average value of inter-packet time gap (IPT).

   Inter-packet time gap is the time that elapses between two consecutive packets. First, an average E and a standard deviation σ of IPT values are calculated. Then, outliers are rejected by dropping all values greater than $E + 1.645\,\sigma$. This assumes that IPT follows the standard distribution and thus rejects about 5% of the input values. Finally, the average value is recomputed and used as the coordinate value.
3. Average differences in IPT.

   This coordinate holds the average difference between two consecutive IPT values, after outlier detection. It is to mimic the concept of packet jitter.
4. Numeric representation of the transport protocol.

Motivation behind introducing these 4 coordinates is to improve the performance of the resultant system. Undisturbed values of these "flow-level" characteristics are usually available in a typical network scenario, and can contribute to the discriminating power of the feature vectors.

2.1.3.3 Complex decision process

In the original algorithm, a single endpoint can be classified many times, once for each signature window. In the thesis however, three reconciliation algorithms, as suggested in sect. IV-B of [Fin09], were implemented.

1. "Simple" - uses the last classification directly, as in the original algorithm.
2. "Best" - uses the classification with the highest probability so far.
3. "EWMA" - tracks the *Exponentially Weighted Moving Average* of classification probability for each possible outcome, and uses the one with the highest value seen so far.

2.1.3.4 Unknown protocol detection

A potential drawback of KISS is that it requires good quality "Background" traffic samples during the learning phase (sect. V-C in [Fin09]), in order to detect the applications that are unknown to the classification system.

In the thesis, this was solved using a special functionality of the *libsvm* library (see 3.2). For each classification, a vector with relative class membership probability is obtained. The sum of probabilities is always equal 1, so in order to give the vector values an absolute meaning, the difference between the first and the second highest value is calculated, and the result is treated as the *classification probability*. An unknown protocol is detected by introducing a simple threshold mechanism. If the classification probability is below the threshold, classification is considered "not certain enough" and rejected.

2.2. System architecture

Previous section describes the main algorithm with its two stages of feature extraction and decision process. In order to implement it as a practical system, these stages have to be decomposed, and the whole algorithm needs to be complemented with a few elements, as presented on Fig. 2.2.

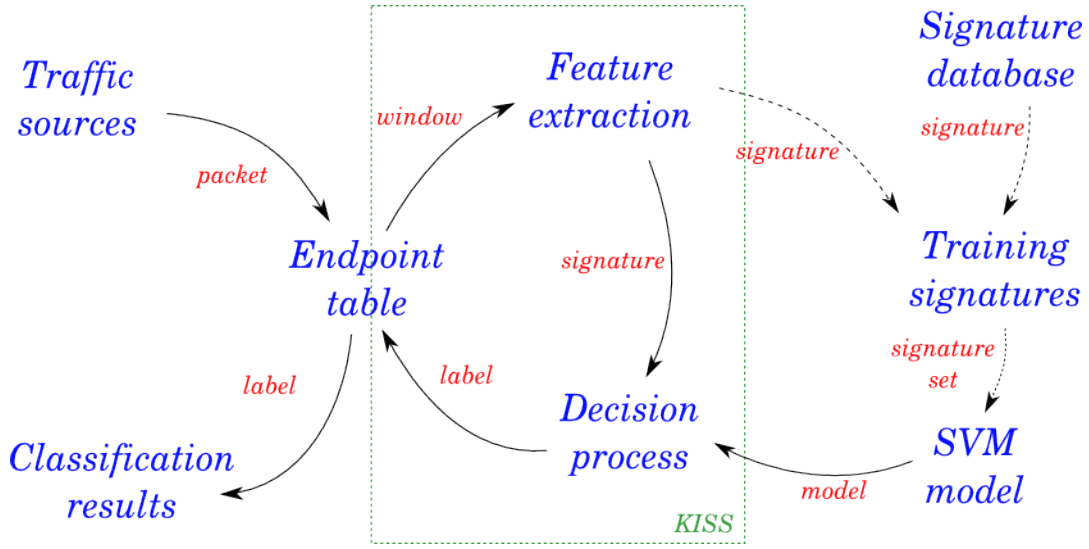

Fig. 2.2: ***General system architecture.*** *System elements presented in blue. Arrows show data paths, with data kinds as red labels. Dashed arrows represent paths used for system training. Green frame roughly shows elements defined in the original KISS algorithm.*

The system consists of the following elements:

1. *Signature database*
2. *Training signatures*
3. *SVM model*
4. *Traffic sources*
5. *Endpoint table*
6. *Feature extraction*
7. *Decision process*
8. *Classification results*

Following subsections describe these components.

2.2.1. Signature database

The process of obtaining signatures from IP packets can be computationally intensive (recall 2.1.1). For instance, if one wants to train the system so it is able to properly recognize 20 applications, for good-quality results about 500 signatures need to be extracted for each of them, giving a total of almost 1 million packets.

This "knowledge" of the system is lost once the program is terminated. However, the process of learning it once again can be sped up by saving the extracted signatures (annotated with labels) on a hard disk. Hence, the next time the program is run, no time-consuming packet analysis is required. Signatures are immediately loaded into the program memory.

This is the task of the Signature database. It reads signatures from hard disk and feeds them into the Training signatures element. Furthermore, new signatures that will be obtained through IP packet analysis during program execution will be stored in the database, too.

2.2.2. Training signatures and the SVM model

Before SVM is able to make classifications, it needs to be trained with samples annotated with the target class (recall 2.1.2). This is also a computationally intensive task.

Moreover, the model can not be incrementally updated. Even a single new learning signature can cause the whole model to be recomputed. In a real-time system, this could happen many times per second.

The task of the Training signatures element is to be a kind of a buffer to the SVM model update process. It ensures that the model is recomputed at most once per specified amount of time, e.g. 10 seconds. Thus, there is a chance that several new signatures are buffered before the update process is run.

2.2.3. Traffic sources

Traffic sources supply the system with IP packets. For off-line classification, it is a file with IP packets stored in the PCAP format [PCAP]. For real-time operation, it is a network interface – e.g. an Ethernet NIC – on which all the traffic passing through it is captured.

The task of this element is to provide an interface to access these underlying packet sources in a common way. At the same time, many packet sources can work simultaneously, some of which being off-line sources, and some real-time. All packets are passed to the Endpoint table.

Usually, a traffic source supplies packets of unknown applications. If the application is known, packets are used for training. In such case, they do not enter the Decision process, but supplement the Training signatures. Alternatively, they follow the standard path of classification, but the Classification results element will verify if the system produced a valid answer, thus giving an insight into the system performance.

2.2.4. Endpoint table

The Endpoint table element is the central component of the whole system. It has two fundamental tasks of gathering IP packets in groups of endpoints and collecting classification decisions made on them.

Special care needs to be taken for off-line packet sources. It may happen, that packets stored in two files are distant in time – for instance, packets captured in year 2011 and 2008. For this reason, packets from two off-line sources can not be grouped together in a single endpoint. For real-time sources, this is allowed.

The table is periodically swept by a *garbage collector*, which removes all endpoints, for which the last packet is older than 5 minutes. For off-line sources, the time distance to the last packet received from this source is checked.

The Endpoint table feeds the Feature extraction element once at least C=80 packets (see 2.1.1) are collected in a single endpoint.

2.2.5. Feature extraction and the decision process

The task of these two elements was described in section 2.1. Decision process requires an already prepared, valid SVM model. If it is absent, no classification can be made.

The result is stored in the Endpoint table as an endpoint property. Whenever the value of this property changes, the Classification results element is notified.

2.2.6. Classification results

The task of this element is to inform the system user that an application identity was recognized at a given Internet endpoint. The resultant numeric label used internally (e.g. "7") is translated into string representation (e.g. "Skype"). Name of the traffic source is also given (e.g. "eth0" or "~/traces/skype.pcap"). In case the application behind the traffic source is known, the result will be verified and used to compute the system performance metrics.

2.3. Methodology

There are two possible kinds of a traffic classifier output – its result is either valid or invalid. This will be the fundamental field of performance evaluation in the thesis, as described below (based on [Fin09]).

Two notions of a True/False and a Positive/Negative need to be presented. Endpoint classification is *True* if it is valid, and *False* otherwise. All classifications belonging to the particular application identity are *Positive*, and others are *Negative*. Thus, classifier output can be either a True Positive (TP) or True Negative (TN) if it is valid, and False Positive (FP) or False Negative (FN) otherwise.

Now, four metrics can be introduced:

- False Positive Percentage for application identity x

$$\%FP_x = 100 \cdot \frac{FP_x}{\bar{s}_x} \qquad (1.1)$$

  - FP_x is number of False Positives for x
  - $\bar{s}_x$ is number of endpoints not belonging to x

- False Negative Percentage for x

$$\%FN_x = 100 \cdot \frac{FN_x}{s_x} \qquad (1.2)$$

  - FN_x is number of False Negatives for x
  - s_x is number of endpoints belonging to x

- True Positive Percentage for x is $\%TP_x = 100 - \%FN_x$ (1.3)
- True Negative Percentage for x is $\%TN_x = 100 - \%FP_x$ (1.4)

For instance, if there are 100 “Skype” endpoints and the classifier says 10 of them are “BitTorrent”, then $\%FN_{Skype} = 10$.

Finally, the system performance can be evaluated using traffic samples of applications, whose identities are already known. This knowledge of true application identities is called *ground truth*. It can be obtained in several ways, for instance using a DPI packet classifier like [L7] or a simple TP port classifier.

Samples are fed into the system and the result is compared to the ground truth. Observation of two metrics – %TP and %FP – and their statistics allows to make conclusions on the performance of the system. A well-performing system is characterized by high %TP values while keeping the %FP metric as low as possible.

3. IMPLEMENTATION

A *real-time* traffic classification system requires a robust and fast implementation. Nowadays, backbone links of ISP companies often carry hundreds of thousands of packets per second. This gives just a few microseconds for handling each packet.

Moreover, the goal of the thesis is to provide a *practical* system. Thus, it needs to be directly usable for many tasks. This includes application at a typical ISP company, which wants to e.g. block P2P traffic and computer viruses, while prioritizing Internet telephony. In such scenario, resultant software will be just a constituent element of a greater firewall system. On the other hand, a government agency whose mission is to discover society trends, will most likely want to work on sets of off-line traffic files, probably in a graphical environment. Again, the classification element will be just a part of a GUI application. These two examples show that an implementation of a traffic classifier needs to be flexible, portable and embeddable.

The system introduced in the thesis works under Linux operating system, which belongs to the family of UNIX-like systems and heavily relies on Open Source software. This implicitly means that the implementation needs to obey the rules set by other open source systems, and by general guidelines for programs working in UNIX environments.

This chapter discusses implementation of the thesis software in detail. Section 3.1. gives an introductory view on the program architecture, section 3.2. references external libraries that the system relies on. Sections 3.3. and 3.4. give detailed documentation of the source code.

3.1. Architecture

The system is entirely implemented in C language under the Linux operating system, on a PC system. However, this does not pose a tight limit on the area of its possible applications. It would require little or no modification to port the system to another UNIX-like platform, including embedded systems.

The system works as a single-threaded process with one global event loop. The loop is used for external communication and for exchange of internal messages. Such architecture enables quasi-parallel processing. For instance, handling incoming IP packets is possible while making classification decisions at the same time.

Compared to a multi-threaded architecture, such approach greatly simplifies the whole system, but under some circumstances it might cause higher latency, i.e. the time that lapses between the moment in which a new packet enters the system and the event of its classification. Specifically, this would happen on a multiprocessor machine. However, in such case, each CPU could be assigned different part of the IP addressing space, as presented on Fig. 3.1.

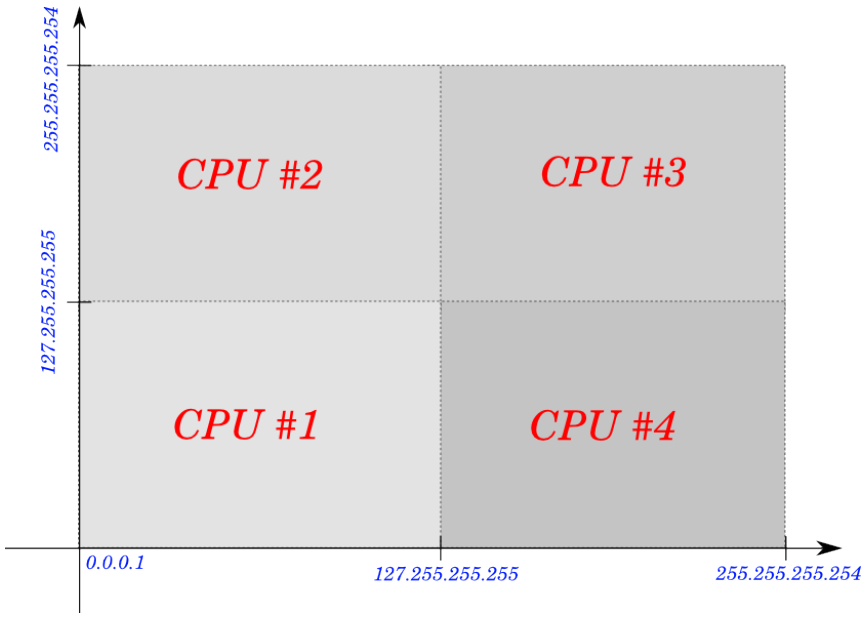

Fig. 3.1: ***Classification on a multiprocessor machine.*** *Each CPU handles different part of the IPv4 addressing space.*

Functionality of the system is divided into multiple modules. In many cases, no direct procedure call is possible between the modules. Instead, a notification mechanism is used in which one functional element informs another, for example, that a new signature window is ready to be classified. This is realized by means of *internal events* (or *messages*), transported by the main program loop. The source module *announces* that some event happened, while there can be any number of *subscribers*, which will handle the event, with some time delay. Event announcements are queued, so this delay depends on the current queue length and its processing speed.

Still, program modules can write to shared memory, so there is a direct data communication path by means of variables. No locking or synchronization mechanisms are needed, as the program is single-threaded. Indeed, all modules have direct access to a common instance of the fundamental data structure: `struct spi` (see 3.3.2.1).

The classification system is almost entirely implemented as a library, called *libspi*, where "spi" stands for *Statistical Packet Inspection*. A simple command-line *front-end* program utilizing this library was also created, named *spid*.

Modular architecture of the system lets for replacement of the main algorithm (i.e. the feature extraction phase and the basic decision process, 2.1) and the reconciliation algorithm (i.e. the complex decision process, 2.1.3.3). Special efforts were made in order to make such a potential modification to the system easy and straightforward.

3.2. External libraries and facilities

The system relies on several external components.

For implementation of the main program loop, the *libevent* [Pro00] library is used. It handles the task of monitoring traffic sources for new packets, periodically executes the endpoint table garbage collector and is used as a queue for internal events.

The network traffic is obtained via the *libpcap* library [PCAP]. It reads packet trace files and captures live traffic from network interfaces. However, the latter depends on packet capture facilities delivered by the operating system, which will deny access to the data without adequate privileges.

As a general tool set, a library based on *libasn* [For05] is adopted. It provides basic data structures of hashing table and linked list, memory management utilities and some mathematical functions in form of C pre-processor macros.

For realization of the SVM decision process, a popular implementation of *libsvm* [CC01a] is employed. For training, its `svm_train()` function is used, and `svm_predict_probability()` for classification with probability output.

3.3. Main program: the *libspi* library

3.3.1. File list

libspi source code consists of the following files:

- `datastructures.h` : declarations of data types and structures used throughout the system and in the API
- `settings.h`: holds pre-processor constants used as main algorithm parameters and some of the fundamental options of the whole system
- `spi.h`: declarations of the API – public functions exported by the library
- `spi.c`: main program (see 3.3.3.1); implements initialization procedure, management of traffic sources, internal event system, garbage collector, training signature queues, and memory management routines
- `source.h`: declares `source.c` functions that can be called internally
- `source.c`: implements two traffic sources; PCAP files and network interface sniffing via the *libpcap* library; holds a generic IP packet parser which is the source of signature windows
- `flow.h`: declares `flow.c` functions that can be called internally
- `flow.c`: TCP flow table; implements the P limit (recall 2.1.1) and detects RST/FIN flags which close connections
- `ep.h`: declares `ep.c` functions that can be called internally
- `ep.c`: the endpoint table; implements storage of new packets

- `kissp.h`: declares `kissp.c` functions and data structures that are used internally
- `kissp.c`: implements the extended KISS algorithm; connects to the *libsvm* library
- `verdict.h`: declares `verdict.c` functions and data structures
- `verdict.c`: implements the complex decision process (see 2.1.3.3): "simple", "best" and "EWMA" algorithms; produces final verdict of endpoint classification

3.3.2. Data structures and variables

The `datastructures.h` header file contains declarations of data structures, with a few declarations put in `kissp.h` and `verdict.h` files. This subsection gives documentation of the most important data structures and their application as variables.

3.3.2.1 Main structure: `struct spi`

The `struct spi` is a root data structure, instance of which is passed to every function. Conceptually, this is similar to the special variable `this` used in the C++ object-oriented programming language. Every data piece used by the program can be reached from this structure. Each instance of `struct spi` thus represents an instance of the whole *libspi*. Structure synopsis presented on Listing 3.1. See 7.1.1 for details.

```
1.    struct spi {
2.        mmatic *mm;
3.        struct spi_options options;
4.        bool running;
5.        bool quitting;
6.        struct event_base *eb;
7.        struct event *evgc;
8.        thash *subscribers;
9.        tlist *sources;
10.       thash *eps;
11.       thash *flows;
12.       tlist *traindata;
13.       tlist *trainqueue;
14.       struct spi_stats stats;
15.       void *cdata;
16.       void *vdata;
17.   };
```

*Listing 3.1: **`struct spi`**. Main data structure.*

3.3.2.2 Internal events: `spi.subscribers`, `struct spi_subscribers`, `spi_event_cb_t` and `struct spi_event`

Inter-module control flow is realized by means of internal events (see 3.1). Each event is uniquely identified by its name, e.g. `classifierModelUpdated`. Event subscriptions are registered in the `subscribers` member of `struct spi`, which is a hashing table. For table keys, event names are used, so multiple subscriptions to the same event are stored in the same place. Each table value is an instance of `struct spi_subscribers`.

The main task of `struct spi_subscribers` is to hold references on event handler functions (so-called *callbacks*). Each callback address is stored in a data type of `spi_event_cb_t`. Synopsis of both of them is given on Listing 3.2. See 7.1.2 for details.

```
1.    struct spi_subscribers {
2.        tlist *hl;
3.        tlist *ahl;
4.        enum spi_aggstatus {
5.            SPI_AGG_DISABLED = 0,
6.            SPI_AGG_READY,
7.            SPI_AGG_PENDING
8.        } aggstatus;
9.    };
10.   typedef bool spi_event_cb_t(
11.       struct spi *spi,
12.       const char *evname,
13.       void *arg
14.   );
```

*Listing 3.2: **`struct spi_subscribers` and `typedef spi_event_cb_t`**. Structures used for handling of internal event subscriptions.*

Whenever an internal event is announced, an instance of `struct spi_event` is created, holding data which will be later used during event handling. This instance is finally stored in an event queue managed by *libevent*. Synopsis given on Listing 3.3:

```
1.    struct spi_event {
2.        struct spi *spi;
3.        const char *evname;
4.        struct spi_subscribers *ss;
5.        void *arg;
6.        bool argfree;
7.    };
```

*Listing 3.3 **`struct spi_event`**. Representation of an event announcement.*

3.3.2.3 IP traffic: `struct spi_source`, `spi_source_t` and `struct spi_pkt`

A traffic source is represented by `struct spi_source`. It holds information on the underlying *libpcap* source of packets, packet counters, etc. Source type is represented by `spi_source_t`. Synopsis given on Listing 3.4. Refer to 7.1.3 for details.

```
1.    typedef enum {
2.        SPI_SOURCE_FILE = 1,
3.        SPI_SOURCE_SNIFF
4.    } spi_source_t;
5.    struct spi_source {
6.        struct spi *spi;
7.        spi_source_t type;
8.        spi_label_t label;
9.        bool testing;
10.       int fd;
11.       struct event *evread;
12.       unsigned int counter;
13.       unsigned int signatures;
14.       unsigned int learned;
15.       unsigned int eps;
16.       bool closed;
17.       union {
18.           struct {
19.               pcap_t *pcap;
20.               const char *path;
21.               struct timeval time;
22.               struct timeval gctime;
23.           } file;
24.           struct {
25.               pcap_t *pcap;
26.               const char *ifname;
27.           } sniff;
28.       } as;
29.   };
```

Listing 3.4 ***struct spi_source.*** *Representation of a traffic source.*

Incoming packets are stored in `struct spi_pkt` – see Listing 3.5:

```
1.    struct spi_pkt {
2.        uint8_t *payload;
3.        struct timeval ts;
4.        uint16_t size;
5.    };
```

Listing 3.5: ***struct spi_pkt****. Representation of an IP packet.*

3.3.2.4 Endpoints: `spi_epaddr_t`, `spi.eps`, and `struct spi_ep`

Each endpoint can be uniquely identified by a tuple of *{TP protocol, IP address, TP port}*. In source code, such tuple – referred to as an *endpoint address* – is stored in a special variable type of `spi_epaddr_t`.

The `spi.eps` hash table tracks all active endpoints. As table keys, endpoint addresses are used, and table values are instances of `struct spi_ep`. This structure gathers various endpoint data – packet list and classification verdict being the most important ones. Synopsis given on Listing 3.6, with details available in 7.1.4.

```
1.    typedef uint64_t spi_epaddr_t;
2.    struct spi_ep {
3.        mmatic *mm;
4.        struct spi_source *source;
5.        spi_epaddr_t epa;
6.        struct timeval last;
7.        tlist *pkts;
8.        bool gclock;
9.        uint32_t predictions;
10.       spi_label_t verdict;
11.       double verdict_prob;
12.       uint32_t verdict_count;
13.       void *vdata;
14.   };
```

*Listing 3.6: **`spi_epaddr_t` and `struct spi_ep`**. Representation of a traffic endpoint.*

The task of `spi_epaddr_t` is to store three constitutive properties of an endpoint address in a single, 64-bit value. In C language, it is constructed in the following way:

```
1.    spi_epaddr_t epa = (proto << 48) | (ip_addr << 16) | port;
```

Where `proto`, `ip_addr` and `port` are: the transport protocol, the IPv4 address and the TP port, respectively.

3.3.2.5 Signatures: `struct spi_signature` and `spi_label_t`

Each application identity can be uniquely identified by its numeric label, which is stored in a special data type of `spi_label_t` – an 8-bit integer. This limits the label range to 0 – 255. Value of 0 is regarded as "unknown identity". Window signature data is represented by `struct spi_signature`. Synopsis on Listing 3.7, details in 7.1.5.

```
1.    typedef uint8_t spi_label_t;
2.    struct spi_signature {
3.        spi_label_t label;
4.        struct spi_coordinate { int index; double value; } *c;
5.    };
```

*Listing 3.7: **spi_label_t and struct spi_signature**. Representation of a window signature.*

3.3.2.6 Classification results: `struct spi_classresult` and `spi_cprob_t`

The output of SVM classification is stored in `struct spi_classresult`. Functionality of membership probability of *libsvm* is used, which requires additional storage place. This is handled by variables of `spi_cprob_t` type. Synopsis given on Listing 3.8. See 7.1.6 for details.

```
1.    typedef double spi_cprob_t[SPI_LABEL_MAX + 1];
2.    struct spi_classresult {
3.        struct spi_ep *ep;
4.        spi_label_t result;
5.        spi_cprob_t cprob_lib;
6.        spi_cprob_t cprob;
7.    };
```

*Listing 3.8: **spi_cprob_t and spi_classresult**. Representation of SVM classification results.*

Indeed, `spi_cprob_t` is a 256-element array of `double`. Cell *i* holds a floating-point number in range 0.0 – 1.0: a relative probability that the input vector belongs to class *i*.

3.3.2.7 Performance evaluation: `struct spi_stats`

Data required for performance assessment (see 2.3) are collected in `struct spi_stats`. It contains counters, which may be later used for calculation of the %TP and %FP metrics. Synopsis given on Listing 3.9, details in 7.1.7.

```
1.    struct spi_stats {
2.        uint32_t learned_pkt;
3.        uint32_t learned_tq;
4.        uint32_t test_all;
5.        uint32_t test_is[SPI_LABEL_MAX + 1];
6.        uint32_t test_ok;
7.        uint32_t test_FN[SPI_LABEL_MAX + 1];
8.        uint32_t test_FP[SPI_LABEL_MAX + 1];
9.    };
```

*Listing 3.9: **spi_stats**. Collection of system performance counters.*

3.3.2.8 KISS algorithm: `struct kissp`

The system is designed in such a way, that application of a different method for the main algorithm would be possible, hence the pointer at `spi.cdata` is of generic `void *` type. However, it holds a pointer to the structure representing the KISS algorithm used in the thesis – `struct kissp`. Synopsis on Listing 3.10. Refer to 7.1.8 for details.

```
1.    struct kissp {
2.        int feature_num;
3.        struct { bool pktstats; } options;
4.        struct {
5.            struct svm_model *model;
6.            struct svm_parameter params;
7.            int *labels;
8.            int nr_class;
9.        } svm;
10.   };
```

*Listing 3.10: **`struct kissp`**. Internal data of the modified KISS algorithm.*

3.3.2.9 Complex decision process: `struct verdict` and `struct ewma_verdict`

The two structures of `struct verdict` and `struct ewma_verdict` hold data of the reconciliation algorithm (see 2.1.3.3) for the whole system and for each endpoint, respectively. Synopsis given on Listing 3.11, and details in 7.1.9.

```
1.    struct ewma_verdict {
2.        spi_cprob_t cprob;
3.    };
4.    struct verdict {
5.        enum {
6.            SPI_VERDICT_SIMPLE,
7.            SPI_VERDICT_EWMA,
8.            SPI_VERDICT_BEST
9.        } type;
10.       struct {
11.           uint16_t N;
12.       } ewma;
13.   };
```

*Listing 3.11: **`struct ewma_verdict`** and **`struct verdict`**. Data of the complex decision process.*

3.3.3. Control flow and events

3.3.3.1 System initialization

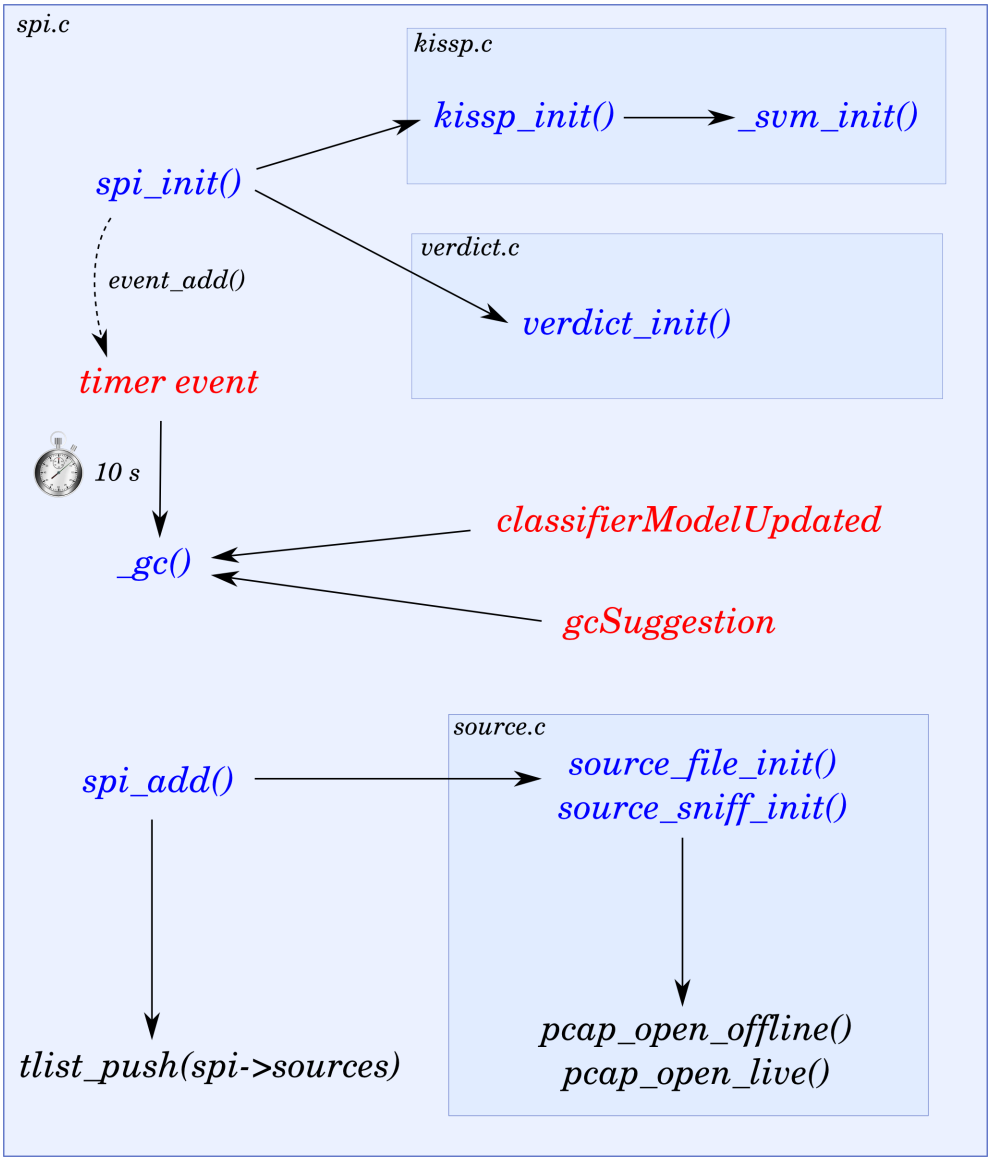

Fig. 3.2: ***spi.c file.*** *Procedures of library initialization, garbage collector and of adding a traffic source.*

The `spi.c` file holds several important procedures, including the `spi_init()` function, which implements the task of system initialization (see Fig. 3.2). It prepares an instance of `struct spi` which may be further used. During this process, the main algorithm structure is initialized by means of `kissp_init()` and `_svm_init()` functions,

latter of which sets up parameters of the *libsvm* library. The `verdict_init()` function is called in order to prepare the reconciliation algorithm variables.

During initialization, the *libevent* library is prepared for event handling. This includes setting up execution of the garbage collector each 10 seconds, which is implemented in function named `_gc()`. This function will iterate through all entries in `spi.eps` and `spi.flows`, and remove old entries, i.e. those, for which there were no new packets for more than 5 minutes. The garbage collector is also run if either the `classifierModelUpdated` or the `gcSuggestion` event is announced.

New traffic sources can be added using a function named `spi_add()`. Depending on the type of the source, it calls either `source_file_init()` for off-line traffic files or `source_sniff_init()` for network interfaces. As the result, functions of the *libpcap* library are called, respectively either `pcap_open_offline()`, or `pcap_open_live()`. This operation opens a new file descriptor, which is monitored for new data available to be read, using the *libevent* library. Finally, the source is appended to the list of system traffic sources, located in `spi.sources` (see 3.3.2.1).

3.3.3.2 Route of a packet

Figure 3.3 presents the route of a packet in the system.

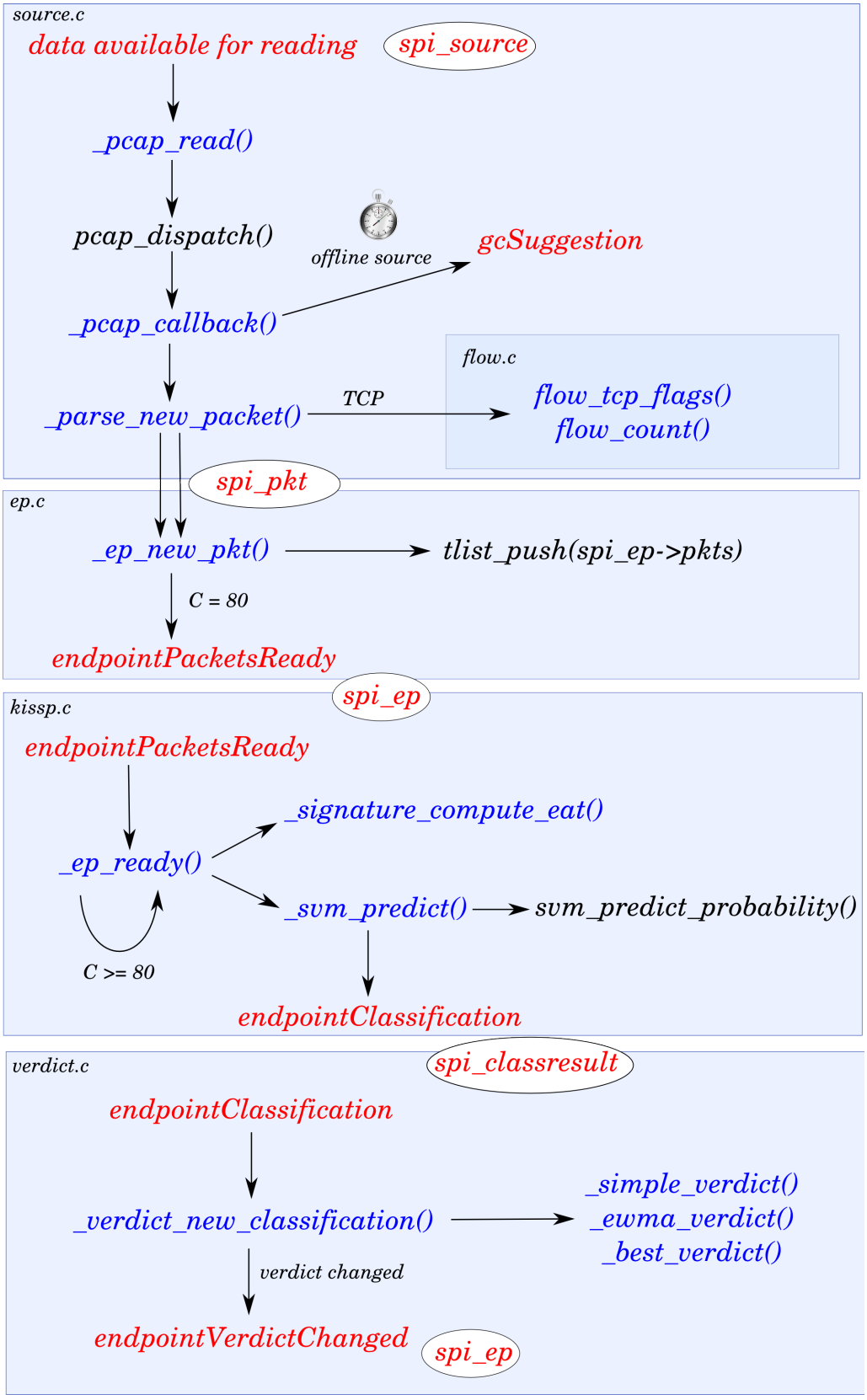

Fig. 3.3: ***Flow of an IP packet.*** *Route from entering the system till the final classification.*

First, when a packet arrives on the traffic source monitored by *libevent*, an event is generated. It is handled in `_pcap_read()`, which calls a *libpcap* function named `pcap_dispatch()` in order to fetch the packet. This function can read many packets at once, calling `_pcap_callback()` for each of them.

If the packet is received from an off-line source – i.e. a traffic file – then a special check is made if the virtual time in file is more than 10 seconds since last execution of the garbage collector. In such case, an event of `gcSuggestion` is generated.

Then the packet is parsed, and the values inside IP and TP headers are extracted. In case the packet belongs to a TCP stream, `flow_tcp_flags()` and `flow_count()` functions are called for TCP flow tracking. Then, if the packet is long enough, its data are stored in the endpoint table. They are passed to `_ep_new_pkt()` twice – for the source endpoint and for the destination endpoint.

In `_ep_new_pkt()`, packet data are copied to a new instance of `struct spi_pkt` and appended to the list of endpoint packets. If there are at least 80 packets on the list, a new event of `endpointPacketsReady` is announced, along with a pointer to the endpoint that generated the event.

This event is handled by `_ep_ready()` in the `kissp.c` file. In a loop, C=80 packets are consumed by the feature extractor implemented in `_signature_compute_eat()`, which produces a window signature. It is passed to `_svm_predict()` which invokes `svm_predict_probability()` function of the *libsvm* library and announces the result with an event of `endpointClassification`, with relevant data put in an instance of `struct spi_classresult`.

Finally, the reconciliation algorithm is run, whose task is to join many classifications of the same endpoint into a single result. The `_verdict_new_classification()` function handles the `endpointClassification` event, calling implementation of the appropriate method. If the final result is different from the current endpoint classification, a new event of `endpointVerdictChanged` is announced, along with a pointer to the endpoint that caused the event.

3.3.3.3 System training

Figure 3.4 presents the process of training the system using IP packets.

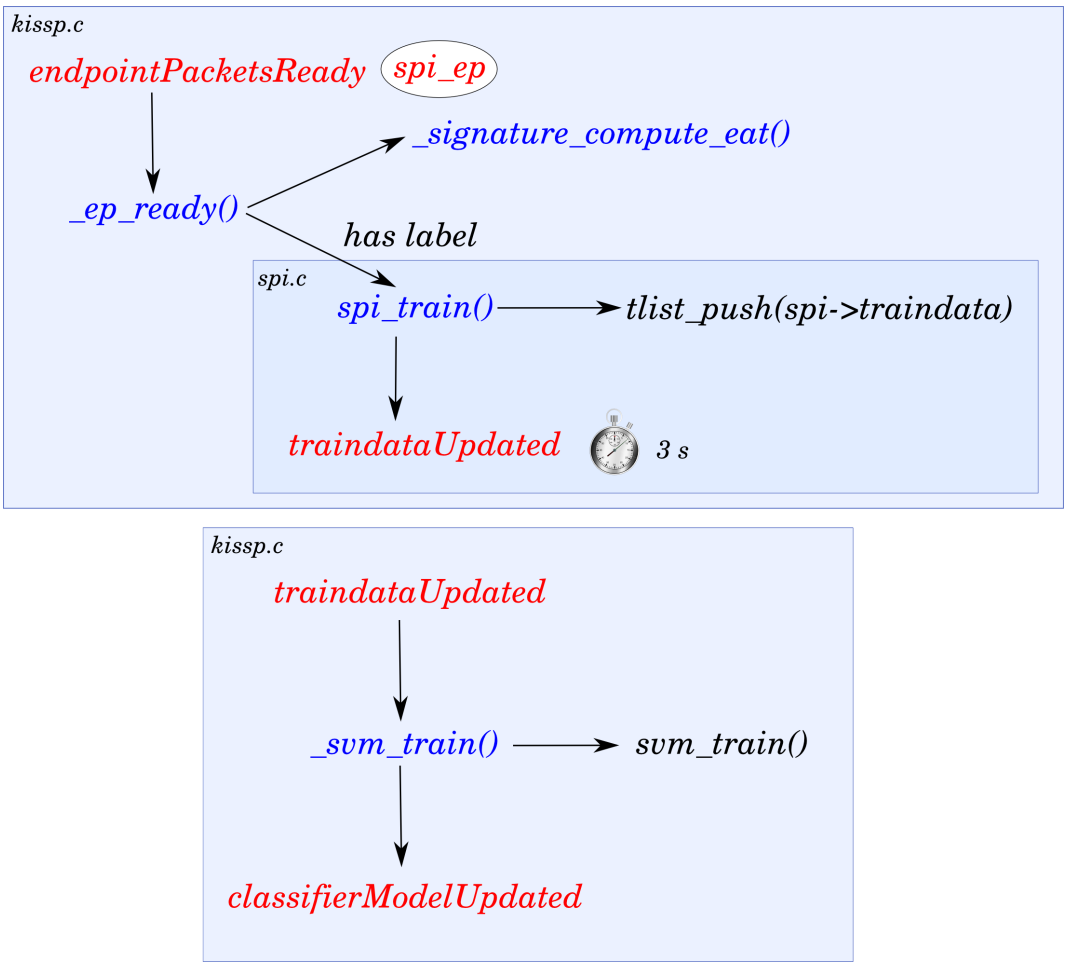

Fig. 3.4: ***System training****. Resultant signature is used in the SVM model with a delay.*

If a traffic source has a label, the application identity of all its IP packets is known. Hence, when a window signature is extracted, it may be used for training. The buffer of training signatures is supplemented through the `spi_train()` function. When a new signature is appended to the list, a delayed event of `traindataUpdated` is announced. There are 3 seconds for further updates from the moment in which the first signature extends the list.

Once the event is delivered, it is handled by `_svm_train()`, which creates a new SVM model by calling `svm_train()` from the *libsvm* library.

3.3.4. Application Programming Interface

Functionality of *libspi* can be accessed from external programs using its Application Programming Interface, which is stored in `spi.h` file. Listing 3.12 presents the set of essential API functions. Refer to 7.2 for a detailed description of the API.

```
1.    struct spi *spi_init(struct spi_options *so);
2.    void spi_free(struct spi *spi);
3.    int spi_add(struct spi *spi, spi_source_t type,
4.        spi_label_t label, bool test, const char *args);
5.    int spi_loop(struct spi *spi);
6.    void spi_stop(struct spi *spi);
7.    void spi_announce(struct spi *spi, const char *evname,
8.        uint32_t delay_ms, void *arg, bool argfree);
9.    void spi_subscribe(struct spi *spi, const char *evname,
10.       spi_event_cb_t *cb, bool aggregate);
11.   void spi_train(struct spi *spi, struct spi_signature *sign);
12.   void spi_trainqueue(struct spi *spi, struct spi_signature *sign);
13.   void spi_trainqueue_commit(struct spi *spi);
14.   double spi_stats_fp(struct spi *spi, spi_label_t label);
15.   double spi_stats_fn(struct spi *spi, spi_label_t label);
```

*Listing 3.12: **libspi API**. Set of essential functions.*

3.4. Front-end: the *spid* program

As a part of the thesis, a simple front-end program using *libspi* was written. Its task is to make the *libspi* functionality accessible from the command-line.

3.4.1. File list

Source code consists of the following files:

- `spid.h`: holds data structures and forward function declarations of `spid.c`
- `spid.c`: the main program; communication with the *libspi* library and display of classification results
- `samplefile.h`: forward declarations of `samplefile.c` functions
- `samplefile.c`: implementation of functions for reading and writing files with signatures

3.4.2. Data structures

Program data structures are defined in `spid.h` file. Synopsis of the two structures – `struct spid` and `struct source` – given below on Listing 3.13.

```
1.    struct source {
2.        char *proto;
3.        char *cmd;
4.        bool test;
5.    };
6.    struct spid {
7.        struct mmatic *mm;
8.        struct spi *spi;
9.        struct spi_options spi_opts;
10.       thash *proto2label;
11.       thash *label2proto;
12.       tlist *learn;
13.       tlist *detect;
14.       struct options;
15.   };
```

*Listing 3.13: **`struct spid and struct source`**. Two data structures of the spid program.*

`struct source` represents a traffic source, and the task of `struct spid` is similar to `struct spi` – to be the main instance data structure. See 7.3 for details.

3.4.3. Control flow and communication with *libspi*

Program control flow is presented on Fig. 3.5.

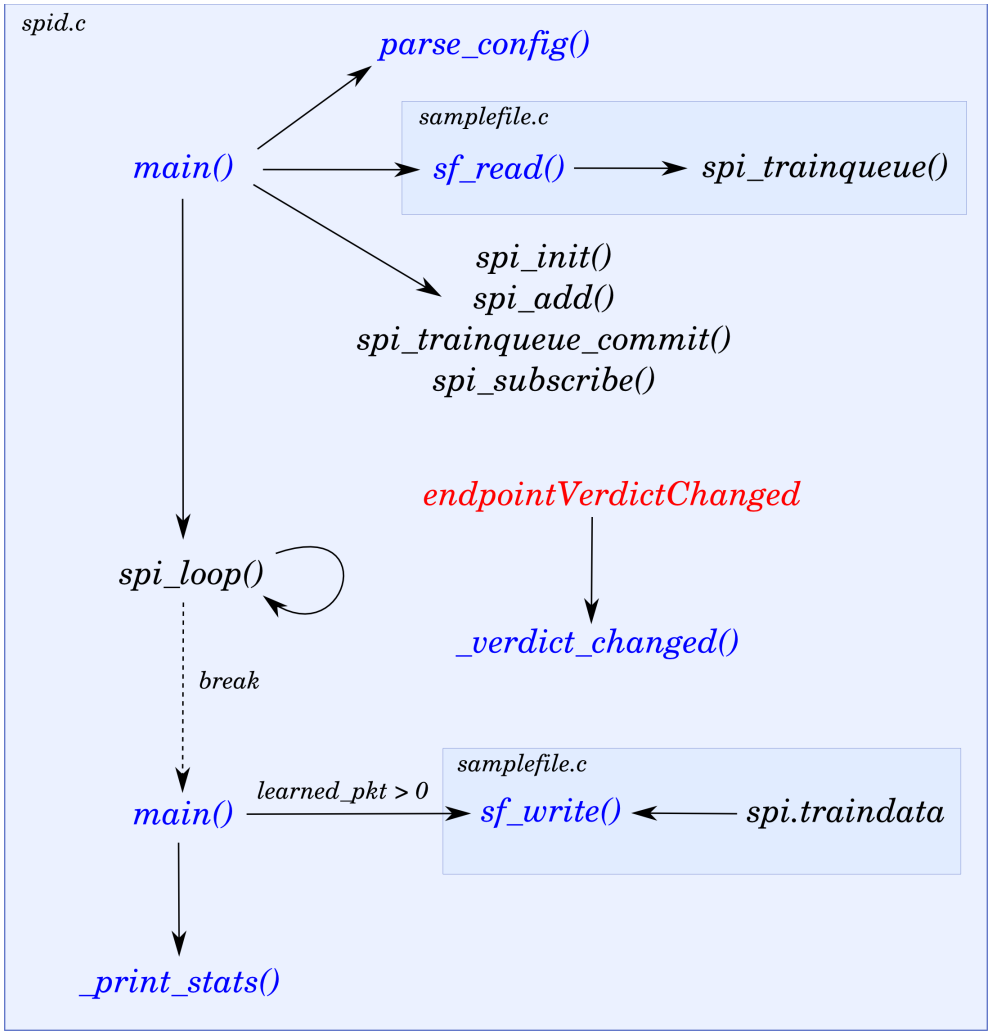

Fig. 3.5: ***spid control flow****. Program is constructed around the main libspi loop.*

Execution starts in the `main()` function, which after data structure initialization calls `parse_config()`. Command-line arguments and any configuration files referenced in program invocation are parsed, and the control is returned to `main()`.

In case a signature database file is given, a parser is called, implemented by `sf_read()` in `samplefile.c`. This function reads the database line-by-line and calls `spi_trainqueue()` for each signature properly read.

A set of *libspi* functions is used in order to set up the classification system properply before it is started. Subscriptions to *libspi* events are made, and finally the main loop is started by means of `spi_loop()`.

Each time the system issues a new endpoint classification, the `_verdict_changed()` function is called. The result is displayed on the standard output, in a form readable by a human operator.

When the system is stopped – e.g. due to the end of packets in all traffic sources – program control is returned to `main()`. If the system learned new signatures from IP packets, the signature database file is rewritten by `sf_write()`, which stores all signatures from `spi.traindata` (see 3.3.2.1) on disk.

Finally, system performance statistics are printed to the standard output – see Błąd: Nie znaleziono źródła odwołania.

3.4.4. User interface

The *spid* program is invoked from the command-line, according to the following syntax:

```
1.    spid [OPTIONS] [<traffic sources...>]
```

Where *OPTIONS* is a list of options, from the following set:

- `--learn=<lspec>`: train the system according to `<lspec>` (see 7.4.1)
- `--learndb=<file>`: train the system according to given index file (see 7.4.2)
- `--signdb=<file>`: use given signature database file (see 7.4.3)
- `--test=<lspec>`: test the system according to `<lspec>`
- `--testdb=<file>`: test the system according to given index file
- `--kiss-std`: disable KISS signature extensions (see 2.1.3.2)
- `--verdict-threshold=<t>`: ignore classifications with probability below `<t>`% (see 2.1.3.3)
- `--verdict-simple`: use the "Simple" method as decision reconciliation algorithm (EWMA is used by default)
- `--verdict-best`: use the "Best" method

- `--verdict-ewma-len=<n>`: set the number of samples for “EWMA” method
- `--stats`: print system performance metrics when program finishes
- `--print-probs`: include probability information in classification output
- `--debug=<n>`: set debugging level to `<n>`; this enables output of internal diagnostic messages
- `--help`: show short usage manual

After *OPTIONS*, there is a space-separated list of traffic sources for detection. Each entry must conform to the command-line source specification format (see 7.4.1).

Refer to 7.4.4 and 7.4.5 for documentation of the output format.

4. EVALUATION

This chapter describes system performance evaluation using metrics defined in 2.3. Four practical experiments are made and their outcomes are presented.

Although the system is capable of working in real-time, i.e. capturing IP traffic on network interfaces, all performance evaluation tests were made in off-line mode, using trace files. Such approach gives meaningful results for all kinds of traffic sources. Moreover, an off-line evaluation test-bed is easier to develop and supervise.

Section 4.1 gives description of datasets used for experiments, i.e. traces of IP traffic. Section 4.2 defines the tests and gives their results, with discussion in the last Section 4.3.

4.1. Datasets

For evaluation of the system performance, three datasets were used:

- *Trace1*: personal dataset, collected by thesis author
- *Trace2:* Skype UDP traces, collected in a test-bed environment
- *Trace3*: IP-TV traces, collected in a real network

Trace1 is result of a constant packet capture on a typical desktop computer for 30 days – July till August 2011. Dataset consists of 144 files, about 100 thousand IP packets each, of total size 1GB.

Trace2 and *Trace3* were obtained from the *Tstat* project [Tstat], held by the authors of [Fin09] and [Fin10]. Both datasets were collected and organized with the support of the *Robust and Efficient traffic Classification in IP nEtworks* [RECIPE] and *MIsure sperimentali e MOdelli di traffico dati multiServizio A pacchetto* [MIMOSA] projects.

Trace2 was created by merging the first 9 files available for download from the Tstat Skype Testbed Traces [TstatSkype], obtaining a file of 148MB.

Trace3 was created by extracting the first 200 000 packets from the first file available for download from the Tstat Multicast IP-TV Traces [TstatIPTV], resulting in a file of 14MB. IP packets contained in this dataset were collected in a real network of an Italian company FastWeb, located in Torino. Captured traffic are multicast IP-TV transmissions encoded with MPEG-2 algorithms, encrypted and encapsulated in a proprietary UDP protocol.

For all packets in *Trace2* and *Trace3* the application identities are known. For *Trace1*, a simple packet classification using port matching (see 1.3.1) was adopted. As a result, packets belonging to the following applications were extracted:

- *dns:* BPF ([BPF]) filter of `udp and port 53`
- *openvpn*: `udp and port 1194`
- *bittorrent*: `tcp and port 51413`
- *http*: `tcp and port 80`

- *https*: `tcp and port 443`

Finally, traces of 7 different TCP and UDP applications were obtained from *Trace1*, *Trace2* and *Trace3*, giving a total of almost 9 million packets and almost 30 thousand KISS signatures. Table 4.1 summarizes the testing data sets.

Source	Application	Transport protocol	Size (MB)	Packets (thousands)	Endpoints	Signatures
Trace1	dns	UDP	8	82	40724	1028
Trace1	openvpn	UDP	6	50	9	1239
Trace2	skype	UDP	148	729	15	18215
Trace3	iptv	UDP	14	200	142	4921
Trace1	bittorrent	TCP	114	1151	4866	134
Trace1	http	TCP	595	6153	138923	2586
Trace1	https	TCP	39	404	9910	214
		TOTAL	**924**	**8769**	**194589**	**28337**

Table 4.1 ***Summary of testing data sets****. TCP applications need more packets per signature.*

It is important to note that the signatures are not uniformly distributed across the endpoints, i.e. only some endpoints are capable of generating a signature, due to the C and P limits (recall 2.1.1.). However, the number of packets can provide a hint on the expected number of signatures.

4.2. Results

4.2.1. Test 1: performance vs. training set size

The goal of the first test was to observe the impact of training set size on system performance, similarly to [Fin09] sect. V-E.

First, 4 applications of DNS, Skype, IPTV and HTTP were chosen. For each of them, its trace file was divided into two halves (in terms of packets ordered by their timestamps) – A and B. The B part was further fed into the system for training, obtaining a signature database file of several lines.

For each application, a subset of its signature database was created, by extracting the first N signatures. This was repeated for N = 5, 10, 25, 50, 100, 250 and 500. Signature databases of all protocols with the same value of N were merged.

Finally, for increasing values of N, the system was trained using merged signature databases. In each step, all A parts were fed into the system for classification, in test mode (the `--testdb` option of *spid*, see 3.4.4).

Performance metrics for each value of N were collected. Results are presented in Table 4.2 and on Fig. 4.1. The last two columns of the table hold percentages of valid and invalid decisions.

Signatures	**HTTP**		**IPTV**		**Skype**		**DNS**		**Average**		**Valid**	**Invalid**
number	*%TP*	*%FP*	*%TP*	*%FP*	*%TP*	*%FP*	*%TP*	*%FP*	*%TP*	*%FP*	*decisions*	*decisions*
5	87	0	0	0	0	0	0	0	22	0	258	182
10	100	0	1	0	100	0	100	0	75	0	300	140
25	100	8	1	0	100	0	100	0	75	2	302	138
50	100	9	1	0	100	0	100	0	75	2	302	138
100	100	10	86	0	100	0	100	0	97	3	420	20
250	100	7	86	0	100	0	100	0	97	2	420	20
500	100	0	100	0	100	0	100	0	100	0	440	0

Table 4.2: ***Test 1 results.*** *Dependence of classification performance on the number of training signatures.*

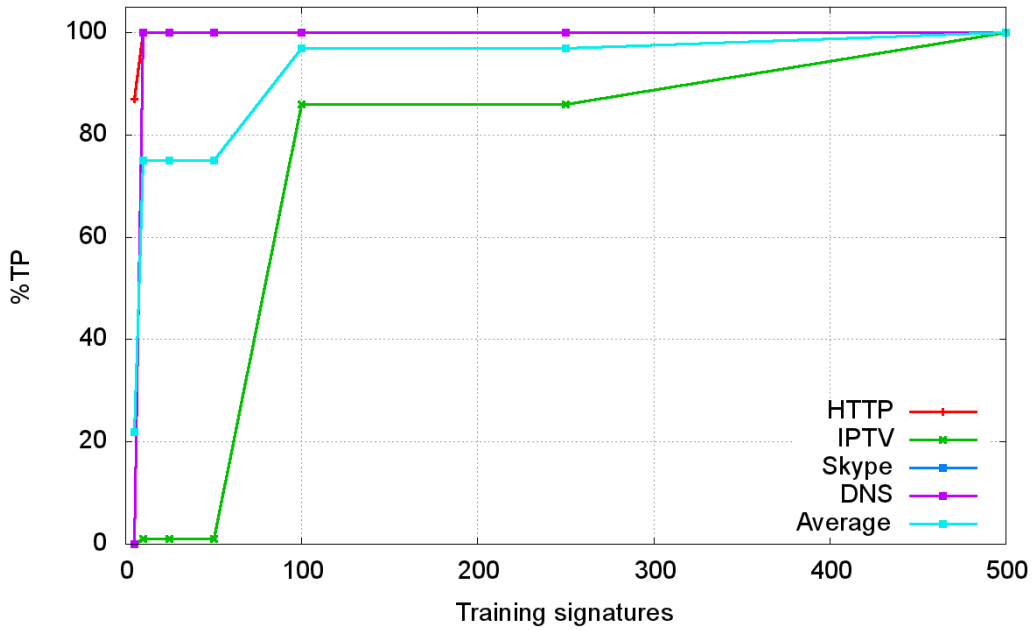

Fig. 4.1: ***True Positive Percentage as function of training signatures number.*** *Each protocol has its own minimal number of training signatures for good performance results.*

4.2.2. Test 2: overall system performance

In Test 2, the system was evaluated in an overall performance test, and compared with the standard KISS algorithm, i.e. without the modifications introduced in the thesis.

First, the whole testing data set was divided into two subsets: one for training and one for testing. Taking into account the results of Test 1, the division was made in such way, that at least 500 signatures for each application were present in the training subset. In case of BitTorrent and HTTPS this was not possible, hence these trace files were divided into two halves. Table 4.3 presents the obtained input traffic traces for Test 2.

	Training			Testing		
	Packets	*Signatures*	*Endpoints*	*Packets*	*Signatures*	*Endpoints*
BitTorrent	370000	62	2756	781088	74	2146
DNS	40000	500	19754	42407	529	20969
HTTP	2500000	502	26735	3653539	2086	112191
HTTPS	200000	111	4894	204626	98	5048
OpenVPN	20000	495	9	29747	742	2
Skype	20000	497	3	709275	17713	14

Table 4.3: ***Input data set for Test 2.*** *Note reduced training sets for BitTorrent and HTTPS.*

In order to evaluate the impact of the main algorithm modifications (see 2.1.3.), the test was repeated 3 times, with different *spid* options (see 3.4.4.):

1. **The thesis version:** all default options.
2. **The original KISS with EWMA reconciliation algorithm:** `--kiss-std`
3. **The original KISS algorithm**: `--kiss-std --verdict-simple --verdict-threshold=0`

Table 4.4 presents obtained results.

	KISS+		KISS with EWMA		KISS	
	%TP	*%FP*	*%TP*	*%FP*	*%TP*	*%FP*
OpenVPN	100.00	0.00	100.00	0.00	100.00	0.00
Skype	88.89	0.00	88.89	0.00	88.89	0.00
DNS	100.00	0.00	100.00	0.00	100.00	0.00
IP-TV	100.00	0.00	94.29	0.00	95.71	0.23
BitTorrent	100.00	0.00	100.00	0.40	50.00	0.90
HTTP	99.75	0.00	98.77	0.55	99.39	1.10
HTTPS	96.43	0.00	96.43	0.00	96.43	0.00
AVERAGE	97.87	0.00	96.91	0.14	90.06	0.32

Table 4.4: ***Test 2 results.*** *Algorithm modifications affect system performance.*

4.2.3. Test 3: unknown protocol detection

In Test 3, the previous test was repeated, but this time the training traces of HTTP and DNS were removed, i.e. the system was not trained to detect these protocols. In the testing traces, they were marked as “unknown” and the whole evaluation procedure was run once again. Results presented in Table 4.5.

	KISS+		KISS with EWMA		KISS	
	%TP	*%FP*	*%TP*	*%FP*	*%TP*	*%FP*
OpenVPN	100.00	0.00	100.00	0.00	100.00	0.00
Skype	88.89	0.00	88.89	0.00	88.89	0.00
IP-TV	100.00	9.04	94.29	91.14	95.71	94.99
BitTorrent	100.00	0.20	0.00	0.40	50.00	1.10
HTTPS	96.43	0.00	96.43	0.00	96.43	0.00
AVERAGE	97.06	1.85	75.92	18.31	86.21	19.22

Table 4.5: ***Test 3 results.*** *Presence of applications unknown to the system affects the results.*

4.2.4. Test 4: processing speed

The goal of Test 4 was to experimentally evaluate processing speed of the system, giving insight into the following questions:

1. What is the processing speed of the system, i.e. the time needed for classifying given number of IP packets?
2. What does the speed depend on?

All tests were run on a low-power laptop computer, with an Intel Pentium M 1.80GHz CPU and 1 GB of RAM, under Ubuntu 11.04 Linux distribution. An assumption is made, that the operating memory is big enough so that all system variables reside in RAM, i.e. there is no swapping.

First, the experiment from Test 1 was repeated, but for each value of N the *spid* program was run 10 times in a row. For each N, the total elapsed wall-clock time was measured using a UNIX tool of `time(1)`. Table 4.6 presents obtained results. Its third column "Average" holds the average time needed for single execution of *spid*.

Signatures *number*	**Total run time** *seconds*	**Average** *seconds*
5	44	4.40
10	44	4.40
25	45	4.50
50	44	4.40
100	44	4.40
250	45	4.50
500	44	4.40
Average		4.43

Table 4.6: ***Program run time versus number of training signatures.*** *First column shows number of training signatures for each of 4 applications.*

Then, another experiment was performed. The system was trained using the signature database from Test 1 for N=500. All packet traces introduced in section 4.1 were merged. From this merged file, several smaller files were generated, by extracting the first K = 100000, 200000, ..., 1000000 of its packets.

For increasing values of K, the *spid* program was run 10 times in a row. Again, the `time(1)` tool was used to measure the total elapsed wall clock time needed for classification.

Table 4.7 present obtained results. Fig. 4.2 graphically shows the dependence of time needed for single execution on K.

Packets *number*	**File size** *MB*	**Total run time** *seconds*	**Average** *seconds*	**Average pps** *pps x 1000*	**Average bps** *Mbps*
100000	10	10	1.00	100.00	80.00
200000	21	18	1.80	111.11	93.33
300000	51	29	2.90	103.45	140.69
400000	79	41	4.10	97.56	154.15
500000	104	50	5.00	100.00	166.40
600000	127	60	6.00	100.00	169.33
700000	143	72	7.20	97.22	158.89
800000	150	86	8.60	93.02	139.53
900000	157	95	9.50	94.74	132.21
1000000	167	101	10.10	99.01	132.28
			Average	99.61	136.68

Table 4.7: ***Program run time versus number of packets.***

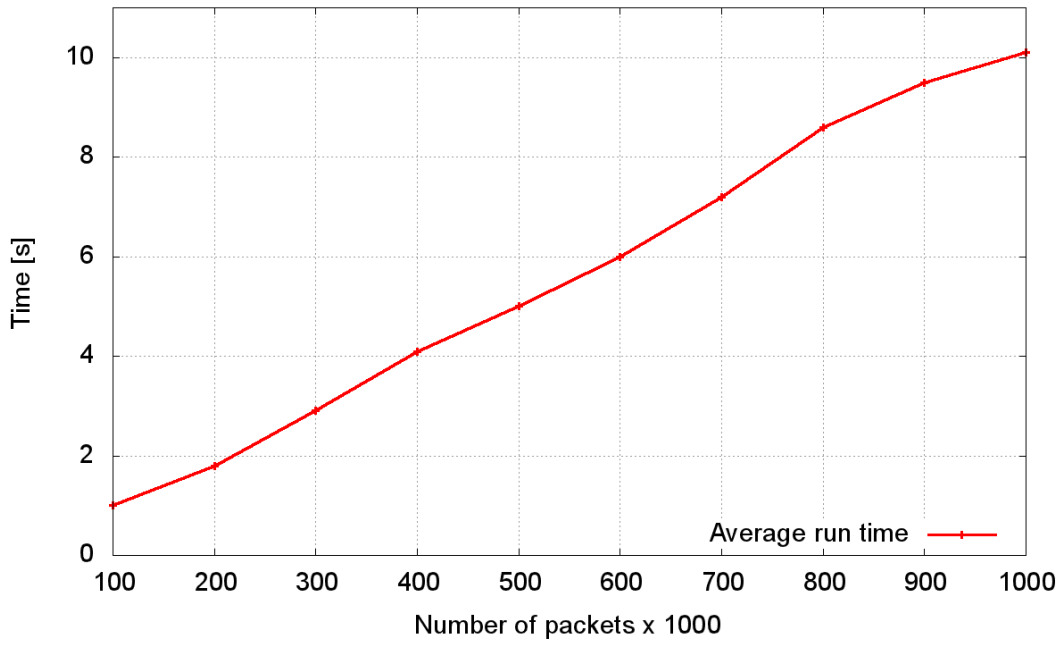

Fig. 4.2: ***Average run time versus number of IP packets.***

4.3. Discussion

4.3.1. Test 1

The system apparently needs a few hundred signatures in order to learn a particular application identity. The minimal number of signatures varies and depends on the protocol. For most applications, even 10 signatures were enough. However, results for 25 and more signatures revealed that the system needs at least 500 signatures for stable operation and consistent results, i.e. %TP close to 100 while keeping %FP close to 0.

Comparing to sect. V-E of [Fin09], the %FP metric is quite low, most probably due to modifications made to the KISS algorithm (2.1.3.). However, direct result comparison is impossible because the system classifies endpoints instead of flows (recall 2.1.3.1).

4.3.2. Test 2

The test of overall performance gave a very good result of average %TP=97.87 and %FP=0.00. The $\%TP_{Skype}=88.89$ could probably be improved by using greater number of training samples.

For experiments without the thesis modifications made to the main algorithm, relatively worse results were obtained. Indeed, signature extensions (recall 2.1.3.2) let for about 1% of improvement, whereas the EWMA classification reconciliation algorithm (see 2.1.3.3) combined with the unknown protocol detection (see 2.1.3.4) brought a 6% improvement to %TP.

4.3.3. Test 3

In Test 3, the system was able to properly detect unknown protocols, with only a 0.81% worse %TP metric and a 1.85% increase in %FP.

Again, compared to the original KISS algorithm, the thesis system produced much better results. In case of experiment without signature extensions, results of %TP=75.92 and %FP=18.31 were obtained.

However, such comparison is unfair, as the original algorithm relies on relatively good-quality “Background” traffic for the training phase.

4.3.4. Test 4

On a low powered machine the system was able to handle 136.68Mbps and almost 100 000 packets per second, on average. This roughly means that one packet is handled in 10μs.

Processing time seems to linearly depend on the number of packets for classification. It does not depend on the number of training signatures, which is a great advantage of the *libsvm* library and SVM in general.

5. CONCLUSIONS

1. In the thesis a practical system for statistical classification of IP traffic was implemented. The system fulfils the thesis goals set in section 1.2. In particular, it works in real-time, under control of the Linux operating system.
2. Resultant software is portable, fast and embeddable. It is implemented in C language as a single-threaded software library. It has a modular architecture based on an event loop.
3. The system can simultaneously: handle off-line and live traffic sources, classify new packets, learn new applications, and evaluate system performance.
4. Application of the modified KISS algorithm allowed to achieve classification performance results of %TP=97.87 and %FP=0, on average. This is consistent with the results claimed by the algorithm authors (see 2.1).
5. Modifications of the KISS algorithm proposed in the thesis resulted in better classification performance.
6. Application of SVM class membership probability resulted in an alternative method for detection of unknown protocols. This is an improvement compared to the original KISS algorithm, in which a special “Background” traffic class is required in such case.
7. SVM model generation process was optimized using a queue of training vectors.
8. A several hundred training signatures of a particular application are required in order to achieve good results. For UDP, 40-80 packets are required on average to form a signature. For TCP, 2000-9000 packets.
9. The system was able to properly recognize applications which use encryption, particularly Skype, OpenVPN, HTTPS, and a proprietary IP-TV application. No protocol reverse-engineering was required in order to train the system to do so.
10. On a low-powered machine, the system achieved average processing speed of almost 100 000 packets per second and over 130Mbps.

6. SUMMARY

The thesis presented a practical system for statistical classification of IP traffic. Two novel algorithms were applied, extended and implemented. The system was written in C language, in a portable and flexible manner as a software library. Evaluation of the resultant software performance yielded very good results, in terms of quality and processing speed, achieving %TP>97 and %FP=0 on average.

Results introduced by the thesis answer positively to the question on applicability of statistical traffic classification in practice. The system works in real-time, under the Linux operating system, which is very popular amongst small and mid-sized Internet Service Providers. It can be trained to recognize new kinds of traffic in a timely manner.

However, further work is needed in order to overcome some limitations. In particular, the algorithm implemented in the thesis is able to classify only about 5% of Internet endpoints, yet carrying more than 98% of bytes. Probably, a solution covering the whole problem area would need to combine several methods.

Moreover, another problem that the research community has to solve is a common set of traffic samples. They are crucial for evaluation of classification performance. Currently, a usual situation is that each research group introduces its own input data set. Thus, comparison of their results – and proposed classification methods – is often difficult.

Traffic classification has numerous applications. It can be used as the fundamental element of traffic shaping systems, firewalls, intrusion detection systems, network maintenance tools, and much more. For instance, the thesis software could be used in a highly congested network in order to prioritize Skype traffic and improve voice quality.

7. APPENDIX: IMPLEMENTATION DETAILS

7.1. *libspi* data structures

7.1.1. Main structure: `struct spi`

Structure members:

- `mm`: used for memory management by the *libasn* library
- `options`: stores run-time program options
- `running`: holds `true` if the program is currently inside an iteration of the main loop, `false` otherwise
- `quitting`: used for breaking the main loop; if the value is `true`, then *libspi* will exit on its next iteration
- `eb`: a *libevent* variable holding its main instance data
- `evgc`: a *libevent* variable holding the event which schedules the *libspi* garbage collector execution
- `subscribers`: subscribers of internal events; a hash table of `struct spi_subscribers` (see 3.3.2.2)
- `sources`: traffic sources (see 2.2.3); a linked list of `struct spi_source` (see 3.3.2.3)
- `eps`: the endpoint table (see 2.2.4); a hash table of `struct spi_ep` (see 3.3.2.4)
- `flows`: TCP flow table used for the P limit (see 2.1.1); a hash table of `struct spi_flow`; similar to `spi.eps`
- `traindata`: training signatures (see 2.2.2); a hash table of `struct spi_signature` (see 3.3.2.5)

- `trainqueue`: signature database - queue of training signatures (see 2.2.1); a hash table of `struct spi_signatures`; it constitutes an intermediate buffer between signatures in the database and `spi.trainqueue`
- `stats`: statistical data used for calculation of system performance metrics; see 3.3.2.7
- `cdata`: a generic pointer to main algorithm data, in this case `struct kissp`; see 3.3.2.8
- `vdata`: a generic pointer to implementation of the complex decision process, in this case `struct verdict`; see 3.3.2.9

7.1.2. Internal events: `struct spi_subscribers`, `spi_event_cb_t` and `struct spi_event`

Members of `struct spi_subscribers`:

- `hl`: list of handlers to call when the event is announced; a linked list of `spi_event_cb_t`
- `ahl`: a list of handlers to call after all handlers from `hl` finish; a linked list of `spi_event_cb_t`
- `aggstatus`: an enumeration for tracking the current state of the event in the system; used for aggregation of multiple event announcements into one round of handler call; possible values:
  - `SPI_AGG_DISABLED`: state tracking disabled
  - `SPI_AGG_READY`: no event announcements in the system
  - `SPI_AGG_PENDING`: the event was announced and it is pending for being handled

Arguments of `spi_event_cb_t`:

- `spi`: reference on the global instance of `struct spi`
- `evname`: name of the event that caused the handler to be called
- `arg`: optional generic pointer passed during event announcement

A handler returns `false` if it requests to be unsubscribed from the list of event handlers. Otherwise, it returns `true`.

Members of `struct spi_event`:

- `spi`: reference on the global instance of `struct spi`
- `evname`: name of the event that was announced
- `ss`: reference on the relevant entry in `spi.subscribers`
- `arg`: optional announcement argument; a generic pointer passed to `spi_event_cb_t`
- `argfree`: decides if the memory pointed by `arg` should be released after event handling is finished; value of `true` enables this functionality

7.1.3. IP traffic: `struct spi_source` and `struct spi_pkt`

Members of `struct spi_source`:

- `spi`: reference on the global instance of `struct spi`
- `type`: type of the source, either:
  - `SPI_SOURCE_FILE`: offline source – a packet trace file
  - `SPI_SOURCE_SNIFF`: online source – a network interface traffic sniffer
- `label`: optional application label; if the value is greater than 0, the source can be used for system training or performance evaluation (recall 2.2.3)
- `testing`: if `true`, this source is in testing mode and will be used for performance evaluation instead of training
- `fd`: a UNIX file descriptor of the underlying packet source, extracted from *libpcap*; the file descriptor is monitored using *libevent* for new data available for reading, thus this is the most important, original source of activity in the system
- `evread`: a *libevent* variable representing the event of new data available for reading from `fd`
- `counter`: number of IP packets read from the source so far
- `signatures`: number of signatures extracted from the source so far
- `learned`: number of source signatures used for training so far
- `eps`: number of endpoints identified in the source so far
- `closed`: set to `true` when source is considered to be closed, e.g. when the end of packet trace file is encountered

- `as`: a union of two structures, used in interchangeable way, depending on the value of the `type` member
- `as.file`: structure used if `type` is `SPI_SOURCE_FILE`
  - `pcap`: an instance variable of the *libpcap* library
  - `path`: file system path to packet trace file
  - `time`: time-stamp of the most recently read IP packet; used as virtual current moment of time (i.e. "now"), local to the particular traffic source
  - `gctime`: value of the `time` member during last execution of the endpoint table garbage collector; used in order to schedule the garbage collector using the virtual time (e.g. once per minute), even if in the reality the file is parsed much faster (e.g. a virtual week of traffic each second)
- `as.sniff`: structure used if `type` is `SPI_SOURCE_SNIFF`
  - `pcap`: an instance variable of the *libpcap* library
  - `ifname`: name of the network interface to capture the traffic on

Members of `struct spi_pkt`:

- `payload`: stores first N bytes of the packet payload (recall 2.1.1)
- `ts`: packet time-stamp; the value has a meaning relative to the traffic source (see `spi_source.as.file.time`)
- `size`: total packet size, in bytes

7.1.4. Endpoints: `struct spi_ep`

Members of `struct spi_ep`:

- `mm`: a memory management object of *libasn* representing the whole memory occupied by the endpoint data, including the parent instance of `mm`
- `source`: traffic source that caused creation of this endpoint (see 3.3.2.3)
- `epa`: endpoint address
- `last`: time-stamp of the last packet registered in this endpoint
- `pkts`: accumulated packets; a linked list of `struct spi_pkt`
- `gclock`: if `true`, endpoint must not be removed by the garbage collector
- `predictions`: number of SVM predictions so far

- `verdict`: stores current classification verdict (see 3.3.2.5 for `spi_label_t`)
- `verdict_prob`: stores probability of `verdict`; allowed range 0.0 – 1.0
- `verdict_count`: number of changes of `verdict` values
- `vdata`: a generic pointer to endpoint-specific data of the complex decision process (see 3.3.2.9)

7.1.5. Signatures: `struct spi_signature`

Members of `struct spi_signature`:

- `label`: optional application identity – if not 0, the signature can be utilized for system training or testing
- `c`: feature vector; an array of `struct spi_coordinate`, compatible with `struct svm_node` of *libsvm* (in terms of members and their placement)
  - `index`: coordinate number (first coordinate has `index` of value 1, not 0)
  - `value`: coordinate value

7.1.6. Classification results: `struct spi_classresult`

Members of `struct spi_classresult`:

- `ep`: endpoint that generated the input feature vector
- `result`: SVM classification result; the return value from the `svm_predict_probability()` function of *libsvm*
- `cprob_lib`: SVM classification probability array; the third argument to the `svm_predict_probability()` function
- `cprob`: SVM classification probability after translation; *libsvm* can mix the array indices, so additional translation from the concept of *libsvm* class to *libspi* label is required

7.1.7. Performance evaluation: `struct spi_stats`

Members of `struct spi_stats`:

- `learned_pkt`: number of signatures that were used for system training, coming from traffic sources

- `learned_tq`: number of signatures that were used for system training, coming from the signature database
- `test_all`: number of classifications, for which testing information is available
- `test_is`: like `test_all`, but for each label separately
- `test_ok`: number of True classifications
- `test_FN`: number of False Negatives for each label
- `test_FP`: number of False Positives for each label

7.1.8. KISS algorithm: `struct kissp`

Members of `struct kissp`:

- `feature_num`: number of coordinates in feature vectors – either 24 ($2 \times N$, see 2.1.1) or 28 (extended signature, see 2.1.3.2)
- `options.pktstats`: if `true`, enable extended signatures
- `svm`: gathers data required for handling of the *libsvm* library
  - `model`: the SVM model; output of the `svm_train()` function
  - `params`: *libsvm* parameters
  - `labels`: translation from *libsvm* class number to *libspi* label; an array, in which cell *i* represents class number *i*, and its value is the label
  - `nr_class`: number of SVM classes, output of the `svm_get_nr_class()` function

7.1.9. Complex decision process: `struct verdict` and `struct ewma_verdict`

Members of `struct ewma_verdict`:

- `cprob`: endpoint classification probabilities; each array cell holds a separately calculated EWMA

Members of `struct verdict`:

- `type`: chosen method for the final decision
  - `SPI_VERDICT_SIMPLE`: use method “Simple”
  - `SPI_VERDICT_EWMA`: use method “EWMA” (default)

- `SPI_VERDICT_BEST`: use method "Best"

- `ewma.N`: number of samples for EWMA, default 5

7.2. *libspi* Application Programming Interface

- `spi_init()`: initialize *libspi*, see 3.3.3.1; this function must be called before other functions can be used
  - returns an initialized instance of `struct spi`, which must be passed as the first argument to all other API functions
  - `so` arguments: run-time system options; if NULL, default values will be used
- `spi_free()`: does the opposite to `spi_init()`, deallocating all memory occupied by the system
- `spi_add()`: add a traffic source, see 3.3.3.1
  - returns 0 on success or a different value in case of an error
  - arguments:
    - `type`: source type
    - `label`: source label, may be 0 if unknown
    - `test`: if `true` and `label` is not 0, source will be used for system testing
    - `args`: additional arguments, specific to source type:
      - for traffic files, string of format "`<path> <filter>`", where:
        - `<path>`: path to the traffic file
        - `<filter>`: optional *libpcap* filter in BPF format [BPF]; if not specified, "`tcp or udp`" is used
      - for network interfaces, string of format "`<name> <filter>`", where:
        - `<name>`: name of the network interface to capture the traffic on
        - `<filter>`: same as for traffic sources
- `spi_loop()`: make one iteration of the main program loop
  - returns 0 on success, -1 on temporary error, 1 on permanent error or 2 if stopping the whole system was requested
- `spi_stop()`: make a request to stop the main loop
- `spi_announce()`: announce an internal event

  - arguments:
    - `evname`: event name
    - `delay_ms`: event delay, in milliseconds; if no delay is desired, set to 0
    - `arg`: optional event argument
    - `argfree`: if `true`, then the memory pointed by `arg` will be deallocated after event handling
- `spi_subscribe()`: subscribe to an internal event
  - arguments:
    - `evname`: event name
    - `cb`: event callback
    - `aggregate`: if `true`, aggregate multiple announcements of the same event into one, until the first announcement is handled
- `spi_train()`: add a training signature (see 3.3.3.3)
  - `sign` argument: the signature to add; it must have the `label` member set
- `spi_trainqueue()`: add a signature to the queue of training signatures (signature database, see 2.2.1.); does not announce the `traindataUpdated` event
  - takes same arguments as in `spi_train()`
- `spi_trainqueue_commit()`: move all signatures from the training queue to training signatures, announcing the `traindataUpdated` event
- `spi_stats_fp()`: get the %FP metric for given application identity
  - returns either a real number in range 0.0-100.0 or -1 if the metric is unavailable
  - `label` argument: application identity label
- `spi_stats_fn()`: like `spi_stats_fp()`, but returns the %FN metric

7.3. *spid* data structures

Members of `struct source`:

- `proto`: optional name of application identity; if NULL, the traffic source supplies packets of unknown identity
- `cmd`: traffic source specification, either:
  - a file path – the source will be used as `SPI_SOURCE_FILE`

- a network interface name – the source will be used as `SPI_SOURCE_SNIFF`

- `test:` if `true` and `proto` is not NULL, the source will be used for system testing

Members of `struct spid`:

- `mm:` a memory management variable used by *libasn*
- `spi:` *libspi* instance data
- `spi_opts:` *libspi* options to pass during its initialization
- `proto2label:` used for translation from application identity name (e.g. “Skype”) into a numeric label (e.g. 7); a hashing table
- `label2proto:` as `proto2label`, but in the opposite direction
- `learn:` list of traffic sources for system training; a linked list of `struct source`
- `detect:` like `learn`, but holds traffic sources for classification and system testing
- `options:` run-time options of `spid`

7.4. spid data formats

7.4.1. Command-line source specification format

The command-line format for traffic files is presented in a few examples:

```
1.    ./file.pcap
2.    smtp:/home/user/file.pcap tcp and port 25
```

In the first line, a file path relative to the current working directory of the *spid* process is given. In the second line, an absolute file path is given, with a BPF [BPF] filter attached, after a space character. The “smtp:” prefix tells the application identity behind IP packets, so the source can be used for system training or testing.

For network interfaces, it is:

```
1.    wlan0
2.    dns:eth0 udp and port 53
```

Both lines give the interface name. Optionally, line can be prefixed with application name and a BPF filter can be attached, as in case of traffic files.

7.4.2. Packet trace index file format

An index file is used for referencing many traffic files at once. It follows a syntax similar to the command-line syntax, but the application name prefix is necessary. A few exemplary lines given on Listing 7.1.

```
1.    # Exemplary packet trace index file.
2.    bittorrent /home/user/dumps/bittorrent1.pcap
3.    bittorrent /home/user/dumps/bittorrent2.pcap
4.    bittorrent-tcp /home/user/dumps/bittorrent-tcp1.pcap
5.    dns /home/user/dumps/dns1.pcap
6.    dns /home/user/dumps/dns3.pcap
7.    http /home/user/dumps/http1.pcap
8.    http /home/user/dumps/http2.pcap
9.    skype /home/user/dumps/skype1.pcap
```

*Listing 7.1: **Exemplary packet trace index file**. Comments start with a hash character.*

7.4.3. Signature database file format

Signatures (recall 3.3.2.5) are stored in text format, one per line. Each entry begins with application name, and then coordinate values, separated with spaces, follow. Exemplary file given on Listing 7.2.

```
1.    bittorrent 0.875 0.875 0.875 0.875 0.871667 0.897333 0.407333
0.407 0.896667 0.872333 0.871667 0.896667 0.872 0.871667 0.872 0.871667
0.875 0.875 0.875 0.875 1 0.875 0.875 0.875 0.0986 0.0237397 0.0285972 2
2.    dns 0.0386667 0.0293333 0.0306667 0.02 1 0.466667 1 0.466667 1 1 1
1 1 1 0.649333 1 1 1 0.692 1 1 1 1 1 0.0561667 0.0297895 0.04048 2
3.    dns 0.0266667 0.0133333 0.02 0.0186667 1 0.466667 1 0.466667 1 1 1
1 1 1 0.509333 1 1 1 0.658667 1 1 1 1 1 0.0615333 0.0234933 0.0269054 2
4.    skype 0.0116667 0.012 0.0163333 0.0136667 1 0.0773333 0.01
0.0143333 0.00933333 0.016 0.00633333 0.00833333 0.0196667 0.0173333
0.01 0.0106667 0.00966667 0.00866667 0.0156667 0.01 0.011 0.012
0.0166667 0.006 0.141983 0.0267949 0.0105065 2
5.    skype 0.00866667 0.0123333 0.0146667 0.013 1 0.0753333 0.014
0.00933333 0.0106667 0.0273333 0.00633333 0.007 0.00666667 0.00966667
0.00833333 0.00866667 0.013 0.0123333 0.00833333 0.0143333 0.00633333
0.0223333 0.0143333 0.011 0.150242 0.0148 0.0240926 2
6.    openvpn 0.469667 0.922333 0.455667 0.455667 0.455667 0.455667
0.455667 0.973667 0.455667 0.455667 0.469667 0.455667 0.455667 0.455667
0.455667 0.973667 0.455667 0.455667 0.453667 0.973667 0.973667 0.973667
0.973667 0.973667 0.0744583 0.0247179 0.0449351 2
```

*Listing 7.2: **Exemplary signature database file**. Each line starts with application name.*

7.4.4. Endpoint classification output format

New endpoint verdict generates a new line on the standard program output. Example given on Listing 7.3.

```
1.    $ ./spid --signdb=test/signdb test/1.pcap
2.    1.pcap: UDP 192.168.7.124:19313 is skype
3.    1.pcap: UDP 149.13.32.247:46822 is skype
4.    1.pcap: UDP 192.168.7.124:50084 is openvpn
5.    1.pcap: UDP 91.200.172.23:1198 is openvpn
6.    1.pcap: TCP 91.197.13.248:80 is http
7.    1.pcap: TCP 212.91.8.233:80 is http
8.    1.pcap: TCP 77.79.214.25:80 is http
9.    1.pcap: TCP 91.197.13.247:80 is http
```

*Listing 7.3: **Exemplary spid output**. A traffic file ./test/1.pcap is classified.*

Each line starts with source name, which is either a shortened file path or an interface name. This is followed by endpoint address – transport protocol, IP address and transport protocol port. Finally, name of the recognized application identity is given.

In case the `--print-probs` command-line option is enabled, program output looks like on Listing 7.4.

```
1.    $ ./spid --signdb=test/signdb test/1.pcap --print-probs
2.    1.pcap: UDP 192.168.7.124:19313 is skype 90 1
3.    1.pcap: UDP 149.13.32.247:46822 is skype 100 1
4.    1.pcap: UDP 192.168.7.124:50084 is openvpn 98 1
5.    1.pcap: UDP 91.200.172.23:1198 is openvpn 100 1
6.    1.pcap: TCP 91.197.13.248:80 is http 99 1
7.    1.pcap: TCP 212.91.8.233:80 is http 98 1
8.    1.pcap: TCP 77.79.214.25:80 is http 98 1
9.    1.pcap: TCP 91.197.13.247:80 is http 99 1
```

*Listing 7.4: **Program output with probability information**. Format enabled by the `–print-probs` option.*

Comparing to the output from Listing 7.3, the last two numbers are respectively the classification probability and the number of verdict changes made so far for this endpoint.

7.4.5. Performance metrics output format

In case the `--stats` command-line option is enabled and adequate testing sources were provided, the output as on Listing 7.5 will be generated at the end of program execution.

```
1.    PROTOCOL STATISTICS:
2.                 dns TP 100% / FP  0% in 9 endpoints
3.          bittorrent TP 100% / FP  1% in 10 endpoints
4.               skype TP 100% / FP  0% in 2 endpoints
5.             openvpn TP 100% / FP  0% in 4 endpoints
6.             sopcast TP 100% / FP  0% in 80 endpoints
7.                http TP 100% / FP  0% in 20 endpoints
8.      bittorrent-tcp TP 50% / FP  0% in 2 endpoints
9.             AVERAGE TP 93% / FP  0% in total of 127 endpoints
10.   ENDPOINT STATISTICS:
11.              valid  126 (99%)
12.            invalid    1 ( 1%)
13.              TOTAL  127
```

*Listing 7.5: **Program performance metrics output**. Functionality enabled by the `--stats` option.*

In the part entitled "PROTOCOL STATISTICS", the %TP and %FP metrics defined in section 2.3. are given – for each application identity, and as an average. The last part – "ENDPOINT STATISTICS" – tells how many valid and invalid classifications the system made.

8. LITERATURE

[RFC791]: J. Postel, Internet Protocol, 1981

[RFC768]: J. Postel, User Datagram Protocol, 1980

[RFC793]: J. Postel, Transmission Control Protocol, 1981

[Fin09]: A. Finamore, M. Mellia, M. Meo, D. Rossi, KISS: Stochastic Packet Inspection, 2009

[Fin10]: G. Mantia, D. Rossi, A. Finamore, M. Mellia, M. Meo, Stochastic Packet Inspection for TCP Traffic, 2010

[RFC2474]: K. Nichols, S. Blake, F. Baker and D. Black, Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers, 1998

[IUS]: World Internet Users and Population Statistics. Internet World Stats, Retrieved August 11, 2011, from <http://www.internetworldstats.com/stats.htm>

[Kar04]: Thomas Karagiannis, Andre Broido, Nevil Brownlee, Kimberly C. Claffy, Michalis Faloutsos, Is P2P dying or just hiding?, 2004

[Ellacoya]: Arbor Networks. Arbor eSeries: Deep Packet Inspection (DPI), Retrieved August 2011, from <http://www.arbornetworks.com/>

[L7]: Application Layer Packet Classifier for Linux, Retrieved July 2011, from <http://l7-filter.clearfoundation.com/>

[ODPI]: OpenDPI - The open source deep packet inspection engine, Retrieved August, 2011, from <http://code.google.com/p/opendpi/>

[Kar05]: Karagiannis et al., BLINC: multilevel traffic classification in the dark, 2004

[Cla94]: Kimberly Claire Claffy, Internet traffic characterization, 1994

[Rou04]: M. Roughan et. al., Class-of-service mapping for QoS: a statistical signature-based approach to IP traffic classification, 2004

[Moo05]: AW Moore et al., Internet traffic classification using bayesian analysis techniques, 2005

[Zan05]: S. Zander et al., Automated traffic classification and application identification using machine learning, 2005

[Byu11]: Byungchul Park, James Won-Ki Hong, Young J. Won, Toward Fine-Grained Traffic Classification, 2011

[Kim08]: H. Kim, et al., Internet Traffic Classification Demystified: Myths, Caveats, and the Best Practices, 2008

[Sal07]: L. Salgarelli, et al., Comparing traffic classifiers, 2007

[Cor95]: C. Cortes, V. Vapnik, Support-vector networks, 1995

[PCAP]: TCPDUMP/LIBPCAP: public repository, Retrieved June, 2011, from <http://www.tcpdump.org/>

[Pro00]: libevent, Retrieved August, 2011, from <http://monkey.org/~provos/libevent/>

[For05]: libasn, Retrieved August, 2011, from <http://labs.asn.pl/asnlibs/wiki/libasn>

[CC01a]: Chang, Chih-Chung and Lin, Chih-Jen, LIBSVM: A library for support vector machines, 2011

[Tstat]: Tstat project, Retrieved August, 2011, from <http://tstat.tlc.polito.it/traces-skype.shtml>

[RECIPE]: RECIPE, Retrieved August, 2011, from <http://recipe.dis.unina.it/>

[MIMOSA]: MIMOSA, Retrieved August, 2011, from <http://risorse.dei.polimi.it/mimosa/index_en.html>

[TstatSkype]: Tstat Skype Testbed Traces, Retrieved August, 2011, from <http://tstat.tlc.polito.it/traces-skype.shtml>

[TstatIPTV]: Tstat Multicast IP-TV Traces, Retrieved August, 2011, from <http://tstat.tlc.polito.it/traces-IPTV.shtml>

[BPF]: pcap-filter(7) manual page, Retrieved August, 2011, from <http://www.manpagez.com/man/7/pcap-filter/>

9. SUMMARY IN POLISH

STATYSTYCZNA KLASYFIKACJA RUCHU IP W CZASIE RZECZYWISTYM W SYSTEMIE OPERACYJNYM LINUX

Streszczenie: Praca prezentuje system statystycznej klasyfikacji ruchu IP działający w praktyce. Zostały zastosowane i poszerzone dwa nowatorskie algorytmy oparte o klasyfikację wektorów cech przy użyciu SVM. System zaimplementowano w języku C w formie biblioteki, która umożliwia zarówno monitorowanie interfejsów sieciowych w czasie rzeczywistym, jak i pracę w trybie off-line, przez odczyt plików śladu ruchu. Możliwe jest równoczesne klasyfikowanie, uczenie systemu i ocena jego wydajności. Otrzymano bardzo dobre wyniki jakościowe i szybkości przetwarzania pakietów, osiągając średnio %TP>97 i %FP=0.

STATISTICAL, REAL-TIME CLASSIFICATION OF IP TRAFFIC IN LINUX OPERATING SYSTEM

Summary: The thesis introduces a practical system for statistical classification of IP traffic. Two novel algorithms are applied and extended. They are based on feature vector classification using SVM. A software library written in C language is presented. Resultant system can monitor network interfaces in real-time and read off-line packet trace files. Simultaneous classification, system training, and performance evaluation is possible. The system yields very good results, in terms of quality and packet processing speed, achieving %TP>97 and %FP=0 on average.